Functions

Release 10.3

The Sage Development Team

Mar 20, 2024

CONTENTS

CHAPTER ONE

BUILT-IN FUNCTIONS

1.1 Logarithmic functions

AUTHORS:

- Yoora Yi Tenen (2012-11-16): Add documentation for `log()` (github issue #12113)
- Tomas Kalvoda (2015-04-01): Add `exp_polar()` (github issue #18085)

class `sage.functions.log.`**`Function_dilog`**

Bases: `GinacFunction`

The dilogarithm function $\mathrm{Li}_2(z) = \sum_{k=1}^{\infty} z^k/k^2$.

This is simply an alias for `polylog(2, z)`.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: dilog(1)
1/6*pi^2
sage: dilog(1/2)
1/12*pi^2 - 1/2*log(2)^2
sage: dilog(x^2+1)
dilog(x^2 + 1)
sage: dilog(-1)
-1/12*pi^2
sage: dilog(-1.0)
-0.822467033424113
sage: dilog(-1.1)
-0.890838090262283
sage: dilog(1/2)
1/12*pi^2 - 1/2*log(2)^2
sage: dilog(.5)
0.582240526465012
sage: dilog(1/2).n()
0.582240526465012
sage: var('z')
z
sage: dilog(z).diff(z, 2)
log(-z + 1)/z^2 - 1/((z - 1)*z)
sage: dilog(z).series(z==1/2, 3)
(1/12*pi^2 - 1/2*log(2)^2) + (-2*log(1/2))*(z - 1/2)
 + (2*log(1/2) + 2)*(z - 1/2)^2 + Order(1/8*(2*z - 1)^3)

sage: latex(dilog(z))                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
{\rm Li}_2\left(z\right)
```

Dilog has a branch point at 1. Sage's floating point libraries may handle this differently from the symbolic package:

```
sage: # needs sage.symbolic
sage: dilog(1)
1/6*pi^2
sage: dilog(1.)
1.64493406684823
sage: dilog(1).n()
1.64493406684823
sage: float(dilog(1))
1.6449340668482262
```

class sage.functions.log.**Function_exp**

Bases: GinacFunction

The exponential function, $\exp(x) = e^x$.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: exp(-1)
e^(-1)
sage: exp(2)
e^2
sage: exp(2).n(100)
7.3890560989306502272304274606
sage: exp(x^2 + log(x))
e^(x^2 + log(x))
sage: exp(x^2 + log(x)).simplify()
x*e^(x^2)
sage: exp(2.5)
12.1824939607035
sage: exp(I*pi/12)
(1/4*I + 1/4)*sqrt(6) - (1/4*I - 1/4)*sqrt(2)

sage: exp(float(2.5))
12.182493960703473
sage: exp(RDF('2.5'))
12.182493960703473
```

To prevent automatic evaluation, use the hold parameter:

```
sage: exp(I*pi, hold=True)                                              #␣
↪needs sage.symbolic
e^(I*pi)
sage: exp(0, hold=True)                                                 #␣
↪needs sage.symbolic
e^0
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: exp(0, hold=True).simplify()                                         #␣
↪needs sage.symbolic
1
```

```
sage: # needs sage.symbolic
sage: exp(pi*I/2)
I
sage: exp(pi*I)
-1
sage: exp(8*pi*I)
1
sage: exp(7*pi*I/2)
-I
```

For the sake of simplification, the argument is reduced modulo the period of the complex exponential function, $2\pi i$:

```
sage: k = var('k', domain='integer')                                       #␣
↪needs sage.symbolic
sage: exp(2*k*pi*I)                                                        #␣
↪needs sage.symbolic
1
sage: exp(log(2) + 2*k*pi*I)                                               #␣
↪needs sage.symbolic
2
```

The precision for the result is deduced from the precision of the input. Convert the input to a higher precision explicitly if a result with higher precision is desired:

```
sage: t = exp(RealField(100)(2)); t                                        #␣
↪needs sage.rings.real_mpfr
7.3890560989306502272304274606
sage: t.prec()                                                             #␣
↪needs sage.rings.real_mpfr
100
sage: exp(2).n(100)                                                        #␣
↪needs sage.symbolic
7.3890560989306502272304274606
```

class sage.functions.log.**Function_exp_polar**

Bases: BuiltinFunction

Representation of a complex number in a polar form.

INPUT:

- z – a complex number $z = a + ib$.

OUTPUT:

A complex number with modulus $\exp(a)$ and argument b.

If $-\pi < b \leq \pi$ then $\text{exp\_polar}(z) = \exp(z)$. For other values of b the function is left unevaluated.

EXAMPLES:

The following expressions are evaluated using the exponential function:

```
sage: exp_polar(pi*I/2)                                                  #␣
↪needs sage.symbolic
I
sage: x = var('x', domain='real')                                        #␣
↪needs sage.symbolic
sage: exp_polar(-1/2*I*pi + x)                                           #␣
↪needs sage.symbolic
e^(-1/2*I*pi + x)
```

The function is left unevaluated when the imaginary part of the input z does not satisfy $-\pi < \Im(z) \leq \pi$:

```
sage: exp_polar(2*pi*I)                                                  #␣
↪needs sage.symbolic
exp_polar(2*I*pi)
sage: exp_polar(-4*pi*I)                                                 #␣
↪needs sage.symbolic
exp_polar(-4*I*pi)
```

This fixes github issue #18085:

```
sage: integrate(1/sqrt(1+x^3), x, algorithm='sympy')                     #␣
↪needs sage.symbolic
1/3*x*gamma(1/3)*hypergeometric((1/3, 1/2), (4/3,), -x^3)/gamma(4/3)
```

See also:

Examples in Sympy documentation, Sympy source code of exp_polar

REFERENCES:

Wikipedia article Complex_number#Polar_form

class `sage.functions.log.`**`Function_harmonic_number`**

Bases: `BuiltinFunction`

Harmonic number function, defined by:

$$H_n = H_{n,1} = \sum_{k=1}^{n} \frac{1}{k}$$

$$H_s = \int_0^1 \frac{1-x^s}{1-x}$$

See the docstring for *`Function_harmonic_number_generalized()`*.

This class exists as callback for `harmonic_number` returned by Maxima.

class `sage.functions.log.`**`Function_harmonic_number_generalized`**

Bases: `BuiltinFunction`

Harmonic and generalized harmonic number functions, defined by:

$$H_n = H_{n,1} = \sum_{k=1}^{n} \frac{1}{k}$$

$$H_{n,m} = \sum_{k=1}^{n} \frac{1}{k^m}$$

They are also well-defined for complex argument, through:

$$H_s = \int_0^1 \frac{1-x^s}{1-x}$$
$$H_{s,m} = \zeta(m) - \zeta(m, s-1)$$

If called with a single argument, that argument is `s` and `m` is assumed to be 1 (the normal harmonic numbers H_s).

ALGORITHM:

Numerical evaluation is handled using the mpmath and FLINT libraries.

REFERENCES:

- Wikipedia article Harmonic_number

EXAMPLES:

Evaluation of integer, rational, or complex argument:

```
sage: harmonic_number(5)                                                    #␣
↪needs mpmath
137/60

sage: # needs sage.symbolic
sage: harmonic_number(3, 3)
251/216
sage: harmonic_number(5/2)
-2*log(2) + 46/15
sage: harmonic_number(3., 3)
zeta(3) - 0.0400198661225573
sage: harmonic_number(3., 3.)
1.16203703703704
sage: harmonic_number(3, 3).n(200)
1.16203703703703703703703...
sage: harmonic_number(1 + I, 5)
harmonic_number(I + 1, 5)
sage: harmonic_number(5, 1. + I)
1.57436810798989 - 1.06194728851357*I
```

Solutions to certain sums are returned in terms of harmonic numbers:

```
sage: k = var('k')                                                          #␣
↪needs sage.symbolic
sage: sum(1/k^7,k,1,x)                                                      #␣
↪needs sage.symbolic
harmonic_number(x, 7)
```

Check the defining integral at a random integer:

```
sage: n = randint(10,100)
sage: bool(SR(integrate((1-x^n)/(1-x),x,0,1)) == harmonic_number(n))       #␣
↪needs sage.symbolic
True
```

There are several special values which are automatically simplified:

```
sage: harmonic_number(0)                                                    #␣
↪needs mpmath
0
```

(continues on next page)

(continued from previous page)

```
sage: harmonic_number(1)                                          #␣
↪needs mpmath
1
sage: harmonic_number(x, 1)                                       #␣
↪needs sage.symbolic
harmonic_number(x)
```

class sage.functions.log.**Function_lambert_w**

Bases: BuiltinFunction

The integral branches of the Lambert W function $W_n(z)$.

This function satisfies the equation

$$z = W_n(z)e^{W_n(z)}$$

INPUT:

- n – an integer. $n = 0$ corresponds to the principal branch.
- z – a complex number

If called with a single argument, that argument is z and the branch n is assumed to be 0 (the principal branch).

ALGORITHM:

Numerical evaluation is handled using the mpmath and SciPy libraries.

REFERENCES:

- Wikipedia article Lambert_W_function

EXAMPLES:

Evaluation of the principal branch:

```
sage: lambert_w(1.0)                                              #␣
↪needs scipy
0.567143290409784
sage: lambert_w(-1).n()                                           #␣
↪needs mpmath
-0.318131505204764 + 1.33723570143069*I
sage: lambert_w(-1.5 + 5*I)                                       #␣
↪needs mpmath sage.symbolic
1.17418016254171 + 1.10651494102011*I
```

Evaluation of other branches:

```
sage: lambert_w(2, 1.0)                                           #␣
↪needs scipy
-2.40158510486800 + 10.7762995161151*I
```

Solutions to certain exponential equations are returned in terms of lambert_w:

```
sage: S = solve(e^(5*x)+x==0, x, to_poly_solve=True)              #␣
↪needs sage.symbolic
sage: z = S[0].rhs(); z                                           #␣
↪needs sage.symbolic
-1/5*lambert_w(5)
sage: N(z)                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
-0.265344933048440
```

Check the defining equation numerically at $z = 5$:

```
sage: N(lambert_w(5)*exp(lambert_w(5)) - 5)                              #␣
↪needs mpmath
0.000000000000000
```

There are several special values of the principal branch which are automatically simplified:

```
sage: lambert_w(0)                                                       #␣
↪needs mpmath
0
sage: lambert_w(e)                                                       #␣
↪needs sage.symbolic
1
sage: lambert_w(-1/e)                                                    #␣
↪needs sage.symbolic
-1
```

Integration (of the principal branch) is evaluated using Maxima:

```
sage: integrate(lambert_w(x), x)                                         #␣
↪needs sage.symbolic
(lambert_w(x)^2 - lambert_w(x) + 1)*x/lambert_w(x)
sage: integrate(lambert_w(x), x, 0, 1)                                   #␣
↪needs sage.symbolic
(lambert_w(1)^2 - lambert_w(1) + 1)/lambert_w(1) - 1
sage: integrate(lambert_w(x), x, 0, 1.0)                                 #␣
↪needs sage.symbolic
0.3303661247616807
```

Warning: The integral of a non-principal branch is not implemented, neither is numerical integration using GSL. The `numerical_integral()` function does work if you pass a lambda function:

```
sage: numerical_integral(lambda x: lambert_w(x), 0, 1)                   #␣
↪needs sage.modules
(0.33036612476168054, 3.667800782666048e-15)
```

class sage.functions.log.**Function_log1**

Bases: `GinacFunction`

The natural logarithm of `x`.

See `log()` for extensive documentation.

EXAMPLES:

```
sage: ln(e^2)                                                            #␣
↪needs sage.symbolic
2
sage: ln(2)                                                              #␣
↪needs sage.symbolic
log(2)
sage: ln(10)                                                             #␣
↪needs sage.symbolic
log(10)
```

class sage.functions.log.**Function_log2**

Bases: GinacFunction

Return the logarithm of x to the given base.

See log() for extensive documentation.

EXAMPLES:

```
sage: from sage.functions.log import logb
sage: logb(1000, 10)                                                        #␣
↪needs sage.symbolic
3
```

class sage.functions.log.**Function_polylog**

Bases: GinacFunction

The polylog function $\mathrm{Li}_s(z) = \sum_{k=1}^{\infty} z^k / k^s$.

The first argument is s (usually an integer called the weight) and the second argument is z: polylog(s, z).

This definition is valid for arbitrary complex numbers s and z with $|z| < 1$. It can be extended to $|z| \geq 1$ by the process of analytic continuation, with a branch cut along the positive real axis from 1 to $+\infty$. A NaN value may be returned for floating point arguments that are on the branch cut.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: polylog(2.7, 0)
0.000000000000000
sage: polylog(2, 1)
1/6*pi^2
sage: polylog(2, -1)
-1/12*pi^2
sage: polylog(3, -1)
-3/4*zeta(3)
sage: polylog(2, I)
I*catalan - 1/48*pi^2
sage: polylog(4, 1/2)
polylog(4, 1/2)
sage: polylog(4, 0.5)
0.517479061673899

sage: # needs sage.symbolic
sage: polylog(1, x)
-log(-x + 1)
sage: polylog(2, x^2 + 1)
dilog(x^2 + 1)
sage: f = polylog(4, 1); f
1/90*pi^4
sage: f.n()
1.08232323371114
sage: polylog(4, 2).n()
2.42786280675470 - 0.174371300025453*I
sage: complex(polylog(4, 2))
(2.4278628067547032-0.17437130002545306j)
sage: float(polylog(4, 0.5))
0.5174790616738993
sage: z = var('z')
sage: polylog(2, z).series(z==0, 5)
```

(continues on next page)

(continued from previous page)

```
1*z + 1/4*z^2 + 1/9*z^3 + 1/16*z^4 + Order(z^5)

sage: loads(dumps(polylog))
polylog

sage: latex(polylog(5, x))                                              #␣
↪needs sage.symbolic
{\rm Li}_{5}(x)
sage: polylog(x, x)._sympy_()                                           #␣
↪needs sympy sage.symbolic
polylog(x, x)
```

1.2 Trigonometric functions

class sage.functions.trig.**Function_arccos**

Bases: GinacFunction

The arccosine function.

EXAMPLES:

```
sage: arccos(0.5)
1.04719755119660
sage: arccos(1/2)                                                       #␣
↪needs sage.symbolic
1/3*pi
sage: arccos(1 + 1.0*I)                                                 #␣
↪needs sage.symbolic
0.904556894302381 - 1.06127506190504*I
sage: arccos(3/4).n(100)                                                #␣
↪needs sage.symbolic
0.72273424781341561117837735264
```

We can delay evaluation using the `hold` parameter:

```
sage: arccos(0, hold=True)                                              #␣
↪needs sage.symbolic
arccos(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccos(0, hold=True); a.simplify()                            #␣
↪needs sage.symbolic
1/2*pi
```

`conjugate(arccos(x))==arccos(conjugate(x))`, unless on the branch cuts, which run along the real axis outside the interval [-1, +1].:

```
sage: # needs sage.symbolic
sage: conjugate(arccos(x))
conjugate(arccos(x))
sage: var('y', domain='positive')
y
```

(continues on next page)

(continued from previous page)

```
sage: conjugate(arccos(y))
conjugate(arccos(y))
sage: conjugate(arccos(y+I))
conjugate(arccos(y + I))
sage: conjugate(arccos(1/16))
arccos(1/16)
sage: conjugate(arccos(2))
conjugate(arccos(2))
sage: conjugate(arccos(-2))
pi - conjugate(arccos(2))
```

class sage.functions.trig.**Function_arccot**

Bases: GinacFunction

The arccotangent function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: arccot(1/2)
arccot(1/2)
sage: RDF(arccot(1/2))  # abs tol 2e-16
1.1071487177940906
sage: arccot(1 + I)
arccot(I + 1)
sage: arccot(1/2).n(100)
1.1071487177940905030170654602
sage: float(arccot(1/2))  # abs tol 2e-16
1.1071487177940906
sage: bool(diff(acot(x), x) == -diff(atan(x), x))
True
sage: diff(acot(x), x)
-1/(x^2 + 1)
```

We can delay evaluation using the `hold` parameter:

```
sage: arccot(1, hold=True)                                              #␣
↪needs sage.symbolic
arccot(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccot(1, hold=True); a.simplify()                            #␣
↪needs sage.symbolic
1/4*pi
```

class sage.functions.trig.**Function_arccsc**

Bases: GinacFunction

The arccosecant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: arccsc(2)
arccsc(2)
```

(continues on next page)

(continued from previous page)

```
sage: RDF(arccsc(2))  # rel tol 1e-15
0.5235987755982988
sage: arccsc(2).n(100)
0.52359877559829887307710723055
sage: float(arccsc(2))
0.52359877559829...
sage: arccsc(1 + I)
arccsc(I + 1)
sage: diff(acsc(x), x)
-1/(sqrt(x^2 - 1)*x)
sage: arccsc(x)._sympy_()                                                    #␣
↪needs sympy
acsc(x)
```

We can delay evaluation using the `hold` parameter:

```
sage: arccsc(1, hold=True)                                                   #␣
↪needs sage.symbolic
arccsc(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arccsc(1, hold=True); a.simplify()                                 #␣
↪needs sage.symbolic
1/2*pi
```

class sage.functions.trig.**Function_arcsec**

Bases: `GinacFunction`

The arcsecant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: arcsec(2)
arcsec(2)
sage: arcsec(2.0)
1.04719755119660
sage: arcsec(2).n(100)
1.0471975511965977461542144611
sage: arcsec(1/2).n(100)
1.3169578969248167086250463473*I
sage: RDF(arcsec(2))  # abs tol 1e-15
1.0471975511965976
sage: arcsec(1 + I)
arcsec(I + 1)
sage: diff(asec(x), x)
1/(sqrt(x^2 - 1)*x)
sage: arcsec(x)._sympy_()                                                    #␣
↪needs sympy
asec(x)
```

We can delay evaluation using the `hold` parameter:

```
sage: arcsec(1, hold=True)                                          #␣
↪needs sage.symbolic
arcsec(1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arcsec(1, hold=True); a.simplify()                        #␣
↪needs sage.symbolic
0
```

class sage.functions.trig.**Function_arcsin**

Bases: `GinacFunction`

The arcsine function.

EXAMPLES:

```
sage: arcsin(0.5)
0.523598775598299
sage: arcsin(1/2)                                                   #␣
↪needs sage.symbolic
1/6*pi
sage: arcsin(1 + 1.0*I)                                             #␣
↪needs sage.symbolic
0.666239432492515 + 1.06127506190504*I
```

We can delay evaluation using the `hold` parameter:

```
sage: arcsin(0, hold=True)                                          #␣
↪needs sage.symbolic
arcsin(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = arcsin(0, hold=True); a.simplify()                        #␣
↪needs sage.symbolic
0
```

`conjugate(arcsin(x))==arcsin(conjugate(x))`, unless on the branch cuts which run along the real axis outside the interval [-1, +1].:

```
sage: # needs sage.symbolic
sage: conjugate(arcsin(x))
conjugate(arcsin(x))
sage: var('y', domain='positive')
y
sage: conjugate(arcsin(y))
conjugate(arcsin(y))
sage: conjugate(arcsin(y+I))
conjugate(arcsin(y + I))
sage: conjugate(arcsin(1/16))
arcsin(1/16)
sage: conjugate(arcsin(2))
conjugate(arcsin(2))
sage: conjugate(arcsin(-2))
-conjugate(arcsin(2))
```

class sage.functions.trig.**Function_arctan**

Bases: GinacFunction

The arctangent function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: arctan(1/2)
arctan(1/2)
sage: RDF(arctan(1/2))  # rel tol 1e-15
0.46364760900080615
sage: arctan(1 + I)
arctan(I + 1)
sage: arctan(1/2).n(100)
0.46364760900080611621425623146
```

We can delay evaluation using the hold parameter:

```
sage: arctan(0, hold=True)                                              #␣
↪needs sage.symbolic
arctan(0)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: a = arctan(0, hold=True); a.simplify()                            #␣
↪needs sage.symbolic
0
```

conjugate(arctan(x))==arctan(conjugate(x)), unless on the branch cuts which run along the imaginary axis outside the interval [-I, +I].:

```
sage: # needs sage.symbolic
sage: conjugate(arctan(x))
conjugate(arctan(x))
sage: var('y', domain='positive')
y
sage: conjugate(arctan(y))
arctan(y)
sage: conjugate(arctan(y+I))
conjugate(arctan(y + I))
sage: conjugate(arctan(1/16))
arctan(1/16)
sage: conjugate(arctan(-2*I))
conjugate(arctan(-2*I))
sage: conjugate(arctan(2*I))
conjugate(arctan(2*I))
sage: conjugate(arctan(I/2))
arctan(-1/2*I)
```

class sage.functions.trig.**Function_arctan2**

Bases: GinacFunction

The modified arctangent function.

Returns the arc tangent (measured in radians) of y/x, where unlike arctan(y/x), the signs of both x and y are considered. In particular, this function measures the angle of a ray through the origin and (x, y), with the positive x-axis the zero mark, and with output angle θ being between $-\pi < \theta <= \pi$.

Hence, `arctan2(y,x) = arctan(y/x)` only for $x > 0$. One may consider the usual arctan to measure angles of lines through the origin, while the modified function measures rays through the origin.

Note that the y-coordinate is by convention the first input.

EXAMPLES:

Note the difference between the two functions:

```
sage: arctan2(1, -1)                                                    #␣
↪needs sage.symbolic
3/4*pi
sage: arctan(1/-1)                                                      #␣
↪needs sage.symbolic
-1/4*pi
```

This is consistent with Python and Maxima:

```
sage: maxima.atan2(1, -1)                                               #␣
↪needs sage.symbolic
(3*%pi)/4
sage: math.atan2(1,-1)
2.356194490192345
```

More examples:

```
sage: arctan2(1, 0)                                                     #␣
↪needs sage.symbolic
1/2*pi
sage: arctan2(2, 3)                                                     #␣
↪needs sage.symbolic
arctan(2/3)
sage: arctan2(-1, -1)                                                   #␣
↪needs sage.symbolic
-3/4*pi
```

Of course we can approximate as well:

```
sage: arctan2(-1/2, 1).n(100)                                           #␣
↪needs sage.symbolic
-0.46364760900080611621425623146
sage: arctan2(2, 3).n(100)                                              #␣
↪needs sage.symbolic
0.58800260354756755124561108063
```

We can delay evaluation using the `hold` parameter:

```
sage: arctan2(-1/2, 1, hold=True)                                       #␣
↪needs sage.symbolic
arctan2(-1/2, 1)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: arctan2(-1/2, 1, hold=True).simplify()                            #␣
↪needs sage.symbolic
-arctan(1/2)
```

The function also works with numpy arrays as input:

```
sage: # needs numpy
sage: import numpy
sage: a = numpy.linspace(1, 3, 3)
sage: b = numpy.linspace(3, 6, 3)
sage: atan2(a, b)
array([0.32175055, 0.41822433, 0.46364761])

sage: atan2(1,a)                                                            #␣
↪needs numpy
array([0.78539816, 0.46364761, 0.32175055])

sage: atan2(a, 1)                                                           #␣
↪needs numpy
array([0.78539816, 1.10714872, 1.24904577])
```

class sage.functions.trig.**Function_cos**

Bases: GinacFunction

The cosine function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: cos(pi)
-1
sage: cos(x).subs(x==pi)
-1
sage: cos(2).n(100)
-0.41614683654714238699756822950
sage: cos(x)._sympy_()                                                      #␣
↪needs sympy
cos(x)
```

We can prevent evaluation using the hold parameter:

```
sage: cos(0, hold=True)                                                     #␣
↪needs sage.symbolic
cos(0)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: a = cos(0, hold=True); a.simplify()                                   #␣
↪needs sage.symbolic
1
```

If possible, the argument is also reduced modulo the period length 2π, and well-known identities are directly evaluated:

```
sage: # needs sage.symbolic
sage: k = var('k', domain='integer')
sage: cos(1 + 2*k*pi)
cos(1)
sage: cos(k*pi)
cos(pi*k)
sage: cos(pi/3 + 2*k*pi)
1/2
```

class sage.functions.trig.**Function_cot**

Bases: GinacFunction

The cotangent function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: cot(pi/4)
1
sage: RR(cot(pi/4))
1.00000000000000
sage: cot(1/2)
cot(1/2)
sage: cot(0.5)
1.83048772171245

sage: latex(cot(x))                                                      #␣
↪needs sage.symbolic
\cot\left(x\right)
sage: cot(x)._sympy_()                                                   #␣
↪needs sympy sage.symbolic
cot(x)
```

We can prevent evaluation using the `hold` parameter:

```
sage: cot(pi/4, hold=True)                                               #␣
↪needs sage.symbolic
cot(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = cot(pi/4, hold=True); a.simplify()                             #␣
↪needs sage.symbolic
1
```

EXAMPLES:

```
sage: # needs sage.symbolic
sage: cot(pi/4)
1
sage: cot(x).subs(x==pi/4)
1
sage: cot(pi/7)
cot(1/7*pi)
sage: cot(x)
cot(x)

sage: # needs sage.symbolic
sage: n(cot(pi/4), 100)
1.0000000000000000000000000000
sage: float(cot(1))
0.64209261593433...
sage: bool(diff(cot(x), x) == diff(1/tan(x), x))
True
sage: diff(cot(x), x)
-cot(x)^2 - 1
```

class sage.functions.trig.**Function_csc**

Bases: GinacFunction

The cosecant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: csc(pi/4)
sqrt(2)
sage: csc(x).subs(x==pi/4)
sqrt(2)
sage: csc(pi/7)
csc(1/7*pi)
sage: csc(x)
csc(x)
sage: RR(csc(pi/4))
1.41421356237310
sage: n(csc(pi/4), 100)
1.4142135623730950488016887242
sage: float(csc(pi/4))
1.4142135623730951
sage: csc(1/2)
csc(1/2)
sage: csc(0.5)
2.08582964293349

sage: # needs sage.symbolic
sage: bool(diff(csc(x), x) == diff(1/sin(x), x))
True
sage: diff(csc(x), x)
-cot(x)*csc(x)
sage: latex(csc(x))
\csc\left(x\right)
sage: csc(x)._sympy_()                                                    #␣
↪needs sympy
csc(x)
```

We can prevent evaluation using the `hold` parameter:

```
sage: csc(pi/4, hold=True)                                                #␣
↪needs sage.symbolic
csc(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = csc(pi/4,hold=True); a.simplify()                              #␣
↪needs sage.symbolic
sqrt(2)
```

class sage.functions.trig.**Function_sec**

Bases: GinacFunction

The secant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: sec(pi/4)
sqrt(2)
sage: sec(x).subs(x==pi/4)
sqrt(2)
sage: sec(pi/7)
sec(1/7*pi)
sage: sec(x)
sec(x)
sage: RR(sec(pi/4))
1.41421356237310
sage: n(sec(pi/4),100)
1.4142135623730950488016887242
sage: float(sec(pi/4))
1.4142135623730951
sage: sec(1/2)
sec(1/2)
sage: sec(0.5)
1.13949392732455

sage: # needs sage.symbolic
sage: bool(diff(sec(x), x) == diff(1/cos(x), x))
True
sage: diff(sec(x), x)
sec(x)*tan(x)
sage: latex(sec(x))
\sec\left(x\right)
sage: sec(x)._sympy_()                                                    #␣
↪needs sympy
sec(x)
```

We can prevent evaluation using the `hold` parameter:

```
sage: sec(pi/4, hold=True)                                                #␣
↪needs sage.symbolic
sec(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = sec(pi/4, hold=True); a.simplify()                              #␣
↪needs sage.symbolic
sqrt(2)
```

class sage.functions.trig.**Function_sin**

Bases: `GinacFunction`

The sine function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: sin(0)
0
sage: sin(x).subs(x==0)
0
sage: sin(2).n(100)
0.90929742682568169539601986591
```

(continues on next page)

(continued from previous page)

```
sage: sin(x)._sympy_()                                                    #␣
↪needs sympy
sin(x)
```

We can prevent evaluation using the `hold` parameter:

```
sage: sin(0, hold=True)                                                   #␣
↪needs sage.symbolic
sin(0)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = sin(0, hold=True); a.simplify()                                 #␣
↪needs sage.symbolic
0
```

If possible, the argument is also reduced modulo the period length 2π, and well-known identities are directly evaluated:

```
sage: k = var('k', domain='integer')                                      #␣
↪needs sage.symbolic
sage: sin(1 + 2*k*pi)                                                     #␣
↪needs sage.symbolic
sin(1)
sage: sin(k*pi)                                                           #␣
↪needs sage.symbolic
0
```

class sage.functions.trig.**Function_tan**

Bases: `GinacFunction`

The tangent function.

EXAMPLES:

```
sage: tan(3.1415)
-0.0000926535900581913
sage: tan(3.1415/4)
0.999953674278156

sage: # needs sage.symbolic
sage: tan(pi)
0
sage: tan(pi/4)
1
sage: tan(1/2)
tan(1/2)
sage: RR(tan(1/2))
0.546302489843790
```

We can prevent evaluation using the `hold` parameter:

```
sage: tan(pi/4, hold=True)                                                #␣
↪needs sage.symbolic
tan(1/4*pi)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = tan(pi/4, hold=True); a.simplify()                    #␣
↪needs sage.symbolic
1
```

If possible, the argument is also reduced modulo the period length π, and well-known identities are directly evaluated:

```
sage: k = var('k', domain='integer')                            #␣
↪needs sage.symbolic
sage: tan(1 + 2*k*pi)                                           #␣
↪needs sage.symbolic
tan(1)
sage: tan(k*pi)                                                 #␣
↪needs sage.symbolic
0
```

1.3 Hyperbolic functions

The full set of hyperbolic and inverse hyperbolic functions is available:

- hyperbolic sine: *sinh()*
- hyperbolic cosine: *cosh()*
- hyperbolic tangent: *tanh()*
- hyperbolic cotangent: *coth()*
- hyperbolic secant: *sech()*
- hyperbolic cosecant: *csch()*
- inverse hyperbolic sine: *asinh()*
- inverse hyperbolic cosine: *acosh()*
- inverse hyperbolic tangent: *atanh()*
- inverse hyperbolic cotangent: *acoth()*
- inverse hyperbolic secant: *asech()*
- inverse hyperbolic cosecant: *acsch()*

REFERENCES:

- Wikipedia article Hyperbolic function
- Wikipedia article Inverse hyperbolic functions
- R. Roy, F. W. J. Olver, Elementary Functions, https://dlmf.nist.gov/4

EXAMPLES:

Inverse hyperbolic functions have logarithmic expressions, so expressions of the form `exp(c*f(x))` simplify:

```
sage: # needs sage.symbolic
sage: exp(2*atanh(x))
-(x + 1)/(x - 1)
sage: exp(2*acoth(x))
(x + 1)/(x - 1)
sage: exp(2*asinh(x))
(x + sqrt(x^2 + 1))^2
sage: exp(2*acosh(x))
(x + sqrt(x^2 - 1))^2
sage: exp(2*asech(x))
(sqrt(-x^2 + 1)/x + 1/x)^2
sage: exp(2*acsch(x))
(sqrt(1/x^2 + 1) + 1/x)^2
```

class sage.functions.hyperbolic.**Function_arccosh**

Bases: GinacFunction

The inverse of the hyperbolic cosine function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: acosh(1/2)
arccosh(1/2)
sage: acosh(1 + I*1.0)
1.06127506190504 + 0.904556894302381*I
sage: float(acosh(2))
1.3169578969248168
sage: cosh(float(acosh(2)))
2.0

sage: acosh(complex(1, 2))  # abs tol 1e-15                                    #␣
↪needs sage.rings.complex_double
(1.5285709194809982+1.1437177404024204j)
```

Warning: If the input is in the complex field or symbolic (which includes rational and integer input), the output will be complex. However, if the input is a real decimal, the output will be real or NaN. See the examples for details.

```
sage: acosh(CC(0.5))                                                           #␣
↪needs sage.rings.real_mpfr
1.04719755119660*I

sage: # needs sage.symbolic
sage: acosh(0.5)
NaN
sage: acosh(1/2)
arccosh(1/2)
sage: acosh(1/2).n()
NaN
sage: acosh(0)
1/2*I*pi
sage: acosh(-1)
I*pi
```

To prevent automatic evaluation use the `hold` argument:

```
sage: acosh(-1, hold=True)                                                  #␣
↪needs sage.symbolic
arccosh(-1)
```

To then evaluate again, use the `unhold` method:

```
sage: acosh(-1, hold=True).unhold()                                         #␣
↪needs sage.symbolic
I*pi
```

`conjugate(arccosh(x))==arccosh(conjugate(x))` unless on the branch cut which runs along the real axis from +1 to -inf.:

```
sage: # needs sage.symbolic
sage: conjugate(acosh(x))
conjugate(arccosh(x))
sage: var('y', domain='positive')
y
sage: conjugate(acosh(y))
conjugate(arccosh(y))
sage: conjugate(acosh(y+I))
conjugate(arccosh(y + I))
sage: conjugate(acosh(1/16))
conjugate(arccosh(1/16))
sage: conjugate(acosh(2))
arccosh(2)
sage: conjugate(acosh(I/2))
arccosh(-1/2*I)
```

class sage.functions.hyperbolic.**Function_arccoth**

Bases: `GinacFunction`

The inverse of the hyperbolic cotangent function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: acoth(2.0)
0.549306144334055
sage: acoth(2)
1/2*log(3)
sage: acoth(1 + I*1.0)
0.402359478108525 - 0.553574358897045*I
sage: acoth(2).n(200)
0.54930614433405484569762261846126285232374527891137472586735

sage: bool(diff(acoth(x), x) == diff(atanh(x), x))                          #␣
↪needs sage.symbolic
True
sage: diff(acoth(x), x)                                                     #␣
↪needs sage.symbolic
-1/(x^2 - 1)

sage: float(acoth(2))                                                       #␣
↪needs sage.symbolic
0.5493061443340549
sage: float(acoth(2).n(53))    # Correct result to 53 bits                  #␣
↪needs sage.rings.real_mpfr sage.symbolic
```

(continues on next page)

(continued from previous page)

```
0.5493061443340549
sage: float(acoth(2).n(100))   # Compute 100 bits and then round to 53       #␣
↪needs sage.rings.real_mpfr sage.symbolic
0.5493061443340549
```

class sage.functions.hyperbolic.**Function_arccsch**

Bases: GinacFunction

The inverse of the hyperbolic cosecant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: acsch(2.0)
0.481211825059603
sage: acsch(2)
arccsch(2)
sage: acsch(1 + I*1.0)
0.530637530952518 - 0.452278447151191*I
sage: acsch(1).n(200)
0.88137358701954302523260932497979230902816032826163541075330
sage: float(acsch(1))
0.881373587019543

sage: diff(acsch(x), x)                                                      #␣
↪needs sage.symbolic
-1/(sqrt(x^2 + 1)*x)
sage: latex(acsch(x))                                                        #␣
↪needs sage.symbolic
\operatorname{arcsch}\left(x\right)
```

class sage.functions.hyperbolic.**Function_arcsech**

Bases: GinacFunction

The inverse of the hyperbolic secant function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: asech(0.5)
1.31695789692482
sage: asech(1/2)
arcsech(1/2)
sage: asech(1 + I*1.0)
0.530637530952518 - 1.11851787964371*I
sage: asech(1/2).n(200)
1.3169578969248167086250463473079684440269819714675164797685
sage: float(asech(1/2))
1.3169578969248168

sage: diff(asech(x), x)                                                      #␣
↪needs sage.symbolic
-1/(sqrt(-x^2 + 1)*x)
sage: latex(asech(x))                                                        #␣
↪needs sage.symbolic
\operatorname{arsech}\left(x\right)
sage: asech(x)._sympy_()                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sympy sage.symbolic
asech(x)
```

class sage.functions.hyperbolic.**Function_arcsinh**

Bases: GinacFunction

The inverse of the hyperbolic sine function.

EXAMPLES:

```
sage: asinh
arcsinh
sage: asinh(0.5)
0.481211825059603
sage: asinh(1/2)                                                    #␣
↪needs sage.symbolic
arcsinh(1/2)
sage: asinh(1 + I*1.0)                                              #␣
↪needs sage.symbolic
1.06127506190504 + 0.666239432492515*I
```

To prevent automatic evaluation use the `hold` argument:

```
sage: asinh(-2, hold=True)                                          #␣
↪needs sage.symbolic
arcsinh(-2)
```

To then evaluate again, use the `unhold` method:

```
sage: asinh(-2, hold=True).unhold()                                 #␣
↪needs sage.symbolic
-arcsinh(2)
```

`conjugate(asinh(x))==asinh(conjugate(x))` unless on the branch cuts which run along the imaginary axis outside the interval [-I, +I].:

```
sage: # needs sage.symbolic
sage: conjugate(asinh(x))
conjugate(arcsinh(x))
sage: var('y', domain='positive')
y
sage: conjugate(asinh(y))
arcsinh(y)
sage: conjugate(asinh(y+I))
conjugate(arcsinh(y + I))
sage: conjugate(asinh(1/16))
arcsinh(1/16)
sage: conjugate(asinh(I/2))
arcsinh(-1/2*I)
sage: conjugate(asinh(2*I))
conjugate(arcsinh(2*I))
```

class sage.functions.hyperbolic.**Function_arctanh**

Bases: GinacFunction

The inverse of the hyperbolic tangent function.

EXAMPLES:

```
sage: atanh(0.5)
0.549306144334055
sage: atanh(1/2)                                                            #␣
↪needs sage.symbolic
1/2*log(3)
sage: atanh(1 + I*1.0)                                                      #␣
↪needs sage.symbolic
0.402359478108525 + 1.01722196789785*I
```

To prevent automatic evaluation use the `hold` argument:

```
sage: atanh(-1/2, hold=True)                                                #␣
↪needs sage.symbolic
arctanh(-1/2)
```

To then evaluate again, use the `unhold` method:

```
sage: atanh(-1/2, hold=True).unhold()                                       #␣
↪needs sage.symbolic
-1/2*log(3)
```

`conjugate(arctanh(x)) == arctanh(conjugate(x))` unless on the branch cuts which run along the real axis outside the interval [-1, +1].

```
sage: # needs sage.symbolic
sage: conjugate(atanh(x))
conjugate(arctanh(x))
sage: var('y', domain='positive')
y
sage: conjugate(atanh(y))
conjugate(arctanh(y))
sage: conjugate(atanh(y + I))
conjugate(arctanh(y + I))
sage: conjugate(atanh(1/16))
1/2*log(17/15)
sage: conjugate(atanh(I/2))
arctanh(-1/2*I)
sage: conjugate(atanh(-2*I))
arctanh(2*I)
```

class `sage.functions.hyperbolic.`**`Function_cosh`**

Bases: `GinacFunction`

The hyperbolic cosine function.

EXAMPLES:

```
sage: cosh(3.1415)
11.5908832931176

sage: # needs sage.symbolic
sage: cosh(pi)
cosh(pi)
sage: float(cosh(pi))
11.591953275521519
sage: RR(cosh(1/2))
1.12762596520638
sage: latex(cosh(x))
```

(continues on next page)

(continued from previous page)

```
\cosh\left(x\right)
sage: cosh(x)._sympy_()                                                    #␣
↪needs sympy
cosh(x)
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: cosh(arcsinh(x), hold=True)                                          #␣
↪needs sage.symbolic
cosh(arcsinh(x))
```

To then evaluate again, use the `unhold` method:

```
sage: cosh(arcsinh(x), hold=True).unhold()                                 #␣
↪needs sage.symbolic
sqrt(x^2 + 1)
```

class sage.functions.hyperbolic.**Function_coth**

Bases: `GinacFunction`

The hyperbolic cotangent function.

EXAMPLES:

```
sage: coth(3.1415)
1.00374256795520
sage: coth(complex(1, 2))   # abs tol 1e-15                                #␣
↪needs sage.rings.complex_double
(0.8213297974938518+0.17138361290918508j)

sage: # needs sage.symbolic
sage: coth(pi)
coth(pi)
sage: coth(0)
Infinity
sage: coth(pi*I)
Infinity
sage: coth(pi*I/2)
0
sage: coth(7*pi*I/2)
0
sage: coth(8*pi*I/2)
Infinity
sage: coth(7.*pi*I/2)
-I*cot(3.50000000000000*pi)
sage: float(coth(pi))
1.0037418731973213
sage: RR(coth(pi))
1.00374187319732

sage: # needs sage.symbolic
sage: bool(diff(coth(x), x) == diff(1/tanh(x), x))
True
sage: diff(coth(x), x)
-1/sinh(x)^2
sage: latex(coth(x))
\coth\left(x\right)
```

(continues on next page)

(continued from previous page)

```
sage: coth(x)._sympy_()                                                    #␣
↪needs sympy
coth(x)
```

class sage.functions.hyperbolic.**Function_csch**

Bases: GinacFunction

The hyperbolic cosecant function.

EXAMPLES:

```
sage: csch(3.1415)
0.0865975907592133

sage: # needs sage.symbolic
sage: csch(pi)
csch(pi)
sage: float(csch(pi))
0.0865895375300469...
sage: RR(csch(pi))
0.0865895375300470
sage: csch(0)
Infinity
sage: csch(pi*I)
Infinity
sage: csch(pi*I/2)
-I
sage: csch(7*pi*I/2)
I
sage: csch(7.*pi*I/2)
-I*csc(3.50000000000000*pi)

sage: # needs sage.symbolic
sage: bool(diff(csch(x), x) == diff(1/sinh(x), x))
True
sage: diff(csch(x), x)
-coth(x)*csch(x)
sage: latex(csch(x))
\operatorname{csch}\left(x\right)
sage: csch(x)._sympy_()                                                    #␣
↪needs sympy
csch(x)
```

class sage.functions.hyperbolic.**Function_sech**

Bases: GinacFunction

The hyperbolic secant function.

EXAMPLES:

```
sage: sech(3.1415)
0.0862747018248192

sage: # needs sage.symbolic
sage: sech(pi)
sech(pi)
sage: float(sech(pi))
```

(continues on next page)

(continued from previous page)

```
0.0862667383340544...
sage: RR(sech(pi))
0.0862667383340544
sage: sech(0)
1
sage: sech(pi*I)
-1
sage: sech(pi*I/2)
Infinity
sage: sech(7*pi*I/2)
Infinity
sage: sech(8*pi*I/2)
1
sage: sech(8.*pi*I/2)
sec(4.00000000000000*pi)

sage: # needs sage.symbolic
sage: bool(diff(sech(x), x) == diff(1/cosh(x), x))
True
sage: diff(sech(x), x)
-sech(x)*tanh(x)
sage: latex(sech(x))
\operatorname{sech}\left(x\right)
sage: sech(x)._sympy_()                                                    #␣
↪needs sympy
sech(x)
```

class sage.functions.hyperbolic.**Function_sinh**

Bases: GinacFunction

The hyperbolic sine function.

EXAMPLES:

```
sage: sinh(3.1415)
11.5476653707437

sage: # needs sage.symbolic
sage: sinh(pi)
sinh(pi)
sage: float(sinh(pi))
11.54873935725774...
sage: RR(sinh(pi))
11.5487393572577
sage: latex(sinh(x))
\sinh\left(x\right)
sage: sinh(x)._sympy_()                                                    #␣
↪needs sympy
sinh(x)
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: sinh(arccosh(x), hold=True)                                          #␣
↪needs sage.symbolic
sinh(arccosh(x))
```

To then evaluate again, use the `unhold` method:

```
sage: sinh(arccosh(x), hold=True).unhold()                           #␣
↪needs sage.symbolic
sqrt(x + 1)*sqrt(x - 1)
```

class sage.functions.hyperbolic.**Function_tanh**

Bases: GinacFunction

The hyperbolic tangent function.

EXAMPLES:

```
sage: tanh(3.1415)
0.996271386633702
sage: tan(3.1415/4)
0.999953674278156

sage: # needs sage.symbolic
sage: tanh(pi)
tanh(pi)
sage: float(tanh(pi))
0.99627207622075
sage: tanh(pi/4)
tanh(1/4*pi)
sage: RR(tanh(1/2))
0.462117157260010
```

```
sage: CC(tanh(pi + I*e))                                             #␣
↪needs sage.rings.real_mpfr sage.symbolic
0.997524731976164 - 0.00279068768100315*I
sage: ComplexField(100)(tanh(pi + I*e))                              #␣
↪needs sage.rings.real_mpfr sage.symbolic
0.99752473197616361034204366446 - 0.0027906876810031453884245163923*I
sage: CDF(tanh(pi + I*e))  # rel tol 2e-15                           #␣
↪needs sage.rings.complex_double sage.symbolic
0.9975247319761636 - 0.002790687681003147*I
```

To prevent automatic evaluation, use the `hold` parameter:

```
sage: tanh(arcsinh(x), hold=True)                                    #␣
↪needs sage.symbolic
tanh(arcsinh(x))
```

To then evaluate again, use the `unhold` method:

```
sage: tanh(arcsinh(x), hold=True).unhold()                           #␣
↪needs sage.symbolic
x/sqrt(x^2 + 1)
```

1.4 Number-theoretic functions

class sage.functions.transcendental.**DickmanRho**

Bases: BuiltinFunction

Dickman's function is the continuous function satisfying the differential equation

$$x\rho'(x) + \rho(x-1) = 0$$

with initial conditions $\rho(x) = 1$ for $0 \le x \le 1$. It is useful in estimating the frequency of smooth numbers as asymptotically

$$\Psi(a, a^{1/s}) \sim a\rho(s)$$

where $\Psi(a, b)$ is the number of b-smooth numbers less than a.

ALGORITHM:

Dickmans's function is analytic on the interval $[n, n+1]$ for each integer n. To evaluate at $n+t, 0 \le t < 1$, a power series is recursively computed about $n+1/2$ using the differential equation stated above. As high precision arithmetic may be needed for intermediate results the computed series are cached for later use.

Simple explicit formulas are used for the intervals [0,1] and [1,2].

EXAMPLES:

```
sage: # needs sage.symbolic
sage: dickman_rho(2)
0.306852819440055
sage: dickman_rho(10)
2.77017183772596e-11
sage: dickman_rho(10.00000000000000000000000000000000000000)
2.77017183772595898875812120063434232634e-11
sage: plot(log(dickman_rho(x)), (x, 0, 15))                                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

AUTHORS:

- Robert Bradshaw (2008-09)

REFERENCES:

- G. Marsaglia, A. Zaman, J. Marsaglia. "Numerical Solutions to some Classical Differential-Difference Equations." Mathematics of Computation, Vol. 53, No. 187 (1989).

approximate (*x*, *parent=None*)

Approximate using de Bruijn's formula

$$\rho(x) \sim \frac{exp(-x\xi + Ei(\xi))}{\sqrt{2\pi x}\xi}$$

which is asymptotically equal to Dickman's function, and is much faster to compute.

REFERENCES:

- N. De Bruijn, "The Asymptotic behavior of a function occurring in the theory of primes." J. Indian Math Soc. v 15. (1951)

EXAMPLES:

```
sage: dickman_rho.approximate(10)                                         #␣
↪needs sage.rings.real_mpfr
2.41739196365564e-11
sage: dickman_rho(10)                                                     #␣
↪needs sage.symbolic
2.77017183772596e-11
sage: dickman_rho.approximate(1000)                                       #␣
↪needs sage.rings.real_mpfr
4.32938809066403e-3464
```

power_series (*n*, *abs_prec*)

This function returns the power series about $n + 1/2$ used to evaluate Dickman's function. It is scaled such that the interval $[n, n + 1]$ corresponds to x in $[-1, 1]$.

INPUT:

- `n` - the lower endpoint of the interval for which this power series holds
- `abs_prec` - the absolute precision of the resulting power series

EXAMPLES:

```
sage: f = dickman_rho.power_series(2, 20); f
-9.9376e-8*x^11 + 3.7722e-7*x^10 - 1.4684e-6*x^9 + 5.8783e-6*x^8 - 0.
↪000024259*x^7 + 0.00010341*x^6 - 0.00045583*x^5 + 0.0020773*x^4 - 0.
↪0097336*x^3 + 0.045224*x^2 - 0.11891*x + 0.13032
sage: f(-1), f(0), f(1)
(0.30685, 0.13032, 0.048608)
sage: dickman_rho(2), dickman_rho(2.5), dickman_rho(3)                    #␣
↪needs sage.symbolic
(0.306852819440055, 0.130319561832251, 0.0486083882911316)
```

class sage.functions.transcendental.**Function_HurwitzZeta**

Bases: `BuiltinFunction`

class sage.functions.transcendental.**Function_stieltjes**

Bases: `GinacFunction`

Stieltjes constant of index `n`.

`stieltjes(0)` is identical to the Euler-Mascheroni constant (`sage.symbolic.constants.EulerGamma`). The Stieltjes constants are used in the series expansions of $\zeta(s)$.

INPUT:

- `n` - non-negative integer

EXAMPLES:

```
sage: # needs sage.symbolic
sage: _ = var('n')
sage: stieltjes(n)
stieltjes(n)
sage: stieltjes(0)
euler_gamma
sage: stieltjes(2)
stieltjes(2)
sage: stieltjes(int(2))
stieltjes(2)
sage: stieltjes(2).n(100)
```

(continues on next page)

(continued from previous page)

```
-0.00969036319287231848453038603521
sage: RR = RealField(200)                                                #␣
↪needs sage.rings.real_mpfr
sage: stieltjes(RR(2))                                                   #␣
↪needs sage.rings.real_mpfr
-0.0096903631928723184845303860352125293590658061013407498807014
```

It is possible to use the `hold` argument to prevent automatic evaluation:

```
sage: stieltjes(0, hold=True)                                            #␣
↪needs sage.symbolic
stieltjes(0)

sage: # needs sage.symbolic
sage: latex(stieltjes(n))
\gamma_{n}
sage: a = loads(dumps(stieltjes(n)))
sage: a.operator() == stieltjes
True
sage: stieltjes(x)._sympy_()                                             #␣
↪needs sympy
stieltjes(x)

sage: stieltjes(x).subs(x==0)                                            #␣
↪needs sage.symbolic
euler_gamma
```

class sage.functions.transcendental.**Function_zeta**

Bases: `GinacFunction`

Riemann zeta function at s with s a real or complex number.

INPUT:

- `s` - real or complex number

If s is a real number, the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

EXAMPLES:

```
sage: RR = RealField(200)                                                #␣
↪needs sage.rings.real_mpfr
sage: zeta(RR(2))                                                        #␣
↪needs sage.rings.real_mpfr
1.6449340668482264364724151666460251892189499012067984377356

sage: # needs sage.symbolic
sage: zeta(x)
zeta(x)
sage: zeta(2)
1/6*pi^2
sage: zeta(2.)
1.64493406684823
sage: zeta(I)
zeta(I)
sage: zeta(I).n()
0.00330022368532410 - 0.418155449141322*I
```

(continues on next page)

(continued from previous page)

```
sage: zeta(sqrt(2))
zeta(sqrt(2))
sage: zeta(sqrt(2)).n()   # rel tol 1e-10
3.02073767948603
```

It is possible to use the `hold` argument to prevent automatic evaluation:

```
sage: zeta(2, hold=True)                                                    #␣
↪needs sage.symbolic
zeta(2)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: a = zeta(2, hold=True); a.simplify()                                  #␣
↪needs sage.symbolic
1/6*pi^2
```

The Laurent expansion of $\zeta(s)$ at $s = 1$ is implemented by means of the Stieltjes constants:

```
sage: s = SR('s')                                                           #␣
↪needs sage.symbolic
sage: zeta(s).series(s==1, 2)                                               #␣
↪needs sage.symbolic
1*(s - 1)^(-1) + euler_gamma + (-stieltjes(1))*(s - 1) + Order((s - 1)^2)
```

Generally, the Stieltjes constants occur in the Laurent expansion of ζ-type singularities:

```
sage: zeta(2*s/(s+1)).series(s==1, 2)                                       #␣
↪needs sage.symbolic
2*(s - 1)^(-1) + (euler_gamma + 1) + (-1/2*stieltjes(1))*(s - 1) + Order((s - 1)^
↪2)
```

class sage.functions.transcendental.**Function_zetaderiv**

Bases: `GinacFunction`

Derivatives of the Riemann zeta function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: zetaderiv(1, x)
zetaderiv(1, x)
sage: zetaderiv(1, x).diff(x)
zetaderiv(2, x)
sage: var('n')
n
sage: zetaderiv(n, x)
zetaderiv(n, x)
sage: zetaderiv(1, 4).n()
-0.0689112658961254
sage: import mpmath; mpmath.diff(lambda x: mpmath.zeta(x), 4)               #␣
↪needs mpmath
mpf('-0.068911265896125382')
```

sage.functions.transcendental.**hurwitz_zeta**(*s*, *x*, ***kwargs*)

The Hurwitz zeta function $\zeta(s, x)$, where s and x are complex.

The Hurwitz zeta function is one of the many zeta functions. It is defined as

$$\zeta(s,x) = \sum_{k=0}^{\infty} (k+x)^{-s}.$$

When $x = 1$, this coincides with Riemann's zeta function. The Dirichlet L-functions may be expressed as linear combinations of Hurwitz zeta functions.

EXAMPLES:

Symbolic evaluations:

```
sage: # needs sage.symbolic
sage: hurwitz_zeta(x, 1)
zeta(x)
sage: hurwitz_zeta(4, 3)
1/90*pi^4 - 17/16
sage: hurwitz_zeta(-4, x)
-1/5*x^5 + 1/2*x^4 - 1/3*x^3 + 1/30*x
sage: hurwitz_zeta(7, -1/2)
127*zeta(7) - 128
sage: hurwitz_zeta(-3, 1)
1/120
```

Numerical evaluations:

```
sage: hurwitz_zeta(3, 1/2).n()                                              #␣
↪needs mpmath
8.41439832211716
sage: hurwitz_zeta(11/10, 1/2).n()                                          #␣
↪needs sage.symbolic
12.1038134956837
sage: hurwitz_zeta(3, x).series(x, 60).subs(x=0.5).n()                      #␣
↪needs sage.symbolic
8.41439832211716
sage: hurwitz_zeta(3, 0.5)                                                  #␣
↪needs mpmath
8.41439832211716
```

REFERENCES:

- Wikipedia article Hurwitz_zeta_function

sage.functions.transcendental.**zeta_symmetric**(s)

Completed function $\xi(s)$ that satisfies $\xi(s) = \xi(1-s)$ and has zeros at the same points as the Riemann zeta function.

INPUT:

- `s` - real or complex number

If s is a real number the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

More precisely,

$$xi(s) = \gamma(s/2+1) * (s-1) * \pi^{-s/2} * \zeta(s).$$

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr
sage: RR = RealField(200)
sage: zeta_symmetric(RR(0.7))
0.49758041465112690357779107525638385212657443284080589766062

sage: # needs sage.libs.pari sage.rings.real_mpfr
sage: zeta_symmetric(0.7)
0.497580414651127
sage: zeta_symmetric(1 - 0.7)
0.497580414651127
sage: C.<i> = ComplexField()
sage: zeta_symmetric(0.5 + i*14.0)
0.000201294444235258 + 1.49077798716757e-19*I
sage: zeta_symmetric(0.5 + i*14.1)
0.0000489893483255687 + 4.40457132572236e-20*I
sage: zeta_symmetric(0.5 + i*14.2)
-0.0000868931282620101 + 7.11507675693612e-20*I
```

REFERENCE:

- I copied the definition of xi from http://web.viu.ca/pughg/RiemannZeta/RiemannZetaLong.html

1.5 Error functions

This module provides symbolic error functions. These functions use the $mpmathlibrary$ for numerical evaluation and Maxima, Pynac for symbolics.

The main objects which are exported from this module are:

- *erf* – The error function
- *erfc* – The complementary error function
- *erfi* – The imaginary error function
- *erfinv* – The inverse error function
- *fresnel_sin* – The Fresnel integral $S(x)$
- *fresnel_cos* – The Fresnel integral $C(x)$

AUTHORS:

- Original authors `erf`/`error_fcn` (c) 2006-2014: Karl-Dieter Crisman, Benjamin Jones, Mike Hansen, William Stein, Burcin Erocal, Jeroen Demeyer, W. D. Joyner, R. Andrew Ohana
- Reorganisation in new file, addition of `erfi`/`erfinv`/`erfc` (c) 2016: Ralf Stephan
- Fresnel integrals (c) 2017 Marcelo Forets

REFERENCES:

- [DLMF-Error]
- [WP-Error]

class sage.functions.error.**Function_Fresnel_cos**

Bases: `BuiltinFunction`

The cosine Fresnel integral.

It is defined by the integral

$$C(x) = \int_0^x \cos\left(\frac{\pi t^2}{2}\right)\, dt$$

for real x. Using power series expansions, it can be extended to the domain of complex numbers. See the Wikipedia article Fresnel_integral.

INPUT:

- x – the argument of the function

EXAMPLES:

```
sage: # needs sage.symbolic
sage: fresnel_cos(0)
0
sage: fresnel_cos(x).subs(x==0)
0
sage: x = var('x')
sage: fresnel_cos(1).n(100)
0.77989340037682282947420641365
sage: fresnel_cos(x)._sympy_()                                              #␣
↪needs sympy
fresnelc(x)
```

class sage.functions.error.**Function_Fresnel_sin**

Bases: BuiltinFunction

The sine Fresnel integral.

It is defined by the integral

$$S(x) = \int_0^x \sin\left(\frac{\pi t^2}{2}\right)\, dt$$

for real x. Using power series expansions, it can be extended to the domain of complex numbers. See the Wikipedia article Fresnel_integral.

INPUT:

- x – the argument of the function

EXAMPLES:

```
sage: # needs sage.symbolic
sage: fresnel_sin(0)
0
sage: fresnel_sin(x).subs(x==0)
0
sage: x = var('x')
sage: fresnel_sin(1).n(100)
0.43825914739035476607675669662
sage: fresnel_sin(x)._sympy_()                                              #␣
↪needs sympy
fresnels(x)
```

class sage.functions.error.**Function_erf**

Bases: BuiltinFunction

The error function.

The error function is defined for real values as

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

This function is also defined for complex values, via analytic continuation.

EXAMPLES:

We can evaluate numerically:

```
sage: erf(2)                                                            #␣
↪needs sage.symbolic
erf(2)
sage: erf(2).n()                                                        #␣
↪needs sage.symbolic
0.995322265018953
sage: erf(2).n(100)                                                     #␣
↪needs sage.symbolic
0.99532226501895273416206925637
sage: erf(ComplexField(100)(2+3j))                                      #␣
↪needs sage.rings.real_mpfr
-20.829461427614568389103088452 + 8.6873182714701631444280787545*I
```

Basic symbolic properties are handled by Sage and Maxima:

```
sage: x = var("x")                                                      #␣
↪needs sage.symbolic
sage: diff(erf(x),x)                                                    #␣
↪needs sage.symbolic
2*e^(-x^2)/sqrt(pi)
sage: integrate(erf(x),x)                                               #␣
↪needs sage.symbolic
x*erf(x) + e^(-x^2)/sqrt(pi)
```

ALGORITHM:

Sage implements numerical evaluation of the error function via the `erf()` function from mpmath. Symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Error_function
- http://mpmath.googlecode.com/svn/trunk/doc/build/functions/expintegrals.html#error-functions

class `sage.functions.error.`**`Function_erfc`**

Bases: `BuiltinFunction`

The complementary error function.

The complementary error function is defined by

$$\frac{2}{\sqrt{\pi}} \int_t^\infty e^{-x^2} dx.$$

EXAMPLES:

```
sage: erfc(6)                                                           #␣
↪needs sage.symbolic
erfc(6)
```

(continues on next page)

(continued from previous page)

```
sage: erfc(6).n()                                                                #␣
↪needs sage.symbolic
2.15197367124989e-17
sage: erfc(RealField(100)(1/2))                                                  #␣
↪needs sage.rings.real_mpfr
0.47950012218695346231725334611

sage: 1 - erfc(0.5)                                                              #␣
↪needs mpmath
0.520499877813047
sage: erf(0.5)                                                                   #␣
↪needs mpmath
0.520499877813047
```

class sage.functions.error.**Function_erfi**

Bases: BuiltinFunction

The imaginary error function.

The imaginary error function is defined by

$$\operatorname{erfi}(x) = -i \operatorname{erf}(ix).$$

class sage.functions.error.**Function_erfinv**

Bases: BuiltinFunction

The inverse error function.

The inverse error function is defined by:

$$\operatorname{erfinv}(x) = \operatorname{erf}^{-1}(x).$$

1.6 Piecewise functions

This module implement piecewise functions in a single variable. See sage.sets.real_set for more information about how to construct subsets of the real line for the domains.

EXAMPLES:

```
sage: f = piecewise([((0,1), x^3), ([-1,0], -x^2)]);  f
piecewise(x|-->x^3 on (0, 1), x|-->-x^2 on [-1, 0]; x)
sage: 2*f
2*piecewise(x|-->x^3 on (0, 1), x|-->-x^2 on [-1, 0]; x)
sage: f(x=1/2)
1/8
sage: plot(f)    # not tested
```

Todo: Implement max/min location and values,

AUTHORS:

- David Joyner (2006-04): initial version

- David Joyner (2006-09): added __eq__, extend_by_zero_to, unextend, convolution, trapezoid, trapezoid_integral_approximation, riemann_sum, riemann_sum_integral_approximation, tangent_line fixed bugs in __mul__, __add__
- David Joyner (2007-03): adding Hann filter for FS, added general FS filter methods for computing and plotting, added options to plotting of FS (eg, specifying rgb values are now allowed). Fixed bug in documentation reported by Pablo De Napoli.
- David Joyner (2007-09): bug fixes due to behaviour of SymbolicArithmetic
- David Joyner (2008-04): fixed docstring bugs reported by J Morrow; added support for Laplace transform of functions with infinite support.
- David Joyner (2008-07): fixed a left multiplication bug reported by C. Boncelet (by defining __rmul__ = __mul__).
- Paul Butler (2009-01): added indefinite integration and default_variable
- Volker Braun (2013): Complete rewrite
- Ralf Stephan (2015): Rewrite of convolution() and other calculus functions; many doctest adaptations
- Eric Gourgoulhon (2017): Improve documentation and user interface of Fourier series

class sage.functions.piecewise.**PiecewiseFunction**

Bases: BuiltinFunction

Piecewise function

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: f = piecewise([((0,1), x^2*y), ([-1,0], -x*y^2)], var=x);  f
piecewise(x|-->x^2*y on (0, 1), x|-->-x*y^2 on [-1, 0]; x)
sage: f(1/2)
1/4*y
sage: f(-1/2)
1/2*y^2
```

class EvaluationMethods

Bases: object

convolution(*parameters*, *variable*, *other*)

Return the convolution function, $f * g(t) = \int_{-\infty}^{\infty} f(u)g(t-u)du$, for compactly supported f, g.

EXAMPLES:

```
sage: x = PolynomialRing(QQ,'x').gen()
sage: f = piecewise([[[0,1],1]])                        ## example 0
sage: g = f.convolution(f); g
piecewise(x|-->x on (0, 1], x|-->-x + 2 on (1, 2]; x)
sage: h = f.convolution(g); h
piecewise(x|-->1/2*x^2 on (0, 1],
          x|-->-x^2 + 3*x - 3/2 on (1, 2],
          x|-->1/2*x^2 - 3*x + 9/2 on (2, 3]; x)
sage: f = piecewise([[(0,1),1], [(1,2),2], [(2,3),1]])  ## example 1
sage: g = f.convolution(f)
sage: h = f.convolution(g); h
piecewise(x|-->1/2*x^2 on (0, 1],
          x|-->2*x^2 - 3*x + 3/2 on (1, 3],
          x|-->-2*x^2 + 21*x - 69/2 on (3, 4],
```

(continues on next page)

(continued from previous page)

```
           x|-->-5*x^2 + 45*x - 165/2 on (4, 5],
           x|-->-2*x^2 + 15*x - 15/2 on (5, 6],
           x|-->2*x^2 - 33*x + 273/2 on (6, 8],
           x|-->1/2*x^2 - 9*x + 81/2 on (8, 9]; x)
sage: f = piecewise([[(-1,1),1]])                      ## example 2
sage: g = piecewise([[(0,3),x]])
sage: f.convolution(g)
piecewise(x|-->1/2*x^2 + x + 1/2 on (-1, 1],
          x|-->2*x on (1, 2],
          x|-->-1/2*x^2 + x + 4 on (2, 4]; x)
sage: g = piecewise([[(0,3),1], [(3,4),2]])
sage: f.convolution(g)
piecewise(x|-->x + 1 on (-1, 1],
          x|-->2 on (1, 2],
          x|-->x on (2, 3],
          x|-->-x + 6 on (3, 4],
          x|-->-2*x + 10 on (4, 5]; x)
```

Check that the bugs raised in github issue #12123 are fixed:

```
sage: f = piecewise([[(-2, 2), 2]])
sage: g = piecewise([[(0, 2), 3/4]])
sage: f.convolution(g)
piecewise(x|-->3/2*x + 3 on (-2, 0],
          x|-->3 on (0, 2],
          x|-->-3/2*x + 6 on (2, 4]; x)
sage: f = piecewise([[(-1, 1), 1]])
sage: g = piecewise([[(0, 1), x], [(1, 2), -x + 2]])
sage: f.convolution(g)
piecewise(x|-->1/2*x^2 + x + 1/2 on (-1, 0],
          x|-->-1/2*x^2 + x + 1/2 on (0, 2],
          x|-->1/2*x^2 - 3*x + 9/2 on (2, 3]; x)
```

critical_points(*parameters*, *variable*)

Return the critical points of this piecewise function.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^0
sage: f2 = 10*x - x^2
sage: f3 = 3*x^4 - 156*x^3 + 3036*x^2 - 26208*x
sage: f = piecewise([[(0,3),f1],[(3,10),f2],[(10,20),f3]])
sage: expected = [5, 12, 13, 14]
sage: all(abs(e-a) < 0.001 for e,a in zip(expected, f.critical_points()))
True
```

domain(*parameters*, *variable*)

Return the domain

OUTPUT:

The union of the domains of the individual pieces as a `RealSet`.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.domain()
[0, 2)
```

domains (*parameters*, *variable*)

Return the individual domains

See also `expressions()`.

OUTPUT:

The collection of domains of the component functions as a tuple of `RealSet`.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.domains()
({0}, (0, 2))
```

end_points (*parameters*, *variable*)

Return a list of all interval endpoints for this function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = x^2-5
sage: f = piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3]])
sage: f.end_points()
[0, 1, 2, 3]
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.end_points()
[0, 2]
```

expression_at (*parameters*, *variable*, *point*)

Return the expression defining the piecewise function at `value`

INPUT:

- `point` – a real number.

OUTPUT:

The symbolic expression defining the function value at the given `point`.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.expression_at(0)
sin(x)
sage: f.expression_at(1)
cos(x)
sage: f.expression_at(2)
Traceback (most recent call last):
...
ValueError: point is not in the domain
```

expressions (*parameters*, *variable*)

Return the individual domains

See also `domains()`.

OUTPUT:

The collection of expressions of the component functions.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.expressions()
(sin(x), cos(x))
```

extension (*parameters*, *variable*, *extension*, *extension_domain=None*)

Extend the function

INPUT:

- `extension` – a symbolic expression
- `extension_domain` – a `RealSet` or `None` (default). The domain of the extension. By default, the entire complement of the current domain.

EXAMPLES:

```
sage: f = piecewise([((-1,1), x)]);  f
piecewise(x|-->x on (-1, 1); x)
sage: f(3)
Traceback (most recent call last):
...
ValueError: point 3 is not in the domain

sage: g = f.extension(0);  g
piecewise(x|-->x on (-1, 1), x|-->0 on (-oo, -1] ∪ [1, +oo); x)
sage: g(3)
0

sage: h = f.extension(1, RealSet.unbounded_above_closed(1));  h
piecewise(x|-->x on (-1, 1), x|-->1 on [1, +oo); x)
sage: h(3)
1
```

fourier_series_cosine_coefficient (*parameters*, *variable*, *n*, *L=None*)

Return the n-th cosine coefficient of the Fourier series of the periodic function f extending the piecewise-defined function `self`.

Given an integer $n \geq 0$, the n-th cosine coefficient of the Fourier series of f is defined by

$$a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos\left(\frac{n\pi x}{L}\right) dx,$$

where L is the half-period of f. For $n \geq 1$, a_n is the coefficient of $\cos(n\pi x/L)$ in the Fourier series of f, while a_0 is twice the coefficient of the constant term $\cos(0x)$, i.e. twice the mean value of f over one period (cf. `fourier_series_partial_sum()`).

INPUT:

- `n` – a non-negative integer
- `L` – (default: `None`) the half-period of f; if none is provided, L is assumed to be the half-width of the domain of `self`

OUTPUT:

- the Fourier coefficient a_n, as defined above

EXAMPLES:

A triangle wave function of period 2:

```
sage: f = piecewise([((0,1), x), ((1,2), 2-x)])
sage: f.fourier_series_cosine_coefficient(0)
1
sage: f.fourier_series_cosine_coefficient(3)
-4/9/pi^2
```

If the domain of the piecewise-defined function encompasses more than one period, the half-period must be passed as the second argument; for instance:

```
sage: f2 = piecewise([((0,1), x), ((1,2), 2 - x),
....:                 ((2,3), x - 2), ((3,4), 2 - (x-2))])
sage: bool(f2.restriction((0,2)) == f)  # f2 extends f on (0,4)
True
sage: f2.fourier_series_cosine_coefficient(3, 1)  # half-period = 1
-4/9/pi^2
```

The default half-period is 2 and one has:

```
sage: f2.fourier_series_cosine_coefficient(3)     # half-period = 2
0
```

The Fourier coefficient $-4/(9\pi^2)$ obtained above is actually recovered for $n = 6$:

```
sage: f2.fourier_series_cosine_coefficient(6)
-4/9/pi^2
```

Other examples:

```
sage: f(x) = x^2
sage: f = piecewise([[(-1,1),f]])
sage: f.fourier_series_cosine_coefficient(2)
pi^(-2)
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_cosine_coefficient(5,pi)
-3/5/pi
```

fourier_series_partial_sum(*parameters*, *variable*, *N*, *L=None*)

Returns the partial sum up to a given order of the Fourier series of the periodic function f extending the piecewise-defined function `self`.

The Fourier partial sum of order N is defined as

$$S_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left[a_n \cos\left(\frac{n\pi x}{L}\right) + b_n \sin\left(\frac{n\pi x}{L}\right) \right],$$

where L is the half-period of f and the a_n's and b_n's are respectively the cosine coefficients and sine coefficients of the Fourier series of f (cf. `fourier_series_cosine_coefficient()` and `fourier_series_sine_coefficient()`).

INPUT:

- N – a positive integer; the order of the partial sum
- L – (default: None) the half-period of f; if none is provided, L is assumed to be the half-width of the domain of self

OUTPUT:

- the partial sum $S_N(x)$, as a symbolic expression

EXAMPLES:

A square wave function of period 2:

```
sage: f = piecewise([((-1,0), -1), ((0,1), 1)])
sage: f.fourier_series_partial_sum(5)
4/5*sin(5*pi*x)/pi + 4/3*sin(3*pi*x)/pi + 4*sin(pi*x)/pi
```

If the domain of the piecewise-defined function encompasses more than one period, the half-period must be passed as the second argument; for instance:

```
sage: f2 = piecewise([((-1,0), -1), ((0,1), 1),
....:                 ((1,2), -1), ((2,3), 1)])
sage: bool(f2.restriction((-1,1)) == f)  # f2 extends f on (-1,3)
True
sage: f2.fourier_series_partial_sum(5, 1)  # half-period = 1
4/5*sin(5*pi*x)/pi + 4/3*sin(3*pi*x)/pi + 4*sin(pi*x)/pi
sage: bool(f2.fourier_series_partial_sum(5, 1) ==
....:      f.fourier_series_partial_sum(5))
True
```

The default half-period is 2, so that skipping the second argument yields a different result:

```
sage: f2.fourier_series_partial_sum(5)  # half-period = 2
4*sin(pi*x)/pi
```

An example of partial sum involving both cosine and sine terms:

```
sage: f = piecewise([((-1,0), 0), ((0,1/2), 2*x),
....:               ((1/2,1), 2*(1-x))])
sage: f.fourier_series_partial_sum(5)
-2*cos(2*pi*x)/pi^2 + 4/25*sin(5*pi*x)/pi^2
 - 4/9*sin(3*pi*x)/pi^2 + 4*sin(pi*x)/pi^2 + 1/4
```

fourier_series_sine_coefficient(*parameters*, *variable*, *n*, *L=None*)

Return the n-th sine coefficient of the Fourier series of the periodic function f extending the piecewise-defined function self.

Given an integer $n \geq 0$, the n-th sine coefficient of the Fourier series of f is defined by

$$b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin\left(\frac{n\pi x}{L}\right) dx,$$

where L is the half-period of f. The number b_n is the coefficient of $\sin(n\pi x/L)$ in the Fourier series of f (cf. *fourier_series_partial_sum()*).

INPUT:

- n – a non-negative integer
- L – (default: None) the half-period of f; if none is provided, L is assumed to be the half-width of the domain of self

OUTPUT:

- the Fourier coefficient b_n, as defined above

EXAMPLES:

A square wave function of period 2:

```
sage: f = piecewise([((-1,0), -1), ((0,1), 1)])
sage: f.fourier_series_sine_coefficient(1)
4/pi
sage: f.fourier_series_sine_coefficient(2)
0
sage: f.fourier_series_sine_coefficient(3)
4/3/pi
```

If the domain of the piecewise-defined function encompasses more than one period, the half-period must be passed as the second argument; for instance:

```
sage: f2 = piecewise([((-1,0), -1), ((0,1), 1),
....:                 ((1,2), -1), ((2,3), 1)])
sage: bool(f2.restriction((-1,1)) == f)  # f2 extends f on (-1,3)
True
sage: f2.fourier_series_sine_coefficient(1, 1)  # half-period = 1
4/pi
sage: f2.fourier_series_sine_coefficient(3, 1)  # half-period = 1
4/3/pi
```

The default half-period is 2 and one has:

```
sage: f2.fourier_series_sine_coefficient(1)  # half-period = 2
0
sage: f2.fourier_series_sine_coefficient(3)  # half-period = 2
0
```

The Fourier coefficients obtained from `f` are actually recovered for $n = 2$ and $n = 6$ respectively:

```
sage: f2.fourier_series_sine_coefficient(2)
4/pi
sage: f2.fourier_series_sine_coefficient(6)
4/3/pi
```

integral(*parameters*, *variable*, *x=None*, *a=None*, *b=None*, *definite=False*, ***kwds*)

By default, return the indefinite integral of the function.

If definite=True is given, returns the definite integral.

AUTHOR:

- Paul Butler

EXAMPLES:

```
sage: f1(x) = 1-x
sage: f = piecewise([((0,1),1), ((1,2),f1)])
sage: f.integral(definite=True)
1/2
```

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = piecewise([((0,pi/2),f1), ((pi/2,pi),f2)])
sage: f.integral(definite=True)
1/2*pi
```

(continues on next page)

(continued from previous page)

```
sage: f1(x) = 2
sage: f2(x) = 3 - x
sage: f = piecewise([[(-2, 0), f1], [(0, 3), f2]])
sage: f.integral()
piecewise(x|-->2*x + 4 on (-2, 0),
          x|-->-1/2*x^2 + 3*x + 4 on (0, 3); x)

sage: f1(y) = -1
sage: f2(y) = y + 3
sage: f3(y) = -y - 1
sage: f4(y) = y^2 - 1
sage: f5(y) = 3
sage: f = piecewise([[[-4,-3],f1], [(-3,-2),f2], [[-2,0],f3],
....:                 [(0,2),f4], [[2,3],f5]])
sage: F = f.integral(y); F
piecewise(y|-->-y - 4 on [-4, -3],
          y|-->1/2*y^2 + 3*y + 7/2 on (-3, -2),
          y|-->-1/2*y^2 - y - 1/2 on [-2, 0],
          y|-->1/3*y^3 - y - 1/2 on (0, 2),
          y|-->3*y - 35/6 on [2, 3]; y)
```

Ensure results are consistent with FTC:

```
sage: F(-3) - F(-4)
-1
sage: F(-1) - F(-3)
1
sage: F(2) - F(0)
2/3
sage: f.integral(y, 0, 2)
2/3
sage: F(3) - F(-4)
19/6
sage: f.integral(y, -4, 3)
19/6
sage: f.integral(definite=True)
19/6
```

```
sage: f1(y) = (y+3)^2
sage: f2(y) = y+3
sage: f3(y) = 3
sage: f = piecewise([[(-infinity, -3), f1], [(-3, 0), f2], [(0, infinity),
↪ f3]])
sage: f.integral()
piecewise(y|-->1/3*y^3 + 3*y^2 + 9*y + 9 on (-oo, -3),
          y|-->1/2*y^2 + 3*y + 9/2 on (-3, 0),
          y|-->3*y + 9/2 on (0, +oo); y)
```

```
sage: f1(x) = e^(-abs(x))
sage: f = piecewise([[(-infinity, infinity), f1]])
sage: result = f.integral(definite=True)
...
sage: result
2
sage: f.integral()
piecewise(x|-->-integrate(e^(-abs(x)), x, x, +Infinity) on (-oo, +oo); x)
```

```
sage: f = piecewise([((0, 5), cos(x))])
sage: f.integral()
piecewise(x|-->sin(x) on (0, 5); x)
```

items (*parameters*, *variable*)

Iterate over the pieces of the piecewise function

Note: You should probably use *pieces()* instead, which offers a nicer interface.

OUTPUT:

This method iterates over pieces of the piecewise function, each represented by a pair. The first element is the support, and the second the function over that support.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))])
sage: for support, function in f.items():
....:     print('support is {0}, function is {1}'.format(support,␣
↪function))
support is {0}, function is sin(x)
support is (0, 2), function is cos(x)
```

laplace (*parameters*, *variable*, *x='x'*, *s='t'*)

Returns the Laplace transform of self with respect to the variable var.

INPUT:

- `x` - variable of self
- `s` - variable of Laplace transform.

We assume that a piecewise function is 0 outside of its domain and that the left-most endpoint of the domain is 0.

EXAMPLES:

```
sage: x, s, w = var('x, s, w')
sage: f = piecewise([[(0,1),1], [[1,2], 1 - x]])
sage: f.laplace(x, s)
-e^(-s)/s + (s + 1)*e^(-2*s)/s^2 + 1/s - e^(-s)/s^2
sage: f.laplace(x, w)
-e^(-w)/w + (w + 1)*e^(-2*w)/w^2 + 1/w - e^(-w)/w^2
```

```
sage: y, t = var('y, t')
sage: f = piecewise([[[1,2], 1 - y]])
sage: f.laplace(y, t)
(t + 1)*e^(-2*t)/t^2 - e^(-t)/t^2
```

```
sage: s = var('s')
sage: t = var('t')
sage: f1(t) = -t
sage: f2(t) = 2
sage: f = piecewise([[[0,1],f1], [(1,infinity),f2]])
sage: f.laplace(t,s)
(s + 1)*e^(-s)/s^2 + 2*e^(-s)/s - 1/s^2
```

pieces (*parameters*, *variable*)

Return the "pieces".

OUTPUT:

A tuple of piecewise functions, each having only a single expression.

EXAMPLES:

```
sage: p = piecewise([((-1, 0), -x), ([0, 1], x)], var=x)
sage: p.pieces()
(piecewise(x|-->-x on (-1, 0); x),
 piecewise(x|-->x on [0, 1]; x))
```

piecewise_add (*parameters*, *variable*, *other*)

Return a new piecewise function with domain the union of the original domains and functions summed. Undefined intervals in the union domain get function value 0.

EXAMPLES:

```
sage: f = piecewise([([0,1], 1), ((2,3), x)])
sage: g = piecewise([((1/2, 2), x)])
sage: f.piecewise_add(g).unextend_zero()
piecewise(x|-->1 on (0, 1/2], x|-->x + 1 on (1/2, 1], x|-->x on (1, 2) ∪
↪(2, 3); x)
```

restriction (*parameters*, *variable*, *restricted_domain*)

Restrict the domain

INPUT:

- `restricted_domain` – a `RealSet` or something that defines one.

OUTPUT:

A new piecewise function obtained by restricting the domain.

EXAMPLES:

```
sage: f = piecewise([((-oo, oo), x)]);  f
piecewise(x|-->x on (-oo, +oo); x)
sage: f.restriction([[-1,1], [3,3]])
piecewise(x|-->x on [-1, 1] ∪ {3}; x)
```

trapezoid (*parameters*, *variable*, *N*)

Return the piecewise line function defined by the trapezoid rule for numerical integration based on a subdivision of each domain interval into N subintervals.

EXAMPLES:

```
sage: f = piecewise([[[0,1], x^2], [RealSet.open_closed(1,2), 5-x^2]])
sage: f.trapezoid(2)
piecewise(x|-->1/2*x on (0, 1/2),
          x|-->3/2*x - 1/2 on (1/2, 1),
          x|-->7/2*x - 5/2 on (1, 3/2),
          x|-->-7/2*x + 8 on (3/2, 2); x)
sage: f = piecewise([[[-1,1], 1 - x^2]])
sage: f.trapezoid(4).integral(definite=True)
5/4
sage: f = piecewise([[[-1,1], 1/2 + x - x^3]])          ## example 3
sage: f.trapezoid(6).integral(definite=True)
1
```

unextend_zero (*parameters*, *variable*)

Remove zero pieces.

EXAMPLES:

```
sage: f = piecewise([((-1,1), x)]);  f
piecewise(x|-->x on (-1, 1); x)
sage: g = f.extension(0);  g
piecewise(x|-->x on (-1, 1), x|-->0 on (-oo, -1] ∪ [1, +oo); x)
sage: g(3)
0
sage: h = g.unextend_zero()
sage: bool(h == f)
True
```

which_function (*parameters*, *variable*, *point*)

Return the expression defining the piecewise function at `value`

INPUT:

- `point` – a real number.

OUTPUT:

The symbolic expression defining the function value at the given `point`.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: f.expression_at(0)
sin(x)
sage: f.expression_at(1)
cos(x)
sage: f.expression_at(2)
Traceback (most recent call last):
...
ValueError: point is not in the domain
```

static in_operands (*ex*)

Return whether a symbolic expression contains a piecewise function as operand

INPUT:

- `ex` – a symbolic expression.

OUTPUT:

Boolean

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))]);  f
piecewise(x|-->sin(x) on {0}, x|-->cos(x) on (0, 2); x)
sage: piecewise.in_operands(f)
True
sage: piecewise.in_operands(1+sin(f))
True
sage: piecewise.in_operands(1+sin(0*f))
False
```

static simplify(*ex*)

Combine piecewise operands into single piecewise function

OUTPUT:

A piecewise function whose operands are not piecewiese if possible, that is, as long as the piecewise variable is the same.

EXAMPLES:

```
sage: f = piecewise([([0,0], sin(x)), ((0,2), cos(x))])
sage: piecewise.simplify(f)
Traceback (most recent call last):
...
NotImplementedError
```

1.7 Spike functions

AUTHORS:

- William Stein (2007-07): initial version
- Karl-Dieter Crisman (2009-09): adding documentation and doctests

class sage.functions.spike_function.**SpikeFunction**(*v*, *eps=1e-07*)

Bases: `object`

Base class for spike functions.

INPUT:

- `v` – list of pairs (x, height)
- `eps` – parameter that determines approximation to a true spike

OUTPUT:

a function with spikes at each point `x` in `v` with the given height.

EXAMPLES:

```
sage: spike_function([(-3,4), (-1,1), (2,3)], 0.001)
A spike function with spikes at [-3.0, -1.0, 2.0]
```

Putting the spikes too close together may delete some:

```
sage: spike_function([(1,1), (1.01,4)], 0.1)
Some overlapping spikes have been deleted.
You might want to use a smaller value for eps.
A spike function with spikes at [1.0]
```

Note this should normally be used indirectly via `spike_function`, but one can use it directly:

```
sage: from sage.functions.spike_function import SpikeFunction
sage: S = SpikeFunction([(0,1), (1,2), (pi,-5)]); S                    #␣
↪needs sage.symbolic
A spike function with spikes at [0.0, 1.0, 3.141592653589793]
sage: S.support                                                         #␣
↪needs sage.symbolic
[0.0, 1.0, 3.141592653589793]
```

plot (*xmin=None*, *xmax=None*, ***kwds*)

Special fast plot method for spike functions.

EXAMPLES:

```
sage: S = spike_function([(-1,1), (1,40)])
sage: P = plot(S)                                                    #␣
↪needs sage.plot
sage: P[0]                                                           #␣
↪needs sage.plot
Line defined by 8 points
```

plot_fft_abs (*samples=4096*, *xmin=None*, *xmax=None*, ***kwds*)

Plot of (absolute values of) Fast Fourier Transform of the spike function with given number of samples.

EXAMPLES:

```
sage: S = spike_function([(-3,4), (-1,1), (2,3)]); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: P = S.plot_fft_abs(8)                                          #␣
↪needs sage.plot
sage: p = P[0]; p.ydata   # abs tol 1e-8                             #␣
↪needs sage.plot
[5.0, 5.0, 3.367958691924177, 3.367958691924177, 4.123105625617661,
 4.123105625617661, 4.759921664218055, 4.759921664218055]
```

plot_fft_arg (*samples=4096*, *xmin=None*, *xmax=None*, ***kwds*)

Plot of (absolute values of) Fast Fourier Transform of the spike function with given number of samples.

EXAMPLES:

```
sage: S = spike_function([(-3,4), (-1,1), (2,3)]); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: P = S.plot_fft_arg(8)                                          #␣
↪needs sage.plot
sage: p = P[0]; p.ydata   # abs tol 1e-8                             #␣
↪needs sage.plot
[0.0, 0.0, -0.211524990023434, -0.211524990023434,
 0.244978663126864, 0.244978663126864, -0.149106180027477,
 -0.149106180027477]
```

vector (*samples=65536*, *xmin=None*, *xmax=None*)

Create a sampling vector of the spike function in question.

EXAMPLES:

```
sage: S = spike_function([(-3,4), (-1,1), (2,3)],0.001); S
A spike function with spikes at [-3.0, -1.0, 2.0]
sage: S.vector(16)                                                   #␣
↪needs sage.modules
(4.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0)
```

sage.functions.spike_function.**spike_function**

alias of *SpikeFunction*

1.8 Orthogonal polynomials

1.8.1 Chebyshev polynomials

The Chebyshev polynomial of the first kind arises as a solution to the differential equation

$$(1-x^2)\,y'' - x\,y' + n^2\,y = 0$$

and those of the second kind as a solution to

$$(1-x^2)\,y'' - 3x\,y' + n(n+2)\,y = 0.$$

The Chebyshev polynomials of the first kind are defined by the recurrence relation

$$T_0(x) = 1, \qquad T_1(x) = x, \qquad T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

The Chebyshev polynomials of the second kind are defined by the recurrence relation

$$U_0(x) = 1, \qquad U_1(x) = 2x, \qquad U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x).$$

For integers m, n, they satisfy the orthogonality relations

$$\int_{-1}^{1} T_n(x)T_m(x)\,\frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & \text{if } n \neq m, \\ \pi & \text{if } n = m = 0, \\ \pi/2 & \text{if } n = m \neq 0, \end{cases}$$

and

$$\int_{-1}^{1} U_n(x)U_m(x)\sqrt{1-x^2}\,dx = \frac{\pi}{2}\delta_{m,n}.$$

They are named after Pafnuty Chebyshev (1821-1894, alternative transliterations: Tchebyshef or Tschebyscheff).

1.8.2 Hermite polynomials

The *Hermite polynomials* are defined either by

$$H_n(x) = (-1)^n e^{x^2/2}\frac{d^n}{dx^n}e^{-x^2/2}$$

(the “probabilists’ Hermite polynomials”), or by

$$H_n(x) = (-1)^n e^{x^2}\frac{d^n}{dx^n}e^{-x^2}$$

(the “physicists’ Hermite polynomials”). Sage (via Maxima) implements the latter flavor. These satisfy the orthogonality relation

$$\int_{-\infty}^{\infty} H_n(x)H_m(x)\,e^{-x^2}\,dx = \sqrt{\pi}n!2^n\delta_{nm}.$$

They are named in honor of Charles Hermite (1822-1901), but were first introduced by Laplace in 1810 and also studied by Chebyshev in 1859.

1.8.3 Legendre polynomials

Each *Legendre polynomial* $P_n(x)$ is an n-th degree polynomial. It may be expressed using Rodrigues' formula:

$$P_n(x) = (2^n n!)^{-1} \frac{d^n}{dx^n} \left[(x^2-1)^n\right].$$

These are solutions to Legendre's differential equation:

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P(x)\right] + n(n+1)P(x) = 0$$

and satisfy the orthogonality relation

$$\int_{-1}^{1} P_m(x)P_n(x)\,dx = \frac{2}{2n+1}\delta_{mn}.$$

The *Legendre function of the second kind* $Q_n(x)$ is another (linearly independent) solution to the Legendre differential equation. It is not an "orthogonal polynomial" however.

The associated Legendre functions of the first kind $P_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$\begin{aligned} P_\ell^m(x) &= (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}P_\ell(x) \\ &= \frac{(-1)^m}{2^\ell \ell!}(1-x^2)^{m/2}\frac{d^{\ell+m}}{dx^{\ell+m}}(x^2-1)^\ell. \end{aligned}$$

Assuming $0 \le m \le \ell$, they satisfy the orthogonality relation:

$$\int_{-1}^{1} P_k^{(m)} P_\ell^{(m)} dx = \frac{2(\ell+m)!}{(2\ell+1)(\ell-m)!}\,\delta_{k,\ell},$$

where $\delta_{k,\ell}$ is the Kronecker delta.

The associated Legendre functions of the second kind $Q_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$Q_\ell^m(x) = (-1)^m(1-x^2)^{m/2}\frac{d^m}{dx^m}Q_\ell(x).$$

They are named after Adrien-Marie Legendre (1752-1833).

1.8.4 Laguerre polynomials

Laguerre polynomials may be defined by the Rodrigues formula

$$L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^n\right).$$

They are solutions of Laguerre's equation:

$$x\,y'' + (1-x)\,y' + n\,y = 0$$

and satisfy the orthogonality relation

$$\int_0^\infty L_m(x)L_n(x)e^{-x}\,dx = \delta_{mn}.$$

The generalized Laguerre polynomials may be defined by the Rodrigues formula:

$$L_n^{(\alpha)}(x) = \frac{x^{-\alpha}e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^{n+\alpha}\right).$$

(These are also sometimes called the associated Laguerre polynomials.) The simple Laguerre polynomials are recovered from the generalized polynomials by setting $\alpha = 0$.

They are named after Edmond Laguerre (1834-1886).

1.8.5 Jacobi polynomials

Jacobi polynomials are a class of orthogonal polynomials. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$P_n^{(\alpha,\beta)}(z) = \frac{(\alpha+1)_n}{n!}\,{}_2F_1\left(-n, 1+\alpha+\beta+n; \alpha+1; \frac{1-z}{2}\right),$$

where $()_n$ is Pochhammer's symbol (for the rising factorial), (Abramowitz and Stegun p561.) and thus have the explicit expression

$$P_n^{(\alpha,\beta)}(z) = \frac{\Gamma(\alpha+n+1)}{n!\Gamma(\alpha+\beta+n+1)}\sum_{m=0}^{n}\binom{n}{m}\frac{\Gamma(\alpha+\beta+n+m+1)}{\Gamma(\alpha+m+1)}\left(\frac{z-1}{2}\right)^m.$$

They are named after Carl Gustav Jaboc Jacobi (1804-1851).

1.8.6 Gegenbauer polynomials

Ultraspherical or *Gegenbauer polynomials* are given in terms of the Jacobi polynomials $P_n^{(\alpha,\beta)}(x)$ with $\alpha = \beta = a - 1/2$ by

$$C_n^{(a)}(x) = \frac{\Gamma(a+1/2)}{\Gamma(2a)}\frac{\Gamma(n+2a)}{\Gamma(n+a+1/2)}P_n^{(a-1/2,a-1/2)}(x).$$

They satisfy the orthogonality relation

$$\int_{-1}^{1}(1-x^2)^{a-1/2}C_m^{(a)}(x)C_n^{(a)}(x)\,dx = \delta_{mn}2^{1-2a}\pi\frac{\Gamma(n+2a)}{(n+a)\Gamma^2(a)\Gamma(n+1)},$$

for $a > -1/2$. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$C_n^{(a)}(z) = \frac{(2a)^{\underline{n}}}{n!}\,{}_2F_1\left(-n, 2a+n; a+\frac{1}{2}; \frac{1-z}{2}\right)$$

where $\underline{n}$ is the falling factorial. (See Abramowitz and Stegun p561.)

They are named for Leopold Gegenbauer (1849-1903).

1.8.7 Krawtchouk polynomials

The *Krawtchouk polynomials* are discrete orthogonal polynomials that are given by the hypergeometric series

$$K_j(x;n,p) = (-1)^j\binom{n}{j}p^j\,{}_2F_1\left(-j,-x;-n;p^{-1}\right).$$

Since they are discrete orthogonal polynomials, they satisfy an orthogonality relation defined on a discrete (in this case finite) set of points:

$$\sum_{m=0}^{n}K_i(m;n,p)K_j(m;n,p)\binom{n}{m}p^m q^{n-m} = \binom{n}{j}(pq)^j\delta_{ij},$$

where $q = 1 - p$. They can also be described by the recurrence relation

$$jK_j(x;n,p) = (x-(n-j+1)p-(j-1)q)K_{j-1}(x;n,p) - pq(n-j+2)K_{j-2}(x;n,p),$$

where $K_0(x;n,p) = 1$ and $K_1(x;n,p) = x - np$.

They are named for Mykhailo Krawtchouk (1892-1942).

1.8.8 Meixner polynomials

The *Meixner polynomials* are discrete orthogonal polynomials that are given by the hypergeometric series

$$M_n(x;n,p) = (-1)^j \binom{n}{j} p^j \, {}_2F_1\left(-j,-x;-n;p^{-1}\right).$$

They satisfy an orthogonality relation:

$$\sum_{k=0}^{\infty} \tilde{M}_n(k;b,c)\tilde{M}_m(k;b,c)\,\frac{(b)_k}{k!}c^k = \frac{c^{-n}n!}{(b)_n(1-c)^b}\delta_{mn},$$

where $\tilde{M}_n(x;b,c) = M_n(x;b,c)/(b)_x$, for $b > 0`and`0 < c < 1$. They can also be described by the recurrence relation

$$\begin{aligned} c(n-1+b)M_n(x;b,c) &= ((c-1)x+n-1+c(n-1+b))(b+n-1)M_{n-1}(x;b,c) \\ &\quad - (b+n-1)(b+n-2)(n-1)M_{n-2}(x;b,c), \end{aligned}$$

where $M_0(x;b,c) = 0$ and $M_1(x;b,c) = (1-c^{-1})x+b$.

They are named for Josef Meixner (1908-1994).

1.8.9 Hahn polynomials

The *Hahn polynomials* are discrete orthogonal polynomials that are given by the hypergeometric series

$$Q_k(x;a,b,n) = {}_3F_2\left(-k,k+a+b+1,-x;a+1,-n;1\right).$$

They satisfy an orthogonality relation:

$$\sum_{k=0}^{n-1} Q_i(k;a,b,n)Q_j(k;a,b,n)\,\rho(k) = \frac{\delta_{ij}}{\pi_i},$$

where

$$\begin{aligned} \rho(k) &= \binom{a+k}{k}\binom{b+n-k}{n-k}, \\ \pi_i &= \delta_{ij}\frac{(-1)^i i!(b+1)_i(i+a+b+1)_{n+1}}{n!(2i+a+b+1)(-n)_i(a+1)_i}. \end{aligned}$$

They can also be described by the recurrence relation

$$AQ_k(x;a,b,n) = (-x+A+C)Q_{k-1}(x;a,b,n) - CQ_{k-2}(x;a,b,n),$$

where $Q_0(x;a,b,n) = 1$ and $Q_1(x;a,b,n) = 1 - \frac{a+b+2}{(a+1)n}x$ and

$$A = \frac{(k+a+b)(k+a)(n-k+1)}{(2k+a+b-1)(2k+a+b)}, \qquad C = \frac{(k-1)(k+b-1)(k+a+b+n)}{(2k+a+b-2)(2k+a+b-1)}.$$

They are named for Wolfgang Hahn (1911-1998), although they were first introduced by Chebyshev in 1875.

1.8.10 Pochhammer symbol

For completeness, the *Pochhammer symbol*, introduced by Leo August Pochhammer, $(x)_n$, is used in the theory of special functions to represent the "rising factorial" or "upper factorial"

$$(x)_n = x(x+1)(x+2)\cdots(x+n-1) = \frac{(x+n-1)!}{(x-1)!}.$$

On the other hand, the *falling factorial* or *lower factorial* is

$$x^{\underline{n}} = \frac{x!}{(x-n)!},$$

in the notation of Ronald L. Graham, Donald E. Knuth and Oren Patashnik in their book Concrete Mathematics.

Todo: Implement Zernike polynomials. Wikipedia article Zernike_polynomials

---

REFERENCES:

- [AS1964]
- Wikipedia article Chebyshev_polynomials
- Wikipedia article Legendre_polynomials
- Wikipedia article Hermite_polynomials
- http://mathworld.wolfram.com/GegenbauerPolynomial.html
- Wikipedia article Jacobi_polynomials
- Wikipedia article Laguerre_polynomia
- Wikipedia article Associated_Legendre_polynomials
- Wikipedia article Kravchuk_polynomials
- Wikipedia article Meixner_polynomials
- Wikipedia article Hahn_polynomials
- Roelof Koekeok and René F. Swarttouw, arXiv math/9602214
- [Koe1999]

AUTHORS:

- David Joyner (2006-06)
- Stefan Reiterer (2010-)
- Ralf Stephan (2015-)

The original module wrapped some of the orthogonal/special functions in the Maxima package "orthopoly" and was written by Barton Willis of the University of Nebraska at Kearney.

class `sage.functions.orthogonal_polys.`**`ChebyshevFunction`**(*name*, *nargs=2*, *latex_name=None*, *conversions=None*)

Bases: `OrthogonalFunction`

Abstract base class for Chebyshev polynomials of the first and second kind.

EXAMPLES:

```
sage: chebyshev_T(3, x)                                                    #␣
↪needs sage.symbolic
4*x^3 - 3*x
```

class sage.functions.orthogonal_polys.**Func_assoc_legendre_P**

Bases: BuiltinFunction

Return the Ferrers function $\mathtt{P}_n^m(x)$ of first kind for $x \in (-1,1)$ with general order m and general degree n.

Ferrers functions of first kind are one of two linearly independent solutions of the associated Legendre differential equation

$$(1-x^2)\frac{\mathrm{d}^2 w}{\mathrm{d}x^2} - 2x\frac{\mathrm{d}w}{\mathrm{d}x} + \left(n(n+1) - \frac{m^2}{1-x^2}\right) w = 0$$

on the interval $x \in (-1,1)$ and are usually denoted by $\mathtt{P}_n^m(x)$.

See also:

The other linearly independent solution is called *Ferrers function of second kind* and denoted by $\mathtt{Q}_n^m(x)$, see *Func_assoc_legendre_Q*.

> **Warning:** Ferrers functions must be carefully distinguished from associated Legendre functions which are defined on $\mathbf{C} \setminus (-\infty, 1]$ and have not yet been implemented.

EXAMPLES:

We give the first Ferrers functions for non-negative integers n and m in the interval $-1 < x < 1$:

```
sage: for n in range(4):                                                   #␣
↪needs sage.symbolic
....:     for m in range(n+1):
....:         print(f"P_{n}^{m}({x}) = {gen_legendre_P(n, m, x)}")
P_0^0(x) = 1
P_1^0(x) = x
P_1^1(x) = -sqrt(-x^2 + 1)
P_2^0(x) = 3/2*x^2 - 1/2
P_2^1(x) = -3*sqrt(-x^2 + 1)*x
P_2^2(x) = -3*x^2 + 3
P_3^0(x) = 5/2*x^3 - 3/2*x
P_3^1(x) = -3/2*(5*x^2 - 1)*sqrt(-x^2 + 1)
P_3^2(x) = -15*(x^2 - 1)*x
P_3^3(x) = -15*(-x^2 + 1)^(3/2)
```

These expressions for non-negative integers are computed by the Rodrigues-type given in *eval_gen_poly()*. Negative values for n are obtained by the following identity:

$$P_{-n}^m(x) = P_{n-1}^m(x).$$

For n being a non-negative integer, negative values for m are obtained by

$$P_n^{-|m|}(x) = (-1)^{|m|}\frac{(n-|m|)!}{(n+|m|)!}P_n^{|m|}(x),$$

where $|m| \leq n$.

Here are some specific values with negative integers:

```
sage: # needs sage.symbolic
sage: gen_legendre_P(-2, -1, x)
1/2*sqrt(-x^2 + 1)
sage: gen_legendre_P(2, -2, x)
-1/8*x^2 + 1/8
sage: gen_legendre_P(3, -2, x)
-1/8*(x^2 - 1)*x
sage: gen_legendre_P(1, -2, x)
0
```

Here are some other random values with floating numbers:

```
sage: # needs sage.symbolic
sage: m = var('m'); assume(m, 'integer')
sage: gen_legendre_P(m, m, .2)
0.960000000000000^(1/2*m)*(-1)^m*factorial(2*m)/(2^m*factorial(m))
sage: gen_legendre_P(.2, m, 0)
sqrt(pi)*2^m/(gamma(-1/2*m + 1.10000000000000)*gamma(-1/2*m + 0.400000000000000))
sage: gen_legendre_P(.2, .2, .2)
0.757714892929573
```

REFERENCES:

- [DLMF-Legendre]

deprecated_function_alias(*issue_number*, *func*)

Create an aliased version of a function or a method which raises a deprecation warning message.

If f is a function or a method, write `g = deprecated_function_alias(issue_number, f)` to make a deprecated aliased version of f.

INPUT:

- `issue_number` – integer. The github issue number where the deprecation is introduced.
- `func` – the function or method to be aliased

EXAMPLES:

```
sage: from sage.misc.superseded import deprecated_function_alias
sage: g = deprecated_function_alias(13109, number_of_partitions)
↪ # needs sage.combinat sage.libs.flint
sage: g(5)
↪ # needs sage.combinat sage.libs.flint
doctest:...: DeprecationWarning: g is deprecated.
Please use sage.combinat.partition.number_of_partitions instead.
See https://github.com/sagemath/sage/issues/13109 for details.
7
```

This also works for methods:

```
sage: class cls():
....:    def new_meth(self): return 42
....:    old_meth = deprecated_function_alias(13109, new_meth)
sage: cls().old_meth()
doctest:...: DeprecationWarning: old_meth is deprecated. Please use new_meth
↪instead.
See https://github.com/sagemath/sage/issues/13109 for details.
42
```

github issue #11585:

```
sage: def a(): pass
sage: b = deprecated_function_alias(13109, a)
sage: b()
doctest:...: DeprecationWarning: b is deprecated. Please use a instead.
See https://github.com/sagemath/sage/issues/13109 for details.
```

AUTHORS:

- Florent Hivert (2009-11-23), with the help of Mike Hansen.
- Luca De Feo (2011-07-11), printing the full module path when different from old path

eval_gen_poly (*n*, *m*, *arg*, ***kwds*)

Return the Ferrers function of first kind $\mathrm{P}_n^m(x)$ for integers $n > -1, m > -1$ given by the following Rodrigues-type formula:

$$\mathrm{P}_n^m(x) = (-1)^{m+n}\frac{(1-x^2)^{m/2}}{2^n n!}\frac{\mathrm{d}^{m+n}}{\mathrm{d}x^{m+n}}(1-x^2)^n.$$

INPUT:

- `n` – an integer degree
- `m` – an integer order
- `x` – either an integer or a non-numerical symbolic expression

EXAMPLES:

```
sage: gen_legendre_P(7, 4, x)                                          #␣
↪needs sage.symbolic
3465/2*(13*x^3 - 3*x)*(x^2 - 1)^2
sage: gen_legendre_P(3, 1, sqrt(x))                                    #␣
↪needs sage.symbolic
-3/2*(5*x - 1)*sqrt(-x + 1)
```

REFERENCE:

- [DLMF-Legendre], Section 14.7 eq. 10 (https://dlmf.nist.gov/14.7#E10)

eval_poly (**args*, ***kwds*)

Deprecated: Use *eval_gen_poly()* instead. See github issue #25034 for details.

class sage.functions.orthogonal_polys.**Func_assoc_legendre_Q**

Bases: `BuiltinFunction`

EXAMPLES:

```
sage: loads(dumps(gen_legendre_Q))
gen_legendre_Q
sage: maxima(gen_legendre_Q(2, 1, 3, hold=True))._sage_().simplify_full()    #␣
↪needs sage.symbolic
1/4*sqrt(2)*(36*pi - 36*I*log(2) + 25*I)
```

eval_recursive (*n*, *m*, *x*, ***kwds*)

Return the associated Legendre Q(n, m, arg) function for integers $n > -1, m > -1$.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: gen_legendre_Q(3, 4, x)
48/(x^2 - 1)^2
sage: gen_legendre_Q(4, 5, x)
-384/((x^2 - 1)^2*sqrt(-x^2 + 1))
sage: gen_legendre_Q(0, 1, x)
-1/sqrt(-x^2 + 1)
sage: gen_legendre_Q(0, 2, x)
-1/2*((x + 1)^2 - (x - 1)^2)/(x^2 - 1)
sage: gen_legendre_Q(2, 2, x).subs(x=2).expand()
9/2*I*pi - 9/2*log(3) + 14/3
```

class sage.functions.orthogonal_polys.**Func_chebyshev_T**

Bases: *ChebyshevFunction*

Chebyshev polynomials of the first kind.

REFERENCE:

- [AS1964] 22.5.31 page 778 and 6.1.22 page 256.

EXAMPLES:

```
sage: chebyshev_T(5, x)
↪# needs sage.symbolic
16*x^5 - 20*x^3 + 5*x
sage: var('k')
↪# needs sage.symbolic
k
sage: test = chebyshev_T(k, x); test
↪# needs sage.symbolic
chebyshev_T(k, x)
```

eval_algebraic(*n*, *x*)

Evaluate chebyshev_T as polynomial, using a recursive formula.

INPUT:

- n – an integer
- x – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_T.eval_algebraic(5, x)                                    #␣
↪needs sage.symbolic
2*(2*(2*x^2 - 1)*x - x)*(2*x^2 - 1) - x
sage: chebyshev_T(-7, x) - chebyshev_T(7, x)                              #␣
↪needs sage.symbolic
0
sage: R.<t> = ZZ[]
sage: chebyshev_T.eval_algebraic(-1, t)
t
sage: chebyshev_T.eval_algebraic(0, t)
1
sage: chebyshev_T.eval_algebraic(1, t)
t
sage: chebyshev_T(7^100, 1/2)
1/2
```

(continues on next page)

(continued from previous page)

```
sage: chebyshev_T(7^100, Mod(2,3))
2
sage: n = 97; x = RIF(pi/2/n)                                            #␣
↪needs sage.symbolic
sage: chebyshev_T(n, cos(x)).contains_zero()                            #␣
↪needs sage.symbolic
True

sage: # needs sage.rings.padics
sage: R.<t> = Zp(2, 8, 'capped-abs')[]
sage: chebyshev_T(10^6 + 1, t)
(2^7 + O(2^8))*t^5 + O(2^8)*t^4 + (2^6 + O(2^8))*t^3 + O(2^8)*t^2
 + (1 + 2^6 + O(2^8))*t + O(2^8)
```

eval_formula (n, x)

Evaluate `chebyshev_T` using an explicit formula. See [AS1964] 227 (p. 782) for details for the recursions. See also [Koe1999] for fast evaluation techniques.

INPUT:

- `n` – an integer
- `x` – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: chebyshev_T.eval_formula(-1, x)
x
sage: chebyshev_T.eval_formula(0, x)
1
sage: chebyshev_T.eval_formula(1, x)
x
sage: chebyshev_T.eval_formula(10, x)
512*x^10 - 1280*x^8 + 1120*x^6 - 400*x^4 + 50*x^2 - 1
sage: chebyshev_T.eval_algebraic(10, x).expand()
512*x^10 - 1280*x^8 + 1120*x^6 - 400*x^4 + 50*x^2 - 1

sage: chebyshev_T.eval_formula(2, 0.1) == chebyshev_T._evalf_(2, 0.1)        #␣
↪needs sage.rings.complex_double
True
```

class sage.functions.orthogonal_polys.**Func_chebyshev_U**

Bases: *ChebyshevFunction*

Class for the Chebyshev polynomial of the second kind.

REFERENCE:

- [AS1964] 22.8.3 page 783 and 6.1.22 page 256.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: chebyshev_U(2,t)
4*t^2 - 1
sage: chebyshev_U(3,t)
8*t^3 - 4*t
```

eval_algebraic(*n*, *x*)

Evaluate `chebyshev_U` as polynomial, using a recursive formula.

INPUT:

- `n` – an integer
- `x` – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: chebyshev_U.eval_algebraic(5, x)                                    #␣
↪needs sage.symbolic
-2*((2*x + 1)*(2*x - 1)*x - 4*(2*x^2 - 1)*x)*(2*x + 1)*(2*x - 1)
sage: parent(chebyshev_U(3, Mod(8,9)))
Ring of integers modulo 9
sage: parent(chebyshev_U(3, Mod(1,9)))
Ring of integers modulo 9
sage: chebyshev_U(-3, x) + chebyshev_U(1, x)                              #␣
↪needs sage.symbolic
0
sage: chebyshev_U(-1,Mod(5,8))
0
sage: parent(chebyshev_U(-1,Mod(5,8)))
Ring of integers modulo 8
sage: R.<t> = ZZ[]
sage: chebyshev_U.eval_algebraic(-2, t)
-1
sage: chebyshev_U.eval_algebraic(-1, t)
0
sage: chebyshev_U.eval_algebraic(0, t)
1
sage: chebyshev_U.eval_algebraic(1, t)
2*t
sage: n = 97; x = RIF(pi/n)                                               #␣
↪needs sage.symbolic
sage: chebyshev_U(n - 1, cos(x)).contains_zero()                          #␣
↪needs sage.symbolic
True
sage: R.<t> = Zp(2, 6, 'capped-abs')[]                                    #␣
↪needs sage.rings.padics
sage: chebyshev_U(10^6 + 1, t)                                            #␣
↪needs sage.rings.padics
(2 + O(2^6))*t + O(2^6)
```

eval_formula(*n*, *x*)

Evaluate `chebyshev_U` using an explicit formula.

See [AS1964] 227 (p. 782) for details on the recursions. See also [Koe1999] for the recursion formulas.

INPUT:

- `n` – an integer
- `x` – a value to evaluate the polynomial at (this can be any ring element)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: chebyshev_U.eval_formula(10, x)
1024*x^10 - 2304*x^8 + 1792*x^6 - 560*x^4 + 60*x^2 - 1
```

(continues on next page)

(continued from previous page)

```
sage: chebyshev_U.eval_formula(-2, x)
-1
sage: chebyshev_U.eval_formula(-1, x)
0
sage: chebyshev_U.eval_formula(0, x)
1
sage: chebyshev_U.eval_formula(1, x)
2*x
sage: chebyshev_U.eval_formula(2,0.1) == chebyshev_U._evalf_(2,0.1)
True
```

class sage.functions.orthogonal_polys.**Func_gen_laguerre**

Bases: *OrthogonalFunction*

REFERENCE:

- [AS1964] 22.5.16, page 778 and page 789.

class sage.functions.orthogonal_polys.**Func_hahn**

Bases: *OrthogonalFunction*

Hahn polynomials $Q_k(x; a, b, n)$.

INPUT:

- `k` – the degree
- `x` – the independent variable x
- `a, b` – the parameters a, b
- `n` – the number of discrete points

EXAMPLES:

We verify the orthogonality for $n = 3$:

```
sage: # needs sage.symbolic
sage: n = 2
sage: a, b = SR.var('a,b')
sage: def rho(k, a, b, n):
....:     return binomial(a + k, k) * binomial(b + n - k, n - k)
sage: M = matrix([[sum(rho(k, a, b, n)
....:                  * hahn(i, k, a, b, n) * hahn(j, k, a, b, n)
....:                  for k in range(n + 1)).expand().factor()
....:              for i in range(n+1)] for j in range(n+1)])
sage: M = M.factor()
sage: P = rising_factorial
sage: def diag(i, a, b, n):
....:    return ((-1)^i * factorial(i) * P(b + 1, i) * P(i + a + b + 1, n + 1)
....:            / (factorial(n) * (2*i + a + b + 1) * P(-n, i) * P(a + 1, i)))
sage: all(M[i,i] == diag(i, a, b, n) for i in range(3))
True
sage: all(M[i,j] == 0 for i in range(3) for j in range(3) if i != j)
True
```

eval_formula (*k, x, a, b, n*)

Evaluate `self` using an explicit formula.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: k, x, a, b, n = var('k,x,a,b,n')
sage: Q2 = hahn.eval_formula(2, x, a, b, n).simplify_full()
sage: Q2.coefficient(x^2).factor()
(a + b + 4)*(a + b + 3)/((a + 2)*(a + 1)*(n - 1)*n)
sage: Q2.coefficient(x).factor()
-(2*a*n - a + b + 4*n)*(a + b + 3)/((a + 2)*(a + 1)*(n - 1)*n)
sage: Q2(x=0)
1
```

eval_recursive(*k*, *x*, *a*, *b*, *n*, **args*, ***kwds*)

Return the Hahn polynomial $Q_k(x; a, b, n)$ using the recursive formula.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, a, b, n = var('x,a,b,n')
sage: hahn.eval_recursive(0, x, a, b, n)
1
sage: hahn.eval_recursive(1, x, a, b, n)
-(a + b + 2)*x/((a + 1)*n) + 1
sage: bool(hahn(2, x, a, b, n) == hahn.eval_recursive(2, x, a, b, n))
True
sage: bool(hahn(3, x, a, b, n) == hahn.eval_recursive(3, x, a, b, n))
True
sage: bool(hahn(4, x, a, b, n) == hahn.eval_recursive(4, x, a, b, n))
True
sage: M = matrix([[-1/2, -1], [1, 0]])                                    #␣
↪needs sage.modules
sage: ret = hahn.eval_recursive(2, M, 1, 2, n).simplify_full().factor()   #␣
↪needs sage.modules
sage: ret                                                                 #␣
↪needs sage.modules
[1/4*(4*n^2 + 8*n - 19)/((n - 1)*n)          3/2*(4*n + 3)/((n - 1)*n)]
[         -3/2*(4*n + 3)/((n - 1)*n)          (n^2 - n - 7)/((n - 1)*n)]
```

class sage.functions.orthogonal_polys.**Func_hermite**

Bases: `GinacFunction`

Return the Hermite polynomial for integers $n > -1$.

REFERENCE:

- [AS1964] 22.5.40 and 22.5.41, page 779.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x = PolynomialRing(QQ, 'x').gen()
sage: hermite(2, x)
4*x^2 - 2
sage: hermite(3, x)
8*x^3 - 12*x
sage: hermite(3, 2)
40
sage: S.<y> = PolynomialRing(RR)
sage: hermite(3, y)
8.00000000000000*y^3 - 12.0000000000000*y
```

(continues on next page)

(continued from previous page)

```
sage: R.<x,y> = QQ[]
sage: hermite(3, y^2)
8*y^6 - 12*y^2
sage: w = var('w')
sage: hermite(3, 2*w)
64*w^3 - 24*w
sage: hermite(5, 3.1416)
5208.69733891963
sage: hermite(5, RealField(100)(pi))
5208.6167627118104649470287166
```

Check that github issue #17192 is fixed:

```
sage: # needs sage.symbolic
sage: x = PolynomialRing(QQ, 'x').gen()
sage: hermite(0, x)
1
sage: hermite(-1, x)
Traceback (most recent call last):
...
RuntimeError: hermite_eval: The index n must be a nonnegative integer
sage: hermite(-7, x)
Traceback (most recent call last):
...
RuntimeError: hermite_eval: The index n must be a nonnegative integer
sage: m, x = SR.var('m,x')
sage: hermite(m, x).diff(m)
Traceback (most recent call last):
...
RuntimeError: derivative w.r.t. to the index is not supported yet
```

class sage.functions.orthogonal_polys.**Func_jacobi_P**

Bases: *OrthogonalFunction*

Return the Jacobi polynomial $P_n^{(a,b)}(x)$ for integers $n > -1$ and a and b symbolic or $a > -1$ and $b > -1$.

The Jacobi polynomials are actually defined for all a and b. However, the Jacobi polynomial weight $(1-x)^a(1+x)^b$ is not integrable for $a \leq -1$ or $b \leq -1$.

REFERENCE:

- Table on page 789 in [AS1964].

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: jacobi_P(2,0,0,x)
3/2*x^2 - 1/2
sage: jacobi_P(2,1,2,1.2)                                               #␣
↪needs sage.symbolic
5.01000000000000
```

class sage.functions.orthogonal_polys.**Func_krawtchouk**

Bases: *OrthogonalFunction*

Krawtchouk polynomials $K_j(x; n, p)$.

INPUT:

- `j` – the degree
- `x` – the independent variable x
- `n` – the number of discrete points
- `p` – the parameter p

See also:

`sage.coding.delsarte_bounds.krawtchouk()` $\bar{K}^{n,q}_l(x)$, which are related by

$$(-q)^j \bar{K}^{n,q^{-1}}_j(x) = K_j(x; n, 1-q).$$

EXAMPLES:

We verify the orthogonality for $n = 4$:

```
sage: n = 4
sage: p = SR.var('p')                                                    #␣
↪needs sage.symbolic
sage: matrix([[sum(binomial(n,m) * p**m * (1-p)**(n-m)                   #␣
↪needs sage.symbolic
....:               * krawtchouk(i,m,n,p) * krawtchouk(j,m,n,p)
....:               for m in range(n+1)).expand().factor()
....:           for i in range(n+1)] for j in range(n+1)])
[               1                0                0                0              ␣
↪  0]
[               0      -4*(p - 1)*p                0                0              ␣
↪  0]
[               0                0   6*(p - 1)^2*p^2               0              ␣
↪  0]
[               0                0                0 -4*(p - 1)^3*p^3              ␣
↪  0]
[               0                0                0                0    (p - 1)^
↪4*p^4]
```

We verify the relationship between the Krawtchouk implementations:

```
sage: q = SR.var('q')                                                    #␣
↪needs sage.symbolic
sage: all(codes.bounds.krawtchouk(n, 1/q, j, x)*(-q)^j                   #␣
↪needs sage.symbolic
....:     == krawtchouk(j, x, n, 1-q) for j in range(n+1))
True
```

eval_formula (k, x, n, p)

Evaluate `self` using an explicit formula.

EXAMPLES:

```
sage: x, n, p = var('x,n,p')                                             #␣
↪needs sage.symbolic
sage: krawtchouk.eval_formula(3, x, n, p).expand().collect(x)            #␣
↪needs sage.symbolic
-1/6*n^3*p^3 + 1/2*n^2*p^3 - 1/3*n*p^3 - 1/2*(n*p - 2*p + 1)*x^2
 + 1/6*x^3 + 1/6*(3*n^2*p^2 - 9*n*p^2 + 3*n*p + 6*p^2 - 6*p + 2)*x
```

eval_recursive (*j*, *x*, *n*, *p*, **args*, ***kwds*)

Return the Krawtchouk polynomial $K_j(x; n, p)$ using the recursive formula.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, n, p = var('x,n,p')
sage: krawtchouk.eval_recursive(0, x, n, p)
1
sage: krawtchouk.eval_recursive(1, x, n, p)
-n*p + x
sage: krawtchouk.eval_recursive(2, x, n, p).collect(x)
1/2*n^2*p^2 + 1/2*n*(p - 1)*p - n*p^2 + 1/2*n*p
 - 1/2*(2*n*p - 2*p + 1)*x + 1/2*x^2
sage: bool(krawtchouk.eval_recursive(2, x, n, p) == krawtchouk(2, x, n, p))
True
sage: bool(krawtchouk.eval_recursive(3, x, n, p) == krawtchouk(3, x, n, p))
True
sage: bool(krawtchouk.eval_recursive(4, x, n, p) == krawtchouk(4, x, n, p))
True

sage: M = matrix([[-1/2, -1], [1, 0]])                                    #␣
↪needs sage.modules
sage: krawtchouk.eval_recursive(2, M, 3, 1/2)                             #␣
↪needs sage.modules
[ 9/8  7/4]
[-7/4  1/4]
```

class sage.functions.orthogonal_polys.**Func_laguerre**

Bases: *OrthogonalFunction*

REFERENCE:

- [AS1964] 22.5.16, page 778 and page 789.

class sage.functions.orthogonal_polys.**Func_legendre_P**

Bases: GinacFunction

EXAMPLES:

```
sage: # needs sage.symbolic
sage: legendre_P(4, 2.0)
55.3750000000000
sage: legendre_P(1, x)
x
sage: legendre_P(4, x + 1)
35/8*(x + 1)^4 - 15/4*(x + 1)^2 + 3/8
sage: legendre_P(1/2, I+1.)
1.05338240025858 + 0.359890322109665*I
sage: legendre_P(0, SR(1)).parent()
Symbolic Ring

sage: legendre_P(0, 0)                                                    #␣
↪needs sage.symbolic
1
sage: legendre_P(1, x)                                                    #␣
↪needs sage.symbolic
x
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.symbolic
sage: legendre_P(4, 2.)
55.3750000000000
sage: legendre_P(5.5, 1.00001)
1.00017875754114
sage: legendre_P(1/2, I + 1).n()
1.05338240025858 + 0.359890322109665*I
sage: legendre_P(1/2, I + 1).n(59)
1.0533824002585801 + 0.35989032210966539*I
sage: legendre_P(42, RR(12345678))
2.66314881466753e309
sage: legendre_P(42, Reals(20)(12345678))
2.6632e309
sage: legendre_P(201/2, 0).n()
0.0561386178630179
sage: legendre_P(201/2, 0).n(100)
0.056138617863017877699963095883

sage: # needs sage.symbolic
sage: R.<x> = QQ[]
sage: legendre_P(4, x)
35/8*x^4 - 15/4*x^2 + 3/8
sage: legendre_P(10000, x).coefficient(x, 1)
0
sage: var('t,x')
(t, x)
sage: legendre_P(-5, t)
35/8*t^4 - 15/4*t^2 + 3/8
sage: legendre_P(4, x + 1)
35/8*(x + 1)^4 - 15/4*(x + 1)^2 + 3/8
sage: legendre_P(4, sqrt(2))
83/8
sage: legendre_P(4, I*e)
35/8*e^4 + 15/4*e^2 + 3/8

sage: # needs sage.symbolic
sage: n = var('n')
sage: derivative(legendre_P(n,x), x)
(n*x*legendre_P(n, x) - n*legendre_P(n - 1, x))/(x^2 - 1)
sage: derivative(legendre_P(3,x), x)
15/2*x^2 - 3/2
sage: derivative(legendre_P(n,x), n)
Traceback (most recent call last):
...
RuntimeError: derivative w.r.t. to the index is not supported yet
```

class sage.functions.orthogonal_polys.**Func_legendre_Q**

Bases: BuiltinFunction

EXAMPLES:

```
sage: loads(dumps(legendre_Q))
legendre_Q
sage: maxima(legendre_Q(20, x, hold=True))._sage_().coefficient(x, 10)      #␣
↪needs sage.symbolic
-29113619535/131072*log(-(x + 1)/(x - 1))
```

eval_formula (*n*, *arg*, ***kwds*)

Return expanded Legendre `Q(n, arg)` function expression.

REFERENCE:

- T.M. Dunster, Legendre and Related Functions, https://dlmf.nist.gov/14.7#E2

EXAMPLES:

```
sage: # needs sage.symbolic
sage: legendre_Q.eval_formula(1, x)
1/2*x*(log(x + 1) - log(-x + 1)) - 1
sage: legendre_Q.eval_formula(2, x).expand().collect(log(1+x)).collect(log(1-
↪x))
1/4*(3*x^2 - 1)*log(x + 1) - 1/4*(3*x^2 - 1)*log(-x + 1) - 3/2*x
sage: legendre_Q.eval_formula(20, x).coefficient(x, 10)
-29113619535/131072*log(x + 1) + 29113619535/131072*log(-x + 1)
sage: legendre_Q(0, 2)
-1/2*I*pi + 1/2*log(3)

sage: legendre_Q(0, 2.)                                                     #␣
↪needs mpmath
0.549306144334055 - 1.57079632679490*I
```

eval_recursive(*n*, *arg*, ***kwds*)

Return expanded Legendre Q(n, arg) function expression.

EXAMPLES:

```
sage: legendre_Q.eval_recursive(2, x)                                       #␣
↪needs sage.symbolic
3/4*x^2*(log(x + 1) - log(-x + 1)) - 3/2*x - 1/4*log(x + 1) + 1/4*log(-x + 1)
sage: legendre_Q.eval_recursive(20, x).expand().coefficient(x, 10)          #␣
↪needs sage.symbolic
-29113619535/131072*log(x + 1) + 29113619535/131072*log(-x + 1)
```

class sage.functions.orthogonal_polys.**Func_meixner**

Bases: *OrthogonalFunction*

Meixner polynomials $M_n(x; b, c)$.

INPUT:

- `n` – the degree
- `x` – the independent variable x
- `b, c` – the parameters b, c

eval_formula(*n*, *x*, *b*, *c*)

Evaluate `self` using an explicit formula.

EXAMPLES:

```
sage: x, b, c = var('x,b,c')                                                #␣
↪needs sage.symbolic
sage: meixner.eval_formula(3, x, b, c).expand().collect(x)                  #␣
↪needs sage.symbolic
-x^3*(3/c - 3/c^2 + 1/c^3 - 1) + b^3
 + 3*(b - 2*b/c + b/c^2 - 1/c - 1/c^2 + 1/c^3 + 1)*x^2 + 3*b^2
 + (3*b^2 + 6*b - 3*b^2/c - 3*b/c - 3*b/c^2 - 2/c^3 + 2)*x + 2*b
```

eval_recursive (*n*, *x*, *b*, *c*, **args*, ***kwds*)

Return the Meixner polynomial $M_n(x; b, c)$ using the recursive formula.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, b, c = var('x,b,c')
sage: meixner.eval_recursive(0, x, b, c)
1
sage: meixner.eval_recursive(1, x, b, c)
-x*(1/c - 1) + b
sage: meixner.eval_recursive(2, x, b, c).simplify_full().collect(x)
-x^2*(2/c - 1/c^2 - 1) + b^2 + (2*b - 2*b/c - 1/c^2 + 1)*x + b
sage: bool(meixner(2, x, b, c) == meixner.eval_recursive(2, x, b, c))
True
sage: bool(meixner(3, x, b, c) == meixner.eval_recursive(3, x, b, c))
True
sage: bool(meixner(4, x, b, c) == meixner.eval_recursive(4, x, b, c))
True
sage: M = matrix([[-1/2, -1], [1, 0]])
sage: ret = meixner.eval_recursive(2, M, b, c).simplify_full().factor()
sage: for i in range(2):  # make the output polynomials in 1/c
....:     for j in range(2):
....:         ret[i, j] = ret[i, j].collect(c)
sage: ret
[b^2 + 1/2*(2*b + 3)/c - 1/4/c^2 - 5/4    -2*b + (2*b - 1)/c + 3/2/c^2 - 1/2]
[    2*b - (2*b - 1)/c - 3/2/c^2 + 1/2             b^2 + b + 2/c - 1/c^2 - 1]
```

class sage.functions.orthogonal_polys.**Func_ultraspherical**

Bases: GinacFunction

Return the ultraspherical (or Gegenbauer) polynomial `gegenbauer(n,a,x)`,

$$C_n^a(x) = \sum_{k=0}^{\lfloor n/2 \rfloor} (-1)^k \frac{\Gamma(n-k+a)}{\Gamma(a)k!(n-2k)!}(2x)^{n-2k}.$$

When n is a nonnegative integer, this formula gives a polynomial in z of degree n, but all parameters are permitted to be complex numbers. When $a = 1/2$, the Gegenbauer polynomial reduces to a Legendre polynomial.

Computed using Pynac.

For numerical evaluation, consider using the mpmath library, as it also allows complex numbers (and negative n as well); see the examples below.

REFERENCE:

- [AS1964] 22.5.27

EXAMPLES:

```
sage: # needs sage.symbolic
sage: ultraspherical(8, 101/11, x)
795972057547264/214358881*x^8 - 62604543852032/19487171*x^6...
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(2, 3/2, x)
15/2*x^2 - 3/2
sage: ultraspherical(1, 1, x)
2*x
sage: t = PolynomialRing(RationalField(), "t").gen()
```

(continues on next page)

(continued from previous page)

```
sage: gegenbauer(3, 2, t)
32*t^3 - 12*t
sage: x = SR.var('x')
sage: n = ZZ.random_element(5, 5001)
sage: a = QQ.random_element().abs() + 5
sage: s = (  (n + 1)*ultraspherical(n + 1, a, x)
....:      - 2*x*(n + a)*ultraspherical(n, a, x)
....:      + (n + 2*a - 1)*ultraspherical(n - 1, a, x) )
sage: s.expand().is_zero()
True
sage: ultraspherical(5, 9/10, 3.1416)
6949.55439044240
sage: ultraspherical(5, 9/10, RealField(100)(pi))                          #␣
↪needs sage.rings.real_mpfr
6949.4695419382702451843080687

sage: # needs sage.symbolic
sage: a, n = SR.var('a,n')
sage: gegenbauer(2, a, x)
2*(a + 1)*a*x^2 - a
sage: gegenbauer(3, a, x)
4/3*(a + 2)*(a + 1)*a*x^3 - 2*(a + 1)*a*x
sage: gegenbauer(3, a, x).expand()
4/3*a^3*x^3 + 4*a^2*x^3 + 8/3*a*x^3 - 2*a^2*x - 2*a*x
sage: gegenbauer(10, a, x).expand().coefficient(x, 2)
1/12*a^6 + 5/4*a^5 + 85/12*a^4 + 75/4*a^3 + 137/6*a^2 + 10*a
sage: ex = gegenbauer(100, a, x)
sage: (ex.subs(a==55/98) - gegenbauer(100, 55/98, x)).is_trivial_zero()
True

sage: # needs sage.symbolic
sage: gegenbauer(2, -3, x)
12*x^2 + 3
sage: gegenbauer(120,-99/2,3)
1654502372608570682112687530178328494861923493372493824
sage: gegenbauer(5, 9/2, x)
21879/8*x^5 - 6435/4*x^3 + 1287/8*x
sage: gegenbauer(15,3/2,5)
3903412392243800

sage: derivative(gegenbauer(n, a, x), x)                                   #␣
↪needs sage.symbolic
2*a*gegenbauer(n - 1, a + 1, x)
sage: derivative(gegenbauer(3, a, x), x)                                   #␣
↪needs sage.symbolic
4*(a + 2)*(a + 1)*a*x^2 - 2*(a + 1)*a
sage: derivative(gegenbauer(n, a, x), a)                                   #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
RuntimeError: derivative w.r.t. to the second index is not supported yet
```

Numerical evaluation with the mpmath library:

```
sage: # needs mpmath
sage: from mpmath import gegenbauer as gegenbauer_mp
sage: from mpmath import mp
```

(continues on next page)

(continued from previous page)

```
sage: mp.pretty = True; mp.dps=25
sage: gegenbauer_mp(-7,0.5,0.3)
0.1291811875
sage: gegenbauer_mp(2+3j, -0.75, -1000j)
(-5038991.358609026523401901 + 9414549.285447104177860806j)
```

class sage.functions.orthogonal_polys.**OrthogonalFunction**(*name*, *nargs=2*, *latex_name=None*, *conversions=None*)

Bases: BuiltinFunction

Base class for orthogonal polynomials.

This class is an abstract base class for all orthogonal polynomials since they share similar properties. The evaluation as a polynomial is either done via maxima, or with pynac.

Convention: The first argument is always the order of the polynomial, the others are other values or parameters where the polynomial is evaluated.

eval_formula(**args*)

Evaluate this polynomial using an explicit formula.

EXAMPLES:

```
sage: from sage.functions.orthogonal_polys import OrthogonalFunction
sage: P = OrthogonalFunction('testo_P')
sage: P.eval_formula(1,2.0)
Traceback (most recent call last):
...
NotImplementedError: no explicit calculation of values implemented
```

1.9 Other functions

class sage.functions.other.**Function_Order**

Bases: GinacFunction

The order function.

This function gives the order of magnitude of some expression, similar to O-terms.

See also:

Order(), big_oh

EXAMPLES:

```
sage: x = SR('x')                                                    #␣
↪needs sage.symbolic
sage: x.Order()                                                      #␣
↪needs sage.symbolic
Order(x)
sage: (x^2 + x).Order()                                              #␣
↪needs sage.symbolic
Order(x^2 + x)
```

class sage.functions.other.**Function_abs**

Bases: GinacFunction

The absolute value function.

EXAMPLES:

```
sage: abs(-2)
2

sage: # needs sage.symbolic
sage: var('x y')
(x, y)
sage: abs(x)
abs(x)
sage: abs(x^2 + y^2)
abs(x^2 + y^2)
sage: sqrt(x^2)
sqrt(x^2)
sage: abs(sqrt(x))
sqrt(abs(x))
sage: complex(abs(3*I))
(3+0j)

sage: f = sage.functions.other.Function_abs()
sage: latex(f)
\mathrm{abs}
sage: latex(abs(x))                                                     #␣
↪needs sage.symbolic
{\left| x \right|}
sage: abs(x)._sympy_()                                                  #␣
↪needs sympy sage.symbolic
Abs(x)
```

Test pickling:

```
sage: loads(dumps(abs(x)))                                              #␣
↪needs sage.symbolic
abs(x)
```

class sage.functions.other.**Function_arg**

Bases: BuiltinFunction

The argument function for complex numbers.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: arg(3+i)
arctan(1/3)
sage: arg(-1+i)
3/4*pi
sage: arg(2+2*i)
1/4*pi
sage: arg(2+x)
arg(x + 2)
sage: arg(2.0+i+x)
arg(x + 2.00000000000000 + 1.00000000000000*I)
sage: arg(-3)
```

(continues on next page)

(continued from previous page)

```
pi
sage: arg(3)
0
sage: arg(0)
0

sage: # needs sage.symbolic
sage: latex(arg(x))
{\rm arg}\left(x\right)
sage: maxima(arg(x))
atan2(0,_SAGE_VAR_x)
sage: maxima(arg(2+i))
atan(1/2)
sage: maxima(arg(sqrt(2)+i))
atan(1/sqrt(2))
sage: arg(x)._sympy_()                                                  #␣
↪needs sympy
arg(x)

sage: arg(2+i)                                                          #␣
↪needs sage.symbolic
arctan(1/2)
sage: arg(sqrt(2)+i)                                                    #␣
↪needs sage.symbolic
arg(sqrt(2) + I)
sage: arg(sqrt(2)+i).simplify()                                         #␣
↪needs sage.symbolic
arctan(1/2*sqrt(2))
```

class sage.functions.other.**Function_binomial**

Bases: GinacFunction

Return the binomial coefficient

$$\binom{x}{m} = x(x-1)\cdots(x-m+1)/m!$$

which is defined for $m \in \mathbf{Z}$ and any x. We extend this definition to include cases when $x - m$ is an integer but m is not by

$$\binom{x}{m} = \binom{x}{x-m}$$

If $m < 0$, return 0.

INPUT:

- x, m - numbers or symbolic expressions. Either m or x-m must be an integer, else the output is symbolic.

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: binomial(5, 2)
10
sage: binomial(2, 0)
1
sage: binomial(1/2, 0)                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.pari
1
sage: binomial(3, -1)
0
sage: binomial(20, 10)
184756
sage: binomial(-2, 5)
-6
sage: n = var('n'); binomial(n, 2)
1/2*(n - 1)*n
sage: n = var('n'); binomial(n, n)
1
sage: n = var('n'); binomial(n, n - 1)
n
sage: binomial(2^100, 2^100)
1

sage: binomial(RealField()('2.5'), 2)                                    #␣
↪needs sage.rings.real_mpfr
1.87500000000000
```

```
sage: k, i = var('k,i')                                                  #␣
↪needs sage.symbolic
sage: binomial(k,i)                                                      #␣
↪needs sage.symbolic
binomial(k, i)
```

We can use a `hold` parameter to prevent automatic evaluation:

```
sage: SR(5).binomial(3, hold=True)                                       #␣
↪needs sage.symbolic
binomial(5, 3)
sage: SR(5).binomial(3, hold=True).simplify()                            #␣
↪needs sage.symbolic
10
```

class sage.functions.other.**Function_cases**

Bases: `GinacFunction`

Formal function holding `(condition, expression)` pairs.

Numbers are considered conditions with zero being `False`. A true condition marks a default value. The function is not evaluated as long as it contains a relation that cannot be decided by Pynac.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: ex = cases([(x==0, pi), (True, 0)]); ex
cases(((x == 0, pi), (1, 0)))
sage: ex.subs(x==0)
pi
sage: ex.subs(x==2)
0
sage: ex + 1
cases(((x == 0, pi), (1, 0))) + 1
sage: _.subs(x==0)
pi + 1
```

The first encountered default is used, as well as the first relation that can be trivially decided:

```
sage: cases(((True, pi), (True, 0)))                                        #␣
↪needs sage.symbolic
pi

sage: # needs sage.symbolic
sage: _ = var('y')
sage: ex = cases(((x==0, pi), (y==1, 0))); ex
cases(((x == 0, pi), (y == 1, 0)))
sage: ex.subs(x==0)
pi
sage: ex.subs(x==0, y==1)
pi
```

class sage.functions.other.**Function_ceil**

Bases: BuiltinFunction

The ceiling function.

The ceiling of x is computed in the following manner.

1. The `x.ceil()` method is called and returned if it is there. If it is not, then Sage checks if x is one of Python's native numeric data types. If so, then it calls and returns `Integer(math.ceil(x))`.
2. Sage tries to convert x into a `RealIntervalField` with 53 bits of precision. Next, the ceilings of the endpoints are computed. If they are the same, then that value is returned. Otherwise, the precision of the `RealIntervalField` is increased until they do match up or it reaches `bits` of precision.
3. If none of the above work, Sage returns a `Expression` object.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: a = ceil(2/5 + x); a
ceil(x + 2/5)
sage: a(x=4)
5
sage: a(x=4.0)
5
sage: ZZ(a(x=3))
4
sage: a = ceil(x^3 + x + 5/2); a
ceil(x^3 + x + 5/2)
sage: a.simplify()
ceil(x^3 + x + 1/2) + 2
sage: a(x=2)
13
```

```
sage: ceil(sin(8)/sin(2))                                                   #␣
↪needs sage.symbolic
2
```

```
sage: ceil(5.4)
6
sage: type(ceil(5.4))
<class 'sage.rings.integer.Integer'>
```

```
sage: ceil(factorial(50)/exp(1))                                          #␣
↪needs sage.symbolic
11188719610782480504630258070757734324011354208865721592720336801
sage: ceil(SR(10^50 + 10^(-50)))                                          #␣
↪needs sage.symbolic
100000000000000000000000000000000000000000000000001
sage: ceil(SR(10^50 - 10^(-50)))                                          #␣
↪needs sage.symbolic
100000000000000000000000000000000000000000000000000
```

Small numbers which are extremely close to an integer are hard to deal with:

```
sage: ceil((33^100 + 1)^(1/100))                                          #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: cannot compute ceil(...) using 256 bits of precision
```

This can be fixed by giving a sufficiently large `bits` argument:

```
sage: ceil((33^100 + 1)^(1/100), bits=500)                                #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: cannot compute ceil(...) using 512 bits of precision
sage: ceil((33^100 + 1)^(1/100), bits=1000)                               #␣
↪needs sage.symbolic
34
```

```
sage: ceil(sec(e))                                                        #␣
↪needs sage.symbolic
-1

sage: latex(ceil(x))                                                      #␣
↪needs sage.symbolic
\left \lceil x \right \rceil
sage: ceil(x)._sympy_()                                                   #␣
↪needs sympy sage.symbolic
ceiling(x)
```

```
sage: import numpy                                                        #␣
↪needs numpy
sage: a = numpy.linspace(0,2,6)                                           #␣
↪needs numpy
sage: ceil(a)                                                             #␣
↪needs numpy
array([0., 1., 1., 2., 2., 2.])
```

Test pickling:

```
sage: loads(dumps(ceil))
ceil
```

class sage.functions.other.**Function_conjugate**

Bases: GinacFunction

Returns the complex conjugate of the input.

It is possible to prevent automatic evaluation using the `hold` parameter:

```
sage: conjugate(I, hold=True)                                              #␣
↪needs sage.symbolic
conjugate(I)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: conjugate(I, hold=True).simplify()                                   #␣
↪needs sage.symbolic
-I
```

class `sage.functions.other.`**`Function_crootof`**

Bases: `BuiltinFunction`

Formal function holding `(polynomial, index)` pairs.

The expression evaluates to a floating point value that is an approximation to a specific complex root of the polynomial. The ordering is fixed so you always get the same root.

The functionality is imported from SymPy, see http://docs.sympy.org/latest/_modules/sympy/polys/rootoftools.html

EXAMPLES:

```
sage: # needs sage.symbolic
sage: c = complex_root_of(x^6 + x + 1, 1); c
complex_root_of(x^6 + x + 1, 1)
sage: c.n()
-0.790667188814418 + 0.300506920309552*I
sage: c.n(100)
-0.79066718881441764449859281847 + 0.30050692030955162512001002521*I
sage: (c^6 + c + 1).n(100) < 1e-25
True
```

class `sage.functions.other.`**`Function_elementof`**

Bases: `BuiltinFunction`

Formal set membership function that is only accessible internally.

This function is called to express a set membership statement, usually as part of a solution set returned by `solve()`. See `sage.sets.set.Set` and `sage.sets.real_set.RealSet` for possible set arguments.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.functions.other import element_of
sage: element_of(x, SR(ZZ))
element_of(x, Integer Ring)
sage: element_of(sin(x), SR(QQ))
element_of(sin(x), Rational Field)
sage: element_of(x, SR(RealSet.open_closed(0,1)))
element_of(x, (0, 1])
sage: element_of(x, SR(Set([4,6,8])))
element_of(x, {8, 4, 6})
```

class `sage.functions.other.`**`Function_factorial`**

Bases: `GinacFunction`

Returns the factorial of n.

INPUT:

- `n` – a non-negative integer, a complex number (except negative integers) or any symbolic expression

OUTPUT: an integer or symbolic expression

EXAMPLES:

```
sage: factorial(0)
1
sage: factorial(4)
24
sage: factorial(10)
3628800
sage: factorial(6) == 6*5*4*3*2
True

sage: # needs sage.symbolic
sage: x = SR.var('x')
sage: f = factorial(x + factorial(x)); f
factorial(x + factorial(x))
sage: f(x=3)
362880
sage: factorial(x)^2
factorial(x)^2
```

To prevent automatic evaluation use the `hold` argument:

```
sage: factorial(5, hold=True)                                          #␣
↪needs sage.symbolic
factorial(5)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: factorial(5, hold=True).simplify()                               #␣
↪needs sage.symbolic
120
```

We can also give input other than nonnegative integers. For other nonnegative numbers, the `sage.functions.gamma.gamma()` function is used:

```
sage: factorial(1/2)                                                   #␣
↪needs sage.symbolic
1/2*sqrt(pi)
sage: factorial(3/4)                                                   #␣
↪needs sage.symbolic
gamma(7/4)
sage: factorial(2.3)                                                   #␣
↪needs sage.symbolic
2.68343738195577
```

But negative input always fails:

```
sage: factorial(-32)
Traceback (most recent call last):
...
ValueError: factorial only defined for non-negative integers
```

And very large integers remain unevaluated:

```
sage: factorial(2**64)                                                    #␣
↪needs sage.symbolic
factorial(18446744073709551616)
sage: SR(2**64).factorial()                                               #␣
↪needs sage.symbolic
factorial(18446744073709551616)
```

class sage.functions.other.**Function_floor**

Bases: `BuiltinFunction`

The floor function.

The floor of x is computed in the following manner.

1. The `x.floor()` method is called and returned if it is there. If it is not, then Sage checks if x is one of Python's native numeric data types. If so, then it calls and returns `Integer(math.floor(x))`.
2. Sage tries to convert x into a `RealIntervalField` with 53 bits of precision. Next, the floors of the endpoints are computed. If they are the same, then that value is returned. Otherwise, the precision of the `RealIntervalField` is increased until they do match up or it reaches `bits` of precision.
3. If none of the above work, Sage returns a symbolic `Expression` object.

EXAMPLES:

```
sage: floor(5.4)
5
sage: type(floor(5.4))
<class 'sage.rings.integer.Integer'>

sage: # needs sage.symbolic
sage: var('x')
x
sage: a = floor(5.25 + x); a
floor(x + 5.25000000000000)
sage: a.simplify()
floor(x + 0.25) + 5
sage: a(x=2)
7
```

```
sage: # needs sage.symbolic
sage: floor(cos(8) / cos(2))
0
sage: floor(log(4) / log(2))
2
sage: a = floor(5.4 + x); a
floor(x + 5.40000000000000)
sage: a.subs(x==2)
7
sage: floor(log(2^(3/2)) / log(2) + 1/2)
2
sage: floor(log(2^(-3/2)) / log(2) + 1/2)
-1
```

```
sage: floor(factorial(50)/exp(1))                                         #␣
↪needs sage.symbolic
11188719610782480504630258070757734324011354208865721592720336800
```

(continues on next page)

(continued from previous page)

```
sage: floor(SR(10^50 + 10^(-50)))                                       #␣
↪needs sage.symbolic
100000000000000000000000000000000000000000000000000
sage: floor(SR(10^50 - 10^(-50)))                                       #␣
↪needs sage.symbolic
99999999999999999999999999999999999999999999999999
sage: floor(int(10^50))
100000000000000000000000000000000000000000000000000
```

Small numbers which are extremely close to an integer are hard to deal with:

```
sage: floor((33^100 + 1)^(1/100))                                       #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: cannot compute floor(...) using 256 bits of precision
```

This can be fixed by giving a sufficiently large `bits` argument:

```
sage: floor((33^100 + 1)^(1/100), bits=500)                             #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: cannot compute floor(...) using 512 bits of precision
sage: floor((33^100 + 1)^(1/100), bits=1000)                            #␣
↪needs sage.symbolic
33
```

```
sage: import numpy                                                      #␣
↪needs numpy
sage: a = numpy.linspace(0,2,6)                                         #␣
↪needs numpy
sage: floor(a)                                                          #␣
↪needs numpy
array([0., 0., 0., 1., 1., 2.])
sage: floor(x)._sympy_()                                                #␣
↪needs sympy sage.symbolic
floor(x)
```

Test pickling:

```
sage: loads(dumps(floor))
floor
```

class sage.functions.other.**Function_frac**

Bases: `BuiltinFunction`

The fractional part function $\{x\}$.

`frac(x)` is defined as $\{x\} = x - \lfloor x \rfloor$.

EXAMPLES:

```
sage: frac(5.4)
0.400000000000000
sage: type(frac(5.4))
<class 'sage.rings.real_mpfr.RealNumber'>
```

(continues on next page)

(continued from previous page)

```
sage: frac(456/123)
29/41

sage: # needs sage.symbolic
sage: var('x')
x
sage: a = frac(5.4 + x); a
frac(x + 5.40000000000000)
sage: frac(cos(8)/cos(2))
cos(8)/cos(2)
sage: latex(frac(x))
\operatorname{frac}\left(x\right)
sage: frac(x)._sympy_()                                              #␣
↪needs sympy
frac(x)
```

Test pickling:

```
sage: loads(dumps(floor))
floor
```

class sage.functions.other.**Function_imag_part**

Bases: GinacFunction

Returns the imaginary part of the (possibly complex) input.

It is possible to prevent automatic evaluation using the hold parameter:

```
sage: imag_part(I, hold=True)                                        #␣
↪needs sage.symbolic
imag_part(I)
```

To then evaluate again, we currently must use Maxima via sage.symbolic.expression.Expression.simplify():

```
sage: imag_part(I, hold=True).simplify()                             #␣
↪needs sage.symbolic
1
```

class sage.functions.other.**Function_limit**

Bases: BuiltinFunction

Placeholder symbolic limit function that is only accessible internally.

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

```
sage: a = lim(exp(x^2)*(1-erf(x)), x=infinity); a                       #␣
↪needs sage.symbolic
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
```

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.functions.other import symbolic_limit as slimit
sage: slimit(1/x, x, +oo)
limit(1/x, x, +Infinity)
sage: var('minus,plus')
```

(continues on next page)

(continued from previous page)

```
(minus, plus)
sage: slimit(1/x, x, +oo)
limit(1/x, x, +Infinity)
sage: slimit(1/x, x, 0, plus)
limit(1/x, x, 0, plus)
sage: slimit(1/x, x, 0, minus)
limit(1/x, x, 0, minus)
```

class sage.functions.other.**Function_prod**

Bases: BuiltinFunction

Placeholder symbolic product function that is only accessible internally.

EXAMPLES:

```
sage: from sage.functions.other import symbolic_product as sprod
sage: r = sprod(x, x, 1, 10); r                                    #␣
↪needs sage.symbolic
product(x, x, 1, 10)
sage: r.unhold()                                                   #␣
↪needs sage.symbolic
3628800
```

class sage.functions.other.**Function_real_nth_root**

Bases: BuiltinFunction

Real n-th root function $x^{\frac{1}{n}}$.

The function assumes positive integer n and real number x.

EXAMPLES:

```
sage: real_nth_root(2, 3)                                          #␣
↪needs sage.symbolic
2^(1/3)
sage: real_nth_root(-2, 3)                                         #␣
↪needs sage.symbolic
-2^(1/3)
sage: real_nth_root(8, 3)
2
sage: real_nth_root(-8, 3)
-2

sage: real_nth_root(-2, 4)
Traceback (most recent call last):
...
ValueError: no real nth root of negative real number with even n
```

For numeric input, it gives a numerical approximation.

```
sage: real_nth_root(2., 3)
1.25992104989487
sage: real_nth_root(-2., 3)
-1.25992104989487
```

Some symbolic calculus:

```
sage: # needs sage.symbolic
sage: f = real_nth_root(x, 5)^3; f
real_nth_root(x^3, 5)
sage: f.diff()
3/5*x^2*real_nth_root(x^(-12), 5)
sage: result = f.integrate(x)
...
sage: result
integrate((abs(x)^3)^(1/5)*sgn(x^3), x)
sage: _.diff()
(abs(x)^3)^(1/5)*sgn(x^3)
```

class sage.functions.other.**Function_real_part**

Bases: GinacFunction

Returns the real part of the (possibly complex) input.

It is possible to prevent automatic evaluation using the `hold` parameter:

```
sage: real_part(I, hold=True)                                               #␣
↪needs sage.symbolic
real_part(I)
```

To then evaluate again, we currently must use Maxima via `sage.symbolic.expression.Expression.simplify()`:

```
sage: real_part(I, hold=True).simplify()                                    #␣
↪needs sage.symbolic
0
```

EXAMPLES:

```
sage: z = 1+2*I                                                             #␣
↪needs sage.symbolic
sage: real(z)                                                               #␣
↪needs sage.symbolic
1
sage: real(5/3)
5/3
sage: a = 2.5
sage: real(a)
2.50000000000000
sage: type(real(a))
<class 'sage.rings.real_mpfr.RealLiteral'>
sage: real(1.0r)
1.0
sage: real(complex(3, 4))
3.0
```

Sage can recognize some expressions as real and accordingly return the identical argument:

```
sage: # needs sage.symbolic
sage: SR.var('x', domain='integer').real_part()
x
sage: SR.var('x', domain='integer').imag_part()
0
sage: real_part(sin(x)+x)
x + sin(x)
```

(continues on next page)

(continued from previous page)

```
sage: real_part(x*exp(x))
x*e^x
sage: imag_part(sin(x)+x)
0
sage: real_part(real_part(x))
x
sage: forget()
```

class sage.functions.other.**Function_sqrt**

Bases: object

class sage.functions.other.**Function_sum**

Bases: BuiltinFunction

Placeholder symbolic sum function that is only accessible internally.

EXAMPLES:

```
sage: from sage.functions.other import symbolic_sum as ssum
sage: r = ssum(x, x, 1, 10); r                                          #␣
↪needs sage.symbolic
sum(x, x, 1, 10)
sage: r.unhold()                                                        #␣
↪needs sage.symbolic
55
```

1.10 Miscellaneous special functions

This module provides easy access to many of Maxima and PARI's special functions.

Maxima's special functions package (which includes spherical harmonic functions, spherical Bessel functions (of the 1st and 2nd kind), and spherical Hankel functions (of the 1st and 2nd kind)) was written by Barton Willis of the University of Nebraska at Kearney. It is released under the terms of the General Public License (GPL).

Support for elliptic functions and integrals was written by Raymond Toy. It is placed under the terms of the General Public License (GPL) that governs the distribution of Maxima.

Next, we summarize some of the properties of the functions implemented here.

- **Spherical harmonics**: Laplace's equation in spherical coordinates is:

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right)+\frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right)+\frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\varphi^2}=0.$$

  Note that the spherical coordinates θ and φ are defined here as follows: θ is the colatitude or polar angle, ranging from $0\leq\theta\leq\pi$ and φ the azimuth or longitude, ranging from $0\leq\varphi<2\pi$.

  The general solution which remains finite towards infinity is a linear combination of functions of the form

$$r^{-1-\ell}\cos(m\varphi)P_\ell^m(\cos\theta)$$

  and

$$r^{-1-\ell}\sin(m\varphi)P_\ell^m(\cos\theta)$$

where P_ℓ^m are the associated Legendre polynomials (cf. `Func_assoc_legendre_P`), and with integer parameters $\ell \geq 0$ and m from 0 to ℓ. Put in another way, the solutions with integer parameters $\ell \geq 0$ and $-\ell \leq m \leq \ell$, can be written as linear combinations of:

$$U_{\ell,m}(r,\theta,\varphi) = r^{-1-\ell}Y_\ell^m(\theta,\varphi)$$

where the functions Y are the spherical harmonic functions with parameters ℓ, m, which can be written as:

$$Y_\ell^m(\theta,\varphi) = \sqrt{\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}\,e^{im\varphi}\,P_\ell^m(\cos\theta).$$

The spherical harmonics obey the normalisation condition

$$\int_{\theta=0}^{\pi}\int_{\varphi=0}^{2\pi} Y_\ell^m Y_{\ell'}^{m'*}\,d\Omega = \delta_{\ell\ell'}\delta_{mm'} \qquad d\Omega = \sin\theta\,d\varphi\,d\theta.$$

- The **incomplete elliptic integrals** (of the first kind, etc.) are:

$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2}}\,dx,$$
$$\int_0^\phi \sqrt{1-m\sin(x)^2}\,dx,$$
$$\int_0^\phi \frac{\sqrt{1-mt^2}}{\sqrt{(1-t^2)}}\,dx,$$
$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2\sqrt{1-n\sin(x)^2}}}\,dx,$$

and the complete ones are obtained by taking $\phi = \pi/2$.

Warning: SciPy's versions are poorly documented and seem less accurate than the Maxima and PARI versions. Typically they are limited by hardware floats precision.

REFERENCES:

- Abramowitz and Stegun: *Handbook of Mathematical Functions* [AS1964]
- Wikipedia article Spherical_harmonics
- Wikipedia article Helmholtz_equation
- Online Encyclopedia of Special Functions

AUTHORS:

- David Joyner (2006-13-06): initial version
- David Joyner (2006-30-10): bug fixes to pari wrappers of Bessel functions, hypergeometric_U
- William Stein (2008-02): Impose some sanity checks.
- David Joyner (2008-02-16): optional calls to scipy and replace all `#random` by `...`
- David Joyner (2008-04-23): addition of elliptic integrals
- Eviatar Bach (2013): making elliptic integrals symbolic
- Eric Gourgoulhon (2022): add Condon-Shortley phase to spherical harmonics

class `sage.functions.special.`**`EllipticE`**

Bases: `BuiltinFunction`

Return the incomplete elliptic integral of the second kind:

$$E(\varphi\,|\,m) = \int_0^\varphi \sqrt{1 - m\sin(x)^2}\, dx.$$

EXAMPLES:

```
sage: z = var("z")                                                      #␣
↪needs sage.symbolic
sage: elliptic_e(z, 1)                                                  #␣
↪needs sage.symbolic
elliptic_e(z, 1)
sage: elliptic_e(z, 1).simplify()        # not tested                   #␣
↪needs sage.symbolic
2*round(z/pi) - sin(pi*round(z/pi) - z)
sage: elliptic_e(z, 0)                                                  #␣
↪needs sage.symbolic
z
sage: elliptic_e(0.5, 0.1)   # abs tol 2e-15                            #␣
↪needs mpmath
0.498011394498832
sage: elliptic_e(1/2, 1/10).n(200)                                      #␣
↪needs sage.symbolic
0.4980113944988315331154610406...
```

See also:

- Taking $\varphi = \pi/2$ gives *`elliptic_ec()`*.
- Taking $\varphi = \arcsin(\mathrm{sn}(u, m))$ gives *`elliptic_eu()`*.

REFERENCES:

- Wikipedia article Elliptic_integral#Incomplete_elliptic_integral_of_the_second_kind
- Wikipedia article Jacobi_elliptic_functions

class `sage.functions.special.`**`EllipticEC`**

Bases: `BuiltinFunction`

Return the complete elliptic integral of the second kind:

$$E(m) = \int_0^{\pi/2} \sqrt{1 - m\sin(x)^2}\, dx.$$

EXAMPLES:

```
sage: elliptic_ec(0.1)                                                  #␣
↪needs mpmath
1.53075763689776
sage: elliptic_ec(x).diff()                                             #␣
↪needs sage.symbolic
1/2*(elliptic_ec(x) - elliptic_kc(x))/x
```

See also:

- *elliptic_e()*.

REFERENCES:

- Wikipedia article Elliptic_integral#Complete_elliptic_integral_of_the_second_kind

class sage.functions.special.**EllipticEU**

Bases: BuiltinFunction

Return Jacobi's form of the incomplete elliptic integral of the second kind:

$$E(u,m) = \int_0^u \mathrm{dn}(x,m)^2\, dx = \int_0^\tau \frac{\sqrt{1-mx^2}}{\sqrt{1-x^2}}\, dx.$$

where $\tau = \mathrm{sn}(u, m)$.

Also, `elliptic_eu(u, m) = elliptic_e(asin(sn(u,m)),m)`.

EXAMPLES:

```
sage: elliptic_eu(0.5, 0.1)                                             #␣
↪needs mpmath
0.496054551286597
```

See also:

- *elliptic_e()*.

REFERENCES:

- Wikipedia article Elliptic_integral#Incomplete_elliptic_integral_of_the_second_kind
- Wikipedia article Jacobi_elliptic_functions

class sage.functions.special.**EllipticF**

Bases: BuiltinFunction

Return the incomplete elliptic integral of the first kind.

$$F(\varphi\,|\,m) = \int_0^\varphi \frac{dx}{\sqrt{1 - m\sin(x)^2}},$$

Taking $\varphi = \pi/2$ gives *elliptic_kc()*.

EXAMPLES:

```
sage: z = var("z")                                                      #␣
↪needs sage.symbolic
sage: elliptic_f(z, 0)                                                  #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
z
sage: elliptic_f(z, 1).simplify()                                    #␣
↪needs sage.symbolic
log(tan(1/4*pi + 1/2*z))
sage: elliptic_f(0.2, 0.1)                                           #␣
↪needs mpmath
0.200132506747543
```

See also:

- *elliptic_e()*.

REFERENCES:

- Wikipedia article Elliptic_integral#Incomplete_elliptic_integral_of_the_first_kind

class sage.functions.special.**EllipticKC**

Bases: BuiltinFunction

Return the complete elliptic integral of the first kind:

$$K(m) = \int_0^{\pi/2} \frac{dx}{\sqrt{1 - m\sin(x)^2}}.$$

EXAMPLES:

```
sage: elliptic_kc(0.5)                                               #␣
↪needs mpmath
1.85407467730137
```

See also:

- *elliptic_f()*.
- *elliptic_ec()*.

REFERENCES:

- Wikipedia article Elliptic_integral#Complete_elliptic_integral_of_the_first_kind
- Wikipedia article Elliptic_integral#Incomplete_elliptic_integral_of_the_first_kind

class sage.functions.special.**EllipticPi**

Bases: BuiltinFunction

Return the incomplete elliptic integral of the third kind:

$$\Pi(n, t, m) = \int_0^t \frac{dx}{(1 - n\sin(x)^2)\sqrt{1 - m\sin(x)^2}}.$$

INPUT:

- n – a real number, called the "characteristic"
- t – a real number, called the "amplitude"
- m – a real number, called the "parameter"

EXAMPLES:

```
sage: N(elliptic_pi(1, pi/4, 1))                                        #␣
↪needs sage.symbolic
1.14779357469632
```

Compare the value computed by Maxima to the definition as a definite integral (using GSL):

```
sage: elliptic_pi(0.1, 0.2, 0.3)                                        #␣
↪needs mpmath
0.200665068220979
sage: numerical_integral(1/(1-0.1*sin(x)^2)/sqrt(1-0.3*sin(x)^2), 0.0, 0.2)     #␣
↪needs sage.symbolic
(0.2006650682209791, 2.227829789769088e-15)
```

REFERENCES:

- Wikipedia article Elliptic_integral#Incomplete_elliptic_integral_of_the_third_kind

class sage.functions.special.**SphericalHarmonic**

Bases: BuiltinFunction

Returns the spherical harmonic function $Y_n^m(\theta, \varphi)$.

For integers $n > -1$, $|m| \leq n$, simplification is done automatically. Numeric evaluation is supported for complex n and m.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, y = var('x, y')
sage: spherical_harmonic(3, 2, x, y)
1/8*sqrt(30)*sqrt(7)*cos(x)*e^(2*I*y)*sin(x)^2/sqrt(pi)
sage: spherical_harmonic(3, 2, 1, 2)
1/8*sqrt(30)*sqrt(7)*cos(1)*e^(4*I)*sin(1)^2/sqrt(pi)
sage: spherical_harmonic(3 + I, 2., 1, 2)
-0.351154337307488 - 0.415562233975369*I
sage: latex(spherical_harmonic(3, 2, x, y, hold=True))
Y_{3}^{2}\left(x, y\right)
sage: spherical_harmonic(1, 2, x, y)
0
```

The degree n and the order m can be symbolic:

```
sage: # needs sage.symbolic
sage: n, m = var('n m')
sage: spherical_harmonic(n, m, x, y)
spherical_harmonic(n, m, x, y)
sage: latex(spherical_harmonic(n, m, x, y))
Y_{n}^{m}\left(x, y\right)
sage: diff(spherical_harmonic(n, m, x, y), x)
m*cot(x)*spherical_harmonic(n, m, x, y)
 + sqrt(-(m + n + 1)*(m - n))*e^(-I*y)*spherical_harmonic(n, m + 1, x, y)
sage: diff(spherical_harmonic(n, m, x, y), y)
I*m*spherical_harmonic(n, m, x, y)
```

The convention regarding the Condon-Shortley phase $(-1)^m$ is the same as for SymPy's spherical harmonics and Wikipedia article Spherical_harmonics:

```
sage: # needs sage.symbolic
sage: spherical_harmonic(1, 1, x, y)
-1/4*sqrt(3)*sqrt(2)*e^(I*y)*sin(x)/sqrt(pi)
sage: from sympy import Ynm                                                  #␣
↪needs sympy
sage: Ynm(1, 1, x, y).expand(func=True)                                      #␣
↪needs sympy
-sqrt(6)*exp(I*y)*sin(x)/(4*sqrt(pi))
sage: spherical_harmonic(1, 1, x, y) - Ynm(1, 1, x, y)                       #␣
↪needs sympy
0
```

It also agrees with SciPy's spherical harmonics:

```
sage: spherical_harmonic(1, 1, pi/2, pi).n()  # abs tol 1e-14                #␣
↪needs sage.symbolic
0.345494149471335
sage: from scipy.special import sph_harm  # NB: arguments x and y are swapped   #␣
↪needs scipy
sage: sph_harm(1, 1, pi.n(), (pi/2).n())  # abs tol 1e-14                    #␣
↪needs scipy sage.symbolic
(0.3454941494713355-4.231083042742082e-17j)
```

Note that this convention differs from the one in Maxima, as revealed by the sign difference for odd values of m:

```
sage: maxima.spherical_harmonic(1, 1, x, y).sage()                           #␣
↪needs sage.symbolic
1/2*sqrt(3/2)*e^(I*y)*sin(x)/sqrt(pi)
```

It follows that, contrary to Maxima, SageMath uses the same sign convention for spherical harmonics as SymPy, SciPy, Mathematica and Wikipedia article Table_of_spherical_harmonics.

REFERENCES:

- Wikipedia article Spherical_harmonics

`sage.functions.special.`**`elliptic_eu_f`**`(`*u*, *m*`)`

Internal function for numeric evaluation of `elliptic_eu`, defined as $E(\mathrm{am}(u, m)|m)$, where E is the incomplete elliptic integral of the second kind and am is the Jacobi amplitude function.

EXAMPLES:

```
sage: from sage.functions.special import elliptic_eu_f
sage: elliptic_eu_f(0.5, 0.1)                                                #␣
↪needs mpmath
mpf('0.49605455128659691')
```

`sage.functions.special.`**`elliptic_j`**`(`*z*, *prec=53*`)`

Returns the elliptic modular j-function evaluated at z.

INPUT:

- `z` (complex) – a complex number with positive imaginary part.
- `prec` (default: 53) – precision in bits for the complex field.

OUTPUT:

(complex) The value of $j(z)$.

ALGORITHM:

Calls the `pari` function `ellj()`.

AUTHOR:

John Cremona

EXAMPLES:

```
sage: elliptic_j(CC(i))                                                    #␣
↪needs sage.rings.real_mpfr
1728.00000000000
sage: elliptic_j(sqrt(-2.0))                                               #␣
↪needs sage.rings.complex_double
8000.00000000000
sage: z = ComplexField(100)(1, sqrt(11))/2                                 #␣
↪needs sage.rings.real_mpfr sage.symbolic
sage: elliptic_j(z)                                                        #␣
↪needs sage.rings.real_mpfr sage.symbolic
-32768.000...
sage: elliptic_j(z).real().round()                                         #␣
↪needs sage.rings.real_mpfr sage.symbolic
-32768
```

```
sage: tau = (1 + sqrt(-163))/2                                             #␣
↪needs sage.symbolic
sage: (-elliptic_j(tau.n(100)).real().round())^(1/3)                       #␣
↪needs sage.symbolic
640320
```

This example shows the need for higher precision than the default one of the $ComplexField$, see github issue #28355:

```
sage: # needs sage.symbolic
sage: -elliptic_j(tau)  # rel tol 1e-2
2.62537412640767e17 - 732.558854258998*I
sage: -elliptic_j(tau, 75)  # rel tol 1e-2
2.625374126407680000000e17 - 0.0001309913593909879441262*I
sage: -elliptic_j(tau, 100)  # rel tol 1e-2
2.625374126407679999999999999999e17 - 1.301282240035688712294511979e-12*I
sage: (-elliptic_j(tau, 100).real().round())^(1/3)
640320
```

1.11 Hypergeometric functions

This module implements manipulation of infinite hypergeometric series represented in standard parametric form (as ${}_pF_q$ functions).

AUTHORS:

- Fredrik Johansson (2010): initial version
- Eviatar Bach (2013): major changes

EXAMPLES:

Examples from github issue #9908:

```
sage: maxima('integrate(bessel_j(2, x), x)').sage()
1/24*x^3*hypergeometric((3/2,), (5/2, 3), -1/4*x^2)
sage: sum(((2*I)^x/(x^3 + 1)*(1/4)^x), x, 0, oo)
hypergeometric((1, 1, -1/2*I*sqrt(3) - 1/2, 1/2*I*sqrt(3) - 1/2),...
(2, -1/2*I*sqrt(3) + 1/2, 1/2*I*sqrt(3) + 1/2), 1/2*I)
sage: res = sum((-1)^x/((2*x + 1)*factorial(2*x + 1)), x, 0, oo)
sage: res  # not tested - depends on maxima version
hypergeometric((1/2,), (3/2, 3/2), -1/4)
sage: res in [hypergeometric((1/2,), (3/2, 3/2), -1/4), sin_integral(1)]
True
```

Simplification (note that `simplify_full` does not yet call `simplify_hypergeometric`):

```
sage: hypergeometric([-2], [], x).simplify_hypergeometric()
x^2 - 2*x + 1
sage: hypergeometric([], [], x).simplify_hypergeometric()
e^x
sage: a = hypergeometric((hypergeometric((), (), x),), (),
....:                    hypergeometric((), (), x))
sage: a.simplify_hypergeometric()
1/((-e^x + 1)^e^x)
sage: a.simplify_hypergeometric(algorithm='sage')
1/((-e^x + 1)^e^x)
```

Equality testing:

```
sage: bool(hypergeometric([], [], x).derivative(x) ==
....:      hypergeometric([], [], x))  # diff(e^x, x) == e^x
True
sage: bool(hypergeometric([], [], x) == hypergeometric([], [1], x))
False
```

Computing terms and series:

```
sage: # needs sage.symbolic
sage: var('z')
z
sage: hypergeometric([], [], z).series(z, 0)
Order(1)
sage: hypergeometric([], [], z).series(z, 1)
1 + Order(z)
sage: hypergeometric([], [], z).series(z, 2)
1 + 1*z + Order(z^2)
sage: hypergeometric([], [], z).series(z, 3)
1 + 1*z + 1/2*z^2 + Order(z^3)

sage: # needs sage.symbolic
sage: hypergeometric([-2], [], z).series(z, 3)
1 + (-2)*z + 1*z^2
sage: hypergeometric([-2], [], z).series(z, 6)
1 + (-2)*z + 1*z^2
sage: hypergeometric([-2], [], z).series(z, 6).is_terminating_series()
True
sage: hypergeometric([-2], [], z).series(z, 2)
1 + (-2)*z + Order(z^2)
sage: hypergeometric([-2], [], z).series(z, 2).is_terminating_series()
False
```

(continues on next page)

(continued from previous page)

```
sage: hypergeometric([1], [], z).series(z, 6)                                  #␣
↪needs sage.symbolic
1 + 1*z + 1*z^2 + 1*z^3 + 1*z^4 + 1*z^5 + Order(z^6)
sage: hypergeometric([], [1/2], -z^2/4).series(z, 11)                         #␣
↪needs sage.symbolic
1 + (-1/2)*z^2 + 1/24*z^4 + (-1/720)*z^6 + 1/40320*z^8 +...
(-1/3628800)*z^10 + Order(z^11)

sage: hypergeometric([1], [5], x).series(x, 5)
1 + 1/5*x + 1/30*x^2 + 1/210*x^3 + 1/1680*x^4 + Order(x^5)

sage: sum(hypergeometric([1, 2], [3], 1/3).terms(6)).n()                      #␣
↪needs sage.symbolic
1.29788359788360
sage: hypergeometric([1, 2], [3], 1/3).n()                                    #␣
↪needs sage.symbolic
1.29837194594696
sage: hypergeometric([], [], x).series(x, 20)(x=1).n() == e.n()
True
```

Plotting:

```
sage: # needs sage.symbolic
sage: f(x) = hypergeometric([1, 1], [3, 3, 3], x)
sage: plot(f, x, -30, 30)                                                     #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: g(x) = hypergeometric([x], [], 2)
sage: complex_plot(g, (-1, 1), (-1, 1))
Graphics object consisting of 1 graphics primitive
```

Numeric evaluation:

```
sage: # needs sage.symbolic
sage: hypergeometric([1], [], 1/10).n()  # geometric series
1.11111111111111
sage: hypergeometric([], [], 1).n()  # e
2.71828182845905
sage: hypergeometric([], [], 3., hold=True)
hypergeometric((), (), 3.00000000000000)
sage: hypergeometric([1, 2, 3], [4, 5, 6], 1/2).n()
1.02573619590134
sage: hypergeometric([1, 2, 3], [4, 5, 6], 1/2).n(digits=30)
1.02573619590133865036584139535
sage: hypergeometric([5 - 3*I], [3/2, 2 + I, sqrt(2)], 4 + I).n()
5.52605111678803 - 7.86331357527540*I
sage: hypergeometric((10, 10), (50,), 2.)
-1705.75733163554 - 356.749986056024*I
```

Conversions:

```
sage: maxima(hypergeometric([1, 1, 1], [3, 3, 3], x))                         #␣
↪needs sage.symbolic
hypergeometric([1,1,1],[3,3,3],_SAGE_VAR_x)
sage: hypergeometric((5, 4), (4, 4), 3)._sympy_()                             #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
hyper((5, 4), (4, 4), 3)
sage: hypergeometric((5, 4), (4, 4), 3)._mathematica_init_()             #␣
↪needs sage.symbolic
'HypergeometricPFQ[{5,4},{4,4},3]'
```

Arbitrary level of nesting for conversions:

```
sage: maxima(nest(lambda y: hypergeometric([y], [], x), 3, 1))           #␣
↪needs sage.symbolic
1/(1-_SAGE_VAR_x)^(1/(1-_SAGE_VAR_x)^(1/(1-_SAGE_VAR_x)))
sage: maxima(nest(lambda y: hypergeometric([y], [3], x), 3, 1))._sage_()  #␣
↪needs sage.symbolic
hypergeometric((hypergeometric((hypergeometric((1,), (3,), x),), (3,),...
x),), (3,), x)
sage: nest(lambda y: hypergeometric([y], [], x), 3, 1)._mathematica_init_()  #␣
↪needs sage.symbolic
'HypergeometricPFQ[{HypergeometricPFQ[{HypergeometricPFQ[{1},{},x]},...
```

The confluent hypergeometric functions can arise as solutions to second-order differential equations (example from here):

```
sage: var('m')                                                            #␣
↪needs sage.symbolic
m
sage: y = function('y')(x)                                                #␣
↪needs sage.symbolic
sage: desolve(diff(y, x, 2) + 2*x*diff(y, x) - 4*m*y, y,                  #␣
↪needs sage.symbolic
....:         contrib_ode=true, ivar=x)
[y(x) == _K1*hypergeometric_M(-m, 1/2, -x^2) +...
 _K2*hypergeometric_U(-m, 1/2, -x^2)]
```

Series expansions of confluent hypergeometric functions:

```
sage: hypergeometric_M(2, 2, x).series(x, 3)                              #␣
↪needs sage.symbolic
1 + 1*x + 1/2*x^2 + Order(x^3)
sage: hypergeometric_U(2, 2, x).series(x == 3, 100).subs(x=1).n()         #␣
↪needs sage.symbolic
0.403652637676806
sage: hypergeometric_U(2, 2, 1).n()                                       #␣
↪needs mpmath
0.403652637676806
```

class `sage.functions.hypergeometric.`**`Hypergeometric`**

Bases: `BuiltinFunction`

Represent a (formal) generalized infinite hypergeometric series.

It is defined as

$$ {}_pF_q(a_1,\ldots,a_p;b_1,\ldots,b_q;z) = \sum_{n=0}^{\infty} \frac{(a_1)_n\cdots(a_p)_n}{(b_1)_n\cdots(b_q)_n}\,\frac{z^n}{n!}, $$

where $(x)_n$ is the rising factorial.

class EvaluationMethods

Bases: `object`

deflated(*a*, *b*, *z*)

Rewrite as a linear combination of functions of strictly lower degree by eliminating all parameters `a[i]` and `b[j]` such that `a[i]` = `b[i]` + `m` for nonnegative integer `m`.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x = hypergeometric([6, 1], [3, 4, 5], 10)
sage: y = x.deflated()
sage: y
1/252*hypergeometric((4,), (7, 8), 10)
 + 1/12*hypergeometric((3,), (6, 7), 10)
 + 1/2*hypergeometric((2,), (5, 6), 10)
 + hypergeometric((1,), (4, 5), 10)
sage: x.n(); y.n()
2.87893612686782
2.87893612686782

sage: # needs sage.symbolic
sage: x = hypergeometric([6, 7], [3, 4, 5], 10)
sage: y = x.deflated()
sage: y
25/27216*hypergeometric((), (11,), 10)
 + 25/648*hypergeometric((), (10,), 10)
 + 265/504*hypergeometric((), (9,), 10)
 + 181/63*hypergeometric((), (8,), 10)
 + 19/3*hypergeometric((), (7,), 10)
 + 5*hypergeometric((), (6,), 10)
 + hypergeometric((), (5,), 10)
sage: x.n(); y.n()
63.0734110716969
63.0734110716969
```

eliminate_parameters(*a*, *b*, *z*)

Eliminate repeated parameters by pairwise cancellation of identical terms in `a` and `b`.

EXAMPLES:

```
sage: hypergeometric([1, 1, 2, 5], [5, 1, 4],                               #␣
↪needs sage.symbolic
....:                1/2).eliminate_parameters()
hypergeometric((1, 2), (4,), 1/2)
sage: hypergeometric([x], [x], x).eliminate_parameters()
hypergeometric((), (), x)
sage: hypergeometric((5, 4), (4, 4), 3).eliminate_parameters()             #␣
↪needs sage.symbolic
hypergeometric((5,), (4,), 3)
```

is_absolutely_convergent(*a*, *b*, *z*)

Determine whether `self` converges absolutely as an infinite series. `False` is returned if not all terms are finite.

EXAMPLES:

Degree giving infinite radius of convergence:

```
sage: hypergeometric([2, 3], [4, 5],                                        #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
....:                6).is_absolutely_convergent()
True
sage: hypergeometric([2, 3], [-4, 5],                                          #␣
↪needs sage.symbolic
....:                6).is_absolutely_convergent()  # undefined
False
sage: (hypergeometric([2, 3], [-4, 5], Infinity)                               #␣
↪needs sage.symbolic
....:  .is_absolutely_convergent())  # undefined
False
```

Ordinary geometric series (unit radius of convergence):

```
sage: # needs sage.symbolic
sage: hypergeometric([1], [], 1/2).is_absolutely_convergent()
True
sage: hypergeometric([1], [], 2).is_absolutely_convergent()
False
sage: hypergeometric([1], [], 1).is_absolutely_convergent()
False
sage: hypergeometric([1], [], -1).is_absolutely_convergent()
False
sage: hypergeometric([1], [], -1).n()  # Sum still exists
0.500000000000000
```

Degree $p = q + 1$ (unit radius of convergence):

```
sage: # needs sage.symbolic
sage: hypergeometric([2, 3], [4], 6).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [4], 1).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [5], 1).is_absolutely_convergent()
False
sage: hypergeometric([2, 3], [6], 1).is_absolutely_convergent()
True
sage: hypergeometric([-2, 3], [4],
....:                5).is_absolutely_convergent()
True
sage: hypergeometric([2, -3], [4],
....:                5).is_absolutely_convergent()
True
sage: hypergeometric([2, -3], [-4],
....:                5).is_absolutely_convergent()
True
sage: hypergeometric([2, -3], [-1],
....:                5).is_absolutely_convergent()
False
```

Degree giving zero radius of convergence:

```
sage: hypergeometric([1, 2, 3], [4],                                           #␣
↪needs sage.symbolic
....:                2).is_absolutely_convergent()
False
sage: hypergeometric([1, 2, 3], [4],                                           #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
....:                   1/2).is_absolutely_convergent()
False
sage: (hypergeometric([1, 2, -3], [4], 1/2)                            #␣
↪needs sage.symbolic
....:  .is_absolutely_convergent())   # polynomial
True
```

is_terminating(*a*, *b*, *z*)

Determine whether the series represented by `self` terminates after a finite number of terms.

This happens if any of the numerator parameters are nonnegative integers (with no preceding nonnegative denominator parameters), or $z = 0$.

If terminating, the series represents a polynomial of z.

EXAMPLES:

```
sage: hypergeometric([1, 2], [3, 4], x).is_terminating()
False
sage: hypergeometric([1, -2], [3, 4], x).is_terminating()
True
sage: hypergeometric([1, -2], [], x).is_terminating()
True
```

is_termwise_finite(*a*, *b*, *z*)

Determine whether all terms of `self` are finite.

Any infinite terms or ambiguous terms beyond the first zero, if one exists, are ignored.

Ambiguous cases (where a term is the product of both zero and an infinity) are not considered finite.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: hypergeometric([2], [3, 4], 5).is_termwise_finite()
True
sage: hypergeometric([2], [-3, 4], 5).is_termwise_finite()
False
sage: hypergeometric([-2], [-3, 4], 5).is_termwise_finite()
True
sage: hypergeometric([-3], [-3, 4],
....:                   5).is_termwise_finite()   # ambiguous
False

sage: # needs sage.symbolic
sage: hypergeometric([0], [-1], 5).is_termwise_finite()
True
sage: hypergeometric([0], [0],
....:                   5).is_termwise_finite()   # ambiguous
False
sage: hypergeometric([1], [2], Infinity).is_termwise_finite()
False
sage: (hypergeometric([0], [0], Infinity)
....:  .is_termwise_finite())   # ambiguous
False
sage: (hypergeometric([0], [], Infinity)
....:  .is_termwise_finite())   # ambiguous
False
```

sorted_parameters (*a*, *b*, *z*)

Return with parameters sorted in a canonical order.

EXAMPLES:

```
sage: hypergeometric([2, 1, 3], [5, 4],                                    #␣
↪needs sage.symbolic
....:                1/2).sorted_parameters()
hypergeometric((1, 2, 3), (4, 5), 1/2)
```

terms (*a*, *b*, *z*, *n=None*)

Generate the terms of `self` (optionally only `n` terms).

EXAMPLES:

```
sage: list(hypergeometric([-2, 1], [3, 4], x).terms())
[1, -1/6*x, 1/120*x^2]
sage: list(hypergeometric([-2, 1], [3, 4], x).terms(2))
[1, -1/6*x]
sage: list(hypergeometric([-2, 1], [3, 4], x).terms(0))
[]
```

class sage.functions.hypergeometric.**Hypergeometric_M**

Bases: `BuiltinFunction`

The confluent hypergeometric function of the first kind, $y = M(a, b, z)$, is defined to be the solution to Kummer's differential equation

$$zy'' + (b - z)y' - ay = 0.$$

This is not the same as Kummer's U-hypergeometric function, though it satisfies the same DE that M does.

> **Warning:** In the literature, both are called "Kummer confluent hypergeometric" functions.

EXAMPLES:

```
sage: # needs mpmath
sage: hypergeometric_M(1, 1, 1)
hypergeometric_M(1, 1, 1)
sage: hypergeometric_M(1, 1, 1.)
2.71828182845905
sage: hypergeometric_M(1, 1, 1).n(70)
2.7182818284590452354
sage: hypergeometric_M(1, 1, 1).simplify_hypergeometric()
e
sage: hypergeometric_M(1, 3/2, 1).simplify_hypergeometric()
1/2*sqrt(pi)*erf(1)*e

sage: hypergeometric_M(1, 1/2, x).simplify_hypergeometric()                #␣
↪needs sage.symbolic
(-I*sqrt(pi)*x*erf(I*sqrt(-x))*e^x + sqrt(-x))/sqrt(-x)
```

class EvaluationMethods

Bases: `object`

generalized(a, b, z)

Return as a generalized hypergeometric function.

EXAMPLES:

```
sage: var('a b z')                                                      #␣
↪needs sage.symbolic
(a, b, z)
sage: hypergeometric_M(a, b, z).generalized()                           #␣
↪needs sage.symbolic
hypergeometric((a,), (b,), z)
```

class sage.functions.hypergeometric.**Hypergeometric_U**

Bases: BuiltinFunction

The confluent hypergeometric function of the second kind, $y = U(a, b, z)$, is defined to be the solution to Kummer's differential equation

$$zy'' + (b - z)y' - ay = 0.$$

This satisfies $U(a, b, z) \sim z^{-a}$, as $z \to \infty$, and is sometimes denoted $z^{-a}{}_2F_0(a, 1 + a - b; ; -1/z)$. This is not the same as Kummer's M-hypergeometric function, denoted sometimes as ${}_1F_1(\alpha, \beta, z)$, though it satisfies the same DE that U does.

Warning: In the literature, both are called "Kummer confluent hypergeometric" functions.

EXAMPLES:

```
sage: # needs mpmath
sage: hypergeometric_U(1, 1, 1)
hypergeometric_U(1, 1, 1)
sage: hypergeometric_U(1, 1, 1.)
0.596347362323194
sage: hypergeometric_U(1, 1, 1).n(70)
0.59634736232319407434
sage: hypergeometric_U(10^4, 1/3, 1).n()                                #␣
↪needs sage.libs.pari
6.60377008885811e-35745
sage: hypergeometric_U(1, 2, 2).simplify_hypergeometric()
1/2

sage: hypergeometric_U(2 + I, 2, 1).n()                                 #␣
↪needs sage.symbolic
0.183481989942099 - 0.458685959185190*I
sage: hypergeometric_U(1, 3, x).simplify_hypergeometric()               #␣
↪needs sage.symbolic
(x + 1)/x^2
```

class EvaluationMethods

Bases: object

generalized(a, b, z)

Return in terms of the generalized hypergeometric function.

EXAMPLES:

```
sage: var('a b z')                                                    #␣
↪needs sage.symbolic
(a, b, z)
sage: hypergeometric_U(a, b, z).generalized()                         #␣
↪needs sage.symbolic
hypergeometric((a, a - b + 1), (), -1/z)/z^a
sage: hypergeometric_U(1, 3, 1/2).generalized()                       #␣
↪needs mpmath
2*hypergeometric((1, -1), (), -2)
sage: hypergeometric_U(3, I, 2).generalized()                         #␣
↪needs sage.symbolic
1/8*hypergeometric((3, -I + 4), (), -1/2)
```

sage.functions.hypergeometric.**closed_form**(*hyp*)

Try to evaluate `hyp` in closed form using elementary (and other simple) functions.

It may be necessary to call `Hypergeometric.deflated()` first to find some closed forms.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.functions.hypergeometric import closed_form
sage: var('a b c z')
(a, b, c, z)
sage: closed_form(hypergeometric([1], [], 1 + z))
-1/z
sage: closed_form(hypergeometric([], [], 1 + z))
e^(z + 1)
sage: closed_form(hypergeometric([], [1/2], 4))
cosh(4)
sage: closed_form(hypergeometric([], [3/2], 4))
1/4*sinh(4)
sage: closed_form(hypergeometric([], [5/2], 4))
3/16*cosh(4) - 3/64*sinh(4)
sage: closed_form(hypergeometric([], [-3/2], 4))
19/3*cosh(4) - 4*sinh(4)
sage: closed_form(hypergeometric([-3, 1], [var('a')], z))
-3*z/a + 6*z^2/((a + 1)*a) - 6*z^3/((a + 2)*(a + 1)*a) + 1
sage: closed_form(hypergeometric([-3, 1/3], [-4], z))
7/162*z^3 + 1/9*z^2 + 1/4*z + 1
sage: closed_form(hypergeometric([], [], z))
e^z
sage: closed_form(hypergeometric([a], [], z))
1/((-z + 1)^a)
sage: closed_form(hypergeometric([1, 1, 2], [1, 1], z))
(z - 1)^(-2)
sage: closed_form(hypergeometric([2, 3], [1], x))
-1/(x - 1)^3 + 3*x/(x - 1)^4
sage: closed_form(hypergeometric([1/2], [3/2], -5))
1/10*sqrt(5)*sqrt(pi)*erf(sqrt(5))
sage: closed_form(hypergeometric([2], [5], 3))
4
sage: closed_form(hypergeometric([2], [5], 5))
48/625*e^5 + 612/625
sage: closed_form(hypergeometric([1/2, 7/2], [3/2], z))
1/5*z^2/(-z + 1)^(5/2) + 2/3*z/(-z + 1)^(3/2) + 1/sqrt(-z + 1)
sage: closed_form(hypergeometric([1/2, 1], [2], z))
-2*(sqrt(-z + 1) - 1)/z
```

(continues on next page)

(continued from previous page)

```
sage: closed_form(hypergeometric([1, 1], [2], z))
-log(-z + 1)/z
sage: closed_form(hypergeometric([1, 1], [3], z))
-2*((z - 1)*log(-z + 1)/z - 1)/z
sage: closed_form(hypergeometric([1, 1, 1], [2, 2], x))
hypergeometric((1, 1, 1), (2, 2), x)
```

sage.functions.hypergeometric.**rational_param_as_tuple**(x)

Utility function for converting rational ${}_pF_q$ parameters to tuples (which mpmath handles more efficiently).

EXAMPLES:

```
sage: from sage.functions.hypergeometric import rational_param_as_tuple
sage: rational_param_as_tuple(1/2)
(1, 2)
sage: rational_param_as_tuple(3)
3
sage: rational_param_as_tuple(pi)                                    #␣
↪needs sage.symbolic
pi
```

1.12 Jacobi elliptic functions

This module implements the 12 Jacobi elliptic functions, along with their inverses and the Jacobi amplitude function.

Jacobi elliptic functions can be thought of as generalizations of both ordinary and hyperbolic trig functions. There are twelve Jacobian elliptic functions. Each of the twelve corresponds to an arrow drawn from one corner of a rectangle to another.

```
n ------------------- d
|                     |
|                     |
|                     |
s ------------------- c
```

Each of the corners of the rectangle are labeled, by convention, `s`, `c`, `d`, and `n`. The rectangle is understood to be lying on the complex plane, so that `s` is at the origin, `c` is on the real axis, and `n` is on the imaginary axis. The twelve Jacobian elliptic functions are then pq(x), where `p` and `q` are one of the letters `s`, `c`, `d`, `n`.

The Jacobian elliptic functions are then the unique doubly-periodic, meromorphic functions satisfying the following three properties:

1. There is a simple zero at the corner `p`, and a simple pole at the corner `q`.
2. The step from `p` to `q` is equal to half the period of the function pq(x); that is, the function pq(x) is periodic in the direction `pq`, with the period being twice the distance from `p` to `q`. pq(x) is periodic in the other two directions as well, with a period such that the distance from `p` to one of the other corners is a quarter period.
3. If the function pq(x) is expanded in terms of x at one of the corners, the leading term in the expansion has a coefficient of 1. In other words, the leading term of the expansion of pq(x) at the corner `p` is x; the leading term of the expansion at the corner `q` is $1/x$, and the leading term of an expansion at the other two corners is 1.

We can write

$$\mathrm{pq}(x) = \frac{\mathrm{pr}(x)}{\mathrm{qr}(x)}$$

where p, q, and r are any of the letters s, c, d, n, with the understanding that $\mathrm{ss} = \mathrm{cc} = \mathrm{dd} = \mathrm{nn} = 1$.

Let

$$u = \int_0^\phi \frac{d\theta}{\sqrt{1 - m \sin^2 \theta}},$$

then the *Jacobi elliptic function* $\mathrm{sn}(u)$ is given by

$$\mathrm{sn}\, u = \sin \phi$$

and $\mathrm{cn}(u)$ is given by

$$\mathrm{cn}\, u = \cos \phi$$

and

$$\mathrm{dn}\, u = \sqrt{1 - m \sin^2 \phi}.$$

To emphasize the dependence on m, one can write $\mathrm{sn}(u|m)$ for example (and similarly for cn and dn). This is the notation used below.

For a given k with $0 < k < 1$ they therefore are solutions to the following nonlinear ordinary differential equations:

- $\mathrm{sn}\,(x; k)$ solves the differential equations

$$\frac{d^2 y}{dx^2} + (1 + k^2) y - 2 k^2 y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = (1 - y^2)(1 - k^2 y^2).$$

- $\mathrm{cn}(x; k)$ solves the differential equations

$$\frac{d^2 y}{dx^2} + (1 - 2k^2) y + 2 k^2 y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = (1 - y^2)(1 - k^2 + k^2 y^2).$$

- $\mathrm{dn}(x; k)$ solves the differential equations

$$\frac{d^2 y}{dx^2} - (2 - k^2) y + 2 y^3 = 0 \quad \text{and} \quad \left(\frac{dy}{dx}\right)^2 = y^2(1 - k^2 - y^2).$$

  If $K(m)$ denotes the complete elliptic integral of the first kind (named `elliptic_kc` in Sage), the elliptic functions $\mathrm{sn}(x|m)$ and $\mathrm{cn}(x|m)$ have real periods $4K(m)$, whereas $\mathrm{dn}(x|m)$ has a period $2K(m)$. The limit $m \rightarrow 0$ gives $K(0) = \pi/2$ and trigonometric functions: $\mathrm{sn}(x|0) = \sin x$, $\mathrm{cn}(x|0) = \cos x$, $\mathrm{dn}(x|0) = 1$. The limit $m \rightarrow 1$ gives $K(1) \rightarrow \infty$ and hyperbolic functions: $\mathrm{sn}(x|1) = \tanh x$, $\mathrm{cn}(x|1) = \mathrm{sech}\, x$, $\mathrm{dn}(x|1) = \mathrm{sech}\, x$.

REFERENCES:

- Wikipedia article Jacobi%27s_elliptic_functions
- [KS2002]

AUTHORS:

- David Joyner (2006): initial version
- Eviatar Bach (2013): complete rewrite, new numerical evaluation, and addition of the Jacobi amplitude function

class sage.functions.jacobi.**InverseJacobi**(*kind*)

  Bases: `BuiltinFunction`

  Base class for the inverse Jacobi elliptic functions.

class sage.functions.jacobi.**Jacobi**(*kind*)

Bases: BuiltinFunction

Base class for the Jacobi elliptic functions.

class sage.functions.jacobi.**JacobiAmplitude**

Bases: BuiltinFunction

The Jacobi amplitude function $\mathrm{am}(x|m) = \int_0^x \mathrm{dn}(t|m)dt$ for $-K(m) \leq x \leq K(m)$, $F(\mathrm{am}(x|m)|m) = x$.

sage.functions.jacobi.**inverse_jacobi**(*kind*, *x*, *m*, ***kwargs*)

The inverses of the 12 Jacobi elliptic functions. They have the property that

$$\mathrm{pq}(\mathrm{arcpq}(x|m)|m) = \mathrm{pq}(\mathrm{pq}^{-1}(x|m)|m) = x.$$

INPUT:

- kind – a string of the form 'pq', where p, q are in c, d, n, s
- x – a real number
- m – a real number; note that $m = k^2$, where k is the elliptic modulus

EXAMPLES:

```
sage: jacobi('dn', inverse_jacobi('dn', 3, 0.4), 0.4)                      #␣
↪needs mpmath
3.00000000000000
sage: inverse_jacobi('dn', 10, 1/10).n(digits=50)                          #␣
↪needs mpmath
2.4777736267904273296523691232988240759001423661683*I
sage: inverse_jacobi_dn(x, 1)                                              #␣
↪needs sage.symbolic
arcsech(x)
sage: inverse_jacobi_dn(1, 3)                                              #␣
↪needs mpmath
0

sage: # needs sage.symbolic
sage: m = var('m')
sage: z = inverse_jacobi_dn(x, m).series(x, 4).subs(x=0.1, m=0.7)
sage: jacobi_dn(z, 0.7)
0.0999892750039819...
sage: inverse_jacobi_nd(x, 1)
arccosh(x)

sage: # needs mpmath
sage: inverse_jacobi_nd(1, 2)
0
sage: inverse_jacobi_ns(10^-5, 3).n()
5.77350269202456e-6 + 1.17142008414677*I
sage: jacobi('sn', 1/2, 1/2)
jacobi_sn(1/2, 1/2)
sage: jacobi('sn', 1/2, 1/2).n()
0.470750473655657
sage: inverse_jacobi('sn', 0.47, 1/2)
0.499098231322220
sage: inverse_jacobi('sn', 0.4707504, 0.5)
0.499999911466555
sage: P = plot(inverse_jacobi('sn', x, 0.5), 0, 1)                         #␣
↪needs sage.plot
```

`sage.functions.jacobi.`**`inverse_jacobi_f`**`(`*kind*, *x*, *m*`)`

Internal function for numerical evaluation of a continuous complex branch of each inverse Jacobi function, as described in [Tee1997]. Only accepts real arguments.

`sage.functions.jacobi.`**`jacobi`**`(`*kind*, *z*, *m*, ***kwargs*`)`

The 12 Jacobi elliptic functions.

INPUT:

- `kind` – a string of the form `'pq'`, where `p`, `q` are in `c`, `d`, `n`, `s`
- `z` – a complex number
- `m` – a complex number; note that $m = k^2$, where k is the elliptic modulus

EXAMPLES:

```
sage: # needs mpmath
sage: jacobi('sn', 1, 1)
tanh(1)
sage: jacobi('cd', 1, 1/2)
jacobi_cd(1, 1/2)
sage: RDF(jacobi('cd', 1, 1/2))
0.7240097216593705
sage: (RDF(jacobi('cn', 1, 1/2)), RDF(jacobi('dn', 1, 1/2)),
....:  RDF(jacobi('cn', 1, 1/2) / jacobi('dn', 1, 1/2)))
(0.5959765676721407, 0.8231610016315962, 0.7240097216593705)

sage: jsn = jacobi('sn', x, 1)                                              #␣
↪needs sage.symbolic
sage: P = plot(jsn, 0, 1)                                                   #␣
↪needs sage.plot sage.symbolic
```

`sage.functions.jacobi.`**`jacobi_am_f`**`(`*x*, *m*`)`

Internal function for numeric evaluation of the Jacobi amplitude function for real arguments. Procedure described in [Eh2013].

1.13 Airy functions

This module implements Airy functions and their generalized derivatives. It supports symbolic functionality through Maxima and numeric evaluation through mpmath and scipy.

Airy functions are solutions to the differential equation $f''(x) - xf(x) = 0$.

Four global function symbols are immediately available, please see

- *`airy_ai()`*: for the Airy Ai function
- *`airy_ai_prime()`*: for the first differential of the Airy Ai function
- *`airy_bi()`*: for the Airy Bi function
- ***`airy_bi_prime()`*: for the first differential**
  of the Airy Bi function

AUTHORS:

- Oscar Gerardo Lazo Arjona (2010): initial version
- Douglas McNeil (2012): rewrite

EXAMPLES:

Verify that the Airy functions are solutions to the differential equation:

```
sage: diff(airy_ai(x), x, 2) - x * airy_ai(x)                              #␣
↪needs sage.symbolic
0
sage: diff(airy_bi(x), x, 2) - x * airy_bi(x)                              #␣
↪needs sage.symbolic
0
```

class sage.functions.airy.**FunctionAiryAiGeneral**

Bases: BuiltinFunction

The generalized derivative of the Airy Ai function

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Ai}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Ai}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.functions.airy import airy_ai_general
sage: x, n = var('x n')
sage: airy_ai_general(-2, x)
airy_ai(-2, x)
sage: derivative(airy_ai_general(-2, x), x)
airy_ai(-1, x)
sage: airy_ai_general(n, x)
airy_ai(n, x)
sage: derivative(airy_ai_general(n, x), x)
airy_ai(n + 1, x)
```

class sage.functions.airy.**FunctionAiryAiPrime**

Bases: BuiltinFunction

The derivative of the Airy Ai function; see *airy_ai()* for the full documentation.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, n = var('x n')
sage: airy_ai_prime(x)
airy_ai_prime(x)
sage: airy_ai_prime(0)
-1/3*3^(2/3)/gamma(1/3)
sage: airy_ai_prime(x)._sympy_()                                     #␣
↪needs sympy
airyaiprime(x)
```

class sage.functions.airy.**FunctionAiryAiSimple**

Bases: BuiltinFunction

The class for the Airy Ai function.

EXAMPLES:

```
sage: from sage.functions.airy import airy_ai_simple
sage: f = airy_ai_simple(x); f                                          #␣
↪needs sage.symbolic
airy_ai(x)
sage: airy_ai_simple(x)._sympy_()                                       #␣
↪needs sage.symbolic
airyai(x)
```

class sage.functions.airy.**FunctionAiryBiGeneral**

Bases: BuiltinFunction

The generalized derivative of the Airy Bi function.

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\mathrm{Bi}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \mathrm{Bi}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function

EXAMPLES:

```
sage: # needs sage.symbolic
sage: from sage.functions.airy import airy_bi_general
sage: x, n = var('x n')
sage: airy_bi_general(-2, x)
airy_bi(-2, x)
sage: derivative(airy_bi_general(-2, x), x)
airy_bi(-1, x)
sage: airy_bi_general(n, x)
airy_bi(n, x)
sage: derivative(airy_bi_general(n, x), x)
airy_bi(n + 1, x)
```

class sage.functions.airy.**FunctionAiryBiPrime**

Bases: BuiltinFunction

The derivative of the Airy Bi function; see *airy_bi()* for the full documentation.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, n = var('x n')
sage: airy_bi_prime(x)
airy_bi_prime(x)
sage: airy_bi_prime(0)
3^(1/6)/gamma(1/3)
sage: airy_bi_prime(x)._sympy_()                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sympy
airybiprime(x)
```

class sage.functions.airy.**FunctionAiryBiSimple**

Bases: BuiltinFunction

The class for the Airy Bi function.

EXAMPLES:

```
sage: from sage.functions.airy import airy_bi_simple
sage: f = airy_bi_simple(x); f                                         #␣
↪needs sage.symbolic
airy_bi(x)
sage: f._sympy_()                                                      #␣
↪needs sympy sage.symbolic
airybi(x)
```

sage.functions.airy.**airy_ai**(*alpha*, *x=None*, *hold_derivative=True*, ***kwds*)

The Airy Ai function

The Airy Ai function $\operatorname{Ai}(x)$ is (along with $\operatorname{Bi}(x)$) one of the two linearly independent standard solutions to the Airy differential equation $f''(x) - xf(x) = 0$. It is defined by the initial conditions:

$$\operatorname{Ai}(0) = \frac{1}{2^{2/3}\Gamma\left(\frac{2}{3}\right)},$$

$$\operatorname{Ai}'(0) = -\frac{1}{2^{1/3}\Gamma\left(\frac{1}{3}\right)}.$$

Another way to define the Airy Ai function is:

$$\operatorname{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos\left(\frac{1}{3}t^3 + xt\right) dt.$$

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\operatorname{Ai}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \operatorname{Ai}(z)$$

$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function
- hold_derivative – Whether or not to stop from returning higher derivatives in terms of $\operatorname{Ai}(x)$ and $\operatorname{Ai}'(x)$

See also:

airy_bi()

EXAMPLES:

```
sage: n, x = var('n x')                                                #␣
↪needs sage.symbolic
sage: airy_ai(x)                                                       #␣
↪needs sage.symbolic
airy_ai(x)
```

It can return derivatives or integrals:

```
sage: # needs sage.symbolic
sage: airy_ai(2, x)
airy_ai(2, x)
sage: airy_ai(1, x, hold_derivative=False)
airy_ai_prime(x)
sage: airy_ai(2, x, hold_derivative=False)
x*airy_ai(x)
sage: airy_ai(-2, x, hold_derivative=False)
airy_ai(-2, x)
sage: airy_ai(n, x)
airy_ai(n, x)
```

It can be evaluated symbolically or numerically for real or complex values:

```
sage: airy_ai(0)                                                          #␣
↪needs sage.symbolic
1/3*3^(1/3)/gamma(2/3)
sage: airy_ai(0.0)                                                        #␣
↪needs mpmath
0.355028053887817
sage: airy_ai(I)                                                          #␣
↪needs sage.symbolic
airy_ai(I)
sage: airy_ai(1.0*I)                                                      #␣
↪needs sage.symbolic
0.331493305432141 - 0.317449858968444*I
```

The functions can be evaluated numerically either using mpmath. which can compute the values to arbitrary precision, and scipy:

```
sage: airy_ai(2).n(prec=100)                                              #␣
↪needs sage.symbolic
0.034924130423274379135322080792
sage: airy_ai(2).n(algorithm='mpmath', prec=100)                          #␣
↪needs sage.symbolic
0.034924130423274379135322080792
sage: airy_ai(2).n(algorithm='scipy')  # rel tol 1e-10                    #␣
↪needs scipy sage.symbolic
0.03492413042327323
```

And the derivatives can be evaluated:

```
sage: airy_ai(1, 0)                                                       #␣
↪needs sage.symbolic
-1/3*3^(2/3)/gamma(1/3)
sage: airy_ai(1, 0.0)                                                     #␣
↪needs mpmath
-0.258819403792807
```

Plots:

```
sage: plot(airy_ai(x), (x, -10, 5)) + plot(airy_ai_prime(x),             #␣
↪needs sage.plot sage.symbolic
....:  (x, -10, 5), color='red')
Graphics object consisting of 2 graphics primitives
```

REFERENCES:

- Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 10"
- Wikipedia article Airy_function

sage.functions.airy.**airy_bi**(*alpha*, *x=None*, *hold_derivative=True*, ***kwds*)

The Airy Bi function

The Airy Bi function $\operatorname{Bi}(x)$ is (along with $\operatorname{Ai}(x)$) one of the two linearly independent standard solutions to the Airy differential equation $f''(x) - xf(x) = 0$. It is defined by the initial conditions:

$$\operatorname{Bi}(0) = \frac{1}{3^{1/6}\Gamma\left(\frac{2}{3}\right)},$$
$$\operatorname{Bi}'(0) = \frac{3^{1/6}}{\Gamma\left(\frac{1}{3}\right)}.$$

Another way to define the Airy Bi function is:

$$\operatorname{Bi}(x) = \frac{1}{\pi}\int_0^\infty \left[\exp\left(xt - \frac{t^3}{3}\right) + \sin\left(xt + \frac{1}{3}t^3\right)\right] dt.$$

INPUT:

- alpha – Return the α-th order fractional derivative with respect to z. For $\alpha = n = 1, 2, 3, \ldots$ this gives the derivative $\operatorname{Bi}^{(n)}(z)$, and for $\alpha = -n = -1, -2, -3, \ldots$ this gives the n-fold iterated integral.

$$f_0(z) = \operatorname{Bi}(z)$$
$$f_n(z) = \int_0^z f_{n-1}(t)dt$$

- x – The argument of the function
- hold_derivative – Whether or not to stop from returning higher derivatives in terms of $\operatorname{Bi}(x)$ and $\operatorname{Bi}'(x)$

See also:

airy_ai()

EXAMPLES:

```
sage: n, x = var('n x')                                                    #␣
↪needs sage.symbolic
sage: airy_bi(x)                                                           #␣
↪needs sage.symbolic
airy_bi(x)
```

It can return derivatives or integrals:

```
sage: # needs sage.symbolic
sage: airy_bi(2, x)
airy_bi(2, x)
sage: airy_bi(1, x, hold_derivative=False)
airy_bi_prime(x)
sage: airy_bi(2, x, hold_derivative=False)
x*airy_bi(x)
sage: airy_bi(-2, x, hold_derivative=False)
airy_bi(-2, x)
sage: airy_bi(n, x)
airy_bi(n, x)
```

It can be evaluated symbolically or numerically for real or complex values:

```
sage: airy_bi(0)                                                          #␣
↪needs sage.symbolic
1/3*3^(5/6)/gamma(2/3)
sage: airy_bi(0.0)                                                        #␣
↪needs mpmath
0.614926627446001
sage: airy_bi(I)                                                          #␣
↪needs sage.symbolic
airy_bi(I)
sage: airy_bi(1.0*I)                                                      #␣
↪needs sage.symbolic
0.648858208330395 + 0.344958634768048*I
```

The functions can be evaluated numerically using mpmath, which can compute the values to arbitrary precision, and scipy:

```
sage: airy_bi(2).n(prec=100)                                              #␣
↪needs sage.symbolic
3.2980949999782147102806044252
sage: airy_bi(2).n(algorithm='mpmath', prec=100)                          #␣
↪needs sage.symbolic
3.2980949999782147102806044252
sage: airy_bi(2).n(algorithm='scipy')  # rel tol 1e-10                    #␣
↪needs scipy sage.symbolic
3.2980949999782134
```

And the derivatives can be evaluated:

```
sage: airy_bi(1, 0)                                                       #␣
↪needs sage.symbolic
3^(1/6)/gamma(1/3)
sage: airy_bi(1, 0.0)                                                     #␣
↪needs mpmath
0.448288357353826
```

Plots:

```
sage: plot(airy_bi(x), (x, -10, 5)) + plot(airy_bi_prime(x),             #␣
↪needs sage.plot sage.symbolic
....:  (x, -10, 5), color='red')
Graphics object consisting of 2 graphics primitives
```

REFERENCES:

- Abramowitz, Milton; Stegun, Irene A., eds. (1965), “Chapter 10”
- Wikipedia article Airy_function

1.14 Bessel functions

This module provides symbolic Bessel and Hankel functions, and their spherical versions. These functions use the mpmath library for numerical evaluation and Maxima, GiNaC, Pynac for symbolics.

The main objects which are exported from this module are:

- `bessel_J(n, x)` – The Bessel J function
- `bessel_Y(n, x)` – The Bessel Y function
- `bessel_I(n, x)` – The Bessel I function
- `bessel_K(n, x)` – The Bessel K function
- `Bessel(...)` – A factory function for producing Bessel functions of various kinds and orders
- `hankel1(nu, z)` – The Hankel function of the first kind
- `hankel2(nu, z)` – The Hankel function of the second kind
- `struve_H(nu, z)` – The Struve function
- `struve_L(nu, z)` – The modified Struve function
- `spherical_bessel_J(n, z)` – The Spherical Bessel J function
- `spherical_bessel_Y(n, z)` – The Spherical Bessel J function
- `spherical_hankel1(n, z)` – The Spherical Hankel function of the first kind
- `spherical_hankel2(n, z)` – The Spherical Hankel function of the second kind
- Bessel functions, first defined by the Swiss mathematician Daniel Bernoulli and named after Friedrich Bessel, are canonical solutions y(x) of Bessel's differential equation:

  $$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + \left(x^2 - \nu^2\right) y = 0,$$

  for an arbitrary complex number ν (the order).
- In this module, J_ν denotes the unique solution of Bessel's equation which is non-singular at $x = 0$. This function is known as the Bessel Function of the First Kind. This function also arises as a special case of the hypergeometric function ${}_0F_1$:

  $$J_\nu(x) = \frac{x^n}{2^\nu \Gamma(\nu + 1)} {}_0F_1(\nu + 1, -\frac{x^2}{4}).$$

- The second linearly independent solution to Bessel's equation (which is singular at $x = 0$) is denoted by Y_ν and is called the Bessel Function of the Second Kind:

  $$Y_\nu(x) = \frac{J_\nu(x) \cos(\pi \nu) - J_{-\nu}(x)}{\sin(\pi \nu)}.$$

- There are also two commonly used combinations of the Bessel J and Y Functions. The Bessel I Function, or the Modified Bessel Function of the First Kind, is defined by:

  $$I_\nu(x) = i^{-\nu} J_\nu(ix).$$

  The Bessel K Function, or the Modified Bessel Function of the Second Kind, is defined by:

  $$K_\nu(x) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(x) - I_n(x)}{\sin(\pi \nu)}.$$

  We should note here that the above formulas for Bessel Y and K functions should be understood as limits when ν is an integer.

- It follows from Bessel's differential equation that the derivative of $J_n(x)$ with respect to x is:

$$\frac{d}{dx}J_n(x) = \frac{1}{x^n}\left(x^n J_{n-1}(x) - nx^{n-1}J_n(z)\right)$$

- Another important formulation of the two linearly independent solutions to Bessel's equation are the Hankel functions $H_\nu^{(1)}(x)$ and $H_\nu^{(2)}(x)$, defined by:

$$H_\nu^{(1)}(x) = J_\nu(x) + iY_\nu(x)$$

$$H_\nu^{(2)}(x) = J_\nu(x) - iY_\nu(x)$$

  where i is the imaginary unit (and J_* and Y_* are the usual J- and Y-Bessel functions). These linear combinations are also known as Bessel functions of the third kind; they are also two linearly independent solutions of Bessel's differential equation. They are named for Hermann Hankel.

- When solving for separable solutions of Laplace's equation in spherical coordinates, the radial equation has the form:

$$x^2\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + [x^2 - n(n+1)]y = 0.$$

  The spherical Bessel functions j_n and y_n, are two linearly independent solutions to this equation. They are related to the ordinary Bessel functions J_n and Y_n by:

$$j_n(x) = \sqrt{\frac{\pi}{2x}}J_{n+1/2}(x),$$

$$y_n(x) = \sqrt{\frac{\pi}{2x}}Y_{n+1/2}(x) = (-1)^{n+1}\sqrt{\frac{\pi}{2x}}J_{-n-1/2}(x).$$

EXAMPLES:

Evaluate the Bessel J function symbolically and numerically:

```
sage: # needs sage.symbolic
sage: bessel_J(0, x)
bessel_J(0, x)
sage: bessel_J(0, 0)
1
sage: bessel_J(0, x).diff(x)
-1/2*bessel_J(1, x) + 1/2*bessel_J(-1, x)
sage: N(bessel_J(0, 0), digits=20)
1.0000000000000000000
sage: find_root(bessel_J(0,x), 0, 5)                                    ␣
↪   # needs scipy
2.404825557695773
```

Plot the Bessel J function:

```
sage: f(x) = Bessel(0)(x); f                                            ␣
↪   # needs sage.symbolic
x |--> bessel_J(0, x)
sage: plot(f, (x, 1, 10))                                               ␣
↪   # needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

Visualize the Bessel Y function on the complex plane (set plot_points to a higher value to get more detail):

```
sage: complex_plot(bessel_Y(0, x), (-5, 5), (-5, 5), plot_points=20)              ␣
↪   # needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

Evaluate a combination of Bessel functions:

```
sage: # needs sage.symbolic
sage: f(x) = bessel_J(1, x) - bessel_Y(0, x)
sage: f(pi)
bessel_J(1, pi) - bessel_Y(0, pi)
sage: f(pi).n()
-0.0437509653365599
sage: f(pi).n(digits=50)
-0.043750965336559909054985168023342675387737118378169
```

Symbolically solve a second order differential equation with initial conditions $y(1) = a$ and $y'(1) = b$ in terms of Bessel functions:

```
sage: # needs sage.symbolic
sage: y = function('y')(x)
sage: a, b = var('a, b')
sage: diffeq = x^2*diff(y,x,x) + x*diff(y,x) + x^2*y == 0
sage: f = desolve(diffeq, y, [1, a, b]); f
(a*bessel_Y(1, 1) + b*bessel_Y(0, 1))*bessel_J(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1)) -
(a*bessel_J(1, 1) + b*bessel_J(0, 1))*bessel_Y(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1))
```

For more examples, see the docstring for `Bessel()`.

AUTHORS:

- Some of the documentation here has been adapted from David Joyner's original documentation of Sage's special functions module (2006).

REFERENCES:

- [AS-Bessel]
- [AS-Spherical]
- [AS-Struve]
- [DLMF-Bessel]
- [DLMF-Struve]
- [WP-Bessel]
- [WP-Struve]

`sage.functions.bessel.`**`Bessel`**`(`*`*args`*`, `*`**kwds`*`)`

A function factory that produces symbolic I, J, K, and Y Bessel functions. There are several ways to call this function:

- `Bessel(order, type)`
- `Bessel(order)` – type defaults to `'J'`
- `Bessel(order, typ=T)`
- `Bessel(typ=T)` – order is unspecified, this is a 2-parameter function

- `Bessel()` – order is unspecified, type is `'J'`

where `order` can be any integer and `T` must be one of the strings `'I'`, `'J'`, `'K'`, or `'Y'`.

See the EXAMPLES below.

EXAMPLES:

Construction of Bessel functions with various orders and types:

```
sage: Bessel()
bessel_J
sage: Bessel(typ='K')
bessel_K

sage: # needs sage.symbolic
sage: Bessel(1)(x)
bessel_J(1, x)
sage: Bessel(1, 'Y')(x)
bessel_Y(1, x)
sage: Bessel(-2, 'Y')(x)
bessel_Y(-2, x)
sage: Bessel(0, typ='I')(x)
bessel_I(0, x)
```

Evaluation:

```
sage: f = Bessel(1)
sage: f(3.0)                                                                    #␣
↪needs mpmath
0.339058958525936

sage: # needs sage.symbolic
sage: f(3)
bessel_J(1, 3)
sage: f(3).n(digits=50)
0.33905895852593645892551459720647889697308041819801
sage: g = Bessel(typ='J')
sage: g(1,3)
bessel_J(1, 3)
sage: g(2, 3+I).n()
0.634160370148554 + 0.0253384000032695*I
sage: abs(numerical_integral(1/pi*cos(3*sin(x)), 0.0, pi)[0]
....:     - Bessel(0, 'J')(3.0)) < 1e-15
True
```

Symbolic calculus:

```
sage: f(x) = Bessel(0, 'J')(x)                                                  #␣
↪needs sage.symbolic
sage: derivative(f, x)                                                          #␣
↪needs sage.symbolic
x |--> -1/2*bessel_J(1, x) + 1/2*bessel_J(-1, x)
sage: derivative(f, x, x)                                                       #␣
↪needs sage.symbolic
x |--> 1/4*bessel_J(2, x) - 1/2*bessel_J(0, x) + 1/4*bessel_J(-2, x)
```

Verify that J_0 satisfies Bessel's differential equation numerically using the `test_relation()` method:

```
sage: y = bessel_J(0, x)                                                    #␣
↪needs sage.symbolic
sage: diffeq = x^2*derivative(y,x,x) + x*derivative(y,x) + x^2*y == 0       #␣
↪needs sage.symbolic
sage: diffeq.test_relation(proof=False)                                     #␣
↪needs sage.symbolic
True
```

Conversion to other systems:

```
sage: # needs sage.symbolic
sage: x,y = var('x,y')
sage: f = Bessel(typ='K')(x,y)
sage: expected = f.derivative(y)
sage: actual = maxima(f).derivative('_SAGE_VAR_y').sage()
sage: bool(actual == expected)
True
```

Compute the particular solution to Bessel's Differential Equation that satisfies $y(1) = 1$ and $y'(1) = 1$, then verify the initial conditions and plot it:

```
sage: # needs sage.symbolic
sage: y = function('y')(x)
sage: diffeq = x^2*diff(y,x,x) + x*diff(y,x) + x^2*y == 0
sage: f = desolve(diffeq, y, [1, 1, 1]); f
(bessel_Y(1, 1) + bessel_Y(0, 1))*bessel_J(0, x)/(bessel_J(0,
1)*bessel_Y(1, 1) - bessel_J(1, 1)*bessel_Y(0, 1)) - (bessel_J(1,
1) + bessel_J(0, 1))*bessel_Y(0, x)/(bessel_J(0, 1)*bessel_Y(1, 1)
- bessel_J(1, 1)*bessel_Y(0, 1))
sage: f.subs(x=1).n()  # numerical verification
1.00000000000000
sage: fp = f.diff(x)
sage: fp.subs(x=1).n()
1.00000000000000

sage: f.subs(x=1).simplify_full()  # symbolic verification                  #␣
↪needs sage.symbolic
1
sage: fp = f.diff(x)                                                        #␣
↪needs sage.symbolic
sage: fp.subs(x=1).simplify_full()                                          #␣
↪needs sage.symbolic
1

sage: plot(f, (x,0,5))                                                      #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

Plotting:

```
sage: f(x) = Bessel(0)(x); f                                                #␣
↪needs sage.symbolic
x |--> bessel_J(0, x)
sage: plot(f, (x, 1, 10))                                                   #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

(continues on next page)

(continued from previous page)

```
sage: plot([Bessel(i, 'J') for i in range(5)], 2, 10)                    #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives

sage: G = Graphics()                                                       #␣
↪needs sage.plot
sage: G += sum(plot(Bessel(i), 0, 4*pi, rgbcolor=hue(sin(pi*i/10)))       #␣
↪needs sage.plot sage.symbolic
....:          for i in range(5))
sage: show(G)                                                              #␣
↪needs sage.plot
```

A recreation of Abramowitz and Stegun Figure 9.1:

```
sage: # needs sage.plot sage.symbolic
sage: G  = plot(Bessel(0, 'J'), 0, 15, color='black')
sage: G += plot(Bessel(0, 'Y'), 0, 15, color='black')
sage: G += plot(Bessel(1, 'J'), 0, 15, color='black', linestyle='dotted')
sage: G += plot(Bessel(1, 'Y'), 0, 15, color='black', linestyle='dotted')
sage: show(G, ymin=-1, ymax=1)
```

class sage.functions.bessel.**Function_Bessel_I**

Bases: BuiltinFunction

The Bessel I function, or the Modified Bessel Function of the First Kind.

DEFINITION:

$$I_\nu(x) = i^{-\nu} J_\nu(ix)$$

EXAMPLES:

```
sage: bessel_I(1.0, 1.0)                                                   #␣
↪needs mpmath
0.565159103992485

sage: # needs sage.symbolic
sage: bessel_I(1, x)
bessel_I(1, x)
sage: n = var('n')
sage: bessel_I(n, x)
bessel_I(n, x)
sage: bessel_I(2, I).n()
-0.114903484931900
```

Examples of symbolic manipulation:

```
sage: # needs sage.symbolic
sage: a = bessel_I(pi, bessel_I(1, I))
sage: N(a, digits=20)
0.00026073272117205890524 - 0.00115289548890805722268*I
sage: f = bessel_I(2, x)
sage: f.diff(x)
1/2*bessel_I(3, x) + 1/2*bessel_I(1, x)
```

Special identities that bessel_I satisfies:

```
sage: # needs sage.symbolic
sage: bessel_I(1/2, x)
sqrt(2)*sqrt(1/(pi*x))*sinh(x)
sage: eq = bessel_I(1/2, x) == bessel_I(0.5, x)
sage: eq.test_relation()
True
sage: bessel_I(-1/2, x)
sqrt(2)*sqrt(1/(pi*x))*cosh(x)
sage: eq = bessel_I(-1/2, x) == bessel_I(-0.5, x)
sage: eq.test_relation()
True
```

Examples of asymptotic behavior:

```
sage: limit(bessel_I(0, x), x=oo)                                          #␣
↪needs sage.symbolic
+Infinity
sage: limit(bessel_I(0, x), x=0)                                           #␣
↪needs sage.symbolic
1
```

High precision and complex valued inputs:

```
sage: bessel_I(0, 1).n(128)                                                #␣
↪needs sage.symbolic
1.2660658777520083355982446252147175376
sage: bessel_I(0, RealField(200)(1))                                       #␣
↪needs sage.rings.real_mpfr
1.2660658777520083355982446252147175376076703113549622068081
sage: bessel_I(0, ComplexField(200)(0.5+I))                                #␣
↪needs sage.symbolic
0.80644357583493619472428518415019222845373366024179916785502
 + 0.22686958987911161141397453401487525043310874687430711021434*I
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_I(1, x), (x, 0, 5), color='blue')                        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_I(1, x), (-5, 5), (-5, 5), plot_points=20)      #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

REFERENCES:

- [AS-Bessel]
- [DLMF-Bessel]
- [WP-Bessel]

class sage.functions.bessel.**Function_Bessel_J**

Bases: BuiltinFunction

The Bessel J Function, denoted by bessel_J(ν, x) or $J_\nu(x)$. As a Taylor series about $x = 0$ it is equal to:

$$J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!\Gamma(k+\nu+1)} \left(\frac{x}{2}\right)^{2k+\nu}$$

The parameter ν is called the order and may be any real or complex number; however, integer and half-integer values are most common. It is defined for all complex numbers x when ν is an integer or greater than zero and it diverges as $x \to 0$ for negative non-integer values of ν.

For integer orders $\nu = n$ there is an integral representation:

$$J_n(x) = \frac{1}{\pi} \int_0^{\pi} \cos(nt - x\sin(t))\, dt$$

This function also arises as a special case of the hypergeometric function ${}_0F_1$:

$$J_\nu(x) = \frac{x^n}{2^\nu \Gamma(\nu+1)} {}_0F_1\left(\nu+1, -\frac{x^2}{4}\right).$$

EXAMPLES:

```
sage: bessel_J(1.0, 1.0)                                                    #␣
↪needs mpmath
0.440050585744933

sage: # needs sage.symbolic
sage: bessel_J(2, I).n(digits=30)
-0.135747669767038281182852569995
sage: bessel_J(1, x)
bessel_J(1, x)
sage: n = var('n')
sage: bessel_J(n, x)
bessel_J(n, x)
```

Examples of symbolic manipulation:

```
sage: # needs sage.symbolic
sage: a = bessel_J(pi, bessel_J(1, I)); a
bessel_J(pi, bessel_J(1, I))
sage: N(a, digits=20)
0.00059023706363796717363 - 0.0026098820470081958110*I
sage: f = bessel_J(2, x)
sage: f.diff(x)
-1/2*bessel_J(3, x) + 1/2*bessel_J(1, x)
```

Comparison to a well-known integral representation of $J_1(1)$:

```
sage: A = numerical_integral(1/pi*cos(x - sin(x)), 0, pi)                   #␣
↪needs sage.symbolic
sage: A[0]  # abs tol 1e-14                                                 #␣
↪needs sage.symbolic
0.44005058574493355
sage: bessel_J(1.0, 1.0) - A[0] < 1e-15                                     #␣
↪needs sage.symbolic
True
```

Integration is supported directly and through Maxima:

```
sage: f = bessel_J(2, x)                                                    #␣
↪needs sage.symbolic
sage: f.integrate(x)                                                        #␣
↪needs sage.symbolic
1/24*x^3*hypergeometric((3/2,), (5/2, 3), -1/4*x^2)
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_J(1,x), (x,0,5), color='blue')                            #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_J(1, x), (-5, 5), (-5, 5), plot_points=20)        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

Check whether the return value is real whenever the argument is real (github issue #10251):

```
sage: bessel_J(5, 1.5) in RR                                                #␣
↪needs mpmath
True
```

REFERENCES:

- [AS-Bessel]
- [DLMF-Bessel]
- [AS-Bessel]

class sage.functions.bessel.**Function_Bessel_K**

Bases: BuiltinFunction

The Bessel K function, or the modified Bessel function of the second kind.

DEFINITION:

$$K_\nu(x) = \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin(\nu \pi)}$$

EXAMPLES:

```
sage: bessel_K(1.0, 1.0)                                                    #␣
↪needs mpmath
0.601907230197235

sage: # needs sage.symbolic
sage: bessel_K(1, x)
bessel_K(1, x)
sage: n = var('n')
sage: bessel_K(n, x)
bessel_K(n, x)
sage: bessel_K(2, I).n()
-2.59288617549120 + 0.180489972066962*I
```

Examples of symbolic manipulation:

```
sage: # needs sage.symbolic
sage: a = bessel_K(pi, bessel_K(1, I)); a
bessel_K(pi, bessel_K(1, I))
sage: N(a, digits=20)
3.8507583115005220156 + 0.068528298579883425456*I
sage: f = bessel_K(2, x)
sage: f.diff(x)
-1/2*bessel_K(3, x) - 1/2*bessel_K(1, x)
sage: bessel_K(1/2, x)
sqrt(1/2)*sqrt(pi)*e^(-x)/sqrt(x)
sage: bessel_K(1/2, -1)
-I*sqrt(1/2)*sqrt(pi)*e
sage: bessel_K(1/2, 1)
sqrt(1/2)*sqrt(pi)*e^(-1)
```

Examples of asymptotic behavior:

```
sage: bessel_K(0, 0.0)                                                  #␣
↪needs mpmath
+infinity
sage: limit(bessel_K(0, x), x=0)                                        #␣
↪needs sage.symbolic
+Infinity
sage: limit(bessel_K(0, x), x=oo)                                       #␣
↪needs sage.symbolic
0
```

High precision and complex valued inputs:

```
sage: bessel_K(0, 1).n(128)                                             #␣
↪needs sage.symbolic
0.42102443824070833333562737921260903614
sage: bessel_K(0, RealField(200)(1))                                    #␣
↪needs sage.rings.real_mpfr
0.42102443824070833333562737921260903613621974822666047229897
sage: bessel_K(0, ComplexField(200)(0.5+I))                             #␣
↪needs sage.rings.real_mpfr sage.symbolic
0.058365979093103864080375311643360048144715516692187818271179
 - 0.67645499731334483535184142196073004335768129348518210260256*I
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_K(1,x), (x,0,5), color='blue')                        #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_K(1, x), (-5, 5), (-5, 5), plot_points=20)    #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

REFERENCES:

- [AS-Bessel]
- [DLMF-Bessel]

- [WP-Bessel]

class sage.functions.bessel.**Function_Bessel_Y**

Bases: BuiltinFunction

The Bessel Y functions, also known as the Bessel functions of the second kind, Weber functions, or Neumann functions.

$Y_\nu(z)$ is a holomorphic function of z on the complex plane, cut along the negative real axis. It is singular at $z = 0$. When z is fixed, $Y_\nu(z)$ is an entire function of the order ν.

DEFINITION:

$$Y_n(z) = \frac{J_\nu(z)\cos(\nu z) - J_{-\nu}(z)}{\sin(\nu z)}$$

Its derivative with respect to z is:

$$\frac{d}{dz}Y_n(z) = \frac{1}{z^n}\left(z^n Y_{n-1}(z) - nz^{n-1}Y_n(z)\right)$$

EXAMPLES:

```
sage: bessel_Y(1, x)                                                    #␣
↪needs sage.symbolic
bessel_Y(1, x)
sage: bessel_Y(1.0, 1.0)                                                #␣
↪needs mpmath
-0.781212821300289

sage: # needs sage.symbolic
sage: n = var('n')
sage: bessel_Y(n, x)
bessel_Y(n, x)
sage: bessel_Y(2, I).n()
1.03440456978312 - 0.135747669767038*I
sage: bessel_Y(0, 0).n()
-infinity
sage: bessel_Y(0, 1).n(128)
0.088256964215676957982926766023515162828
```

Examples of symbolic manipulation:

```
sage: # needs sage.symbolic
sage: a = bessel_Y(pi, bessel_Y(1, I)); a
bessel_Y(pi, bessel_Y(1, I))
sage: N(a, digits=20)
4.2059146571791095708 + 21.307914215321993526*I
sage: f = bessel_Y(2, x)
sage: f.diff(x)
-1/2*bessel_Y(3, x) + 1/2*bessel_Y(1, x)
```

High precision and complex valued inputs (see github issue #4230):

```
sage: bessel_Y(0, 1).n(128)                                             #␣
↪needs sage.symbolic
0.088256964215676957982926766023515162828
sage: bessel_Y(0, RealField(200)(1))                                    #␣
↪needs sage.rings.real_mpfr
0.088256964215676957982926766023515162827817523090675546711044
```

(continues on next page)

(continued from previous page)

```
sage: bessel_Y(0, ComplexField(200)(0.5+I))                                   #␣
↪needs sage.symbolic
0.077763160184438051408593468823822434235010300228009867784073
 + 1.0142336049916069152644677682828326441579314239591288411739*I
```

Visualization (set plot_points to a higher value to get more detail):

```
sage: plot(bessel_Y(1, x), (x, 0, 5), color='blue')                           #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: complex_plot(bessel_Y(1, x), (-5, 5), (-5, 5), plot_points=20)          #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

ALGORITHM:

Numerical evaluation is handled by the mpmath library. Symbolics are handled by a combination of Maxima and Sage (Ginac/Pynac).

REFERENCES:

- [AS-Bessel]
- [DLMF-Bessel]
- [WP-Bessel]

class sage.functions.bessel.**Function_Hankel1**

Bases: BuiltinFunction

The Hankel function of the first kind

DEFINITION:

$$H_\nu^{(1)}(z) = J_\nu(z) + iY_\nu(z)$$

EXAMPLES:

```
sage: hankel1(3, x)                                                           #␣
↪needs sage.symbolic
hankel1(3, x)
sage: hankel1(3, 4.)                                                          #␣
↪needs mpmath
0.430171473875622 - 0.182022115953485*I
sage: latex(hankel1(3, x))                                                    #␣
↪needs sage.symbolic
H_{3}^{(1)}\left(x\right)
sage: hankel1(3., x).series(x == 2, 10).subs(x=3).n()   # abs tol 1e-12      #␣
↪needs sage.symbolic
0.309062682819597 - 0.512591541605233*I
sage: hankel1(3, 3.)                                                          #␣
↪needs mpmath
0.309062722255252 - 0.538541616105032*I
```

REFERENCES:

- [AS-Bessel] see 9.1.6

class sage.functions.bessel.**Function_Hankel2**

Bases: BuiltinFunction

The Hankel function of the second kind

DEFINITION:

$$H_\nu^{(2)}(z) = J_\nu(z) - iY_\nu(z)$$

EXAMPLES:

```
sage: hankel2(3, x)                                                         #␣
↪needs sage.symbolic
hankel2(3, x)
sage: hankel2(3, 4.)                                                        #␣
↪needs mpmath
0.430171473875622 + 0.182022115953485*I
sage: latex(hankel2(3, x))                                                  #␣
↪needs sage.symbolic
H_{3}^{(2)}\left(x\right)
sage: hankel2(3., x).series(x == 2, 10).subs(x=3).n()   # abs tol 1e-12         #␣
↪needs sage.symbolic
0.309062682819597 + 0.512591541605234*I
sage: hankel2(3, 3.)                                                        #␣
↪needs mpmath
0.309062722255252 + 0.538541616105032*I
```

REFERENCES:

- [AS-Bessel] see 9.1.6

class sage.functions.bessel.**Function_Struve_H**

Bases: BuiltinFunction

The Struve functions, solutions to the non-homogeneous Bessel differential equation:

$$x^2\frac{d^2y}{dx^2}+x\frac{dy}{dx}+(x^2-\alpha^2)y = \frac{4\left(\frac{x}{2}\right)^{\alpha+1}}{\sqrt{\pi}\Gamma(\alpha+\frac{1}{2})},$$

$$\mathrm{H}_\alpha(x) = y(x)$$

EXAMPLES:

```
sage: struve_H(-1/2, x)                                                     #␣
↪needs sage.symbolic
sqrt(2)*sqrt(1/(pi*x))*sin(x)
sage: struve_H(2, x)                                                        #␣
↪needs sage.symbolic
struve_H(2, x)
sage: struve_H(1/2, pi).n()                                                 #␣
↪needs sage.symbolic
0.900316316157106
```

REFERENCES:

- [AS-Struve]
- [DLMF-Struve]
- [WP-Struve]

class sage.functions.bessel.**Function_Struve_L**

Bases: BuiltinFunction

The modified Struve functions.

$$\mathrm{L}_\alpha(x) = -i \cdot e^{-\frac{i\alpha\pi}{2}} \cdot \mathrm{H}_\alpha(ix)$$

EXAMPLES:

```
sage: struve_L(2, x)                                                     #␣
↪needs sage.symbolic
struve_L(2, x)
sage: struve_L(1/2, pi).n()                                              #␣
↪needs sage.symbolic
4.76805417696286
sage: diff(struve_L(1,x), x)                                             #␣
↪needs sage.symbolic
1/3*x/pi - 1/2*struve_L(2, x) + 1/2*struve_L(0, x)
```

REFERENCES:

- [AS-Struve]
- [DLMF-Struve]
- [WP-Struve]

class sage.functions.bessel.**SphericalBesselJ**

Bases: BuiltinFunction

The spherical Bessel function of the first kind

DEFINITION:

$$j_n(z) = \sqrt{\frac{\pi}{2z}}\, J_{n+\frac{1}{2}}(z)$$

EXAMPLES:

```
sage: spherical_bessel_J(3, 3.)                                          #␣
↪needs mpmath
0.152051662030533
sage: spherical_bessel_J(2.,3.)      # rel tol 1e-10                     #␣
↪needs mpmath
0.2986374970757335

sage: # needs sage.symbolic
sage: spherical_bessel_J(3, x)
spherical_bessel_J(3, x)
sage: spherical_bessel_J(3 + 0.2 * I, 3)
0.150770999183897 - 0.0260662466510632*I
sage: spherical_bessel_J(3, x).series(x == 2, 10).subs(x=3).n()
0.152051648665037
sage: spherical_bessel_J(4, x).simplify()
-((45/x^2 - 105/x^4 - 1)*sin(x) + 5*(21/x^2 - 2)*cos(x)/x)/x
sage: integrate(spherical_bessel_J(1,x)^2,(x,0,oo))
1/6*pi
sage: latex(spherical_bessel_J(4, x))
j_{4}\left(x\right)
```

REFERENCES:

- [AS-Spherical]
- [DLMF-Bessel]
- [WP-Bessel]

class sage.functions.bessel.**SphericalBesselY**

Bases: BuiltinFunction

The spherical Bessel function of the second kind

DEFINITION:

$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z)$$

EXAMPLES:

```
sage: # needs sage.symbolic
sage: spherical_bessel_Y(3, x)
spherical_bessel_Y(3, x)
sage: spherical_bessel_Y(3 + 0.2 * I, 3)
-0.505215297588210 - 0.0508835883281404*I
sage: spherical_bessel_Y(-3, x).simplify()
((3/x^2 - 1)*sin(x) - 3*cos(x)/x)/x
sage: spherical_bessel_Y(3 + 2 * I, 5 - 0.2 * I)
-0.270205813266440 - 0.615994702714957*I
sage: integrate(spherical_bessel_Y(0, x), x)
-1/2*Ei(I*x) - 1/2*Ei(-I*x)
sage: integrate(spherical_bessel_Y(1,x)^2,(x,0,oo))
-1/6*pi
sage: latex(spherical_bessel_Y(0, x))
y_{0}\left(x\right)
```

REFERENCES:

- [AS-Spherical]
- [DLMF-Bessel]
- [WP-Bessel]

class sage.functions.bessel.**SphericalHankel1**

Bases: BuiltinFunction

The spherical Hankel function of the first kind

DEFINITION:

$$h_n^{(1)}(z) = \sqrt{\frac{\pi}{2z}} H_{n+\frac{1}{2}}^{(1)}(z)$$

EXAMPLES:

```
sage: # needs sage.symbolic
sage: spherical_hankel1(3, x)
spherical_hankel1(3, x)
sage: spherical_hankel1(3 + 0.2 * I, 3)
0.201654587512037 - 0.531281544239273*I
sage: spherical_hankel1(1, x).simplify()
-(x + I)*e^(I*x)/x^2
sage: spherical_hankel1(3 + 2 * I, 5 - 0.2 * I)
```

(continues on next page)

(continued from previous page)

```
1.25375216869913 - 0.518011435921789*I
sage: integrate(spherical_hankel1(3, x), x)
Ei(I*x) - 6*gamma(-1, -I*x) - 15*gamma(-2, -I*x) - 15*gamma(-3, -I*x)
sage: latex(spherical_hankel1(3, x))
h_{3}^{(1)}\left(x\right)
```

REFERENCES:

- [AS-Spherical]
- [DLMF-Bessel]
- [WP-Bessel]

class sage.functions.bessel.**SphericalHankel2**

Bases: BuiltinFunction

The spherical Hankel function of the second kind

DEFINITION:

$$h_n^{(2)}(z) = \sqrt{\frac{\pi}{2z}}\,H_{n+\frac{1}{2}}^{(2)}(z)$$

EXAMPLES:

```
sage: # needs sage.symbolic
sage: spherical_hankel2(3, x)
spherical_hankel2(3, x)
sage: spherical_hankel2(3 + 0.2 * I, 3)
0.0998874108557565 + 0.479149050937147*I
sage: spherical_hankel2(1, x).simplify()
-(x - I)*e^(-I*x)/x^2
sage: spherical_hankel2(2,i).simplify()
-e
sage: spherical_hankel2(2,x).simplify()
(-I*x^2 - 3*x + 3*I)*e^(-I*x)/x^3
sage: spherical_hankel2(3 + 2*I, 5 - 0.2*I)
0.0217627632692163 + 0.0224001906110906*I
sage: integrate(spherical_hankel2(3, x), x)
Ei(-I*x) - 6*gamma(-1, I*x) - 15*gamma(-2, I*x) - 15*gamma(-3, I*x)
sage: latex(spherical_hankel2(3, x))
h_{3}^{(2)}\left(x\right)
```

REFERENCES:

- [AS-Spherical]
- [DLMF-Bessel]
- [WP-Bessel]

sage.functions.bessel.**spherical_bessel_f**(F, n, z)

Numerically evaluate the spherical version, f, of the Bessel function F by computing $f_n(z) = \sqrt{\frac{1}{2}\pi/z}F_{n+\frac{1}{2}}(z)$. According to Abramowitz & Stegun, this identity holds for the Bessel functions J, Y, K, I, $H^{(1)}$, and $H^{(2)}$.

EXAMPLES:

```
sage: from sage.functions.bessel import spherical_bessel_f
sage: spherical_bessel_f('besselj', 3, 4)                                  #␣
↪needs mpmath
mpf('0.22924385795503024')
sage: spherical_bessel_f('hankel1', 3, 4)                                  #␣
↪needs mpmath
mpc(real='0.22924385795503024', imag='-0.21864196590306359')
```

1.15 Exponential integrals

AUTHORS:

- Benjamin Jones (2011-06-12)

This module provides easy access to many exponential integral special functions. It utilizes Maxima's special functions package and the mpmath library.

REFERENCES:

- [AS1964] Abramowitz and Stegun: *Handbook of Mathematical Functions*
- Wikipedia article Exponential_integral
- Online Encyclopedia of Special Function: http://algo.inria.fr/esf/index.html
- NIST Digital Library of Mathematical Functions: https://dlmf.nist.gov/
- Maxima special functions package
- mpmath library

AUTHORS:

- Benjamin Jones

  Implementations of the classes `Function_exp_integral_*`.

- David Joyner and William Stein

  Authors of the code which was moved from special.py and trans.py. Implementation of `exp_int()` (from sage/functions/special.py). Implementation of *`exponential_integral_1()`* (from sage/functions/transcendental.py).

class sage.functions.exp_integral.**Function_cos_integral**

Bases: `BuiltinFunction`

The trigonometric integral $\operatorname{Ci}(z)$ defined by

$$\operatorname{Ci}(z) = \gamma + \log(z) + \int_0^z \frac{\cos(t) - 1}{t}\, dt,$$

where γ is the Euler gamma constant (`euler_gamma` in Sage), see [AS1964] 5.2.1.

EXAMPLES:

```
sage: z = var('z')                                                         #␣
↪needs sage.symbolic
sage: cos_integral(z)                                                      #␣
↪needs sage.symbolic
cos_integral(z)
```

(continues on next page)

(continued from previous page)

```
sage: cos_integral(3.0)                                                    #␣
↪needs mpmath
0.119629786008000
sage: cos_integral(0)                                                      #␣
↪needs sage.symbolic
cos_integral(0)
sage: N(cos_integral(0))                                                   #␣
↪needs mpmath
-infinity
```

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: cos_integral(3.0)                                                    #␣
↪needs mpmath
0.119629786008000
```

The alias `Ci` can be used instead of `cos_integral`:

```
sage: Ci(3.0)                                                              #␣
↪needs mpmath
0.119629786008000
```

Compare `cos_integral(3.0)` to the definition of the value using numerical integration:

```
sage: a = numerical_integral((cos(x)-1)/x, 0, 3)[0]                        #␣
↪needs sage.symbolic
sage: abs(N(euler_gamma + log(3)) + a - N(cos_integral(3.0))) < 1e-14      #␣
↪needs sage.symbolic
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(cos_integral(3), digits=30)                                        #␣
↪needs sage.symbolic
0.119629786008000327626472281177
sage: cos_integral(ComplexField(100)(3+I))                                 #␣
↪needs sage.symbolic
0.078134230477495714401983633057 - 0.37814733904787920181190368789*I
```

The limit $\mathrm{Ci}(z)$ as $z \to \infty$ is zero:

```
sage: N(cos_integral(1e23))                                                #␣
↪needs mpmath
-3.24053937643003e-24
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = cos_integral(x)
sage: f.diff(x)
cos(x)/x
sage: f.integrate(x)
x*cos_integral(x) - sin(x)
```

The Nielsen spiral is the parametric plot of (Si(t), Ci(t)):

```
sage: # needs sage.symbolic
sage: t = var('t')
sage: f(t) = sin_integral(t)
sage: g(t) = cos_integral(t)
sage: P = parametric_plot([f, g], (t, 0.5 ,20))                        #␣
↪needs sage.plot
sage: show(P, frame=True, axes=False)                                  #␣
↪needs sage.plot
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Trigonometric_integral
- mpmath documentation: ci

class sage.functions.exp_integral.**Function_cosh_integral**

Bases: BuiltinFunction

The trigonometric integral $\mathrm{Chi}(z)$ defined by

$$\mathrm{Chi}(z) = \gamma + \log(z) + \int_0^z \frac{\cosh(t) - 1}{t}\, dt,$$

see [AS1964] 5.2.4.

EXAMPLES:

```
sage: z = var('z')                                                     #␣
↪needs sage.symbolic
sage: cosh_integral(z)                                                 #␣
↪needs sage.symbolic
cosh_integral(z)
sage: cosh_integral(3.0)                                               #␣
↪needs mpmath
4.96039209476561
```

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: cosh_integral(1.0)                                               #␣
↪needs mpmath
0.837866940980208
```

The alias `Chi` can be used instead of `cosh_integral`:

```
sage: Chi(1.0)                                                         #␣
↪needs mpmath
0.837866940980208
```

Here is an example from the mpmath documentation:

```
sage: f(x) = cosh_integral(x)                                          #␣
↪needs sage.symbolic
sage: find_root(f, 0.1, 1.0)                                           #␣
↪needs scipy sage.symbolic
0.523822571389...
```

Compare `cosh_integral(3.0)` to the definition of the value using numerical integration:

```
sage: a = numerical_integral((cosh(x)-1)/x, 0, 3)[0]                    #␣
↪needs sage.symbolic
sage: abs(N(euler_gamma + log(3)) + a - N(cosh_integral(3.0))) < 1e-14  #␣
↪needs sage.symbolic
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(cosh_integral(3), digits=30)                                    #␣
↪needs sage.symbolic
4.96039209476560976029791763669
sage: cosh_integral(ComplexField(100)(3+I))                             #␣
↪needs sage.symbolic
3.9096723099686417127843516794 + 3.0547519627014217273323873274*I
```

The limit of $\operatorname{Chi}(z)$ as $z \to \infty$ is ∞:

```
sage: N(cosh_integral(Infinity))                                        #␣
↪needs mpmath
+infinity
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = cosh_integral(x)
sage: f.diff(x)
cosh(x)/x
sage: f.integrate(x)
x*cosh_integral(x) - sinh(x)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Trigonometric_integral
- mpmath documentation: chi

class `sage.functions.exp_integral.`**`Function_exp_integral`**

Bases: `BuiltinFunction`

The generalized complex exponential integral Ei(z) defined by

$$\operatorname{Ei}(x) = \int_{-\infty}^{x} \frac{e^t}{t}\, dt$$

for x > 0 and for complex arguments by analytic continuation, see [AS1964] 5.1.2.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: Ei(10)
Ei(10)
sage: Ei(I)
Ei(I)
```

(continues on next page)

(continued from previous page)

```
sage: Ei(3+I)
Ei(I + 3)
sage: Ei(10r)
Ei(10)

sage: Ei(1.3)                                                              #␣
↪needs mpmath
2.72139888023202
sage: Ei(1.3r)                                                             #␣
↪needs mpmath
2.7213988802320235
```

The branch cut for this function is along the negative real axis:

```
sage: Ei(-3 + 0.1*I)                                                       #␣
↪needs sage.symbolic
-0.0129379427181693 + 3.13993830250942*I
sage: Ei(-3 - 0.1*I)                                                       #␣
↪needs sage.symbolic
-0.0129379427181693 - 3.13993830250942*I
```

The precision for the result is deduced from the precision of the input. Convert the input to a higher precision explicitly if a result with higher precision is desired:

```
sage: Ei(RealField(300)(1.1))                                              #␣
↪needs sage.rings.real_mpfr
2.
↪16737827956340282358378734233807621497112737591639704719499002090327541763352339357795426
```

ALGORITHM: Uses mpmath.

class sage.functions.exp_integral.**Function_exp_integral_e**

Bases: BuiltinFunction

The generalized complex exponential integral $E_n(z)$ defined by

$$E_n(z) = \int_1^\infty \frac{e^{-zt}}{t^n} \, dt$$

for complex numbers n and z, see [AS1964] 5.1.4.

The special case where $n = 1$ is denoted in Sage by `exp_integral_e1`.

EXAMPLES:

Numerical evaluation is handled using mpmath:

```
sage: N(exp_integral_e(1, 1))                                              #␣
↪needs sage.symbolic
0.219383934395520
sage: exp_integral_e(1, RealField(100)(1))                                 #␣
↪needs sage.symbolic
0.21938393439552027367716377546
```

We can compare this to PARI's evaluation of *exponential_integral_1()*:

```
sage: N(exponential_integral_1(1))                                         #␣
↪needs sage.symbolic
0.219383934395520
```

We can verify one case of [AS1964] 5.1.45, i.e. $E_n(z) = z^{n-1}\Gamma(1-n, z)$:

```
sage: N(exp_integral_e(2, 3+I))                                              #␣
↪needs sage.symbolic
0.00354575823814662 - 0.00973200528288687*I
sage: N((3+I)*gamma(-1, 3+I))                                                #␣
↪needs sage.symbolic
0.00354575823814662 - 0.00973200528288687*I
```

Maxima returns the following improper integral as a multiple of `exp_integral_e(1,1)`:

```
sage: uu = integral(e^(-x)*log(x+1), x, 0, oo); uu                           #␣
↪needs sage.symbolic
e*exp_integral_e(1, 1)
sage: uu.n(digits=30)                                                        #␣
↪needs sage.symbolic
0.596347362323194074341078499369
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = exp_integral_e(2,x)
sage: f.diff(x)
-exp_integral_e(1, x)
sage: f.integrate(x)
-exp_integral_e(3, x)
sage: f = exp_integral_e(-1, x)
sage: f.integrate(x)
Ei(-x) - gamma(-1, x)
```

Some special values of `exp_integral_e` can be simplified. [AS1964] 5.1.23:

```
sage: exp_integral_e(0, x)                                                   #␣
↪needs sage.symbolic
e^(-x)/x
```

[AS1964] 5.1.24:

```
sage: # needs sage.symbolic
sage: exp_integral_e(6, 0)
1/5
sage: nn = var('nn')
sage: assume(nn > 1)
sage: f = exp_integral_e(nn, 0)
sage: f.simplify()
1/(nn - 1)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

class sage.functions.exp_integral.**Function_exp_integral_e1**

Bases: `BuiltinFunction`

The generalized complex exponential integral $E_1(z)$ defined by

$$E_1(z) = \int_z^\infty \frac{e^{-t}}{t}\, dt$$

see [AS1964] 5.1.4.

EXAMPLES:

```
sage: exp_integral_e1(x)                                                    #␣
↪needs sage.symbolic
exp_integral_e1(x)
sage: exp_integral_e1(1.0)                                                  #␣
↪needs mpmath
0.219383934395520
```

Numerical evaluation is handled using mpmath:

```
sage: N(exp_integral_e1(1))                                                 #␣
↪needs sage.symbolic
0.219383934395520
sage: exp_integral_e1(RealField(100)(1))                                    #␣
↪needs sage.rings.real_mpfr
0.21938393439552027367716377546
```

We can compare this to PARI's evaluation of `exponential_integral_1()`:

```
sage: N(exp_integral_e1(2.0))                                               #␣
↪needs mpmath
0.0489005107080611
sage: N(exponential_integral_1(2.0))                                        #␣
↪needs sage.rings.real_mpfr
0.0489005107080611
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = exp_integral_e1(x)
sage: f.diff(x)
-e^(-x)/x
sage: f.integrate(x)
-exp_integral_e(2, x)
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

class sage.functions.exp_integral.**Function_log_integral**

Bases: `BuiltinFunction`

The logarithmic integral $\mathrm{li}(z)$ defined by

$$\mathrm{li}(x) = \int_0^z \frac{dt}{\ln(t)} = \mathrm{Ei}(\ln(x))$$

for x > 1 and by analytic continuation for complex arguments z (see [AS1964] 5.1.3).

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: N(log_integral(3))                                                    #␣
↪needs sage.symbolic
2.16358859466719
```

(continues on next page)

(continued from previous page)

```
sage: N(log_integral(3), digits=30)                                    #␣
↪needs sage.symbolic
2.16358859466719197287692236735
sage: log_integral(ComplexField(100)(3+I))                             #␣
↪needs sage.symbolic
2.2879892769816826157078450911 + 0.87232935488528370139883806779*I
sage: log_integral(0)                                                  #␣
↪needs mpmath
0
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = log_integral(x)
sage: f.diff(x)
1/log(x)
sage: f.integrate(x)
x*log_integral(x) - Ei(2*log(x))
```

Here is a test from the mpmath documentation. There are 1,925,320,391,606,803,968,923 many prime numbers less than 1e23. The value of `log_integral(1e23)` is very close to this:

```
sage: log_integral(1e23)                                               #␣
↪needs mpmath
1.92532039161405e21
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Logarithmic_integral_function
- mpmath documentation: logarithmic-integral

class sage.functions.exp_integral.**Function_log_integral_offset**

Bases: `BuiltinFunction`

The offset logarithmic integral, or Eulerian logarithmic integral, $\operatorname{Li}(x)$ is defined by

$$\operatorname{Li}(x)=\int_2^x \frac{dt}{\ln(t)}=\operatorname{li}(x)-\operatorname{li}(2)$$

for $x \geq 2$.

The offset logarithmic integral should also not be confused with the polylogarithm (also denoted by $\operatorname{Li}(x)$), which is implemented as *sage.functions.log.Function_polylog*.

$\operatorname{Li}(x)$ is identical to $\operatorname{li}(x)$ except that the lower limit of integration is 2 rather than 0 to avoid the singularity at $x=1$ of

$$\frac{1}{\ln(t)}$$

See *Function_log_integral* for details of $\operatorname{li}(x)$. Thus $\operatorname{Li}(x)$ can also be represented by

$$\operatorname{Li}(x)=\operatorname{li}(x)-\operatorname{li}(2)$$

So we have:

```
sage: li(4.5) - li(2.0) - Li(4.5)                                            #␣
↪needs mpmath
0.000000000000000
```

$\text{Li}(x)$ is extended to complex arguments z by analytic continuation (see [AS1964] 5.1.3):

```
sage: Li(6.6 + 5.4*I)                                                        #␣
↪needs sage.symbolic
3.97032201503632 + 2.62311237593572*I
```

The function Li is an approximation for the number of primes up to x. In fact, the famous Riemann Hypothesis is

$$|\pi(x) - \text{Li}(x)| \leq \sqrt{x}\log(x).$$

For "small" x, $\text{Li}(x)$ is always slightly bigger than $\pi(x)$. However it is a theorem that there are very large values of x (e.g., around 10^{316}), such that $\exists x : \pi(x) > \text{Li}(x)$. See "A new bound for the smallest x with $\pi(x) > \text{li}(x)$", Bays and Hudson, Mathematics of Computation, 69 (2000) 1285-1296.

Note: Definite integration returns a part symbolic and part numerical result. This is because when Li(x) is evaluated it is passed as li(x)-li(2).

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: # needs sage.symbolic
sage: N(log_integral_offset(3))
1.11842481454970
sage: N(log_integral_offset(3), digits=30)
1.11842481454969918803233347815
sage: log_integral_offset(ComplexField(100)(3+I))
1.2428254968641898308632562019 + 0.87232935488528370139883806779*I
sage: log_integral_offset(2)
0
sage: for n in range(1,7):                                                   #␣
↪needs primecountpy
....:  print('%-10s%-10s%-20s'%(10^n, prime_pi(10^n), N(Li(10^n))))
10        4         5.12043572466980
100       25        29.0809778039621
1000      168       176.564494210035
10000     1229      1245.09205211927
100000    9592      9628.76383727068
1000000   78498     78626.5039956821
```

Here is a test from the mpmath documentation. There are 1,925,320,391,606,803,968,923 prime numbers less than 1e23. The value of `log_integral_offset(1e23)` is very close to this:

```
sage: log_integral_offset(1e23)                                              #␣
↪needs mpmath
1.92532039161405e21
```

Symbolic derivatives are handled by Sage and integration by Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = log_integral_offset(x)
```

(continues on next page)

(continued from previous page)

```
sage: f.diff(x)
1/log(x)
sage: f.integrate(x)
-x*log_integral(2) + x*log_integral(x) - Ei(2*log(x))
sage: Li(x).integrate(x, 2.0, 4.5).n(digits=10)
3.186411697
sage: N(f.integrate(x, 2.0, 3.0))   # abs tol 1e-15
0.601621785860587
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Logarithmic_integral_function
- mpmath documentation: logarithmic-integral

class sage.functions.exp_integral.**Function_sin_integral**

Bases: `BuiltinFunction`

The trigonometric integral $\mathrm{Si}(z)$ defined by

$$\mathrm{Si}(z) = \int_0^z \frac{\sin(t)}{t}\, dt,$$

see [AS1964] 5.2.1.

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: sin_integral(0)                                                   #␣
↪needs mpmath
0
sage: sin_integral(0.0)                                                 #␣
↪needs mpmath
0.000000000000000
sage: sin_integral(3.0)                                                 #␣
↪needs mpmath
1.84865252799947
sage: N(sin_integral(3), digits=30)                                     #␣
↪needs sage.symbolic
1.84865252799946825639773025111
sage: sin_integral(ComplexField(100)(3+I))                              #␣
↪needs sage.symbolic
2.0277151656451253616038525998 + 0.015210926166954211913653130271*I
```

The alias `Si` can be used instead of `sin_integral`:

```
sage: Si(3.0)                                                           #␣
↪needs mpmath
1.84865252799947
```

The limit of $\mathrm{Si}(z)$ as $z \to \infty$ is $\pi/2$:

```
sage: N(sin_integral(1e23))                                             #␣
↪needs mpmath
```

(continues on next page)

(continued from previous page)

```
1.57079632679490
sage: N(pi/2)                                                            #␣
↪needs sage.symbolic
1.57079632679490
```

At 200 bits of precision $\mathrm{Si}(10^{23})$ agrees with $\pi/2$ up to 10^{-24}:

```
sage: sin_integral(RealField(200)(1e23))                                 #␣
↪needs sage.rings.real_mpfr
1.5707963267948966192313288218697837425815368604836679189519
sage: N(pi/2, prec=200)                                                  #␣
↪needs sage.symbolic
1.5707963267948966192313216916397514420985846996875529104875
```

The exponential sine integral is analytic everywhere:

```
sage: sin_integral(-1.0)                                                 #␣
↪needs mpmath
-0.946083070367183
sage: sin_integral(-2.0)                                                 #␣
↪needs mpmath
-1.60541297680269
sage: sin_integral(-1e23)                                                #␣
↪needs mpmath
-1.57079632679490
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f = sin_integral(x)
sage: f.diff(x)
sin(x)/x
sage: f.integrate(x)
x*sin_integral(x) + cos(x)
sage: integrate(sin(x)/x, x)
-1/2*I*Ei(I*x) + 1/2*I*Ei(-I*x)
```

Compare values of the functions $\mathrm{Si}(x)$ and $f(x) = (1/2)i \cdot \mathrm{Ei}(-ix) - (1/2)i \cdot \mathrm{Ei}(ix) - \pi/2$, which are both anti-derivatives of $\sin(x)/x$, at some random positive real numbers:

```
sage: f(x) = 1/2*I*Ei(-I*x) - 1/2*I*Ei(I*x) - pi/2                      #␣
↪needs sage.symbolic
sage: g(x) = sin_integral(x)                                             #␣
↪needs sage.symbolic
sage: R = [abs(RDF.random_element()) for i in range(100)]
sage: all(abs(f(x) - g(x)) < 1e-10 for x in R)                           #␣
↪needs sage.symbolic
True
```

The Nielsen spiral is the parametric plot of (Si(t), Ci(t)):

```
sage: # needs sage.symbolic
sage: x = var('x')
sage: f(x) = sin_integral(x)
sage: g(x) = cos_integral(x)
```

(continues on next page)

(continued from previous page)

```
sage: P = parametric_plot([f, g], (x, 0.5 ,20))                          #␣
↪needs sage.plot
sage: show(P, frame=True, axes=False)                                    #␣
↪needs sage.plot
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Trigonometric_integral
- mpmath documentation: si

class sage.functions.exp_integral.**Function_sinh_integral**

Bases: BuiltinFunction

The trigonometric integral $\operatorname{Shi}(z)$ defined by

$$\operatorname{Shi}(z) = \int_0^z \frac{\sinh(t)}{t}\, dt,$$

see [AS1964] 5.2.3.

EXAMPLES:

Numerical evaluation for real and complex arguments is handled using mpmath:

```
sage: sinh_integral(3.0)                                                 #␣
↪needs mpmath
4.97344047585981
sage: sinh_integral(1.0)                                                 #␣
↪needs mpmath
1.05725087537573
sage: sinh_integral(-1.0)                                                #␣
↪needs mpmath
-1.05725087537573
```

The alias Shi can be used instead of sinh_integral:

```
sage: Shi(3.0)                                                           #␣
↪needs mpmath
4.97344047585981
```

Compare sinh_integral(3.0) to the definition of the value using numerical integration:

```
sage: a = numerical_integral(sinh(x)/x, 0, 3)[0]                         #␣
↪needs sage.symbolic
sage: abs(a - N(sinh_integral(3))) < 1e-14                               #␣
↪needs sage.symbolic
True
```

Arbitrary precision and complex arguments are handled:

```
sage: N(sinh_integral(3), digits=30)                                     #␣
↪needs sage.symbolic
4.97344047585980679771041838252
sage: sinh_integral(ComplexField(100)(3+I))                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
3.9134623660329374406788354078 + 3.0427678212908839256360163759*I
```

The limit $\mathrm{Shi}(z)$ as $z \to \infty$ is ∞:

```
sage: N(sinh_integral(Infinity))                                        #␣
↪needs mpmath
+infinity
```

Symbolic derivatives and integrals are handled by Sage and Maxima:

```
sage: x = var('x')                                                      #␣
↪needs sage.symbolic
sage: f = sinh_integral(x)                                              #␣
↪needs sage.symbolic
sage: f.diff(x)                                                         #␣
↪needs sage.symbolic
sinh(x)/x

sage: f.integrate(x)                                                    #␣
↪needs sage.symbolic
x*sinh_integral(x) - cosh(x)
```

Note that due to some problems with the way Maxima handles these expressions, definite integrals can sometimes give unexpected results (typically when using inexact endpoints) due to inconsistent branching:

```
sage: integrate(sinh_integral(x), x, 0, 1/2)                            #␣
↪needs sage.symbolic
-cosh(1/2) + 1/2*sinh_integral(1/2) + 1
sage: integrate(sinh_integral(x), x, 0, 1/2).n()  # correct             #␣
↪needs sage.symbolic
0.125872409703453
sage: integrate(sinh_integral(x), x, 0, 0.5).n()  # fixed in maxima 5.29.1      #␣
↪needs sage.symbolic
0.125872409703453
```

ALGORITHM:

Numerical evaluation is handled using mpmath, but symbolics are handled by Sage and Maxima.

REFERENCES:

- Wikipedia article Trigonometric_integral
- mpmath documentation: shi

`sage.functions.exp_integral.`**`exponential_integral_1`**`(x, n=0)`

Returns the exponential integral $E_1(x)$. If the optional argument n is given, computes list of the first n values of the exponential integral $E_1(xm)$.

The exponential integral $E_1(x)$ is

$$E_1(x) = \int_x^\infty \frac{e^{-t}}{t}\, dt$$

INPUT:

- `x` – a positive real number

- n – (default: 0) a nonnegative integer; if nonzero, then return a list of values E_1(x*m) for m = 1,2,3,…,n. This is useful, e.g., when computing derivatives of L-functions.

OUTPUT:

A real number if n is 0 (the default) or a list of reals if n > 0. The precision is the same as the input, with a default of 53 bits in case the input is exact.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: exponential_integral_1(2)
0.0489005107080611
sage: exponential_integral_1(2, 4)  # abs tol 1e-18
[0.0489005107080611, 0.00377935240984891, 0.000360082452162659, 0.
↪0000376656228439245]
sage: exponential_integral_1(40, 5)
[0.000000000000000, 2.22854325868847e-37, 6.33732515501151e-55,
 2.02336191509997e-72, 6.88522610630764e-90]
sage: r = exponential_integral_1(RealField(150)(1)); r
0.21938393439552027367716377546012164903104729
sage: parent(r)
Real Field with 150 bits of precision
sage: exponential_integral_1(RealField(150)(100))
3.6835977616820321802351926205081189876552201e-46

sage: exponential_integral_1(0)
+Infinity
```

ALGORITHM: use the PARI C-library function eint1.

REFERENCE:

- See Proposition 5.6.12 of Cohen's book "A Course in Computational Algebraic Number Theory".

1.16 Wigner, Clebsch-Gordan, Racah, and Gaunt coefficients

Collection of functions for calculating Wigner 3-j, 6-j, 9-j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [RH2003].

Please see the description of the individual functions for further details and examples.

AUTHORS:

- Jens Rasch (2009-03-24): initial version for Sage
- Jens Rasch (2009-05-31): updated to sage-4.0

sage.functions.wigner.**clebsch_gordan**(*j_1*, *j_2*, *j_3*, *m_1*, *m_2*, *m_3*, *prec=None*)

Return the Clebsch-Gordan coefficient $\langle j_1 m_1 \; j_2 m_2 | j_3 m_3 \rangle$.

The reference for this function is [Ed1974].

INPUT:

- j_1, j_2, j_3, m_1, m_2, m_3 – integer or half integer
- prec – precision, default: None. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: simplify(clebsch_gordan(3/2,1/2,2, 3/2,1/2,2))                    #␣
↪needs sage.symbolic
1
sage: clebsch_gordan(1.5,0.5,1, 1.5,-0.5,1)                             #␣
↪needs sage.symbolic
1/2*sqrt(3)
sage: clebsch_gordan(3/2,1/2,1, -1/2,1/2,0)                             #␣
↪needs sage.symbolic
-sqrt(3)*sqrt(1/6)
```

Note: The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3-j symbols:

$$\langle j_1 m_1 \; j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2 j_3 + 1} \begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & -m_3 \end{pmatrix}$$

See also the documentation on Wigner 3-j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

AUTHORS:

- Jens Rasch (2009-03-24): initial version

`sage.functions.wigner.`**`gaunt`**(*l_1, l_2, l_3, m_1, m_2, m_3, prec=None*)

Return the Gaunt coefficient.

The Gaunt coefficient is defined as the integral over three spherical harmonics:

$$\begin{aligned} &Y(l_1, l_2, l_3, m_1, m_2, m_3) \\ &= \int Y_{l_1,m_1}(\Omega)\, Y_{l_2,m_2}(\Omega)\, Y_{l_3,m_3}(\Omega)\, d\Omega \\ &= \sqrt{\frac{(2l_1+1)(2l_2+1)(2l_3+1)}{4\pi}} \\ &\qquad \times \begin{pmatrix} l_1 & l_2 & l_3 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} l_1 & l_2 & l_3 \\ m_1 & m_2 & m_3 \end{pmatrix} \end{aligned}$$

INPUT:

- `l_1`, `l_2`, `l_3`, `m_1`, `m_2`, `m_3` – integer
- `prec` – precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: gaunt(1,0,1,1,0,-1)
-1/2/sqrt(pi)
sage: gaunt(1,0,1,1,0,0)
0
```

(continues on next page)

(continued from previous page)

```
sage: gaunt(29,29,34,10,-5,-5)
1821867940156/215552371055153321*sqrt(22134)/sqrt(pi)
sage: gaunt(20,20,40,1,-1,0)
28384503878959800/74029560764440771/sqrt(pi)
sage: gaunt(12,15,5,2,3,-5)
91/124062*sqrt(36890)/sqrt(pi)
sage: gaunt(10,10,12,9,3,-12)
-98/62031*sqrt(6279)/sqrt(pi)
sage: gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448
```

If the sum of the l_i is odd, the answer is zero, even for Python ints (see github issue #14766):

```
sage: gaunt(1,2,2,1,0,-1)
0
sage: gaunt(int(1),int(2),int(2),1,0,-1)
0
```

It is an error to use non-integer values for l or m:

```
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

ALGORITHM:

This function uses the algorithm of [LdB1982] to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [RH2003].

AUTHORS:

- Jens Rasch (2009-03-24): initial version for Sage

`sage.functions.wigner.`**`racah`**`(`*aa*, *bb*, *cc*, *dd*, *ee*, *ff*, *prec=None*`)`

Return the Racah symbol $W(aa, bb, cc, dd; ee, ff)$.

INPUT:

- `aa`, ..., `ff` – integer or half integer
- `prec` – precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: racah(3,3,3,3,3,3)                                                    #␣
↪needs sage.symbolic
-1/14
```

Note: The Racah symbol is related to the Wigner 6-j symbol:

$$\begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix} = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4; j_3, j_6)$$

Please see the 6-j symbol for its much richer symmetries and for additional properties.

ALGORITHM:

This function uses the algorithm of [Ed1974] to calculate the value of the 6-j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [RH2003].

AUTHORS:

- Jens Rasch (2009-03-24): initial version

sage.functions.wigner.**wigner_3j**(*j_1, j_2, j_3, m_1, m_2, m_3, prec=None*)

Return the Wigner 3-j symbol $\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$.

INPUT:

- `j_1`, `j_2`, `j_3`, `m_1`, `m_2`, `m_3` – integer or half integer
- `prec` – precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: wigner_3j(2, 6, 4, 0, 0, 0)                                        #␣
↪needs sage.symbolic
sqrt(5/143)
sage: wigner_3j(2, 6, 4, 0, 0, 1)
0
sage: wigner_3j(0.5, 0.5, 1, 0.5, -0.5, 0)                               #␣
↪needs sage.symbolic
sqrt(1/6)
sage: wigner_3j(40, 100, 60, -10, 60, -50)                               #␣
↪needs sage.symbolic
95608/18702538494885*sqrt(21082735836735314343364163310/220491455010479533763)
sage: wigner_3j(2500, 2500, 5000, 2488, 2400, -4888, prec=64)            #␣
↪needs sage.rings.real_mpfr
7.60424456883448589e-12
```

It is an error to have arguments that are not integer or half integer values:

```
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer or half integer
```

The Wigner 3-j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where $J = j_1+j_2+j_3$):

$$\begin{aligned}\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix} &= \begin{pmatrix} j_3 & j_1 & j_2 \\ m_3 & m_1 & m_2 \end{pmatrix} = \begin{pmatrix} j_2 & j_3 & j_1 \\ m_2 & m_3 & m_1 \end{pmatrix} \\ &= (-1)^J \begin{pmatrix} j_3 & j_2 & j_1 \\ m_3 & m_2 & m_1 \end{pmatrix} = (-1)^J \begin{pmatrix} j_1 & j_3 & j_2 \\ m_1 & m_3 & m_2 \end{pmatrix} = (-1)^J \begin{pmatrix} j_2 & j_1 & j_3 \\ m_2 & m_1 & m_3 \end{pmatrix}\end{aligned}$$

- invariant under space inflection, i.e.

$$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix} = (-1)^J \begin{pmatrix} j_1 & j_2 & j_3 \\ -m_1 & -m_2 & -m_3 \end{pmatrix}$$

- symmetric with respect to the 72 additional symmetries based on the work by [Reg1958]
- zero for j_1, j_2, j_3 not fulfilling triangle relation
- zero for $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions $j_1 \geq |m_1|$, $j_2 \geq |m_2|$, $j_3 \geq |m_3|$

ALGORITHM:

This function uses the algorithm of [Ed1974] to calculate the value of the 3-j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [RH2003].

AUTHORS:

- Jens Rasch (2009-03-24): initial version

`sage.functions.wigner.`**`wigner_6j`**`(`*j_1, j_2, j_3, j_4, j_5, j_6, prec=None*`)`

Return the Wigner 6-j symbol $\begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix}$.

INPUT:

- `j_1`, …, `j_6` – integer or half integer
- `prec` – precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: wigner_6j(3,3,3,3,3,3)
-1/14
sage: wigner_6j(5,5,5,5,5,5)
1/52
sage: wigner_6j(6,6,6,6,6,6)
309/10868
sage: wigner_6j(8,8,8,8,8,8)
-12219/965770
sage: wigner_6j(30,30,30,30,30,30)
36082186869033479581/87954851694828981714124
sage: wigner_6j(0.5,0.5,1,0.5,0.5,1)
```

(continues on next page)

(continued from previous page)

```
1/6
sage: wigner_6j(200,200,200,200,200,200, prec=1000)*1.0
0.000155903212413242
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle␣
↪relation
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle␣
↪relation
```

The Wigner 6-j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

$$\begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix} = (-1)^{j_1+j_2+j_4+j_5} W(j_1, j_2, j_5, j_4; j_3, j_6)$$

The Wigner 6-j symbol obeys the following symmetry rules:

- Wigner 6-j symbols are left invariant under any permutation of the columns:

$$\begin{aligned} \begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix} &= \begin{Bmatrix} j_3 & j_1 & j_2 \\ j_6 & j_4 & j_5 \end{Bmatrix} = \begin{Bmatrix} j_2 & j_3 & j_1 \\ j_5 & j_6 & j_4 \end{Bmatrix} \\ &= \begin{Bmatrix} j_3 & j_2 & j_1 \\ j_6 & j_5 & j_4 \end{Bmatrix} = \begin{Bmatrix} j_1 & j_3 & j_2 \\ j_4 & j_6 & j_5 \end{Bmatrix} = \begin{Bmatrix} j_2 & j_1 & j_3 \\ j_5 & j_4 & j_6 \end{Bmatrix} \end{aligned}$$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

$$\begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix} = \begin{Bmatrix} j_1 & j_5 & j_6 \\ j_4 & j_2 & j_3 \end{Bmatrix} = \begin{Bmatrix} j_4 & j_2 & j_6 \\ j_1 & j_5 & j_3 \end{Bmatrix} = \begin{Bmatrix} j_4 & j_5 & j_3 \\ j_1 & j_2 & j_6 \end{Bmatrix}$$

- additional 6 symmetries [Reg1959] giving rise to 144 symmetries in total
- only non-zero if any triple of j's fulfill a triangle relation

ALGORITHM:

This function uses the algorithm of [Ed1974] to calculate the value of the 6-j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [RH2003].

`sage.functions.wigner.`**`wigner_9j`**(*j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, prec=None*)

Return the Wigner 9-j symbol $\begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \\ j_7 & j_8 & j_9 \end{Bmatrix}$.

INPUT:

- `j_1`, …, `j_9` – integer or half integer
- `prec` – precision, default: `None`. Providing a precision can drastically speed up the calculation.

OUTPUT:

Rational number times the square root of a rational number (if `prec=None`), or real number if a precision is given.

EXAMPLES:

A couple of examples and test cases, note that for speed reasons a precision is given:

```
sage: # needs sage.symbolic
sage: wigner_9j(1,1,1, 1,1,1, 1,1,0, prec=64) # ==1/18
0.0555555555555555555
sage: wigner_9j(1,1,1, 1,1,1, 1,1,1)
0
sage: wigner_9j(1,1,1, 1,1,1, 1,1,2, prec=64) # ==1/18
0.0555555555555555556
sage: wigner_9j(1,2,1, 2,2,2, 1,2,1, prec=64) # ==-1/150
-0.00666666666666666667
sage: wigner_9j(3,3,2, 2,2,2, 3,3,2, prec=64) # ==157/14700
0.0106802721088435374
sage: wigner_9j(3,3,2, 3,3,2, 3,3,2, prec=64) # ==3221*sqrt(70)/
↪(246960*sqrt(105)) - 365/(3528*sqrt(70)*sqrt(105))
0.00944247746651111739
sage: wigner_9j(3,3,1, 3.5,3.5,2, 3.5,3.5,1, prec=64) # ==3221*sqrt(70)/
↪(246960*sqrt(105)) - 365/(3528*sqrt(70)*sqrt(105))
0.0110216678544351364
sage: wigner_9j(100,80,50, 50,100,70, 60,50,100, prec=1000)*1.0
1.05597798065761e-7
sage: wigner_9j(30,30,10, 30.5,30.5,20, 30.5,30.5,10, prec=1000)*1.0 #␣
↪==(80944680186359968990/95103769817469)*sqrt(1/682288158959699477295)
0.0000325841699408828
sage: wigner_9j(64,62.5,114.5, 61.5,61,112.5, 113.5,110.5,60, prec=1000)*1.0
-3.41407910055520e-39
sage: wigner_9j(15,15,15, 15,3,15, 15,18,10, prec=1000)*1.0
-0.0000778324615309539
sage: wigner_9j(1.5,1,1.5, 1,1,1, 1.5,1,1.5)
0
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_9j(0.5,0.5,0.5, 0.5,0.5,0.5, 0.5,0.5,0.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle␣
↪relation
sage: wigner_9j(1,1,1, 0.5,1,1.5, 0.5,1,2.5,prec=64)                        #␣
↪needs sage.rings.real_mpfr
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the triangle␣
↪relation
```

ALGORITHM:

This function uses the algorithm of [Ed1974] to calculate the value of the 3-j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [RH2003].

1.17 Generalized functions

Sage implements several generalized functions (also known as distributions) such as Dirac delta, Heaviside step functions. These generalized functions can be manipulated within Sage like any other symbolic functions.

AUTHORS:

- Golam Mortuza Hossain (2009-06-26): initial version

EXAMPLES:

Dirac delta function:

```
sage: dirac_delta(x)                                                    #␣
↪needs sage.symbolic
dirac_delta(x)
```

Heaviside step function:

```
sage: heaviside(x)                                                      #␣
↪needs sage.symbolic
heaviside(x)
```

Unit step function:

```
sage: unit_step(x)                                                      #␣
↪needs sage.symbolic
unit_step(x)
```

Signum (sgn) function:

```
sage: sgn(x)                                                            #␣
↪needs sage.symbolic
sgn(x)
```

Kronecker delta function:

```
sage: m, n = var('m,n')                                                 #␣
↪needs sage.symbolic
sage: kronecker_delta(m, n)                                             #␣
↪needs sage.symbolic
kronecker_delta(m, n)
```

class sage.functions.generalized.**FunctionDiracDelta**

Bases: BuiltinFunction

The Dirac delta (generalized) function, $\delta(x)$ (dirac_delta(x)).

INPUT:

- x - a real number or a symbolic expression

DEFINITION:

Dirac delta function $\delta(x)$, is defined in Sage as:

$\delta(x) = 0$ for real $x \neq 0$ and $\int_{-\infty}^{\infty} \delta(x)dx = 1$

Its alternate definition with respect to an arbitrary test function $f(x)$ is

$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a)$

EXAMPLES:

```
sage: # needs sage.symbolic
sage: dirac_delta(1)
0
sage: dirac_delta(0)
dirac_delta(0)
sage: dirac_delta(x)
dirac_delta(x)
sage: integrate(dirac_delta(x), x, -1, 1, algorithm='sympy')                #␣
↪needs sympy
1
```

REFERENCES:

- Wikipedia article Dirac_delta_function

class sage.functions.generalized.**FunctionHeaviside**

Bases: `GinacFunction`

The Heaviside step function, $H(x)$ (`heaviside(x)`).

INPUT:

- `x` - a real number or a symbolic expression

DEFINITION:

The Heaviside step function, $H(x)$ is defined in Sage as:

$H(x) = 0$ for $x < 0$ and $H(x) = 1$ for $x > 0$

See also:

unit_step()

EXAMPLES:

```
sage: # needs sage.symbolic
sage: heaviside(-1)
0
sage: heaviside(1)
1
sage: heaviside(0)
heaviside(0)
sage: heaviside(x)
heaviside(x)

sage: heaviside(-1/2)                                                        #␣
↪needs sage.symbolic
0
sage: heaviside(exp(-1000000000000000000000))                                #␣
↪needs sage.symbolic
1
```

REFERENCES:

- Wikipedia article Heaviside_function

class sage.functions.generalized.**FunctionKroneckerDelta**

Bases: `BuiltinFunction`

The Kronecker delta function $\delta_{m,n}$ (`kronecker_delta(m, n)`).

INPUT:

- `m` - a number or a symbolic expression
- `n` - a number or a symbolic expression

DEFINITION:

Kronecker delta function $\delta_{m,n}$ is defined as:

$\delta_{m,n} = 0$ for $m \neq n$ and $\delta_{m,n} = 1$ for $m = n$

EXAMPLES:

```
sage: kronecker_delta(1,2)                                                    #␣
↪needs sage.rings.complex_interval_field
0
sage: kronecker_delta(1,1)                                                    #␣
↪needs sage.rings.complex_interval_field
1
sage: m, n = var('m,n')                                                       #␣
↪needs sage.symbolic
sage: kronecker_delta(m, n)                                                   #␣
↪needs sage.symbolic
kronecker_delta(m, n)
```

REFERENCES:

- Wikipedia article Kronecker_delta

class sage.functions.generalized.**FunctionSignum**

Bases: `BuiltinFunction`

The signum or sgn function $\mathrm{sgn}(x)$ (`sgn(x)`).

INPUT:

- `x` - a real number or a symbolic expression

DEFINITION:

The sgn function, $\mathrm{sgn}(x)$ is defined as:

$\mathrm{sgn}(x) = 1$ for $x > 0$, $\mathrm{sgn}(x) = 0$ for $x = 0$ and $\mathrm{sgn}(x) = -1$ for $x < 0$

EXAMPLES:

```
sage: sgn(-1)
-1
sage: sgn(1)
1
sage: sgn(0)
0
sage: sgn(x)                                                                  #␣
↪needs sage.symbolic
sgn(x)
```

We can also use `sign`:

```
sage: sign(1)
1
sage: sign(0)
0
```

(continues on next page)

(continued from previous page)

```
sage: a = AA(-5).nth_root(7)                                              #␣
↪needs sage.rings.number_field
sage: sign(a)                                                             #␣
↪needs sage.rings.number_field
-1
```

REFERENCES:

- Wikipedia article Sign_function

class sage.functions.generalized.**FunctionUnitStep**

Bases: GinacFunction

The unit step function, $\mathrm{u}(x)$ (unit_step(x)).

INPUT:

- x - a real number or a symbolic expression

DEFINITION:

The unit step function, $\mathrm{u}(x)$ is defined in Sage as:

$\mathrm{u}(x) = 0$ for $x < 0$ and $\mathrm{u}(x) = 1$ for $x \geq 0$

See also:

heaviside()

EXAMPLES:

```
sage: # needs sage.symbolic
sage: unit_step(-1)
0
sage: unit_step(1)
1
sage: unit_step(0)
1
sage: unit_step(x)
unit_step(x)
sage: unit_step(-exp(-10000000000000000000))
0
```

1.18 Counting primes

EXAMPLES:

```
sage: z = sage.functions.prime_pi.PrimePi()
sage: loads(dumps(z))
prime_pi
sage: loads(dumps(z)) == z
True
```

AUTHORS:

- R. Andrew Ohana (2009): initial version of efficient prime_pi
- William Stein (2009): fix plot method

- R. Andrew Ohana (2011): complete rewrite, ~5x speedup
- Dima Pasechnik (2021): removed buggy cython code, replaced it with calls to primecount/primecountpy spkg

class sage.functions.prime_pi.**PrimePi**

Bases: BuiltinFunction

The prime counting function, which counts the number of primes less than or equal to a given value.

INPUT:

- x – a real number
- prime_bound – (default 0) a real number < 2^32; prime_pi() will make sure to use all the primes up to prime_bound (although, possibly more) in computing prime_pi, this can potentially speedup the time of computation, at a cost to memory usage.

OUTPUT:

integer – the number of primes $\leq$ x

EXAMPLES:

These examples test common inputs:

```
sage: # needs sage.symbolic
sage: prime_pi(7)
4
sage: prime_pi(100)
25
sage: prime_pi(1000)
168
sage: prime_pi(100000)
9592
sage: prime_pi(500509)
41581
```

The following test is to verify that github issue #4670 has been essentially resolved:

```
sage: prime_pi(10^10)                                                   #␣
↪needs sage.symbolic
455052511
```

The prime_pi() function also has a special plotting method, so it plots quickly and perfectly as a step function:

```
sage: P = plot(prime_pi, 50, 100)                                       #␣
↪needs sage.plot sage.symbolic
```

plot(*xmin=0*, *xmax=100*, *vertical_lines=True*, ***kwds*)

Draw a plot of the prime counting function from xmin to xmax. All additional arguments are passed on to the line command.

WARNING: we draw the plot of prime_pi as a stairstep function with explicitly drawn vertical lines where the function jumps. Technically there should not be any vertical lines, but they make the graph look much better, so we include them. Use the option vertical_lines=False to turn these off.

EXAMPLES:

```
sage: plot(prime_pi, 1, 100)                                            #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 1 graphics primitive
```

(continues on next page)

(continued from previous page)

```
sage: prime_pi.plot(1, 51, thickness=2, vertical_lines=False)           #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 16 graphics primitives
```

sage.functions.prime_pi.**legendre_phi**(*x*, *a*)

Legendre's formula, also known as the partial sieve function, is a useful combinatorial function for computing the prime counting function (the `prime_pi` method in Sage). It counts the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes.

INPUT:

- `x` – a real number
- `a` – a non-negative integer

OUTPUT:

integer – the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes

EXAMPLES:

```
sage: legendre_phi(100, 0)
100
sage: legendre_phi(29375, 1)
14688
sage: legendre_phi(91753, 5973)
2893
sage: legendre_phi(4215701455, 6450023226)
1
```

sage.functions.prime_pi.**partial_sieve_function**(*x*, *a*)

Legendre's formula, also known as the partial sieve function, is a useful combinatorial function for computing the prime counting function (the `prime_pi` method in Sage). It counts the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes.

INPUT:

- `x` – a real number
- `a` – a non-negative integer

OUTPUT:

integer – the number of positive integers $\leq$ `x` that are not divisible by the first `a` primes

EXAMPLES:

```
sage: legendre_phi(100, 0)
100
sage: legendre_phi(29375, 1)
14688
sage: legendre_phi(91753, 5973)
2893
sage: legendre_phi(4215701455, 6450023226)
1
```

1.19 Symbolic minimum and maximum

Sage provides a symbolic maximum and minimum due to the fact that the Python builtin `max()` and `min()` are not able to deal with variables as users might expect. These functions wait to evaluate if there are variables.

Here you can see some differences:

```
sage: max(x, x^2)
↪# needs sage.symbolic
x
sage: max_symbolic(x, x^2)
↪# needs sage.symbolic
max(x, x^2)
sage: f(x) = max_symbolic(x, x^2); f(1/2)
↪# needs sage.symbolic
1/2
```

This works as expected for more than two entries:

```
sage: # needs sage.symbolic
sage: max(3, 5, x)
5
sage: min(3, 5, x)
3
sage: max_symbolic(3, 5, x)
max(x, 5)
sage: min_symbolic(3, 5, x)
min(x, 3)
```

class sage.functions.min_max.**MaxSymbolic**

Bases: *MinMax_base*

Symbolic max function.

The Python builtin `max()` function does not work as expected when symbolic expressions are given as arguments. This function delays evaluation until all symbolic arguments are substituted with values.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: max_symbolic(3, x)
max(3, x)
sage: max_symbolic(3, x).subs(x=5)
5
sage: max_symbolic(3, 5, x)
max(x, 5)
sage: max_symbolic([3, 5, x])
max(x, 5)
```

class sage.functions.min_max.**MinMax_base**

Bases: `BuiltinFunction`

eval_helper(*this_f*, *builtin_f*, *initial_val*, *args*)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: max_symbolic(3, 5, x)   # indirect doctest
max(x, 5)
```

(continues on next page)

(continued from previous page)

```
sage: max_symbolic([5.0r])   # indirect doctest
5.0
sage: min_symbolic(3, 5, x)
min(x, 3)
sage: min_symbolic([5.0r])   # indirect doctest
5.0
```

class sage.functions.min_max.**MinSymbolic**

Bases: *MinMax_base*

Symbolic min function.

The Python builtin `min()` function does not work as expected when symbolic expressions are given as arguments. This function delays evaluation until all symbolic arguments are substituted with values.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: min_symbolic(3, x)
min(3, x)
sage: min_symbolic(3, x).subs(x=5)
3
sage: min_symbolic(3, 5, x)
min(x, 3)
sage: min_symbolic([3, 5, x])
min(x, 3)
```

Please find extensive developer documentation for creating new functions in Symbolic Calculus, in particular in the section Classes for symbolic functions.

CHAPTER

TWO

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

f

INDEX

G

H

I

T

U

V

W

Z