Symbolic Calculus

Release 10.3

The Sage Development Team

Mar 20, 2024
# CONTENTS

1 Using calculus 3

2 Internal functionality supporting calculus 5
   2.1 Symbolic Expressions ......................................................... 5
   2.2 Callable Symbolic Expressions .............................................. 143
   2.3 Assumptions ................................................................. 146
   2.4 Symbolic Equations and Inequalities ...................................... 155
   2.5 Symbolic Computation ....................................................... 172
   2.6 Units of measurement ....................................................... 198
   2.7 The symbolic ring .......................................................... 204
   2.8 Subrings of the Symbolic Ring ............................................. 211
   2.9 Classes for symbolic functions ........................................... 216
   2.10 Factory for symbolic functions .......................................... 220
   2.11 Functional notation support for common calculus methods ......... 224
   2.12 Symbolic Integration ...................................................... 235
   2.13 TESTS:: ................................................................. 245
   2.14 A Sample Session using SymPy ........................................... 246
   2.15 Calculus Tests and Examples ............................................. 250
   2.16 Conversion of symbolic expressions to other types ................... 253
   2.17 Complexity Measures .................................................... 275
   2.18 Further examples from Wester’s paper .................................. 276
   2.19 Solving ordinary differential equations .................................. 286
   2.20 Discrete Wavelet Transform ............................................... 304
   2.21 Discrete Fourier Transforms .............................................. 307
   2.22 Fast Fourier Transforms Using GSL ..................................... 315
   2.23 Solving ODE numerically by GSL ......................................... 319
   2.24 Numerical Integration ..................................................... 324
   2.25 Riemann Mapping .......................................................... 328
   2.26 Real Interpolation using GSL ............................................. 339
   2.27 Complex Interpolation .................................................... 342
   2.28 Calculus functions ......................................................... 345
   2.29 Symbolic variables ........................................................ 346
   2.30 Access to Maxima methods ................................................ 352
   2.31 Operators ................................................................. 353
   2.32 Benchmarks ............................................................... 355
   2.33 Randomized tests of GiNaC / PyNaC .................................... 357

3 Indices and Tables 363

Python Module Index 365
Calculus is done using symbolic expressions which consist of symbols and numeric objects linked by operators (functions).

**Note:** While polynomial manipulation can be done with expressions, it is more efficient to use polynomial ring elements.
CHAPTER ONE

USING CALCULUS

• Symbolic Computation
• Examples
  – Calculus examples
  – Calculus Tests and Examples
  – Further examples from Wester’s paper
• More about symbolic variables and functions
• Main operations on symbolic expressions
• Assumptions about symbols and functions
• Symbolic Equations and Inequalities
• Symbolic Integration
• Solving ordinary differential equations
• Solving ODE numerically by GSL
• Numerical Integration
• Real Interpolation using GSL
• Transforms
  – Discrete Wavelet Transform
  – Discrete Fourier Transforms
  – Fast Fourier Transforms Using GSL
• Vector Calculus
• Riemann Mapping
• Other calculus functionality
• Complexity Measures
• Units of measurement
INTERNAL FUNCTIONALITY SUPPORTING CALCULUS

- The symbolic ring
- Subrings of the Symbolic Ring
- Operators
- Classes for symbolic functions
- Functional notation support for common calculus methods
- Factory for symbolic functions
- Internals of Callable Symbolic Expressions
- Conversion of symbolic expressions to other types
- Benchmarks
- Randomized tests of GiNaC / PyNaC
- Access to Maxima methods
- External integrators
- External interpolators

2.1 Symbolic Expressions

RELATIONAL EXPRESSIONS:
We create a relational expression:

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

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
```

This can transform a true relation into a false one:
We can do arithmetic with relations:

\begin{verbatim}
 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)
\end{verbatim}

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

\begin{verbatim}
 sage: (x^3 + x <= x - 17) + (-x <= x - 10)
x^3 <= 2*x - 27
\end{verbatim}

Here they are not:

\begin{verbatim}
 sage: (x^3 + x <= x - 17) + (-x >= x - 10)
 Traceback (most recent call last):
   ...
 TypeError: incompatible relations
\end{verbatim}

**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:

\begin{verbatim}
 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*b*v + 2*c*v + 2*u*v + v^2
\end{verbatim}

\texttt{class} sage.symbolic.expression.E
Bases: *Expression*

Dummy class to represent base of the natural logarithm.

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

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

**EXAMPLES:**

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

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

Arithmetic works:

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

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

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

(continues on next page)
sage: loads(dumps(e))
e
sage: float(e)
2.718281828459045...

sage: e.__float__()
2.718281828459045...

sage: e._mpfr_(RealField(100))
2.7182818284590452353602874714

sage: e._real_double_(RDF) # abs tol 5e-16
2.718281828459045

sage: import sympy

˓→ needs sympy

˓→ needs sympy

sage: sympy.E == e # indirect doctest
True

class sage.symbolic.expression.Expression

Bases: Expression

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

Order (hold=False)

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

OUTPUT:

A symbolic expression.

EXAMPLES:

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

WZ_certificate (n, k)

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

To prove the identity \( \sum_k F(n, k) = \text{const} \) it suffices to show that

\[
F(n + 1, k) - F(n, k) = G(n, k + 1) - G(n, k)
\]

with \( G = RF \) and \( R \) the WZ certificate.

EXAMPLES:

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

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

(continues on next page)
Symbolic Calculus, Release 10.3

(n, k) --> 1/2*k*binomial(n, k)/(2^n*(k - n - 1))
sage: (F(n+1,k) - F(n,k) - G(n,k+1) + G(n,k)).simplify_full()
0

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

add (hold=False, *args)
Return the sum of the current expression and the given arguments.
To prevent automatic evaluation use the hold argument.

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

To then evaluate again, we use unhold():
sage: a = x.add(x, hold=True); a.unhold()
2*x

add_to_both_sides (x)
Return a relation obtained by adding x to both sides of this relation.

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

(continues on next page)
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()
1.59279480777613
sage: plot(lambda x: SR(x).arccos(), -1,1)  # needs sage.plot
```

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)
arccosh(1)
sage: arccos(1)
0
```

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

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

arccosh (hold=False)

Return the inverse hyperbolic cosine of self.

EXAMPLES:

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

To prevent automatic evaluation use the `hold` argument:
Symbolic Calculus, Release 10.3

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

```

**arcsin** *(hold=False)*

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

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

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

```

**arcsinh** *(hold=False)*

Return the inverse hyperbolic sine of self.

**EXAMPLES:**

`sage: x.arcsinh()
arcsinh(x)`

`sage: SR(0).arcsinh()
0`

`sage: SR(1).arcsinh()
arcsinh(1)`

`sage: SR(1.0).arcsinh()
0.881373587019543`

`sage: maxima('asinh(2.0)')
1.4436354751788...`

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 use `unhold()`:

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

`arctan(hold=False)`

Return the arc tangent of self.

**EXAMPLES:**

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

`sage: x.arctan()
arctan(x)`

`sage: SR(1).arctan()
1/4*pi`

`sage: SR(1/2).arctan()
arctan(1/2)`

`sage: SR(0.5).arctan()
0.463647609000806`

`sage: plot(lambda x: SR(x).arctan(), -20,20)  # needs sage.plot
Graphics object consisting of 1 graphics primitive`

To prevent automatic evaluation use the `hold` argument:

`sage: SR(1).arctan(hold=True)
arctan(1)`
This also works using functional notation:

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

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

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

`arctan2 (x, hold=False)`
Return the inverse of the 2-variable tan function on self and x.

EXAMPLES:

```python
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:

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

This also works using functional notation:

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

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

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

`arctanh (hold=False)`
Return the inverse hyperbolic tangent of self.

EXAMPLES:

```python
sage: x.arctanh()
arctanh(x)
sage: SR(0).arctanh()
0
sage: SR(1/2).arctanh()
1/2*log(3)
sage: SR(0.5).arctanh()
```

(continues on next page)
To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:

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

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

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

### `args()`

**EXAMPLES:**

```python
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:**

```python
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 $x > 2$:

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

Bool returns True below if the inequality is definitely known to be True.
Symbolic Calculus, Release 10.3

**bool**

- `bool(x > 0)`
  - True
- `bool(x < 0)`
  - False

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

- `bool(x > 3)`
  - False

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

```python
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()
```

### binomial(k, hold=False)

Return binomial coefficient “self choose k”.

**OUTPUT:**

A symbolic expression.

**EXAMPLES:**

```python
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:

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

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

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

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

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

2.1. Symbolic Expressions 15
**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
def canonicalize_radical():  def log_expand():
    y = SR.symbol('y')
    f = log(x*y)
    f.log_expand()
    log(x) + log(y)
    f.canonicalize_radical()
    log(x) + log(y)
```

And also handles some exponential functions:

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

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

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

```sage
def canonicalize_radical():  def a = SR.symbol('a')
    f = (log(x+x^2)-log(x))^(a/log(1+x)^(a/2))
    f.canonicalize_radical()
    log(x + 1)^(1/2*a)
```
The simplest example of counter-intuitive behavior is what happens when we take the square root of a square:

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

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 \( \log(x) = \log(r) + i(t + 2k\pi) \) 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 \( e_1 \) and \( e_2 \) 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 [GitHub issue #3520](https://github.com/sagemath/sage/issues/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
```

### coefficient \((s, n=1)\)

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

**INPUT:**

- \( s \) - expression
Symbolic Calculus, Release 10.3

- n - expression, default 1

OUTPUT:
A symbolic expression. The coefficient of \( s^n \).

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

EXAMPLES:

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

Any coefficient can be queried:

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

**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 \texttt{sparse},

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

EXAMPLES:

```python
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
-x^2 + x + 1
sage: p.coefficients(a)
[[[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]
```

\texttt{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.

\textbf{Note: } The expression is not expanded or factored before the grouping takes place. For best results, call \texttt{expand()} on the expression before \texttt{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 \(4y + z\):

```python
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
```

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

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

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

`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`(deep=False)

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

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

**EXAMPLES:**

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

```
sage: fcombine()
((x - 1)*x + y^2)/(x^2 - 7) + (b + c)/a + 1/(x + 1)
```

```
sage: (1/x + 1/x^2 + (x+1)/x).combine()
(1 + x + 1/x^2
```

(continues on next page)
\[
\frac{(x + 2)}{x} + \frac{((x + 1)/x - 1/x)}{x^2}
\]

```sage```
ex.combine(deep=True)
(x + 2)/x + 1/x^2
```

```sage```
(1+sin((x + 1)/x - 1/x)).combine(deep=True)
sin(1) + 1
```

**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```
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 use `unhold()`:

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

**content** *(s)*

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

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

**INPUT:**

- s – a symbolic expression.
OUTPUT:

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

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

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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:
\texttt{sage}: \ a = \text{var}'a'\)
\texttt{sage}: \ a - a\text{.convert}()
\texttt{0}

Raises \texttt{ValueError} if self and target are not convertible:

\texttt{sage}: \ \text{units\_mass\_kilogram}\text{.convert}\left(\text{units\_length\_foot}\right)
Traceback (most recent call last):
...\n\texttt{ValueError: Incompatible units}
\texttt{sage}: \ \left(\text{units\_length\_meter}^2\right)\text{.convert}\left(\text{units\_length\_foot}\right)
Traceback (most recent call last):
...\n\texttt{ValueError: Incompatible units}

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

\texttt{sage}: \ \left(\text{units\_length\_foot}/\text{units\_time\_second}^2\right)\text{.convert}\left(\text{units\_acceleration.}\rightarrow\text{galileo}\right)
762/25*\text{galileo}
\texttt{sage}: \ \left(\text{units\_mass\_kilogram*units\_length\_meter}/\text{units\_time\_second}^2\right)\text{.convert}\left(\text{units\_force\_newton}\right)
newton
\texttt{sage}: \ \left(\text{units\_length\_foot}^3\right)\text{.convert}\left(\text{units\_area\_acre*units\_length\_inch}\right)
1/3630*\text{(acre\_inch)}
\texttt{sage}: \ \left(\text{units\_charge\_coulomb}\right)\text{.convert}\left(\text{units\_current\_ampere*units\_time\_second}\right)
\left(\text{ampere*second}\right)
\texttt{sage}: \ \left(\text{units\_pressure\_pascal*units\_si\_prefixes\_kilo}\right)\text{.convert}\left(\text{units\_pressure.}\rightarrow\text{pounds\_per\_square\_inch}\right)
129032000000/889643230521*\text{pounds\_per\_square\_inch}

For decimal answers multiply by 1.0:

\texttt{sage}: \ \left(\text{units\_pressure\_pascal*units\_si\_prefixes\_kilo}\right)\text{.convert}\left(\text{units\_pressure.}\rightarrow\text{pounds\_per\_square\_inch}\right)*1.0
0.145037737730209*\text{pounds\_per\_square\_inch}

Converting temperatures works as well:

\texttt{sage}: \ s = 68*\text{units\_temperature\_fahrenheit}
\texttt{sage}: \ s\text{.convert}(\text{units\_temperature\_celsius})
20*\text{celsius}
\texttt{sage}: \ s\text{.convert}()
293.150000000000*\text{kelvin}

Trying to multiply temperatures by another unit then converting raises a \texttt{ValueError}:

\texttt{sage}: \ \text{wrong} = 50*\text{units\_temperature\_celsius*units\_length\_foot}
\texttt{sage}: \ \text{wrong}\text{.convert}()
Traceback (most recent call last):
...\n\texttt{ValueError: cannot convert}

\texttt{cos (hold=False)}
Return the cosine of self.

\textbf{EXAMPLES}:
Symbolic Calculus, Release 10.3

```
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 use `unhold()`:

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

`:math:`\cosh (\text{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(1.00000000000000000000000000000000).cosh()
1.5430806348152437784779056
sage: SR(RIF(1)).cosh()
1.543080634815244?
```
To prevent automatic evaluation use the `hold` argument:

```
\begin{verbatim}
sage: arcsinh(x).cosh()
sqrt(x^2 + 1)
sage: arcsinh(x).cosh(hold=True)
cosh(arcsinh(x))
\end{verbatim}
```

This also works using functional notation:

```
\begin{verbatim}
sage: cosh(arcsinh(x), hold=True)
cosh(arcsinh(x))
sage: cosh(arcsinh(x))
sqrt(x^2 + 1)
\end{verbatim}
```

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

```
\begin{verbatim}
sage: a = arcsinh(x).cosh(hold=True); a.unhold()
sqrt(x^2 + 1)
\end{verbatim}
```

`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:**

```
\begin{verbatim}
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
\end{verbatim}
```

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

```
\begin{verbatim}
sage: SR(I).csgn(hold=True)
csgn(I)
\end{verbatim}
```

`decl_assume (decl)`
**decl_forget** *(decl)*

**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 power of $s$ in self.

**OUTPUT:**

An integer

**EXAMPLES:**

```
sage: var('x, y, a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^10 + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
sage: f.degree(x)
3
sage: f.degree(y)
1
sage: f.degree(sin(x*y))
1
sage: (x^-3+y).degree(x)
0
sage: (1/x+1/x**2).degree(x)
-1
```

**demoivre** *(force=False)*

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

**EXAMPLES:**

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

(continues on next page)
cos(x) + I*sin(x)
sage: exp(x).demoivre()
e^x
sage: exp(x).demoivre(force=True)
cosh(x) + sinh(x)

\textbf{denominator} (normalize=True)

Return the denominator of this symbolic expression

**INPUT:**

- normalize – (default: True) a boolean.

If \texttt{normalize} is True, the expression is first normalized to have it as a fraction before getting the denominator.

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

\textbf{See also:}

\texttt{normalize()}, \texttt{numerator()}, \texttt{numerator_denominator()}, \texttt{combine()}

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{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 \texttt{derivative()} function for more details.

\textbf{See also:}

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

\textbf{EXAMPLES:}

\begin{verbatim}
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
\end{verbatim}
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)

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

sage: _ = var('x', domain='real')
sage: _ = var('w z')
sage: (x^z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
sage: (w^z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
sage: atanh(x).real_part().diff(x)
-1/(x^2 - 1)
sage: atanh(x).imag_part().diff(x)
0
sage: atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
sage: atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
sage: forget()
sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x,y)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y,x)
(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)*cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
sage: derivative(h,x,3)
8*sin(x)*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)
**Symbolic Calculus, Release 10.3**

```
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)
```

**diff**(*args*)

Return the derivative of this expression 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)
```

Some expressions can’t be cleanly differentiated by the chain rule:
```python
sage: _ = var('x', domain='real')
sage: _ = var('w z')
sage: (x^z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
sage: (w^z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
sage: atanh(x).real_part().diff(x)
-1/(x^2 - 1)
sage: atanh(x).imag_part().diff(x)
0
sage: atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
sage: atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
sage: forget()
sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
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))
```

30 Chapter 2. Internal functionality supporting calculus
**differentiate (*args)**

Return the derivative of this expression 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:**

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

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

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

```python
sage: _ = var('x', domain='real')
sage: _.real_part().diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: forget()
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```
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)
```

distribute (recursive=True)

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

Implemented cases (so far):

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

INPUT:

- recursive = (default : True) the distribution proceeds along the subtrees of the expression.

AUTHORS:
divide_both_sides \( (x, \text{checksign} = \text{None}) \)

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:

```python
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 \((\text{hold} = \text{False})\)

Return exponential function of self, i.e., \( e \) to the power of self.

EXAMPLES:

```python
sage: x.exp()
e^x
sage: SR(0).exp()
1
sage: SR(1/2).exp()
e^{1/2}
1.64872127070013
sage: SR(0.5).exp()
1.6487212707001282
sage: math.exp(0.5)
1.6487212707001282
sage: (pi*I).exp().log()
0.500000000000000
sage: (pi*I).exp()
e^{I*pi}
-1
```

To prevent automatic evaluation use the hold argument:

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

This also works using functional notation:

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

To then evaluate again, we use unhold():

```python
sage: a = (pi*I).exp(hold=True); a.unhold()
-1
```
**expand (side=None)**

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

**EXAMPLES:**

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

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

We expand some other expressions:

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

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

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

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
```

**expand_log (algorithm='products')**

Simplifies 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 \( a/b, a\neq1 \). \( \log(1/b) \), for integer \( b \), always simplifies. If it is set to false, all of these simplifications will be turned off.”

ALIAS: \texttt{log\_expand()} and \texttt{expand\_log()} are the same

EXAMPLES:

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

\begin{verbatim}
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
\end{verbatim}

To expand also \( \log(3/4) \) use \texttt{algorithm='all'}:

\begin{verbatim}
sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
\end{verbatim}

To expand only the power use \texttt{algorithm='powers'}:

\begin{verbatim}
sage: (log(x^6)).log_expand('powers')
6*log(x)
\end{verbatim}

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

\begin{verbatim}
sage: (log((3*x)^6)).log_expand('powers')
log(729*x^6)
\end{verbatim}

This shows that the option \texttt{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):

\begin{verbatim}
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
\end{verbatim}

AUTHORS:

- Robert Marik (11-2009)

\texttt{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.

\begin{verbatim}
sage: x, y = var('x, y')
sage: a = (x-y)^5
sage: a.expand() (continues on next page)
\end{verbatim}
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))
\frac{x^3}{y - 1} - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)

sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2

Observe that expand() also expands function arguments:

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

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

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)^((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 github issue #9424 for making the following no longer raise an error:
sage: f(8).n()
31.7752256945384

expand_trig (full=False, half_angles=False, plus=True, times=True)
Expand trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in self.
For best results, self should already be expanded.

INPUT:
• full – (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(2x)) will take place only if times is True.

OUTPUT:
A symbolic expression.

EXAMPLES:
sage: sin(5*x).expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
sage: cos(2*x + var('y')).expand_trig()
cos(2*x)*cos(y) - sin(2*x)*sin(y)

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))^3)*x)
sage: f.expand_trig(full=True)
sin((3*cos(cos(2*x))^2)*cos((sin(2*x))^2)
   + sin((cos(2*x))^2)*sin((sin(2*x))^2))
   - 2*cos(cos(2*x)^2)*sin((sin(2*x))^2)
   - cos((sin(2*x))^2)*cos((sin(2*x))^2)
   - cos((cos(2*x))^2)*sin((sin(2*x))^2))^3*x)
sage: sin(2*x).expand_trig(times=False)
sin(2*x)
sage: sin(2*x).expand_trig(times=True)
2*cos(x)*sin(x)
sage: sin(2 + x).expand_trig(plus=False)
2*cos(x)*sin(x)
sage: sin(2 + x).expand_trig(plus=True)
2*cos(x)*sin(2) + cos(2)*cos(x)
If the expression contains terms which are factored, we expand first:

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

**ALIAS:**

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

**exponentialize()**

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

**EXAMPLES:**

```python
sage: x = SR.var("x")
sage: sin(x).exponentialize()  
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
```

**factor** *(don'tfactor=*)

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

**INPUT:**

- `self` - a symbolic expression
- `don'tfactor` - 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 'don'tfactor' list.

**EXAMPLES:**

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

```python
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
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=None)`

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

**INPUT:**

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

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

**EXAMPLES:**

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

We factor a fairly straightforward expression:

```sage
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
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:
Symbolic Calculus, Release 10.3

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

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 use unhold():

\[
sage: a = SR(5).factorial(hold=True); a.unhold()
120
\]

factorial_simplify()

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

ALIAS: factorial_simplify and simplify_factorial are the same

EXAMPLES:
Some examples are relatively clear:
Symbolic Calculus, Release 10.3

```
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) + 1/2*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
```

**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]
```

2.1. Symbolic Expressions 41
**find_local_maximum** 

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

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

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

**EXAMPLES:**

```
sage: f = x*cos(x)
sage: f.find_local_maximum(0,5)  # needs scipy
(0.5610963381910451, 0.8603335890471894)
sage: f.find_local_maximum(0,5, tol=0.1, maxfun=10)  # needs scipy
(0.5610903234580815, 0.8579265014564914)
```

**find_local_minimum** 

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

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

**INPUT:**

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

**OUTPUT:**

A tuple `(minval, x)`, where

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

**EXAMPLES:**

```
sage: # needs scipy
sage: f = x*cos(x)
sage: f.find_local_minimum(1, 5)
(-3.288371395590..., 3.425618469580498)
sage: f.find_local_minimum(1, 5, tol=1e-3)
(-3.288371361890..., 3.425750790371699)
sage: f.find_local_minimum(1, 5, tol=1e-2, maxfun=10)
(-3.288370845983..., 3.425084022017125)
sage: show(f.plot(0, 20))  # needs sage.plot
sage: f.find_local_minimum(1, 15)
(-9.477294259479..., 9.529334409421940)
```

**ALGORITHM:**

Uses `sage.numerical.optimize.find_local_minimum()`.

**AUTHORS:**
• William Stein (2007-12-07)

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

Numerically find a root of self on the closed interval \([a,b]\) (or \([b,a]\)) if possible, where self is a function in 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 \(\geq 0\). 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 \(\geq 0\).
- **full_output** - bool (default: False), if True, also return object that contains information about convergence.
- **imaginary_tolerance** – (default: \(1e-8\)); if an imaginary number arises (due, for example, to numerical issues), this tolerance specifies how large it has to be in magnitude before we raise an error. In other words, imaginary parts smaller than this are ignored when we are expecting a real answer.

**EXAMPLES:**

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

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

More examples:

```python
sage: (sin(x) + exp(x)).find_root(-10, 10) % needs scipy
-0.588532743981862...
sage: sin(x).find_root(-1,1) % needs scipy
0.0
```

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

```
sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)  # needs scipy
-1.618033988749895
```

Some examples that Ted Kosan came up with:

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

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

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

It is easy to approximate with \texttt{find_root}:

```
sage: a.find_root(0,0.002)  # needs scipy
0.000411051404934985
```

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

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

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.
```
**forget()**

Forget the given constraint.

**EXAMPLES:**

```python
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]
```

**fraction(base_ring)**

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

**EXAMPLES:**

```python
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: 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
```

**free_variables()**

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

**EXAMPLES:**

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

**full_simplify()**

Apply `simplify_factorial()`, `simplify_rectform()`, `simplify_trig()`, `simplify_rational()`, and then `expand_sum()` to self (in that order).
Symbolic Calculus, Release 10.3

ALIAS: simplify_full and full_simplify are the same.

EXAMPLES:

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

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

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

**function (**args**)**

Return a callable symbolic expression with the given variables.

EXAMPLES:

We will use several symbolic variables in the examples below:

```python
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)
```

```python
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:

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

This notation also works:

```python
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 \( f(x,y) = f \):
\texttt{sage}: f = x^n + y^n; f
\texttt{x^n + y^n}
\texttt{sage}: f(x,y) = f
\texttt{sage}: f
\texttt{(x, y) \rightarrow x^n + y^n}
\texttt{sage}: f(2,3)
\texttt{3^n + 2^n}

\texttt{gamma}(\texttt{hold}=\texttt{False})

Return the Gamma function evaluated at self.

EXAMPLES:

\texttt{sage}: x = \texttt{var}('x')
\texttt{sage}: x.gamma()
\texttt{gamma}(x)
\texttt{sage}: \texttt{SR}(2).gamma()
\texttt{1}
\texttt{sage}: \texttt{SR}(10).gamma()
\texttt{362880}
\texttt{sage}: \texttt{SR}(10.0r).gamma() \texttt{# For ARM: rel tol 2e-15}
\texttt{362880.0}
\texttt{sage}: \texttt{SR}(%CF{(1,1)}).gamma()
\texttt{0.498015668183567 - 0.15494982830181067*I}

\texttt{sage}: \texttt{gp}('\texttt{gamma}(1+I)')
\texttt{0.49801566818356042713691117\ldots - 0.154949828301806851249551305*I \# 32-bit}
\texttt{0.4980156681835604271369111746219809195\ldots - 0.154949828301806851249551304838660520*I \# 64-bit}

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

\texttt{sage}: \texttt{plot}(\texttt{gamma}(x), -6, 4).\texttt{show}(\texttt{ymin}=-3, \texttt{ymax}=3) \texttt{# needs sage.plot}

To prevent automatic evaluation use the \texttt{hold} argument:

\texttt{sage}: \texttt{SR}(1/2).gamma()
\texttt{sqrt(pi)}
\texttt{sage}: \texttt{SR}(1/2).gamma(\texttt{hold=True})
\texttt{gamma}(1/2)

This also works using functional notation:

\texttt{sage}: \texttt{gamma}(1/2,\texttt{hold=True})
\texttt{gamma}(1/2)
\texttt{sage}: \texttt{gamma}(1/2)
\texttt{sqrt(pi)}

To then evaluate again, we use \texttt{unhold()}:

\texttt{sage}: a = \texttt{SR}(1/2).gamma(\texttt{hold=True}); a.\texttt{unhold()}
\texttt{sqrt(pi)}

\texttt{gamma_normalize}()

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

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

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

Check that github issue #22826 is fixed:

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

**gcd**

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

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

EXAMPLES:

```python
sage: var('x,y')
(x, y)
sage: SR(10).gcd(SR(15))
5
sage: (x^3 - 1).gcd(x-1) / (x-1) in QQ
True
sage: (x^3 - 1).gcd(x^2+x+1) / (x^2+x+1) in QQ
True
sage: (x^3 - x^2*pi + x^2 - pi^2).gcd(x-pi) / (x-pi) in QQ
True
sage: gcd(sin(x)^2 + sin(x), sin(x)^2 - 1) / (sin(x) + 1) in QQ
True
sage: gcd(x^3 - y^3, x-y) / (x-y) in QQ
True
```

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

```python
sage: gcd(I - I*x, x^2 - 1)
x - 1
sage: gcd(I + I*x, x^2 - 1)
x + 1
sage: alg = SR(QQbar(sqrt(2) + I*sqrt(3)))
sage: gcd(alg + alg*x, x^2 - 1) # known bug (trac #28489)
x + 1
sage: gcd(alg - alg*x, x^2 - 1) # known bug (trac #28489)
```
Symbolic Calculus, Release 10.3

(continued from previous page)

```
x - 1
sage: sqrt2 = SR(QQbar(sqrt(2)))
sage: gcd(sqrt2 + x, x^2 - 2)  # known bug
1
```

**gosper_sum(**``#args``**)

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

**INPUT:**

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

**EXAMPLES:**

```
sage: a, b, k, m, n = var('a b k m n')
sage: SR(1).gosper_sum(n)
n
sage: n.gosper_sum(n)
1/2*(n - 1)*n
sage: n.gosper_sum(n,0,5)
15
sage: n.gosper_sum(n,0,m)
1/2*(m + 1)*m
sage: n.gosper_sum(n,a,b)
-1/2*(a + b)*(a - b - 1)
sage: (factorial(m + n)/factorial(n)).gosper_sum(n)
n*factorial(m + n)/((m + 1)*factorial(n))
sage: (binomial(m + n, n)).gosper_sum(n)
1/120*(m + 1)*(m + 2)*(m + 3)*(m + 4)*(m + 5)*(m + 6)*gamma(a + 1)*gamma(b + 1)
```

**gosper_term(**``n``**)

Return Gosper’s hypergeometric term for self.

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

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

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

**EXAMPLES:**
Symbolic Calculus, Release 10.3

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

**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:**

\begin{verbatim}
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))
\end{verbatim}

See also:

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

**has** *(pattern)*

**EXAMPLES:**

\begin{verbatim}
sage: var('x,y,a'); w0 = SR.wild(); w1 = SR.wild()
(x, y, a)
sage: (x*sin(x + y + 2*a)).has(y)
True
sage: (x*sin(x + y + 2*a)).has(x+y)
False
sage: (x*sin(x + y + 2*a)).has(x + y + w0)
True
\end{verbatim}

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

\begin{verbatim}
sage: (x*sin(x + y + 2*a)).has(x+y)
False
sage: (x*sin(x + y + 2*a)).has(x + y + w0)
True
\end{verbatim}

The following fails because “2*(x+y)” automatically gets converted to “2*x+2*y” of which “x+y” is not a subexpression:
Although \( x^1 = x \) and \( x^0 = 1 \), neither "x" nor "1" are actually of the form "x^something":

\[
sage: (x+1).has(x^w0) \quad \text{False}
\]

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 \texttt{coeff()} function instead.

\[
sage: (4*x^2 - x + 3).has(w0*x) \quad \text{True}
sage: (4*x^2 + x + 3).has(w0*x) \quad \text{False}
sage: (4*x^2 + x + 3).has(x) \quad \text{True}
sage: (4*x^2 - x + 3).coefficient(x,1) \quad -1
sage: (4*x^2 + x + 3).coefficient(x,1) \quad 1
\]

\textbf{has\_wild()}

Return \texttt{True} if this expression contains a wildcard.

**EXAMPLES:**

\[
sage: (1 + x^2).has\_wild() \quad \text{False}
sage: (SR.wild(0) + x^2).has\_wild() \quad \text{True}
sage: SR.wild(0).has\_wild() \quad \text{True}
\]

\textbf{hessian()}

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

**EXAMPLES:**

\[
sage: x, y = var('x y')
sage: f = x^2+y^2
sage: f.hessian()
\begin{bmatrix}
2 & 0 \\
0 & 2 \\
\end{bmatrix}
sage: g(x,y) = x^2+y^2
sage: g.hessian()
\begin{bmatrix}
(2*x, y) |--> 2 (x, y) |--> 0 \\
(2*y, x) |--> 0 (x, y) |--> 2 \\
\end{bmatrix}
\]

\textbf{horner(x)}

Rewrite this expression as a polynomial in Horner form in \( x \).

**EXAMPLES:**

\[
sage: add((i+1)*x^i for i in range(5)).horner(x) \quad (((5*x + 4)*x + 3)*x + 2)*x + 1
\]
Symbolic Calculus, Release 10.3

(continued from previous page)

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

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

sage: expr.horner(sin(x))
(5*cos(x)^4 - (10*cos(x)^2 - sin(x)^2)*sin(x)^2)*sin(x)

sage: expr.horner(cos(x))
sin(x)^5 + 5*(cos(x)^2*sin(x) - 2*sin(x)^3)*cos(x)^2

hypergeometric_simplify(algorithm='maxima')

Simplify an expression containing hypergeometric or confluent hypergeometric functions.

INPUT:

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

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

EXAMPLES:

sage: hypergeometric((5, 4), (4, 1, 2, 3), x).simplify_hypergeometric()
1/144*x^2*hypergeometric((,), (3, 4), x) + ...
1/3*x*hypergeometric((,), (2, 3), x) + hypergeometric((,), (1, 2), x)

sage: (2*hypergeometric((,), ()), x)).simplify_hypergeometric()
2*e^x

sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1) # not tested,
˓→ unstable
....: .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)

sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1) # not tested,
˓→ unstable
....: .simplify_hypergeometric(algorithm='sage'))
hypergeometric(hypergeometric((e^x,),(1,),x),(1,),x)

sage: hypergeometric_M(1, 3, x).simplify_hypergeometric()
-2*(x + 1)*e^(-x) - 1)*e^x/x^2

sage: (2*hypergeometric_U(1, 3, x)).simplify_hypergeometric()
2*(x + 1)/x^2

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 −π < Im(z) <= π, 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))
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: SR(I).imag_part()
1
sage: SR(I).imag_part(hold=True)
imag_part(I)
```

This also works using functional notation:

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

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

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

`imag_part` *(hold=False)*

Return the imaginary part of this symbolic expression.

**EXAMPLES:**

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

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

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

A more symbolic example:

```
sage: var('a, b')
(a, b)
```
Symbolic Calculus, Release 10.3

(continued from previous page)

\[ \text{sage: } f = \log(a + b \cdot i) \]
\[ \text{sage: } f.\text{imag_part()} \]
\[ \text{arctan2}(\text{imag_part}(a) + \text{real_part}(b), -\text{imag_part}(b) + \text{real_part}(a)) \]

Using the \texttt{hold} parameter it is possible to prevent automatic evaluation:

\[ \text{sage: } \text{SR}(\text{I}).\text{imag_part()} \]
\[ 1 \]
\[ \text{sage: } \text{SR}(\text{I}).\text{imag_part(}\text{hold=True}) \]
\[ \text{imag_part}(\text{I}) \]

This also works using functional notation:

\[ \text{sage: } \text{imag_part(}\text{I}, \text{hold=True}) \]
\[ \text{imag_part}(\text{I}) \]
\[ \text{sage: } \text{imag_part(SR}\text{(I)}) \]
\[ 1 \]

To then evaluate again, we use \texttt{unhold()}:

\[ \text{sage: } a = \text{SR}(\text{I}).\text{imag_part(}\text{hold=True}) \]; \text{a.unhold()} \]
\[ 1 \]

\texttt{implicit_derivative}(Y, X, n=1)

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

\textbf{INPUT:}

\begin{itemize}
  \item \textit{Y} – The dependent variable of the implicit expression.
  \item \textit{X} – The independent variable with respect to which the derivative is taken.
  \item \textit{n} – (default: 1) the order of the derivative.
\end{itemize}

\textbf{EXAMPLES:}

\[ \text{sage: } \text{var('x, y') } \]
\[ (x, y) \]
\[ \text{sage: } f = \cos(x) \cdot \sin(y) \]
\[ \text{sage: } f.\text{implicit_derivative}(y, x) \]
\[ \sin(x) \cdot \sin(y)/(\cos(x) \cdot \cos(y)) \]
\[ \text{sage: } g = x \cdot y^2 \]
\[ \text{sage: } g.\text{implicit_derivative}(y, x, 3) \]
\[ -1/4*(y + 2*y/x)/x^2 + 1/4*(2*y^2/x - y^2/x^2)/(x*y) - 3/4*y/x^3 \]

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

\[ \text{sage: } (\cos(x) \cdot \sin(x)).\text{implicit_derivative}(y, x) \]
\[ \text{Traceback (most recent call last):} \]
\[ ... \]
\[ \text{ValueError: Expression } \cos(x) \cdot \sin(x) \text{ contains no } y \text{ terms} \]

\texttt{integral(*args, **kwds)}

Compute the integral of \texttt{self}.

Please see \texttt{sage.symbolic.integration.integral.integrate()} for more details.

\textbf{EXAMPLES:}
Symbolic Calculus, Release 10.3

\begin{Verbatim}
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
\end{Verbatim}

\texttt{integrate}(\texttt{\ast args, \ast kwds})

Compute the integral of \texttt{self}.

Please see \texttt{sage.symbolic.integration.integral.integrate()} for more details.

\textbf{EXAMPLES:}

\begin{Verbatim}
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
\end{Verbatim}

\begin{Verbatim}
inverse_laplace(t, s)

Return inverse Laplace transform of \texttt{self}.

See \texttt{sage.calculus.calculus.inverse_laplace}

\textbf{EXAMPLES:}

\begin{Verbatim}
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)
\end{Verbatim}

\begin{Verbatim}
is_algebraic()

Return True if this expression is known to be algebraic.

\textbf{EXAMPLES:}

\begin{Verbatim}
sage: sqrt(2).is_algebraic()
True
sage: (5*sqrt(2)).is_algebraic()
True
sage: (sqrt(2) + 2^(1/3) - 1).is_algebraic()
True
sage: (I*golden_ratio + sqrt(2)).is_algebraic()
True
sage: (sqrt(2) + pi).is_algebraic()
False
sage: SR(QQ(2/3)).is_algebraic()
True
sage: SR(1.2).is_algebraic()
False
sage: complex_root_of(x^3 - x^2 - x - 1, 0).is_algebraic()
True
\end{Verbatim}

\begin{Verbatim}
is_callable()

Return True if \texttt{self} is a callable symbolic expression.

\textbf{EXAMPLES:}

\end{Verbatim}
is_constant()  
Return whether this symbolic expression is a constant.  
A symbolic expression is constant if it does not contain any variables.

EXAMPLES:

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

is_exact()  
Return True if this expression only contains exact numerical coefficients.

EXAMPLES:

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

is_infinity()  
Return True if self is an infinite expression.

EXAMPLES:

```plaintext
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
```

Assumptions on symbols are handled correctly:

```
sage: y = var('y')
sage: assume(y < 0)
sage: y.is_positive()
False
sage: y.is_negative()
True
sage: forget()
```

**is_negative_infinity()**

Return True if self is a negative infinite expression.

**EXAMPLES:**

```
sage: SR(oo).is_negative_infinity()
False
sage: SR(-oo).is_negative_infinity()
True
sage: x.is_negative_infinity()
False
```

**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
```

(continues on next page)
**is_numeric**

Return **False** if this expression is known to be numeric.

**is_polynomial**(var)

Return **True** if self is a polynomial in the given variable.

**is_positive**()

Return **True** if this expression is known to be positive.

**is_positive**(var)

Return **True** if self is a polynomial in the given variable.
sage: f = function('f')(x)
sage: assume(f>0)
sage: f.is_positive()
True
sage: forget()

\textbf{is\_positive\_infinity()}

Return \texttt{True} if \texttt{self} is a positive infinite expression.

\textbf{EXAMPLES:}

sage: SR(oo).is_positive_infinity()
True
sage: SR(-oo).is_positive_infinity()
False
sage: x.is_infinity()
False

\textbf{is\_rational\_expression()}

Return \texttt{True} if this expression if a rational expression, i.e., a quotient of polynomials.

\textbf{EXAMPLES:}

sage: var('x y z')
(x, y, z)
sage: ((x + y + z)/(1 + x^2)).is_rational_expression()
True
sage: ((1 + x + y)^10).is_rational_expression()
True
sage: ((1/x + z)^5 - 1).is_rational_expression()
True
sage: (1/(x + y)).is_rational_expression()
True
sage: (exp(x) + 1).is_rational_expression()
False
sage: (sin(x*y) + z^3).is_rational_expression()
False
sage: (exp(x) + exp(-x)).is_rational_expression()
False

\textbf{is\_real()}

Return \texttt{True} if this expression is known to be a real number.

\textbf{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+t1).is_real()
False
sage: (t0*t1).is_real()
True

(continues on next page)
The following is real, but we cannot deduce that:

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

Assumption of real has the same effect as setting the domain:

```sage
sage: forget()
sage: assume(x, 'real')
sage: x.is_real()
True
sage: cosh(x).is_real()
True
sage: forget()
```

The real domain is also set with the integer domain:

```sage
sage: SR.var('x', domain='integer').is_real()
True
```

**is_relational()**

Return True if self is a relational expression.

EXAMPLES:

```sage
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.is relational()
True
```
\textbf{is\_square()} 
Return \texttt{True} if \texttt{self} is the square of another symbolic expression. 

This is \texttt{True} for all constant, non-relational expressions (containing no variables or comparison), and not implemented otherwise. 

\textbf{EXAMPLES:}

\begin{verbatim}
sage: SR(4).is_square() True sage: SR(5).is_square() True sage: pi.is_square() True sage: x.is_square() Traceback (most recent call last): ... NotImplementedError: is_square() not implemented for non-constant or relational elements of Symbolic Ring sage: r = SR(4) == SR(5) sage: r.is_square() Traceback (most recent call last): ... NotImplementedError: is_square() not implemented for non-constant or relational elements of Symbolic Ring
\end{verbatim}

\textbf{is\_symbol()} 
Return \texttt{True} if this symbolic expression consists of only a symbol, i.e., a symbolic variable. 

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{is\_terminating\_series()} 
Return \texttt{True} if \texttt{self} is a series without order term. 

A series is terminating if it can be represented exactly, without requiring an order term. You can explicitly request terminating series by setting the order to positive infinity. 

\textbf{OUTPUT:}
Boolean. Whether `self` was constructed by `series()` and has no order term.

**EXAMPLES:**

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

```python
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:

```python
sage: t = pi + (pi - 1)*pi - pi^2
sage: t.is_trivial_zero()
False
sage: t.is_zero()
True
sage: t = pi + x*pi + (pi - 1 - x)*pi - pi^2
sage: t.is_trivial_zero()
True
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_trivially_equal(other)`

Check if this expression is trivially equal to the argument expression, without any simplification.

Note that the expressions may still be subject to immediate evaluation.

This method is intended to be used in library code where trying to obtain a mathematically correct result by applying potentially expensive rewrite rules is not desirable.
EXAMPLES:

```python
sage: (x^2).is_trivially_equal(x^2)
True
sage: ((x+1)^2 - 2*x - 1).is_trivially_equal(x^2)
False
sage: (x*(x+1)).is_trivially_equal((x+1)*x)
True
sage: (x^2 + x).is_trivially_equal((x+1)*x)
False
sage: ((x+1)*(x+1)).is_trivially_equal((x+1)^2)
True
sage: (x^2 + 2*x + 1).is_trivially_equal((x+1)^2)
False
sage: (x^-1).is_trivially_equal(1/x)
True
sage: (x/x^2).is_trivially_equal(1/x)
True
sage: ((x^2+x) / (x+1)).is_trivially_equal(1/x)
False
```

**is_unit()**

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

Note that a proof may be attempted to get the result. To avoid this use `(ex-1).is_trivial_zero()`.

EXAMPLES:

```python
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:

```python
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:

```python
sage: x.iterator()
Traceback (most recent call last):
... ValueError: expressions containing only a numeric coefficient, constant or symbol have no operands
```

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

(continues on next page)
ValueError: expressions containing only a numeric coefficient, constant or symbol have no operands

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

```python
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)
```

```python
laurent_polynomial(base_ring=None, ring=None)
```

Return this symbolic expression as a Laurent polynomial over the given base ring, if possible.

INPUT:

- *base_ring* (optional) the base ring for the polynomial
- *ring* (optional) the parent for the polynomial

You can specify either the base ring (`base_ring`) you want the output Laurent polynomial to be over, or you can specify the full Laurent polynomial ring (`ring`) you want the output laurent polynomial to be an element of.

EXAMPLES:

```
sage: f = x^2 -2/3/x + 1
sage: f.laurent_polynomial(QQ)
-2/3*x^-1 + 1 + x^2
sage: f.laurent_polynomial(GF(19))
12*x^-1 + 1 + x^2
```

```python
lcm(b)
```

Return the lcm of self and b.

The lcm is computed from the gcd of `self` and `b` implicitly from the relation `self * b = gcd(self, b) * lcm(self, b)`.

**Note:** In agreement with the convention in use for integers, if `self * b == 0`, then `gcd(self, b) == max(self, b)` and `lcm(self, b) == 0`.

**Note:** Since the polynomial lcm is computed from the gcd, and the polynomial gcd is unique up to a constant factor (which can be negative), the polynomial lcm is unique up to a factor of `-1`.

EXAMPLES:
Symbolic Calculus, Release 10.3

\begin{verbatim}
sage: var('x,y')
(x, y)
sage: SR(10).lcm(SR(15))
30
sage: (x^3 - 1).lcm(x-1)
x^3 - 1
sage: (x^3 - 1).lcm(x^2+x+1)
x^3 - 1
sage: (x^3 - sage.symbolic.constants.pi).lcm(x-sage.symbolic.constants.pi)
(pi - x^3)*(pi - x)

sage: lcm(x^3 - y^3, x-y) / (x^3 - y^3) in [1,-1]
True
sage: lcm(x^100-y^100, x^10-y^10) / (x^100 - y^100) in [1,-1]
True

sage: a = expand((x^2+17*x+3/7*y)*(x^5 - 17*y + 2/3))
sage: b = expand((x^13+17*x+3/7*y)*(x^5 - 17*y + 2/3))
sage: gcd(a,b) * lcm(a,b) / (a * b) in [1,-1]
True

The result is not automatically simplified:

\begin{verbatim}
sage: ex = lcm(sin(x)^2 - 1, sin(x)^2 + sin(x)); ex
(sin(x)^2 + sin(x))*(sin(x)^2 - 1)/(sin(x) + 1)
sage: ex.simplify_full()
sin(x)^3 - sin(x)
\end{verbatim}

\end{verbatim}

\textbf{leading\_coeff} (s)

Return the leading coefficient of s in self.

\textbf{EXAMPLES}:

\begin{verbatim}
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
\end{verbatim}

\textbf{leading\_coefficient} (x)

Return the leading coefficient of s in self.

\textbf{EXAMPLES}:

\begin{verbatim}
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
\end{verbatim}

2.1. Symbolic Expressions 65
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:

```python
sage: var('x, y, a')
(x, y, a)
sage: (x^5).list()
[0, 0, 0, 0, 1]
sage: p = x - x^3 + 5/7*x^5
sage: p.list()
[0, -1, 0, 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]
```

\( \log(b=None, \text{hold}=False) \)

Return the logarithm of self.

EXAMPLES:

```python
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) #...
needs sage.plot
```

To prevent automatic evaluation use the \texttt{hold} argument:

```python
sage: I.log()
1/2*I*pi
sage: I.log(hold=True)
\log(I)
```

To then evaluate again, we use \texttt{unhold()}:
The `hold` parameter also works in functional notation:

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

`log_expand` *(algorithm='products')*

Simplify symbolic expression, which can contain logs.

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

**INPUT:**

- `self` - expression to be simplified
- `algorithm` - (default: 'products') 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 \( a/b, a \neq 1 \). (\( \log(1/b) \), 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:

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

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

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

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

```python
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:

```python
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):

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

**AUTHORS:**

- Robert Marik (11-2009)

### log_gamma

**(hold=False)**

Return the log gamma function evaluated at self. This is the logarithm of gamma of self, where gamma is a complex function such that \(\gamma(n)\) equals \(\text{factorial}(n-1)\).

**EXAMPLES:**

```python
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()
3.1805383034795
sage: SR(5-1).factorial().log()
log(24)
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: plot(lambda x: SR(x).log_gamma(), -7,8, plot_points=1000).show()  # needs sage.plot
1.648721270700128
```

To prevent automatic evaluation use the `hold` argument:

```python
sage: SR(5).log_gamma(hold=True)
log_gamma(5)
```

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

```python
sage: a = SR(5).log_gamma(hold=True); a.n()
3.1805383034795
```

### 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 \( \text{algorithm} \).

**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
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
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
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
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
sage: f = log(x)+2*log(y)-log(t)
sage: f.simplify_log('one')
2*log(y) + log(x/t)
```
\[\pi\] is an irrational number; to contract logarithms in the following example we have to set \texttt{algorithm} to \texttt{'constants'} or \texttt{'all'}:

\begin{verbatim}
    sage: f = log(x)+log(y)-\pi*log((x+1))
    sage: f.simplify_log('constants')
    log(x*y/(x + 1)^{\pi})
\end{verbatim}

\(x*\log(9)\) is contracted only if \texttt{algorithm} is \texttt{'all'}:

\begin{verbatim}
    sage: (x*log(9)).simplify_log('all')
    log(3^{2*x})
\end{verbatim}

\textbf{AUTHORS:}

\begin{itemize}
    \item Robert Marik (11-2009)
\end{itemize}

\textbf{low_degree}(\(s\))

\begin{quote}
Return the exponent of the lowest power of \(s\) in \texttt{self}.
\end{quote}

\textbf{OUTPUT:}

An integer

\textbf{EXAMPLES:}

\begin{verbatim}
    sage: var('x,y,a')
    (x, y, a)
    sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^{10} + 2*sin(x*y)/x; f
    x^{3}*\sin(x*y) + a*x + x*y + 2*\sin(x*y)/x + x/y^{10} + 100
    sage: f.low_degree(x)
    -1
    sage: f.low_degree(y)
    -10
    sage: f.low_degree(sin(x*y))
    0
    sage: (x^3+y).low_degree(x)
    0
    sage: (x+x^2).low_degree(x)
    1
\end{verbatim}

\textbf{match}(\texttt{pattern})

\begin{quote}
Check if \texttt{self} matches the given pattern.
\end{quote}

\textbf{INPUT:}

\begin{itemize}
    \item \texttt{pattern} – a symbolic expression, possibly containing wildcards to match for
\end{itemize}

\textbf{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)^a).match(w0^w0)
{$0: x + y}$
sage: t = ((a+b)*(a+c)).match((a+w0)*(a+w1))
sage: set([t[w0], t[w1]]) == set([b, c])
True
sage: ((a+b)*(a+c)).match((w0+b)*(w0+c))
{$0: a}$
sage: t = ((a+b)*(a+c)).match((w0+w1)*(w0+w2))
sage: t[w0]  
(a+b)
set([t[w1], t[w2]]) == set([b, c])
True
sage: t = ((a+b)*(a+c)).match((w0+w1)*(w1+w2))
sage: t[w1]
(a+b)
set([t[w0], t[w2]]) == set([b, c])
True
sage: t = (a*(x+y)+a*z+b).match(a*w0+w1)
sage: s = set([t[w0], t[w1]])
sage: s == set([x+y, a*z+b]) or s == set([z, a*(x+y)+b])
True
sage: print((a+b+c+d+f+g).match(c))
None
sage: (a+b+c+d+f+g).has(c)
True
sage: (a+b+c+d+f+g).match(c+w0)
{$0: a + b + d + f + g}$
sage: (a+b+c+d+f+g).match(c+g+w0)
{$0: a + b + d + f}$
sage: (a+b).match(a+b+w0)  # known bug
{$0: 0}$
sage: print((a*b^2).match(a^w0*b^w1))
None
sage: (a*b^2).match(a*b^w1)
{$1: 2}$
sage: (x*x.arctan2(x^2)).match(w0*w0.arctan2(w0^2))
{$0: x$}
```

Beware that behind-the-scenes simplification can lead to surprising results in matching:
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:**

```python
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
```

```python
sage: res = t.maxima_methods().logcontract(); res
log((sqrt(2) + 1)*(sqrt(2) - 1))
```

```python
sage: type(res)
<class 'sage.symbolic.expression.Expression'>
```

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

Return the minimal polynomial of this symbolic expression.

**EXAMPLES:**

```python
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:**

```python
sage: x.mul(x)
x^2
```

```python
sage: x.mul(x, hold=True)
x^2
```

```python
sage: x.mul(x, (2+x), hold=True)
x^2*(x + 2)
```

```python
sage: x.mul(x, (2+x), x, hold=True)
x^2*(x + 2)*x
```

```python
sage: x.mul(x, (2+x), x, 2*x, hold=True)
x^2*(x + 2)*x^2
```

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

```python
sage: a = x.mul(x, hold=True); a.unhold()
x^2
```

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

Return a relation obtained by multiplying both sides of this relation by \(x\).

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

```python
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:

```python
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:

```python
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)
```

**negation()**

Return the negated version of self.

This is the relation that is False iff self is True.

EXAMPLES:

```python
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:

```python
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 operands 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

$$
\text{norm}(c) = \text{norm}(a + bi) = c \cdot \overline{c} = a^2 + b^2.
$$

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 $\mathbb{Z}[i]$ 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

See also:

`numerator()`, `denominator()`, `numerator_denominator()`, `combine()`

**EXAMPLES:**
Symbolic Calculus, Release 10.3

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^2*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

**number_of_operands()**

Return the number of operands of this expression.

EXAMPLES:

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

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)
```

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()

EXAMPLES:

```
sage: x, y, a = var("x y a")
sage: ((x+y)^2/(x-y)^3*x^3).numerator_denominator()
((x + y)^2* x^3, (x - y)^3)
sage: ((x+y)^2/(x-y)^3*x^3).numerator_denominator(False)
```

(continues on next page)
\[(x + y)^{2}x^{3}, (x - y)^{3}\]

\[
\begin{align*}
sage: & \quad g = x + y/(x + 2) \\
sage: & \quad g.numerator_denominator() \\
& \quad (x^2 + 2*x + y, x + 2) \\
sage: & \quad g.numerator_denominator(normalize=False) \\
& \quad (x + y/(x + 2), 1) \\
sage: & \quad g = x^2*(x + 2) \\
sage: & \quad g.numerator_denominator() \\
& \quad ((x + 2)*x^2, 1) \\
sage: & \quad g.numerator_denominator(normalize=False) \\
& \quad ((x + 2)*x^2, 1)
\end{align*}
\]

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

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

No guarantee is made about the accuracy of the result.

**INPUT:**

- **prec** – precision in bits
- **digits** – precision in decimal digits (only used if prec is not given)
- **algorithm** – which algorithm to use to compute this approximation

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

**EXAMPLES:**

\[
\begin{align*}
sage: & \quad \sin(x).subs(x=5).n() \\
& \quad -0.958924274663138 \\
sage: & \quad \sin(x).subs(x=5).n(100) \\
& \quad -0.9589242746631384689315440616 \\
sage: & \quad \sin(x).subs(x=5).n(digits=50) \\
& \quad -0.9589242746631384689315440615599397335246154396460 \\
sage: & \quad \text{zeta}(x).subs(x=2).numerical_approx(digits=50) \\
& \quad 1.6449340668482264364724151666460251892189499012068 \\
sage: & \quad \cos(3).numerical_approx(200) \\
& \quad -0.98999249660044545727157279473126130239367909661558832881409 \\
sage: & \quad \text{numerical_approx}(\cos(3)), 200) \\
& \quad -0.98999249660044545727157279473126130239367909661558832881409 \\
sage: & \quad \text{numerical_approx}(\cos(3), digits=10) \\
& \quad -0.9899924966 \\
sage: & \quad (1 + i)\text{numerical_approx}(32) \\
& \quad 1.00000000 + 1.00000000*I \\
sage: & \quad (\pi + e + \sqrt{2})\text{numerical_approx}(100) \\
& \quad 7.2740880444219335226246195788
\end{align*}
\]

**op**

Provide access to the operands of an expression through a property.

**EXAMPLES:**

\[
\begin{align*}
sage: & \quad t = 1+x+x^2 \\
sage: & \quad t.op \\
& \quad \text{Operands of } x^2 + x + 1
\end{align*}
\]
Indexing directly with `t[1]` causes problems with numpy types.

```
sage: t[1] Traceback (most recent call last): ... TypeError: 'sage.symbolic.expression.Expression' object...
```

**operands()**

Return a list containing the operands of this expression.

**EXAMPLES:**

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

**operator()**

Return the topmost operator in this expression.

**EXAMPLES:**

```
sage: x,y,z = var('x,y,z')
sage: (x+y).operator()
<function add_vararg ...>
sage: (x^y).operator()<built-in function pow>
sage: (x^y * z).operator() <function mul_vararg ...>
sage: (x < y).operator()<built-in function lt>
sage: abs(x).operator() abs
sage: r = gamma(x).operator(); type(r)
<class 'sage.functions.gamma.Function_gamma'>
sage: psi = function('psi', nargs=1)
sage: psi(x).operator() psi
sage: r = psi(x).operator()
sage: r == psi True
sage: f = function('f', nargs=1, conjugate_func=lambda self, x: 2*x)
sage: nf = f(x).operator()
```

(continues on next page)
partial_fraction (\texttt{var=None})

Return the partial fraction expansion of \texttt{self} with respect to the given variable.

INPUT:

- \texttt{var} – variable name or string (default: first variable)

OUTPUT:

A symbolic expression

See also:

\texttt{partial_fraction_decomposition()}

EXAMPLES:

\begin{verbatim}
sage: f = x^2/(x+1)^3
sage: f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
\end{verbatim}

Notice that the first variable in the expression is used by default:

\begin{verbatim}
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
\end{verbatim}

You can explicitly specify which variable is used:

\begin{verbatim}
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
\end{verbatim}

partial_fraction_decomposition (\texttt{var=None})

Return the partial fraction decomposition of \texttt{self} with respect to the given variable.

INPUT:

- \texttt{var} – variable name or string (default: first variable)

OUTPUT:

A list of symbolic expressions

See also:

\texttt{partial_fraction()}

EXAMPLES:
**Symbolic Calculus, Release 10.3**

```
sage: f = x^2/(x+1)^3
sage: f.partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3)]
sage: (4+f).partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3), 4]
```

Notice that the first variable in the expression is used by default:

```
sage: y = var('y')
sage: f = y^2/(y+1)^3
sage: f.partial_fraction_decomposition()
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3)]
sage: f = y^2/(y+1)^3 + x/(x-1)^3
sage: f.partial_fraction_decomposition()
[y^2/(y^3 + 3*y^2 + 3*y + 1), (x - 1)^(-2), (x - 1)^(-3)]
```

You can explicitly specify which variable is used:

```
sage: f.partial_fraction_decomposition(y)
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3), x/(x^3 - 3*x^2 + 3*x - 1)]
```

**plot (**args, **kwargs)**

Plot a symbolic expression. All arguments are passed onto the standard plot command.

**EXAMPLES:**

This displays a straight line:

```
sage: sin(2).plot((x,0,3))   # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

This draws a red oscillatory curve:

```
sage: sin(x^2).plot((x,0,2*pi), rgbcolor=(1,0,0))   # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Another plot using the variable theta:

```
sage: var('theta')
theta
sage: (cos(theta) - erf(theta)).plot((theta,-2*pi,2*pi))   # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

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)   # needs sage.plot
```

You can embed 2d plots in 3d space as follows:

```
sage: plot(sin(x^2), (x, -pi, pi), thickness=2).plot3d(z=1)   # long...
```

2.1. Symbolic Expressions 81
A more complicated family:

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

A plot involving the floor function:

```
sage: plot(1.0 - x * floor(1/x), (x,0.00001,1.0))  # needs sage.plot
```

Sage used to allow symbolic functions with “no arguments”; this no longer works:

```
sage: plot(2*sin, -4, 4)  # needs sage.plot
Traceback (most recent call last):
  ... TypeError: unsupported operand parent(s) for *: 'Integer Ring' and 'class...
```

You should evaluate the function first:

```
sage: plot(2*sin(x), -4, 4)  # needs sage.plot
```

**poly** 
(x=\text{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** (\text{base\_ring=\text{None}}, \text{ring=\text{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 (\text{base\_ring}) you want the output polynomial to be over, or you can specify the full polynomial ring (\text{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:**

```python
sage: f = x^2 - 2/3*x + 1
sage: f.polynomial(QQ)
```

```
x^2 - 2/3*x + 1
```

```python
sage: f.polynomial(GF(19))
```

```
x^2 + 12*x + 1
```

Polynomials can be useful for getting the coefficients of an expression:

```python
sage: g = 6*x^2 - 5
sage: g.coefficients()

```

```
[-5, 0], [6, 2]
```

```python
sage: g.polynomial(QQ).list()
```

```
[-5, 0, 6]
```

```python
sage: g.polynomial(QQ).dict()
```

```
{0: -5, 2: 6}
```

```python
sage: f = x^2*e + x + pi/e
sage: f.polynomial(RDF)  # abs tol 5e-16
```

```
2.718281828459045*x^2 + x + 1.1557273497909217
```

```python
sage: g = f.polynomial(RR); g
```

```
2.71828182845905*x^2 + x + 1.15572734979092
```

```python
sage: g.parent()
```

```
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
```

```python
sage: f.polynomial(RealField(100))
```

```
2.7182818284590452353602874714*x^2 + x + 1.1557273497909217179100931833
```

```python
sage: f.polynomial(CDF)  # abs tol 5e-16
```

```
2.718281828459045*x^2 + x + 1.15572734979092
```

```python
sage: f.polynomial(CC)
```

```
2.71828182845905*x^2 + x + 1.15572734979092
```

We coerce a multivariate polynomial with complex symbolic coefficients:

```python
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
```

```python
sage: f.polynomial(CDF)  # abs tol 1e-15
```

```
(-2.718281828459045)*y^2 + 31.006276680299816*x - 1.0*I
```

```python
sage: f.polynomial(ComplexField(70))
```

```
(-2.7182818284590452354)*y^2 + 31.006276680299820175*x - 1.0000000000000000000*I
```

Another polynomial:
Symbolic Calculus, Release 10.3

```python
sage: f = sum((e*I)^n*x^n for n in range(5)); f
x^4*e^4 - I*x^3*e^3 - x^2*e^2 + I*x*e + 1
sage: f.polynomial(CDF)  # abs tol 5e-16
54.598150033144236*x^4 - 20.085536923187668*I*x^3 - 7.38905609893065*x^2
 + 2.718281828459045*I*x + 1.0
sage: f.polynomial(CC)
54.5981500331442*x^4 - 20.0855369231877*I*x^3 - 7.38905609893065*x^2
 + 2.71828182845905*I*x + 1.00000000000000
```

A multivariate polynomial over a finite field:

```python
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:

```python
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”:

```python
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:**

```python
sage: (x^2).power(2)
x^4
sage: (x^2).power(2, hold=True)
(x^2)^2
```

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

```python
sage: a = (x^2).power(2, hold=True); a.unhold()
x^4
```

### power_series (base_ring)

Return algebraic power series associated to this symbolic expression, which must be a polynomial in one variable, with coefficients coercible to the base ring.

The power series is truncated one more than the degree.

**EXAMPLES:**
Symbolic Calculus, Release 10.3

```
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)
```

**prod** *(args, **kwds)*

Return the symbolic product \(\prod_{v=a}^{b}\text{self}\).

This is the product respect to the variable *v* with endpoints *a* and *b*.

**INPUT:**

- **expression** – a symbolic expression
- **v** – a variable or variable name
- **a** – lower endpoint of the product
- **b** – upper endpoint of the product
- **algorithm** – (default: 'maxima') one of
  - 'maxima' – use Maxima (the default)
  - 'giac' – (optional) use Giac
  - 'sympy' – use SymPy
- **hold** – (default: False) if True, don’t evaluate
**pyobject()**

Get the underlying Python object.

**OUTPUT:**

The Python object corresponding to this expression, assuming this expression is a single numerical value or an infinity representable in Python. Otherwise, a **TypeError** is raised.

**EXAMPLES:**

```plaintext
sage: var('x')
x
sage: b = -17.3
sage: a = SR(b)
sage: a.pyobject()
-17.3000000000000
sage: a.pyobject() is b
True
```

Integers and Rationals are converted internally though, so you won’t get back the same object:

```plaintext
sage: b = -17/3
sage: a = SR(b)
sage: a.pyobject()
-17/3
sage: a.pyobject() is b
False
```

**rational_expand**(side=None)

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

**EXAMPLES:**

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

```plaintext
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:

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

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

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

We can expand individual sides of a relation:
```python
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
```

**rational_simplify**(algorithm='full', map=False)

Simplify rational expressions.

**INPUT:**

- **self** - symbolic expression
- **algorithm** - (default: 'full') string which switches the algorithm for simplifications. Possible values are
  - 'simple' (simplify rational functions into quotient of two polynomials),
  - 'full' (apply repeatedly, if necessary)
  - 'noexpand' (convert to common denominator and add)
- **map** - (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:**

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

```python
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 results are shorter for functions which are combination of rational and nonrational functions. In the following example, we use this option if we want not to combine logarithm and the rational function into one fraction:

```python
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:
Symbolic Calculus, Release 10.3

```
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)
0
```

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

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

**real_part** *(hold=False)*

Return the real part of this symbolic expression.

**EXAMPLES:**
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)
0
```

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

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

```
rectform()
```

Convert this symbolic expression to rectangular form; that is, the form \(a + bi\) where \(a\) and \(b\) are real numbers and \(i\) is the imaginary unit.

**Note:** The name "rectangular" comes from the fact that, in the complex plane, \(a\) and \(bi\) are perpendicular.

**INPUT:**

- `self` – the expression to convert.

**OUTPUT:**

A new expression, equivalent to the original, but expressed in the form \(a + bi\).

**ALGORITHM:**

We call Maxima's `rectform()` and return the result unmodified.

**EXAMPLES:**

The exponential form of \(\sin(x)\):

```
sage: f = (e^(I*x) - e^(-I*x)) / (2*I)
sage: f.rectform()
sin(x)
```

---

2.1. Symbolic Expressions 89
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)
```

**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()
-sin(y)^2 + 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)
```

**ALIAS:** trig_reduce() and reduce_trig() are the same

**residue** *(symbol)*

Calculate the residue of *self* with respect to *symbol*.

**INPUT:**

- *symbol* - a symbolic variable or symbolic equality such as \(x == 5\). If an equality is given, the expansion is around the value on the right hand side of the equality, otherwise at 0.

**OUTPUT:**

The residue of *self*. 

---
Say, symbol is \( x = a \), then this function calculates the residue of \( \text{self} \) at \( x = a \), i.e., the coefficient of \( 1/(x - a) \) of the series expansion of \( \text{self} \) around \( a \).

**EXAMPLES:**

```python
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')
sage: zeta(s).residue(s == 1)
1
```

We can also compute the residue at more general places, given that the pole is recognized:

```python
sage: k = var('k', domain='integer')
sage: (gamma(1+x)/(1 - exp(-x))).residue(x==2*I*pi*k)
gamma(2*I*pi*k + 1)
sage: csc(x).residue(x==2*pi*k)
1
```

**resultant (other, var)**

Compute the resultant of this polynomial expression and the first argument with respect to the variable given as the second argument.

**EXAMPLES:**

```python
sage: == var('a b n k u x y')
sage: x.resultant(y, x)
y
sage: (x+y).resultant(x-y, x)
-2*y
sage: r = (x^4*y^2+x^2*y-y).resultant(x*y-y*a-x*b+a*b+u,x)
sage: r.coefficient(a^4)
b^4*y^2 - 4*b^3*y^3 + 6*b^2*y^4 - 4*b*y^5 + y^6
sage: x.resultant(sin(x), x)
Traceback (most recent call last):
...
RuntimeError: resultant(): arguments must be polynomials
```

**rhs ()**

If \( \text{self} \) is a relational expression, return the right hand side of the relation. Otherwise, raise a `ValueError`.

**EXAMPLES:**

```python
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()  # (continues on next page)
```
right()

If `self` is a relational expression, return the right hand side of the relation. Otherwise, raise a `ValueError`.

EXAMPLES:

```python
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:

```python
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:**
A simple example:

```python
sage: ((x^2-1)^2).roots()
[(-1, 2), (1, 2)]
sage: ((x^2-1)^2).roots(multiplicities=False)
[-1, 1]
```

A complicated example:

```python
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
sage: f.roots()
[(x, 1)]
```

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

```python
sage: f.poly(a)
x^9 - 2*x^7 + 2*x^3 - (x^8 - 2*x^6 + 2*x^2 - 1)*a - x
```

As a polynomial in a, x is indeed a root:

```python
sage: f(a=x)
0
```

The roots in terms of x are what we expect:

```python
sage: f.roots(x)
[(a, 1), (-I, 1), (I, 1), (1, 3), (-1, 3)]
```

Only one root of \( \sin(x) = 0 \) is given:

```python
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:

```python
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()`

```python
sage: var('x')
x
sage: f = x^(1/9) + (2^(8/9) - 2^(1/9))*(x - 1) - x^(8/9)
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
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:

```python
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)]
```

Now let us find some roots over different rings:

```python
sage: f.roots(ring=CC)
[(-0.058811522318449..., 1),
 (-1.331099917875796? - 1.522416551837318?*I, 1),
 (-1.331099917875796? + 1.522416551837318?*I, 1),
 (1.360505679035020? - 1.51880872209650?*I, 1),
 (1.360505679035020? + 1.51880872209650?*I, 1)]
```

Root finding over finite fields:

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

`round()`

Round this expression to the nearest integer.

**EXAMPLES:**

```python
sage: u = sqrt(43203735824841025516773866131535024)
sage: u.round()
```

(continues on next page)
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
defines symbols 'x,y'
(x, y)
sage: f = (x^3 - sin(y)*x^2 - 5*x + 3); f
x^3 - x^2*\sin(y) - 5*x + 3
sage: g = f.series(x, 4); g
3 + (-5)*x + (-\sin(y))*x^2 + 1*x^3 + Order(x^4)
sage: g.truncate()
x^3 + x^2*\sin(y) - 5*x + 3
sage: g = f.series(x==1, 4); g
(-\sin(y) - 1) + (-2*\sin(y) - 2)*(x - 1) + (-2*\cos(y) + 5/2*\sin(y))*(x - 1)^2
+ 1*(x - 1)^3 + Order((x - 1)^4)
sage: h = g.truncate(); h
(x - 1)^3 - (x - 1)^2*(\sin(y) - 3) - 2*(x - 1)*(\sin(y) + 1) - \sin(y) - 1
sage: h.expand()
x^3 - x^2*\sin(y) - 5*x + 3
```

We compute another series expansion of an analytic function:

```sage
f = sin(x)/x^2
sage: f.series(x, 7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x)
1*x^(-1) + (-1/6)*x + ... + Order(x^20)
sage: f.series(x==1,3)
(\sin(1)) + (\cos(1) - 2*\sin(1))*(x - 1) + (-2*\cos(1) + 5/2*\sin(1))*(x - 1)^2
+ Order((x - 1)^3)
```

(continues on next page)
Expressions formed by combining series can be expanded by applying series again:

```sage
t = (1/(1-x)).series(x, 3) + (1/(1+x)).series(x, 3)
t = _.series(x, 3)	t = (1/(1-x)).series(x, 3) * (1/(1+x)).series(x, 3)
t = _.series(x, 3)
```

Following the GiNaC tutorial, we use John Machin’s amazing formula \( \pi = 16 \tan^{-1}(1/5) - 4 \tan^{-1}(1/239) \) to compute digits of \( \pi \). We expand the arc tangent around 0 and insert the fractions 1/5 and 1/239.

```sage
x = var('x')
f = atan(x).series(x, 10); f
f = 1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
float(16*f.subs(x==1/5) - 4*f.subs(x==1/239))
3.1415926824043994
```

**show()**

Pretty-print this symbolic expression.

This typesets it nicely and prints it immediately.

**OUTPUT:**

This method does not return anything. Like print, output is sent directly to the screen.

Note that the output depends on the display preferences. For details, see `pretty_print()`.

**EXAMPLES:**

```sage
(x^2 + 1).show()
x^2 + 1
```

**EXAMPLES:**

```sage
%display ascii_art  # not tested
(x^2 + 1).show()
x + 1
```

**simplify(algorithm='maxima', **kwds)**

Return a simplified version of this symbolic expression.

**INPUT:**

- `algorithm` - one of:
  - `maxima`: (default) sends the expression to maxima and converts it back to Sage
  - `sympy`: converts the expression to sympy, simplifies it (passing any optional keyword(s)), and converts the result to Sage
  - `giac`: converts the expression to giac, simplifies it, and converts the result to Sage
– fricas: converts the expression to fricas, simplifies it, and converts the result to Sage

See also:

simplify_full(), simplify_trig(), simplify_rational(), simplify_rectform()
simplify_factorial(), simplify_log(), simplify_real(), simplify_hypergeometric(), canonicalize_radical()

EXAMPLES:

```
sage: a = var('a'); f = x*sin(2)/(x^a); f
x*sin(2)/x^a
sage: f.simplify()
x^(-a + 1)*sin(2)
```

Some simplifications are quite algorithm-specific:

```
sage: x, t = var("x, t")
sage: ex = cos(t).exponentialize()
sage: ex = ex.subs((sin(t).exponentialize()==x).solve(t)[0])
sage: ex.simplify()
I/2*x + I/2*sqrt(x^2 - 1) + 1/(2*I*x + 2*I*sqrt(x^2 - 1))
sage: ex.simplify(algorithm="sympy")
I*(x^2 + sqrt(x^2 - 1)*x - 1)/(x + sqrt(x^2 - 1))
sage: ex.simplify(algorithm="giac")
I*sqrt(x^2 - 1)
sage: ex.simplify(algorithm="fricas")  # optional - fricas
(I*x^2 + I*sqrt(x^2 - 1)*x - I)/(x + sqrt(x^2 - 1))
```

**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: a = var('a'); f = factorial(n)*n!/factorial(n); 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()
binomial(n, k)*factorial(k)*factorial(-k + 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

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)

simplify_hypergeometric(algorithm='maxima')

Simplify an expression containing hypergeometric or confluent hypergeometric functions.

INPUT:

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

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

EXAMPLES:

sage: hypergeometric((5, 4), (4, 1, 2, 3), ...
....: x).simplify_hypergeometric()
1/144*x^2*hypergeometric([], (3, 4), x) +...
1/3*x*hypergeometric([], (2, 3), x) + hypergeometric([], (1, 2), x)
sage: (2*hypergeometric([], [], x)).simplify_hypergeometric()
2*e^x

sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)
....: # not tested, → unstable
....: .simplify_hypergeometric())

laguerre(-laguerre(-e^x, x), x)
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)
....: # not tested, → unstable
....: .simplify_hypergeometric(algorithm='sage'))

hypergeometric(hypergeometric(e^x,), (1,), x), (1,), x)
sage: hypergeometric_M(1, 3, x).simplify_hypergeometric()
-2*(x + 1)*e^(-x) - 1)*e^x/x^2

(continues on next page)
\[ \texttt{sage: } (2 \times \text{hypergeometric}_U(1, 3, x)).\text{simplify}\text{\_hypergeometric}() \\
\quad \frac{2(x + 1)}{x^2} \]

**simplify\_log** *(algorithm=None)*

Simplify a (real) symbolic expression that contains logarithms.

The given expression is scanned recursively, transforming subexpressions of the form \(a \log(b) + c \log(d)\) into \(\log(b^a d^c)\) before simplifying within the \(\log()\).

The user can specify conditions that \(a\) and \(c\) must satisfy before this transformation will be performed using the optional parameter **algorithm**.

**Warning:** This is only safe to call if every variable in the given expression is assumed to be real. The simplification it performs is in general not valid over the complex numbers. For example:

```
\texttt{sage: } x, y = \text{SR}.\text{var}('x, y') \\
\texttt{sage: } f = \log(x*y) - (\log(x) + \log(y)) \\
\texttt{sage: } f(\text{x=-1, y=i}) \\
\quad -2*I*pi \\
\texttt{sage: } f.\text{simplify}\_\text{log}() \\
\quad 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 `\text{logcontract()}` command.

**ALIAS:**

`\text{log\_simplify()}` and `\text{simplify\_log()}` are the same.

**EXAMPLES:**

```
\texttt{sage: } x, y, t = \text{var}('x y t')
```

Only two first terms are contracted in the following example; the logarithm with coefficient \(\frac{1}{2}\) is not contracted:

```
\texttt{sage: } f = \log(x)+2*\log(y)+1/2*\log(t) \\
\texttt{sage: } f.\text{simplify}\_\text{log}() \\
\quad \log(x*y^2) + 1/2*\log(t)
```

To contract all terms in the previous example, we use the `'ratios'` algorithm:
To contract terms with no coefficient (more precisely, with coefficients 1 and $-1$), we use the 'one' algorithm:

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

\[ \pi \] is an irrational number; to contract logarithms in the following example we have to set algorithm to 'constants' or 'all':

```python
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':

```python
sage: (x*log(9)).simplify_log('all')
log(3^(2*x))
```

AUTHORS:

- Robert Marik (11-2009)

\texttt{simplify\_rational}\texttt{(algorithm='full', map=False)}

Simplify rational expressions.

INPUT:

- self - symbolic expression
- algorithm - (default: 'full') string which switches the algorithm for simplifications. Possible values are
  - 'simple' (simplify rational functions into quotient of two polynomials),
  - 'full' (apply repeatedly, if necessary)
  - 'noexpand' (convert to common denominator and add)
- map - (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: \texttt{rational\_simplify()} and \texttt{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:
Symbolic Calculus, Release 10.3

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

```sage
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 results are shorter for functions which are combination of rational and nonrational functions. In the following example, we use this option if we want not to combine logarithm and the rational function into one fraction:

```sage
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
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
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
sage: f = sqrt(x^2)
sage: f.simplify_real()
abs(x)
```

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

`simplify_rectform(complexity_measure='string_length')`

Attempt to simplify this expression by expressing it in the form \(a + bi\) where both \(a\) and \(b\) are real. This transformation is generally not a simplification, so we use the given `complexity_measure` to discard non-simplifications.

**INPUT:**

- `self` – the expression to simplify.
- `complexity_measure` – (default: `sage.symbolic.complexity_measures.string_length`) a function taking a symbolic expression as an argument and returning a measure of that expression's 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 \(\tan(x)\):

```python
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:

```python
sage: f = e^(I*x)
sage: f.simplify_rectform()
e^(I*x)
```

However, if we pass `None` as our complexity measure, it is:

```python
sage: f = e^(I*x)
sage: f.simplify_rectform(complexity_measure = None)
cos(x) + I*sin(x)
```

`simplify_trig(expand=True)`

Optionally expand and then employ identities such as \(\sin(x)^2 + \cos(x)^2 = 1\), \(\cosh(x)^2 - \sinh(x)^2 = 1\), \(\sin(x) \csc(x) = 1\), or \(\tanh(x) = \sinh(x) / \cosh(x)\) to simplify expressions containing \(\tan, \sec, \) etc., to \(\sin, \cos, \sinh, \cosh\).

**INPUT:**

- `self` - symbolic expression
• expand - (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:

```python
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)
1
sage: k = tanh(x)*cosh(2*x)
(2*sinh(x)^3 + sinh(x))/cosh(x)
```

In some cases we do not want to expand:

```python
sage: f = tan(3*x)
sage: f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
sage: f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

\texttt{sin(hold=False)}

EXAMPLES:

```python
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(RR(1)))
0.84147098480789650665250232163029899962256306
```

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

```python
sage: SR(0).sin()
0
sage: SR(0).sin(hold=True)
sin(0)
```

This also works using functional notation:

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

To then evaluate again, we use \texttt{unhold()}:
\texttt{sage: a = SR(0).sin(hold=True); a.unhold()}
0

\texttt{sage: SR(0).sin(hold=False)}
\texttt{sinh(0)}

\texttt{Return sinh of self.}

We have \( \sinh(x) = \frac{e^x - e^{-x}}{2} \).

\textbf{EXAMPLES:}

\texttt{sage: x.sinh()}
\texttt{sinh(x)}
\texttt{sage: SR(1).sinh()}
\texttt{sinh(1)}
\texttt{sage: SR(0).sinh()}
\texttt{0}
\texttt{sage: SR(1.0).sinh()}
\texttt{1.17520119364380}
\texttt{sage: maxima('sinh(1.0)')}
\texttt{1.17520119364380...}

\texttt{sinh(1.0000000000000000000)}
\texttt{sage: SR(1).sinh()}.n(90)
\texttt{1.1752011936438014568823819}
\texttt{sage: SR(RIF(1)).sinh()}
\texttt{1.175201193643802?}

To prevent automatic evaluation use the \texttt{hold} argument:

\texttt{sage: arccosh(x).sinh()}
\texttt{sqrt(x + 1)*sqrt(x - 1)}
\texttt{sage: arccosh(x).sinh(hold=True)}
\texttt{sinh(arccosh(x))}

This also works using functional notation:

\texttt{sage: sinh(arccosh(x),hold=True)}
\texttt{sinh(arccosh(x))}
\texttt{sage: sinh(arccosh(x))}
\texttt{sqrt(x + 1)*sqrt(x - 1)}

To then evaluate again, we use \texttt{unhold()}:  

\texttt{sage: a = arccosh(x).sinh(hold=True); a.simplify()}
\texttt{sqrt(x + 1)*sqrt(x - 1)}

\texttt{solve}(x, multiplicities=False, solution_dict=False, explicit_solutions=False, to_poly_solve=False, algorithm=None, domain=None)

Analytically solve the equation \texttt{self == 0} or a univariate inequality for the variable \(x\).

\textbf{Warning:}  This is not a numerical solver – use \texttt{find_root()} to solve for \texttt{self == 0} numerically on an interval.

\textbf{INPUT:}

- \(x\) – variable(s) to solve for
Symbolic Calculus, Release 10.3

• **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:**

```python
sage: z = var('z')
sage: (z^5 - 1).solve(z)
[z == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4, 
  z == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4, 
  z == -1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4, 
  z == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4, 
  z == 1]
sage: solve((z^3-1)^3, z, multiplicities=True)
([z == 1/2*I*sqrt(3) - 1/2, z == -1/2*I*sqrt(3) - 1/2, z == 1], [3, 3, 3])
```

**solve_diophantine** (x=None, solution_dict=False)

Solve a polynomial equation in the integers (a so-called Diophantine).

If the argument is just a polynomial expression, equate to zero. If **solution_dict**=True return a list of dictionaries instead of a list of tuples.

**EXAMPLES:**

```python
sage: x, y = var('x,y')
sage: solve_diophantine(3*x == 4)  # needs sympy
[]
sage: solve_diophantine(x^2 - 9)  # needs sympy
([-3, 3])
sage: sorted(solve_diophantine(x^2 + y^2 == 25))  # needs sympy
[(-5, 0), (-4, -3), (-4, 3), (-3, -4), (-3, 4), (0, -5)...]
```

The function is used when **solve()** is called with all variables assumed integer:

```python
sage: assume(x, 'integer')
sage: assume(y, 'integer')
sage: sorted(solve(x^2*y == 1, (x,y)))  # needs sympy
[(-1, -1), (1, 1)]
```

You can also pick specific variables, and get the solution as a dictionary:

2.1. Symbolic Expressions

105
Symbolic Calculus, Release 10.3

```python
sage: # needs sympy
sage: solve_diophantine(x*y == 10, x)
[-10, -5, -2, -1, 1, 2, 5, 10]
sage: sorted(solve_diophantine(x*y - y == 10, (x,y)))
[(-9, -1), (-4, -2), (-1, -5), (0, -10), (2, 10), (3, 5), (6, 2), (11, 1)]
sage: res = solve_diophantine(x*y - y == 10, solution_dict=True)
sage: sol = [{y: -5, x: -1}, {y: -10, x: 0}, {y: -1, x: -9}, {y: -2, x: -4},
......: {y: 10, x: 2}, {y: 1, x: 11}, {y: 2, x: 6}, {y: 5, x: 3}]
sage: all(solution in res
.....: for solution in sol) and bool(len(res) == len(sol))
True
```

If the solution is parametrized the parameter(s) are not defined, but you can substitute them with specific integer values:

```python
sage: # needs sympy
sage: x,y,z = var('x,y,z')
sage: sol = solve_diophantine(x^2-y == 0); sol
(t, t^2)
sage: [(sol[0].subs(t=t),sol[1].subs(t=t)) for t in range(-3,4)]
[(-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), (3, 9)]
sage: sol = solve_diophantine(x^2 + y^2 == z^2); sol
(2*p*q, p^2 - q^2, p^2 + q^2)
sage: [(sol[0].subs(p=p,q=q), sol[1].subs(p=p,q=q), sol[2].subs(p=p,q=q))
.....: for p in range(1,4) for q in range(1,4)]
[(2, 0, 2), (4, -3, 5), (6, -8, 10), (4, 3, 5), (8, 0, 8),
(12, -5, 13), (6, 8, 10), (12, 5, 13), (18, 0, 18)]
```

Solve Brahmagupta-Pell equations:

```python
sage: sol = sorted(solve_diophantine(x^2 - 2*y^2 == 1), key=str); sol
#˓→needs sympy
[(-sqrt(2)*(2*sqrt(2) + 3)^t + sqrt(2)*(-2*sqrt(2) + 3)^t
˓→- 3/2*(2*sqrt(2) + 3)^t - 3/2*(-2*sqrt(2) + 3)^t,...
sage: [(sol[1][0].subs(t=t).simplify_full(),
˓→needs sympy
.....: sol[1][1].subs(t=t).simplify_full()) for t in range(-1,5)]
[(1, 0), (3, -2), (17, -12), (99, -70), (577, -408), (3363, -2378)]
```

See also:

http://docs.sympy.org/latest/modules/solvers/diophantine.html

`sqrt` *(hold=False)*

Return the square root of this expression

**EXAMPLES:**

```python
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)
```

Immediate simplifications are applied:
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 use `unhold()`:

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

To use this parameter in functional notation, you must coerce to the symbolic ring:

```
sage: sqrt(SR(4),hold=True)
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 unit step function, which is 0 for negative x, 1 for 0, and 1 for positive x.

See also:

`sage.functions.generalized.FunctionUnitStep`

EXAMPLES:

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

Using the \texttt{hold} parameter it is possible to prevent automatic evaluation:

sage: SR(2).step()
1
sage: SR(2).step(hold=True)
unit_step(2)

\textbf{subs} (*\texttt{args}, **\texttt{kwds})

Substitute the given subexpressions in this expression.

\textbf{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
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2
sage: t.subs(b=19, x=z)
(y + z)^3 + a^2 + 361
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
sage: t.subs(w0^2 == w0^3)
a^3 + b^3 + (x + y)^3
sage: t.subs(w0 == w0^2)
a^8 + b^8 + (x^2 + y^2)^6
sage: t.subs(a == b, b == c)
(x + y)^3 + b^2 + c^2

Any number of arguments is accepted:

sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2
sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2
sage: t.subs([x == 3, y == 2], a == 2, {b:3})
138

It can even accept lists of lists:

sage: eqn1 = (a*x + b*y == 0)
sage: eqn2 = (1 + y == 0)
sage: soln = solve([eqn1, eqn2], [x, y])
sage: soln
[[x == b/a, y == -1]]
sage: f = x + y
sage: f.subs(soln)
b/a - 1

Duplicate assignments will throw an error:

sage: t.subs({a:b}, a=c)
Traceback (most recent call last):
... ValueError: duplicate substitution for a, got values b and c
sage: t.subs([x == 1], a = 1, b = 2, x = 2)
Traceback (most recent call last):
... ValueError: duplicate substitution for x, got values 1 and 2

All substitutions are performed at the same time:

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

Substitutions are done term by term, in other words Sage is not able to identify partial sums in a substitution (see github issue #18396):

sage: f = x + x^2 + x^4
sage: f.subs(x = y)
y^4 + y^2 + y
sage: f.subs(x^2 == y)  # one term is fine
x^4 + x + y
sage: f.subs(x + x^2 == y)  # partial sum does not work
x^4 + x^2 + x
sage: f.subs(x + x^2 + x^4 == y)  # whole sum is fine
y

Note that it is the very same behavior as in Maxima:

sage: E = 'x^4 + x^2 + x'
sage: subs = [('{x}', '{y}'), ('{x^2}', '{y}'), ('{x^2+x}', '{y}'), ('{x^4+x^2+x}', '{y}')]  
sage: cmd = '{}, {}={}'.
sage: for s1, s2 in subs:
    ...:     maxima.eval(cmd.format(E, s1, s2))
'y^4+y^2+y'
y+x^4+x'

2.1. Symbolic Expressions
Symbolic Calculus, Release 10.3

(continued from previous page)

\[ x^4 + x^2 + x \]
\[ y \]

Or as in Maple:

```python
sage: cmd = 'subs({}, {}, {})'  # optional - maple
sage: for s1, s2 in subs:  # optional - maple
    maple.eval(cmd.format(s1, s2, E))

'y' + y^2 + y
'x' + y
'x' + x^2 + x
'y'
```

But Mathematica does something different on the third example:

```python
sage: cmd = '{}/. {} -> {}'  # optional - mathematica
sage: for s1, s2 in subs:  # optional - mathematica
    mathematica.eval(cmd.format(E, s1, s2))

\[ 2 + 4y + 4y^2 + 4x + 4x^4 + y \]

The same, with formatting more suitable for cut and paste:

```python
sage: for s1, s2 in subs:  # optional - mathematica
    mathematica(cmd.format(E, s1, s2))

y + y^2 + y^4
x + x^4 + y
x^4 + y
y
```

Warning: Unexpected results may occur if the left-hand side of some substitution is not just a single variable (or is a “wildcard” variable). For example, the result of `cos(cos(cos(x)))`. `subs({cos(x) : x})` is `x`, because the substitution is applied repeatedly. Such repeated substitutions (and pattern-matching code that may be somewhat unpredictable) are disabled only in the basic case where the left-hand side of every substitution is a variable. In particular, although the result of `(x^2).subs({x : sqrt(x)})` is `x`, the result of `(x^2).subs({x : sqrt(x), y^2 : y})` is `sqrt(x)`, because repeated substitution is enabled by the presence of the expression `y^2` in the left-hand side of one of the substitutions, even though that particular substitution does not get applied.

```python
substitute_function(*args, **kwds)
```

Substitute the given functions by their replacements in this expression.

EXAMPLES:

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

Substitute with a dictionary:
Symbolic Calculus, Release 10.3

```
sage: f.substitute_function({foo: bar})
1/bar(pi*y) + bar(x)
sage: f.substitute_function({foo(x): bar(x)})
1/bar(pi*y) + bar(x)
```

If the function expression to be substituted includes its arguments, the right hand side can be an arbitrary symbolic expression:

```
sage: f.substitute_function({foo(x): x^2})
x^2 + 1/(pi^2*y^2)
```

Substitute with keyword arguments (works only if no function arguments are given):

```
sage: f.substitute_function(foo=bar)
1/bar(pi*y) + bar(x)
```

Substitute with a relational expression:

```
sage: f.substitute_function(foo(x)==bar(x))
1/bar(pi*y) + bar(x)
sage: f.substitute_function(foo(x)==bar(x+1))
1/bar(pi*y + 1) + bar(x + 1)
```

All substitutions are performed at the same time:

```
sage: g = foo(x) + 1/bar(pi*y)
sage: g.substitute_function((foo: bar, bar: foo))
1/foo(pi*y) + bar(x)
```

Any number of arguments is accepted:

```
sage: g.substitute_function({foo: bar}, bar(x) == x^2)
1/(pi^2*y^2) + bar(x)
```

As well as lists of substitutions:

```
sage: g.substitute_function([foo(x) == 1, bar(x) == x])
1/(pi*y) + 1
```

Alternative syntax:

```
sage: g.substitute_function(foo, bar)
1/bar(pi*y) + bar(x)
```

Duplicate assignments will throw an error:

```
sage: g.substitute_function((foo:bar), foo(x) == x^2)
Traceback (most recent call last):
  ...
ValueError: duplicate substitution for foo, got values bar and x |--> x^2
```

```
sage: g.substitute_function([foo(x) == x^2, foo = bar])
Traceback (most recent call last):
  ...
ValueError: duplicate substitution for foo, got values x |--> x^2 and bar
```

2.1. Symbolic Expressions 111
**substitution_delayed** (*pattern, replacement*)

Replace all occurrences of pattern by the result of replacement.

In contrast to `subs()`, the pattern may contain wildcards and the replacement can depend on the particular term matched by the pattern.

**INPUT:**

- `pattern` – an `Expression`, usually containing wildcards.
- `replacement` – a function. Its argument is a dictionary mapping the wildcard occurring in `pattern` to the actual values. If it returns `None`, this occurrence of `pattern` is not replaced. Otherwise, it is replaced by the output of `replacement`.

**OUTPUT:**

An `Expression`.

**EXAMPLES:**

```sage
sage: var('x y')
(x, y)
sage: w0 = SR.wild(0)
sage: sqrt(1 + 2*x + x^2).substitution_delayed(
    ....:     sqrt(w0),
    ....:     lambda d: sqrt(factor(d[w0])))
sqrt((x + 1)^2)
sage: def r(d):
    ....:     if x not in d[w0].variables():
    ....:         return cos(d[w0])
sage: (sin(x^2 + x) + sin(y^2 + y)).substitution_delayed(sin(w0), r)
cos(y^2 + y) + sin(x^2 + x)
```

See also:

`match()`

**subtract_from_both_sides** (*x*)

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

**EXAMPLES:**

```sage
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 ∑_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
Symbolic Calculus, Release 10.3

- 'maxima' – use Maxima (the default)
- 'maple' – (optional) use Maple
- 'mathematica' – (optional) use Mathematica
- 'giac' – (optional) use Giac
- 'sympy' – use SymPy

EXAMPLES:

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

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

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

A well known binomial identity:

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

And some truncations thereof:

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

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

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

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

The binomial theorem:

```sage
sage: x, y = var('x, y')
sage: (binomial(n,k) * x^k * y^(n-k)).sum(k, 0, n)
(x + y)^n
```

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

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

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

Summing a hypergeometric term:

```sage
sage: (binomial(n, k) * factorial(k) / factorial(n+1+k)).sum(k, 0, n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

---

2.1. Symbolic Expressions 113
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')
```

pi*coth(pi)

Use Giac to perform this summation:

```sage
(sum(1/(1+k^2), k, -oo, oo, algorithm = 'giac')).factor()
```

pi*(e^(2*pi) + 1)/((e^pi + 1)*(e^pi - 1))

Use Maple as a backend for summation:

```sage
(binomial(n,k)*x^k).sum(k, 0, n, algorithm = 'maple')
```

(x + 1)^n

Note:

1. Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertible into a usable Sage expression.

```
\tan(\text{hold} = \text{False})
```

EXAMPLES:

```sage
var('x, y')
(sage: tan(x^2 + y^2)
tan(x^2 + y^2)
```

(continues on next page)
To prevent automatic evaluation use the `hold` argument:

```python
sage: (pi/12).tan()
tan(1/12*pi)
sage: (pi/12).tan(hold=True)
tan(1/12*pi)
```

This also works using functional notation:

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

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

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

### `tanh` (`hold=False`)

Return tanh of self.

We have \( \tanh(x) = \frac{\sinh(x)}{\cosh(x)} \).

**EXAMPLES:**

```python
sage: x.tanh()
tanh(x)
sage: SR(1).tanh()
tanh(1)
sage: SR(0).tanh()
0
sage: SR(1.0).tanh()
0.76159415555765
sage: maxima('tanh(1.0)')
0.761594155557649
sage: plot(lambda x: SR(x).tanh(), -1, 1)  # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:
Symbolic Calculus, Release 10.3

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

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

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

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)
-15377/7983360*x^4 - 6767/604800*x^2 + 11/120/x^2 + 1/2/x^4 - 1/x^6 - 347/15120
```

test_relation (ntests=20, domain=None, proof=True)

Test this relation at several random values, attempting to find a contradiction. If this relation has no variables, it will also test this relation after casting into the domain.

Because the interval fields never return false positives, we can be assured that if True or False is returned (and proof is False) then the answer is correct.

**INPUT:**
• **ntests** – (default 20) the number of iterations to run
  • **domain** – (optional) the domain from which to draw the random values defaults to CIF for equality testing and RIF for order testing
  • **proof** – (default True) if False and the domain is an interval field, regard overlapping (potentially equal) intervals as equal, and return True if all tests succeeded.

**OUTPUT:**

**Boolean or NotImplemented, meaning**

• **True** – this relation holds in the domain and has no variables.
  • **False** – a contradiction was found.
  • **NotImplemented** – no contradiction found.

**EXAMPLES:**

```sage
sage: (3 < pi).test_relation()
True
sage: (0 >= pi).test_relation()
False
sage: (exp(pi) - pi).n()
19.990999791895
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
```

**to_gamma()**

Convert factorial, binomial, and Pochhammer symbol expressions to their gamma function equivalents.

**EXAMPLES:**

```sage
sage: m,n = var('m n', domain='integer')
```

```sage
sage: factorial(n).to_gamma()
gamma(n + 1)
sage: binomial(m,n).to_gamma()
gamma(m + 1)/(gamma(m - n + 1)*gamma(n + 1))
```

**trailing_coeff(s)**

Return the trailing coefficient of s in self, i.e., the coefficient of the smallest power of s in self.

**EXAMPLES:**

```sage
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)
```

(continues on next page)
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
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
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()
-2*sin(2*x)*sin(y) - 2*cos(2*x)*cos(y)
```

We illustrate various options to this function:

```sage
sage: f = sin(sin(3*cos(2*x))*x)
sage: f.expand_trig()  # default
sin(3*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(2*x))^2*cos(sin(x)^2)
  + sin(cos(x)^2)*sin(sin(x)^2))^2*(cos(sin(x)^2)*sin(cos(x)^2)```

(continues on next page)
If the expression contains terms which are factored, we expand first:

\[
\begin{align*}
\text{sage:} & \quad \cos((k_1-k_2)x).\text{expand().expand_trig()} \\
\text{sage:} & \quad \cos(k_1x)\cos(k_2x) + \sin(k_1x)\sin(k_2x)
\end{align*}
\]

**ALIAS:**
- `trig_expand()` and `expand_trig()` are the same
- `trig_reduce()` (\texttt{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:**

\[
\begin{align*}
\text{sage:} & \quad y = \text{var('y')}
\text{sage:} & \quad f = \sin(x)\cos(x)^3+\sin(y)^2
\text{sage:} & \quad f.\text{reduce_trig()}
\text{sage:} & \quad -1/2\cos(2y) + 1/8\sin(4x) + 1/4\sin(2x) + 1/2
\end{align*}
\]

To reduce only the expressions involving \(x\) we use optional parameter:

\[
\begin{align*}
\text{sage:} & \quad f.\text{reduce_trig(x)}
\text{sage:} & \quad \sin(y)^2 + 1/8\sin(4x) + 1/4\sin(2x)
\end{align*}
\]

**ALIAS:** `trig_reduce()` and `reduce_trig()` are the same

**trig_simplify** (\texttt{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:**

```python
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:

```python
sage: f = tan(3*x)
sage: f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
sage: f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

**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:**

```python
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().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
˓→2*sin(1)
```

**unhold(exclude=None)**

Evaluates any held operations (with the `hold` keyword) in the expression.

**INPUT:**

• **self** – an expression with held operations
**exclude** — (default: None) a list of operators to exclude from evaluation. Excluding arithmetic operators does not yet work (see github issue #10169).

**OUTPUT:**

A new expression with held operations, except those in `exclude`, evaluated

**EXAMPLES:**

```sage
sage: a = exp(I * pi, hold=True)
sage: a
\text{e}^{(\text{i}\pi)}
sage: a.unhold()
-1
sage: b = x.add(x, hold=True)
sage: b
x + x
sage: b.unhold()
2x
sage: (a + b).unhold()
2x - 1
sage: c = (x.mul(x, hold=True)).add(x.mul(x, hold=True), hold=True)
sage: c
x^2 + x^2
sage: c.unhold()
2x^2
sage: sin(tan(0, hold=True), hold=True).unhold() 0
sage: sin(tan(0, hold=True), hold=True).unhold(exclude=[sin])
\sin(0)
sage: (e^sgn(0, hold=True)).unhold() 1
sage: (e^sgn(0, hold=True)).unhold(exclude=[exp])
e^0
sage: log(3).unhold()
\log(3)
```

**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
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
```

(continues on next page)
Symbolic Calculus, Release 10.3

```python
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:**

```python
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:**

```python
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:**

```python
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
```

(continues on next page)
To prevent automatic evaluation use the `hold` argument:

```python
sage: SR(2).zeta(hold=True)
zeta(2)
```

This also works using functional notation:

```python
sage: zeta(2, hold=True)
zeta(2)
sage: zeta(2)
1/6*pi^2
```

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

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

---

```python
class sage.symbolic.expression.ExpressionIterator
Bases: object
class sage.symbolic.expression.OperandsWrapper
Bases: sage.symbolic.expression.SageObject

Operands wrapper for symbolic expressions.

EXAMPLES:

```python
sage: x, y, z = var('x, y, z')
sage: e = x + x*y + z*y + 3*y*z; e
x*y + 3*y*z + x + z*y
sage: e.op[1]
3*y*z
sage: e.op[1,1]
z
sage: e.op[-1]
z^y
sage: e.op[1:]
[3*y*z, x, z^y]
sage: e.op[2]
x*y, 3*y*z
sage: e.op[-2:]
[x, z^y]
sage: e.op[1:-2]
x*y, 3*y*z
sage: e.op[-5]
Traceback (most recent call last):
  ... IndexError: operand index out of range, got -5, expect between -4 and 3
sage: e.op[5]
Traceback (most recent call last):
  ... IndexError: operand index out of range, got 5, expect between -4 and 3
sage: e.op[1,1,0]
Traceback (most recent call last):
  ... IndexError: operand index out of range, got -5, expect between -4 and 3
```

(continues on next page)
class sage.symbolic.expression.PynacConstant

Bases: object

expression()

Returns this constant as an Expression.

EXAMPLES:

    sage: from sage.symbolic.expression import PynacConstant
    sage: f = PynacConstant('foo', 'foo', 'real')
    sage: f + 2
    Traceback (most recent call last):
    ...TypeError: unsupported operand parent(s) for +: 'class sage.symbolic.expression.PynacConstant' and 'Integer Ring'

    sage: foo = f.expression(); foo
    foo
    sage: foo + 2
    foo + 2

name()

Returns the name of this constant.

EXAMPLES:

    sage: from sage.symbolic.expression import PynacConstant
    sage: f = PynacConstant('foo', 'foo', 'real')
    sage: f.name()
    'foo'

serial()

Returns the underlying Pynac serial for this constant.

EXAMPLES:

    sage: from sage.symbolic.expression import PynacConstant
    sage: f = PynacConstant('foo', 'foo', 'real')
    sage: f.serial()  #random
    15

class sage.symbolic.expression.SubstitutionMap

Bases: SageObject
apply_to(expr, options)

Apply the substitution to a symbolic expression

EXAMPLES:

```
sage: from sage.symbolic.expression import make_map
sage: subs = make_map({x:x+1})
sage: subs.apply_to(x^2, 0)
(x + 1)^2
```

class sage.symbolic.expression.SymbolicSeries

Bases: Expression

Trivial constructor.

EXAMPLES:

```
sage: loads(dumps((x+x^3).series(x,2)))
1*x + Order(x^2)
```

coefficients(x=None, sparse=True)

Return the coefficients of this symbolic series as a list of pairs.

INPUT:

- `x` – optional variable.
- `sparse` – Boolean. If False return a list with as much entries as the order of the series.

OUTPUT:

Depending on the value of `sparse`,

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

EXAMPLES:

```
sage: s = (1/(1-x)).series(x,6); s
1 + 1*x + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^5 + Order(x^6)
sage: s.coefficients()
[[1, 0], [1, 1], [1, 2], [1, 3], [1, 4], [1, 5]]
sage: s.coefficients(x, sparse=False)
[1, 1, 1, 1, 1, 1]
sage: x,y = var("x,y")
sage: s = (1/(1-y*x-x)).series(x,3); s
1 + (y + 1)*x + ((y + 1)^2)*x^2 + Order(x^3)
sage: s.coefficients(x, sparse=False)
[1, y + 1, (y + 1)^2]
```

default_variable()

Return the expansion variable of this symbolic series.

EXAMPLES:
Symbolic Calculus, Release 10.3

```
sage: s = (1/(1-x)).series(x,3); s
1 + 1*x + 1*x^2 + Order(x^3)
sage: s.default_variable()
x
```

**is_terminating_series()**

Return True if the series is without order term.

A series is terminating if it can be represented exactly, without requiring an order term. You can explicitly request terminating series by setting the order to positive infinity.

**OUTPUT:**

Boolean. True if the series has no order term.

**EXAMPLES:**

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

**power_series**(base_ring)

Return the algebraic power series associated to this symbolic series.

The coefficients must be coercible to the base ring.

**EXAMPLES:**

```
sage: ex = (gamma(1-x)).series(x,3); ex
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + Order(x^3)
sage: g = ex.power_series(SR); g
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + O(x^3)
sage: g.parent()
Power Series Ring in x over Symbolic Ring
```

**truncate()**

Given a power series or expression, return the corresponding expression without the big oh.

**OUTPUT:**

A symbolic expression.

**EXAMPLES:**

```
sage: f = sin(x)/x^2
sage: f.truncate()
sin(x)/x^2
sage: f.series(x,7).truncate()
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
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)
```

126 Chapter 2. Internal functionality supporting calculus
sage.symbolic.expression.call_registered_function(serial, nargs, args, hold, allow_numeric_result, result_parent)

Call a function registered with Pynac (GiNaC).

INPUT:

- `serial` - serial number of the function
- `nargs` - declared number of args (0 is variadic)
- `args` - a list of arguments to pass to the function; each must be an `Expression`
- `hold` - whether to leave the call unevaluated
- `allow_numeric_result` - if True, keep numeric results numeric; if False, make all results symbolic expressions
- `result_parent` - an instance of `SymbolicRing`

EXAMPLES:

```
sage: from sage.symbolic.expression import find_registered_function, call_registered_function
sage: s_arctan = find_registered_function('arctan', 1)
sage: call_registered_function(s_arctan, 1, [SR(1)], False, True, SR)
1/4*pi
sage: call_registered_function(s_arctan, 1, [SR(1)], True, True, SR)
apartan(1)
sage: call_registered_function(s_arctan, 1, [SR(0)], False, True, SR)
0
sage: call_registered_function(s_arctan, 1, [SR(0)], False, False, SR).parent()
Integer Ring
sage: call_registered_function(s_arctan, 1, [SR(0)], False, False, SR).parent()
Symbolic Ring
```

sage.symbolic.expression.doublefactorial(n)

The double factorial combinatorial function:

\[ n!! = n \times (n-2) \times (n-4) \times \ldots \times (1 \text{ or } 2) \] with \( 0!! = (-1)!! = 1 \).

INPUT:

- `n` - an integer \( \geq 1 \)

EXAMPLES:

```
sage: from sage.symbolic.expression import doublefactorial
sage: doublefactorial(-1)
1
sage: doublefactorial(0)
1
sage: doublefactorial(1)
1
sage: doublefactorial(5)
15
sage: doublefactorial(20)
3715891200
sage: prod([20,18,..,2])
3715891200
```
sage.symbolic.expression.find_registered_function(name, nargs)

Look up a function registered with Pynac (GiNaC).

Raise a ValueError if the function is not registered.

OUTPUT:

• serial number of the function, for use in call_registered_function()

EXAMPLES:

```
sage: from sage.symbolic.expression import find_registered_function
sage: find_registered_function('arctan', 1)  # random
19
sage: find_registered_function('archenemy', 1)
Traceback (most recent call last):
  ... ValueError: cannot find GiNaC function with name archenemy and 1 arguments
```

sage.symbolic.expression.get_fn_serial()

Return the overall size of the Pynac function registry which corresponds to the last serial value plus one.

EXAMPLES:

```
sage: from sage.symbolic.expression import get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: get_fn_serial() > 125
True
sage: print(get_sfunction_from_serial(get_fn_serial()))
None
sage: get_sfunction_from_serial(get_fn_serial() - 1) is not None
True
```

sage.symbolic.expression.get_ginac_serial()

Number of C++ level functions defined by GiNaC. (Defined mainly for testing.)

EXAMPLES:

```
sage: sage.symbolic.expression.get_ginac_serial() >= 35
True
```

sage.symbolic.expression.get_sfunction_from_hash(myhash)

Return an already created SymbolicFunction given the hash.

EXAMPLES:

```
sage: from sage.symbolic.expression import get_sfunction_from_hash
sage: get_sfunction_from_hash(1)  # random
```

sage.symbolic.expression.get_sfunction_from_serial(serial)

Return an already created SymbolicFunction given the serial.

These are stored in the dictionary sfunction_serial_dict.

EXAMPLES:

```
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: get_sfunction_from_serial(65)  # random
```
class sage.symbolic.expression.hold_class

Bases: object

Instances of this class can be used with Python with.

EXAMPLES:

```
sage: with hold:
....:  tan(1/12*pi)
....:  tan(1/12*pi)
sage: tan(1/12*pi)
-sqrt(3) + 2
sage: with hold:
....:  2^5
....:  32
sage: with hold:
....:  SR(2)^5
....:  2^5
sage: with hold:
....:  t=tan(1/12*pi)
....:  t
sage: t
atan(1/12*pi)
sage: t.unhold()
-sqrt(3) + 2
```

start()

Start a hold context.

EXAMPLES:

```
sage: hold.start()
sage: SR(2)^5
2^5
sage: hold.stop()
sage: SR(2)^5
32
```

stop()

Stop any hold context.

EXAMPLES:

```
sage: hold.start()
sage: SR(2)^5
2^5
sage: hold.stop()
sage: SR(2)^5
32
```

sage.symbolic.expression.init_function_table()

Initializes the function pointer table in Pynac. This must be called before Pynac is used; otherwise, there will be segfaults.

sage.symbolic.expression.init_pynac_I()

Initialize the numeric I object in pynac. We use the generator of QQ(i).

2.1. Symbolic Expressions
EXAMPLES:

```
sage: from sage.symbolic.constants import I as symbolic_I
sage: symbolic_I
I
sage: symbolic_I^2
-1
```

Note that conversions to real fields will give `TypeError`:

```
sage: float(symbolic_I)
Traceback (most recent call last):
...  
TypeError: unable to simplify to float approximation
sage: gp(symbolic_I)
I
sage: RR(symbolic_I)
Traceback (most recent call last):
...
TypeError: unable to convert '1.00000000000000*1' to a real number
```

We can convert to complex fields:

```
sage: C = ComplexField(200); C
Complex Field with 200 bits of precision
sage: C(symbolic_I)
1.0000000000000000000000000000000000000000000000000000000000*I
sage: symbolic_I._complex_mpfr_field_(ComplexField(53))
1.0000000000000000000000000000000000000000000000000000000000*I
sage: symbolic_I._complex_double_(CDF)
1.0*I
sage: CDF(symbolic_I)
1.0*I
sage: z = symbolic_I + symbolic_I; z
2*I
sage: C(z)
2.0000000000000000000000000000000000000000000000000000000000*I
sage: 1e8*symbolic_I
1.0000000000000000000000000000000000000000000000000000000000*I
```

```
sage: complex(symbolic_I)
lj
sage: QQbar(symbolic_I)
I
sage: abs(symbolic_I)
1
sage: symbolic_I.minpoly()
x^2 + 1
sage: maxima(2*symbolic_I)
2*%i
```

```
sage.symbolic.expression.is_SymbolicEquation(x)
    
Return True if x is a symbolic equation.
```
This function is deprecated.

EXAMPLES:

The following two examples are symbolic equations:

```python
sage: from sage.symbolic.expression import is_SymbolicEquation
sage: is_SymbolicEquation(sin(x) == x)
DeprecationWarning: is_SymbolicEquation is deprecated; use
'isinstance(x, sage.structure.element.Expression) and x.is_relational()' instead
See https://github.com/sagemath/sage/issues/35505 for details.
True
sage: is_SymbolicEquation(sin(x) < x)
True
sage: is_SymbolicEquation(x)
False
```

This is not, since \(2==3\) evaluates to the boolean \(\text{False}\):

```python
sage: is_SymbolicEquation(2 == 3)
False
```

However here since both 2 and 3 are coerced to be symbolic, we obtain a symbolic equation:

```python
sage: is_SymbolicEquation(SR(2) == SR(3))
True
```

`sage.symbolic.expression.make_map(subs_dict)`

Construct a new substitution map

OUTPUT:

A new `SubstitutionMap` for doctesting

EXAMPLES:

```python
sage: from sage.symbolic.expression import make_map
sage: make_map({x:x+1})
SubsMap
```

`sage.symbolic.expression.math_sorted(expressions)`

Sort a list of symbolic numbers in the “Mathematics” order

INPUT:

- expressions – a list/tuple/iterable of symbolic expressions, or something that can be converted to one.

OUTPUT:

The list sorted by ascending (real) value. If an entry does not define a real value (or plus/minus infinity), or if the comparison is not known, a `ValueError` is raised.

EXAMPLES:

```python
sage: from sage.symbolic.expression import math_sorted
sage: math_sorted([SR(1), SR(e), SR(pi), sqrt(2)])
[1, sqrt(2), e, pi]
```
sage.symbolic.expression.mixed_order(lhs, rhs)

Comparison in the mixed order

INPUT:

• lhs, rhs – two symbolic expressions or something that can be converted to one.

OUTPUT:

Either −1, 0, or +1 indicating the comparison. An exception is raised if the arguments cannot be converted into the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.expression import mixed_order
sage: mixed_order(1, oo)
-1
sage: mixed_order(e, oo)
-1
sage: mixed_order(pi, oo)
-1
sage: mixed_order(1, sqrt(2))
-1
sage: mixed_order(x + x^2, x*(x+1))
-1
```

Check that github issue #12967 is fixed:

```
sage: mixed_order(SR(oo), sqrt(2))
1
```

Ensure that github issue #32185 is fixed:

```
sage: mixed_order(pi, 0)
1
sage: mixed_order(golden_ratio, 0)
1
sage: mixed_order(log2, 0)
1
```

sage.symbolic.expression.mixed_sorted(expressions)

Sort a list of symbolic numbers in the “Mixed” order

INPUT:

• expressions – a list/tuple/iterable of symbolic expressions, or something that can be converted to one.

OUTPUT:

In the list the numeric values are sorted by ascending (real) value, and the expressions with variables according to print order. If an entry does not define a real value (or plus/minus infinity), or if the comparison is not known, a ValueError is raised.

EXAMPLES:

```
sage: from sage.symbolic.expression import mixed_sorted
sage: mixed_sorted([SR(1), SR(e), SR(pi), sqrt(2), x, sqrt(x), sin(1/x)])
[1, sqrt(2), e, pi, sin(1/x), sqrt(x), x]
```
sage.symbolic.expression.newExpression(parent, x)

Convert $x$ into the symbolic expression ring $parent$. This is the element constructor.

EXAMPLES:

```python
sage: a = SR(-3/4); a
-3/4
sage: type(a)
<class 'sage.symbolic.expression.Expression'>
sage: a.parent()
Symbolic Ring
sage: K.<a> = QuadraticField(-3)
# needs sage.rings.number_field
sage: a + sin(x)
# needs sage.rings.number_field
I*sqrt(3) + sin(x)
sage: x = var('x'); y0,y1 = PolynomialRing(ZZ,2,'y').gens()
sage: x+y0/y1
x + y0/y1
sage: x.subs(x=y0/y1)
y0/y1
sage: x + int(1)
x + 1
```

sage.symbolic.expression.newExpression_from_pyobject(parent, x, force=True, recursive=True)

Wrap the given Python object in a symbolic expression even if it cannot be coerced to the Symbolic Ring.

INPUT:

- `parent` - a symbolic ring.
- `x` - a Python object.
- `force` - bool, default True, if True, the Python object is taken as is without attempting coercion or list traversal.
- `recursive` - bool, default True, disables recursive traversal of lists.

EXAMPLES:

```python
sage: t = SR._force_pyobject(QQ); t  # indirect doctest
Rational Field
sage: type(t)
<class 'sage.symbolic.expression.Expression'>
```

```python
sage: from sage.symbolic.expression import newExpression_from_pyobject
sage: t = newExpression_from_pyobject(SR, 17); t
17
sage: type(t)
<class 'sage.symbolic.expression.Expression'>
```

```python
sage: t2 = newExpression_from_pyobject(SR, t, False); t2
17
sage: t2 is t
True
sage: tt = newExpression_from_pyobject(SR, t, True); tt
```

(continues on next page)
sage.symbolic.expression.new_Expression_symbol

Look up or create a symbol.

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...

sage.symbolic.expression.new_Expression_wild

Return the n-th wild-card for pattern matching and substitution.

INPUT:

- parent - a symbolic ring.
- 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
sin($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)
sage.symbolic.expression.normalize_index_for_doctests(arg, nops)
Wrapper function to test normalize_index.

sage.symbolic.expression.paramset_from_Expression(e)

EXAMPLES:

sage: from sage.symbolic.expression import paramset_from_Expression
sage: f = function('f')
sage: paramset_from_Expression(f(x).diff(x))
[0]

sage.symbolic.expression.print_order(lhs, rhs)
Comparison in the print order

INPUT:

• lhs, rhs – two symbolic expressions or something that can be converted to one.

OUTPUT:

Either −1, 0, or +1 indicating the comparison. An exception is raised if the arguments cannot be converted into the symbolic ring.

EXAMPLES:

sage: from sage.symbolic.expression import print_order
sage: print_order(1, oo)
1
sage: print_order(e, oo)
-1
sage: print_order(pi, oo)
1
sage: print_order(1, sqrt(2))
1

Check that github issue #12967 is fixed:

sage: print_order(SR(oo), sqrt(2))
1

sage.symbolic.expression.print_sorted(expressions)
Sort a list in print order

INPUT:

• expressions – a list/tuple/iterable of symbolic expressions, or something that can be converted to one.

OUTPUT:

The list sorted by print_order().

EXAMPLES:

sage: from sage.symbolic.expression import print_sorted
sage: print_sorted([SR(1), SR(e), SR(pi), sqrt(2)])
[e, sqrt(2), pi, 1]

sage.symbolic.expression.py_atan2_for_doctests(x, y)
Wrapper function to test py_atan2.

2.1. Symbolic Expressions 135
sage.symbolic.expression.py_denom_for_doctests(n)

This function is used to test py_denom().

EXAMPLES:

```python
sage: from sage.symbolic.expression import py_denom_for_doctests
sage: py_denom_for_doctests(2/3)
3
```

sage.symbolic.expression.py_eval_infinity_for_doctests()

This function tests py_eval_infinity.

sage.symbolic.expression.py_eval_neg_infinity_for_doctests()

This function tests py_eval_neg_infinity.

sage.symbolic.expression.py_eval_unsigned_infinity_for_doctests()

This function tests py_eval_unsigned_infinity.

sage.symbolic.expression.py_exp_for_doctests(x)

This function tests py_exp.

EXAMPLES:

```python
sage: from sage.symbolic.expression import py_exp_for_doctests
sage: py_exp_for_doctests(CC(2))
7.38905609893065
```

sage.symbolic.expression.py_factorial_py(x)

This function is a python wrapper around py_factorial(). This wrapper is needed when we override the eval() method for GiNaC's factorial function in sage.functions.other.Function_factorial.

sage.symbolic.expression.py_float_for_doctests(n, kwds)

This function is for testing py_float.

EXAMPLES:

```python
sage: from sage.symbolic.expression import py_float_for_doctests
sage: py_float_for_doctests(pi, {parent:RealField(80)})
3.1415926535897932384626
sage: py_float_for_doctests(pi, {parent:RealField(80)})
3.1415926535897932384626
sage: py_float_for_doctests(I, {parent:RealField(80)})
1.0000000000000000000000*I
sage: py_float_for_doctests(I, {parent:float})
1j
sage: py_float_for_doctests(pi, {parent:complex})
(3.141592653589793+0j)
```

sage.symbolic.expression.py_imag_for_doctests(x)

Used for doctesting py_imag.

sage.symbolic.expression.py_is_cinteger_for_doctests(x)

Returns True if pynac should treat this object as an element of \(\mathbb{Z}(i)\).

sage.symbolic.expression.py_is_crational_for_doctests(x)

Return True if pynac should treat this object as an element of \(\mathbb{Q}(i)\).

sage.symbolic.expression.py_is_integer_for_doctests(x)

Used internally for doctesting purposes.
sage.symbolic.expression.py_latex_fderivative_for_doctests(id, params, args)

Used internally for writing doctests for certain cdef’d functions.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_latex_fderivative_for_doctests as..
    →py_latex_fderivative, get_ginac_serial, get_fn_serial

sage: var('x,y,z')
(x, y, z)
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\text{D}_{0, 1, 0, 1}\left({\rm foo}\right)(x, y^{z})
```

Test latex_name:

```
sage: foo = function('foo', nargs=2, latex_name=r'\text{bar}')
sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\text{D}_{0, 1, 0, 1}\text{bar}(x, y^{z})
```

Test custom func:

```
sage: def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr, args)) + ")"sage: foo = function('foo', nargs=2, print_latex_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\text{D}_{0, 1, 0, 1}\text{func_with_args}(x, y^{z})
```

sage.symbolic.expression.py_latex_function_pystring(id, args, fname_paren=False)

Return a string with the latex representation of the symbolic function specified by the given id applied to args.

See documentation of py_print_function_pystring for more information.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_latex_function_pystring, get_ginac_,
    →serial, get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: var('x,y,z')
(x, y, z)
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
(continues on next page)
```
Symbolic Calculus, Release 10.3

....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_function_pystring(i, (x,y^z))
'\left(\mathrm{foo}\right)\left(x, y^{z}\right)'

Test latex_name:

sage: foo = function('foo', nargs=2, latex_name=r'\mathrm{bar}')
sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ....:   if get_sfunction_from_serial(i) == foo: break
sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_function_pystring(i, (x,y^z))
'\mathrm{bar}\left(x, y^{z}\right)'

Test custom func:

sage: def my_print(self, *args): return "my args are: " + ', '.join(map(repr, ...
˓→args))
sage: foo = function('foo', nargs=2, print_latex_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ....:   if get_sfunction_from_serial(i) == foo: break
sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_function_pystring(i, (x,y^z))
'my args are: x, y^{z}'

sage.symbolic.expression.py_latex_variable_for_doctests(x)
Internal function used so we can doctest a certain cdef’d method.

EXAMPLES:

sage: sage.symbolic.expression.py_latex_variable_for_doctests('x')
x
sage: sage.symbolic.expression.py_latex_variable_for_doctests('sigma')
\sigma

sage.symbolic.expression.py_lgamma_for_doctests(x)
This function tests py_lgamma.

EXAMPLES:

sage: from sage.symbolic.expression import py_lgamma_for_doctests
sage: py_lgamma_for_doctests(CC(I))
-0.650923199301856 - 1.87243664726243*I

sage.symbolic.expression.py_li2_for_doctests(x)
This function is a python wrapper so py_psi2 can be tested. The real tests are in the docstring for py_psi2.

EXAMPLES:
sage: from sage.symbolic.expression import py_li2_for_doctests
sage: py_li2_for_doctests(-1.1)
-0.890838090262283

sage.symbolic.expression.py_li_for_doctests(x, n, parent)
This function is a python wrapper so py_li can be tested. The real tests are in the docstring for py_li.

EXAMPLES:

sage: from sage.symbolic.expression import py_li_for_doctests
sage: py_li_for_doctests(0,2,float)
0.000000000000000

sage.symbolic.expression.py_log_for_doctests(x)
This function tests py_log.

EXAMPLES:

sage: from sage.symbolic.expression import py_log_for_doctests
sage: py_log_for_doctests(CC(e))
1.00000000000000

sage.symbolic.expression.py_mod_for_doctests(x, n)
This function is a python wrapper so py_mod can be tested. The real tests are in the docstring for py_mod.

EXAMPLES:

sage: from sage.symbolic.expression import py_mod_for_doctests
sage: py_mod_for_doctests(5, 2)
1

sage.symbolic.expression.py_numer_for_doctests(n)
This function is used to test py_numer().

EXAMPLES:

sage: from sage.symbolic.expression import py_numer_for_doctests
sage: py_numer_for_doctests(2/3)
2

sage.symbolic.expression.py_print_fderivative_for_doctests(id, params, args)
Used for testing a cdef’d function.

EXAMPLES:

sage: from sage.symbolic.expression import py_print_fderivative_for_doctests as...
    →py_print_fderivative, get_ginac_serial, get_fn_serial
sage: var("x,y,z")
(x, y, z)

sage: from sage.symbolic.function import get_sfunction_from_serial
sage: foo = function('foo', nargs=2)

sage: for i in range(get_ginac_serial(), get_fn_serial()):
    ...:     if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_fderivative(i, (0, 1, 0, 1), (x, y^z))
D[0, 1, 0, 1](foo)(x, y^z)
Test custom print function:

```python
sage: def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr, args)) +")"

sage: foo = function('foo', nargs=2, print_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:     if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_fderivative(i, (0, 1, 0, 1), (x, y^z))
D[0, 1, 0, 1]func_with_args(x, y^z)
```

`sage.symbolic.expression.py_print_function_pystring(id, args, fname_paren=False)`

Return a string with the representation of the symbolic function specified by the given id applied to args.

**INPUT:**
- `id` – serial number of the corresponding symbolic function
- `params` – Set of parameter numbers with respect to which to take the derivative.
- `args` – arguments of the function.

**EXAMPLES:**

```python
sage: from sage.symbolic.expression import py_print_function_pystring, get_ginac_serial, get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: var('x,y,z')
(x, y, z)
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:     if get_sfunction_from_serial(i) == foo: break
sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_function_pystring(i, (x,y))
foo(x, y)
sage: py_print_function_pystring(i, (x,y), True)
'(foo)(x, y)
```

`sage.symbolic.expression.py_psi2_for_doctests(n, x)`

This function is a python wrapper so py_psi2 can be tested. The real tests are in the docstring for py_psi2.

**EXAMPLES:**

```python
sage: from sage.symbolic.expression import py_psi2_for_doctests
sage: py_psi2_for_doctests(1, 2)
0.644934066848226
```
sage.symbolic.expression.py_psi_for_doctests(x)

This function is a python wrapper so py_psi can be tested. The real tests are in the docstring for py_psi.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_psi_for_doctests
g Sage: py_psi_for_doctests(2)
0.422784335098467
```

sage.symbolic.expression.py_real_for_doctests(x)

Used for doctesting py_real.

sage.symbolic.expression.py_stieltjes_for_doctests(x)

This function is for testing py_stieltjes().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_stieltjes_for_doctests
g Sage: py_stieltjes_for_doctests(0.0)
0.577215664901533
```

sage.symbolic.expression.py_tgamma_for_doctests(x)

This function is for testing py_tgamma().

sage.symbolic.expression.py_zeta_for_doctests(x)

This function is for testing py_zeta().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_zeta_for_doctests
g Sage: py_zeta_for_doctests(CC.0)
0.00330022368532410 - 0.418155449141322*I
```

sage.symbolic.expression.register_or_update_function(self, name, latex_name, nargs, evalf_params_first, update)

Register the function self with Pynac (GiNaC).

OUTPUT:

- serial number of the function, for use in call_registered_function()

EXAMPLES:

```
sage: from sage.symbolic.function import BuiltinFunction
g Sage: class Archosaurian(BuiltinFunction):
        ....: def __init__(self):
        ....:     BuiltinFunction.__init__(self, 'archsaur', nargs=1)
        ....: def _eval_(self, x):
        ....:     return x * exp(x)
        sage: archsaur = Archosaurian()  # indirect doctest
        sage: archsaur(2)
       2*e^2
```

sage.symbolic.expression.restore_op_wrapper(expr)

sage.symbolic.expression.solve_diophantine(f, *args, **kwds)

Solve a Diophantine equation.

The argument, if not given as symbolic equation, is set equal to zero. It can be given in any form that can be converted to symbolic. Please see Expression.solve_diophantine() for a detailed synopsis.
EXAMPLES:

```python
sage: R.<a,b> = PolynomialRing(ZZ); R
Multivariate Polynomial Ring in a, b over Integer Ring
sage: solve_diophantine(a^2 - 3*b^2 + 1)
[]
sage: sorted(solve_diophantine(a^2 - 3*b^2 + 2), key=str)
[(-1/2*sqrt(3)*(sqrt(3) + 2)^t + 1/2*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^t,
-1/6*sqrt(3)*(sqrt(3) + 2)^t + 1/6*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^t,
(1/2*sqrt(3)*(sqrt(3) + 2)^t - 1/2*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^t,
1/6*sqrt(3)*(sqrt(3) + 2)^t - 1/6*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^t)
```

**sage.symbolic.expression.test_binomial**(\(n, k\))

The Binomial coefficients. It computes the binomial coefficients. For integer \(n\) and \(k\) and positive \(n\) this is the number of ways of choosing \(k\) objects from \(n\) distinct objects. If \(n\) is negative, the formula \(\text{binomial}(n, k) = (-1)^k \cdot \text{binomial}(n-k-1, k)\) is used to compute the result.

**INPUT:**
- \(n, k\) – integers, with \(k \geq 0\).

**OUTPUT:**
- integer

**EXAMPLES:**

```python
sage: import sage.symbolic.expression
sage: sage.symbolic.expression.test_binomial(5, 2)
10
sage: sage.symbolic.expression.test_binomial(-5, 3)
-35
sage: -sage.symbolic.expression.test_binomial(3-(-5)-1, 3)
-35
```

**sage.symbolic.expression.tolerant_is_symbol**(\(a\))

Utility function to test if something is a symbol.

Returns False for arguments that do not have an is_symbol attribute. Returns the result of calling the is_symbol method otherwise.

**EXAMPLES:**

```python
sage: from sage.symbolic.expression import tolerant_is_symbol
sage: tolerant_is_symbol(var("x"))
True
sage: tolerant_is_symbol(None)
False
sage: None.is_symbol()
Traceback (most recent call last):
  ...
AttributeError: 'NoneType' object has no attribute 'is_symbol'...
```

**sage.symbolic.expression.unpack_operands**(\(ex\))

**EXAMPLES:**
2.2 Callable Symbolic Expressions

EXAMPLES:

When you do arithmetic with:

\[
sage: f(x, y, z) = \sin(x+y+z) \\
sage: g(x, y) = y + 2x \\
sage: f + g \\
(x, y, z) \mapsto 2x + y + \sin(x + y + z)
\]

\[
sage: f(x, y, z) = \sin(x+y+z) \\
sage: g(w, t) = \cos(w - t) \\
sage: f + g \\
(t, w, x, y, z) \mapsto \cos(-t + w) + \sin(x + y + z)
\]

\[
sage: f(x, y, t) = y^*(x^2-t) \\
sage: g(x, y, w) = x + y - \cos(w) \\
sage: f\cdot g \\
(x, y, t, w) \mapsto (x^2 - t)^*(x + y - \cos(w))*y
\]

\[
sage: f(x,y, t) = x+y \\
sage: g(x, y, w) = w + t \\
sage: f + g \\
(x, y, t, w) \mapsto t + w + x + y
\]
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 ≠ 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

```python
class sage.symbolic.callable.CallableSymbolicExpressionRingFactory
    Bases: UniqueFactory

create_key (args, check=True)
    EXAMPLES:
    sage: x, y = var('x, y')
    sage: CallableSymbolicExpressionRing.create_key((x, y))
    (x, y)
```

```python
create_object (version, key, **extra_args)
    Return a CallableSymbolicExpressionRing given a version and a key.
    EXAMPLES:
    sage: x, y = var('x, y')
    sage: CallableSymbolicExpressionRing.create_object(0, (x, y))
    Callable function ring with arguments (x, y)
```

```python
class sage.symbolic.callable.CallableSymbolicExpressionRing_class (arguments)
    Bases: SymbolicRing, CallableSymbolicExpressionRingRing

    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
```

```python
args ()
    Return the arguments of self.
    The order that the variables appear in self.arguments() is the order that is used in evaluating the elements of self.
    EXAMPLES:
    sage: x, y = var('x, y')
    sage: f(x, y) = 2*x+y
    sage: f.parent().arguments()
    (x, y)
    sage: f(y, x) = 2*x+y
    sage: f.parent().arguments()
    (y, x)
```

```python
arguments ()
    Return the arguments of self.
    The order that the variables appear in self.arguments() is the order that is used in evaluating the elements of self.
    EXAMPLES:
```
Symbolic Calculus, Release 10.3

```
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)
```

2.3 Assumptions

The *GenericDeclaration* class provides assumptions about a symbol or function in verbal form. Such assumptions can be made using the *assume()* function in this module, which also can take any relation of symbolic expressions as argument. Use *forget()* to clear all assumptions. Creating a variable with a specific domain is equivalent with making an assumption about it.

There is only rudimentary support for consistency and satisfiability checking in Sage. Assumptions are used both in Maxima and Pynac to support or refine some computations. In the following we show how to make and query assumptions. Please see the respective modules for more practical examples.

In addition to the global *assumptions()* database, *assuming()* creates reusable, stackable context managers allowing for temporary updates of the database for evaluation of a (block of) statements.

EXAMPLES:

The default domain of a symbolic variable is the complex plane:

```
sage: var('x')
x
sage: x.is_real()
False
sage: assume(x,'real')
sage: x.is_real()
True
sage: forget()
sage: x.is_real()
False
```

Here is the list of acceptable features:

```
sage: ', '.join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary, complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, constant, commutative, lassociative, rassociative, symmetric, antisymmetric, integervalued'
```

Set positive domain using a relation:
Symbolic Calculus, Release 10.3

```
sage: assume(x>0)
sage: x.is_positive()
True
sage: x.is_real()
True
sage: assumptions()
[x > 0]
```

Assumptions also affect operations that do not use Maxima:

```
sage: forget()
sage: assume(x, 'even')
sage: assume(x, 'real')
sage: (-1)^x
1
sage: (-gamma(pi))^x
gamma(pi)^x
sage: binomial(2*x, x).is_integer()
True
```

Assumptions are added and in some cases checked for consistency:

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

```class sage.symbolic.assumptions.GenericDeclaration(var, assumption)
Bases: UniqueRepresentation

This class represents generic assumptions, such as a variable being an integer or a function being increasing. It passes such information to Maxima's declare (wrapped in a context so it is able to forget) and to Pynac.

INPUT:

• var – the variable about which assumptions are being made
• assumption – 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)
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:

2.3. Assumptions 147
Symbolic Calculus, Release 10.3

```
sage: ", ",.join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun,
constant, commutative, lassociative, rassociative, symmetric,
antisymmetric, integervalued'
```

Test unique representation behavior:

```
sage: GenericDeclaration(x, 'integer') is GenericDeclaration(SR.var("x"), 'integer →')
True
```

`assume()`

Make this assumption.

`contradicts(soln)`

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

**INPUT:**

- `soln` – Either a dictionary with variables as keys or a symbolic relation with a variable on the left hand side.

**EXAMPLES:**

```
sage: from sage.symbolic.assumptions import GenericDeclaration
sage: GenericDeclaration(x, 'integer').contradicts(x==4)
False
sage: GenericDeclaration(x, 'integer').contradicts(x==4.0)
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
```

(continues on next page)
forget()  
Forget this assumption.

has(arg)  
Check if this assumption contains the argument arg.

EXAMPLES:

```python
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 -- a variable-length sequence of assumptions, each consisting of:
  - any number of symbolic inequalities, like $0 < x$, $x < 1$
  - a subsequence of variable names, followed by some property that should be assumed for those variables; for example, $x$, $y$, $z$, 'integer' would assume that each of $x$, $y$, and $z$ are integer variables, and $x$, 'odd' would assume that $x$ is odd (as opposed to even).

The two types can be combined, but a symbolic inequality cannot appear in the middle of a list of variables.

OUTPUT:

If everything goes as planned, there is no output.

If you assume something that is not one of the two forms above, then an AttributeError is raised as we try to call its assume method.

If you make inconsistent assumptions (for example, that $x$ is both even and odd), then a ValueError is raised.

**Warning:** Do not use Python's chained comparison notation in assumptions. Python literally translates the expression $0 < x < 1$ to $(0 < x) \land (x < 1)$, but the value of `bool(0 < x)` is `False` when $x$ is a symbolic variable. Therefore, by the definition of Python's logical “and” operator, the entire expression is equal to $0 < x$.

EXAMPLES:

Assumptions are typically used to ensure certain relations are evaluated as true that are not true in general.

Here, we verify that for $x > 0$, $\sqrt{x^2} = x$:
Symbolic Calculus, Release 10.3

```python
sage: assume(x > 0)
sage: bool(sqrt(x^2) == x)
True
```

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

```python
sage: bool(sqrt(x^2) == x)
True
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:

```python
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:

```python
sage: n,P,r,r2 = SR.var('n, P, r, r2')
sage: assume(n, 'integer')
sage: c = P*e^(r*n)
sage: d = P*(1+r2)^n
sage: solve(c==d,r2)
[r2 == e^r - 1]
sage: forget()
```

Simplifying certain well-known identities works as well:

```python
sage: n = SR.var('n')
sage: assume(n, 'integer')
sage: sin(n*pi)
0
sage: forget()
sage: sin(n*pi).simplify()
```

Instead of using chained comparison notation, each relationship should be passed as a separate assumption:
symbolic calculus, release 10.3

sage: x = SR.var('x')
sage: assume(0 < x, x < 1) # instead of assume(0 < x < 1)
sage: assumptions()
[0 < x, x < 1]
sage: forget()

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()

class sage.symbolic.assumptions.assuming(*args,**kwds)
Bases: object

Temporarily modify assumptions.

Create a context manager in which temporary assumptions are added (or substituted) to the current assumptions set.

The set of possible assumptions and declarations is the same as for assume().

This can be useful in interactive mode to discover the assumptions necessary to a given integration, or the exact solution to a system of equations.

2.3. Assumptions 151
It can also be used to explore the branches of a `cases()` expression.

As with `assume()`, it is an error to add an assumption either redundant or inconsistent with the current assumption set (unless `replace=True` is used). See examples.

**INPUT:**

- *args – assumptions (same format as for `assume()`).

- **replace** – a boolean (default `[False]`) Specifies whether the new assumptions are added to (default) or replace (if `replace=True`) the current assumption set.

**OUTPUT:**

A context manager useable in a `with` statement (see examples).

**EXAMPLES:**

Basic functionality: inside a `with assuming:()` block, Sage uses the updated assumptions database. After exit, the original database is restored.

```
sage: var("x")
x
sage: forget(assumptions())
sage: solve(x^2 == 4,x)
[x == -2, x == 2]
sage: with assuming(x > 0):
    ....: solve(x^2 == 4,x)
[x == 2]
sage: assumptions()
[]
```

The local assumptions can be stacked. We can use this functionality to discover incrementally the assumptions necessary to a given calculation (and by the way, to check that Sage’s default integrator (Maxima’s, that is), sometimes nitpicks for naught).

```
sage: var("y,k,theta")
(y, k, theta)
sage: dgamma(y,k,theta)=y^(k-1)*e^(-y/theta)/(theta^k*gamma(k))
sage: integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
  ... ValueError: Computation failed since Maxima requested additional constraints;→ using the 'assume' command before evaluation *may* help (example of legal→ syntax is 'assume(theta>0)', see 'assume?' for more details)
Is theta positive or negative?
sage: a1=assuming(theta>0)
sage: with a1:integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
  ... ValueError: Computation failed since Maxima requested additional constraints;→ using the 'assume' command before evaluation *may* help (example of legal→ syntax is 'assume(k>0)', see 'assume?' for more details)
Is k positive, negative or zero?
sage: a2=assuming(k>0)
sage: with a1,a2:integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
... ValueError: Computation failed since Maxima requested additional constraints;
```

(continues on next page)
Is k an integer?

```python
sage: a3=assuming(k,"noninteger")
sage: with a1,a2,a3:integrate(dgamma(y,k,theta),y,0,oo)
1
```

As mentioned above, it is an error to try to introduce redundant or inconsistent assumptions.

```python
sage: assume(x > 0)
sage: with assuming(x > -1): "I wont see this"
Traceback (most recent call last):
...
ValueError: Assumption is redundant
```

```python
sage: assume(x < -1)
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
```

### `sage.symbolic.assumptions.assumptions (*args)`

List all current symbolic assumptions.

**INPUT:**
- `args` – list of variables which can be empty.

**OUTPUT:**
- list of assumptions on variables. If `args` is empty it returns all assumptions

**EXAMPLES:**

```python
sage: var('x, y, z, w')
(x, y, z, w)
sage: forget()
sage: assume(x^2+y^2 > 0)
sage: assumptions()
[x^2 + y^2 > 0]
sage: forget(x^2+y^2 > 0)
sage: assumptions()
[]
sage: assume(x > y)
sage: assume(z > w)
sage: sorted(assumptions(), key=lambda x: str(x))
[x > y, z > w]
sage: forget()
sage: assumptions()
[]
```

It is also possible to query for assumptions on a variable independently:

```python
sage: x, y, z = var('x y z')
sage: assume(x, 'integer')
sage: assume(y > 0)
```

(continues on next page)
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: 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.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:
2.4 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')
qe = (a*x^2 + b*x + c == 0)
print(solve(qe, x))
```

```
[ x == -1/2*(b + sqrt(b^2 - 4*a*c))/a,
  x == -1/2*(b - sqrt(b^2 - 4*a*c))/a
]
```

2.4.1 The operator, left hand side, and right hand side

Operators:

```sage
eqn = x^3 + 2/3 >= x - pi
print(eqn.operator())
print((x^3 + 2/3 < x - pi).operator())
print((x^3 + 2/3 == x - pi).operator())
```

Left hand side:

```sage
eqn = x^3 + 2/3 >= x - pi
eqn.lhs()
```

```
x^3 + 2/3
```

Right hand side:

```sage
(x + sqrt(2) >= sqrt(3) + 5/2).right()
```

```
sqrt(3) + 5/2
```
2.4.2 Arithmetic

Add two symbolic equations:

```python
sage: var('a,b')
(a, b)
sage: m = 144 == -10 * a + b
sage: n = 136 == 10 * a + b
sage: m + n
280 == 2*b
```

Subtract two symbolic equations:

```python
sage: var('a,b')
(a, b)
sage: m = 144 == 20 * a + b
sage: n = 136 == 10 * a + b
sage: m - n
8 == 10*a
```

Multiply two symbolic equations:

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

Divide two symbolic equations:

```python
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi, hold=True)
sage: m/n
```

```
x/sin(x) == (5*x + 1)/sin(2*pi + x)
```

```
sage: m = x != 5*x + 1
sage: n = sin(x) != sin(x+2*pi, hold=True)
sage: m/n
```

```
x/sin(x) != (5*x + 1)/sin(2*pi + x)
```
2.4.3 Substitution

Substitution into relations:

```python
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)
```

You can even substitute multivariable and matrix expressions:

```python
sage: x, y = var('x, y')
sage: M = Matrix([[x+1, y], [x^2, y^3]]); M
[x + 1  y]
[x^2  y^3]
sage: M.substitute({x:0, y:1})
[1 1]
[0 1]
```

2.4.4 Solving

We can solve equations:

```python
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 can also solve equations involving matrices. The following example defines a multivariable function \( f(x, y) \), then solves for where the partial derivatives with respect to \( x \) and \( y \) are zero. Then it substitutes one of the solutions into the Hessian matrix \( H \) for \( f \):

```python
sage: f(x,y) = x^2*y+y^2+y
sage: solutions = solve(list(f.diff()), [x, y], solution_dict=True)
sage: solutions
[{x: -I, y: 0}, {x: I, y: 0}, {x: 0, y: -1/2}]
```
Symbolic Calculus, Release 10.3

(continued from previous page)

True
sage: H = f.diff(2) # Hessian matrix
sage: H.subs(solutions[2])
[(x, y) |--> -1 (x, y) |--> 0]
[(x, y) |--> 0 (x, y) |--> 2]
sage: H(x,y).subs(solutions[2])
[-1 0]
[ 0 2]

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.
2.4.5 Assumptions

Forgetting assumptions:

```sage
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()
[]
```

2.4.6 Miscellaneous

Conversion to Maxima:

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

Comparison:

```sage
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:
Symbolic Calculus, Release 10.3

\begin{verbatim}
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 comparisons do not automatically simplify even in trivial situations:

\begin{verbatim}
sage: SR(I)^2 == -1
-1 == -1
sage: SR(I)^2 < 0
-1 < 0
sage: (SR(I)+1)^4 > 0
-4 > 0

Nevertheless, if you force the comparison, you get the right answer (github issue #7160):

\begin{verbatim}
sage: bool(SR(I)^2 == -1)
True
sage: bool(SR(I)^2 < 0)
True
sage: bool((SR(I)+1)^4 > 0)
False
\end{verbatim}
\end{verbatim}

2.4.7 More Examples

\begin{verbatim}
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 == 1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(2*sqrt(5) + 10) - 1), x == -1/4*(-a)^(1/5)*(sqrt(5) - I*sqrt(-2*sqrt(5) + 10) + 1), x == -1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(-2*sqrt(5) + 10) + 1), x == 1/4*(-a)^(1/5)*(sqrt(5) - I*sqrt(2*sqrt(5) + 10) - 1), x == (-a)^(1/5)]
\end{verbatim}

You can also do arithmetic with inequalities, as illustrated below:

\begin{verbatim}
sage: var('x y')
(x, y)
sage: f = x + 3 == y - 2
sage: g = f - 3; g
x == y - 5
sage: h = x^3 + sqrt(2) == x*y*sin(x)
\end{verbatim}
\end{verbatim}

(continues on next page)
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).
- `algorithm` - string (default: ‘maxima’); to use SymPy’s solvers set this to ‘sympy’. Note that SymPy is always used for diophantine equations. Another choice is ‘giac’.
- `domain` - string (default: ‘complex’); setting this to ‘real’ changes the way SymPy solves single equations; inequalities are always solved in the real domain.

**EXAMPLES:**

```python
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
```
Symbolic Calculus, Release 10.3

Whenever possible, answers will be symbolic, but with systems of equations, at times approximations will be given by Maxima, due to the underlying algorithm:

```
sage: sols = solve([x^3==y, y^2==x], [x, y]); sols[-1], sols[0]  
([x == 0, y == 0],  
  [x == (0.3090169943749475 + 0.9510565162951535*I), 
   y == (-0.8090169943749475 - 0.5877852522924731*I)])
sage: sols[0][0].rhs().pyobject().parent() 
Complex Double Field
sage: solve(y^6==y, y)  
[y == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,  
y == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,  
y == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,  
y == 1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,  
y == 1,  
y == 0]
sage: solve( y^6 == y, y)==solve( y^6 == y, y) 
True
```

Here we demonstrate very basic use of the optional keywords:

```
sage: ((x^2-1)^2).solve(x)  
[x == -1, x == 1]  
sage: ((x^2-1)^2).solve(x,multiplicities=True)  
([(x == -1, x == 1), [2, 2]])  
sage: solve(sin(x)==x,x)  
[x == sin(x)]  
sage: solve(sin(x)==x,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)  
(continues on next page)
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, \( r_1 \) is an arbitrary constant (because of the \( r \)):

```
sage: forget()
sage: x, y = var("x,y")
sage: solve([x+y == 3, 2*x+2*y == 6], x, y)
[{
  x: -r1 + 3,  
  y: r1}
]
sage: var("b, c")
(b, c)
sage: solve((b-1)*(c-1), [b,c])
[{
  b: 1,  
  c: r...},
  {
  b: r...,  
  c: 1}
]
```

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

```
sage: solve(\sin(x)==\cos(x), x, to_poly_solve=True)
[{
  x: 1/4*pi + pi*z...}
]
sage: solve((\cos(x)\sin(x) == 1/2, x+y == 0], x, y)
[{
  x: 1/4*pi + pi*z...,  
  y: -1/4*pi - pi*z...}
]
```

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

Inequalities can also be solved:

```
sage: solve(x^2>8,x)
[[x < -2*sqrt(2)], [x > 2*sqrt(2)]]
sage: x,y = var(x,y); (ln(x)-ln(y)>0).solve(x)
[[log(x) - log(y) > 0]]
sage: x,y = var(x,y); (ln(x)>ln(y)).solve(x)
# random
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

A simple example to show the use of the `multiplicities` keyword:

```
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):
... Not Implemented Error: 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)
```

Symbolic Calculus, Release 10.3

[x == -10, x == 12]

sage: var('Q')
Q
sage: solve(Q*sqrt(Q^2 + 2) - 1, Q)

The following example is a regression in Maxima 5.39.0. It used to be possible to get one more solution here, namely \(1/\sqrt{\sqrt{2} + 1}\), see https://sourceforge.net/p/maxima/bugs/3276/:

sage: solve(Q*sqrt(Q^2 + 2) - 1, Q, to_poly_solve=True)

An effort is made to only return solutions that satisfy the current assumptions:

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

The same may also apply if a returned unsolved expression has a denominator, but the original one did not:

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

(continued on next page)
We use SymPy for Diophantine equations, see `Expression.solve_diophantine`:

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

The following shows some more of SymPy’s capabilities that cannot be handled by Maxima:

```python
sage: _ = var('t')
sage: r = solve([x^2 - y^2/exp(x), y-1], x, y, algorithm='sympy')
sage: r[0][x], r[0][y]
(2*lambert_w(-1/2), 1)
sage: solve(-2*x^3 + 4*x^2 - 2*x + 6 > 0, x, algorithm='sympy')
[x < 1/3*(1/2)^(1/3)*(9*sqrt(77) + 79)^(1/3) + 2/3*(1/2)^(2/3)/(9*sqrt(77) + 79)^(1/3) + 2/3]
sage: solve(sqrt(2*x^2 - 7) - (3 - x), x, algorithm='sympy')
[x == -8, x == 2]
sage: solve(sqrt(2*x + 9) - sqrt(x + 1) - sqrt(x + 4), x, algorithm='sympy')
[x == 0]
sage: r = solve([x^2+y+z, y+x^2+z, x+y+z^2], x, y,z, algorithm='sympy')
sage: (r[0][x], r[0][y])
(z, -(z + 1)*z)
sage: (r[1][x], r[1][y])
(-z + 1, -z^2 + z - 1)
sage: solve(abs(x + 3) - 2*abs(x - 3), x, algorithm='sympy', domain='real')
[x == -8, x == 2]
```

We cannot translate all results from SymPy but we can at least print them:

```python
sage: solve(sinh(x) - 2*cosh(x), x, algorithm='sympy')
[ImageSet(Lambda(_n, I*(2*_n*pi + pi/2) + log(sqrt(3))), Integers), ImageSet(Lambda(_n, I*(2*_n*pi - pi/2) + log(sqrt(3))), Integers)]
sage: solve(2*sin(x) - 2*sin(2*x), x, algorithm='sympy')
[ImageSet(Lambda(_n, 2*_n*pi), Integers), ImageSet(Lambda(_n, 2*_n*pi + pi), Integers), ImageSet(Lambda(_n, 2*_n*pi + 5*pi/3), Integers), ImageSet(Lambda(_n, 2*_n*pi + pi/3), Integers)]
sage: solve(x^5 + 3*x^3 + 7, x, algorithm='sympy')[0] # known bug
complex_root_of(x^5 + 3*x^3 + 7, 0)
```

A basic interface to Giac is provided:

```python
sage: solve([(2/3)^x-2], [x], algorithm='giac')
...[[(-log(2))/(log(3) - log(2))]]
```
sage: \( f = (\sin(x) - 8\cos(x)\sin(x))(\sin(x)^2 + \cos(x)) - (2\cos(x)\sin(x) - \sin(x))^2 \)  
\( -2\sin(x)^2 + 2\cos(x)^2 - \cos(x) \)  
sage: solve(f, x, algorithm='giac')  
...[-2*arctan(sqrt(2)), 0, 2*arctan(sqrt(2)), pi]  
sage: x, y = SR.var('x,y')  
sage: solve([x+y-4,x*y-3],[x,y],algorithm='giac')  
([[1, 3], [3, 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_fourier). See the documentation of these functions for more details.

INPUT:

- **ineq** - one inequality or a list of inequalities

  Case1: If *ineq* is one equality, then it should be rational expression in one variable. This input is passed to sage.symbolic.relation.solve_ineq_univar function.

  Case2: If *ineq* is a list involving one or more inequalities, than the input is passed to sage.symbolic.relation.solve_ineq_fourier function. This function can be used for system of linear inequalities and for some types of nonlinear inequalities. See http://maxima.cvs.sourceforge.net/viewvc/maxima/maxima/share/contrib/fourier_elim/rtest_fourier_elim.mac for a big gallery of problems covered by this algorithm.

- **vars** - optional parameter with list of variables. This list is used only if Fourier elimination is used. If omitted or if rational inequality is solved, then variables are determined automatically.

OUTPUT:

- **list** – output is list of solutions as a list of simple inequalities output \([A,B,C]\) means \((A \text{ or } B \text{ or } C)\) each \(A, B, C\) is again a list and if \(A=[a,b]\), then \(A\) means \((a \text{ and } b)\).

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])  # 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)

```python
sage.symbolic.relation.solve_ineq_fourier(ineq, vars=None)
```

Solves system of inequalities using Maxima and Fourier elimination

Can be used for system of linear inequalities and for some types of nonlinear inequalities. For examples, see the example section below and http://maxima.cvs.sourceforge.net/viewvc/maxima/maxima/share/contrib/fourier_elim/rtest_fourier_elim.mac

INPUT:

- `ineq` - list with system of inequalities
- `vars` - optionally list with variables for Fourier elimination.

OUTPUT:

- `list` - output is list of solutions as a list of simple inequalities output [A,B,C] means (A or B or C) each A, B, C is again a list and if A=[a,b], then A means (a and b). The list is empty if there is no solution.

EXAMPLES:

```python
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)]
```

Note that different systems will find default variables in different orders, so the following is not tested:

```python
sage: solve_ineq_fourier([log(x)>log(y)],[x,y])
[[0 < y, y < x, 0 < x]]
```

ALGORITHM:

Calls Maxima command `fourier_elim`

AUTHORS:

- Robert Marik (01-2010)

```python
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:

```python
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:

```python
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:

```python
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, 1), (2, 2, 1)]
```

We can solve with respect to a bigger modulus if it consists only of small prime factors:
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.

---

**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()
2.5 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 github issue #9217

EXAMPLES:
The basic units of the calculus package are symbolic expressions which are elements of the symbolic expression ring (SR). To create a symbolic variable object in Sage, use the `var()` function, whose argument is the text of that variable. Note that Sage is intelligent about LaTeXing variable names.

```
sage: x1 = var('x1'); x1
x1
sage: latex(x1)
x_{1}
sage: theta = var('theta'); theta
theta
sage: latex(theta)
\theta
```

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

```
sage: x^2
x^2
sage: type(x)
<class 'sage.symbolic.expression.Expression'>
```

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: 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))]```
Complex exponentiation works, but may require a patched version of maxima (github issue #32898) for now:

```
sage: M = i*matrix([[pi]])
sage: e^M  # not tested, requires patched maxima
[-1]
sage: M = i*matrix([[pi,0],[0,2*pi]])
sage: e^M
[-1 0]
[ 0 1]
sage: M = matrix([[0,pi],[-pi,0]])
sage: e^M
[-1 0]
[ 0 -1]
```

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)
```

Another example:

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

It is no longer allowed to call expressions with positional arguments:

```
sage: f = sin(x)
sage: f(y)
Traceback (most recent call last):
  ...
TypeError: Substitution using function-call syntax and unnamed arguments has been removed. You can use named arguments instead, like
EXPR(x=..., y=...)
sage: f(x=pi)
0
```

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:
Symbolic Calculus, Release 10.3

\[
sage: f(x,y) = \log((2-x)/(y+5))
\]
\[
sage: f(7,t)
\]
\[
\log(-5/(t + 5))
\]

Some further examples:

\[
sage: f = 5\sin(x)
\]
\[
sage: f
\]
\[
5\sin(x)
\]
\[
sage: f(x=2)
\]
\[
5\sin(2)
\]
\[
sage: f(x=\pi)
\]
\[
0
\]
\[
sage: \text{float}(f(x=\pi))
\]
\[
0.0
\]

Another example:

\[
sage: f = \text{integrate}(1/\sqrt{9+x^2}, x); f
\]
\[
\arcsinh(1/3*x)
\]
\[
sage: f(x=3)
\]
\[
\arcsinh(1)
\]
\[
sage: f.\text{derivative}(x)
\]
\[
1/\sqrt{x^2 + 9}
\]

We compute the length of the parabola from 0 to 2:

\[
sage: x = \text{var}'(x')
\]
\[
sage: y = x^2
\]
\[
sage: dy = \text{derivative}(y,x)
\]
\[
sage: z = \text{integral}(\sqrt{1 + dy^2}, x, 0, 2)
\]
\[
\sqrt{17} + 1/4*\arcsinh(4)
\]
\[
sage: \text{n}(z,200)
\]
\[
4.646783762432935873382615567490459188510486987432887508703
\]
\[
sage: \text{float}(z)
\]
\[
4.646783762432936
\]

We test pickling:

\[
sage: x, y = \text{var}'(x,y')
\]
\[
sage: f = -\text{sqrt}(\pi)*(x^3 + \sin(x/\cos(y)))
\]
\[
sage: \text{bool}(\text{loads}(\text{dumps}(f)) == f)
\]
\[
True
\]

Coercion examples:

We coerce various symbolic expressions into the complex numbers:

\[
sage: \text{CC}(I)
\]
\[
1.00000000000000*I
\]
\[
sage: \text{CC}(2*I)
\]
\[
2.00000000000000*I
\]
\[
sage: \text{ComplexField}(200)(2*I)
\]
\[
2.0000000000000000000000000000000000000000000000000000000000*I
\]
\[
sage: \text{ComplexField}(200)(\sin(I))
\]
\[
1.1752011936438014568823818505956008151557179813340958702296*I
\]
\[
sage: f = \sin(I) + \cos(I/2); f
\]

(continues on next page)
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:

\[
sage: f(n9=9, n7=n6)
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n6^7 + 1/n8^8 + 387420490/387420489
\]

The `sage.calculus.calculus.at` function parses 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:

\[
sage: g = x^3 - 3
sage: from sage.calculus.calculus import at
sage: at(g, x=1)
-2
\]

Wefind a formal Taylor expansion:

\[
sage: h, x = var('h, x')
sage: u = function('u')
sage: u(x + h)
u(h + x)
sage: diff(u(x+h), x)
D[0](u)(h + x)
sage: taylor(u(x+h), h, 0, 4)
1/24*h^4*diff(u(x), x, x, x, x) + 1/6*h^3*diff(u(x), x, x, x) + 1/2*h^2*diff(u(x), x, x) + h*diff(u(x), x) + u(x)
\]

We compute a Laplace transform:

\[
sage: var('s, t')
(s, t)
\]
sage: f = function('f')(t)
sage: f.diff(t, 2)
diff(f(t), t, t)
sage: f.diff(t,2).laplace(t,s)
s^2*laplace(f(t), t, s) - s*f(0) - D[0](f)(0)

We can also accept a non-keyword list of expression substitutions, like Maxima does (github issue #12796):

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

sage.calculus.calculus.dummy_diff(*args)
This function is called when 'diff' appears in a Maxima string.

EXAMPLES:

sage: from sage.calculus.calculus import dummy_diff
sage: x,y = var(x,y)
sage: dummy_diff(sin(x*y), x, SR(2), y, SR(1))
-x*y^2*cos(x*y) - 2*y*sin(x*y)

Here the function is used implicitly:

sage: a = var('a')
sage: f = function('cr')(a)
sage: g = f.diff(a); g
diff(cr(a), a)

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: 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: 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:

2.5. Symbolic Computation
sage: from sage.calculus.calculus import dummy_laplace
sage: s,t = var('s,t')

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

sage.calculus.calculus.dummy_pochhammer(*args)

This function is called to create formal wrappers of Pochhammer symbols

EXAMPLES:

sage: from sage.calculus.calculus import dummy_pochhammer
sage: s,t = var('s,t')
sage: dummy_pochhammer(s, t)

sage.calculus.calculus.inverse_laplace(ex, s, t, algorithm='maxima')

Return the inverse Laplace transform with respect to the variable \( t \) and transform parameter \( s \), if possible.

If this function cannot find a solution, a formal function is returned. The function that is returned may be viewed as a function of \( t \).

DEFINITION:

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

\[
f(t) = \frac{1}{2\pi i} \int_{\gamma - i\infty}^{\gamma + i\infty} e^{st} F(s) ds,
\]

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

INPUT:

- \( \text{ex} \) – a symbolic expression
- \( s \) – transform parameter
- \( t \) – independent variable
- \( \text{algorithm} \) – (default: 'maxima') one of
  - 'maxima' – use Maxima (the default)
  - 'sympy' – use SymPy
  - 'giac' – use Giac

See also:

laplace()

EXAMPLES:

sage: var('w, m')

(\( w, m \))
sage: f = (1/(w^2+10)).inverse_laplace(w, m); f

1/10*sqrt(10)*sin(sqrt(10)*m)
sage: laplace(f, m, w)

1/(w^2 + 10)

sage: f(t) = t*cos(t)
sage: s = var('s')
Symbolic Calculus, Release 10.3

sage: L = laplace(f, t, s); L
\texttt{t \mapsto 2s^2/(s^2 + 1)^2 - 1/(s^2 + 1)}
sage: inverse_laplace(L, s, t)
\texttt{t \mapsto t \cos(t)}
sage: inverse_laplace(1/(s^3+1), s, t)
\frac{1}{3}(\sqrt{3}\sin(1/2\sqrt{3}*t) - \cos(1/2\sqrt{3}*t))*e^{(1/2*t)} + 1/3*e^{(-t)}

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

sage: inverse_laplace(cos(s), s, t)
\texttt{ilt(cos(s), s, t)}

Transform an expression involving a time-shift, via SymPy:

sage: inverse_laplace(1/s^2*exp(-s), s, t, algorithm='sympy').simplify()
(t - 1)*heaviside(t - 1)

The same instance with Giac:

sage: inverse_laplace(1/s^2*exp(-s), s, t, algorithm='giac')
(t - 1)*heaviside(t - 1)

Transform a rational expression:

sage: inverse_laplace((2*s^2*exp(-2*s) - exp(-s))/(s^3+1), s, t,
......:     algorithm='giac')
-1/3*(sqrt(3)*e^(1/2*t - 1/2)*sin(1/2*sqrt(3)*(t - 1))
- cos(1/2*sqrt(3)*(t - 1))*e^(1/2*t - 1/2) + e^(-t + 1))*heaviside(t - 1)
+ 2/3*(2*cos(1/2*sqrt(3)*(t - 2))*e^(1/2*t - 1) + e^(-t + 2))*heaviside(t - 2)
sage: inverse_laplace(1/(s - 1), s, x)
\texttt{e^x}

The inverse Laplace transform of a constant is a delta distribution:

sage: inverse_laplace(1, s, t)
dirac_delta(t)
sage: inverse_laplace(1, s, t, algorithm='sympy')
dirac_delta(t)
sage: inverse_laplace(1, s, t, algorithm='giac')
dirac_delta(t)

sage.calculus.calculus.\texttt{laplace}(ex, t, s, algorithm='maxima')

Return the Laplace transform with respect to the variable \( t \) and transform parameter \( s \), if possible.

If this function cannot find a solution, a formal function is returned. The function that is returned may be viewed as a function of \( s \).

DEFINITION:

The Laplace transform of a function \( f(t) \), defined for all real numbers \( t \geq 0 \), is the function \( F(s) \) defined by

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

INPUT:

\begin{itemize}
  \item \texttt{ex} – a symbolic expression
\end{itemize}
• \( t \) – independent variable
• \( s \) – transform parameter
• algorithm – (default: 'maxima') one of
  – 'maxima' – use Maxima (the default)
  – 'sympy' – use SymPy
  – 'giac' – use Giac

Note: The 'sympy' algorithm returns the tuple \((F, a, \text{cond})\) where \(F\) is the Laplace transform of \(f(t)\), \(\Re(s) > a\) is the half-plane of convergence, and \(\text{cond}\) are auxiliary convergence conditions.

See also:

\textit{inverse_laplace()}

EXAMPLES:

We compute a few Laplace transforms:

\begin{verbatim}
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
\end{verbatim}

We do a formal calculation:

\begin{verbatim}
sage: f = function('f')(x)
sage: g = f.diff(x); g
diff(f(x), x)
sage: g.laplace(x, s)
s*laplace(f(x), x, s) - f(0)
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)
\end{verbatim}

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.

\begin{verbatim}
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)
\end{verbatim}
Next we form the augmented matrix of the above system:

\[
\begin{bmatrix}
s & 16 & 270 \\
1 & s & 90 + 1/s
\end{bmatrix}
\]

\[
\begin{align*}
sage: & A = \text{matrix}([[s, 16, 270], [1, s, 90+1/s]]) \\
sage: & E = A.echelon_form() \\
sage: & xt = E[0,2].inverse_laplace(s,t) \\
sage: & yt = E[1,2].inverse_laplace(s,t) \\
sage: & xt = -91/2*e^(4*t) + 629/2*e^(-4*t) + 1 \\
sage: & yt = 91/8*e^(4*t) + 629/8*e^(-4*t)
\end{align*}
\]

\[
\begin{align*}
sage: & p1 = \text{plot}(xt, 0, 1/2, \text{rgbcolor}=(1,0,0)) \\
sage: & p2 = \text{plot}(yt, 0, 1/2, \text{rgbcolor}=(0,1,0))
\end{align*}
\]

Another example:

\[
\begin{align*}
sage: & \text{var}('a,s,t') \\
& (a, s, t) \\
sage: & f = \exp(2*t + a) * \sin(t) * t; f \\
& t*e^(a + 2*t)*\sin(t) \\
sage: & L = \text{laplace}(f, t, s); L \\
& 2*(s - 2)*e^a/(s^2 - 4*s + 5)^2 \\
sage: & \text{inverse_laplace}(L, s, t) \\
& t*e^(a + 2*t)*\sin(t)
\end{align*}
\]

The Laplace transform of the exponential function:

\[
\begin{align*}
sage: & \text{laplace}(\exp(x), x, s) \\
& 1/(s - 1)
\end{align*}
\]

Dirac’s delta distribution is handled (the output of SymPy is related to a choice that has to be made when defining Laplace transforms of distributions):

\[
\begin{align*}
sage: & \text{laplace(dirac_delta(t), t, s)} \\
& 1 \\
sage: & F, a, cond = \text{laplace(dirac_delta(t), t, s, algorithm='sympy')} \\
sage: & a, cond # random - sympy <1.10 gives (-oo, True) \\
& (0, True) \\
sage: & F # random - sympy <1.9 includes undefined heaviside(0) in answer \\
& 1 \\
sage: & \text{laplace(dirac_delta(t), t, s, algorithm='giac')} \\
& 1
\end{align*}
\]

Heaviside step function can be handled with different interfaces. Try with Maxima:

\[
\begin{align*}
sage: & \text{laplace}(\text{heaviside(t-1)}, t, s) \\
& e^(-s)/s
\end{align*}
\]

Try with giac:

\[
\begin{align*}
sage: & \text{laplace}(\text{heaviside(t-1)}, t, s, \text{algorithm='giac'}) \\
& e^(-s)/s
\end{align*}
\]

Try with SymPy:

\[
\begin{align*}
sage: & \text{laplace}(\text{heaviside(t-1)}, t, s, \text{algorithm='sympy'}) \\
& e^(-s)/s
\end{align*}
\]

2.5. Symbolic Computation 181
\texttt{sage: laplace(heaviside(t-1), t, s, algorithm='sympy')} \\
\begin{verbatim}
(e^(-s)/s, 0, True)
\end{verbatim}

\texttt{sage.calculus.calculus.lim(\texttt{expr}, \texttt{dir=\texttt{None}}, \texttt{taylor=\texttt{False}}, \texttt{algorithm=\texttt{\textquotesingle maxima\textquotesingle}}, \texttt{\texttt{**argv}})}

Return the limit as the variable \(v\) approaches \(a\) from the given direction.

\begin{verbatim}
expr.limit(x = a) \\
expr.limit(x = a, dir='+')
\end{verbatim}

**INPUT:**

- \texttt{dir} – (default: \texttt{None}); may have the value 'plus' (or '+' or 'right' or 'above') for a limit from above, 'minus' (or '-' or 'left' or 'below') for a limit from below, or may be omitted (implying a two-sided limit is to be computed).

- \texttt{taylor} – (default: \texttt{False}); if \texttt{True}, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).

- \texttt{\texttt{**argv}} - 1 named parameter

**Note:** The output may also use \texttt{und} (undefined), \texttt{ind} (indefinite but bounded), and \texttt{infinity} (complex infinity).

**EXAMPLES:**

\begin{verbatim}
sage: x = var('x') 
\texttt{sage: f} = (1 + 1/x)^x 
\texttt{sage: f.limit(x=\infty)} 
\begin{verbatim}
e
\texttt{sage: f.limit(x=5)} 
7776/3125
\end{verbatim}
\end{verbatim}

Domain to real, a regression in 5.46.0, see https://sf.net/p/maxima/bugs/4138

\begin{verbatim}
sage: maxima_calculus.eval("domain:real") 
... 
\texttt{sage: f.limit(x=1.2).n()} 
2.06961575467... 
\texttt{sage: maxima_calculus.eval("domain:complex"); 
...}
\end{verbatim}

Otherwise, it works

\begin{verbatim}
sage: f.limit(x=I, taylor=\texttt{True}) 
(-I + 1)^I
\texttt{sage: f(x=1.2)} 
2.0696157546720... 
\texttt{sage: f(x=I)} 
(-I + 1)^I
\texttt{sage: CDF(f(x=I))} 
2.0628722350809046 + 0.7450070621797239*I 
\texttt{sage: CDF(f.limit(x=I))} 
2.0628722350809046 + 0.7450070621797239*I
\end{verbatim}

Notice that Maxima may ask for more information:
With this example, Maxima is looking for a LOT of information:

```
sage: assume(a>0)
sage: limit(x^a, x=0)  # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
'may' help (example of legal syntax is 'assume(a>0)', see
'assume?' for more details)
Is a an integer?
sage: assume(a,'integer')
sage: limit(x^a, x=0)  # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation 'may' help
(example of legal syntax is 'assume(a>0)', see 'assume?' for
more details)
Is a an even number?
sage: assume(a, 'even')
sage: limit(x^a, x=0)
0
sage: forget()
```

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)
```

(continues on next page)
Symbolic Calculus, Release 10.3

```
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

We can use other packages than maxima, namely “sympy”, “giac”, “fricas”.

With the standard package Giac:

```
sage: from sage.libs.giac.giac import libgiac          # random
sage: (exp(-x)/(2+sin(x))).limit(x=oo, algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='right', algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='giac')
+Infinity
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

With the optional package FriCAS:

```
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='fricas')  # _
                            →optional - fricas
1
sage: limit(e^(-1/x), x=0, dir='right', algorithm='fricas')       #_
                            →optional - fricas
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='fricas')        #_
                            →optional - fricas
+Infinity
```

One can also call Mathematica’s online interface:

```
sage: limit(pi+log(x)/x,x=oo, algorithm='mathematica_free')    # optional - internet
pi
```

```
Return the limit as the variable \( v \) approaches \( a \) from the given direction.

expr.limit(x = a)
expr.limit(x = a, dir='+')
```

INPUT:

- \( \text{dir} \) (default: None); may have the value 'plus' (or '+') or 'right' or 'above') for a limit from above, 'minus' (or '-' or 'left' or 'below') for a limit from below, or may be omitted (implying a
two-sided limit is to be computed).

- `taylor` – (default: `False`): if `True`, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `**argv - 1` named parameter

**Note:** The output may also use `und` (undefined), `ind` (indefinite but bounded), and `infinity` (complex infinity).

**EXAMPLES:**

```sage
sage: x = var('x')
sage: f = (1 + 1/x)^x
e
sage: f.limit(x=oo)
e
sage: f.limit(x=5)
7776/3125
```

Domain to real, a regression in 5.46.0, see [https://sf.net/p/maxima/bugs/4138](https://sf.net/p/maxima/bugs/4138)

```sage
sage: maxima_calculus.eval("domain:real")
...
sage: f.limit(x=1.2).n()
2.06961575467...
sage: maxima_calculus.eval("domain:complex");
...
```

Otherwise, it works

```sage
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
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:
Symbolic Calculus, Release 10.3

```python
sage: assume(a>0)
sage: limit(x^a, x=0) # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints; using the 'assume' command before evaluation *may* help (example of legal syntax is 'assume(a>0)', see 'assume?' for more details)
Is a an integer?
sage: assume(a,'integer')
sage: limit(x^a, x=0) # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints; using the 'assume' command before evaluation *may* help (example of legal syntax is 'assume(a>0)', see 'assume?' for more details)
Is a an even number?
sage: assume(a, 'even')
sage: limit(x^a, x=0)
0
sage: forget()
```

More examples:

```python
sage: limit(x*log(x), x=0, dir='+')
0
sage: limit((x+1)^(1/x), x=0)
e
sage: limit(e^x/x, x=oo)
+Infinity
sage: limit(e^x/x, x=-oo)
0
sage: limit(-e^x/x, x=oo)
-Infinity
sage: limit((cos(x))/(x^2), x=0)
+Infinity
sage: limit(sqrt(x^2+1) - x, x=oo)
0
sage: limit(x^2/(sec(x)-1), x=0)
2
sage: limit(cos(x)/(cos(x)-1), x=0)
-Infinity
sage: limit(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
```

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

Here ind means “indefinite but bounded”:
We can use other packages than maxima, namely “sympy”, “giac”, “fricas”.

With the standard package Giac:

```python
sage: from sage.libs.giac.giac import libgiac   # random
sage: (exp(-x)/(2+sin(x))).limit(x=oo, algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='right', algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='giac')
+Infinity
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

With the optional package FriCAS:

```python
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='fricas')  #--
1
sage: limit(e^(-1/x), x=0, dir='right', algorithm='fricas')  #--
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='fricas')  #--
+Infinity
```

One can also call Mathematica’s online interface:

```python
sage: limit(pi+log(x)/x,x=oo, algorithm='mathematica_free')  # optional - internet
pi
```

```
2.5. Symbolic Computation

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:

```python
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:

```python
```
Return the minimal polynomial of self, if possible.

INPUT:

- var – polynomial variable name (default 'x')
- algorithm – 'algebraic' or 'numerical' (default both, but with numerical first)
- bits – the number of bits to use in numerical approx
- degree – the expected algebraic degree
- epsilon – return without error as long as f(self) < epsilon, in the case that the result cannot be proven.

All of the above parameters are optional, with epsilon=0, bits and degree tested up to 1000 and 24 by default respectively. The numerical algorithm will be faster if bits and/or degree are given explicitly. The algebraic algorithm ignores the last three parameters.

OUTPUT: The minimal polynomial of self. If the numerical algorithm is used, then it is proved symbolically when epsilon=0 (default).

If the minimal polynomial could not be found, two distinct kinds of errors are raised. If no reasonable candidate was found with the given bits/degree parameters, a ValueError will be raised. If a reasonable candidate was found but (perhaps due to limits in the underlying symbolic package) was unable to be proved correct, a NotImplementedError will be raised.

ALGORITHM: Two distinct algorithms are used, depending on the algorithm parameter. By default, the numerical algorithm is attempted first, then the algebraic one.

Algebraic: Attempt to evaluate this expression in QQbar, using cyclotomic fields to resolve exponential and trig functions at rational multiples of \( \pi \), field extensions to handle roots and rational exponents, and computing composites 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 \texttt{parialgdep} to get a candidate minpoly \( f \). If \( f(\text{self}) \), evaluated to a higher precision, is close enough to 0 then evaluate \( f(\text{self}) \) symbolically, attempting to prove vanishing. If this fails, and epsilon is non-zero, return \( f \) if and only if \( f(\text{self}) < \epsilon \). 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
```

(continues on next page)
Here we verify it gives the same result as the abstract number field.

```
sage: (sqrt(2) + sqrt(3) + sqrt(6)).minpoly()
```

\[ x^4 - 22x^2 - 48x - 23 \]

Here we solve a cubic and then recover it from its complicated radical expansion.

```
sage: f = x^3 - x + 1
sage: a = f.solve(x)[0].rhs(); a
-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1)
- 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)
```

```
sage: a.minpoly()
```

\[ x^3 - x + 1 \]

Note that simplification may be necessary to see that the minimal polynomial is correct.

```
sage: a = sqrt(2)+sqrt(3)+sqrt(5)
sage: f = a.minpoly(); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
```

```
sage: f(a)
```

\[ (\sqrt{5} + \sqrt{3} + \sqrt{2})^8 - 40*(\sqrt{5} + \sqrt{3} + \sqrt{2})^6 \
+ 352*(\sqrt{5} + \sqrt{3} + \sqrt{2})^4 - 960*(\sqrt{5} + \sqrt{3} + \sqrt{2})^2 \
+ 576 \]

```
sage: f(a).expand()
```

\[ 0 \]

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):

...
Symbolic Calculus, Release 10.3

ValueError: Could not find minimal polynomial (100 bits, degree 3).

```sage```
```
```
```sage```
```
```
```sage```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```sage```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
- desired_relative_error – (default: 1e-8) the desired relative error
- maximum_num_subintervals – (default: 200) maximal number of subintervals

OUTPUT:
- float: approximation to the integral
- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - 0 – no problems were encountered
  - 1 – too many subintervals were done
  - 2 – excessive roundoff error
  - 3 – extremely bad integrand behavior
  - 4 – failed to converge
  - 5 – integral is probably divergent or slowly convergent
  - 6 – the input is invalid; this includes the case of desired_relative_error being too small to be achieved

ALIAS: nintegrate() is the same as nintegral()

REMARK: There is also a function numerical_integral() that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral due to limitations in quadpack. In the following example, remark that the last value of the returned tuple is 6, indicating that the input was invalid, in this case because of a too high desired precision.

```
sage: f = x
sage: f.nintegral(x, 0, 1, 1e-14)
(0.0, 0.0, 0, 6)
```

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)
-7.499274028018143112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.499274028018143112064614366622348652078895136533593355718e-13
```

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

```
sage: f.nintegrate(x,0,1)
(-480.000000000000..., 5.32907051820075...e-12, 21, 0)
```

It is just because every floating point evaluation of \( f \) returns \(-480.0\) in floating point.

Important note: using PARI/GP one can compute numerical integrals to high precision:

```
sage: gp.eval(intnum(x=17,42,exp(-x^2)*log(x)))
2.56572850056105147934096410 E-127 # 32-bit
2.5657285005610514829176211363206621657 E-127 # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval(intnum(x=17,42,exp(-x^2)*log(x)))
2.5657285005610514829173563961304957417746108003917 E-127
sage: gp.set_real_precision(old_prec)
```

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 \( \text{self} \) 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
- \( \text{desired\_relative\_error} \) – (default: 1e-8) the desired relative error
- \( \text{maximum\_num\_subintervals} \) – (default: 200) maximal number of subintervals

**OUTPUT:**

- float: approximation to the integral
- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - 0 – no problems were encountered
  - 1 – too many subintervals were done
  - 2 – excessive roundoff error
  - 3 – extremely bad integrand behavior
  - 4 – failed to converge
5. integral is probably divergent or slowly convergent
6. the input is invalid; this includes the case of desired_relative_error being too small to be achieved

ALIAS: nintegrate() is the same as nintegral()

REMARK: There is also a function numerical_integral() that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral due to limitations in quadpack. In the following example, remark that the last value of the returned tuple is 6, indicating that the input was invalid, in this case because of a too high desired precision.

```
sage: f = x
sage: f.nintegral(x, 0, 1, 1e-14)
(0.0, 0.0, 0, 6)
```

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)
(0.5284822232253147, 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)
-7.49927402801814311206461436662234865207889513653359355718e-13
sage: f.n(300)
-7.49927402801814311206461436662663009137292462589621789352095066181709095575681963967103004e-13
```

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

```
sage: f.nintegrate(x,0,1)
(-480.0000000000000000..., 5.32907051820075...e-12, 21, 0)
```

It is just because every floating point evaluation of \( f \) returns \(-480.0\) in floating point.

Important note: using PARI/GP one can compute numerical integrals to high precision:

```
2.5. Symbolic Computation

193
```
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
2.565728500561051474934096410 E-127  # 32-bit
2.5657285005610514829176211363206621657 E-127  # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
2.5657285005610514829173563961304957417746108003917 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 copy of 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('?@f(x),x=2)#1')
f(2) != 1
sage: a = sage.calculus.calculus.maxima("x#0"); a
x # 0
sage: a.sage()
x != 0

sage.calculus.calculus.symbolic_expression_from_string(s, syms={}, accept_sequence=None, parser=False)

Given a string, (attempt to) parse it and return the corresponding Sage symbolic expression. Normally used to return Maxima output to the user.

INPUT:

- s – a string
- syms – (default: {}) dictionary of strings to be regarded as symbols or functions; keys are pairs (string, number of arguments)
- accept_sequence – (default: False) controls whether to allow a (possibly nested) set of lists and tuples as input
- parser – (default: SR_parser) parser for internal use

EXAMPLES:
Return the symbolic product ∏_{v=a}^{b} expression with respect to the variable v with endpoints a and b.

INPUT:

• expression – a symbolic expression
• v – a variable or variable name
• a – lower endpoint of the product
• b – upper endpoint of the product
• algorithm – (default: 'maxima') one of
  – 'maxima' – use Maxima (the default)
  – 'giac' – use Giac
  – 'sympy' – use SymPy
  – 'mathematica' – (optional) use Mathematica
• hold - (default: False) if True, don’t evaluate

EXAMPLES:

```
sage: i, k, n = var('i,k,n')
sage: from sage.calculus.calculus import symbolic_product
sage: symbolic_product(k, k, 1, n)
factorial(n)
sage: symbolic_product(x + i*(i+1)/2, i, 1, 4)
x^4 + 20*x^3 + 127*x^2 + 288*x + 180
sage: symbolic_product(i^2, i, 1, 7)
25401600
sage: f = function('f')
sage: symbolic_product(f(i), i, 1, 7)
f(7)*f(6)*f(5)*f(4)*f(3)*f(2)*f(1)
sage: symbolic_product(f(i), i, 1, n)
product(f(i), i, 1, n)
sage: assume(k>0)
sage: symbolic_product(integrate (x^k, x, 0, 1), k, 1, n)
1/factorial(n + 1)
sage: symbolic_product(f(i), i, 1, n).log().log_expand()
sum(log(f(i)), i, 1, n)
```
Symbolic Calculus, Release 10.3

- 'maxima' – use Maxima (the default)
- 'maple' – (optional) use Maple
- 'mathematica' – (optional) use Mathematica
- 'giac' – (optional) use Giac
- 'sympy' – use SymPy

• hold – (default: False) if True, don’t evaluate

EXAMPLES:

```python
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:

```python
sage: symbolic_sum(binomial(n,k), k, 0, n)  
2^n

And some truncations thereof:

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

The binomial theorem:

```python
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:
Symbolic Calculus, Release 10.3

\[
\text{sage: symbolic_sum(binomial(n, k) \ast factorial(k) / factorial(n+1+k), k, 0, n)}
\]
\[
1/2*sqrt(pi)/factorial(n + 1/2)
\]

We check a well known identity:

\[
\text{sage: bool(symbolic_sum(k^3, k, 1, n) == symbolic_sum(k, k, 1, n)^2)}
\]
\[
True
\]

A geometric sum:

\[
\text{sage: a, q = var('a, q')}
\]
\[
\text{sage: symbolic_sum(a\ast q^k, k, 0, n)}
\]
\[
(a\ast q^{n + 1} - a)/(q - 1)
\]

For the geometric series, we will have to assume the right values for the sum to converge:

\[
\text{sage: assume(abs(q) < 1)}
\]
\[
\text{sage: symbolic_sum(a\ast q^k, k, 0, oo)}
\]
\[
-a/(q - 1)
\]

A divergent geometric series. Don't forget to forget your assumptions:

\[
\text{sage: forget()}
\]
\[
\text{sage: assume(q > 1)}
\]
\[
\text{sage: symbolic_sum(a\ast q^k, k, 0, oo)}
\]
\[
\text{Traceback (most recent call last):}
\]
\[
... ValueError: Sum is divergent.
\]
\[
\text{sage: forget()}
\]
\[
\text{sage: assumptions()} \quad \# \text{check the assumptions were really forgotten}
\]
\[
[]
\]

A summation performed by Mathematica:

\[
\text{sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='mathematica')} \quad \# \text{optional...}
\]
\[
\text{← mathematica}
\]
\[
\text{pi*coth(pi)}
\]

An example of this summation with Giac:

\[
\text{sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='giac').factor()}
\]
\[
\pi*(e^{2*pi} + 1)/((e^pi + 1)*(e^pi - 1))
\]

The same summation is solved by SymPy:

\[
\text{sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='sympy')}\]
\[
\text{pi/tanh(pi)}
\]

SymPy and Maxima 5.39.0 can do the following (see github issue #22005):

\[
\text{sage: sum(1/((2*n+1)^2-4)^2, n, 0, Infinity, algorithm='sympy')}
\]
\[
1/64*pi^2
\]
\[
\text{sage: sum(1/((2*n+1)^2-4)^2, n, 0, Infinity)}
\]
\[
1/64*pi^2
\]

Use Maple as a backend for summation:

2.5. Symbolic Computation 197
Symbolic Calculus, Release 10.3

```
sage: symbolic_sum(binomial(n,k)*x^k, k, 0, n, algorithm='maple')  # optional␣→ maple
(x + 1)^n
```

If you don’t want to evaluate immediately give the hold keyword:

```
sage: s = sum(n, n, 1, k, hold=True); s
sum(n, n, 1, k)
sage: s.unhold()
1/2*k^2 + 1/2*k
sage: s.subs(k == 10)
sum(n, n, 1, 10)
sage: s.subs(k == 10).unhold()
55
sage: s.subs(k == 10).n()
55.0000000000000
```

Note: Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertible into a Sage expression.

### 2.6 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.__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:

```python
sage: t.convert()
1/100000 * kilogram * meter / second^2
```

Giving improper units to convert to raises a `ValueError`:

```python
sage: t.convert(units.charge.coulomb)
Traceback (most recent call last):
  ... 
ValueError: Incompatible units
```

Converting temperatures works as well:

```python
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`:

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

AUTHORS:

- David Ackerman
- William Stein

```python
class sage.symbolic.units.UnitExpression
    Bases: Expression
    A symbolic unit.

EXAMPLES:

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

```python
class sage.symbolic.units.Units(data, name="")
    Bases: ExtraTabCompletion
    A collection of units of some type.

EXAMPLES:

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

```python
sage.symbolic.units.base_units(unit)
    Converts unit to base SI units.

INPUT:

- unit -- a unit
```
OUTPUT:

• a symbolic expression

EXAMPLES:

```python
sage: sage.symbolic.units.base_units(units.length.foot)
381/1250*meter
```

If unit is already a base unit, it just returns that unit:

```python
sage: sage.symbolic.units.base_units(units.length.meter)
meter
```

Derived units get broken down into their base parts:

```python
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:

```python
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:

• a symbolic expression

EXAMPLES:

```python
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:

```python
sage: sage.symbolic.units.convert(units.mass.kilogram, units.length.foot)
Traceback (most recent call last):
... ValueError: Incompatible units
```

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

sage: sage.symbolic.units.convert(units.length.foot/units.time.second^2, units.acceleration.galileo)
762/25*galileo

sage: sage.symbolic.units.convert(units.mass.kilogram*units.length.meter/units.time.second^2, units.force.newton)
newton

sage: sage.symbolic.units.convert(units.length.foot^3, units.area.acre*units.length.inch)
1/3630*(acre*inch)

sage: sage.symbolic.units.convert(units.charge.coulomb, units.current.ampere*units.time.second)
(ampere*second)

sage: sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo, units.pressure.pounds_per_square_inch)
1290320000000/8896443230521*pounds_per_square_inch

For decimal answers multiply 1.0:

sage: sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo, units.pressure.pounds_per_square_inch)*1.0
0.145037737730209*pounds_per_square_inch

You can also convert quantities of units:

sage: sage.symbolic.units.convert(cos(50) * units.angles.radian, units.angles.degree)
degree*(180*cos(50)/pi)

sage: sage.symbolic.units.convert(cos(30) * units.angles.radian, units.angles.degree).polynomial(RR)
8.83795706233228*degree

sage: sage.symbolic.units.convert(50 * units.length.light_year / units.time.year, units.length.foot / units.time.second)
6249954068750/127*(foot/second)

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:

• a symbolic expression

EXAMPLES:

sage: t = 32*units.temperature.fahrenheit
sage: t.convert(units.temperature.celsius)
0
sage: t.convert(units.temperature.kelvin)
273.1500000000000*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.fahrenheit)
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)`

Return a boolean when asked whether the input is in the list of units.

INPUT:

• s – an object

OUTPUT:

• a boolean

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 – a string

OUTPUT:
• a UnitExpression

EXAMPLES:

```python
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 – a string, name of a unit type such as ‘area’, ‘volume’, etc.

OUTPUT:
• a symbolic expression

EXAMPLES:

```python
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:

```python
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 – a unit

OUTPUT:
• a string

EXAMPLES:
symbolic: sage.symbolic.units.unitdocs('meter')
'SI base unit of length.
Defined to be the distance light travels in vacuum in 1/
→299792458 of a second.'
sage: sage.symbolic.units.unitdocs('amu')
'Abbreviation for atomic mass unit.
Approximately 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 – a string

OUTPUT:
• a list of strings (unit names)

EXAMPLES:
sage: sage.symbolic.units.vars_in_str('mass/(length*time)')
['mass', 'length', 'time']

2.7 The symbolic ring

class sage.symbolic.ring.NumpyToSRMorphism
Bases: Morphism

A morphism from numpy types to the symbolic ring.
class sage.symbolic.ring.SymbolicRing
Bases: SymbolicRing

Symbolic Ring, parent object for all symbolic expressions.

I()
The imaginary unit, viewed as an element of the symbolic ring.

EXAMPLES:
sage: SR.I()^2
-1
sage: SR.I().parent()
Symbolic Ring

characteristic()
Return the characteristic of the symbolic ring, which is 0.

OUTPUT:
• a Sage integer
EXAMPLES:

```
sage: c = SR.characteristic(); c
0
sage: type(c)
<class 'sage.rings.integer.Integer'>
```

cleanup_var (symbol)

Cleans up a variable, removing assumptions about the variable and allowing for it to be garbage collected

INPUT:

- symbol – a variable or a list of variables

is_exact ()

Return False, because there are approximate elements in the symbolic ring.

EXAMPLES:

```
sage: SR.is_exact()
False
```

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
```

is_finite ()

Return False, since the Symbolic Ring is infinite.

EXAMPLES:

```
sage: SR.is_finite()
False
```

pi ()

EXAMPLES:

```
sage: SR.pi() is pi
True
```

subring (*args, **kwds)

Create a subring of this symbolic ring.

INPUT:

Choose one of the following keywords to create a subring.

- accepting_variables (default: None) – a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in only these variables is created.
Symbolic Calculus, Release 10.3

- rejecting_variables (default: None) – a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in variables distinct to these variables is created.
- no_variables (default: False) – a boolean. If set, then a symbolic subring of constant expressions (i.e., expressions without a variable) is created.

OUTPUT:
A ring.

EXAMPLES:
Let us create a couple of symbolic variables first:

\begin{verbatim}
sage: V = var('a, b, r, s, x, y')
\end{verbatim}

Now we create a symbolic subring only accepting expressions in the variables \(a\) and \(b\):

\begin{verbatim}
sage: A = SR.subring(accepting_variables=(a, b)); A
Symbolic Subring accepting the variables a, b
\end{verbatim}

An element is

\begin{verbatim}
sage: A.an_element()
a
\end{verbatim}

From our variables in \(V\) the following are valid in \(A\):

\begin{verbatim}
sage: tuple(v for v in V if v in A)
(a, b)
\end{verbatim}

Next, we create a symbolic subring rejecting expressions with given variables:

\begin{verbatim}
sage: R = SR.subring(rejecting_variables=(r, s)); R
Symbolic Subring rejecting the variables r, s
\end{verbatim}

An element is

\begin{verbatim}
sage: R.an_element()
some_variable
\end{verbatim}

From our variables in \(V\) the following are valid in \(R\):

\begin{verbatim}
sage: tuple(v for v in V if v in R)
(a, b, x, y)
\end{verbatim}

We have a third kind of subring, namely the subring of symbolic constants:

\begin{verbatim}
sage: C = SR.subring(no_variables=True); C
Symbolic Constants Subring
\end{verbatim}

Note that this subring can be considered as a special accepting subring; one without any variables.

An element is

\begin{verbatim}
sage: C.an_element()
I*pi*e
\end{verbatim}

None of our variables in \(V\) is valid in \(C\):
See also:
Subrings of the Symbolic Ring

symbol (name=None, latex_name=None, domain=None)

EXAMPLES:

```
sage: t0 = SR.symbol("t0")
sage: t0.conjugate()
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...
```

We propagate the domain to the assumptions database:

```
sage: n = var('n', domain='integer')
sage: solve([n^2 == 3],n)
[]
```

symbols
temp_var (n=None, domain=None)

Return one or multiple new unique symbolic variables as an element of the symbolic ring. Use this instead of SR.var() if there is a possibility of name clashes occurring. Call SR.cleanup_var() once the variables are no longer needed or use a with SR.temp_var() as... construct.

INPUT:

• n – (optional) positive integer; number of symbolic variables
• domain – (optional) specify the domain of the variable(s);

EXAMPLES:

Simple definition of a functional derivative:

```
sage: def functional_derivative(expr,f,x):
    ....:     with SR.temp_var() as a:
    ....:         return expr.subs({f(x):a}).diff(a).subs({a:f(x)})
sage: f = function('f')
```

(continues on next page)
Contrast this to a similar implementation using SR.var(), which gives a wrong result in our example:

```python
sage: def functional_derivative(expr,f,x):
    a = SR.var('a')
    return expr.subs({f(x):a}).diff(a).subs({a:f(x)})

sage: f = function('f')

sage: functional_derivative(f(a)^2+a,f,a)
2*f(a) + 1
```

```
var(name, latex_name=None, n=None, domain=None)
```

Return a symbolic variable as an element of the symbolic ring.

**INPUT:**

- name – string or list of strings with the name(s) of the symbolic variable(s)
- latex_name – (optional) string used when printing in latex mode, if not specified use 'name'
- n – (optional) positive integer; number of symbolic variables, indexed from 0 to $n - 1$
- domain – (optional) specify the domain of the variable(s); it is None by default, and possible options are (non-exhaustive list, see note below): 'real', 'complex', 'positive', 'integer' and 'noninteger'

**OUTPUT:**

Symbolic expression or tuple of symbolic expressions.

**See also:**

This function does not inject the variable(s) into the global namespace. For that purpose see var().

**Note:** For a comprehensive list of acceptable features type 'maxima('features')', and see also the documentation of Assumptions.

**EXAMPLES:**

Create a variable `zz`:

```python
sage: zz = SR.var('zz'); zz
```

The return type is a symbolic expression:

```python
sage: type(zz)
<class 'sage.symbolic.expression.Expression'>
```

We can specify the domain as well:

```python
sage: zz = SR.var('zz', domain='real')
sage: zz.is_real()
True
```

The real domain is also set with the integer domain:
Symbolic Calculus, Release 10.3

```python
sage: SR.var('x', domain='integer').is_real()
True
```

The `name` argument does not have to match the left-hand side variable:

```python
sage: t = SR.var('theta2'); t
theta2
```

Automatic indexing is available as well:

```python
sage: x = SR.var('x', 4)
sage: x[0], x[3]
(x0, x3)
sage: sum(x)
x0 + x1 + x2 + x3
```

**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:**

```python
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.TemporaryVariables` *(iterable=(),/)*

**Bases:** tuple

Instances of this class can be used with Python with to automatically clean up after themselves.

**class** `sage.symbolic.ring.UnderscoreSageMorphism`

**Bases:** Morphism

A Morphism which constructs Expressions from an arbitrary Python object by calling the `_sage_` method on the object.

**EXAMPLES:**

```python
sage: # needs sympy
sage: import sympy
sage: from sage.symbolic.ring import UnderscoreSageMorphism
sage: b = sympy.var('b')
sage: f = UnderscoreSageMorphism(type(b), SR)
sage: f(b)
```

(continues on next page)
sage.symbolic.ring.isidentifier(x)

Return whether \( x \) is a valid identifier.

**INPUT:**

- \( x \) – a string

**OUTPUT:**

Boolean. Whether the string \( x \) can be used as a variable name.

This function should return `False` for keywords, so we can not just use the `isidentifier` method of strings, because, for example, it returns `True` for “def” and for “None”.

**EXAMPLES:**

```
sage: from sage.symbolic.ring import isidentifier
sage: isidentifier('x')
True
sage: isidentifier(' x')  # can't start with space
False
sage: isidentifier('ceci_n_est_pas_une_pipe')
True
sage: isidentifier('1 + x')
False
sage: isidentifier('2good')
False
sage: isidentifier('good2')
True
sage: isidentifier('lambda s:s+1')
False
sage: isidentifier('None')
False
sage: isidentifier('lambda')
False
sage: isidentifier('def')
False
```

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:**
```python
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
```

2.8 Subrings of the Symbolic Ring

Subrings of the symbolic ring can be created via the `subring()` method of `SR`. This will call `SymbolicSubring` of this module.

The following kinds of subrings are supported:

- A symbolic subring of expressions, whose variables are contained in a given set of symbolic variables (see `SymbolicSubringAcceptingVars`). E.g.
  ```python
  sage: SR.subring(accepting_variables=('a', 'b'))
  Symbolic Subring accepting the variables a, b
  ```

- A symbolic subring of expressions, whose variables are disjoint to a given set of symbolic variables (see `SymbolicSubringRejectingVars`). E.g.
  ```python
  sage: SR.subring(rejecting_variables=('r', 's'))
  Symbolic Subring rejecting the variables r, s
  ```

- The subring of symbolic constants (see `SymbolicConstantsSubring`). E.g.
  ```python
  sage: SR.subring(no_variables=True)
  Symbolic Constants Subring
  ```

AUTHORS:

- Daniel Krenn (2015)

2.8.1 Classes and Methods

```python
class sage.symbolic.subring.GenericSymbolicSubring(vars)
    Bases: SymbolicRing
    An abstract base class for a symbolic subring.
    INPUT:
    • vars -- a tuple of symbolic variables.

has_valid_variable(variable)
    Return whether the given variable is valid in this subring.
    INPUT:
    • variable -- a symbolic variable.
```
OUTPUT:
A boolean.

EXAMPLES:
```
sage: from sage.symbolic.subring import GenericSymbolicSubring
c Sage: GenericSymbolicSubring(vars=tuple()).has_valid_variable(x)
Traceback (most recent call last):
... 
NotImplementedError: Not implemented in this abstract base class
```

class sage.symbolic.subring.GenericSymbolicSubringFunctor(vars)
Bases: ConstructionFunctor
A base class for the functors constructing symbolic subrings.

INPUT:
• vars – a tuple, set, or other iterable of symbolic variables.

EXAMPLES:
```
sage: from sage.symbolic.subring import SymbolicSubring
sage: SymbolicSubring(no_variables=True).construction()[0] # indirect doctest
Subring<accepting no variable>
```

See also:
sage.categories.pushout.ConstructionFunctor.

coercion_reversed = True

merge(other)
Merge this functor with other if possible.

INPUT:
• other – a functor.

OUTPUT:
A functor or None.

EXAMPLES:
```
sage: from sage.symbolic.subring import SymbolicSubring
sage: F = SymbolicSubring(accepting_variables=('a',)).construction()[0]
sage: F.merge(F) is F
True
```

rank = 11

class sage.symbolic.subring.SymbolicConstantsSubring(vars)
Bases: SymbolicSubringAcceptingVars
The symbolic subring consisting of symbolic constants.

has_valid_variable(variable)
Return whether the given variable is valid in this subring.

INPUT:
• variable – a symbolic variable.
OUTPUT:
A boolean.

EXAMPLES:

```python
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(no_variables=True)
sage: S.has_valid_variable('a')
False
sage: S.has_valid_variable('r')
False
sage: S.has_valid_variable('x')
False
```

```python
class sage.symbolic.subring.SymbolicSubringAcceptingVars(var)
    Bases: GenericSymbolicSubring
    The symbolic subring consisting of symbolic expressions in the given variables.
    construction()
        Return the functorial construction of this symbolic subring.
        OUTPUT:
        A tuple whose first entry is a construction functor and its second is the symbolic ring.
        EXAMPLES:
        ```python
        sage: from sage.symbolic.subring import SymbolicSubring
        sage: S = SymbolicSubring(accepting_variables=('a',)).construction()
        '(Subring<accepting a>, Symbolic Ring)
        ```
    has_valid_variable(variable)
        Return whether the given variable is valid in this subring.
        INPUT:
        • variable – a symbolic variable.
        OUTPUT:
        A boolean.
        EXAMPLES:
        ```python
        sage: from sage.symbolic.subring import SymbolicSubring
        sage: S = SymbolicSubring(accepting_variables=('a',))
        sage: S.has_valid_variable('a')
        True
        sage: S.has_valid_variable('r')
        False
        sage: S.has_valid_variable('x')
        False
        ```
```

```python
class sage.symbolic.subring.SymbolicSubringAcceptingVarsFunctor(var)
    Bases: GenericSymbolicSubringFunctor
    merge(other)
        Merge this functor with other if possible.
        INPUT:
```
Symbolic Calculus, Release 10.3

- other – a functor.

OUTPUT:
A functor or None.

EXAMPLES:

```python
class sage.symbolic.subring.SymbolicSubringFactory
Bases: UniqueFactory
A factory creating a symbolic subring.

INPUT:
Specify one of the following keywords to create a subring.

• accepting_variables (default: None) – a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in only these variables is created.

• rejecting_variables (default: None) – a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in variables distinct to these variables is created.

• no_variables (default: False) – a boolean. If set, then a symbolic subring of constant expressions (i.e., expressions without a variable) is created.

EXAMPLES:
```
create_key_and_extra_args(accepting_variables=None, rejecting_variables=None, no_variables=False, **kwds)

Given the arguments and keyword, create a key that uniquely determines this object.

See SymbolicSubringFactory for details.

create_object(version, key, **kwds)

Create an object from the given arguments.

See SymbolicSubringFactory for details.

class sage.symbolic.subring.SymbolicSubringRejectingVars(var)

Bases: GenericSymbolicSubring

The symbolic subring consisting of symbolic expressions whose variables are none of the given variables.

construction()

Return the functorial construction of this symbolic subring.

OUTPUT:

A tuple whose first entry is a construction functor and its second is the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(rejecting_variables=('r',)).construction()
((Subring<rejecting r>, Symbolic Ring),)
```

has_valid_variable(variable)

Return whether the given variable is valid in this subring.

INPUT:

• variable—a symbolic variable.

OUTPUT:

A boolean.

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(rejecting_variables=('r',))
sage: S.has_valid_variable('a')
True
sage: S.has_valid_variable('r')
False
sage: S.has_valid_variable('x')
True
```

class sage.symbolic.subring.SymbolicSubringRejectingVarsFunctor(var)

Bases: GenericSymbolicSubringFunctor

merge(other)

Merge this functor with other if possible.

INPUT:

• other—a functor.
OUTPUT:
A functor or None.

EXAMPLES:
```
sage: from sage.symbolic.subring import SymbolicSubring
sage: F = SymbolicSubring(accepting_variables=('a',)).construction()[0]
sage: G = SymbolicSubring(rejecting_variables=('r',)).construction()[0]
sage: G.merge(G)  # G is G
True
sage: G.merge(F)  # G is G
True
```

2.9 Classes for symbolic functions

To enable their usage as part of symbolic expressions, symbolic function classes are derived from one of the subclasses of `Function`:

- **BuiltinFunction**: the code of these functions is written in Python; many special functions are of this type
- **GinacFunction**: the code of these functions is written in C++ and part of the Pynac support library; most elementary functions are of this type
- **SymbolicFunction**: symbolic functions defined on the Sage command line are of this type

Sage uses `BuiltinFunction` and `GinacFunction` for its symbolic built-in functions. Users can define any other additional `SymbolicFunction` through the `function()` factory, see `Factory for symbolic functions`

Several parameters are supported by the superclass’ `__init__()` method. Examples follow below.

- **nargs**: the number of arguments
- **name**: the string that is printed on the CLI; the name of the member functions that are attempted for evaluation of Sage element arguments; also the name of the Pynac function that is associated with a `GinacFunction`
- **alt_name**: the second name of the member functions that are attempted for evaluation of Sage element arguments
- **latex_name**: what is printed when `latex(f(...))` is called
- **conversions**: a dict containing the function’s name in other CAS
- **evalf_params_first**: if False, when floating-point evaluating the expression do not evaluate function arguments before calling the `_evalf_()` member of the function
- **preserved_arg**: if nonzero, the index (starting with 1) of the function argument that determines the return type. Note that, e.g., `atan2()` uses both arguments to determine return type, through a different mechanism

Function classes can define the following Python member functions:

- **_eval_(*args)**: the only mandatory member function, evaluating the argument and returning the result; if None is returned the expression stays unevaluated
- **_eval_numpy_(*args)**: evaluation of `f(args)` with arguments of numpy type
- **_evalf_(*args, **kwds)**: called when the expression is floating-point evaluated; may receive a parent keyword specifying the expected parent of the result. If not defined an attempt is made to convert the result of `_eval_()`.
- **_conjugate_(*args)**, **_real_part_(*args)**, **_imag_part_(*args)**: return conjugate, real part, imaginary part of the expression `f(args)`
Symbolic Calculus, Release 10.3

- `_derivative_(*args, index)`: return derivative with respect to the parameter indexed by index (starting with 0) of f(args)
- `_tderivative_()`: same as `_derivative_()` but don’t apply chain rule; only one of the two functions may be defined
- `_power_(*args, expo)`: return \( f(args)^{expo} \)
- `_series_(*args, **kwds)`: return the power series at at up to order with respect to var of f(args); these three values are received in kwds. If not defined the series is attempted to be computed by differentiation.
- `print(*args)`: return what should be printed on the CLI with f(args)
- `print_latex(*args)`: return what should be output with latex(f(args))

The following examples are intended for Sage developers. Users can define functions interactively through the `function()` factory, see Factory for symbolic functions.

EXAMPLES:

The simplest example is a function returning nothing, it practically behaves like a symbol. Setting \( \text{nargs}=0 \) allows any number of arguments:

```
sage: from sage.symbolic.function import BuiltinFunction
sage: class Test1(BuiltinFunction):
....:     def __init__(self):
....:         BuiltinFunction.__init__(self, test, nargs=0)
....:     def _eval_(self, *args):
....:         pass
sage: f = Test1()
sage: f()
# needs sage.symbolic
test()
sage: f(1, 2, 3)*f(1, 2, 3)
# needs sage.symbolic
test(1, 2, 3)^2
```

In the following the `sin` function of \( \text{CBF}(0) \) is called because with floating point arguments the \( \text{CBF} \) element’s \( \text{my_sin()} \) member function is attempted, and after that \( \sin() \) which succeeds:

```
sage: class Test2(BuiltinFunction):
....:     def __init__(self):
....:         BuiltinFunction.__init__(self, my_sin, alt_name=sin, latex_name=r'\text{SIN}', nargs=1)
....:     def _eval_(self, x):
....:         return 5
....:     def _evalf_(self, x, **kwds):
....:         return 3.5
sage: f = Test2()
sage: f(0)
5
sage: f(0, hold=True)
# needs sage.symbolic
my_sin(0)
sage: f(0, hold=True).n()
# needs sage.rings.real_mpfr
3.50000000000000
sage: f(CBF(0))
# needs sage.libs.flint
0
```

(continues on next page)
sage: latex(f(0, hold=True))          # needs sage.symbolic
\sin\left(0\right)
sage: f(1,2)
Traceback (most recent call last):
...TypeError: Symbolic function my_sin takes exactly 1 arguments (2 given)

class sage.symbolic.function.BuiltinFunction
    Bases: Function

This is the base class for symbolic functions defined in Sage.

If a function is provided by the Sage library, we don’t need to pickle the custom methods, since we can just initialize the same library function again. This allows us to use Cython for custom methods.

We assume that each subclass of this class will define one symbolic function. Make sure you use subclasses and not just call the initializer of this class.

class sage.symbolic.function.Function
    Bases: SageObject

Base class for symbolic functions defined through Pynac in Sage.

This is an abstract base class, with generic code for the interfaces and a __call__() method. Subclasses should implement the _is_registered() and _register_function() methods.

This class is not intended for direct use, instead use one of the subclasses BuiltinFunction or SymbolicFunction.

default_variable()
    Return a default variable.

    EXAMPLES:

    sage: sin.default_variable()          # needs sage.symbolic
    x

name()
    Return the name of this function.

    EXAMPLES:

    sage: foo = function("foo", nargs=2)      # needs sage.symbolic
    sage: foo.name()                          # needs sage.symbolic
    'foo'

number_of_arguments()
    Return the number of arguments that this function takes.

    EXAMPLES:

    sage: # needs sage.symbolic
    sage: foo = function("foo", nargs=2)
variables()
Return the variables (of which there are none) present in this function.

EXAMPLES:

```
sage: sin.variables()
()```

class sage.symbolic.function.GinacFunction
Bases: BuiltinFunction
This class provides a wrapper around symbolic functions already defined in Pynac/GiNaC. GiNaC provides custom methods for these functions defined at the C++ level. It is still possible to define new custom functionality or override those already defined.

There is also no need to register these functions.

class sage.symbolic.function.SymbolicFunction
Bases: Function
This is the basis for user defined symbolic functions. We try to pickle or hash the custom methods, so subclasses must be defined in Python not Cython.

```
sage.symbolic.function.pickle_wrapper(f)
Return a pickled version of the function f.

If f is None, just return None.

This is a wrapper around pickle_function().

EXAMPLES:
```
```
sage: from sage.symbolic.function import pickle_wrapper
sage: def f(x):
   return x*x
sage: isinstance(pickle_wrapper(f), bytes)
True
sage: pickle_wrapper(None) is None
True```
```
sage.symbolic.function.unpickle_wrapper(p)
Return a unpickled version of the function defined by p.

If p is None, just return None.

This is a wrapper around unpickle_function().

EXAMPLES:

```
```
2.10 Factory for symbolic functions

`sage.symbolic.function_factory.function`(s,**kwds)

Create a formal symbolic function with the name `s`.

**INPUT:**

- `nargs=0` - number of arguments the function accepts, defaults to variable number of arguments, or 0
- `latex_name` - name used when printing in latex mode
- `conversions` - 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:**

```python
sage: from sage.symbolic.function_factory import function
sage: var('a, b')
(a, b)
sage: cr = function('cr')
```
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))
```

Basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
  ...TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class 'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

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^2*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

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
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
sage: bar = function("bar", nargs=1, eval_func=ef)
sage: bar(x)
bar(x)

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)
foo(x).n()
6
sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

def deriv(self, *args,**kwds):
    ....:       print("{} {}".format(args, kwds))
    ....:       return args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

def pow(self, x, power_param=None):
    ....:       print("{} {}".format(x, power_param))
    ....:       return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)*y

def expand(self, *args, **kwds):
    ....:       print("{} {}".format(args, sorted(kwds.items())))
    ....:       return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
y^4 + y^3 + y^2 + y + 1

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
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)
latex(foo(x,y^z))
my args are: x, y^z
```

Chain rule:

Chain rule:
Sage: def print_args(self, *args, **kwds): print("args: {}\n".format(args)); print("kwds: {}\n".format(kwds)); return args[0]
Sage: foo = function('t', nargs=2, tderivative_func=print_args)
Sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
Sage: foo = function('t', nargs=2, derivative_func=print_args)
Sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x

Sage: from sage.symbolic.function_factory import function_factory
Sage: f = function_factory(f, 2, '\foo', {mathematica:'Foo'})
Sage: f(2,4)
\foo(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: from sage.symbolic.function_factory import unpickle_function
Sage: nf = unpickle_function(f, 2, '\foo', {mathematica:'Foo'}, True, [])
Sage: nf

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

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

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

\begin{verbatim}
from sage.symbolic.function import pickle_wrapper

def evalf_f(self, x, parent=None, algorithm=None): return 2*r*x + 5r
def conjugate_f(self, x): return x/2r

nf = unpickle_function('g', 1, None, None, True, [None, pickle_wrapper(evalf_f), pickle_wrapper(conjugate_f)] + [None]*8)

nf(1, 2)
f(1, 2)
latex(nf(x,x)) \foo\left(x, x\right)
f._mathematica_init_()
'

2.11 Functional notation support for common calculus methods

EXAMPLES: We illustrate each of the calculus functional functions.

\begin{verbatim}
simplify(x - x)
0

a = var('a')
derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
diff(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
integral(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
integrate(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
limit(a*sin(x)/x, x=0)
a
taylor(a*sin(x)/x, x, 0, 4)
1/120*a*x^4 - 1/6*a*x^2 + a
expand((x - a)^3)
-a^3 + 3*a^2*x - 3*a*x^2 + x^3
\end{verbatim}

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:
Symbolic Calculus, Release 10.3

```
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)
-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
```

(continues on next page)
We differentiate a scalar field on a manifold:

```sage
M = Manifold(2, 'M')
X.<x,y> = M.chart()
f = M.scalar_field(x^2*y, name='f')
derivative(f)
```

We differentiate a differentiable form, getting its exterior derivative:

```sage
a = M.one_form(-y, x, name='a'); a
```

```sage
da = 2 dx^dy
```

```sage.calculus.functional.diff(f, *args, **kwds)
```

The derivative of \( f \).

Repeated differentiation is supported by the syntax given in the examples below.

ALIAS: diff

EXAMPLES: We differentiate a callable symbolic function:

```sage
f(x,y) = x*y + sin(x^2) + e^(-x)
```

```sage
f(x, y) |--> x*y + e^(-x)
```

```sage
derivative(f, x)
```

```sage
derivative(f, y)
```

We differentiate a polynomial:

```sage
f = (1-t)^5; f
```

```sage
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
```

```sage
derivative(f, t, t)
```

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

We differentiate a scalar field on a manifold:

sage: M = Manifold(2, 'M')
sage: X.<x,y> = M.chart()
sage: f = M.scalar_field(x^2*y, name=f)
sage: derivative(f)
1-form df on the 2-dimensional differentiable manifold M
sage: derivative(f).display()
df = 2*x*y dx + x^2 dy

We differentiate a differentiable form, getting its exterior derivative:

sage: a = M.one_form(-y, x, name='a'); a.display()
a = -y dx + x dy
sage: derivative(a)
2-form da on the 2-dimensional differentiable manifold M
sage: derivative(a).display()
da = 2 dx∧dy

sage.calculus.functional.expand(x, *args, **kwds)

EXAMPLES:
Symbolic Calculus, Release 10.3

You can also use expand on polynomial, integer, and other factorizations:

\[
\text{sage: } x = \text{polygen}(\mathbb{Z}) \\
\text{sage: } F = \text{factor}(x^{12} - 1); F \\
(x - 1) \cdot (x + 1) \cdot (x^2 - x + 1) \cdot (x^2 + 1) \cdot (x^2 + x + 1) \cdot (x^4 - x^2 + 1) \\
\text{sage: } \text{expand}(F) \\
x^{12} - 1 \\
\text{sage: } F.\text{expand}() \\
x^{12} - 1 \\
\text{sage: } F = \text{factor}(2007); F \\
3^2 \cdot 223 \\
\text{sage: } \text{expand}(F) \\
2007
\]

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.

\[
\text{sage: } x = \text{polygen}(\mathbb{Z}) \\
\text{# polynomial over a given base ring.} \\
\text{sage: } f = \text{sum}(x^n \text{ for } n \text{ in range(5)}) \\
x^8 + 2x^7 + 3x^6 + 4x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1
\]

The integral of \( f \).

**EXAMPLES:**

\[
\text{sage: } \text{integral}(\sin(x), x) \\
-\cos(x) \\
\text{sage: } \text{integral}(\sin(x)^2, x, \pi, 123*\pi/2) \\
121/4*\pi \\
\text{sage: } \text{integral}(\sin(x), x, 0, \pi) \\
2
\]

We integrate a symbolic function:

\[
\text{sage: } f(x,y,z) = x*y/z + \sin(z) \\
\text{sage: } \text{integral}(f, z) \\
(x, y, z) \mapsto x*y*log(z) - \cos(z)
\]

\[
\text{sage: } \text{var('a,b')} \\
(a, b) \\
\text{sage: } \text{assume}(b-a>0) \\
\text{sage: } \text{integral}(\sin(x), x, a, b) \\
\cos(a) - \cos(b) \\
\text{sage: } \text{forget()} \\
\text{sage: } \text{integral}(x/(x^3-1), x) \\
1/3*\text{sqrt}(3)*\text{arctan}(1/3*\text{sqrt}(3)*(2*x+1)) - 1/6*\log(x^2 + x + 1) + 1/3*\log(x - 1)
\]
We define the Gaussian, plot and integrate it numerically and symbolically:

\[
sage: f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}
\]
\[
sage: P = plot(f, -4, 4, hue=0.8, thickness=2)
\]
\[
sage: P.show(ymin=0, ymax=0.4)
\]
\[
sage: numerical_integral(f, -4, 4)
\]
\[(0.99993665765733376, 1.1101527003413533e-14)\]
\[
sage: integrate(f, x)
\]
\[x \rightarrow \frac{1}{2} \text{erf}\left(\frac{1}{2}\sqrt{2}x\right)\]

You can have Sage calculate multiple integrals. For example, consider the function \(e^{x^2}\) 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:

\[
sage: area = integral(integral(exp(y^2), x, 0, y), y, 0, 1); area
\]
\[\frac{1}{2}e - \frac{1}{2}\]
\[
sage: float(area)
\]
\[0.859140914229522...\]

We compute the line integral of \(\sin(x)\) along the arc of the curve \(x = y^4\) from \((1, -1)\) to \((1, 1)\):

\[
sage: t = var('t')
\]
\[
sage: (x,y) = (t^4,t)
\]
\[
(dx,dy) = (diff(x,t), diff(y,t))
\]
\[
sage: integral(sin(x)*dx, t, -1, 1)
\]
\[0\]
\[
sage: restore('x,y')\]  
\[# restore the symbolic variables x and y\]

Sage is now \(\text{github issue } #27958\) able to compute the following integral:

\[
sage: integral(exp(-x^2)*\log(x), x) \# long time
\]
\[\frac{1}{2}\sqrt{\pi}\text{erf}(x)\log(x) - x\text{hypergeometric}((1/2, 1/2), (3/2, 3/2), -x^2)\]

and its value:

\[
sage: integral( exp(-x^2)*\ln(x), x, 0, \infty)
\]
\[-\frac{1}{4}\sqrt{\pi}\text{erf}(\ln(2)) (\eulerGamma + 2\log(2))\]

This definite integral is easy:

\[
sage: integral( \ln(x)/x, x, 1, 2)
\]
\[1/2\log(2)^2\]

Sage cannot do this elliptic integral (yet):

\[
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
\]
\[\integrate(1/(\sqrt{2t^2 + 1})\sqrt{t^2 - 2)}, t, 2, 3)\]

A double integral:

\[
sage: y = var('y')
\]
\[
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
\]
\[32/5\]

This illustrates using assumptions:
We integrate and differentiate a huge mess:

\[
sage: \ f = \frac{x^2-1+3(1+x^2)^{1/3}}{(1+x^2)^{2/3}x/(x^2+2)^2}
\]
\[
sage: \ h = \ f - \text{diff}(\ g, \ x)
\]

\[
sage: \ [\text{float}(h(x=i)) \ for \ i \ in \ \text{range}(5)] \ # \text{random}
\]
\[
[0.0, -1.1102230246251565e-16, -5.5511151231257827e-17, -5.5511151231257827e-17, -6.9388939039072284e-17]
\]
\[
sage: \ h.\text{factor}()
0
\]
\[
sage: \ \text{bool}(h == 0)
True
\]

\[
\text{sage.calculus.functional}.\text{integrate}(f, *\text{args}, **\text{kwds})
\]

The integral of \( f \).

EXAMPLES:

\[
sage: \ \text{integral}(\sin(x), \ x)
-\cos(x)
\]
\[
sage: \ \text{integral}(\sin(x)^2, \ x, \ \pi, 123*\pi/2)
\frac{121}{4*\pi}
\]
\[
sage: \ \text{integral}(\ \sin(x), \ x, \ 0, \ \pi)
2
\]

We integrate a symbolic function:

\[
sage: \ f(x,y,z) = x*y/z + \sin(z)
\]
\[
sage: \ \text{integral}(f, \ z)
(x, y, z) \rightarrow x*y*\log(z) - \cos(z)
\]
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 \(e^\left(x^2\right)\) 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 now (github issue #27958) able to compute the following integral:

```
sage: integral(exp(-x^2)*log(x), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
```

and its value:

```
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 cannot do this elliptic integral (yet):
Symbolic Calculus, Release 10.3

```python
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
```

A double integral:

```python
sage: y = var('y')
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

This illustrates using assumptions:

```python
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:

```python
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)
```

```python
sage: [float(h(x=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
```

```python
sage.calculus.functional.lim(f, dir=None, taylor=False, **argv)
Return the limit as the variable \( v \) approaches \( a \) from the given direction.
```

```python
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=oo)
-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
```

(continues on next page)
Sage does not know how to do this limit (which is 0), so it returns it unevaluated:

\[
\text{sage: lim(exp(x^2)*(1-erf(x)), x=\infty)}
\]

\[
\text{lim}((\text{erf}(x) - 1)\cdot e^{x^2}, x, +\infty)
\]

\[
\text{sage.calculus.functional.simplify}(f, algorithm='maxima', **kwds)
\]

Simplify the expression \( f \).

See the documentation of the \text{simplify()}\ method of symbolic expressions for details on options.

**EXAMPLES:**

We simplify the expression \( i + x - x \):

\[
\text{sage: f = I + x - x; simplify(f)}
\]

\( I \)

In fact, printing \( f \) yields the same thing - i.e., the simplified form.

Some simplifications are algorithm-specific:

\[
\text{sage: x, t = var("x, t")}
\]

\[
\text{sage: ex = 1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/2/(I*x + I*sqrt(x^2 - 1))}
\]

\[
\text{sage: simplify(ex)}
\]

\( \frac{1}{2}i\cdot x + \frac{1}{2}i\cdot \sqrt{x^2 - 1} + \frac{1}{2}(x^2 - 1) \)

\[
\text{sage: simplify(ex, algorithm="giac")}
\]

\( i\cdot \sqrt{x^2 - 1} \)

\[
\text{sage.calculus.functional.taylor}(f, *\text{args})
\]

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
- \((x, a), (y, b), \ldots, n\) - variables with points, degree of polynomial

**EXAMPLES:**

\[
\text{sage: var('x, k, n')}
\]

\((x, k, n)\)

\[
\text{sage: taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6)}
\]

\(-1/720*(45*k^6 - 60*k^4 + 16*k^2)*x^6 - 1/24*(3*k^4 - 4*k^2)*x^4 - 1/2*k^2*x^2 + 1\)

\[
\text{sage: taylor ((x + 1)^n, x, 0, 4)}
\]

\(1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 - \ldots - n)*x^2 + n*x + 1\)

\[
\text{sage: taylor ((x + 1)^n, x, 0, 4)}
\]

\(1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 - \ldots - n)*x^2 + n*x + 1\)

Taylor polynomial in two variables:
Symbolic Calculus, Release 10.3

```python
sage: x, y = var('x y');
taylor(x*y^3, (x, 1), (y, -1), 4)
(x - 1)*(y + 1)^3 - 3*(x - 1)*(y + 1)^2 + (y + 1)^3 + 3*(x - 1)*(y + 1) - 3*(y + 1)^2 - x + 3*y + 3
```

### 2.12 Symbolic Integration

**class** `sage.symbolic.integration.integral.DefiniteIntegral`

Bases: `BuiltinFunction`

The symbolic function representing a definite integral.

**EXAMPLES:**

```python
sage: from sage.symbolic.integration.integral import definite_integral
sage: definite_integral(sin(x), x, 0, pi)
```

**class** `sage.symbolic.integration.integral.IndefiniteIntegral`

Bases: `BuiltinFunction`

Class to represent an indefinite integral.

**EXAMPLES:**

```python
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)*

Return the indefinite integral with respect to the variable *v*, ignoring the constant of integration. Or, if endpoints *a* and *b* are specified, returns the definite integral over the interval [*a*, *b*].

If `self` has only one variable, then it returns the integral with respect to that variable.

If definite integration fails, it could be still possible to evaluate the definite integral using indefinite integration with the Newton-Leibniz theorem (however, the user has to ensure that the indefinite integral is continuous on the compact interval [*a*, *b*] and this theorem can be applied).

**INPUT:**

- *v* - a variable or variable name. This can also be a tuple of the variable (optional) and endpoints (i.e., *(x, 0, 1)* or *(0, 1)*).
- *a* - (optional) lower endpoint of definite integral
- *b* - (optional) upper endpoint of definite integral
- *algorithm* - (default: ‘maxima’, ‘libgiac’ and ‘sympy’) one of
  - ‘maxima’ - use maxima
Symbolic Calculus, Release 10.3

- ‘sympy’ - use sympy (also in Sage)
- ‘mathematica_free’ - use http://integrals.wolfram.com/
- ‘fricas’ - use FriCAS (the optional fricas spkg has to be installed)
- ‘giac’ - use Giac
- ‘libgiac’ - use libgiac

To prevent automatic evaluation use the \texttt{hold} argument.

See also:

To integrate a polynomial over a polytope, use the optional \texttt{latte_int} package \texttt{sage.geometry.polyhedron.base.Polyhedron_base.integrate()}.  

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

The variable is required, but the endpoints are optional:

\begin{verbatim}
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)
\end{verbatim}

Using the \texttt{hold} parameter it is possible to prevent automatic evaluation, which can then be evaluated via \texttt{simplify()}:

\begin{verbatim}
sage: integral(x^2, x, 0, 3)
9
\end{verbatim}

(continues on next page)
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)
We integrate the same function in both Mathematica and Sage (via Maxima):

```sage
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: g = mathematica(f)  # optional - mathematica
sage: print(g)  # optional - mathematica
x^2 \text{ y} + \sin[x ]
```

```sage
sage: print(g.Integrate(x))  # optional - mathematica
x^2 \text{ y} + \sqrt[2]{\text{pi}}\text{ FresnelS}[\sqrt[2]{\text{pi}}\text{ x}]
```

```sage
sage: print(f.integral(x))
x^2 \text{ y} + 1/16*\sqrt[2]{\text{pi}}\text{ FresnelS}[\sqrt[2]{\text{pi}}\text{ x}]
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```sage
sage: _ = var('x, y, z')  # optional - internet
sage: f = sin(x^2) + y^z  # optional - internet
sage: f.integrate(x, algorithm='mathematica_free')  # optional - internet
x^2 \text{ y} + \sqrt[2]{\text{pi}}\text{ FresnelS}[\sqrt[2]{\text{pi}}\text{ x}]
```

We can also use Sympy:

```sage
sage: integrate(x*sin(log(x)), x)
-1/5*x^2*(\cos(\log(x)) - 2*\sin(\log(x)))
```

```sage
sage: integrate(x*sin(log(x)), x, algorithm='sympy')  #.. needs sympy
-1/5*x^2*\cos(\log(x)) + 2/5*x^2*\sin(\log(x))
```

```sage
sage: _ = var('y, z')
sage: (x^y - z).integrate(y)
-y*z + x^y/\log(x)
```

```sage
sage: (x^y - z).integrate(y, algorithm="sympy")  #.. needs sympy
-y*z + \text{cases}(((\log(x) != 0, x^y/\log(x)), (1, y)))
```

We integrate the above function in Maple now:

```sage
sage: g = maple(f); g.sort()  # optional - maple
y^z+\sin(x^2)
```

```sage
sage: g.integrate(x).sort()  # optional - maple
x^y^z+1/2*2^(1/2)*\pi^(1/2)*\text{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
sage: A = integral(1/ ((x^4) * (x^4+x+1)), x); A
\int \frac{1}{(x^4 + x + 1)*(x - 4)} dx
```

Sometimes, in this situation, using the algorithm “maxima” gives instead a partially integrated answer:
\begin{verbatim}
integral(1/(x^7-1),x,algorithm='maxima')
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1), x) + 1/7*log(x - 1)
\end{verbatim}

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

\begin{verbatim}
integral(e^(-x^2),(x, 0, 0.1))
0.05623145800914245*sqrt(pi)
\end{verbatim}

An example of an integral that fricas can integrate:

\begin{verbatim}
f(x) = sqrt(x+sqrt(1+x^2))/x
integrate(f(x), x, algorithm="fricas")
# optional - fricas
2*sqrt(x + sqrt(x^2 + 1)) - 2*arctan(sqrt(x + sqrt(x^2 + 1))) - log(sqrt(x + sqrt(x^2 + 1)) + 1) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
\end{verbatim}

where the default integrator obtains another answer:

\begin{verbatim}
integrate(f(x), x) # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4), (1/2, 3/4), -1/x^2)/(pi*gamma(3/4))
\end{verbatim}

The following definite integral is not found by maxima:

\begin{verbatim}
f(x) = (x^4 - 3*x^2 + 6) / (x^6 - 5*x^4 + 5*x^2 + 4)
integrate(f(x), x, 1, 2, algorithm='maxima')
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 1, 2)
\end{verbatim}

but is nevertheless computed:

\begin{verbatim}
integrate(f(x), x, 1, 2)
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
\end{verbatim}

Both fricas and sympy give the correct result:

\begin{verbatim}
integrate(f(x), x, 1, 2, algorithm="fricas") # optional - fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
integrate(f(x), x, 1, 2, algorithm="sympy") # needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
\end{verbatim}

Using Giac to integrate the absolute value of a trigonometric expression:

\begin{verbatim}
integrate(abs(cos(x)), x, 0, 2*pi, algorithm='giac')
4
\end{verbatim}

ALIASES: integral() and integrate() are the same.

EXAMPLES:

Here is an example where we have to use assume:
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^(7/3) -
- 1/9*b^2*log((b*x + a)^(2/3) + a^(1/3))/a^(7/3) + 2/
- 9*b^2*log((b*x + a)^(1/3) - a^(1/3))/a^(7/3) + 1/6*(4*(b*x + a)^(5/3)*b^2 -
- 7*(b*x + a)^(2/3)*a*b^2)/(b*x + a)^2*a^2 - 2*(b*x + a)*a^3 + a^4)
```

The Sage function `sage.symbolic.integration.integral.integrate` is used to perform indefinite integration with respect to a variable, ignoring the constant of integration. If endpoints $a$ and $b$ are specified, it returns the definite integral over the interval $[a, b]$.

If self has only one variable, then it returns the integral with respect to that variable.

If definite integration fails, it could be still possible to evaluate the definite integral using indefinite integration with the Newton-Leibniz theorem (however, the user has to ensure that the indefinite integral is continuous on the compact interval $[a, b]$ and this theorem can be applied).

**INPUT:**

- **v** - a variable or variable name. This can also be a tuple of the variable (optional) and endpoints (i.e., $(x, 0, 1)$ or $(0, 1)$).
- **a** - (optional) lower endpoint of definite integral
- **b** - (optional) upper endpoint of definite integral
- **algorithm** - (default: 'maxima', 'libgiac' and 'sympy') one of
  - ‘maxima’ - use maxima
  - ‘sympy’ - use sympy (also in Sage)
  - ‘mathematica_free’ - use http://integrals.wolfram.com/
  - ‘fricas’ - use FriCAS (the optional fricas spkg has to be installed)
  - ‘giac’ - use Giac
  - ‘libgiac’ - use libgiac

To prevent automatic evaluation use the `hold` argument.

**See also:**

To integrate a polynomial over a polytope, use the optional `latte_int` package sage.geometry.polyhedron.base.Polyhedron_base.integrate().

**EXAMPLES:**
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: h.integral(x)
1/cos(x)

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 $-1$?

\begin{verbatim}
sage: assume(n > 0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
\end{verbatim}

Usually the constraints are of sign, but others are possible:

\begin{verbatim}
sage: assume(n==-1)
sage: integral(x^n,x)
\log(x)
sage: forget()
\end{verbatim}

Note that an exception is raised when a definite integral is divergent:

\begin{verbatim}
sage: integrate(1/x^3,(x,0,1))
ValueError: Integral is divergent.
sage: integrate(1/x^3,x,-1,3)
ValueError: Integral is divergent.
\end{verbatim}

But Sage can calculate the convergent improper integral of this function:

\begin{verbatim}
sage: integrate(1/x^3,x,1,infinity)
1/2
\end{verbatim}

The examples in the Maxima documentation:

\begin{verbatim}
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)
\end{verbatim}

We integrate the same function in both Mathematica and Sage (via Maxima):

\begin{verbatim}
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       2
x y + Sqrt[--] FresnelS[Sqrt[--] x]
2       Pi
\end{verbatim}

(continues on next page)
Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```
sage: _ = var('x, y, z')  # optional - internet
sage: f = sin(x^2) + y^z  # optional - internet
sage: f.integrate(x, algorithm="mathematica_free")  # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnel_sin(sqrt(2)*x/sqrt(pi))
```

We can also use Sympy:

```
sage: integrate(x*sin(log(x)), x)  # optional - sympy
-1/5*x^2*(cos(log(x)) - 2*sin(log(x)))
```

```
sage: integrate(x*y^z, y)  # optional - sympy
-y*z + x^y/log(x)
```

We integrate the above function in Maple now:

```
sage: g = maple(f); g.sort()  # optional - maple
y^z+sin(x^2)
```

Sometimes, in this situation, using the algorithm “maxima” gives instead a partially integrated answer:

```
sage: integral(1/(x**7-1),x,algorithm='maxima')
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1), x) + 1/7*log(x - 1)
```

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

```
sage: integral(e^(-x^2),(x, 0, 0.1))
0.05623145800914245*sqrt(pi)
```

An example of an integral that fricas can integrate:

```
sage: f(x) = sqrt(x+sqrt(1+x^2))/x
sage: integrate(f(x), x, algorithm="fricas")  # optional - fricas
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

\[ 2\sqrt{x + \sqrt{x^2 + 1}} - 2\arctan\left(\sqrt{x + \sqrt{x^2 + 1}}\right) + 1 + \log\left(\sqrt{x + \sqrt{x^2 + 1}} - 1\right) \]

where the default integrator obtains another answer:

```
sage: integrate(f(x), x)  # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4), (1/2, 3/4), -1/x^2)/(pi*gamma(3/4))
```

The following definite integral is not found by maxima:

```
sage: f(x) = (x^4 - 3*x^2 + 6) / (x^6 - 5*x^4 + 5*x^2 + 4)
sage: integrate(f(x), x, 1, 2, algorithm=maxima)
```

but is nevertheless computed:

```
sage: integrate(f(x), x, 1, 2)
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Both fricas and sympy give the correct result:

```
sage: integrate(f(x), x, 1, 2, algorithm="fricas")  # optional - fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
sage: integrate(f(x), x, 1, 2, algorithm="sympy")  # needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Using Giac to integrate the absolute value of a trigonometric expression:

```
sage: integrate(abs(cos(x)), x, 0, 2*pi, algorithm='giac')
4
```

ALIASES: integral() and integrate() are the same.

EXAMPLES:

Here is an example where we have to use assume:

```
sage: a, b = var('a,b')
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
Traceback (most recent call last):
... ValueError: Computation failed since Maxima requested additional constraints; using the 'assume' command before evaluation may help (example of legal syntax is 'assume(a>0)', see `assume?' for more details)
Is a positive or negative?
```

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)
```

(continues on next page)
2.13 TESTS::

sage.symbolic.integration.external.fracas_integrator(expression, v, a=None, b=None, noPole=True)

Integration using FriCAS

EXAMPLES:

sage: # optional - fricas
sage: from sage.symbolic.integration.external import fracas_integrator
def fricas_integrator(expression, variable):
    """Integration using FriCAS"
    return expression.integrate(variable)

sage: fricas_integrator(sin(x), x)
-cos(x)

sage: fricas_integrator(cos(x), x)
sin(x)

sage: fricas_integrator(1/(x^2-2), x, 0, 1)
-1/8*sqrt(2)*(log(2) - log(-24*sqrt(2) + 34))

sage: fricas_integrator(1/(x^2+6), x, -oo, oo)
1/6*sqrt(6)*pi

sage.symbolic.integration.external.giac_integrator(expression, v, a=None, b=None)

Integration using Giac

EXAMPLES:

sage: from sage.symbolic.integration.external import giac_integrator
def giac_integrator(expression, variable):
    """Integration using Giac"
    return expression.integrate(variable)

sage: giac_integrator(sin(x), x)
-cos(x)

sage: giac_integrator(1/(x^2+6), x, -oo, oo)
1/6*sqrt(6)*pi

sage.symbolic.integration.external.libgiac_integrator(expression, v, a=None, b=None)

Integration using libgiac

EXAMPLES:

sage: from sage.symbolic.integration.external import libgiac_integrator
def libgiac_integrator(expression, variable):
    """Integration using libgiac"
    return expression.integrate(variable)

sage: libgiac_integrator(sin(x), x)
-cos(x)

sage: libgiac_integrator(1/(x^2+6), x, -oo, oo)
1/6*sqrt(6)*pi

No checks were made for singular points of antiderivative...

sage.symbolic.integration.external.maxima_integrator(expression, v, a=None, b=None)

Integration using Maxima

EXAMPLES:
Symbolic Calculus, Release 10.3

```python
sage: from sage.symbolic.integration.external import maxima_integrator
sage: maxima_integrator(sin(x), x)
-cos(x)
sage: maxima_integrator(cos(x), x)
sin(x)
sage: f(x) = function('f')(x)
sage: maxima_integrator(f(x), x)
integrate(f(x), x)
```

```
sage.symbolic.integration.external.mma_free_integrator(expression, v, a=None, b=None)
Integration using Mathematica’s online integrator

EXAMPLES:
```n
sage: from sage.symbolic.integration.external import mma_free_integrator
sage: mma_free_integrator(sin(x), x)  # optional - internet
-cos(x)
```

```
sage: mma_free_integrator(e^(-x), x, a=0, b=oo)  # optional - internet
1
```

```
sage.symbolic.integration.external.sympy_integrator(expression, v, a=None, b=None)
Integration using SymPy

EXAMPLES:
```n
sage: from sage.symbolic.integration.external import sympy_integrator
sage: sympy_integrator(sin(x), x)  # needs sympy
-cos(x)
sage: sympy_integrator(cos(x), x)  # needs sympy
sin(x)
```

### 2.14 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.

```python
sage: a = Rational((1,2))
sage: a
1/2
sage: a*2
1
sage: Rational(2)^50 / Rational(10)^50
1/8881784197001252323389053347265625
sage: 1.0/2
0.5
sage: pi^2
pi^2
```

(continues on next page)
sage: float(pi)
3.141592653589793
sage: RealField(200)(pi)
3.1415926535897932384626433832795028841971693993751058209749
sage: float(pi + exp(1))
5.8597448204883...

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: cos(x).taylor(x,0,10)
-\frac{1}{240}x^8 - \frac{1}{2}x^4 + 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]])
(continues on next page)
And here are some actual tests of sympy:

```sage
sage: from sympy import Symbol, cos, sympify, pprint
# needs sympy
sage: from sympy.abc import x
# needs sympy

sage: e = (1/cos(x)**3)._sympy_(); e  
# needs sympy
\cos(x)^{-3}

sage: f = e.series(x, 0, int(10)); f  
# needs sympy
1 + 3*x**2/2 + 11*x**4/8 + 241*x**6/240 + 8651*x**8/13440 + O(x**10)
```

And the pretty-printer. Since unicode characters are not working on some architectures, we disable it:

```sage
sage: from sympy.printing import pprint_use_unicode  
# needs sympy
sage: prev_use = pprint_use_unicode(False)  
# needs sympy
sage: pprint(e)  
# needs sympy
\frac{1}{\cos(x)^3}

sage: pprint(f)  
# needs sympy
1 + 2*4 6 8
3*x 11*x 241*x 8651*x / 10\ 1 + -- + ---- + ------ + ------- + O(x) / 
2 8 240 13440

sage: pprint_use_unicode(prev_use)  
# needs sympy
False
```

And the functionality to convert from sympy format to Sage format:

```sage
sage: e._sage_()  
(continues on next page)
```
Symbolic Calculus, Release 10.3

(continued from previous page)

\[ \cos(x)^{-3} \]

\[
\text{sage: } e._\text{sage_}().\text{taylor}(x._\text{sage_}(), 0, 8) \quad \#
\]

\[
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
\]

\[
\text{sage: } f._\text{sage_}() \quad \#
\]

\[
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + \text{Order}(x^{10}) + 1
\]

Mixing SymPy with Sage:

\[
\text{sage: } \text{import sympy}
\]

\[
\text{sage: } \text{var("x")._sympy_()} + \text{var("y")._sympy_()}
\]

\[ x + y \]

\[
\text{sage: } o = \text{var("omega")}
\]

\[
\text{sage: } s = \text{sympy.Symbol("x")}
\]

\[
\text{sage: } t1 = s + o
\]

\[
\text{sage: } t2 = o + s
\]

\[
\text{sage: } \text{type}(t1)
\]

\[ \text{<class 'sympy.core.add.Add'>} \]

\[
\text{sage: } \text{type}(t2)
\]

\[ \text{<class 'sage.symbolic.expression.Expression'>} \]

\[
\text{sage: } t1, t2
\]

\[ \{\omega + x, \omega + x\} \]

\[
\text{sage: } e = \text{sympy.sin(var("y")) + sage.functions.trig.cos(sympy.Symbol("x"))}
\]

\[
\text{sage: } \text{type}(e)
\]

\[ \text{<class 'sympy.core.add.Add'>} \]

\[
\text{sage: } e
\]

\[ \sin(y) + \cos(x) \]

\[
\text{sage: } e=e._\text{sage_}()
\]

\[
\text{sage: } \text{type}(e)
\]

\[ \text{<class 'sage.symbolic.expression.Expression'>} \]

\[
\text{sage: } e
\]

\[ \cos(x) + \sin(y) \]

\[
\text{sage: } e = \text{sage.functions.trig.cos(var("y")**3)**4+var("x")**2}
\]

\[
\text{sage: } e = e._\text{sage_}()
\]

\[
\text{sage: } e
\]

\[ x^{**2} + \cos(y^{**3})^{**4} \]

\[
\text{sage: } a = \text{sympy.Matrix([1, 2, 3])}
\]

\[
\text{sage: } a[1]
\]

\[
\text{sage: } \text{sympify}(1.5)
\]

\[ 1.50000000000000 \]

\[
\text{sage: } \text{sympify}(2)
\]

\[ 2 \]

\[
\text{sage: } \text{sympify}(-2)
\]

\[ -2 \]

2.14. A Sample Session using SymPy
2.15 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],[-(1-1/r)^(-1),0,0,0],[0,0,-r^2,0],[0,0,0,-r^2*(sin(theta))^2]])
sage: m
[ -1/r + 1 0 0 0]
[ 0 1/(1/r - 1) 0 0]
[ 0 0 -r^2 0]
[ 0 0 0 -r^2*sin(theta)^2]
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/7)
sage: b = A^14
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)
(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)
diff(f(x), x)
sage: diff( 2*x*f(x^2), x)
4*x^2*D[0](f)(x^2) + 2*f(x^2)
sage: integrate( 1/(x^4 - a^4), x)
-1/2*arctan(x/a)/a^3 - 1/4*log(a + x)/a^3 + 1/4*log(-a + x)/a^3
sage: expand(integrate(log(1-x^2), x))
x*log(-x^2 + 1) - 2*x + log(x + 1) - log(x - 1)
```

This is an apparent regression in Maxima 5.39.0, although the antiderivative is correct, assuming we work with (poly)logs of complex argument. More convenient form is 1/2*log(x^2)*log(-x^2 + 1) + 1/2*dilog(-x^2 + 1). See also https://sourceforge.net/p/maxima/bugs/3275/:

```
sage: integrate(log(1-x^2)/x, x)
log(-x)*log(x + 1) + log(x)*log(-x + 1) + dilog(x + 1) + dilog(-x + 1)
```

No problems here:

```
sage: integrate(exp(1-x^2),x)
1/2*sqrt(pi) * erf(x) * e
sage: integrate(sin(x^2),x)
1/16*pi^1/2 * erfi(sqrt(1/4+x^2) - I*x) + 1/16*pi^1/2 * erfi(sqrt(1/4+x^2) + I*x)
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```
sage: integrate((1-x^2)^n,x)  # long time
  x*hypergeometric((1/2, -n), (3/2,), x^2*exp_polar(2*I*pi))
sage: integrate(x*x,x)
  integrate(x*x, x)
sage: integrate(1/(x^3+1),x)
  1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x - 1)) - 1/6*log(x^2 - x + 1) + 1/3*log(x + 1)
sage: integrate(1/(x^3+1), x, 0, 1)
  1/9*sqrt(3)*pi + 1/3*log(2)
```

```
sage: forget()
sage: c = var('c')
sage: assume(c > 0)
sage: integrate(exp(-c*x^2), x, -oo, oo)
  sqrt(pi)/sqrt(c)
```

Other examples that now (github issue #27958) work:

```
sage: integrate(log(x)*exp(-x^2), x)  # long time
  1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
sage: integrate(log(1+sqrt(1+4*x)/2)/x, x, 0, 1)
  ...  
  ValueError: Integral is divergent.
```

The following is an example of integral that Mathematica can do, but Sage currently cannot do:

```
sage: integrate(ceil(x^2 + floor(x)), x, 0, 5, algorithm='maxima')
  integrate(ceil(x^2) + floor(x), x, 0, 5)
```

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)
  diff(f(x), x)
sage: diff(f(x,y), x, y)
  diff(f(x, y), x, y)
sage: diff(f(x,y), x, y) - diff(f(x,y), y, x)
  0
sage: g = function('g')
sage: var('x y z')
  (x, y, z)
sage: diff(g(x,y,z), x,z,z)
  diff(g(x, y, z), x, z, z)
```

(continues on next page)
### 2.16 Conversion of symbolic expressions to other types

This module provides routines for converting new symbolic expressions to other types. Primarily, it provides a class `Converter` which will walk the expression tree and make calls to methods overridden by subclasses.

**class** `sage.symbolic.expression_conversions.Converter (use_fake_div=False)`

Bases: object

If `use_fake_div` is set to True, then the converter will try to replace expressions whose operator is `operator.mul` with the corresponding expression whose operator is `operator.truediv`.

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
```
Symbolic Calculus, Release 10.3

(continued from previous page)

```sage
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.

**composition** *(ex, operator)*

The input to this method is a symbolic expression and its operator. This method will get called when you have a symbolic function application.

**derivative** *(ex, operator)*

The input to this method is a symbolic expression which corresponds to a relation.

**get_fake_div** *(ex)*

EXAMPIES:

```sage
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
```

```sage
sage: c.get_fake_div(sin(x)/x)
FakeExpression([sin(x), x], <built-in function truediv>)
sage: c.get_fake_div(-1*sin(x))
FakeExpression([sin(x)], <built-in function neg>)
sage: c.get_fake_div(-x)
FakeExpression([x], <built-in function neg>)
sage: c.get_fake_div((2*x^3+2*x-1)/((x-2)*(x+1)))
FakeExpression([2*x^3 + 2*x - 1, FakeExpression([x + 1, x - 2], <built-in...function mul>)], <built-in function truediv>)
```

Check if [github issue #8056](https://github.com/sagemath/sage/issues/8056) is fixed, i.e., if numerator is 1.:

```sage
sage: c.get_fake_div(1/pi/x)
FakeExpression([1, FakeExpression([pi, x], <built-in function mul>)], <built-in...function truediv>)
```

**pyobject** *(ex, obj)*

The input to this method is the result of calling `pyobject()` on a symbolic expression.

**Note**: Note that if a constant such as `pi` is encountered in the expression tree, its corresponding `pyobject` which is an instance of `sage.symbolic.constants.Pi` will be passed into this method. One cannot do arithmetic using such an object.

**relation** *(ex, operator)*

The input to this method is a symbolic expression which corresponds to a relation.

**symbol** *(ex)*

The input to this method is a symbolic expression which corresponds to a single variable. For example, this method could be used to return a generator for a polynomial ring.

**class** `sage.symbolic.expression_conversions.DeMoivre` *(ex, force=False)*

Bases: `ExpressionTreeWalker`

A class that walks a symbolic expression tree and replaces occurrences of complex exponentials (optionally, all exponentials) by their respective trigonometric expressions.

**INPUT:**
• force – boolean (default: False); replace exp(x) with cosh(x) + sinh(x)

EXAMPLES:

```python
sage: a, b = SR.var("a, b")
sage: from sage.symbolic.expression_conversions import DeMoivre
sage: d = DeMoivre(e^a)
sage: d(e^(a+I*b))
(cos(b) + I*sin(b))*e^a
```

composition(ex, op)

Return the composition of self with ex by op.

EXAMPLES:

```python
sage: x, a, b = SR.var('x, a, b')
sage: from sage.symbolic.expression_conversions import DeMoivre
sage: p = exp(x)
sage: q = exp(a+I*b)
sage: s.composition(q, q.operator())
(cos(b) + I*sin(b))*e^a
```

class sage.symbolic.expression_conversions.Exponentialize(ex)

Bases: ExpressionTreeWalker

A class that walks a symbolic expression tree and replace circular and hyperbolic functions by their respective exponential expressions.

EXAMPLES:

```python
sage: from sage.symbolic.expression_conversions import Exponentialize
sage: d = Exponentialize(sin(x))
sage: d(sin(x))
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
sage: d(cosh(x))
1/2*e^(-x) + 1/2*e^x
```

CircDict = {sinh: x |--> -1/2*e^(-x) + 1/2*e^x, cosh: x |--> 1/2*e^(-x) + 1/2*e^x, tanh: x |--> -(e^(-x) - e^x)/(e^(-x) + e^x), coth: x |--> -e^(-x) + e^x)/(e^(-x) - e^x), sech: x |--> 2/(e^(-x) + e^x), csch: x |--> -2/(e^(-x) - e^x), sin: x |--> -1/2*I*e^(-I*x) + 1/2*I*e^(I*x), cos: x |--> 1/2*e^(I*x) + 1/2*e^(-I*x), tan: x |--> (-I*e^(I*x) + I*e^(-I*x))/(e^(I*x) + e^(-I*x)), cot: x |--> (I*e^(I*x) + I*e^(-I*x))/(e^(I*x) - e^(-I*x)), sec: x |--> 2/(e^(I*x) + e^(-I*x)), csc: x |--> 2*I/(e^(I*x) - e^(-I*x))}

Circs = [sin, cos, sec, csc, tan, cot, sinh, cosh, sech, csch, tanh, coth]

I = I

Integer

alias of Integer

SR = Symbolic Ring

composition(ex, op)

Return the composition of self with ex by op.

EXAMPLES:
Symbolic Calculus, Release 10.3

```
sage: x = SR.var("x")
sage: from sage.symbolic.expression_conversions import Exponentialize
sage: p = x
sage: s = Exponentialize(p)
sage: q = sin(x)
sage: s.composition(q, q.operator())
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
```

```
cos = cos
cosh = cosh
cot = cot
coth = coth
csc = csc
csch = csch
e = e
exp = exp

function(s, **kwds)
    Create a formal symbolic function with the name s.
    
    INPUT:
    
    • nargs=0 - number of arguments the function accepts, defaults to variable number of arguments, or 0
    • latex_name - name used when printing in latex mode
    • conversions - 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 call-
      ing the custom evalf function
    • conjugate_func - method used for complex conjugation
    • real_part_func - method used when taking real parts
    • imag_part_func - method used when taking imaginary parts
    • derivative_func - method to be used for (partial) derivation This method should take a keyword
      argument deriv_param specifying the index of the argument to differentiate w.r.t
    • tderivative_func - method to be used for derivatives
    • power_func - method used when taking powers This method should take a keyword argument
      power_param specifying the exponent
    • series_func - method used for series expansion This method should expect keyword arguments -
      order - order for the expansion to be computed - var - variable to expand w.r.t. - at - expand at this
      value
    • print_func - method for custom printing
    • print_latex_func - method for custom printing in latex mode
```
Note that custom methods must be instance methods, i.e., expect the instance of the symbolic function as the first argument.

**Note:** The new function is both returned and automatically injected into the global namespace. If you use this function in library code, it is better to use `sage.symbolic.function_factory.function`, since it will not touch the global namespace.

EXEMPLARY:

We create a formal function called supersin

```
sage: function('supersin')
supersin
```

We can immediately use supersin in symbolic expressions:

```
sage: y, z, A = var('y z A')
sage: supersin(y+z) + A^3
A^3 + supersin(y + z)
```

We can define other functions in terms of supersin:

```
sage: g(x,y) = supersin(x)^2 + sin(y/2)
sage: g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
sage: g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
sage: k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients:

```
sage: r, kappa = var('r,kappa')
sage: psi = function('psi', nargs=1)(r); psi
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

Custom typesetting of symbolic functions in LaTeX, either using `latex_name` keyword:

```
sage: function('riemann', latex_name="\mathcal{R}")
riemann
sage: latex(riemann(x))
\mathcal{R}(x)
```

or passing a custom callable function that returns a latex expression:

```
sage: mu, nu = var('mu,nu')
sage: def my_latex_print(self, *args): return "\psi_{%s}\"%(, .
```

(continues on next page)
\begin{verbatim}
from sympy import Function, latex

# Define a function psi
psi = Function('psi', print_latex_func=print_latex)

# Print psi in LaTeX format
print(latex(psi(mu, nu)))

# Define custom methods for automatic or numeric evaluation, derivation, conjugation, etc.
def ev(self, x):
    return 2 * x

def ef(self, x):
    pass

def evalf_f(self, x, parent=None, algorithm=None):
    return 6

def deriv(self, *args, **kwds):
    print("{} {}".format(args, kwds));
    return args[0]**2

def pow(self, x, power_param=None):
    print("{} {}".format(x, power_param));
    return x * power_param

def expand(self, *args, **kwds):
    print("{} {}".format(args, pformat(kwds))); return
    sum(args[0]**i for i in range(kwds['order']))

def my_print(self, *args):
    print("my args are: " + ', '.join(map(repr, args)))

# Create functions with custom methods
foo = Function('foo', nargs=1, eval_func=ev)
bar = Function('bar', nargs=1, eval_func=ef)

# Example usage
foo(x)
foo(x).n()
foo = function('foo', args=(x, y), derivative_func=deriv)
foo(y).derivative(y)

# Print custom methods
print(ev)
print(ef)
print(evalf_f)
print(deriv)
print(pow)
print(expand)
print(my_print)
\end{verbatim}

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:
Symbolic Calculus, Release 10.3

(continued from previous page)

\begin{Verbatim}
sage: foo(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)
\end{Verbatim}

Chain rule:

\begin{Verbatim}
sage: def print_args(self, *args, **kwds): print("args: {}".format(args));
           \rightarrow print("kwds: {}".format(kwds)); \textbf{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
\end{Verbatim}

Since Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

\begin{Verbatim}
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
  ... TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class 'sage.symbolic.function_factory...NewSymbolicFunction'>'
\end{Verbatim}

You now need to evaluate the function in order to do the arithmetic:

\begin{Verbatim}
sage: 2*f(x)
2*f(x)
\end{Verbatim}

Since Sage 4.0, you need to use \texttt{substitute_function()} to replace all occurrences of a function with another:

\begin{Verbatim}
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b)
sage: g
b*diff(cr(a), a)
\end{Verbatim}

(continues on next page)
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))

half = 1/2
sec = sec
sech = sech
sin = sin
sinh = sinh
tan = tan
tanh = tanh
two = 2
x = x

class sage.symbolic.expression_conversions.ExpressionTreeWalker(ex)
Bases: Converter
A class that walks the tree. Mainly for subclassing.

EXAMPLES:

sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: f = x*foo(x) + pi/foo(x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.arithmetic(f, f.operator()) == f)
True

arithmetic(ex, operator)

EXAMPLES:

composition(ex, operator)

EXAMPLES:
derivative (ex, operator)
EXAMPLES:
```python
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
ten: foo = function('foo')
sage: f = foo(x).diff(x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.derivative(f, f.operator()) == f)
True
```

pyobject (ex, obj)
EXAMPLES:
```python
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
ten: foo = function('foo')
sage: f = foo(x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.pyobject(f, f.pyobject()) == f.pyobject())
True
```

relation (ex, operator)
EXAMPLES:
```python
ten: foo = function('foo')
sage: eq = foo(x) == x
ten: s = ExpressionTreeWalker(eq)
sage: s.relation(eq, eq.operator()) == eq
True
```

symbol (ex)
EXAMPLES:
```python
ten: foo = function('foo')
sage: s = ExpressionTreeWalker(x)
sage: bool(s.symbol(x) == x)
True
```

tuple (ex)
EXAMPLES:
```python
ten: foo = function('foo')
sage: f = hypergeometric((1,2,3,),(x,),x)
ten: s = ExpressionTreeWalker(f)
sage: bool(s() == f)
True
```

class sage.symbolic.expression_conversions.FakeExpression (operands, operator)
Bases: object
Pynac represents $x/y$ as $xy^{-1}$. Often, tree-walkers would prefer to see divisions instead of multiplications and negative exponents. To allow for this (since Pynac internally doesn’t have division at all), there is a possibility to pass `use_fake_div=True`; this will rewrite an Expression into a mixture of Expression and FakeExpression nodes, where the FakeExpression nodes are used to represent divisions. These nodes are intended to act sufficiently like Expression nodes that tree-walkers won’t care about the difference.

**operands()**

**EXAMPLES:**

```python
 sage: from sage.symbolic.expression_conversions import FakeExpression
 sage: import operator
 sage: x, y = var('x, y')
 sage: f = FakeExpression([x, y], operator.truediv)
 sage: f.operands()
 [x, y]
```

**operator()**

**EXAMPLES:**

```python
 sage: from sage.symbolic.expression_conversions import FakeExpression
 sage: import operator
 sage: x, y = var('x, y')
 sage: f = FakeExpression([x, y], operator.truediv)
 sage: f.operator()
 <built-in function truediv>
```

**pyobject()**

**EXAMPLES:**

```python
 sage: from sage.symbolic.expression_conversions import FakeExpression
 sage: import operator
 sage: x, y = var('x, y')
 sage: f = FakeExpression([x, y], operator.truediv)
 sage: f.pyobject()
 Traceback (most recent call last):
   ...
   TypeError: self must be a numeric expression
```

**class** `sage.symbolic.expression_conversions.FastCallableConverter` *(ex, etb)*

**Bases:** `Converter`

**EXAMPLES:**

```python
 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:**

```python
 sage: from sage.ext.fast_callable import ExpressionTreeBuilder
 sage: etb = ExpressionTreeBuilder(vars=['x', 'y'])
 (continues on next page)```
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
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)
```

**composition** *(ex, function)*

Given an ExpressionTreeBuilder, return an Expression representing this value.

**EXAMPLES:**

```python
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:**

```python
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x'])
pi
sage: etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
sage: pi._fast_callable_(etb)
3.141592653589793
```

**relation** *(ex, operator)*

**EXAMPLES:**

```python
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):
... Not Implemented Error
```

**symbol** *(ex)*

Given an ExpressionTreeBuilder, return an Expression representing this value.

**EXAMPLES:**

```python
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
```

(continues on next page)
Symbolic Calculus, Release 10.3

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
etb = ExpressionTreeBuilder(vars=['x'])
SR._force_pyobject((2, 3, x^2))._fast_callable_(etb)
[2, 3, x^2]
```

class sage.symbolic.expression_conversions.FriCASConverter

Bases: InterfaceInit

Converts any expression to FriCAS.

EXAMPLES:

```sage
var('x,y')
(x, y)
f = exp(x^2) - arcsin(pi+x)/y
f._fricas_()  #...
f(x)._fricas_()  #...
2
x
y %e - asin(x + %pi)
-----------
y
```

derivative(ex, operator)

Convert the derivative of self in FriCAS.

INPUT:

- ex – a symbolic expression
- operator – operator

Note that ex.operator() == operator.

EXAMPLES:

```sage
var('x,y,z')
(x, y, z)
f = function("F")
f(x)._fricas_()  #...
f(x)
F(x)
diff(f(x,y,z), x, z, x)._fricas_()  #...
F(x,y,z)
,i,1,3
```
Check that `github issue #25838` is fixed:

```python
sage: var('x')
x
sage: F = function('F')

sage: integrate(F(x), x, algorithm="fricas")
integral(F(x), x)

sage: integrate(diff(F(x), x)*sin(F(x)), x, algorithm="fricas")
-cos(F(x))
```

Check that `github issue #27310` is fixed:

```python
f = function("F")

sage: var("y")
y
sage: ex = (diff(f(x,y), x, x, y)).subs(y=x+y); ex
D[0, 0, 1](F)(x, x + y)

sage: fricas(ex)
F (x,y + x),1,1,2
```

**pyobject** `(ex, obj)`

Return a string which, when evaluated by FriCAS, returns the object as an expression.

We explicitly add the coercion to the FriCAS domains `ExpressionInteger` and `ExpressionComplexInteger` to make sure that elements of the symbolic ring are translated to these. In particular, this is needed for integration, see `github issue #28641` and `github issue #28647`.

**EXAMPLES:**

```python
sage: 2._fricas_().domainOf() #...

sage: (-1/2)._fricas_().domainOf() #...

sage: SR(2)._fricas_().domainOf() #...

sage: (sqrt(2))._fricas_().domainOf() #...

sage: pi._fricas_().domainOf() #...

sage: asin(pi)._fricas_() #...
```

(continues on next page)
Symbolic Calculus, Release 10.3

sage: I._fricas_().domains()  # optional -- fricas
Complex(Integer())

sage: SR(I)._fricas_().domains()  # optional -- fricas
Expression(Complex(Integer()))

sage: ex = (I+sqrt(2)+2)
sage: ex._fricas_().domains()  # optional -- fricas
Expression(Complex(Integer()))

sage: ex^2._fricas_()  # optional -- fricas
(4 + 2 %i)\|2 + 5 + 4 %i

symbol (ex)

Convert the argument, which is a symbol, to FriCAS.

In this case, we do not return an ExpressionInteger, because FriCAS frequently requires elements of domain Symbol or Variable as arguments, for example to integrate. Moreover, FriCAS is able to do the conversion itself, whenever the argument should be interpreted as a symbolic expression.

EXAMPLES:

class sage.symbolic.expression_conversions.HoldRemover (ex, exclude=None)

Bases: ExpressionTreeWalker

A class that walks the tree and evaluates every operator that is not in a given list of exceptions.

EXAMPLES:

sage: from sage.symbolic.expression_conversions import HoldRemover
sage: ex = sin(pi*cos(0, hold=True), hold=True); ex
sin(pi*cos(0))
sage: h = HoldRemover(ex)
sage: h()
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
sage: h = HoldRemover(ex, [sin])
sage: h()
sin(pi)
sage: h = HoldRemover(ex, [cos])
sage: h()
sin(pi*cos(0))
sage: ex = atan2(0, 0, hold=True) + hypergeometric([1,2], [3,4], 0, hold=True)
sage: h = HoldRemover(ex, [atan2])
sage: h()
arctan2(0, 0) + 1
sage: h = HoldRemover(ex, [hypergeometric])
sage: h()
NaN + hypergeometric((1, 2), (3, 4), 0)
```

**composition (ex, operator)**

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import HoldRemover
sage: ex = sin(pi*cos(0, hold=True), hold=True); ex
sin(pi*cos(0))
sage: h = HoldRemover(ex)
sage: h()
0
```

**class** `sage.symbolic.expression_conversions.InterfaceInit` *(interface)*

**Bases:** `Converter`

**EXAMPLES:**

```python
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:**

```python
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.arithmetic(x+2, sage.symbolic.operators.add_vararg)
(_SAGE_VAR_x)+(2)
```

**composition (ex, operator)**

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.composition(sin(x), sin)
'sin(_SAGE_VAR_x)'
```

(continues on next page)
Symbolic Calculus, Release 10.3

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
diff(f(x), x)
sage: print(m.derivative(a, a.operator()))
diff(f(_SAGE_VAR_x), _SAGE_VAR_x, 1)
sage: b = f(x).diff(x, x)
sage: print(m.derivative(b, b.operator()))
diff('f(_SAGE_VAR_x), _SAGE_VAR_x, 2)

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__symbol0), _SAGE_VAR__symbol0, 1), [_SAGE_VAR__symbol0 → (_SAGE_VAR_x)*(_SAGE_VAR_y)])" 

pyobject (ex, obj)

EXAMPLES:

sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: ii = InterfaceInit(gp)
sage: f = 2+SR(I)
sage: ii.pyobject(f, f.pyobject())
I + 2
sage: ii.pyobject(SR(2), 2)
2
sage: ii.pyobject(pi, pi.pyobject())
Pi

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

(continues on next page)
symbol(ex)

EXAMPLES:

sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.symbol(x)
_SAGE_VAR_x
sage: f(x) = x
sage: m.symbol(f)
_SAGE_VAR_x
sage: ii = InterfaceInit(gp)
sage: ii.symbol(x)
'x'
sage: g = InterfaceInit(giac)
sage: g.symbol(x)
sageVARx

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.LaurentPolynomialConverter(ex, base_ring=None, ring=None)

Bases: PolynomialConverter

A converter from symbolic expressions to Laurent polynomials.

See laurent_polynomial() for details.

class sage.symbolic.expression_conversions.PolynomialConverter(ex, base_ring=None, ring=None)

Bases: Converter

A converter from symbolic expressions to polynomials.

See polynomial() for details.

EXAMPLES:

sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: x, y = var('x,y')
sage: p = PolynomialConverter(x+y, base_ring=QQ)
sage: p.base_ring
Rational Field
sage: p.ring
Multivariate Polynomial Ring in x, y over Rational Field

sage: p = PolynomialConverter(x, base_ring=QQ)
Symbolic Calculus, Release 10.3

```
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...
```

**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)
5.85978742682568
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^1.5
```

**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())
```
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: 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)
5.85974482048842
sage: loads(dumps(R))
<sage.symbolic.expression_conversions.RingConverter object at 0x...>
### Symbolic Calculus

**arithmetic** \((\text{ex}, \text{operator})\)

**EXAMPLES:**

```python
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** \((\text{ex}, \text{operator})\)

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(cos(2))
-0.4161468365471424?
```

**pyobject** \((\text{ex}, \text{obj})\)

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(SR(5/2))
2.5000000000000000?
```

**symbol** \((\text{ex})\)

All symbols appearing in the expression must either appear in \(\text{subs_dict}\) or be convertible by the ring’s element constructor in order for the conversion to be successful.

**EXAMPLES:**

```python
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF, subs_dict={x:2})
sage: R(x+pi)
5.141592653589794?

sage: R = RingConverter(RIF)
sage: R(x+pi)
Traceback (most recent call last):
...
TypeError: unable to simplify to a real interval approximation
```

### Class

**class** `sage.symbolic.expression_conversions.SubstituteFunction(ex, *args)`

**Bases:** `ExpressionTreeWalker`

A class that walks the tree and replaces occurrences of a function with another.

**EXAMPLES:**

---

Chapter 2. Internal functionality supporting calculus
Symbolic Calculus, Release 10.3

```
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)
```

**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())
diff(bar(x), x)
```

```
sage.symbolic.expression_conversions.fast_callable(ex, etb)
```

Given an ExpressionTreeBuilder *etb*, return an Expression representing the symbolic expression *ex*.

**EXAMPLES:**

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=[x,y])
sage: x, y = var(x, y)
sage: f = y+2*x^2
sage: f._fast_callable_(etb)
add(mul(ipow(v_0, 2), 2), v_1)
sage: f = (2*x^3+2*x-1)/((x-2)*(x+1))
sage: f._fast_callable_(etb)
div(add(add(mul(ipow(v_0, 3), 2), mul(v_0, 2)), -1), mul(add(v_0, 1), add(v_0, -2)))
```

```
sage.symbolic.expression_conversions.laurent_polynomial(ex, base_ring=None, ring=None)
```

Return a Laurent polynomial from the symbolic expression *ex*.

**INPUT:**

- *ex* – a symbolic expression

2.16. Conversion of symbolic expressions to other types 273
- `base_ring`, `ring` — Either a `base_ring` or a Laurent polynomial `ring` can be specified for the parent of result. If just a `base_ring` is given, then the variables of the `base_ring` will be the variables of the expression `ex`.

**OUTPUT:**

A Laurent polynomial.

**EXAMPLES:**

```sage
definitions:
from sage.symbolic.expression_conversions import laurent_polynomial

sage: f = x^2 + 2/x
sage: laurent_polynomial(f, base_ring=QQ)
2*x^-1 + x^2
sage: _.parent()
Univariate Laurent Polynomial Ring in x over Rational Field

sage: laurent_polynomial(f, ring=LaurentPolynomialRing(QQ, 'x', y))
x^2 + 2*x^-1
sage: _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field

sage: x, y = var('x, y')
sage: laurent_polynomial(x + 1/y^2, ring=LaurentPolynomialRing(QQ, 'x, y'))
x + y^-2
sage: _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field
```

`sage.symbolic.expression_conversions.polynomial`(*ex, *base_ring=*, *ring=*)

Return a polynomial from the symbolic expression `ex`.

**INPUT:**

- `ex` — a symbolic expression
- `base_ring`, `ring` — Either a `base_ring` or a polynomial `ring` can be specified for the parent of result. If just a `base_ring` is given, then the variables of the `base_ring` will be the variables of the expression `ex`.

**OUTPUT:**

A polynomial.

**EXAMPLES:**

```sage
definitions:
from sage.symbolic.expression_conversions import polynomial

sage: f = x^2 + 2
sage: polynomial(f, base_ring=QQ)
x^2 + 2
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field

sage: polynomial(f, ring=QQ['x,y'])
x^2 + 2
sage: _.parent()
Multivariate Polynomial Ring in x, y over Rational Field

sage: x, y = var('x, y')
sage: polynomial(x + y^2, ring=QQ['x,y'])
y^2 + x
sage: _.parent()
```

(continues on next page)
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

2.17 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.

\texttt{sage.symbolic.complexity_measures.string\_length(expr)}

Returns the length of \texttt{expr} after converting it to a string.

INPUT:

\begin{itemize}
  \item \texttt{expr} – the expression whose complexity we want to measure.
\end{itemize}

OUTPUT:

A real number representing the complexity of \texttt{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:

\begin{verbatim}
\sage: from sage.symbolic.complexity_measures import string_length
\sage: f = x^2
\sage: string_length(f)
3
\end{verbatim}
2.18 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)
30410990296096435874423135622368807751920779408261498180000000000000
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: 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: a
1/7
sage: 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).
# The Maxima backend equality checker does this;
# note the equality only holds for one choice of sign,
# but Maxima always chooses the "positive" one
sage: a = sqrt(2*sqrt(3) + 4); b = 1 + sqrt(3)
sage: float(a-b)
0.0
sage: bool(a == b)
True
```

```
sage: # We can, of course, do this in a quadratic field
sage: k.<sqrt3> = QuadraticField(3)
sage: asqr = 2*sqrt3 + 4
sage: b = 1+sqrt3
```

(continues on next page)
\begin{verbatim}
sage: asqr == b^2
True

sage: # (YES) Sqrt(14+3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2))))))=3+Sqrt(2).
sage: a = sqrt(14+3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2)))))
sage: b = 3+sqrt(2)
sage: a, b
(sqrt(3*sqrt(2*sqrt(-12*sqrt(-2*sqrt(2) + 3) + 5) + 3) + 14), sqrt(2) + 3)
sage: bool(a==b)
True
sage: abs(float(a-b)) < 1e-10
True
sage: # 2*Infinity-3=Infinity.
sage: 2*infinity-3 == infinity
True
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 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
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^12 + 167960*x^11 + 184756*x^10 + 173160*x^9 + 135135*x^8 + 88176*x^7 + 48960*x^6 + 23760*x^5 + 9240*x^4 + 2520*x^3 + 454*x^2 + 49*x + 1
\end{verbatim}
Symbolic Calculus, Release 10.3

\[ \begin{align*}
-13 + 125970x^{12} + 167960x^{11} + 184756x^{10} + 167960x^9 + 125970x^8 + 77520x^7 \\
+ 38760x^6 + 15504x^5 + 4845x^4 + 1140x^3 + 190x^2 + 20x + 1
\end{align*} \]

sage: deriv = f.derivative()
sage: deriv
\[ 20x^{19} + 380x^{18} + 3420x^{17} + 19380x^{16} + 542640x^{15} + 232560x^{14} + 542640x^{13} \\
+ 1007760x^{12} + 1511640x^{11} + 1847560x^{10} + 1847560x^9 + 1511640x^8 + 1007760x^7 \\
+ 542640x^6 + 232560x^5 + 77520x^4 + 19380x^3 + 3420x^2 + 380x + 20 \]

sage: deriv.factor()
\[ 20(x + 1)^{19} \]

sage: restore(x)

sage: # next do it symbolically

sage: var('y')
y

sage: f = (1+y)^20; f
\[(y + 1)^{20}\]

sage: g = f.expand(); g
\[ y^{20} + 20y^{19} + 190y^{18} + 1140y^{17} + 4845y^{16} + 15504y^{15} + 38760y^{14} + 77520y^{13} \\
+ 125970y^{12} + 167960y^{11} + 184756y^{10} + 167960y^9 + 125970y^8 + 77520y^7 \\
+ 38760y^6 + 15504y^5 + 4845y^4 + 1140y^3 + 190y^2 + 20y + 1 \]

sage: deriv = g.derivative(); deriv
\[ 20y^{19} + 380y^{18} + 3420y^{17} + 19380y^{16} + 542640y^{15} + 232560y^{14} + 542640y^{13} \\
+ 1007760y^{12} + 1511640y^{11} + 1847560y^{10} + 1847560y^9 + 1511640y^8 + 1007760y^7 \\
+ 542640y^6 + 232560y^5 + 77520y^4 + 19380y^3 + 3420y^2 + 380y + 20 \]

sage: deriv.factor()
\[ 20(y + 1)^{19} \]

sage: # (YES) Factorize \(x^{100}-1\).

sage: factor(x^{100}-1)
\[(x^{40} - x^{30} + x^{20} - x^{10} + 1)(x^{20} + x^{15} + x^{10} + x^5 + 1)(x^{20} - x^{15} + x^{10} - 1)\]

sage: # Also, algebraically

sage: x = polygen(QQ)

sage: factor(x^{100} - 1)
\[(x - 1)(x + 1)(x^2 + 1)(x^4 - x^3 + x^2 - x + 1)(x^4 + x^3 + x^2 + x + 1)\]

sage: # (YES) Factorize \(x^4-3x^2+1\) in the field of rational numbers extended by roots of \(x^2-x-1\).

sage: x = polygen(ZZ, 'x')

sage: k.< a> = NumberField(x^2 - x - 1)

sage: R.< y> = k[]

sage: f = y^4 - 3*y^2 + 1

sage: f
\[ y^4 - 3y^2 + 1 \]

sage: factor(f)
\[(y - a)(y + a)(y^2 - y + 1) \]

sage: # (YES) Factorize \(x^4-3x^2+1\) mod 5.

sage: k.< x > = GF(5) []

sage: f = x^4 - 3*x^2 + 1

sage: f.factor()
\[(x - 3)(x + 3)(x^2 + 1) \]

(continues on next page)
(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: bool(x==z)
True

sage: # (YES) Assuming x>y, y>0, deduce 2*x^2>2*y^2.
sage: forget()
sage: assume(x>y, y>0)
sage: sorted(assumptions())
[x > y, y > 0]
sage: bool(2*x^2 > 2*y^2)
True
sage: forget()
sage: assumptions()
[]

sage: # (NO) Solve the inequality Abs(x-1)>2.
sage: # Maxima doesn't solve inequalities
sage: # (but some Maxima packages do):
sage: eqn = abs(x-1) > 2
sage: eqn.solve(x)
[[x < -1], [3 < x]]

sage: # (NO) Solve the inequality (x-1)*...*(x-5)<0.
sage: eqn = prod(x-i for i in range(1,5 +1)) < 0
sage: # but don't know how to solve
sage: eqn.solve(x)
[[x < 1], [x > 2, x < 3], [x > 4, x < 5]]

sage: # (YES) Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^2 or similar equivalent combination.
sage: f = cos(3*x)/cos(x)
sage: g = cos(x)^2 - 3*sin(x)^2
sage: h = f-g
sage: h.trig_simplify()
0

sage: # (YES) Cos(3*x)/Cos(x)=2*Cos(2*x)-1.
sage: f = cos(3*x)/cos(x)
```
sage: g = 2*cos(2*x) - 1
sage: h = f-g
sage: h.trig_simplify()
    0
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 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: reset(x)
sage: k.<b> = NumberField(x^3-2)
sage: a = (b + b^2)^3 - 6*(b + b^2) - 6
sage: bool(a == 0)
    True
```
```
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]
```
```
(continues on next page)
```
Sage: # Numerically, the expression \( \ln(\tan(x/2+\pi/4)) - \text{ArcSinh}(\tan(x)) = 0 \) and its...
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) \( \ln((2*\text{sqrt}(r) + 1)/\sqrt{4*r + 4*\text{sqrt}(r) + 1}) = 0 \).
Sage: f = log((2*sqrt(r) + 1) / sqrt(4*r + 4*sqrt(r) + 1)); f
log((2*sqrt(r) + 1)/sqrt(4*r + 4*sqrt(r) + 1))
Sage: bool(f == 0)
False
Sage: [abs(float(f(r=i))) < 1e-10 for i in [0.1, 0.3, 0.5]]
[True, True, True]
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()
sin(2*real_part(z))/(cos(2*real_part(z)) + \cosh(2*imag_part(z)))
Sage: a.imag()
sinh(2*imag_part(z))/(cos(2*real_part(z)) + \cosh(2*imag_part(z)))
Sage: # (YES) Simplify \( \ln(\text{Exp}(z)) \) to \( z \) for \(-\pi < \text{Im}(z) <= \pi\).
Sage: # Unfortunately (?), Maxima does this even without
Sage: # any assumptions.
Sage: # We *would* use assume(-\pi < \text{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
domain.
sage: # To stick with the behaviour of previous versions, the domain is set
to 'real' in the following.
sage: # See Issue #10682 for further details.
sage: n = var('n')
sage: f = x^(1/n)*y^(1/n)-(x*y)^(1/n)
sage: assume(real(x) > 0, real(y) > 0)
sage: f.simplify()
x^(1/n)*y^(1/n) - (x*y)^(1/n)
sage: maxima = sage.calculus.calculus.maxima
# 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: zz


```python
sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b, c, d.
sage: var('a,b,c,d')
(a, b, c, d)
sage: m = matrix(SR, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: m
[ 1 a a^2 a^3]
[ 1 b b^2 b^3]
[ 1 c c^2 c^3]
[ 1 d d^2 d^3]
sage: d = m.determinant()
sage: d.factor()
(a - b)*(a - c)*(a - d)*(b - c)*(b - d)*(c - d)
```

```python
sage: # (YES) Do it instead in a multivariate ring
sage: R.<a,b,c,d> = QQ[]
sage: m = matrix(R, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: m
[ 1 a a^2 a^3]
[ 1 b b^2 b^3]
[ 1 c c^2 c^3]
[ 1 d d^2 d^3]
sage: d = m.determinant()
sage: d
a^3*b^2*c - a^2*b^3*c - a^3*b*c^2 + a*b^3*c^2 + a^2*b*c^3 - a*b^2*c^3 - a^3*b^2*d + a^2*b^3*d + a^3*c^2*d - b^3*c^2*d - a^2*c^3*d + b^2*c^3*d + a^3*b*d^2 - a*b^3*d^2 - a^3*c*d^2 + b^2*c*d^2 - a^2*b*d^3 + a*b^2*d^3 + a^2*c*d^3 - b^2*c*d^3
sage: d.factor()
(-1) * (c - d) * (-b + c) * (b - d) * (-a + c) * (-a + b) * (a - d)
```

```python
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])]
```

```python
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)
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

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))
1/2*(x + conjugate(x))/abs(x)
sage: _.simplify_full()
x/abs(x)
sage: _ = var('x', domain='real')
sage: diff(abs(x))
x/abs(x)
sage: forget()

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)
# if(x<0) (-x) else x),
# 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 Tan(x)).
sage: # TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
# (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 - 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 +...

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

\[
-3)\cdot a^3(x - 1)^3 \cdot e^{(-b)} + 1/24 \cdot (3a^2(x - 1)^a + a^2(12b + 5)(x - 1)^a + \\
-12 \cdot b^2(x - 1)^a(x - 1)^2 \cdot e^{(-b)} - 1/2 \cdot (a^2(x - 1)^a + 2b^2(x - 1)^a)(x - 1) \cdot e^{(-b)} + \\
(x - 1)^a \cdot e^{(-b)}
\]

declares the expansion of \(\ln(\sin(x)/x)\) at \(x=0\).

```python
taylor(log(sin(x)/x), x, 0, 10)
```

\[-1/467775\cdot x^{10} - 1/37800\cdot x^{8} - 1/2835\cdot x^{6} - 1/180\cdot x^{4} - 1/6\cdot x^{2}\]

Computes the \(n\)-th term of the Taylor series of \(\ln(\sin(x)/x)\) at \(x=0\).

```python
# (Sort of, with some work)
# Need formal functions
```

Computes the \(n\)-th term of the Taylor series of \(\exp(-x)\cdot \sin(x)\) at \(x=0\).

```python
# (Sort of, with some work)
# Need formal functions
```

Solve \(x=\sin(y)+\cos(y)\) for \(y\) as Taylor series in \(x\) at \(x=1\).

```python
# TestYacas(InverseTaylor(y, 0, 4) Sin(y)+Cos(y),
#    (y-1)+(y-1)^2/2+(y-1)^3/3+(y-1)^4);
# Note that InverseTaylor does not give the series in terms of \(x\) but in...
```

```python
f = sin(y) + cos(y)
g = f.taylor(y, 0, 10)
h = g.power-series(QQ)
k = (h - 1).reverse()
y + 1/2*y^2 + 2/3*y^3 + y^4 + 17/10*y^5 + 37/12*y^6 + 41/7*y^7 + 23/2*y^8 + 1667/72*y^9 + 3803/80*y^10 + O(y^11)
```

Computes Legendre polynomials directly from Rodrigues's formula, \(P[n]=\frac{1}{(2^n*n!)} \cdot \frac{\text{Deriv}(x,n)(x^2-1)^n)}{n} \).

```python
# (OK) Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/
# (2^n*n!) * (Deriv(x,n)(x^2-1)^n).
# P(n,x) := Simplify( 1/(2^n)!! * 
# Deriv(x,n) (x^2-1)^n );
# TestYacas(P(4,x), (35*x^4)/8+(15*x^2)/4+3/8);
```

```python
P = lambda n, x: simplify(diff((x^2-1)^n,x,n) / (2^n * factorial(n)))
P(4,x).expand()
35/8*x^4 - 15/4*x^2 + 3/8
```

Define the polynomial \(p=\sum(i,1,5,a[i]*x^i)\).

```python
# (YES) Define the polynomial p=\sum(i,1,5,a[i]*x^i).
# Symbolically
```

```python
ps = sum(var('a%s%i')*x^i for i in range(1,6)); ps
a^5*x^5 + a^4*x^4 + a^3*x^3 + a^2*x^2 + a^1*x
```

```python
ps.parent()
Symbolic Ring
```

```python
# algebraically
R = PolynomialRing(QQ,5,names='a')
S.<x> = PolynomialRing(R)
p = S(list(R.gens()))*x; p
a^4*x^5 + a^3*x^4 + a^2*x^3 + a^1*x^2 + a^0*x
```

```python
p.parent()
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a0, a1, a2, a3,...
...a4 over Rational Field
```

Convert the above to Horner's form.

```python
# (YES) Convert the above to Horner's form.
# Verify(Horner(p, x), ((((a[5]*x+a[4])*x
# +a[3])*x+a[2])*x+a[1])*x);
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```
sage: restore('x')
sage: SR(p).horner(x)
(((a4*x + a3)*x + a2)*x + a1)*x + a0)*x
```

```
sage: # (NO) Convert the result of problem 127 to Fortran syntax.
sage: # CForm(Horner(p, x));
```

```
sage: # (YES) Verify that True And False=False.
sage: (True and False) is False
True
```

```
sage: # (YES) Prove x Or Not x.
sage: for x in [True, False]:
    ....:     print(x or (not x))
True
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!"))
```

2.19 Solving ordinary differential equations

This file contains functions useful for solving differential equations which occur commonly in a 1st semester differential equations course. For another numerical solver see the ode_solver() function and the optional package Octave.

Solutions from the Maxima package can contain the three constants _C, _K1, and _K2 where the underscore is used to distinguish them from symbolic variables that the user might have used. You can substitute values for them, and make them into accessible usable symbolic variables, for example with var("_C").

Commands:
- desolve() - Compute the “general solution” to a 1st or 2nd order ODE via Maxima.
- desolve_laplace() - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional.
- desolve_rk4() - Solve numerically an IVP for one first order equation, return list of points or plot.
- desolve_system_rk4() - Solve numerically an IVP for a system of first order equations, return list of points.
- desolve_odeint() - Solve numerically a system of first-order ordinary differential equations using odeint from scipy.integrate module.
- desolve_system() - Solve a system of 1st order ODEs of any size using Maxima. Initial conditions are optional.
- eulers_method() - Approximate solution to a 1st order DE, presented as a table.
- eulers_method_2x2() - Approximate solution to a 1st order system of DEs, presented as a table.
- eulers_method_2x2_plot() - Plot the sequence of points obtained from Euler's method.

The following functions require the optional package tides:
**desolve_mintides()** - Numerical solution of a system of 1st order ODEs via the Taylor series integrator method implemented in TIDES.

**desolve_tides_mpfr()** - Arbitrary precision Taylor series integrator implemented in TIDES.

**AUTHORS:**
- David Joyner (3-2006) - Initial version of functions
- Marshall Hampton (7-2007) - Creation of Python module and testing
- Robert Marik (10-2009) - Some bugfixes and enhancements
- Miguel Marco (06-2014) - Tides desolvers

```python
sage.calculus.desolvers.desolve(de, dvar, ics=None, ivar=None, show_method=False, contrib_ode=False, algorithm='maxima')
```

Solve a 1st or 2nd order linear ODE, including IVP and BVP.

**INPUT:**
- `de` – an expression or equation representing the ODE
- `dvar` – the dependent variable (hereafter called `y`)
- `ics` – (optional) the initial or boundary conditions
  - for a first-order equation, specify the initial `x` and `y`
  - for a second-order equation, specify the initial `x`, `y`, and `dy/dx`, i.e. write `[x_0, y(x_0), y'(x_0)]`
  - for a second-order boundary solution, specify initial and final `x` and `y` boundary conditions, i.e. write `[x_0, y(x_0), x_1, y(x_1)]`.
  - gives an error if the solution is not SymbolicEquation (as happens for example for a Clairaut equation)
- `ivar` – (optional) the independent variable (hereafter called `x`), which must be specified if there is more than one independent variable in the equation
- `show_method` – (optional) if True, then Sage returns pair `[solution, method]`, where method is the string describing the method which has been used to get a solution (Maxima uses the following order for first order equations: linear, separable, exact (including exact with integrating factor), homogeneous, bernoulli, generalized homogeneous) - use carefully in class, see below the example of an equation which is separable but this property is not recognized by Maxima and the equation is solved as exact.
- `contrib_ode` – (optional) if True, desolve allows to solve Clairaut, Lagrange, Riccati and some other equations. This may take a long time and is thus turned off by default. Initial conditions can be used only if the result is one SymbolicEquation (does not contain a singular solution, for example).
- `algorithm` – (default: 'maxima') one of
  - 'maxima' - use maxima
  - 'fricas' - use FriCAS (the optional fricas spkg has to be installed)

**OUTPUT:**

In most cases return a SymbolicEquation which defines the solution implicitly. If the result is in the form `y(x) = ...` (happens for linear eqs.), return the right-hand side only. The possible constant solutions of separable ODEs are omitted.

**Note:** Use `desolve? <tab>` if the output in the Sage notebook is truncated.
EXAMPLES:

```sage
sage: x = var('x')
sage: y = function('y')(x)
sage: desolve(diff(y,x) + y - 1, y)
(_C + e^x) * e^(-x)
```

```sage
sage: f = desolve(diff(y,x) + y - 1, y, ics=[10,2]); f
(e^10 + e^x) * e^(-x)
```

```sage
sage: plot(f)
Graphics object consisting of 1 graphics primitive
```

We can also solve second-order differential equations:

```sage
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
sage: f = desolve(de, y, [10,2,1]); f
-x + 7*e^(x - 10) + 5*e^(-x + 10)
```

```sage
sage: f(x=10)
2
```

```sage
sage: diff(f,x)(x=10)
1
```

```sage
sage: de = diff(y,x,2) + y == 0
sage: desolve(de, y)
_K2*cos(x) + _K1*sin(x)
```

```sage
sage: desolve(de, y, [0,1,pi/2,4])
 cos(x) + 4*sin(x)
```

```sage
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
sage: desolve(diff(y,x)^2+x*diff(y,x)-y==0,y,contrib_ode=True,show_method=True)
[[y(x) == _C^2 + _C*x, y(x) == -1/4*x^2], 'clairau...']
```

For equations involving more variables we specify an independent variable:

```sage
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
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 True."
```

Another difficult ODE with error - moreover, it takes a long time:
Some more types of ODEs:

```sage
desolve(x*diff(y,x)^2-(1+x*y)*diff(y,x)+y==0,y,contrib_ode=True,show_method=True)
[[[x == _C - arctan(sqrt(t)), y(x) == -x - sqrt(t)], [x == _C + arctan(sqrt(t)), y(x) == -x + sqrt(t)]]], 'lagrange']
```

These two examples produce an error (as expected, Maxima 5.18 cannot solve equations from initial conditions). Maxima 5.18 returns false answer in this case!

```sage
desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,1])
1/2*(8*x + 6)*e^(-x) + 1/2*sin(x)
```

Second order linear ODE:

```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 == 0,y,show_method=True)
[(_K2*x + _K1)*e^(-x), 'constcoeff']
```
Symbolic Calculus, Release 10.3

**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)

\([(4*x + 3)*e^(-x), \text{\texttt{constcoeff}}]\)

**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)

\([(2*x*(2*e^{1/2*pi} - 3)/pi + 3)*e^(-x), \text{\texttt{constcoeff}}]\)

Using **algorithm='fricas'** we can invoke the differential equation solver from FriCAS. For example, it can solve higher order linear equations:

**sage:** de = x^3*diff(y, x, 3) + x^2*diff(y, x, 2) - 2*x*diff(y, x) + 2*y - 2*x^4

**sage:** Y = desolve(de, y, algorithm="fricas"); Y

\(\left(\frac{2*x^3 - 3*x^2 + 1}{x^2}*_C0/x + (x^3 - 1)*_C1/x + (x^3 - 3*x^2 - 1)*_C2/x + 1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x\right)\)

The initial conditions are then interpreted as \([x_0, y(x_0), y'(x_0), \ldots, y^{(n)}(x_0)]\):

**sage:** Y = desolve(de, y, ics=[1,3,7], algorithm="fricas"); Y

\(\left(\frac{1}{15}*(x^5 + 15*x^3 + 50*x^2 - 21)/x\right)\)

FriCAS can also solve some non-linear equations:

**sage:** de = diff(y, x) == y / (x+y*log(y))

**sage:** Y = desolve(de, y, algorithm="fricas"); Y

\(\left(\frac{1/2*(log(y(x))^2*y(x) - 2*x)}{y(x)}\right)\)

**AUTHORS:**

- David Joyner (1-2006)
- Robert Bradshaw (10-2008)
- Robert Marik (10-2009)

**sage.calculus.desolvers.desolve_laplace** (de, dvar=None, ics=\texttt{None}, ivar=None)

Solve an ODE using Laplace transforms. Initial conditions are optional.

**INPUT:**

- **de** - a lambda expression representing the ODE (e.g. \texttt{de = diff(y, x, 2) == diff(y, x)+sin(x))}
- **dvar** - the dependent variable (e.g. \texttt{y})
- **ivar** - (optional) the independent variable (hereafter called \texttt{x}), which must be specified if there is more than one independent variable in the equation.
- **ics** - a list of numbers representing initial conditions, (e.g. \texttt{f(0)=1, f'(0)=2} corresponds to \texttt{ics = [0,1,2]})

**OUTPUT:**

Solution of the ODE as symbolic expression

**EXAMPLES:**
Symbolic Calculus, Release 10.3

\begin{verbatim}
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

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:

• \texttt{f} – symbolic function. Its first argument will be the independent variable. Its output should be de derivatives of the dependent variables.

• \texttt{ics} – a list or tuple with the initial conditions.

• \texttt{initial} – the starting value for the independent variable.

• \texttt{final} – the final value for the independent value.

• \texttt{delta} – the size of the steps in the output.

• \texttt{tolrel} – the relative tolerance for the method.

• \texttt{tolabs} – the absolute tolerance for the method.

OUTPUT:

• A list with the positions of the IVP.
\end{verbatim}
EXAMPLES:

We integrate a periodic orbit of the Kepler problem along 50 periods:

```sage
var('t,x,y,X,Y')
f(t,x,y,X,Y)=|X, Y, -x/(x^2+y^2)^(3/2), -y/(x^2+y^2)^(3/2)|
ics = [0.8, 0, 0, 1.22474487139159]
t = 100*pi
sol = desolve_rk4(f, ics, 0, t, t, 1e-12, 1e-12) # optional -tides
sol
```

ALGORITHM:

Uses TIDES.

REFERENCES:


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 (c / 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.

- tcr: 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:

```python
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])
```

Lorenz Equations:

```python
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:
Symbolic Calculus, Release 10.3

```python
sage: y = var('y')
sage: epsilon = 0.01
sage: f = y^2*(1-y)
sage: ic = epsilon
sage: t = srange(0,2/epsilon,1)
sage: sol = desolve_odeint(f, ic, t, y, #
needs scipy
....:     rtol=1e-9, atol=1e-10, compute_jac=True)
sage: p = points(zip(t,sol[:,0])) #
needs scipy sage.plot
sage: p.show() #
needs scipy sage.plot
```

Another stiff system with some optional parameters with no default value:

```python
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, #
needs scipy
....:     h0=0.1, hmax=1, hmin=1e-4, mxstep=1000, mxords=17)
```

**AUTHOR:**

- Oriol Castejon (05-2010)

sage.calculus.desolvers.desolve_rk4 (de, dvar=ics=None, ics=None, end_points=None, step=0.1, output='list', **kwds)

Solve numerically one first-order ordinary differential equation.

**INPUT:**

Input is similar to desolve command. The differential equation can be written in a form close to the plot_slope_field or desolve command.

- **Variant 1** (function in two variables)
  - de - right hand side, i.e. the function \( f(x, y) \) from ODE \( y' = f(x, y) \)
  - dvar - dependent variable (symbolic variable declared by var)

- **Variant 2** (symbolic equation)
  - de - equation, including term with \( \text{diff}(y, x) \)
  - dvar - dependent variable (declared as function of independent variable)

- **Other parameters**
  - ics - initial conditions in the form \([x0, y0]\)
  - end_points - the end points of the interval
    - if \( \text{end_points} \) is a or \([a]\), we integrate between \( \min(ics[0], a) \) and \( \max(ics[0], a) \)
    - if \( \text{end_points} \) is None, we use \( \text{end_points}=ics[0]+10 \)
Symbolic Calculus, Release 10.3

* if end_points is [a, b] we integrate 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.

See also:

ode_solver()

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, 1.124191274245578], [1.0, 1.4615901622888197]]

Variant 1 for input - we can pass ODE in the form used by desolve function In this example we integrate backwards, since end_points < ics[0]:

sage: y = function('y')(x)
sage: desolve_rk4(diff(y,x)+y*(y-1) == x-2, y, ics=[1,1], step=0.5, end_points=0)
[[[0.0, 8.9042571089621135], [0.5, 1.9093279453615209], [1.0, 1.0]]

Here we show how to plot simple pictures. For more advanced applications use list_plot instead. To see the resulting picture use show(P) in Sage notebook.

sage: x, y = var('x, y')
sage: P=desolve_rk4(y*(2-y), y, ics=[0, .1], ivar=x, output='slope_field', end_points=[-4, 6], thickness=3)

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)
Symbolic Calculus, Release 10.3

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=None, ics=None, ivar=None, algorithm='maxima')

Solve a system of any size of 1st order ODEs. Initial conditions are optional.

One dimensional systems are passed to desolve_laplace().

INPUT:

• des – list of ODEs

• vars – list of dependent variables

• ics – (optional) list of initial values for ivar and vars; if ics is defined, it should provide initial conditions for each variable, otherwise an exception would be raised

• ivar – (optional) the independent variable, which must be specified if there is more than one independent variable in the equation

• algorithm – (default: 'maxima') one of
  – 'maxima' - use maxima
  – 'fricas' - use FriCAS (the optional fricas spkg has to be installed)

EXAMPLES:

sage: t = var('t')
sage: x = function('x')(t)
sage: y = function('y')(t)
sage: de1 = diff(x,t) + y - 1 == 0
sage: de2 = diff(y,t) - x + 1 == 0
sage: desolve_system([de1, de2], [x,y])
[x(t) == (x(0) - 1)*cos(t) - (y(0) - 1)*sin(t) + 1, y(t) == (y(0) - 1)*cos(t) + (x(0) - 1)*sin(t) + 1]

The same system solved using FriCAS:

sage: desolve_system([de1, de2], [x,y], algorithm='fricas') # optional - fricas
[x(t) == _C0*cos(t) + cos(t)^2 + _C1*sin(t) + sin(t)^2, y(t) == -_C1*cos(t) + _C0*sin(t) + 1]

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

AUTHORS:
Symbolic Calculus, Release 10.3

- 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 \texttt{rk}.

**INPUT:**

input is similar to \texttt{desolve_system} and \texttt{desolve_rk4} commands

- \texttt{des} - right hand sides of the system
- \texttt{vars} - dependent variables
- \texttt{ivar} - (optional) should be specified, if there are more variables or if the equation is autonomous and the independent variable is missing
- \texttt{ics} - initial conditions in the form \([x_0, y_{01}, y_{02}, y_{03}, ...]\)
- \texttt{end_points} - the end points of the interval
  - if \texttt{end_points} is a or \([a]\), we integrate on between \texttt{min(ics[0], a)} and \texttt{max(ics[0], a)}
  - if \texttt{end_points} is None, we use \texttt{end_points=ics[0]+10}
  - if \texttt{end_points} is \([a,b]\) we integrate on between \texttt{min(ics[0], a)} and \texttt{max(ics[0], b)}
- \texttt{step} - (optional, default: 0.1) the length of the step

**OUTPUT:**

Return a list of points.

**See also:**

\texttt{ode_solver()}

**EXAMPLES:**

```
sage: from sage.calculus.desolvers import desolve_system_rk4

Lotka Volterra system:
```

```
sage: x, y, t = var('x y t')
sage: P = desolve_system_rk4([x*(1-y), -y*(1-x)], [x, y], ics=[0, 0.5, 2],...:
                        ivar=t, end_points=20)
sage: Q = [[i,j] for i,j,k in P]
sage: LP = list_plot(Q)  # needs sage.plot
```

**ALGORITHM:**

4th order Runge-Kutta method. Wrapper for command \texttt{rk} in Maxima’s dynamics package. Perhaps could be faster by using \texttt{fast_float} instead.

**AUTHOR:**
Sage can solve numerically a system of first order differential equations using the taylor series integrator in arbitrary precision implemented in tides.

INPUT:

- `f` — symbolic function. Its first argument will be the independent variable. Its output should be de derivatives of the dependent variables.
- `ics` — 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 precision used in the computation.

OUTPUT:

- A list with the positions of the IVP.

EXAMPLES:

We integrate the Lorenz equations with Saltzman values for the parameters along 10 periodic orbits with 100 digits of precision:

```python
sage: var('t,x,y,z')
(t, x, y, z)
sage: f = s*(y-x), x*(r-z)-y,x*y-b*z
sage: x0 = -13.
sage: y0 = -19.
sage: z0 = 27
sage: T = 15.
sage: sol = desolve_tides_mpfr(f, [x0, y0, z0], 0, T, T, 1e-100, 1e-100, 100)
```

(continues on next page)
ALGORITHM:

Uses TIDES.

**Warning:** This requires the package tides.

**REFERENCES:**

- [ABBR2011]
- [ABBR2012]

```python
sage.calculus.desolvers.eulers_method(f, x0, y0, h, x1, algorithm='table')
```

This implements Euler’s method for finding numerically the solution of the 1st order ODE \( y' = f(x, y) \), \( y(a) = c \). The \( x \) column of the table increments from \( x_0 \) to \( x_1 \) by \( h \) (so \( (x_1 - x_0)/h \) must be an integer). In the \( y \) column, the new \( y \)-value equals the old \( y \)-value plus the corresponding entry in the last column.

**Note:** This function is for pedagogical purposes only.

**EXAMPLES:**

```python
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     0
 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]]
```

(continues on next page)
AUTHORS:

- David Joyner

sage.calculus.desolvers.eulers_method_2x2 \((f, g, t0, x0, y0, h, t1, \text{algorithm}='\text{table}')\)

This implements Euler’s method for finding numerically the solution of the 1st order system of two ODEs

\[
\begin{align*}
  x' &= f(t, x, y), \quad x(t_0) = x_0 \\
  y' &= g(t, x, y), \quad y(t_0) = y_0.
\end{align*}
\]

The \(t\) column of the table increments from \(t_0\) to \(t_1\) by \(h\) (so \(\frac{t_1 - t_0}{h}\) must be an integer). In the \(x\) column, the new \(x\)-value equals the old \(x\)-value plus the corresponding entry in the next (third) column. In the \(y\) column, the new \(y\)-value equals the old \(y\)-value plus the corresponding entry in the next (last) column.

**Note:** This function is for pedagogical purposes only.

**EXAMPLES:**

```python
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, 1/9, 0], [1, 1/27, 1/27], [4/3, 4/27, 4/27]]
```

```python
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1)
```

```python
\[
\begin{array}{ccc}
  t & x & h*f(t,x,y) \\
  0 & 0 & 0 \\
  1/3 & 1/9 & 1/27 \\
  1 & 1/27 & 1/27 \\
  4/3 & 4/27 & 4/27 \\
\end{array}
\]

(continues on next page)

2.19. Solving ordinary differential equations 301
To numerically approximate $y(1)$, where $(1+t^2)y'' + y' + y = 0$, $y(0) = 1$, $y'(0) = 0$:

\begin{verbatim}
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 -
0 1 0.00 0 -
1/4 1.0 -0.062 -0.25 -1 -
1/2 0.94 -0.11 -0.054 -0.50 -
3/4 0.63 -0.0078 -0.054 -0.31 -
1 0.63 0.020 0.020 0.079 -
\end{verbatim}

To numerically approximate $y(1)$, where $y'' + ty' + y = 0$, $y(0) = 1$, $y'(0) = 0$:

\begin{verbatim}
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 -
0 1 0.00 0 -
1/4 1.0 -0.062 -0.25 -1 -
1/2 0.94 -0.11 -0.054 -0.50 -
3/4 0.63 -0.0078 -0.054 -0.31 -
1 0.63 0.020 0.020 0.079 -
\end{verbatim}

(continues on next page)
AUTHORS:

- David Joyner

sage.calculus.desolvers.eulers_method_2x2_plot (f, g, t0, x0, y0, h, t1)

Plot solution of ODE.

This plots the solution in the rectangle with sides (xrange[0], xrange[1]) and (yrange[0], yrange[1]), and plots using Euler's method the numerical solution of the 1st order ODEs \( x' = f(t, x, y), x(a) = x_0, y' = g(t, x, y), y(a) = y_0. \)

Note: This function is for pedagogical purposes only.

EXAMPLES:

The following example plots the solution to \( \theta'' + \sin(\theta) = 0, \theta(0) = \frac{3}{4}, \theta'(0) = 0. \) Type \( P[0].show() \) to plot the solution, \( (P[0]+P[1]).show() \) to plot \( (t, \theta(t)) \) and \( (t, \theta'(t)) \):

```sage```
from sage.calculus.desolvers import eulers_method_2x2_plot
sage: f = lambda z : z[2]; g = lambda z : -sin(z[1])
sage: P = eulers_method_2x2_plot(f,g, 0.0, 0.75, 0.0, 0.1, 1.0)  # needs sage.plot
```

sage.calculus.desolvers.fricas_desolve (de, dvar, ics, ivar)

Solve an ODE using FriCAS.

EXAMPLES:

```sage```
x = var('x')
y = function('y')(x)
sage: desolve(diff(y,x) + y - 1, y, algorithm="fricas")  # optional --

\[-\frac{1}{5} \left( 2 \cos(x) \right) y(x)^2 + \frac{4 \sin(x) \cdot y(x)^2 - 5 \cdot e^{-2 \cdot x}}{y(x)^2}\]
```

sage.calculus.desolvers.fricas_desolve_system (des, dvars, ics, ivar)

Solve a system of first order ODEs using FriCAS.

EXAMPLES:

```sage```
t = var('t')
x = function('x')(t)
y = function('y')(t)
d1 = diff(x,t) + y - 1 == 0
d2 = diff(y,t) - x + 1 == 0
sage: desolve_system([d1, d2], [x, y], algorithm="fricas")  # optional --

\[x(t) = _C0 \cdot \cos(t) + \cos(t)^2 + _C1 \cdot \sin(t) + \sin(t)^2,
 y(t) = -_C1 \cdot \cos(t) + _C0 \cdot \sin(t) + 1\]
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

\[
\text{sage: desolve_system([de1, de2], [x,y], [0,1,2], algorithm="fricas") \hspace{1cm} \# optional - } \\
\text{fricas} \\
\text{x(t) == cos(t)^2 + sin(t)^2 - sin(t), y(t) == cos(t) + 1}
\]

2.20 Discrete Wavelet Transform

Wraps GSL's \texttt{gsl_wavelet_transform_forward()}, \texttt{and \textit{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.

\texttt{sage.calculus.transforms.dwt.DWT(n, wavelet_type, wavelet_k)}

This function initializes an GSL\texttt{DoubleArray} of length \(n\) which can perform a discrete wavelet transform.

INPUT:

- \(n\) – a power of \(2\)
- \(T\) – the data in the GSL\texttt{DoubleArray} must be real
- \(\text{wavelet_type}\) – the name of the type of wavelet, valid choices are:
  - 'daubechies'
  - 'daubechies_centered'
  - 'haar'
  - 'haar_centered'
  - 'bspline'
  - 'bspline_centered'

For Daubechies wavelets, \(\text{wavelet_k}\) specifies a Daubechies wavelet with \(k/2\) vanishing moments. \(k = 4, 6, \ldots, 20\) for \(k\) even are the only ones implemented.

For Haar wavelets, \(\text{wavelet_k}\) must be \(2\).

For Bspline wavelets, \(\text{wavelet_k}\) of \(103, 105, 202, 204, 206, 208, 301, 305, 307, 309\) will give biorthogonal B-spline wavelets of order \((i, j)\) where \(\text{wavelet_k}\) is \(100 \ast i + j\). 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:

\[
\text{sage: a = WaveletTransform(128,'daubechies',4)} \\
\text{sage: for i in range(1, 11):} \\
\text{......: a[1] = 1} \\
\text{......: a[128-1] = 1} \\
\text{sage: a.plot().show(ymin=0)}
\]

(continues on next page)
This example gives a simple example of wavelet compression:

```plaintext
sage: # needs sage.symbolic
sage: a = DWT(2048,'daubechies',6)
```

```plaintext
for i in range(2048): a[i]=float(sin((i*5/2048)**2))
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
sage: a.forward_transform()
```

```plaintext
for i in range(1800): a[2048-i-1] = 0
```

```plaintext
sage: a.backward_transform()
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
sage: a = WaveletTransform(128, 'haar', 2)
```

```plaintext
for i in range(1, 11): a[i] = 1; a[128-i] = 1
```

```plaintext
sage: a.plot().show(ymin=0)   #...
```

```plaintext
sage: a = WaveletTransform(128, 'bspline_centered', 103)
```

```plaintext
for i in range(1, 11): a[i] = 1; a[100+i] = 1
```

```plaintext
sage: a.forward_transform()
```

```plaintext
sage: a.plot().show(ymin=0)   #...
```

```plaintext
sage: a = WaveletTransform(128, 'daubechies', 6)
```

```plaintext
for i in range(2048): a[i]=float(sin((i*5/2048)**2))
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
sage: a.forward_transform()
```

```plaintext
for i in range(1800): a[2048-i-1] = 0
```

```plaintext
sage: a.backward_transform()
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
sage: a = WaveletTransform(128, 'haar', 2)
```

```plaintext
for i in range(1, 11): a[i] = 1; a[128-i] = 1
```

```plaintext
sage: a.plot().show(ymin=0)   #...
```

```plaintext
sage: a = WaveletTransform(128, 'bspline_centered', 103)
```

```plaintext
for i in range(1, 11): a[i] = 1; a[100+i] = 1
```

```plaintext
sage: a.forward_transform()
```

```plaintext
sage: a.plot().show(ymin=0)   #...
```

This example gives a simple example of wavelet compression:

```plaintext
sage: # needs sage.symbolic
sage: a = DWT(2048,'daubechies',6)
```

```plaintext
for i in range(2048): a[i]=float(sin((i*5/2048)**2))
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
sage: a.forward_transform()
```

```plaintext
for i in range(1800): a[2048-i-1] = 0
```

```plaintext
sage: a.backward_transform()
```

```plaintext
sage: a.plot().show()   # long time (7s on sage.math, 2011),
```

```plaintext
class sage.calculus.transforms.dwt.DiscreteWaveletTransform
```

Bases: GSLDoubleArray

Discrete wavelet transform class.

backward_transform()

forward_transform()

plot (xmin=None, xmax=None, **args)

sage.calculus.transforms.dwt.WaveletTransform (n, wavelet_type, wavelet_k)

This function initializes an GSLDoubleArray of length n which can perform a discrete wavelet transform.

INPUT:

* n – a power of 2
* T – the data in the GSLDoubleArray must be real
* wavelet_type – the name of the type of wavelet, valid choices are:
  - 'daubechies'
  - 'daubechies_centered'
  - 'haar'
  - 'haar_centered'
  - 'bspline'
  - 'bspline_centered'
For daubechies wavelets, \texttt{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, \texttt{wavelet\_k} must be 2.

For bspline wavelets, \texttt{wavelet\_k} of 103, 105, 202, 204, 206, 208, 301, 305, 307, 309 will give biorthogonal B-spline wavelets of order \((i, j)\) where \texttt{wavelet\_k} is \(100 \ast i + j\). The wavelet transform uses \(J = \log_2(n)\) levels.

**OUTPUT:**

An array of the form \((s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, ..., d_{J-1,2^J-1})\) for \(d_{j,k}\) the detail coefficients of level \(j\). The centered forms align the coefficients of the sub-bands on edges.

**EXAMPLES:**

```python
sage: a = WaveletTransform(128,'daubechies',4)
sage: for i in range(1, 11):
....: a[i] = 1
....: a[128-i] = 1
sage: a.plot().show(ymin=0) # needs sage.plot
sage: a.forward_transform()
sage: a.plot().show() # needs sage.plot
sage: a = WaveletTransform(128,'haar',2)
sage: for i in range(1, 11): a[i] = 1; a[128-i] = 1
sage: a.forward_transform()
sage: a.plot().show(ymin=0) # needs sage.plot
sage: a = WaveletTransform(128,'bspline_centered',103)
sage: for i in range(1, 11): a[i] = 1; a[100+i] = 1
sage: a.forward_transform()
sage: a.plot().show(ymin=0) # needs sage.plot
```

This example gives a simple example of wavelet compression:

```python
sage: # needs sage.symbolic
sage: a = DWT(2048,'daubechies',6)
sage: for i in range(2048): a[i]=float(sin((i*5/2048)**2))
sage: a.plot().show() # long time (7s on sage.math, 2011),
# needs sage.plot
sage: a.forward_transform()
sage: for i in range(1800): a[2048-i-1] = 0
sage: a.backward_transform()
sage: a.plot().show() # long time (7s on sage.math, 2011),
# needs sage.plot
```

`sage.calculus.transforms.dwt.is2pow(n)`
2.21 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 \( \mathbb{Q} \) or \( \mathbb{C} \), indexed by a `range(N)`, \( \mathbb{Z}/N\mathbb{Z} \), 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 \( \mathbb{Q} \) or \( \text{CyclotomicField} \)) indexed by `range(N)` or \( \mathbb{Z}/N\mathbb{Z} \)
  - 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 \( \mathbb{Q} \) or \( \text{CyclotomicField} \)) indexed by `range(N)` or \( \mathbb{Z}/N\mathbb{Z} \)
- `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)
### class `sage.calculus.transforms.dft.IndexedSequence(L, index_object)`

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 \( \mathbb{Q}(\zeta_N)[R] \).

**EXAMPLES:**

```python
sage: J = list(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 \( \mathbb{Z} \), 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:**

```python
sage: J = list(range(5))
sage: A = [ZZ(1) for i in J]
sage: B = [ZZ(1) for i in J]
sage: s = IndexedSequence(A, J)
sage: t = IndexedSequence(B, J)
sage: s.convolution(t)
[1, 2, 3, 4, 5, 4, 3, 2, 1]
```

**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 \( \mathbb{Z} \), then the convolution is

\[
c_j = \sum_{i=0}^{N-1} a_i b_{j-i}.
\]

INPUT:

- \( \text{other} \) – a sequence of elements of \( \mathbb{C}, \mathbb{R} \) or \( \mathbb{F}_q \)

OUTPUT:

The Dirichlet convolution of \( \text{self} \) and \( \text{other} \).

EXAMPLES:

```python
sage: I = list(range(5))
sage: A = [ZZ(1) for i in I]
sage: B = [ZZ(1) for i in I]
sage: s = IndexedSequence(A, I)
sage: t = IndexedSequence(B, I)
sage: s.convolution_periodic(t)
[5, 5, 5, 5, 5, 5, 5, 5, 5]
```

AUTHOR: David Joyner (2006-09)

**dct**

A discrete Cosine transform.

EXAMPLES:

```python
sage: J = list(range(5))
sage: A = [exp(-2*pi*i*I/5) for i in J]  # needs sage.symbolic
sage: s = IndexedSequence(A, J)  # needs sage.symbolic
sage: s.dct()  # needs sage.symbolic
```

**dft** (\( \chi = \text{None} \))

A discrete Fourier transform “over \( \mathbb{Q} \)” using exact \( N \)-th roots of unity.

EXAMPLES:

```python
sage: J = list(range(6))
sage: A = [ZZ(1) for i in J]
sage: s = IndexedSequence(A, J)
sage: s.dft(lambda x: x**2)  # needs sage.rings.number_field
```

(continues on next page)
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 - 1.73205080756888*I,
   -2.99999999999999 + 1.73205080756888*I,
   -9.00000000000000 + 0.0000000000048574425734999*I,
   -0.0000000000000976996261670137 - 0.
  -0.000000000000010658140364015*I]
  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 - 1.73205080756888*I,
   -2.99999999999999 + 1.73205080756888*I,
   -9.00000000000000 + 0.0000000000048574425734999*I,
   -0.0000000000000976996261670137 - 0.
  -0.000000000000010658140364015*I]
  indexed by Cyclic group of order 6 as a permutation group
sage: # needs sage.rings.number_field
sage: p = 7; J = list(range(p)); A = [kronecker_symbol(j,p) for j in J]
sage: s = IndexedSequence(A, J)
sage: Fs = s.dft()
sage: c = Fs.list()[1]; [x/c for x in Fs.list()]; s.list()
[0, 1, 1, -1, 1, -1, -1]
[0, 1, 1, -1, 1, -1, -1]

The DFT of the values of the quadratic residue symbol is itself, up to a constant factor (denoted \( c \) on the last line above).

**Todo:** Read the parent of the elements of \( S \); if \( Q \) or \( C \) leave as is; if AbelianGroup, use abelian_group_dual; if some other implemented Group (permutation, matrix), call .characters() and test if the index list is the set of conjugacy classes.

```
dict()
```

Return a python dict of `self` where the keys are elements in the indexing set.

**EXAMPLES:**
Sage: J = list(range(10))
Sage: A = [1/10 for j in J]
Sage: s = IndexedSequence(A, J)
Sage: s.dict()
{0: 1/10, 1: 1/10, 2: 1/10, 3: 1/10, 4: 1/10, 5: 1/10, 6: 1/10, 7: 1/10, 8: 1/10, 9: 1/10}

**dst ()**

A discrete Sine transform.

**EXAMPLES:**

Sage: J = list(range(5))
Sage: I = CC.0; pi = CC.pi()
Sage: A = [exp(-2*pi*i*I/5) for i in J]
Sage: s = IndexedSequence(A, J)
Sage: s.dst()           # discrete sine
Indexed sequence: [0.000000000000000, 1.11022302462516e-16 - 2.50000000000000*I, ...
indexed by [0, 1, 2, 3, 4]

**dwt (other='haar', wavelet_k=2)**

Wraps the gsl WaveletTransform.forward in dwt (written by Joshua Kantor). Assumes the length of the sample is a power of 2. Uses the GSL function gsl_wavelet_transform_forward().

**INPUT:**

- **other** – the name of the type of wavelet; valid choices are:
  - 'daubechies'
  - 'daubechies_centered'
  - 'haar' (default)
  - 'haar_centered'
  - 'bspline'
  - 'bspline_centered'

- **wavelet_k** – For daubechies wavelets, wavelet_k specifies a daubechlie wavelet with $k/2$ vanishing moments. $k = 4, 6, ..., 20$ for $k$ even are the only ones implemented.

  For Haar wavelets, wavelet_k must be 2.

  For bspline wavelets, wavelet_k equal to 103, 105, 202, 204, 206, 208, 301, 305, 307, 309 will give biorthogonal B-spline wavelets of order $(i,j)$ where wavelet_k equals $100 \cdot i + j$.

  The wavelet transform uses $J = \log_2(n)$ levels.

**EXAMPLES:**

Sage: J = list(range(8))
Sage: A = [RR(1) for i in J]
Sage: s = IndexedSequence(A, J)
Sage: t = s.dwt()           # slightly random output
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.
→0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
   (continues on next page)
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:

```python
sage: J = list(range(5))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.fft(); t
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000]
```

idft()
A discrete inverse Fourier transform. Only works over \( \mathbb{Q} \).

EXAMPLES:

```python
sage: J = list(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]
```

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"
Symbolic Calculus, Release 10.3

- "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:

```python
sage: J = list(range(8))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A, J)
sage: t = s.dwt() # random arch dependent output
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000] indexed by [0, 1, 2, 3, 4, 5, 6, 7]
sage: t.idwt() # random arch dependent output
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000] indexed by [0, 1, 2, 3, 4, 5, 6, 7]
sage: t.idwt() == s
True
sage: J = list(range(16))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A, J)
sage: t = s.dwt("bspline", 103) # random arch dependent output
Indexed sequence: [4.00000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000] indexed by [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: t.idwt("bspline", 103) == s
True
```

\textbf{ifft ()}:

Implements the \texttt{gslFastFourierTransform.inverse} in \texttt{fft}.

If the number of sample points in the input is a power of 2 then the wrapper for the GSL function \texttt{gsl_fft_complex_radix2_inverse()} is automatically called. Otherwise, \texttt{gsl_fft_complex_inverse()} is used.

EXAMPLES:

```python
sage: J = list(range(5))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A, J)
sage: t = s.fft(); t
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000] indexed by [0, 1, 2, 3, 4]
sage: t.ifft()
```

(continues on next page)
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000]
  indexed by [0, 1, 2, 3, 4]
sage: t.ifft() == s
1

index_object()
Return the indexing object.

EXAMPLES:

sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A, J)
sage: s.index_object()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list()
Return the list of self.

EXAMPLES:

sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A, J)
sage: s.list()
[1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10]

plot()
Plot the points of the sequence.

Elements of the sequence are assumed to be real or from a finite field, with a real indexing set I = range(len(self)).

EXAMPLES:

sage: I = list(range(3))
sage: A = [ZZ(i^2)+1 for i in I]
sage: s = IndexedSequence(A, I)
sage: P = s.plot()  # needs sage.plot
sage: show(P)  # not tested  # needs sage.plot

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 \mathbb{R}.

Options are clr, which is an RGB value, and eps, which is the spacing between the bars.

EXAMPLES:

sage: J = list(range(3))
sage: A = [ZZ(i^2)+1 for i in J]
sage: s = IndexedSequence(A, J)
sage: P = s.plot_histogram()  # needs sage.plot
2.22 Fast Fourier Transforms Using GSL

AUTHORS:

• William Stein (2006-9): initial file (radix2)
• D. Joyner (2006-10): Minor modifications (from radix2 to general case and some documentation).
• M. Hansen (2013-3): Fix radix2 backwards transformation
• L.F. Tabera Alonso (2013-3): Documentation

`sage.calculus.transforms.fft.FFT(size, base_ring=None)`

Create an array for fast Fourier transform conversion using gsl.

INPUT:

• `size` – The size of the array
• `base_ring` – Unused (2013-03)

EXAMPLES:

We create an array of the desired size:

```
sage: a = FastFourierTransform(8)
sage: a
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0)]
```

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, -1.0), (-65536, -9.65)]
```

And backwards:

```
sage: a.backward_transform()
sage: a  # abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.0, 0.0), (64.0, 0.0)]
```

Other example:
```
sage: a = FastFourierTransform(128)
sage: for i in range(1, 11):
    ....:     a[i] = 1
    ....:     a[128-i] = 1
sage: a[:6:2]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
```

```
sage: a.plot().show(ymin=0)
# needs sage.plot
sage: a.forward_transform()
```

```
sage: a.plot().show()
# needs sage.plot
```

```
sage.calculus.transforms.fft.FastFourierTransform(size, base_ring=None)
Create an array for fast Fourier transform conversion using gsl.

INPUT:

- `size` – The size of the array
- `base_ring` – Unused (2013-03)

EXAMPLES:

We create an array of the desired size:
```
sage: a = FastFourierTransform(8)
sage: a
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0)]
```

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: abs tol 1e-2
[(36.0, 0.0), (-4.00, 9.65), (-4.0, 4.0), (-4.0, 1.65), (-4.0, 0.0), (-4.0, -1.65), (-4.0, -4.0), (-4.0, -9.65)]
```

And backwards:
```
sage: a.backward_transform()
```

```
sage: abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.0, 0.0), (64.0, 0.0)]
```

Other example:
```
sage: a = FastFourierTransform(128)
sage: for i in range(1, 11):
    ....:     a[i] = 1
    ....:     a[128-i] = 1
sage: a[:6:2]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
```
class `sage.calculus.transforms.fft.FastFourierTransform_base`

Bases: `object`

class `sage.calculus.transforms.fft.FastFourierTransform_complex`

Bases: `FastFourierTransform_base`

Wrapper class for GSL’s fast Fourier transform.

`backward_transform()`

Compute the in-place backwards Fourier transform of this data using the Cooley-Tukey algorithm.

OUTPUT:

- None, the transformation is done in-place.

This is the same as `inverse_transform()` but lacks normalization so that `f.forward_transform().backward_transform() == n*f`. Where `n` is the size of the array.

EXAMPLES:

```python
sage: a = FastFourierTransform(125)
sage: b = FastFourierTransform(125)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.backward_transform()
sage: (a.plot() + b.plot()).show(ymin=0)  # long time (2s on sage.math, ˓→2011), needs sage.plot
sage: abs(sum([CDF(a[i])/125-CDF(b[i]) for i in range(125)]) < 2**-16
True
```

Here we check it with a power of two:

```python
sage: a = FastFourierTransform(128)
sage: b = FastFourierTransform(128)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.backward_transform()
sage: (a.plot() + b.plot()).show(ymin=0)  # needs sage.plot
```

`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.

2.22. Fast Fourier Transforms Using GSL
EXAMPLES:

```python
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 \( f.f[\text{forward_transform}].\text{inverse_transform}() == f \) modulo round-off errors. See also `backward_transform`.

**EXAMPLES:**

```python
sage: a = FastFourierTransform(125)
sage: b = FastFourierTransform(125)
sage: for i in range(1, 60): a[i] = 1
sage: for i in range(1, 60): b[i] = 1
sage: a.forward_transform()
sage: a.inverse_transform()
sage: a.plot() + b.plot()  # needs sage.plot
```

Here we check it with a power of two:

```python
sage: a = FastFourierTransform(128)
sage: b = FastFourierTransform(128)
sage: for i in range(1, 60): a[i] = 1
sage: for i in range(1, 60): b[i] = 1
sage: a.forward_transform()
sage: a.inverse_transform()
sage: a.plot() + b.plot()  # needs sage.plot
```

`plot(style='rect', xmin=None, xmax=None, **args)`

Plot a slice of the array.

- **style** – Style of the plot, options are "rect" or "polar"
  - rect – height represents real part, color represents imaginary part.
  - polar – height represents absolute value, color represents argument.
- **xmin** – The lower bound of the slice to plot. 0 by default.
- **xmax** – The upper bound of the slice to plot. len(self) by default.
- **args** – Passed on to the line plotting function.
OUTPUT:

- A plot of the array.

EXAMPLES:

```
sage: a = FastFourierTransform(16)
sage: for i in range(16): a[i] = (random(),random())
sage: A = plot(a)  # needs sage.plot
sage: B = plot(a, style='polar')  # needs sage.plot
sage: type(A)   # needs sage.plot
<class 'sage.plot.graphics.Graphics'>
sage: type(B)   # needs sage.plot
<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()  # needs sage.plot
```

Graphics object consisting of 250 graphics primitives

```
class sage.calculus.transforms.fft.FourierTransform_complex
    Bases: object

class sage.calculus.transforms.fft.FourierTransform_real
    Bases: object
```

**2.23 Solving ODE numerically by GSL**

AUTHORS:

- Joshua Kantor (2004-2006)
- Robert Marik (2010 - fixed docstrings)

```
class sage.calculus.ode.PyFunctionWrapper
    Bases: object

class sage.calculus.ode.ode_solver (function=None, jacobian=None, h=0.01, error_abs=1e-10, error_rel=1e-10, a=False, a_dydt=False, scale_abs=False, algorithm='rkf45', y_0=None, t_span=None, params=[])

    Bases: object

    ode_solver() is a class that wraps the GSL library's ode solver routines.
```

To use it, instantiate the class:

```
sage: T = ode_solver()
```

To solve a system of the form \( \frac{dy_i}{dt} = f_i(t, y) \), you must supply a vector or tuple/list valued function \( f \) representing \( f_i \). The functions \( f \) and the jacobian should have the form \( \text{foo}(t, y) \) or \( \text{foo}(t, y, \text{params}) \). params
which is optional allows for your function to depend on one or a tuple of parameters. Note if you use it, `params` must be a tuple even if it only has one component. For example if you wanted to solve \( y'' + y = 0 \), you would need to write it as a first order system:

\[
\begin{align*}
y_0' &= y_1 \\
y_1' &= -y_0
\end{align*}
\]

In code:

```python
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 returning a list of lists of the form: \([ \frac{df_1}{dy_1}, \ldots, \frac{df_1}{dy_n}, \ldots, \frac{df_n}{dy_1}, \ldots, \frac{df_n}{dy_n}, \frac{df_n}{dt}, \ldots, \frac{df_n}{dt} ] \). There are examples below, if your jacobian was the function `my_jacobian` you would do:

```python
sage: T.jacobian = my_jacobian  # not tested, since it doesn't make sense to...
```

There are a variety of algorithms available for different types of systems. Possible algorithms are:

- `'rkf45'` – Runge-Kutta-Fehlberg (4,5)
- `'rk2'` – embedded Runge-Kutta (2,3)
- `'rk4'` – 4th order classical Runge-Kutta
- `'rk8pd'` – Runge-Kutta Prince-Dormand (8,9)
- `'rk2imp'` – implicit 2nd order Runge-Kutta at gaussian points
- `'rk4imp'` – implicit 4th order Runge-Kutta at gaussian points
- `'bsimp'` – implicit Burlisch-Stoer (requires jacobian)
- `'gear1'` – M=1 implicit gear
- `'gear2'` – M=2 implicit gear

The default algorithm is `'rkf45'`. If you instead wanted to use `'bsimp'` you would do:

```python
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:

```python
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 \(\text{num\_points}\) is specified and \(\text{t\_span}\) is not length 2 then \(\text{t\_span}\) are used as the time points and \(\text{num\_points}\) is ignored.)

Error is estimated via the expression \(D_i = \text{error\_abs}\cdot s_i + \text{error\_rel}\cdot (a|y_i|+a\cdot \text{dydt}\cdot h\cdot |y_i'|).\)

The user can specify

- \(\text{error\_abs}\) (1e-10 by default),
- \(\text{error\_rel}\) (1e-10 by default),
- \(a\) (1 by default),
- \(a\cdot \text{dydt}\) (0 by default) and
- \(s_i\) (as \(\text{scaling\_abs}\) which should be a tuple and is 1 in all components by default).

If you specify one of \(a\) or \(a\cdot \text{dydt}\) you must specify the other. You may specify \(a\) and \(a\cdot \text{dydt}\) without \(\text{scaling\_abs}\) (which will be taken as 1 be default). \(h\) is the initial step size, which is 1e-2 by default.

\texttt{ode\_solve} solves the solution as a list of tuples of the form, \([ (t_0,[y_1,\ldots,y_n]), (t_1,[y_1,\ldots,y_n]),\ldots,(t_n,[y_1,\ldots,y_n]) ]\).

This data is stored in the variable \texttt{solutions}:

\begin{verbatim}
sage: T.solution  # not tested
\end{verbatim}

**EXAMPLES:**

Consider solving the Van der Pol oscillator \(x''(t) + u x'(t)(x(t)^2 - 1) + x(t) = 0\) between \(t = 0\) and \(t = 100\). As a first order system it is \(x' = y, y' = -x + uy(1-x^2)\). Let us take \(u = 10\) and use initial conditions \((x, y) = (1, 0)\) and use the Runge-Kutta Prince-Dormand algorithm.

\begin{verbatim}
sage: def f_1(t, y, params):
....:     return [y[1], -y[0] - params[0]*y[1]*(y[0]**2-1.0)]
sage: def j_1(t, y, params):
....:     return [[0.0, 1.0],
....:             [-2.0*params[0]*y[0]*y[1] - 1.0, -params[0]*(y[0]*y[0]-1.0)],
....:             [0.0, 0.0]]
sage: T = ode_solver()
sage: T.algorithm = "rk8pd"
sage: T.function = f_1
sage: T.jacobian = j_1
sage: T.ode_solve(y_0=[1,0], t_span=[0,100], params=[10.0], num_points=1000)
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix=".png") as f:  # needs sage.plot
....:     T.plot_solution(filename=f.name)
\end{verbatim}

The solver line is equivalent to:

\begin{verbatim}
sage: T.ode_solve(y_0=[1,0], t_span=\[x/10.0 \text{ for } x \text{ in range(1000)\}], params=[10.0])
\end{verbatim}

Let’s try a system:

\begin{verbatim}
y_0'=y_1*y_2
y_1'=-y_0*y_2
y_2'=-.51*y_0*y_1
\end{verbatim}
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: with tempfile.NamedTemporaryFile(suffix=".png") as f:
    # needs sage.plot
    ....: T.plot_solution(i=0, filename=f.name)
    ....: T.plot_solution(i=1, filename=f.name)
    ....: T.plot_solution(i=2, filename=f.name)
```

The method `interpolate_solution` will return a spline interpolation through the points found by the solver. By default, $y_0$ is interpolated. You can interpolate $y_i$ through the keyword argument `i`.

```
sage: f = T.interpolate_solution()
sage: plot(f,0,12).show()  # needs sage.plot
sage: f = T.interpolate_solution(i=1)
sage: plot(f,0,12).show()  # needs sage.plot
sage: f = T.interpolate_solution(i=2)
sage: plot(f,0,12).show()  # needs sage.plot
sage: f = T.interpolate_solution()
sage: from math import pi
sage: f(pi)
```

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(pi)
```

Unfortunately because Python functions are used, this solver is slow on systems that require many function evaluations. It is possible to pass a compiled function by deriving from the class `ode_system` and overloading `c_f` and `c_j` with C functions that specify the system. The following will work in the notebook:

```
%cython
cimport sage.calculus.ode
cimport sage.libs.gsl.all
cfrom sage.libs.gsl.all import *

cdef class van_der_pol(sage.calculus.ode.ode_system):
    cdef int c_f(self, double t, double *y, double *dydt):
        dydt[0]=y[1]
        dydt[1]=-y[0]-1000*y[1]*y[0]*y[0]-1
        return GSL_SUCCESS

cdef int c_j(self, double t, double *y, double *dfdy, double *dfdt):
```

(continues on next page)
Symbolic Calculus, Release 10.3

After executing the above block of code you can do the following (WARNING: the following is not automatically doctested):

```python
sage: # not tested
sage: T = ode_solver()

sage: T.algorithm = "bsimp"

sage: vander = van_der_pol()

sage: T.function = vander

sage: T.ode_solve(y_0=[1, 0], t_span=[0, 2000],
               num_points=1000)

sage: from tempfile import NamedTemporaryFile

sage: with NamedTemporaryFile(suffix=".png") as f:
...
T.plot_solution(i=0, filename=f.name)
```

interpolate_solution (i=0)

ode_solve (t_span=False, y_0=False, num_points=False, params=[])  

plot_solution (i=0, filename=None, interpolate=False, **kwds)  

Plot a one dimensional projection of the solution.

INPUT:

• i – (non-negative integer) component of the projection

• filename – (string or None) whether to plot the picture or save it in a file

• interpolate – whether to interpolate between the points of the discretized solution

• additional keywords are passed to the graphics primitive

EXAMPLES:

```python
sage: T = ode_solver()

sage: T.function = lambda t,y: [cos(y[0]) * sin(t)]

sage: T.jacobian = lambda t,y: [[-sin(y[0]) * sin(t)]]

sage: T.ode_solve(y_0=[1], t_span=[0, 20], num_points=1000)

sage: T.plot_solution()

# needs sage.plot
```

And with some options:

```python
sage: T.plot_solution(color='red', axes_labels=\"t, x(t)\")

# needs sage.plot
```

class sage.calculus.ode.ode_system

Bases: object
2.24 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): github issue #12047: return 0 when the integration interval is a point; reformat documentation and add to the reference manual.

class sage.calculus.integration.PyFunctionWrapper
    Bases: object

class sage.calculus.integration.compiled_integrand
    Bases: object

sage.calculus.integration.monte_carlo_integral(func, xl, xu, calls, algorithm='plain', params=None)

Integrate \( f \) by Monte-Carlo method.

Integrate \( f \) over the \( \dim \)-dimensional hypercubic region defined by the lower and upper limits in the arrays \( xl \) and \( xu \), each of size \( \dim \).

The integration uses a fixed number of function calls and obtains random sampling points using the default gsl's random number generator.

ALGORITHM: Uses calls to the GSL (GNU Scientific Library) C library. Documentation can be found in [GSL] chapter "Monte Carlo Integration".

INPUT:

- \( f \) – the function to integrate
- \( \text{params} \) – used to pass parameters to your function
- \( xl \) – list of lower limits
- \( xu \) – list of upper limits
- \( \text{calls} \) – number of functions calls used
- \( \text{algorithm} \) – valid choices are:
  - ‘plain’ – The plain Monte Carlo algorithm samples points randomly from the integration region to estimate the integral and its error.
  - ‘miser’ – The MISER algorithm of Press and Farrar is based on recursive stratified sampling
  - ‘vegas’ – The VEGAS algorithm of Lepage is based on importance sampling.

OUTPUT:

A tuple whose first component is the approximated integral and whose second component is an error estimate.

EXAMPLES:

```
sage: x, y = SR.var('x,y')
sage: monte_carlo_integral(x*y, [0,0], [2,2], 10000) # abs tol 0.1
(4.0, 0.0)
sage: integral(integral(x*y, (x,0,2)), (y,0,2))
4
```
An example with a parameter:

```
sage: x, y, z = SR.var('x,y,z')
sage: monte_carlo_integral(x*y*z, [0,0], [2,2], 10000, params=[1.2])  # abs tol 0.1
(4.8, 0.0)
```

Integral of a constant:

```
sage: monte_carlo_integral(3, [0,0], [2,2], 10000)  # abs tol 0.1
(12, 0.0)
```

Test different algorithms:

```
sage: x, y, z = SR.var('x,y,z')
sage: f(x,y,z) = exp(z) * cos(x + sin(y))
sage: for algo in ['plain', 'miser', 'vegas']:  # abs tol 0.01
....:    monte_carlo_integral(f, [0,0,-1], [2,2,1], 10^6, algorithm=algo)
(-1.06, 0.01)
(-1.06, 0.01)
(-1.06, 0.01)
```

Tests with Python functions:

```
sage: def f(u, v): return u * v
sage: monte_carlo_integral(f, [0,0], [2,2], 10000)  # abs tol 0.1
(4.0, 0.0)
sage: monte_carlo_integral(lambda u,v: u*v, [0,0], [2,2], 10000)  # abs tol 0.1
(4.0, 0.0)
sage: def f(x1,x2,x3,x4): return x1*x2*x3*x4
sage: monte_carlo_integral(f, [0,0], [2,2], 1000, params=[0.6,2])  # abs tol 0.2
(4.8, 0.0)
```

**AUTHORS:**

- Vincent Delecroix
- Vincent Klein

```
sage.calculus.integration.numerical_integral(func, a=None, b=None, algorithm='qag',
  max_points=87, params=[], eps_abs=1e-06, eps_rel=1e-06, rule=6)
```

Return the numerical integral of the function on the interval from \( a \) to \( b \) and an error bound.

**INPUT:**

- \( a, b \) – The interval of integration, specified as two numbers or as a tuple/list with the first element the lower bound and the second element the upper bound. Use \(+\infty\) and \(-\infty\) for plus or minus infinity.
- **algorithm** – valid choices are:
  - ‘qag’ – for an adaptive integration
  - ‘qags’ – for an adaptive integration with (integrable) singularities
  - ‘qng’ – for a non-adaptive Gauss-Kronrod (samples at a maximum of 87pts)
- **max_points** – sets the maximum number of sample points
- **params** – used to pass parameters to your function

2.24. Numerical Integration 325
Symbolic Calculus, Release 10.3

- **eps_abs, eps_rel** - sets the absolute and relative error tolerances which satisfy the relation $|\text{RESULT} - I| \leq \max(\text{eps}_\text{abs}, \text{eps}_\text{rel} \times |I|)$, where $I = \int_a^b f(x) \, dx$.

- **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 output (architecture and os dependent)
[(0.49999999999998657, 5.551115123125636e-15), ...
```

(continues on next page)
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:

\[
\text{sage: } f = \exp(-x) \\
\text{sage: } \text{numerical_integral}(f, 0, +\text{Infinity}) \quad \# \text{ random output} \\
(0.99999999999957279, 1.8429811298996553e-07)
\]

Note the coercion to the real field RR, which prevents underflow:

\[
\text{sage: } f = \exp(-x**2) \\
\text{sage: } \text{numerical_integral}(f, -\text{Infinity}, +\text{Infinity}) \quad \# \text{ random output} \\n(1.7724538509060035, 3.4295192165889879e-08)
\]

One can integrate any real-valued callable function:

\[
\text{sage: } \text{numerical_integral}(\lambda x: \text{abs(zeta(x))), [1.1,1.5]} \quad \# \text{ random output} \\
(1.8488570602160455, 2.052643677492633e-14)
\]

We can also numerically integrate symbolic expressions using either this function (which uses GSL) or the native integration (which uses Maxima):

\[
\text{sage: } \exp(-1/x).\text{nintegral}(x, 1, 2) \quad \# \text{via maxima} \\
(0.50479221787318..., 5.60431942934407...e-15, 21, 0) \\
\text{sage: } \text{numerical_integral}(\exp(-1/x), 1, 2) \\
(0.50479221787318..., 5.60431942934407...e-15)
\]

We can also integrate constant expressions:

\[
\text{sage: } \text{numerical_integral}(2, 1, 7) \\
(12.0, 0.0)
\]

If the interval of integration is a point, then the result is always zero (this makes sense within the Lebesgue theory of integration), see github issue #12047:

\[
\text{sage: } \text{numerical_integral}(\log, 0, 0) \\
(0.0, 0.0) \\
\text{sage: } \text{numerical_integral}(\lambda x: \sqrt{x}, (-2.0, -2.0)) \\
(0.0, 0.0)
\]

2.24. Numerical Integration 327
In the presence of integrable singularity, the default adaptive method might fail and it is advised to use 'qags':

```
sage: b = 1.81759643554688
sage: F(x) = sqrt((-x + b)/((x - 1.0)*x))
sage: numerical_integral(F, 1, b)  
( inf, nan )
sage: numerical_integral(F, 1, b, algorithm='qags')  # abs tol 1e-10
(1.1817104238446596, 3.387268288079781e-07)
```

AUTHORS:
- Josh Kantor
- William Stein
- Robert Bradshaw
- Jeroen Demeyer

ALGORITHM: Uses calls to the GSL (GNU Scientific Library) C library. Documentation can be found in [GSL] chapter “Numerical Integration”.

### 2.25 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.

```python
class sage.calculus.riemann.Riemann_Map
    Bases: object

    The Riemann_Map class computes an interior or exterior Riemann map, or an Ahlfors map of a region given by the supplied boundary curve(s) and center point. The class also provides various methods to evaluate, visualize, or extract data from the map.

    A Riemann map conformally maps a simply connected region in the complex plane to the unit disc. The Ahlfors map does the same thing for multiply connected regions.

    Note that all the methods are numerical. As a result all answers have some imprecision. Moreover, maps computed with small number of collocation points, or for unusually shaped regions, may be very inaccurate. Error computations for the ellipse can be found in the documentation for `analytic_boundary()` and `analytic_interior()`.

    [BSV2010] provides an overview of the Riemann map and discusses the research that lead to the creation of this module.

INPUT:
- `fs` – 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: 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: 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 = ", format(m.inverse_riemann_map(m.riemann_map(x)) - x))  
error = (-0.000...+0.0016...j)
```
A fun, complex region for demonstration purposes:

```plaintext
sage: f(t) = e^(I*t)
sage: fp(t) = I*e^(I*t)
sage: ef1(t) = .2*e^(-I*t) +.4+.4*I
sage: ef1p(t) = -I*.2*e^(-I*t)
sage: ef2(t) = .2*e^(-I*t) -.4+.4*I
sage: ef2p(t) = -I*.2*e^(-I*t)
sage: pts = [(-.5,-.15-20/1000),(-.6,-.27-10/1000),(-.45,-.45),(0,-.65+10/1000),(.45,-.45),(.6,-.27-10/1000),(0,-.43+10/1000)]
sage: pts.reverse()
sage: cs = complex_cubic_spline(pts)
sage: mf = lambda x:cs.value(x)
sage: mprime = lambda x: cs.derivative(x)
sage: m = Riemann_Map([f,ef1,ef2,mf],[fp,ef1p,ef2p,mprime],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 [KT1986].

```plaintext
compute_on_grid(plot_range, x_points)
```

Compute the Riemann map on a grid of points.

Note that these points are complex of the form z = x + y*i.

INPUT:

- plot_range – 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:

```plaintext
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,ef1,ef2,mf],[fp,ef1p,ef2p,mprime],0,N = 400) # long time
sage: data = m.compute_on_grid([],5)
sage: data[0][8,1]
(-0.0879...+0.9709...j)
```

```plaintext
get_szego(boundary=-1, absolute_value=False)
```

Return a discretized version of the Szego kernel for each boundary function.

INPUT:

The following inputs may be passed in as named parameters:

- boundary – integer (default: -1) if < 0, get_theta_points() will return the points for all boundaries. If >=0, get_theta_points() will return only the points for the boundary specified.
- absolute_value – boolean (default: False) if True, will return the absolute value of the (complex valued) Szego kernel instead of the kernel itself. Useful for plotting.
OUTPUT:

A list of points of the form \([t \text{ value, value of the Szego kernel at that } t]\).

EXAMPLES:

Generic use:

\[
\begin{align*}
\text{sage: } f(t) &= e^{(I^t)} - 0.5e^{(-I^t)} \\
\text{sage: } fprime(t) &= I^t e^{(I^t)} + 0.5I^t e^{(-I^t)} \\
\text{sage: } m &= \text{Riemann_Map}([f], [fprime], 0) \\
\text{sage: } sz &= m.get_szego(boundary=0) \\
\text{sage: } points &= m.get_szego(absolute_value=True) \\
\text{sage: } \text{list_plot}(points)
\end{align*}
\]

Extending the points by a spline:

\[
\begin{align*}
\text{sage: } s &= \text{spline}(points) \\
\text{sage: } s(3^pi/4) \\
\text{sage: } \text{plot}(s,0,2^pi) & \text{ # plot the kernel}
\end{align*}
\]

The unit circle with a small hole:

\[
\begin{align*}
\text{sage: } f(t) &= e^{(I^t)} \\
\text{sage: } fprime(t) &= I^t e^{(I^t)} \\
\text{sage: } hf(t) &= 0.5e^{(-I^t)} \\
\text{sage: } hfprime(t) &= 0.5^-I^t e^{(-I^t)} \\
\text{sage: } m &= \text{Riemann_Map}([f, hf], [fprime, hfprime], 0.5 + 0.5^I)
\end{align*}
\]

Getting the szego for a specific boundary:

\[
\begin{align*}
\text{sage: } sz0 &= m.get_szego(boundary=0) \\
\text{sage: } sz1 &= m.get_szego(boundary=1)
\end{align*}
\]

\textbf{get_theta_points}(boundary=-1)

Return an array of points of the form \([t \text{ value, theta in } e^{(I^t*theta)}],\) 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 \(e^{(I^t*2)}\).

For multiply connected domains, \text{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:

- \textbf{boundary} – integer (default: -1) if < 0, \text{get_theta_points}() will return the points for all boundaries. If >= 0, \text{get_theta_points}() will return only the points for the boundary specified.

OUTPUT:

A list of points of the form \([t \text{ value, theta in } e^{(I^t*theta)}]\.\)

EXAMPLES:

Getting the list of points, extending it via a spline, getting the points for only the outside of a multiply connected domain:
Symbolic Calculus, Release 10.3

```
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 specific boundary:
```
sage: tp0 = m.get_theta_points(boundary=0)
sage: tp1 = m.get_theta_points(boundary=1)

```

inverse_riemann_map (pt)

Return the inverse Riemann mapping of a point.

That is, given pt on the interior of the unit disc, inverse_riemann_map() will return the point on the original region that would be Riemann mapped to pt. Note that this method does not work for multiply connected domains.

**INPUT:**

- pt – A complex number (usually with absolute value <= 1) representing the point to be inverse mapped.

**OUTPUT:**

The point on the region that Riemann maps to the input point.

**EXAMPLES:**

Can work for different types of complex numbers:
```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: m.inverse_riemann_map(0.5 + sqrt(-0.5))
(0.406880...+0.3614702...j)
sage: m.inverse_riemann_map(0.95)
(0.486319...-4.90019052...j)
sage: m.inverse_riemann_map(0.25 - 0.3*I)
(0.1653244...-0.180936...j)
sage: m.inverse_riemann_map(complex(-0.2, 0.5))
(-0.156280...+0.321819...j)
```

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)
sage: m.plot_boundaries()
```

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()
```

Plot zoomed in on a specific spot:

```
sage: m.plot_colored(plot_range=[0,1,.25,.75])
```

High resolution plot:
To generate the unit circle map, it’s helpful to see what the colors correspond to:

\[
\text{sage: } f(t) = e^{i\, t} \\
\text{sage: } fprime(t) = i\, e^{i\, t} \\
\text{sage: } m = \text{Riemann\_Map}([f], [fprime], 0, 1000) \\
\text{sage: } m.\text{plot\_colored}() \\
\]

Graphics object consisting of 1 graphics primitive

```
sage: m.plot_colored(plot_points=1000)  # long time (29s on sage.math, 2012)
Graphics object consisting of 1 graphics primitive
```

```
plot_spiderweb(spokes=16, circles=4, pts=32, linescale=0.99, rgbcolor=[0, 0, 0], thickness=1, plotjoined=True, withcolor=False, plot_points=200, min_mag=0.001, interpolation='catrom', **options)
```

Generate a traditional “spiderweb plot” of the Riemann map.

This shows what concentric circles and radial lines map to. The radial lines may exhibit erratic behavior near the boundary; if this occurs, decreasing linescale may mitigate the problem.

For multiply connected domains the spiderweb is by necessity generated using the forward mapping. This method is more computationally intensive. In addition, these spiderwebs cannot be added to color plots. Instead the withcolor option must be used.

In addition, spiderweb plots are not currently supported for exterior maps.

**INPUT:**

The following inputs may be passed in as named parameters:

- **spokes** – integer (default: 16) the number of equally spaced radial lines to plot.
- **circles** – integer (default: 4) the number of equally spaced circles about the center to plot.
- **pts** – integer (default: 32) the number of points to plot. Each radial line is made by \(1\times \text{pts}\) points, each circle has \(2\times \text{pts}\) 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:
$\text{sage: } f(t) = e^{(I*t)} - 0.5*e^{(-I*t)}$

$\text{sage: } f\prime(t) = I*e^{(I*t)} + 0.5*I*e^{(-I*t)}$

$\text{sage: } m = \text{Riemann\_Map([f], [f\prime], 0)}$

Default plot:

$\text{sage: } m\text{.plot\_spiderweb()}$

Graphics object consisting of 21 graphics primitives

Simplified plot with many discrete points:

$\text{sage: } m\text{.plot\_spiderweb(spokes=4, circles=1, pts=400, linescale=0.95, plotjoined=False)}$

Graphics object consisting of 6 graphics primitives

Plot with thick, red lines:

$\text{sage: } m\text{.plot\_spiderweb(rgbcolor=[1,0,0], thickness=3)}$

Graphics object consisting of 21 graphics primitives

To generate the unit circle map, it’s helpful to see what the original spiderweb looks like:

$\text{sage: } f(t) = e^{(I*t)}$

$\text{sage: } f\prime(t) = I*e^{(I*t)}$

$\text{sage: } m = \text{Riemann\_Map([f], [f\prime], 0, 1000)}$

$\text{sage: } m\text{.plot\_spiderweb()}$

Graphics object consisting of 21 graphics primitives

A multiply connected region with corners. We set min_mag higher to remove “fuzz” outside the domain:

$\text{sage: } ps = \text{polygon\_spline([(-4,-2),(4,-2),(4,2),(-4,2)])}$

$\text{sage: } z1 = \text{lambda } t: \text{ps.value}(t); z1p = \text{lambda } t: \text{ps.derivative}(t)$

$\text{sage: } z2(t) = -2+\text{exp(-I*t}); z2p(t) = -I*\text{exp(-I*t)}$

$\text{sage: } z3(t) = 2+\text{exp(-I*t)}; z3p(t) = -I*\text{exp(-I*t)}$

$\text{sage: } m = \text{Riemann\_Map([z1,z2,z3],[z1p,z2p,z3p],0,ncorners=4)} \# \text{ long time}$

$\text{sage: } p = m\text{.plot\_spiderweb(withcolor=True, plot_points=500, thickness = 2.0, ... min_mag=0.1)} \# \text{ long time}$

\textbf{riemann\_map}(pr)

Return the Riemann mapping of a point.

That is, given pt on the interior of the mapped region, \text{riemann\_map} will return the point on the unit disk that pt maps to. Note that this method only works for interior points; accuracy breaks down very close to the boundary. To get boundary correspondence, use \text{get\_theta\_points()}.

\textbf{INPUT:}

- pt – A complex number representing the point to be inverse mapped.

\textbf{OUTPUT:}

A complex number representing the point on the unit circle that the input point maps to.

\textbf{EXAMPLES:}

Can work for different types of complex numbers:

$\text{sage: } f(t) = e^{(I*t)} - 0.5*e^{(-I*t)}$

$\text{sage: } f\prime(t) = I*e^{(I*t)} + 0.5*I*e^{(-I*t)}$

(continues on next page)
\[ \text{sage: } m = \text{Riemann\_Map}([f], [fprime], 0) \]
\[ \text{sage: } m.\text{riemann\_map}(0.25 + \sqrt{-0.5}) \]
\[ (0.137514...+0.876696...j) \]
\[ \text{sage: } I = \text{CDF}\text{.gen()} \]
\[ \text{sage: } m.\text{riemann\_map}(1.3*I) \]
\[ (-1.56...e-05+0.989694...j) \]
\[ \text{sage: } m.\text{riemann\_map}(0.4) \]
\[ (0.73324...+3.2...e-06j) \]
\[ \text{sage: } m.\text{riemann\_map}(\text{complex}(-3, 0.0001)) \]
\[ (1.405757...e-05+8.06...e-10j) \]

\textbf{sage.calculus.riemann.analytic\_boundary}(t, n, epsilon)

Provides an exact (for \( n = \infty \)) Riemann boundary correspondence for the ellipse with axes \(1 + \epsilon\) and \(1 - \epsilon\). The boundary is therefore given by \(e^{i t} + \epsilon e^{-i t}\). It is primarily useful for testing the accuracy of the numerical \textit{Riemann\_Map}.

\textbf{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)

\textbf{OUTPUT:}

A \( \theta \) value from 0 to 2*\( \pi \), corresponding to the point on the circle \(e^{i \theta}\)

\textbf{sage.calculus.riemann.analytic\_interior}(z, n, epsilon)

Provides a nearly exact computation of the Riemann Map of an interior point of the ellipse with axes \(1 + \epsilon\) and \(1 - \epsilon\). It is primarily useful for testing the accuracy of the numerical Riemann Map.

\textbf{INPUT:}

- \( z \) – complex - the point to be mapped.
- \( n \) – integer - the number of terms to include. 10 is fairly accurate, 20 is very accurate.

\textbf{sage.calculus.riemann.cauchy\_kernel}(t, args)

Intermediate function for the integration in \textit{analytic\_interior}().

\textbf{INPUT:}

- \( t \) – The boundary parameter, meant to be integrated over
- \( \text{args} \) – 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.
  - \( \text{part} \) – will return the real (‘r’), imaginary (‘i’) or complex (‘c’) value of the kernel

\textbf{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 \textit{plot\_colored()} only.

\textbf{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:

```python
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. ]])
```

sage.calculus.riemann.complex_to_spiderweb($z_{\text{values}}$, $dr$, $d\theta$, 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:**

- $z_{\text{values}}$ – A grid of complex numbers, as a list of lists.
- $dr$ – grid of floats, the r derivative of $z_{\text{values}}$. Used to determine precision.
- $d\theta$ – grid of floats, the theta derivative of $z_{\text{values}}$. 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 $N \times M \times 3$ floating point Numpy array $X$, where $X[i,j]$ is an (r,g,b) tuple.

**EXAMPLES:**

```python
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. ],
        [ 0. ,  0. ,  0. ],
        [ 1. ,  1. ,  1. ]],
       [[ 1. ,  1. ,  1. ],
        [ 0. ,  0. ,  0. ],
        [ 1. ,  1. ,  1. ]]]])
```
sage: complex_to_spiderweb(zval, deriv, deriv, 4, 4, [0,0,0], 1, True, 0.001)
array([[1. , 1. , 1. ],
       [1. , 1. , 1. ],
       [1. , 1. , 1. ]])

sage: 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 comparisons in plot_spiderweb and not for applications that require great precision.

INPUT:

• z_values – The values for a complex function evaluated on a grid in the complex plane, usually from
  compute_on_grid.

• xstep – float, the spacing of the grid points in the real direction

OUTPUT:

• A tuple of arrays, [dr, dtheta], with each array 2 less in both dimensions than z_values
  – dr - the abs of the derivative of the function in the +r direction
  – dtheta - the rate of accumulation of angle in the +theta direction

EXAMPLES:

Standard usage with compute_on_grid:

sage: from sage.calculus.riemann import get_derivatives
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: data = m.compute_on_grid([],19)
sage: xstep = (data[2]-data[1])/19
sage: ystep = (data[4]-data[3])/19
sage: dr, dtheta = get_derivatives(data[0],xstep,ystep)
sage: dr[8,8]
0.241...
sage: dtheta[5,5]
5.907...
2.26 Real Interpolation using GSL

class sage.calculus.interpolation.Spline

Bases: object

Create a spline interpolation object.

Given a list \(v\) of pairs, \(s = \text{spline}(v)\) 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:

```python
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:

```python
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:

```python
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:

```python
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 \(\text{len}(S)\), then points \((0, 0)\) will be inserted between the interpolation points and the point to be added:

```python
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:

```python
sage: v = [(i + RDF(i).sin()/2, i + RDF(i^2).cos()) for i in range(10)]
sage: s = spline(v)
sage: show(point(v) + plot(s,0,9, hue=.8))
# needs sage.plot
```

We compute the area underneath the spline:

```python
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)
-0.1623008526180...

sage: s.derivative(6)
0.2099798628571...

sage: s.derivative(5, order=2)
-3.0874707456138...

sage: s.derivative(6, order=2)
2.6187684827485...
```

Only the first two derivatives are supported:

```
sage: s.derivative(4, order=3)
Traceback (most recent call last):
...
ValueError: Order of derivative must be 1 or 2.
```

```
append(xy)

EXAM PLES:
```

```
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 (github issue #13519):

```
sage: S = spline([(1,1), (2,3), (4,5)]); S
[(1, 1), (2, 3), (4, 5)]

sage: S(3)
4.25

sage: S.append((5, 5)); S
[(1, 1), (2, 3), (4, 5), (5, 5)]

sage: S(3)
4.375
```

```
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.

EXAM PLES:

We draw a cubic spline through three points and compute the area underneath the curve:
```
symbolic calculus, release 10.3

\texttt{sage: s = spline([(0, 0), (1, 3), (2, 0)])}
\texttt{sage: s.definite_integral(0, 2)}
3.75
\texttt{sage: s.definite_integral(0, 1)}
1.875
\texttt{sage: s.definite_integral(0, 1) + s.definite_integral(1, 2)}
3.75
\texttt{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:

\begin{itemize}
  \item \(x\) – value at which to evaluate the derivative.
  \item \texttt{order} (default: 1) – order of the derivative. Must be 1 or 2.
\end{itemize}

EXAMPLES:

We draw a cubic spline through three points and compute the derivatives:

\texttt{sage: s = spline([(0, 0), (2, 3), (4, 0)])}
\texttt{sage: s.derivative(0)}
2.25
\texttt{sage: s.derivative(2)}
0.0
\texttt{sage: s.derivative(4)}
-2.25
\texttt{sage: s.derivative(1, order=2)}
-1.125
\texttt{sage: s.derivative(3, order=2)}
-1.125

list() 

Underlying list of points that this spline goes through.

EXAMPLES:

\texttt{sage: S = spline([(1,1), (2,3), (4,5)]); S.list()} 
\texttt{[(1, 1), (2, 3), (4, 5)]}

This is a copy of the list, not a reference (github issue #13530):

\texttt{sage: S = spline([(1,1), (2,3), (4,5)])}
\texttt{L = S.list(); L}
\texttt{[(1, 1), (2, 3), (4, 5)]}
\texttt{L[2] = (3, 2)}
\texttt{L}
\texttt{[(1, 1), (2, 3), (3, 2)]}
\texttt{S.list()} 
\texttt{[(1, 1), (2, 3), (4, 5)]}

\texttt{sage.calculus.interpolation.spline}

alias of \texttt{Spline}
2.27 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:

```python
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)

Return the derivative (speed and direction of the curve) of a given point from the parameter t.

INPUT:

• t – double, the parameter value for the parameterized curve, between 0 and 2*π.

OUTPUT:

A complex number representing the derivative at the point on the figure corresponding to the input t.

EXAMPLES:

```python
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: cs = complex_cubic_spline(pts)
sage: cs.derivative(3 / 5)
(1.40578892327...-0.225417136326...j)
sage: from math import pi
sage: cs.derivative(0) - cs.derivative(2 * pi)
0j
sage: cs.derivative(-6)
(2.52047692949...-1.89392588310...j)
```

value(t)

Return the location of a given point from the parameter t.

INPUT:

• t – double, the parameter value for the parameterized curve, between 0 and 2*π.

OUTPUT:

A complex number representing the point on the figure corresponding to the input t.

EXAMPLES:
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:

```python
definition
    # Return the derivative (speed and direction of the curve) of a given point from the parameter t.
    # INPUT:
    #   • t – double, the parameter value for the parameterized curve, between 0 and 2*pi.
    # OUTPUT:
    #   A complex number representing the derivative at the point on the polygon corresponding to the input t.
    # EXAMPLES:

    sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
    sage: ps = polygon_spline(pts)
    sage: ps.derivative(0)
    (-1-1j)
    sage: ps.derivative(1 / 3)
    (1.27323954473...+0j)
    sage: from math import pi
    sage: ps.derivative(0) - ps.derivative(2*pi)
    0j
    sage: ps.derivative(10)
    (-1.27323954473...+0j)
```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: ps = polygon_spline(pts)
sage: ps.value(0) - ps.value(2*RDF.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: from math import pi
sage: show(parametric_plot((fx, fy), (0, 2*pi)))

# needs sage.plot

sage: m = Riemann_Map([lambda x: cs.value(real(x))],
                  ....: [lambda x: cs.derivative(real(x))], 0)
sage: show(m.plot_colored() + m.plot_spiderweb())

# needs sage.plot

Polygon approximation of a circle:

sage: from cmath import exp
sage: pts = [exp(1j * t / 25) for t in range(25)]
sage: cs = complex_cubic_spline(pts)
sage: cs.derivative(2)
(-0.0497765406583...+0.151095006434...j)

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)))

# needs sage.plot

(continues on next page)
2.28 Calculus functions

sage.calculus.functions.jacobian(functions, variables)

Return the Jacobian matrix, which is the matrix of partial derivatives in which the i,j entry of the Jacobian matrix
is the partial derivative \text{diff}(\text{functions}[i], \text{variables}[j]).

EXAMPLES:

```
sage: x, y = var('x, y')
sage: g = x^2 - 2*x*y
sage: jacobian(g, (x, y))
[2*x - 2*y  -2*x]
```

The Jacobian of the Jacobian should give us the “second derivative”, which is the Hessian matrix:

```
sage: jacobian(jacobian(g, (x,y)), (x,y))
[ 2  -2]
[-2  0]
sage: g.hessian()
[ 2  -2]
[-2  0]
```

```
sage: f = (x^3*sin(y), cos(x)*sin(y), exp(x))
sage: jacobian(f, (x,y))
[ 3*x^2*sin(y)  x^3*cos(y)]
[-sin(x)*sin(y)  cos(x)*cos(y)]
[ e^x 0]
sage: jacobian(f, (y,x))
[ x^3*cos(y)  3*x^2*sin(y)]
[ cos(x)*cos(y) -sin(x)*sin(y)]
[ 0 e^x]
```

sage.calculus.functions.wronskian(*args)

Return the Wronskian of the provided functions, differentiating with respect to the given variable.

If no variable is provided, \text{diff}(f) is called for each function \(f\).

\text{wronskian}(f_1, \ldots, f_n, x) returns the Wronskian of \(f_1, \ldots, f_n\), with derivatives taken with respect to \(x\).

\text{wronskian}(f_1, \ldots, f_n) returns the Wronskian of \(f_1, \ldots, f_n\) where \(k\)'th derivatives are computed by doing \text{derivative}(k) on each function.

The Wronskian of a list of functions is a determinant of derivatives. The nth row (starting from 0) is a list of the
nth derivatives of the given functions.
For two functions:

\[
W(f, g) = \det \begin{vmatrix} f & g \\ f' & g' \end{vmatrix} = f'g - g'f.
\]

**EXAMPLES:**

```
sage: wronskian(e^x, x^2)
-x^2*e^x + 2*x*e^x
sage: x, y = var('x, y')
sage: wronskian(x*y, log(x), x)
-y*log(x) + y
```

If your functions are in a list, you can use `*` to turn them into arguments to `func`:

```
sage: wronskian(*[x^k for k in range(1, 5)])
12*x^4
```

If you want to use 'x' as one of the functions in the Wronskian, you can't put it last or it will be interpreted as the variable with respect to which we differentiate. There are several ways to get around this.

Two-by-two Wronskian of \(\sin(x)\) and \(e^x\):

```
sage: wronskian(sin(x), e^x, x)
-cos(x)*e^x + e^x*sin(x)
```

Or don’t put \(x\) last:

```
sage: wronskian(x, sin(x), e^x)
(cos(x)*e^x + e^x*sin(x))*x - 2*e^x*sin(x)
```

Example where one of the functions is constant:

```
sage: wronskian(1, e^(-x), e^(2*x))
-6*e^x
```

**REFERENCES:**

- Wikipedia article Wronskian
- http://planetmath.org/encyclopedia/WronskianDeterminant.html

**AUTHORS:**

- Dan Drake (2008-03-12)

### 2.29 Symbolic variables

```python
sage.calculus.var.clear_vars()
```

Delete all 1-letter symbolic variables that are predefined at startup of Sage.

Any one-letter global variables that are not symbolic variables are not cleared.

**EXAMPLES:**
Symbolic Calculus, Release 10.3

```
sage: var('x y z')
(x, y, z)
sage: (x+y)^z
(x + y)^z
sage: k = 15
sage: clear_vars()
sage: (x+y)^z
Traceback (most recent call last):
  ... NameError: name 'x' is not defined
sage: expand((e + i)^2)
e^2 + 2*I*e - 1
sage: k
15
```

```
sage.calculus.var.function(s, **kwds)

Create a formal symbolic function with the name s.

INPUT:

- nargs=0 - number of arguments the function accepts, defaults to variable number of arguments, or 0
- latex_name - name used when printing in latex mode
- conversions - 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.

Note: The new function is both returned and automatically injected into the global namespace. If you use this function in library code, it is better to use sage.symbolic.function_factory.function, since it will not touch the global namespace.

2.29. Symbolic variables 347
EXAMPLES:

We create a formal function called supersin

```
sage: function('supersin')
supersin
```

We can immediately use supersin in symbolic expressions:

```
sage: y, z, A = var('y z A')
sage: supersin(y+z) + A^3
A^3 + supersin(y + z)
```

We can define other functions in terms of supersin:

```
sage: g(x,y) = supersin(x)^2 + sin(y/2)
sage: g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
sage: g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
sage: k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients:

```
sage: r, kappa = var('r,kappa')
sage: psi = function(psi, nargs=1)(r); psi
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

Custom typesetting of symbolic functions in LaTeX, either using `latex_name` keyword:

```
sage: function('riemann', latex_name='mathcal{R}')
riemann
sage: latex(riemann(x))
\mathcal{R}\left(x\right)
```

or passing a custom callable function that returns a latex expression:

```
sage: mu,nu = var('mu,nu')
sage: def my_latex_print(self, *args): return "\psi_{\%s}"%(', '.join(map(latex,␣˓→args))
sage: function('psi', print_latex_func=my_latex_print)
psi
sage: latex(psi(mu,nu))
\psi_{\mu, \nu}
```

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:
Symbolic Calculus, Release 10.3

```python
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)
```

```python
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)
```

```python
sage: foo(x).n()
6
```

```python
sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x
```

```python
sage: def deriv(self, *args,**kwds): print("{} {}".format(args, kwds)); return...
˓→args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2
```

```python
sage: def pow(self, x, power_param=None): print("{} {}".format(x, power_param));
˓→return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)^y
```

```python
sage: from pprint import pformat
sage: def expand(self, *args, **kwds):
˓→print("{} {}".format(args, pformat(kwds)))
˓→return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1
```

```python
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
```

```python
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)
```

(continues on next page)
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
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:**

```python
sage: def print_args(self, *args, **kwds): print("args: ", format(args)); print("kwds: ", format(kwds)); return args[0]
sage: foo = function('t', nargs=2, tderivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

Since Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

```python
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
  ...TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class 'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

You now need to evaluate the function in order to do the arithmetic:

```python
sage: 2*f(x)
2*f(x)
```

Since Sage 4.0, you need to use `substitute_function()` to replace all occurrences of a function with another:

```python
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b)
sage: g
b*diff(cr(a), a)
sage: g.substitute_function(cr, cos)
-b*sin(a)
sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

```
sage.calculus.var.var(*args, **kwds)
Create a symbolic variable with the name s.
```
INPUT:

- **args** – A single string `var('x y')`, a list of strings `var(['x', 'y'])`, or multiple strings `var('x', 'y')`. A single string can be either a single variable name, or a space or comma separated list of variable names. In a list or tuple of strings, each entry is one variable. If multiple arguments are specified, each argument is taken to be one variable. Spaces before or after variable names are ignored.

- **kwds** – keyword arguments can be given to specify domain and custom latex_name for variables. See EXAMPLES for usage.

**Note:** The new variable is both returned and automatically injected into the global namespace. If you need a symbolic variable in library code, you must use either `SR.var()` or `SR.symbol()`.

OUTPUT:

If a single symbolic variable was created, the variable itself. Otherwise, a tuple of symbolic variables. The variable names are checked to be valid Python identifiers and a `ValueError` is raised otherwise.

**EXAMPLES:**

Here are the different ways to define three variables `x`, `y`, and `z` in a single line:

```
*sage:* var('x y z')
(x, y, z)
sage: var('x, y, z')
(x, y, z)
sage: var(['x', 'y', 'z'])
(x, y, z)
sage: var('x', 'y', 'z')
(x, y, z)
sage: var('x'), var('y'), var(z)
(x, y, z)
```

We define some symbolic variables:

```
sage: var('n xx yy zz')
(n, xx, yy, zz)
```

Then we make an algebraic expression out of them:

```
sage: f = xx^n + yy^n + zz^n; f
xx^n + yy^n + zz^n
```

By default, var returns a complex variable. To define real or positive variables we can specify the domain as:

```
sage: x = var('x', domain=RR); x; x.conjugate()
x
sage: y = var('y', domain='real'); y; y.conjugate()
y
sage: y = var('y', domain='positive'); y.abs()
y
```

Custom latex expression can be assigned to variable:

```
sage: x = var('sui', latex_name="s_(u,i)"); x._latex_()
'\textit{s}_{(u,i)}'
```
In notebook, we can also colorize latex expression:

```
sage: x = var('s_{u,i}', latex_name='\color{red}{s_{u,i}}'); x._latex_()
'\color{red}{s_{u,i}}'
```

We can substitute a new variable name for n:

```
sage: f(n = var('sigma'))
x*x^sigma + y*y^sigma + z*z^sigma
```

If you make an important built-in variable into a symbolic variable, you can get back the original value using restore:

```
sage: var('QQ RR')
(QQ, RR)
sage: QQ
QQ
sage: restore('QQ')
sage: QQ
Rational Field
```

We make two new variables separated by commas:

```
sage: var('theta, gamma')
(theta, gamma)
sage: theta^2 + gamma^3
gamma^3 + theta^2
```

The new variables are of type Expression, and belong to the symbolic expression ring:

```
sage: type(theta)
<class 'sage.symbolic.expression.Expression'>
sage: parent(theta)
Symbolic Ring
```

### 2.30 Access to Maxima methods

The wrapper around Sage expressions to give access to Maxima methods.

We convert the given expression to Maxima and convert the return value back to a Sage expression. Tab completion and help strings of Maxima methods also work as expected.

**EXAMPLES:**

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: u = t.maxima_methods(); u
MaximaWrapper(log(sqrt(2) + 1) + log(sqrt(2) - 1))
sage: type(u)
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
sage: u.logcontract()
```

(continues on next page)
\[
\log((\sqrt{2} + 1) * (\sqrt{2} - 1))
\]

```
sage: u.logcontract().parent()
Symbolic Ring
```

**sage()**

Return the Sage expression this wrapper corresponds to.

**EXAMPLES:**

```
sage: t = log(\sqrt{2} - 1) + log(\sqrt{2} + 1); t
\log(\sqrt{2} + 1) + \log(\sqrt{2} - 1)
sage: u = t.maxima_methods().sage()
sage: u is t
True
```

### 2.31 Operators

#### class `sage.symbolic.operators.DerivativeOperator`

Bases: `object`

Derivative operator.

Acting with this operator onto a function gives a new operator (of type `FDerivativeOperator`) representing the function differentiated with respect to one or multiple of its arguments.

This operator takes a list of indices specifying the position of the arguments to differentiate. For example, \(D[0, 0, 1]\) is an operator that differentiates a function twice with respect to its first argument and once with respect to its second argument.

**EXAMPLES:**

```
sage: x, y = var('x,y'); f = function('f')
sage: D[0](f)(x)
diff(f(x), x)
sage: D[0](f)(x, y)
diff(f(x, y), x)
sage: D[0, 1](f)(x, y)
diff(f(x, y), x, y)
sage: D[0, 1](f)(x, x^2)
D[0, 1](f)(x, x^2)
```

#### class `DerivativeOperatorWithParameters(parameter_set)`

Bases: `object`

A function derivative operator.

**EXAMPLES:**

```
sage: x, y = var('x,y'); f = function('f')
sage: D[0](f)(x)
diff(f(x), x)
sage: D[0](f)(x, y)
diff(f(x, y), x)
sage: D[0, 1](f)(x, y)
diff(f(x, y), x, y)
sage: D[0, 1](f)(x, x^2)
D[0, 1](f)(x, x^2)
```

#### class `FDerivativeOperator` *(function, parameter_set)*

Bases: `object`

Function derivative operators.

A function derivative operator represents a partial derivative of a function with respect to some variables.

The underlying data are the function, and the parameter set, a list recording the indices of the variables with respect to which the partial derivative is taken.
change_function\texttt{(new)}

Return a new function derivative operator with the same parameter set but for a new function.

EXAMPLES:

\begin{verbatim}
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
\end{verbatim}

\begin{verbatim}
sage: op = FDerivativeOperator(f, [0, 1])
\end{verbatim}

\begin{verbatim}
sage: op.change_function(bar)
D[0, 1](bar)
\end{verbatim}

\texttt{function()}

Return the function associated to this function derivative operator.

EXAMPLES:

\begin{verbatim}
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
\end{verbatim}

\begin{verbatim}
sage: op = FDerivativeOperator(f, [0, 1])
\end{verbatim}

\begin{verbatim}
sage: op.function()
foo
\end{verbatim}

\texttt{parameter_set()}

Return the parameter set of this function derivative operator. This is the list of indices of variables with respect to which the derivative is taken.

EXAMPLES:

\begin{verbatim}
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
\end{verbatim}

\begin{verbatim}
sage: op = FDerivativeOperator(f, [0, 1])
\end{verbatim}

\begin{verbatim}
sage: op.parameter_set()
[0, 1]
\end{verbatim}

\texttt{sage.symbolic.operators.add\_vararg(first, *rest)}

Return the sum of all the arguments.

INPUT:

• \texttt{first, *rest} – arguments to add

OUTPUT: sum of the arguments

EXAMPLES:

\begin{verbatim}
sage: from sage.symbolic.operators import add_vararg
sage: add_vararg(1, 2, 3, 4, 5, 6, 7)
28
sage: x = SR.var('x')
\end{verbatim}

\begin{verbatim}
sage: s = 1 + x + x^2  # symbolic sum
\end{verbatim}

\begin{verbatim}
sage: bool(s.operator()(*s.operands()) == s)
True
\end{verbatim}

\texttt{sage.symbolic.operators.mul\_vararg(first, *rest)}

Return the product of all the arguments.

INPUT:

• \texttt{first, *rest} – arguments to multiply
OUTPUT: product of the arguments

EXAMPLES:

```
sage: from sage.symbolic.operators import mul_vararg
sage: mul_vararg(9, 8, 7, 6, 5, 4)
60480
sage: x = SR.var('x')
sage: p = x * cos(x) * sin(x) # symbolic product
sage: bool(p.operator()(*p.operands()) == p)
```

2.32 Benchmarks

Tests that will take a long time if something is wrong, but be very quick otherwise. See https://wiki.sagemath.org/symbench. The parameters chosen below are such that with pynac most of these take well less than a second, but would not even be feasible using Sage’s Maxima-based symbolics.

Problem R1

Important note. Below we do s.expand().real() because s.real() takes forever (TODO?).

```
sage: f(z) = sqrt(1/3)*z^2 + i/3
sage: s = f(f(f(f(f(f(f(f(f(f(i/2))))))))))
sage: s.expand().real()

.. mathematica

-\rightarrow 153234901998443180742424736790714109348334942474663857718035703708589611127743908517981666567969026
-\rightarrow 160959987592246947739944859375773744043416001841910423046466880402863187009126824419781711398533250
```

Problem R2:

```
sage: def hermite(n,y):
....:     if n == 1:
....:         return 2*y
....:     if n == 0:
....:         return 1
....:     return expand(2*y*hermite(n-1,y) - 2*(n-1)*hermite(n-2,y))
sage: hermite(15, var('y'))
```

Problem R3:

```
sage: f = sum(var('x,y,z')); a = [bool(f==f) for _ in range(100000)]
```

Problem R4:

```
sage: u = [e,pi,sqrt(2)]; Tuples(u,3).cardinality()
27
```

Problem R5:

```
sage: def blowup(L,n):
....:     for i in [0..n]:
....:         L.append( (L[i] + L[i+1]) * L[i+2] )
sage: L = list(var('x,y,z'))
sage: blowup(L,15)
```

(continues on next page)
Problem R6:

```
sage: sum(((x+sin(i))/x+(x-sin(i))/x) for i in range(100)).expand()
sage: len(set(L))
```

```
19
```

Problem R7:

```
sage: f = x^24+34*x^12+45*x^3+9*x^18 +34*x^10+ 32*x^21
sage: a = [f(x=random()) for _ in range(10^4)]
```

Problem R10:

```
sage: v = [float(z) for z in [-pi,-pi+1/100..,pi]]
```

Problem R11:

```
sage: a = [random() + random()*I for w in [0..100]]
sage: a.sort()
```

Problem W3:

```
sage: acos(cos(x))
```

```
arccos(cos(x))
```

PROBLEM S1:

```
sage: _ = var('x,y,z')
sage: f = (x+y+z+1)^10
sage: g = expand(f*(f+1))
```

PROBLEM S2:

```
sage: _ = var('x,y')
sage: a = expand((x^sin(x) + y^cos(y) - z^(x+y))^100)
```

PROBLEM S3:

```
sage: _ = var('x,y,z')
sage: f = expand((x^y + y^z + z^x)^50)
sage: g = f.diff(x)
```

PROBLEM S4:

```
sage: w = (sin(x)^cos(x)).series(x,400)
```
2.33 Randomized tests of GiNaC / PyNaC

sage.symbolic.random_tests.assert_strict_weak_order \((a, b, c, cmp\_func)\)

Check that \(cmp\_func\) is a strict weak order on the elements \(a, b, c\).

A strict weak order is a binary relation \(<\) such that

- For all \(x\), it is not the case that \(x < x\) (irreflexivity).
- For all \(x \neq y\), if \(x < y\) then it is not the case that \(y < x\) (asymmetry).
- For all \(x, y,\) and \(z\), if \(x < y\) and \(y < z\) then \(x < z\) (transitivity).
- For all \(x, y,\) and \(z\), if \(x\) is incomparable with \(y\), and \(y\) is incomparable with \(z\), then \(x\) is incomparable with \(z\) (transitivity of incomparability).

INPUT:

- \(a, b, c\) – anything that can be compared by \(cmp\_func\).
- \(cmp\_func\) – function of two arguments that returns their comparison (i.e. either \(True\) or \(False\)).

OUTPUT:

Does not return anything. Raises a \(ValueError\) if \(cmp\_func\) is not a strict weak order on the three given elements.

REFERENCES:

Wikipedia article Strict weak ordering

EXAMPLES:

The usual ordering of integers is a strict weak order:

```python
sage: from sage.symbolic.random_tests import assert_strict_weak_order
sage: a, b, c = [randint(-10, 10) for i in range(3)]
sage: assert_strict_weak_order(a, b, c, lambda x, y: x < y)
sage: x = [-SR(oo), SR(0), SR(oo)]
sage: cmp_M = matrix(3, 3, 0)
sage: for i in range(3):
    ....:     for j in range(3):
    ....:         if x[i] < x[j]:
    ....:             cmp_M[i, j] = -1
    ....:         elif x[i] > x[j]:
    ....:             cmp_M[i, j] = 1
sage: cmp_M
[ 0 -1 -1]
[ 1 0 -1]
[ 1 1 0]
```

sage.symbolic.random_tests.choose_from_prob_list \((lst)\)

INPUT:

- \(lst\) - A list of tuples, where the first element of each tuple is a nonnegative float (a probability), and the probabilities sum to one.

OUTPUT:

A tuple randomly selected from the list according to the given probabilities.

EXAMPLES:
sage: from sage.symbolic.random_tests import *
sage: v = [(0.1, False), (0.9, True)]
sage: choose_from_prob_list(v)  # random
(0.900000000000000, True)
sage: true_count = 0
sage: total_count = 0
sage: def more_samples():
    ...:     global true_count, total_count
    ...:     for _ in range(10000):
    ...:         total_count += 1.0
    ...:         if choose_from_prob_list(v)[1]:
    ...:             true_count += 1.0
sage: more_samples()
sage: while abs(true_count/total_count - 0.9) > 0.01:
    ...:     more_samples()

sage.symbolic.random_tests.normalize_prob_list(pl, extra=())

INPUT:

- pl - A list of tuples, where the first element of each tuple is a floating-point number (representing a relative probability). The second element of each tuple may be a list or any other kind of object.
- extra - A tuple which is to be appended to every tuple in pl.

This function takes such a list of tuples (a “probability list”) and normalizes the probabilities so that they sum to one. If any of the values are lists, then those lists are first normalized; then the probabilities in the list are multiplied by the main probability and the sublist is merged with the main list.

For example, suppose we want to select between group A and group B with 50% probability each. Then within group A, we select A1 or A2 with 50% probability each (so the overall probability of selecting A1 is 25%); and within group B, we select B1, B2, or B3 with probabilities in a 1:2:2 ratio.

EXAMPLES:

sage: from sage.symbolic.random_tests import *
sage: A = [(0.5, 'A1'), (0.5, 'A2')]
sage: B = [(1, 'B1'), (2, 'B2'), (2, 'B3')]
sage: top = [(50, A, 'Group A'), (50, B, 'Group B')]
sage: normalize_prob_list(top)
[(0.250000000000000, 'A1', 'Group A'), (0.250000000000000, 'A2', 'Group A'), (0.1, 'B1', 'Group B'), (0.2, 'B2', 'Group B'), (0.2, 'B3', 'Group B')]
2.33. Randomized tests of GiNaC / PyNaC

sage.symbolic.random_tests.random_expr(size, nvars=1, ncoeffs=None, var_frac=0.5, internal=[], 0.6, 0.3, <built-in function add>, 0.1, <built-in function sub>, 0.3, <built-in function mul>, 0.2, <built-in function truediv>, 0.1, <built-in function pow>>, 2, 0.2, (0.8, <built-in function neg>, 0.2, <built-in function invert>], 1), 0.2, (1.0, Ei, 1), (1.0, Order, 1), (1.0, _swap_harmonic, 2), (1.0, abs, 1), (1.0, airy_ai, 1), (1.0, airy_ai_prime, 1), (1.0, airy_bi, 1), (1.0, airy_bi_prime, 1), (1.0, arccos, 1), (1.0, arccosh, 1), (1.0, arccot, 1), (1.0, arccoth, 1), (1.0, arcsine, 1), (1.0, arcsinh, 1), (1.0, arcsec, 1), (1.0, arctan, 1), (1.0, arctan2, 2), (1.0, arctanh, 1), (1.0, arg, 1), (1.0, bessel_I, 2), (1.0, bessel_J, 2), (1.0, bessel_K, 2), (1.0, bessel_Y, 2), (1.0, beta, 2), (1.0, binomial, 2), (1.0, ceil, 1), (1.0, chebyshev_T, 2), (1.0, chebyshev_U, 2), (1.0, complex_root_of, 2), (1.0, conjugate, 1), (1.0, cos, 1), (1.0, cos_integral, 1), (1.0, cosh, 1), (1.0, cosh_integral, 1), (1.0, cot, 1), (1.0, coth, 1), (1.0, csch, 1), (1.0, dickman_rho, 1), (1.0, dilog, 1), (1.0, dirac_delta, 1), (1.0, elliptic_e, 2), (1.0, elliptic_ec, 1), (1.0, elliptic_eu, 2), (1.0, elliptic_f, 2), (1.0, elliptic_kc, 1), (1.0, elliptic_pi, 3), (1.0, erf, 1), (1.0, erfc, 1), (1.0, erfi, 1), (1.0, erfinv, 1), (1.0, exp, 1), (1.0, exp_integral_e, 2), (1.0, exp_integral_e1, 1), (1.0, exp_polar, 1), (1.0, factorial, 1), (1.0, floor, 1), (1.0, frac, 1), (1.0, fresnel_cos, 1), (1.0, fresnel_sin, 1), (1.0, gamma_inc_lower, 2), (1.0, gegenbauer, 3), (1.0, gen_laguerre, 3), (1.0, gen_legendre_P, 3), (1.0, gen_legendre_Q, 3), (1.0, hahn, 5), (1.0, hankel1, 2), (1.0, hankel2, 2), (1.0, harmonic_number, 1), (1.0, heaviside, 1), (1.0, hermite, 2), (1.0, hurwitz_zeta, 2), (1.0, hypergeometric_M, 3), (1.0, hypergeometric_U, 3), (1.0, imag_part, 1), (1.0, integrate, 4), (1.0, inverse_jacobi_cd, 2), (1.0, inverse_jacobi_cn, 2), (1.0, inverse_jacobi_cs, 2), (1.0, inverse_jacobi_dc, 2), (1.0, inverse_jacobi_db, 2), (1.0, inverse_jacobi_ds, 2), (1.0, inverse_jacobi_dn, 2), (1.0, inverse_jacobi_ds, 2), (1.0, inverse_jacobi_id, 2), (1.0, inverse_jacobi_nc, 2), (1.0, inverse_jacobi_md, 2), (1.0, inverse_jacobi_ns, 2), (1.0, inverse_jacobi_sc, 2), (1.0, inverse_jacobi_sd, 2), (1.0, inverse_jacobi_sn, 2), (1.0, jacobi_P, 4), (1.0, jacobi_am, 2), (1.0, jacobi_cd, 2), (1.0, jacobi_cn, 2), (1.0, jacobi_cs, 2), (1.0, jacobi_dc, 2), (1.0, jacobi_dn, 2), (1.0, jacobi_ds, 2), (1.0, jacobi_id, 2), (1.0, jacobi_id, 2), (1.0, jacobi_nc, 2), (1.0, jacobi_md, 2), (1.0, jacobi_ns, 2), (1.0, jacobi_sc, 2), (1.0, jacobi_sd, 2), (1.0, jacobi_sn, 2), (1.0, krawtchouk, 4), (1.0, kronecker_delta, 2), (1.0, laguerre, 2), (1.0, lambert_w, 2), (1.0, legendre_P, 2), (1.0, legendre_Q, 2), (1.0, log, 2), (1.0, log_gamma, 1), (1.0, log_integral, 1), (1.0, log_integral_offset, 1), (1.0, meijer_g, 4), (1.0, polylog, 2), (1.0, prime_pi, 1), (1.0, product, 4), (1.0, real_nth_root, 2), (1.0, real_part, 1), (1.0, sec, 1), (1.0, sech, 1), (1.0, sign, 1), (1.0, sin, 1), (1.0, sinh, 1), (1.0, sinh_integral, 1), (1.0, spherical_bessel_J, 2), (1.0, spherical_bessel_Y, 2), (1.0, spherical_hankel1, 2), (1.0, spherical_hankel2, 2), (1.0, spherical_harmonic, 4), (1.0, stieltjes, 1), (1.0, struve_H, 2), (1.0, struve_L, 2), (1.0, sum, 4), (1.0, tan, 1), (1.0, tanh, 1), (1.0, unit_step, 1), (1.0, zeta, 1), (1.0, zetaderiv, 2)], nullary=[]([1.0, pi], [1.0, e], [0.05, golden_ratio], [0.05, log2], [0.05, euler_gamma], [0.05, catalan], [0.05, khinchin], [0.05, twindprime], [0.05, mertens]], nullary_frac=0.2, coeff_generator=<bound method RationalField.random_element of Rational Field>, verbose=False)
Produce a random symbolic expression of the given size. By default, the expression involves (at most) one variable, an arbitrary number of coefficients, and all of the symbolic functions and constants (from the probability lists full_internal and full_nullary). It is possible to adjust the ratio of leaves between symbolic constants, variables, and coefficients (var_frac gives the fraction of variables, and nullary_frac the fraction of symbolic constants; the remaining leaves are coefficients).

The actual mix of symbolic constants and internal nodes can be modified by specifying different probability lists.

To use a different type for coefficients, you can specify coeff_generator, which should be a function that will return a random coefficient every time it is called.

This function will often raise an error because it tries to create an erroneous expression (such as a division by zero).

**EXAMPLES:**

```python
sage: from sage.symbolic.random_tests import *
sage: some_functions = [arcsinh, arctan, arctan2, arctanh,
...: arg, beta, binomial, ceil, conjugate, cos, cosh, cot, coth,
...: elliptic_pi, erf, exp, factorial, floor, heaviside, imag_part,
...: sech, sgn, sin, sinh, tan, tanh, unit_step, zeta, zetaderiv]
sage: my_internal = [(0.6, full_binary, 2), (0.2, full_unary, 1),
...: (0.2, [(1.0,f,f.number_of_arguments()) for f in some_functions])]
sage: set_random_seed(1)
sage: random_expr(50, nvars=3, internal=my_internal,
...: coeff_generator=CDF.random_element)
(v1^(0.9713408427702117 + 0.195868299334218*I)/cot(-pi + v1^2 + v3) +
→tan(arctan(v2 + arctan2(-0.35859061674557324 + 0.9407509502498164*I, v3) - 0.
→8419115504372718 + 0.30375717982404615*I) + arctan2((0.2275357305882964 - 0.
→8258002386106038*I)/factorial(v2), -v3 - 0.7604559947718565 - 0.
→5543672548552057*I) + ceil(1/arctan2(v1, v1)))/v2
```

In small cases we will see all cases quickly:

```python
sage: def next_expr():
...: return random_expr_helper(6, [(0.5, operator.add, 2), (0.5, operator.neg, 1)],
...: [(0.5, 1), (0.5, x)], False)
```

Instead of picking a random function, we could construct a sequence of small expressions.

```python
sage: def next_expr():
...: return operator.add(random_expr(), x)
```

By making a small change to the program, we can gradually increase the size of the expressions. In the example below, we change the size of the expression.

```python
sage: def next_expr():
...: return operator.add(random_expr(), x)
```

In the example above, we can see that the size of the expression is gradually increasing.

The size of the expression is controlled by the function `random_expr_helper`. The function takes a limit value, `n_nodes`, and checks if the current size is less than this limit. If the current size is less than the limit, the function returns a random internal node. Otherwise, it returns a random leaf.

**EXAMPLES:**

```python
sage: from sage.symbolic.random_tests import *
sage: a = random_expr_helper(9, [(0.5, operator.add, 2), (0.5, operator.neg, 1)],
...: [(0.5, 1), (0.5, x)], True)
About to apply <built-in function inv> to [31]
About to apply sgn to [v1]
About to apply <built-in function add> to [1/31, sgn(v1)]
sgn(v1) + 1/31
```

The function `random_expr_helper` is a useful tool for generating random symbolic expressions of a given size.
Symbolic Calculus, Release 10.3

(continued from previous page)

```python
....:       continue
....:   all_exprs.add(a*x + b)
sage: our_exprs = set()
sage: while our_exprs != all_exprs:
....:       our_exprs.add(next_expr())
```

`sage.symbolic.random_tests.random_integer_vector(n, length)`

Give a random list of length `length`, consisting of nonnegative integers that sum to `n`.

This is an approximation to `IntegerVectors(n, length).random_element()`. That gives values uniformly at random, but might be slow; this routine is not uniform, but should always be fast.

(This routine is uniform if `length` is 1 or 2; for longer vectors, we prefer approximately balanced vectors, where all the values are around `n/length`.)

**EXAMPLES:**

```python
sage: from sage.symbolic.random_tests import *
sage: a = random_integer_vector(100, 2); a  # random
[11, 89]
sage: len(a)
2
sage: sum(a)
100

sage: b = random_integer_vector(10000, 20)
sage: len(b)
20
sage: sum(b)
10000
```

The routine is uniform if `length` is 2:

```python
sage: true_count = 0
sage: total_count = 0
sage: def more_samples():
....:     global true_count, total_count
....:     for _ in range(1000):
....:         total_count += 1.0
....:     if a == random_integer_vector(100, 2):
....:         true_count += 1.0
sage: more_samples()

sage: while abs(true_count/total_count - 0.01) > 0.01:
....:     more_samples()
```

`sage.symbolic.random_tests.test_symbolic_expression_order(repetitions=100)`

Tests whether the comparison of random symbolic expressions satisfies the strict weak order axioms.

This is important because the C++ extension class uses `std::sort()` which requires a strict weak order. See also `github issue #9880`.

**EXAMPLES:**

```python
sage: from sage.symbolic.random_tests import test_symbolic_expression_order
test_symbolic_expression_order(200)
sage: test_symbolic_expression_order(10000)  # long time
```
CHAPTER THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page
PYTHON MODULE INDEX

C
sage.calculus.calculus, 172
sage.calculus.desolvers, 286
sage.calculus.functional, 224
sage.calculus.functions, 345
sage.calculus.integration, 324
sage.calculus.interpolation, 339
sage.calculus.interpolators, 342
sage.calculus.ode, 319
sage.calculus.riemann, 328
sage.calculus.test_sympy, 246
sage.calculus.tests, 250
sage.calculus.transforms.dft, 307
sage.calculus.transforms.dwt, 304
sage.calculus.transforms.fft, 315
sage.calculus.var, 346
sage.calculus.wester, 276

S
sage.symbolic.assumptions, 146
sage.symbolic.benchmark, 355
sage.symbolic.callable, 143
sage.symbolic.complexity_measures, 275
sage.symbolic.expression, 5
sage.symbolic.expression_conversions, 253
sage.symbolic.function, 216
sage.symbolic.function_factory, 220
sage.symbolic.integration.external, 245
sage.symbolic.integration.integral, 235
sage.symbolic.maxima_wrapper, 352
sage.symbolic.operators, 353
sage.symbolic.random_tests, 357
sage.symbolic.relation, 155
sage.symbolic.ring, 204
sage.symbolic.subring, 211
sage.symbolic.units, 198
SubringFactory method), 215

csc  (sage.symbolic.expression_conversions.Exponentialize attribute), 256
csch (sage.symbolic.expression_conversions.Exponentialize attribute), 256
csgn()  (sage.symbolic.expression.Expression method), 25

D
dct()  (sage.calculus.transforms.dft.IndexedSequence method), 309
decl_assume()  (sage.symbolic.expression.Expression method), 25
decl_forget()  (sage.symbolic.expression.Expression method), 25
default_variable()  (sage.symbolic.expression.Expression method), 26
default_variable()  (sage.symbolic.expression.SymbolicSeries method), 125
default_variable()  (sage.symbolic.function.Function method), 218
definite_integral()  (sage.calculus.interpolation.Spline method), 340
DefiniteIntegral  (class in sage.symbolic.integration.integral), 235
degree()  (sage.symbolic.expression.Expression method), 26
DeMoivre  (class in sage.symbolic.expression_conversions), 254
demoivre()  (sage.symbolic.expression.Expression method), 26
denominator()  (sage.symbolic.expression.Expression method), 27
derivative()  (in module sage.calculus.functional), 224
derivative()  (sage.calculus.interpolation.Spline method), 341
derivative()  (sage.calculus.interpolators.CCSpline method), 342
derivative()  (sage.calculus.interpolators.PSpline method), 343
derivative()  (sage.symbolic.expression_conversions.Converter method), 254
derivative()  (sage.symbolic.expression_conversions.ExpressionTreeWalker method), 261
derivative()  (sage.symbolic.expression_conversions.FriCASConverter method), 264
derivative()  (sage.symbolic.expression_conversions.Interfacelnit method), 268
derivative()  (sage.symbolic.expression_conversions.SubstituteFunction method), 273
derivative()  (sage.symbolic.expression.Expression method), 27
DerivativeOperator  (class in sage.symbolic.operators), 353

DerivativeOperator.DerivativeOperator- WithParameters (class in sage.symbolic.operators), 353
desolve()  (in module sage.calculus.desolvers), 287
desolve_laplace()  (in module sage.calculus.desolvers), 291
desolve_mintides()  (in module sage.calculus.desolvers), 292
desolve_odeint()  (in module sage.calculus.desolvers), 293
desolve_rk4()  (in module sage.calculus.desolvers), 295
desolve_rk4_determine_bounds()  (in module sage.calculus.desolvers), 296
desolve_system()  (in module sage.calculus.desolvers), 297
desolve_system_rk4()  (in module sage.calculus.desolvers), 298
desolve_tides_mpfr()  (in module sage.calculus.desolvers), 299
dft()  (sage.calculus.transforms.dft.IndexedSequence method), 309
dict()  (sage.calculus.transforms.dft.IndexedSequence method), 310
diff()  (in module sage.calculus.functional), 226
diff()  (sage.symbolic.expression.Expression method), 29
differentiate()  (sage.symbolic.expression.Expression method), 31
DiscreteWaveletTransform  (class in sage.calculus.transforms.dwt), 305
distribute()  (sage.symbolic.expression.Expression method), 32
divide_both_sides()  (sage.symbolic.expression.Expression method), 33
doublefactorial()  (in module sage.symbolic.expression), 127
dst()  (sage.calculus.transforms.dft.IndexedSequence method), 311
dummy_diff()  (in module sage.calculus.calculus), 177
dummy_integrate()  (in module sage.calculus.calculus), 177
dummy_inverse_laplace()  (in module sage.calculus.calculus), 177
dummy_laplace()  (in module sage.calculus.calculus), 177
dummy_pochhammer()  (in module sage.calculus.calculus), 178
DWT()  (in module sage.calculus.transforms.dwt), 304
dwt()  (sage.calculus.transforms.dft.IndexedSequence method), 311

E
E  (class in sage.symbolic.expression), 6

Index
e (sage.symbolic.expression_conversions.Exponentialize attribute), 256
eulers_method() (in module sage.calculus.desolvers), 300
eulers_method_2x2() (in module sage.calculus.desolvers), 301
eulers_method_2x2_plot() (in module sage.calculus.desolvers), 303
evalunitdict() (in module sage.symbolic.units), 202
exp (sage.symbolic.expression_conversions.Exponentialize attribute), 256
exp() (sage.symbolic.expression.Expression method), 33
expand() (in module sage.calculus.functional), 227
expand() (sage.symbolic.expression.Expression method), 33
expand_log() (sage.symbolic.expression.Expression method), 34
expand_rational() (sage.symbolic.expression.Expression method), 35
expand_sum() (sage.symbolic.expression.Expression method), 36
expand_trig() (sage.symbolic.expression.Expression method), 37
Exponentialize (class in sage.symbolic.expression_conversions), 255
exponentialize() (sage.symbolic.expression.Expression method), 38
Expression (class in sage.symbolic.expression), 8
expression() (sage.symbolic.expression.PynacConstant method), 124
ExpressionIterator (class in sage.symbolic.expression), 123
ExpressionTreeWalker (class in sage.symbolic.expression_conversions), 260

F
de (sage.symbolic.expression_conversions.Exponentialize attribute), 256
FastFourierTransform_base (class in sage.calculus.transforms.fft), 316
FastFourierTransform_complex (class in sage.calculus.transforms.fft), 317
FDerivativeOperator (class in sage.symbolic.operators), 353
FFT() (in module sage.calculus.transforms.fft), 315
fft() (sage.calculus.transforms.dft.IndexedSequence method), 312
find() (sage.symbolic.expression.Expression method), 41
find_local_maximum() (sage.symbolic.expression.Expression method), 41
find_local_minimum() (sage.symbolic.expression.Expression method), 42
find_registered_function() (in module sage.symbolic.expression), 127
find_root() (sage.symbolic.expression.Expression method), 43
forget() (in module sage.symbolic.assumptions), 154
forget() (sage.symbolic.assumptions.GenericDeclaration method), 149
forget() (sage.symbolic.expression.Expression method), 44
forward_transform() (sage.calculus.transforms.dft.DiscreteWaveletTransform method), 305
forward_transform() (sage.calculus.transforms.fft.FastFourierTransform_complex method), 317
FourierTransform_complex (class in sage.calculus.transforms.fft), 319
FourierTransform_real (class in sage.calculus.transforms.fft), 319
fraction() (sage.symbolic.expression.Expression method), 45
free_variables() (sage.symbolic.expression.Expression method), 45
fricas_desolve() (in module sage.calculus.desolvers), 303
fricas_desolve_system() (in module sage.calculus.desolvers), 303
fricas_integrator() (in module sage.symbolic.integration.external), 245
FriCASConverter (class in sage.symbolic.expression_conversions), 264
full_simplify() (sage.symbolic.expression.Expression method), 45
Function (class in sage.symbolic.function), 218
function() (in module sage.calculus.var), 347
function() (in module sage.symbolic.function_factory), 220
function() (sage.symbolic.expression_conversions.Exponentialize method), 256
function() (sage.symbolic.expression.Expression method), 46
function() (sage.symbolic.operators.FDerivativeOper-
Symbolic Calculus, Release 10.3

G

gamma() (sage.symbolic.expression.Expression method), 47
gamma_normalize() (sage.symbolic.expression.Expression method), 47
gcd() (sage.symbolic.expression.Expression method), 48
GenericDeclaration (class in sage.symbolic.assumptions), 147
GenericSymbolicSubring (class in sage.symbolic.subring), 211
GenericSymbolicSubringFunctor (class in sage.symbolic.subring), 212
get_derivatives() (in module sage.calculus.riemann), 338
get_fake_div() (sage.symbolic.expression_conversions.Converter method), 254
get_fn_serial() (in module sage.symbolic.expression), 128
get_ginac_serial() (in module sage.symbolic.expression), 128
get_sfunction_from_hash() (in module sage.symbolic.expression), 128
get_sfunction_from_serial() (in module sage.symbolic.expression), 128
giacc integrator() (in module sage.symbolic.integration.external), 245
GiacFunction (class in sage.symbolic.function), 219
gosper_sum() (sage.symbolic.expression.Expression method), 49
gosper_term() (sage.symbolic.expression.Expression method), 49
gradient() (sage.symbolic.expression.Expression method), 50

H

half (sage.symbolic.expression_conversions.Exponentialize attribute), 260
has() (sage.symbolic.assumptions.GenericDeclaration method), 149
has() (sage.symbolic.expression.Expression method), 50
has_valid_variable() (sage.symbolic.subring.GenericSymbolicSubring method), 213
has_valid_variable() (sage.symbolic.subring.SymbolicSubring method), 215
has_valid_variable() (sage.symbolic.subring.SymbolicSubringRejectingVars method), 215
has_wild() (sage.symbolic.expression.Expression method), 51
hessian() (sage.symbolic.expression.Expression method), 51
hold_class (class in sage.symbolic.expression), 128
HoldRemover (class in sage.symbolic.expression_conversions), 266
horner() (sage.symbolic.expression.Expression method), 51
hypergeometric_simplify() (sage.symbolic.expression.Expression method), 52

I

I (sage.symbolic.expression_conversions.Exponentialize attribute), 255
I() (sage.symbolic.ring.SymbolicRing method), 204
idft() (sage.calculus.transforms.dft.IndexedSequence method), 312
idwt() (sage.calculus.transforms.dft.IndexedSequence method), 312
ifft() (sage.calculus.transforms.dft.IndexedSequence method), 313
imag() (sage.symbolic.expression.Expression method), 52
imag_part() (sage.symbolic.expression.Expression method), 53
implicit_derivative() (sage.symbolic.expression.Expression method), 54
IndefiniteIntegral (class in sage.symbolic.integration.integral), 235
index_object() (sage.calculus.transforms.dft.IndexedSequence method), 314
IndexedSequence (class in sage.calculus.transforms.dft), 308
init_function_table() (in module sage.symbolic.expression), 129
init_pynac_I() (in module sage.symbolic.expression), 129
Integer (sage.symbolic.expression_conversions.Exponentialize attribute), 255
integral() (in module sage.calculus.functional), 228
integral() (in module sage.symbolic.integration.integral), 235
integral() (sage.symbolic.expression.Expression method), 54
integrate() (in module sage.calculus.functional), 230
integrate() (in module sage.symbolic.integration.integral), 240
integrate() (sage.symbolic.expression.Expression method), 55
is_symbol() (sage.calculus.ode.ode_solver method), 327
is_square() (sage.symbolic.expression.Expression method), 59
is_relational() (sage.symbolic.expression.Expression method), 59
is_real() (sage.symbolic.expression.Expression method), 59
is_rational_expression() (sage.symbolic.expression.Expression method), 61
is_positive_infinity() (sage.symbolic.expression.Expression method), 61
is_positive() (sage.symbolic.expression.Expression method), 61
is_polynomial() (sage.symbolic.expression.Expression method), 57
is_numeric() (sage.symbolic.expression.Expression method), 57
is_negative_infinity() (sage.symbolic.expression.Expression method), 57
is_negative() (sage.symbolic.expression.Expression method), 57
is_integer() (sage.symbolic.expression.Expression method), 56
is_infinity() (sage.symbolic.expression.Expression method), 56
is_field() (sage.symbolic.expression.Expression method), 205
is_finite() (sage.symbolic.expression.Expression method), 205
is_exact() (sage.symbolic.expression.Expression method), 56
is_exact() (sage.symbolic.ring.SymbolicRing method), 205
is_algebraic() (sage.symbolic.expression.Conversion method), 251
is2pow() (sage.calculus.transforms.dft.IndexedSequence method), 57
is_algebraic() (sage.calculus.transforms.dft.IndexedSequence method), 57
is_symbol() (sage.symbolic.expression.Expression method), 61
is_symbol() (sage.symbolic.expression.Expression method), 61
is_SymbolicEquation() (module sage.symbolic.expression), 130
is_terminating_series() (sage.symbolic.expression.Expression method), 61
is_terminating_series() (sage.symbolic.expression.SymbolicSeries method), 126
is_trivial_zero() (sage.symbolic.expression.Expression method), 62
is_trivially_equal() (sage.symbolic.expression.Expression method), 62
is_unit() (module sage.symbolic.units), 202
is_unit() (sage.symbolic.expression.Expression method), 63
isidentifier() (module sage.symbolicRING), 210
isidentifier() (module sage.symbolicRING), 210
isidentifier() (module sage.symbolicRING), 210
isidentifier() (module sage.symbolicRING), 210
isidentifier() (module sage.symbolicRING), 210
isidentifier() (module sage.symbolicRING), 210
log_gamma()  (sage.symbolic.expression.Expression method), 69
log_simplify()  (sage.symbolic.expression.Expression method), 69
low_degree()  (sage.symbolic.expression.Expression method), 71

M
make_map()  (in module sage.symbolic.expression), 131
mapped_opts()  (in module sage.calculus.calculus), 187
match()  (sage.symbolic.expression.Expression method), 71
math_sorted()  (in module sage.symbolic.expression), 131
maxima_integrator()  (in module sage.symbolic.integration.external), 245
maxima_methods()  (sage.symbolic.expression.Expression method), 73
maxima_options()  (in module sage.calculus.calculus), 187
MaximaFunctionElementWrapper  (class in sage.symbolic.maxima_wrapper), 352
MaximaWrapper  (class in sage.symbolic.maxima_wrapper), 352
merge()  (sage.symbolic.callable.CallableSymbolicExpressionFunctor method), 144
merge()  (sage.symbolic.subring.GenericSymbolicSubringFunctor method), 212
merge()  (sage.symbolic.subring.SymbolicSubringAcceptingVarsFunctor method), 213
merge()  (sage.symbolic.subring.SymbolicSubringRejectingVarsFunctor method), 215
minpoly()  (in module sage.calculus.calculus), 188
minpoly()  (sage.symbolic.expression.Expression method), 73
mixed_order()  (in module sage.symbolic.expression), 131
mixed_sorted()  (in module sage.symbolic.expression), 132
mma_free_integrator()  (in module sage.symbolic.integration.external), 246
mma_free_limit()  (in module sage.calculus.calculus), 190

N
name()  (sage.symbolic.expression.PynacConstant method), 124
name()  (sage.symbolic.function.Function method), 218
negation()  (sage.symbolic.expression.Expression method), 74
new_Expression()  (in module sage.symbolic.expression), 132
new_Expression_from_pyobject()  (in module sage.symbolic.expression), 133
new_Expression_symbol()  (in module sage.symbolic.expression), 134
new_Expression_wild()  (in module sage.symbolic.expression), 134
nintegral()  (in module sage.calculus.calculus), 190
nintegral()  (sage.symbolic.expression.Expression method), 74
nintegrate()  (in module sage.calculus.calculus), 192
nintegrate() (sage.symbolic.expression.Expression method), 74
nops() (sage.symbolic.expression.Expression method), 75
norm() (sage.symbolic.expression.Expression method), 75
normalize() (sage.symbolic.expression.Expression method), 75
normalize_index_for_doctests() (in module sage.symbolic.expression), 134
normalize_prob_list() (in module sage.symbolic.random_tests), 358
number_of_arguments() (sage.symbolic.expression.Expression method), 76
number_of_arguments() (sage.symbolic.function.Function method), 218
number_of_operands() (sage.symbolic.expression.Expression method), 76
numerator() (sage.symbolic.expression.Expression method), 76
numerator_denominator() (sage.symbolic.expression.Expression method), 77
numerical_approx() (sage.symbolic.expression.Expression method), 78
numerical_integral() (in module sage.calculus.integration), 325
NumpyToSRMorphism (class in sage.symbolic.ring), 204

O
ode_solve() (sage.calculus.ode.ode_solver method), 323
ode_solver (class in sage.calculus.ode), 319
ode_system (class in sage.calculus.ode), 323
op (sage.symbolic.expression.Expression attribute), 78
operands() (sage.symbolic.expression_conversions.FakeExpression method), 262
operands() (sage.symbolic.expression.Expression method), 79
OperandsWrapper (class in sage.symbolic.expression), 123
operator() (sage.symbolic.expression_conversions.FakeExpression method), 262
operator() (sage.symbolic.expression.Expression method), 79
Order() (sage.symbolic.expression.Expression method), 8

P
parameter_set() (sage.symbolic.operators.FDerivativeOperator method), 354
paramset_from_Expression() (in module sage.symbolic.expression), 135
partial_fraction() (sage.symbolic.expression.Expression method), 80
partial_fraction_decomposition() (sage.symbolic.expression.Expression method), 80
pi() (sage.symbolic.ring.SymbolicRing method), 205
pickle_wrapper ()(in module sage.symbolic.function), 219
plot() (sage.calculus.transforms.dft.IndexedSequence method), 314
plot() (sage.calculus.transforms.dwt.DiscreteWaveletTransform method), 305
plot() (sage.calculus.transforms.fft.FastFourierTransform_complex method), 318
plot() (sage.symbolic.expression.Expression method), 81
plot_boundaries() (sage.calculus.riemann.Riemann_Map method), 332
plot_colored() (sage.calculus.riemann.Riemann_Map method), 333
plot_histogram() (sage.calculus.transforms.dft.IndexedSequence method), 314
plot_solution() (sage.calculus.ode.ode_solver method), 323
plot_spiderweb() (sage.calculus.riemann.Riemann_Map method), 334
poly() (sage.symbolic.expression.Expression method), 82
polygon_spline() (in module sage.calculus.interpolators), 344
polynomial() (in module sage.symbolic.expression_conversions), 274
polynomial() (sage.symbolic.expression.Expression method), 82
PolynomialConverter (class in sage.symbolic.expression_conversions), 269
power() (sage.symbolic.expression.Expression method), 84
power_series() (sage.symbolic.expression.Expression method), 84
power_series() (sage.symbolic.expression.SymbolicSeries method), 126
preprocess_assumptions() (in module sage.symbolic.assumptions), 154
primitive_part() (sage.symbolic.expression.Expression method), 85
print_order() (in module sage.symbolic.expression), 135
print_sorted() (in module sage.symbolic.expression), 135
prod() (sage.symbolic.expression.Expression method), 85
PSpline (class in sage.calculus.interpolators), 343
py_atan2_for_doctests() (in module sage.symbolic.expression), 135
py_denom_for_doctests() (in module sage.symbolic.expression), 135
py_eval_infinity_for_doctests() (in module sage.symbolic.expression), 136
py_eval_neg_infinity_for_doctests() (in module sage.symbolic.expression), 136
py_eval_unsigned_infini-
ity_for_doctests() (in module sage.symbolic.expression), 136
py_exp_for_doctests() (in module sage.symbolic.expression), 136
py_factorial_py() (in module sage.symbolic.expression), 136
py_float_for_doctests() (in module sage.symbolic.expression), 136
py_imag_for_doctests() (in module sage.symbolic.expression), 136
py_is_cinteger_for_doctests() (in module sage.symbolic.expression), 136
py_is_cratrional_for_doctests() (in module sage.symbolic.expression), 136
py_is_integer_for_doctests() (in module sage.symbolic.expression), 136
py_latex_fderivative_for_doctests() (in module sage.symbolic.expression), 136
py_latex_function_pystring() (in module sage.symbolic.expression), 137
py_latex_variable_for_doctests() (in module sage.symbolic.expression), 138
py_lgamma_for_doctests() (in module sage.symbolic.expression), 138
py_lli2_for_doctests() (in module sage.symbolic.expression), 138
py_lli_for_doctests() (in module sage.symbolic.expression), 139
py_log_for_doctests() (in module sage.symbolic.expression), 139
py_mod_for_doctests() (in module sage.symbolic.expression), 139
py_numer_for_doctests() (in module sage.symbolic.expression), 139
py_print_fderivative_for_doctests() (in module sage.symbolic.expression), 139
py_print_function_pystring() (in module sage.symbolic.expression), 140
py_psi2_for_doctests() (in module sage.symbolic.expression), 140
py_psi_for_doctests() (in module sage.symbolic.expression), 140
py_real_for_doctests() (in module sage.symbolic.expression), 141
py_stieltjes_for_doctests() (in module sage.symbolic.expression), 141
py_tgamma_for_doctests() (in module sage.symbolic.expression), 141
py_zeta_for_doctests() (in module sage.symbolic.expression), 141
PyFunctionWrapper (class in sage.calculus.integration), 324
PyFunctionWrapper (class in sage.calculus.ode), 319
PynacConstant (class in sage.symbolic.expression), 124
pyobject() (sage.symbolic.expression_conversions.Converter method), 254
pyobject() (sage.symbolic.expression_conversions.ExpressionTreeWalker method), 261
pyobject() (sage.symbolic.expression_conversions.FakeExpression method), 262
pyobject() (sage.symbolic.expression_conversions.FastCallableConverter method), 263
pyobject() (sage.symbolic.expression_conversions.FriCASConverter method), 265
pyobject() (sage.symbolic.expression_conversions.InterfaceInit method), 268
pyobject() (sage.symbolic.expression_conversions.PolynomialConverter method), 270
pyobject() (sage.symbolic.expression_conversions.RingConverter method), 272
pyobject() (sage.symbolic.expression.Expression method), 85
R
random_expr() (in module sage.symbolic.random_tests), 358
random_expr_helper() (in module sage.symbolic.random_tests), 360
random_integer_vector() (in module sage.symbolic.random_tests), 361
rank (sage.symbolic.subring.GenericSymbolicSubring.Factor attribute), 212
rational_expand() (sage.symbolic.expression.Expression method), 86
rational_simplify() (sage.symbolic.expression.Expression method), 87
real() (sage.symbolic.expression.Expression method), 88
real_part() (sage.symbolic.expression.Expression method), 88
rectform() (sage.symbolic.expression.Expression method), 89
reduce_trig() (sage.symbolic.expression.Expression method), 90
register_or_update_function() (in module sage.symbolic.expression), 141
relation() (sage.symbolic.expression_conversions.Converter method), 254
relation() (sage.symbolic.expression_conversions.ExpressionTreeWalker method), 261
relation() (sage.symbolic.expression_conversions.FastCallableConverter method), 263
relation() (sage.symbolic.expression_conversions.InterfaceInit method), 268
relation() (sage.symbolic.expression_conversions.PolynomialConverter method), 271
residue() (sage.symbolic.expression.Expression method), 90

Index
restore_op_wrapper() (in module sage.symbolic.expression), 141
resultant() (sage.symbolic.expression.Expression method), 91
rhs() (sage.symbolic.expression.Expression method), 91
Riemann_Map (class in sage.calculus.riemann), 328
riemann_map() (sage.calculus.riemann.Riemann_Map method), 335
right() (sage.symbolic.expression.Expression method), 92
right_hand_side() (sage.symbolic.expression.Expression method), 92
RingConverter (class in sage.symbolic.expression_conversions), 271
roots() (sage.symbolic.expression.Expression method), 92
round() (sage.symbolic.expression.Expression method), 94
S
sage() (sage.symbolic.maxima_wrapper.MaximaWrapper method), 353
sage.calculus.calculus module, 172
sage.calculus.desolvers module, 286
sage.calculus.functional module, 224
sage.calculus.functions module, 345
sage.calculus.integration module, 324
sage.calculus.interpolation module, 339
sage.calculus.interpolators module, 342
sage.calculus.ode module, 319
sage.calculus.riemann module, 328
sage.calculus.test_sympy module, 246
sage.calculus.tests module, 250
sage.calculus.transforms.dft module, 307
sage.calculus.transforms.dwt module, 304
sage.calculus.transforms.fft module, 315
sage.calculus.var module, 346
sage.calculus.wester module, 276
simplify_rational() (sage.symbolic.expression.Expression method), 100
simplify_real() (sage.symbolic.expression.Expression method), 101
simplify_rectform() (sage.symbolic.expression.Expression method), 102
simplify_trig() (sage.symbolic.expression.Expression method), 102
sin (sage.symbolic.expression_conversions.Exponentialize attribute), 260
sin() (sage.symbolic.expression.Expression method), 103
sinh (sage.symbolic.expression_conversions.Exponentialize attribute), 260
sinh() (sage.symbolic.expression.Expression method), 104
solve() (in module sage.symbolic.relation), 161
solve() (sage.symbolic.expression.Expression method), 104
solve_diophantine() (in module sage.symbolic.expression), 141
solve_diophantine() (sage.symbolic.expression.Expression method), 105
solve_ineq() (in module sage.symbolic.relation), 167
solve_ineq_fourier() (in module sage.symbolic.relation), 168
solve_ineq_univar() (in module sage.symbolic.relation), 168
solve_mod() (in module sage.symbolic.relation), 169
Spline (class in sage.calculus.interpolation), 339
spline (in module sage.calculus.interpolation), 341
sqrt() (sage.symbolic.expression.Expression method), 106
SR (sage.symbolic.expression_conversions.Exponentialize attribute), 255
start() (sage.symbolic.expression.hold_class method), 129
step() (sage.symbolic.expression.Expression method), 107
stop() (sage.symbolic.expression.hold_class method), 129
str_to_unit() (in module sage.symbolic.units), 203
string_length() (in module sage.symbolic.complexity_measures), 275
string_to_list_of_solutions() (in module sage.symbolic.relation), 170
subring() (sage.symbolic.ring.SymbolicRing method), 205
subs() (sage.symbolic.expression.Expression method), 108
substitute_function() (sage.symbolic.expression.Expression method), 110
SubstituteFunction (class in sage.symbolic.expression_conversions), 272
substitution_delayed() (sage.symbolic.expression.Expression method), 111
SubstitutionMap (class in sage.symbolic.expression), 124
subtract_from_both_sides() (sage.symbolic.expression.Expression method), 112
sum() (sage.symbolic.expression.Expression method), 112
symbol() (sage.symbolic.expression_conversions.Converter method), 254
symbol() (sage.symbolic.expression_conversions.ExpressionTreeWalker method), 261
symbol() (sage.symbolic.expression_conversions.FastCallableConverter method), 263
symbol() (sage.symbolic.expression_conversions.FricasConverter method), 266
symbol() (sage.symbolic.expression_conversions.InverseFunction method), 269
symbol() (sage.symbolic.expression_conversions.PolynomialConverter method), 271
symbol() (sage.symbolic.expression_conversionsRINGConverter method), 272
symbol() (sage.symbolic.ring.SymbolicRing method), 207
symbolic_expression_from_maxima_string() (in module sage.calculus.calculus), 194
symbolic_expression_from_string() (in module sage.calculus.calculus), 194
symbolic_product() (in module sage.calculus.calculus), 195
symbolic_sum() (in module sage.calculus.calculus), 195
SymbolicConstantsSubring (class in sage.symbolic.subring), 212
SymbolicFunction (class in sage.symbolic.function), 219
SymbolicRing (class in sage.symbolic.ring), 204
SymbolicSeries (class in sage.symbolic.expression), 125
SymbolicSubringAcceptingVars (class in sage.symbolic.subring), 213
SymbolicSubringAcceptingVarsFunctor (class in sage.symbolic.subring), 213
SymbolicSubringFactory (class in sage.symbolic.subring), 214
SymbolicSubringRejectingVars (class in sage.symbolic.subring), 215
SymbolicSubringRejectingVarsFunctor (class in sage.symbolic.subring), 215
symbols (sage.symbolic.ring.SymbolicRing attribute), 207
sympy_integrator() (in module sage.symbolic.integration.external), 246

T

tan (sage.symbolic.expression_conversions.Exponentialize attribute), 20
tan() (sage.symbolic.expression.Expression method), 114
tanh (sage.symbolic.expression_conversions.Exponentialize attribute), 260
tanh() (sage.symbolic.expression.Expression method), 115
taylor() (in module sage.calculus.functional), 234
taylor() (sage.symbolic.expression.Expression method), 116
temp_var() (sage.symbolic.ring.SymbolicRing method), 207
TemporaryVariables (class in sage.symbolic.ring), 209
test_binominal() (in module sage.symbolic.expression), 142
test_relation() (sage.symbolic.expression.Expression method), 116
test_relation_maxima() (in module sage.symbolic.relation), 170
test_symbolic_expression_order() (in module sage.symbolic.random_tests), 361
the_SymbolicRing() (in module sage.symbolic.ring), 210
to_gamma() (sage.symbolic.expression.Expression method), 117
tolerant_is_symbol() (in module sage.symbolic.expression), 142
trailing_coeff() (sage.symbolic.expression.Expression method), 117
trailing_coefficient() (sage.symbolic.expression.Expression method), 118
trig_expand() (sage.symbolic.expression.Expression method), 118
trig_reduce() (sage.symbolic.expression.Expression method), 119
trig_simplify() (sage.symbolic.expression.Expression method), 119
truncade() (sage.symbolic.expression.Expression method), 120
truncade() (sage.symbolic.expression.SymbolicSeries method), 126
tuple() (sage.symbolic.expression_conversions.ExpressionTreeWalker method), 261
tuple() (sage.symbolic.expression_conversions.FastCallableConverter method), 264
tuple() (sage.symbolic.expression_conversions.InterfaceInit method), 269
two (sage.symbolic.expression_conversions.Exponentialize attribute), 260

U
UnderscoreSageMorphism (class in sage.symbolic.ring), 209

V
value() (sage.calculus.interpolators.CCSpline method), 342
value() (sage.calculus.interpolators.PSpline method), 343
var() (in module sage.calculus.var), 350
var() (in module sage.symbolic.ring), 210
var() (sage.symbolic.ring.SymbolicRing method), 208
variables() (sage.symbolic.expression.Expression method), 122
variables() (sage.symbolic.function.Function method), 219
vars_in_str() (in module sage.symbolic.units), 204

W
WaveletTransform() (in module sage.calculus.transforms.dwt), 305
wild() (sage.symbolic.ring.SymbolicRing method), 209
wronskian() (in module sage.calculus.functions), 345
WZ_certificate() (sage.symbolic.expression.Expression method), 8

X
x (sage.symbolic.expression_conversions.Exponentialize attribute), 260

Z
zeta() (sage.symbolic.expression.Expression method), 122