atan

class sympy.functions.elementary.trigonometric.**atan**(*arg*)

The inverse tangent function.

Returns the arc tangent of x (measured in radians).

Explanation

atan(x) will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the eval class method).

Examples

```
>>> from sympy import atan, oo
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *acot* (page 397), *atan2* (page 400)

References

[R265], [R266], [R267]

inverse(*argindex=1*)

Returns the inverse of this function.

acot

class sympy.functions.elementary.trigonometric.**acot**(*arg*)

The inverse cotangent function.

Returns the arc cotangent of x (measured in radians).

Explanation

acot(x) will evaluate automatically in the cases $x \in \{\infty, -\infty, \tilde{\infty}, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the eval class method).

A purely imaginary argument will lead to an acoth expression.

acot(x) has a branch cut along $(-i, i)$, hence it is discontinuous at 0. Its range for real x is $(-\frac{\pi}{2}, \frac{\pi}{2}]$.

Examples

```
>>> from sympy import acot, sqrt
>>> acot(0)
pi/2
>>> acot(1)
pi/4
>>> acot(sqrt(3) - 2)
-5*pi/12
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *atan2* (page 400)

References

[R268], [R269]

inverse(*argindex=1*)
Returns the inverse of this function.

asec

class sympy.functions.elementary.trigonometric.**asec**(*arg*)
The inverse secant function.

Returns the arc secant of x (measured in radians).

Explanation

asec(x) will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$ and for some instances when the result is a rational multiple of π (see the eval class method).

asec(x) has branch cut in the interval $[-1, 1]$. For complex arguments, it can be defined [R273] as

$$\sec^{-1}(z) = -i\frac{\log\left(\sqrt{1 - z^2} + 1\right)}{z}$$

At `x = 0`, for positive branch cut, the limit evaluates to `zoo`. For negative branch cut, the limit

$$\lim_{z\to 0} -i\frac{\log\left(-\sqrt{1-z^2}+1\right)}{z}$$

simplifies to $-i\log\left(z/2+O\left(z^3\right)\right)$ which ultimately evaluates to `zoo`.

As `acos(x) = asec(1/x)`, a similar argument can be given for `acos(x)`.

Examples

```
>>> from sympy import asec, oo
>>> asec(1)
0
>>> asec(-1)
pi
>>> asec(0)
zoo
>>> asec(-oo)
pi/2
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R270], [R271], [R272], [R273]

inverse(*argindex=1*)

Returns the inverse of this function.

acsc

class sympy.functions.elementary.trigonometric.**acsc**(*arg*)

The inverse cosecant function.

Returns the arc cosecant of x (measured in radians).

Explanation

`acsc(x)` will evaluate automatically in the cases $x \in \{\infty, -\infty, 0, 1, -1\}$` and for some instances when the result is a rational multiple of $\pi$ (see the `eval` class method).

Examples

```
>>> from sympy import acsc, oo
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
>>> acsc(oo)
0
>>> acsc(-oo) == acsc(oo)
True
>>> acsc(0)
zoo
```

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397), *atan2* (page 400)

References

[R274], [R275], [R276]

inverse(*argindex=1*)

Returns the inverse of this function.

atan2

class sympy.functions.elementary.trigonometric.**atan2**(*y*, *x*)

The function `atan2(y, x)` computes $\operatorname{atan}(y/x)$ taking two arguments y and x. Signs of both y and x are considered to determine the appropriate quadrant of $\operatorname{atan}(y/x)$. The range is $(-\pi, \pi]$. The complete definition reads as follows:

$$\operatorname{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \ge 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The y-coordinate is the first argument and the x-coordinate the second.

If either x or y is complex:

$$\operatorname{atan2}(y, x) = -i \log\left(\frac{x + iy}{\sqrt{x^2 + y^2}}\right)$$

Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> from sympy import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary atan function for the point $(x, y) = (-1, 1)$

```
>>> from sympy import atan, S
>>> atan(S(1)/-1)
-pi/4
>>> atan2(1, -1)
3*pi/4
```

where only the atan2 function reurns what we expect. We can differentiate the function with respect to both arguments:

```
>>> from sympy import diff
>>> from sympy.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
```

```
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the atan2 function in terms of complex logarithms:

```
>>> from sympy import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of $(atan)$:

```
>>> from sympy import atan
>>> atan2(y, x).rewrite(atan)
Piecewise((2*atan(y/(x + sqrt(x**2 + y**2))), Ne(y, 0)), (pi, re(x) < 0),
↪ (0, Ne(x, 0)), (nan, True))
```

but note that this form is undefined on the negative real axis.

See also:

sin (page 389), *csc* (page 393), *cos* (page 390), *sec* (page 393), *tan* (page 391), *cot* (page 392), *asin* (page 395), *acsc* (page 399), *acos* (page 396), *asec* (page 398), *atan* (page 397), *acot* (page 397)

References

[R277], [R278], [R279]

sympy.functions.elementary.hyperbolic

Hyperbolic Functions

HyperbolicFunction

`class sympy.functions.elementary.hyperbolic.`**`HyperbolicFunction`**`(*args)`

Base class for hyperbolic functions.

See also:

sinh (page 402), *cosh* (page 403), *tanh* (page 403), *coth* (page 404)

sinh

`class sympy.functions.elementary.hyperbolic.`**`sinh`**`(arg)`

`sinh(x)` is the hyperbolic sine of `x`.

The hyperbolic sine function is $\frac{e^x - e^{-x}}{2}$.

Examples

```
>>> from sympy import sinh
>>> from sympy.abc import x
>>> sinh(x)
sinh(x)
```

See also:

cosh (page 403), *tanh* (page 403), *asinh* (page 405)

`as_real_imag``(deep=True, **hints)`

Returns this function as a complex coordinate.

`fdiff``(argindex=1)`

Returns the first derivative of this function.

`inverse``(argindex=1)`

Returns the inverse of this function.

static taylor_term(*n, x, *previous_terms*)

Returns the next term in the Taylor series expansion.

cosh

class sympy.functions.elementary.hyperbolic.**cosh**(*arg*)

cosh(x) is the hyperbolic cosine of x.

The hyperbolic cosine function is $\frac{e^x+e^{-x}}{2}$.

Examples

```
>>> from sympy import cosh
>>> from sympy.abc import x
>>> cosh(x)
cosh(x)
```

See also:

sinh (page 402), *tanh* (page 403), *acosh* (page 406)

tanh

class sympy.functions.elementary.hyperbolic.**tanh**(*arg*)

tanh(x) is the hyperbolic tangent of x.

The hyperbolic tangent function is $\frac{\sinh(x)}{\cosh(x)}$.

Examples

```
>>> from sympy import tanh
>>> from sympy.abc import x
>>> tanh(x)
tanh(x)
```

See also:

sinh (page 402), *cosh* (page 403), *atanh* (page 406)

inverse(*argindex=1*)

Returns the inverse of this function.

coth

class sympy.functions.elementary.hyperbolic.**coth**(*arg*)

coth(x) is the hyperbolic cotangent of x.

The hyperbolic cotangent function is $\frac{\cosh(x)}{\sinh(x)}$.

Examples

```
>>> from sympy import coth
>>> from sympy.abc import x
>>> coth(x)
coth(x)
```

See also:

sinh (page 402), *cosh* (page 403), *acoth* (page 407)

inverse(*argindex=1*)

Returns the inverse of this function.

sech

class sympy.functions.elementary.hyperbolic.**sech**(*arg*)

sech(x) is the hyperbolic secant of x.

The hyperbolic secant function is $\frac{2}{e^x+e^{-x}}$

Examples

```
>>> from sympy import sech
>>> from sympy.abc import x
>>> sech(x)
sech(x)
```

See also:

sinh (page 402), *cosh* (page 403), *tanh* (page 403), *coth* (page 404), *csch* (page 404), *asinh* (page 405), *acosh* (page 406)

csch

class sympy.functions.elementary.hyperbolic.**csch**(*arg*)

csch(x) is the hyperbolic cosecant of x.

The hyperbolic cosecant function is $\frac{2}{e^x-e^{-x}}$

Examples

```
>>> from sympy import csch
>>> from sympy.abc import x
>>> csch(x)
csch(x)
```

See also:

sinh (page 402), *cosh* (page 403), *tanh* (page 403), *sech* (page 404), *asinh* (page 405), *acosh* (page 406)

fdiff(*argindex=1*)

Returns the first derivative of this function

static taylor_term(*n, x, *previous_terms*)

Returns the next term in the Taylor series expansion

Hyperbolic Inverses

asinh

class sympy.functions.elementary.hyperbolic.**asinh**(*arg*)

asinh(x) is the inverse hyperbolic sine of x.

The inverse hyperbolic sine function.

Examples

```
>>> from sympy import asinh
>>> from sympy.abc import x
>>> asinh(x).diff(x)
1/sqrt(x**2 + 1)
>>> asinh(1)
log(1 + sqrt(2))
```

See also:

acosh (page 406), *atanh* (page 406), *sinh* (page 402)

inverse(*argindex=1*)

Returns the inverse of this function.

acosh

class sympy.functions.elementary.hyperbolic.**acosh**(*arg*)

acosh(x) is the inverse hyperbolic cosine of x.

The inverse hyperbolic cosine function.

Examples

```
>>> from sympy import acosh
>>> from sympy.abc import x
>>> acosh(x).diff(x)
1/(sqrt(x - 1)*sqrt(x + 1))
>>> acosh(1)
0
```

See also:

asinh (page 405), *atanh* (page 406), *cosh* (page 403)

inverse(*argindex=1*)

Returns the inverse of this function.

atanh

class sympy.functions.elementary.hyperbolic.**atanh**(*arg*)

atanh(x) is the inverse hyperbolic tangent of x.

The inverse hyperbolic tangent function.

Examples

```
>>> from sympy import atanh
>>> from sympy.abc import x
>>> atanh(x).diff(x)
1/(1 - x**2)
```

See also:

asinh (page 405), *acosh* (page 406), *tanh* (page 403)

inverse(*argindex=1*)

Returns the inverse of this function.

acoth

class sympy.functions.elementary.hyperbolic.**acoth**(*arg*)

acoth(x) is the inverse hyperbolic cotangent of x.

The inverse hyperbolic cotangent function.

Examples

```
>>> from sympy import acoth
>>> from sympy.abc import x
>>> acoth(x).diff(x)
1/(1 - x**2)
```

See also:

asinh (page 405), *acosh* (page 406), *coth* (page 404)

inverse(*argindex=1*)

Returns the inverse of this function.

asech

class sympy.functions.elementary.hyperbolic.**asech**(*arg*)

asech(x) is the inverse hyperbolic secant of x.

The inverse hyperbolic secant function.

Examples

```
>>> from sympy import asech, sqrt, S
>>> from sympy.abc import x
>>> asech(x).diff(x)
-1/(x*sqrt(1 - x**2))
>>> asech(1).diff(x)
0
>>> asech(1)
0
>>> asech(S(2))
I*pi/3
>>> asech(-sqrt(2))
3*I*pi/4
>>> asech((sqrt(6) - sqrt(2)))
I*pi/12
```

See also:

asinh (page 405), *atanh* (page 406), *cosh* (page 403), *acoth* (page 407)

References

[R280], [R281], [R282]

inverse(*argindex=1*)

Returns the inverse of this function.

acsch

class sympy.functions.elementary.hyperbolic.**acsch**(*arg*)

acsch(x) is the inverse hyperbolic cosecant of x.

The inverse hyperbolic cosecant function.

Examples

```
>>> from sympy import acsch, sqrt, S
>>> from sympy.abc import x
>>> acsch(x).diff(x)
-1/(x**2*sqrt(1 + x**(-2)))
>>> acsch(1).diff(x)
0
>>> acsch(1)
log(1 + sqrt(2))
>>> acsch(S.ImaginaryUnit)
-I*pi/2
>>> acsch(-2*S.ImaginaryUnit)
I*pi/6
>>> acsch(S.ImaginaryUnit*(sqrt(6) - sqrt(2)))
-5*I*pi/12
```

See also:

asinh (page 405)

References

[R283], [R284], [R285]

inverse(*argindex=1*)

Returns the inverse of this function.

sympy.functions.elementary.integers

ceiling

class sympy.functions.elementary.integers.**ceiling**(*arg*)

Ceiling is a univariate function which returns the smallest integer value not less than its argument. This implementation generalizes ceiling to complex numbers by taking the ceiling of the real and imaginary parts separately.

Examples

```
>>> from sympy import ceiling, E, I, S, Float, Rational
>>> ceiling(17)
17
>>> ceiling(Rational(23, 10))
3
>>> ceiling(2*E)
6
>>> ceiling(-Float(0.567))
0
>>> ceiling(I/2)
I
>>> ceiling(S(5)/2 + 5*I/2)
3 + 3*I
```

See also:

sympy.functions.elementary.integers.floor (page 409)

References

[R286], [R287]

floor

class sympy.functions.elementary.integers.**floor**(*arg*)

Floor is a univariate function which returns the largest integer value not greater than its argument. This implementation generalizes floor to complex numbers by taking the floor of the real and imaginary parts separately.

Examples

```
>>> from sympy import floor, E, I, S, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
>>> floor(S(5)/2 + 5*I/2)
2 + 2*I
```

See also:

sympy.functions.elementary.integers.ceiling (page 409)

References

[R288], [R289]

RoundFunction

class sympy.functions.elementary.integers.**RoundFunction**(*arg*)

Abstract base class for rounding functions.

frac

class sympy.functions.elementary.integers.**frac**(*arg*)

Represents the fractional part of x

For real numbers it is defined [R290] as

$$x - \lfloor x \rfloor$$

Examples

```
>>> from sympy import Symbol, frac, Rational, floor, I
>>> frac(Rational(4, 3))
1/3
>>> frac(-Rational(4, 3))
2/3
```

returns zero for integer arguments

```
>>> n = Symbol('n', integer=True)
>>> frac(n)
0
```

rewrite as floor

```
>>> x = Symbol('x')
>>> frac(x).rewrite(floor)
x - floor(x)
```

for complex arguments

```
>>> r = Symbol('r', real=True)
>>> t = Symbol('t', real=True)
>>> frac(t + I*r)
I*frac(r) + frac(t)
```

See also:

sympy.functions.elementary.integers.floor (page 409), *sympy.functions.elementary.integers.ceiling* (page 409)

References

[R290], [R291]

sympy.functions.elementary.exponential

exp

class sympy.functions.elementary.exponential.**exp**(*arg*)

The exponential function, e^x.

Parameters

arg : Expr

Examples

```
>>> from sympy import exp, I, pi
>>> from sympy.abc import x
>>> exp(x)
exp(x)
>>> exp(x).diff(x)
exp(x)
>>> exp(I*pi)
-1
```

See also:

log (page 413)

as_real_imag(*deep=True*, ***hints*)

Returns this function as a 2-tuple representing a complex number.

Examples

```
>>> from sympy import exp, I
>>> from sympy.abc import x
>>> exp(x).as_real_imag()
(exp(re(x))*cos(im(x)), exp(re(x))*sin(im(x)))
>>> exp(1).as_real_imag()
(E, 0)
>>> exp(I).as_real_imag()
(cos(1), sin(1))
>>> exp(1+I).as_real_imag()
(E*cos(1), E*sin(1))
```

See also:

sympy.functions.elementary.complexes.re (page 382), *sympy.functions.elementary.complexes.im* (page 383)

property base

Returns the base of the exponential function.

fdiff(*argindex=1*)

Returns the first derivative of this function.

static taylor_term(*n*, *x*, **previous_terms*)

Calculates the next term in the Taylor series expansion.

LambertW

class sympy.functions.elementary.exponential.**LambertW**(*x*, *k=None*)

The Lambert W function $W(z)$ is defined as the inverse function of $w\exp(w)$ [R292].

Explanation

In other words, the value of $W(z)$ is such that $z = W(z)\exp(W(z))$ for any complex number z. The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution w of the equation $z = w\exp(w)$.

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

Examples

```
>>> from sympy import LambertW
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_real
False
```

References

[R292]

fdiff(*argindex=1*)

Return the first derivative of this function.

log

class sympy.functions.elementary.exponential.**log**(*arg*, *base=None*)

The natural logarithm function $\ln(x)$ or $\log(x)$.

Explanation

Logarithms are taken with the natural base, e. To get a logarithm of a different base b, use `log(x, b)`, which is essentially short-hand for `log(x)/log(b)`.

`log` represents the principal branch of the natural logarithm. As such it has a branch cut along the negative real axis and returns values having a complex argument in $(-\pi, \pi]$.

Examples

```
>>> from sympy import log, sqrt, S, I
>>> log(8, 2)
3
>>> log(S(8)/3, 2)
-log(3)/log(2) + 3
>>> log(-1 + I*sqrt(3))
log(2) + 2*I*pi/3
```

See also:

exp (page 411)

as_base_exp()

Returns this function in the form (base, exponent).

as_real_imag(*deep=True*, ***hints*)

Returns this function as a complex coordinate.

Examples

```
>>> from sympy import I, log
>>> from sympy.abc import x
>>> log(x).as_real_imag()
(log(Abs(x)), arg(x))
>>> log(I).as_real_imag()
(0, pi/2)
>>> log(1 + I).as_real_imag()
(log(sqrt(2)), pi/4)
>>> log(I*x).as_real_imag()
(log(Abs(x)), arg(I*x))
```

fdiff(*argindex=1*)

Returns the first derivative of the function.

inverse(*argindex=1*)

Returns e^x, the inverse function of $\log(x)$.

static taylor_term(*n, x, *previous_terms*)

Returns the next term in the Taylor series expansion of $\log(1 + x)$.

exp_polar

class sympy.functions.elementary.exponential.**exp_polar**(**args*)

Represent a *polar number* (see g-function Sphinx documentation).

Explanation

exp_polar represents the function $Exp : \mathbb{C} \rightarrow \mathcal{S}$, sending the complex number $z = a + bi$ to the polar number $r = exp(a), \theta = b$. It is one of the main functions to construct polar numbers.

Examples

```
>>> from sympy import exp_polar, pi, I, exp
```

The main difference is that polar numbers do not “wrap around” at 2π:

```
>>> exp(2*pi*I)
1
>>> exp_polar(2*pi*I)
exp_polar(2*I*pi)
```

apart from that they behave mostly like classical complex numbers:

```
>>> exp_polar(2)*exp_polar(3)
exp_polar(5)
```

See also:

sympy.simplify.powsimp.powsimp (page 680), *polar_lift* (page 387), *periodic_argument* (page 388), *principal_branch* (page 389)

sympy.functions.elementary.piecewise

ExprCondPair

class sympy.functions.elementary.piecewise.**ExprCondPair**(*expr*, *cond*)

Represents an expression, condition pair.

property cond

Returns the condition of this pair.

property expr

Returns the expression of this pair.

Piecewise

class sympy.functions.elementary.piecewise.**Piecewise**(**_args*)

Represents a piecewise function.

Usage:

Piecewise((expr,cond), (expr,cond), …)

- Each argument is a 2-tuple defining an expression and condition
- The conds are evaluated in turn returning the first that is True. If any of the evaluated conds are not explicitly False, e.g. `x < 1`, the function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, nan will be returned.
- Pairs where the cond is explicitly False, will be removed and no pair appearing after a True condition will ever be retained. If a single pair with a True condition remains, it will be returned, even when evaluation is False.

Examples

```
>>> from sympy import Piecewise, log, piecewise_fold
>>> from sympy.abc import x, y
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise((0, x < -1), (f, x <= 1), (g, True))
>>> p.subs(x,1)
1
>>> p.subs(x,5)
log(5)
```

Booleans can contain Piecewise elements:

```
>>> cond = (x < y).subs(x, Piecewise((2, x < 0), (3, True))); cond
Piecewise((2, x < 0), (3, True)) < y
```

The folded version of this results in a Piecewise whose expressions are Booleans:

```
>>> folded_cond = piecewise_fold(cond); folded_cond
Piecewise((2 < y, x < 0), (3 < y, True))
```

When a Boolean containing Piecewise (like cond) or a Piecewise with Boolean expressions (like folded_cond) is used as a condition, it is converted to an equivalent *ITE* (page 1171) object:

```
>>> Piecewise((1, folded_cond))
Piecewise((1, ITE(x < 0, y > 2, y > 3)))
```

When a condition is an `ITE`, it will be converted to a simplified Boolean expression:

```
>>> piecewise_fold(_)
Piecewise((1, ((x >= 0) | (y > 2)) & ((y > 3) | (x < 0))))
```

See also:

piecewise_fold (page 419), *piecewise_exclusive* (page 418), *ITE* (page 1171)

_eval_integral(*x*, *_first=True*, ***kwargs*)

Return the indefinite integral of the Piecewise such that subsequent substitution of x with a value will give the value of the integral (not including the constant of integration) up to that point. To only integrate the individual parts of Piecewise, use the `piecewise_integrate` method.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> p.piecewise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

See also:

Piecewise.piecewise_integrate (page 417)

as_expr_set_pairs(*domain=None*)

Return tuples for each argument of self that give the expression and the interval in which it is valid which is contained within the given domain. If a condition cannot be converted to a set, an error will be raised. The variable of the conditions is assumed to be real; sets of real values are returned.

Examples

```
>>> from sympy import Piecewise, Interval
>>> from sympy.abc import x
>>> p = Piecewise(
...     (1, x < 2),
...     (2,(x > 0) & (x < 4)),
...     (3, True))
>>> p.as_expr_set_pairs()
[(1, Interval.open(-oo, 2)),
 (2, Interval.Ropen(2, 4)),
 (3, Interval(4, oo))]
>>> p.as_expr_set_pairs(Interval(0, 3))
[(1, Interval.Ropen(0, 2)),
 (2, Interval(2, 3))]
```

doit(***hints*)

Evaluate this piecewise function.

classmethod eval(**_args*)

Either return a modified version of the args or, if no modifications were made, return None.

Modifications that are made here:

1. relationals are made canonical
2. any False conditions are dropped
3. any repeat of a previous condition is ignored
4. any args past one with a true condition are dropped

If there are no args left, nan will be returned. If there is a single arg with a True condition, its corresponding expression will be returned.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> cond = -x < -1
>>> args = [(1, cond), (4, cond), (3, False), (2, True), (5, x < 1)]
>>> Piecewise(*args, evaluate=False)
Piecewise((1, -x < -1), (4, -x < -1), (2, True))
>>> Piecewise(*args)
Piecewise((1, x > 1), (2, True))
```

piecewise_integrate(*x*, ***kwargs*)

Return the Piecewise with each expression being replaced with its antiderivative. To obtain a continuous antiderivative, use the *integrate()* (page 598) function or method.

Examples

```
>>> from sympy import Piecewise
>>> from sympy.abc import x
>>> p = Piecewise((0, x < 0), (1, x < 1), (2, True))
>>> p.piecewise_integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x, True))
```

Note that this does not give a continuous function, e.g. at x = 1 the 3rd condition applies and the antiderivative there is 2*x so the value of the antiderivative is 2:

```
>>> anti = _
>>> anti.subs(x, 1)
2
```

The continuous derivative accounts for the integral *up to* the point of interest, however:

```
>>> p.integrate(x)
Piecewise((0, x < 0), (x, x < 1), (2*x - 1, True))
>>> _.subs(x, 1)
1
```

See also:

Piecewise._eval_integral (page 416)

`sympy.functions.elementary.piecewise.`**`piecewise_exclusive`**(*expr*, ***, *skip_nan=False*, *deep=True*)

Rewrite *Piecewise* (page 415) with mutually exclusive conditions.

Parameters

expr: a SymPy expression.

Any *Piecewise* (page 415) in the expression will be rewritten.

skip_nan: ``bool`` (default ``False``)

If `skip_nan` is set to `True` then a final *NaN* (page 997) case will not be included.

deep: ``bool`` (default ``True``)

If `deep` is `True` then *piecewise_exclusive()* (page 418) will rewrite any *Piecewise* (page 415) subexpressions in `expr` rather than just rewriting `expr` itself.

Returns

An expression equivalent to `expr` but where all *Piecewise* (page 415) have

been rewritten with mutually exclusive conditions.

Explanation

SymPy represents the conditions of a *Piecewise* (page 415) in an "if-elif"-fashion, allowing more than one condition to be simultaneously True. The interpretation is that the first condition that is True is the case that holds. While this is a useful representation computationally it is not how a piecewise formula is typically shown in a mathematical text. The *piecewise_exclusive()* (page 418) function can be used to rewrite any *Piecewise* (page 415) with more typical mutually exclusive conditions.

Note that further manipulation of the resulting *Piecewise* (page 415), e.g. simplifying it, will most likely make it non-exclusive. Hence, this is primarily a function to be used in conjunction with printing the Piecewise or if one would like to reorder the expression-condition pairs.

If it is not possible to determine that all possibilities are covered by the different cases of the *Piecewise* (page 415) then a final *NaN* (page 997) case will be included explicitly. This can be prevented by passing `skip_nan=True`.

Examples

```
>>> from sympy import piecewise_exclusive, Symbol, Piecewise, S
>>> x = Symbol('x', real=True)
>>> p = Piecewise((0, x < 0), (S.Half, x <= 0), (1, True))
>>> piecewise_exclusive(p)
Piecewise((0, x < 0), (1/2, Eq(x, 0)), (1, x > 0))
>>> piecewise_exclusive(Piecewise((2, x > 1)))
Piecewise((2, x > 1), (nan, x <= 1))
>>> piecewise_exclusive(Piecewise((2, x > 1)), skip_nan=True)
Piecewise((2, x > 1))
```

See also:

Piecewise (page 415), *piecewise_fold* (page 419)

`sympy.functions.elementary.piecewise.`**`piecewise_fold`**(*expr*, *evaluate=True*)

Takes an expression containing a piecewise function and returns the expression in piecewise form. In addition, any ITE conditions are rewritten in negation normal form and simplified.

The final Piecewise is evaluated (default) but if the raw form is desired, send `evaluate=False`; if trivial evaluation is desired, send `evaluate=None` and duplicate conditions and processing of True and False will be handled.

Examples

```
>>> from sympy import Piecewise, piecewise_fold, S
>>> from sympy.abc import x
>>> p = Piecewise((x, x < 1), (1, S(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, True))
```

See also:

Piecewise (page 415), *piecewise_exclusive* (page 418)

sympy.functions.elementary.miscellaneous

IdentityFunction

class sympy.functions.elementary.miscellaneous.**IdentityFunction**

The identity function

Examples

```
>>> from sympy import Id, Symbol
>>> x = Symbol('x')
>>> Id(x)
x
```

Min

class sympy.functions.elementary.miscellaneous.**Min**(**args*)

Return, if possible, the minimum value of the list. It is named `Min` and not `min` to avoid conflicts with the built-in function `min`.

Examples

```
>>> from sympy import Min, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Min(x, -2)
Min(-2, x)
>>> Min(x, -2).subs(x, 3)
-2
>>> Min(p, -3)
-3
>>> Min(x, y)
Min(x, y)
>>> Min(n, 8, p, -7, p, oo)
Min(-7, n)
```

See also:

Max **(page 421)**
find maximum values

Max

class sympy.functions.elementary.miscellaneous.**Max**(**args*)

Return, if possible, the maximum value of the list.

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is $\geq$ the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

It is named `Max` and not `max` to avoid conflicts with the built-in function `max`.

Examples

```
>>> from sympy import Max, Symbol, oo
>>> from sympy.abc import x, y, z
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)
```

```
>>> Max(x, -2)
Max(-2, x)
>>> Max(x, -2).subs(x, 3)
3
>>> Max(p, -2)
p
>>> Max(x, y)
Max(x, y)
>>> Max(x, y) == Max(y, x)
True
>>> Max(x, Max(y, z))
Max(x, y, z)
>>> Max(n, 8, p, 7, -oo)
Max(8, p)
>>> Max (1, x, oo)
oo
```

- Algorithm

The task can be considered as searching of supremums in the directed complete partial orders [R293].

The source values are sequentially allocated by the isolated subsets in which supremums are searched and result as Max arguments.

If the resulted supremum is single, then it is returned.

The isolated subsets are the sets of values which are only the comparable with each other in the current set. E.g. natural numbers are comparable with each other, but not comparable with the x symbol. Another example: the symbol x with negative assumption is comparable with a natural number.

Also there are “least” elements, which are comparable with all others, and have a zero property (maximum or minimum for all elements). For example, in case of ∞, the allocation operation is terminated and only this value is returned.

Assumption:

- if $A > B > C$ then $A > C$
- if $A = B$ then B can be removed

See also:

Min **(page 420)**
find minimum values

References

[R293], [R294]

root

sympy.functions.elementary.miscellaneous.**root**(*arg*, *n*, *k=0*, *evaluate=None*)

Returns the k-th n-th root of arg.

Parameters

k : int, optional

Should be an integer in $\{0, 1, ..., n-1\}$. Defaults to the principal root if 0.

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from global_parameters.evaluate.

Examples

```
>>> from sympy import root, Rational
>>> from sympy.abc import x, n
```

```
>>> root(x, 2)
sqrt(x)
```

```
>>> root(x, 3)
x**(1/3)
```

```
>>> root(x, n)
x**(1/n)
```

```
>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
-(-1)**(2/3)*2**(1/3)
```

To get all n n-th roots you can use the rootof function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> from sympy import rootof
```

```
>>> [rootof(x**2 - 1, i) for i in range(2)]
[-1, 1]
```

```
>>> [rootof(x**3 - 1,i) for i in range(3)]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

```
>>> [rootof(x**4 - 1,i) for i in range(4)]
[-1, 1, -I, I]
```

SymPy, like other symbolic algebra systems, returns the complex root of negative numbers. This is the principal root and differs from the text-book result that one might be expecting. For example, the cube root of -8 does not come back as -2:

```
>>> root(-8, 3)
2*(-1)**(1/3)
```

The real_root function can be used to either make the principal result real (or simply to return the real root directly):

```
>>> from sympy import real_root
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

Alternatively, the n//2-th n-th root of a negative number can be computed with root:

```
>>> root(-32, 5, 5//2)
-2
```

See also:

sympy.polys.rootoftools.rootof (page 2431), *sympy.core.power.integer_nthroot* (page 1008), *sqrt* (page 424), *real_root* (page 426)

References

[R295], [R296], [R297], [R298], [R299]

sqrt

sympy.functions.elementary.miscellaneous.**sqrt**(*arg*, *evaluate=None*)

Returns the principal square root.

Parameters

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import sqrt, Symbol, S
>>> x = Symbol('x')
```

```
>>> sqrt(x)
sqrt(x)
```

```
>>> sqrt(x)**2
x
```

Note that sqrt(x**2) does not simplify to x.

```
>>> sqrt(x**2)
sqrt(x**2)
```

This is because the two are not equal to each other in general. For example, consider x == -1:

```
>>> from sympy import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
False
```

This is because sqrt computes the principal square root, so the square may put the argument in a different branch. This identity does hold if x is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the powdenest() function with the force option set to True:

```
>>> from sympy import powdenest
>>> sqrt(x**2)
sqrt(x**2)
```

(continues on next page)

(continued from previous page)

```
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the rootof function:

```
>>> from sympy import rootof
```

```
>>> [rootof(x**2-3,i) for i in (0,1)]
[-sqrt(3), sqrt(3)]
```

Although `sqrt` is printed, there is no `sqrt` function so looking for `sqrt` in an expression will fail:

```
>>> from sympy.utilities.misc import func_name
>>> func_name(sqrt(x))
'Pow'
>>> sqrt(x).has(sqrt)
False
```

To find `sqrt` look for `Pow` with an exponent of `1/2`:

```
>>> (x + 1/sqrt(x)).find(lambda i: i.is_Pow and abs(i.exp) is S.Half)
{1/sqrt(x)}
```

See also:

sympy.polys.rootoftools.rootof (page 2431), *root* (page 422), *real_root* (page 426)

References

[R300], [R301]

cbrt

sympy.functions.elementary.miscellaneous.**cbrt**(*arg*, *evaluate=None*)

Returns the principal cube root.

Parameters

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If `None`, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import cbrt, Symbol
>>> x = Symbol('x')
```

```
>>> cbrt(x)
x**(1/3)
```

```
>>> cbrt(x)**3
x
```

Note that cbrt(x**3) does not simplify to x.

```
>>> cbrt(x**3)
(x**3)**(1/3)
```

This is because the two are not equal to each other in general. For example, consider $x == -1$:

```
>>> from sympy import Eq
>>> Eq(cbrt(x**3), x).subs(x, -1)
False
```

This is because cbrt computes the principal cube root, this identity does hold if x is positive:

```
>>> y = Symbol('y', positive=True)
>>> cbrt(y**3)
y
```

See also:

sympy.polys.rootoftools.rootof (page 2431), *root* (page 422), *real_root* (page 426)

References

[R302], [R303]

real_root

sympy.functions.elementary.miscellaneous.**real_root**(*arg*, *n=None*, *evaluate=None*)

Return the real *n*'th-root of *arg* if possible.

Parameters

n : int or None, optional

If *n* is None, then all instances of $(-n)^{1/\mathrm{odd}}$ will be changed to $-n^{1/\mathrm{odd}}$. This will only create a real root of a principal root. The presence of other factors may cause the result to not be real.

evaluate : bool, optional

The parameter determines if the expression should be evaluated. If None, its value is taken from `global_parameters.evaluate`.

Examples

```
>>> from sympy import root, real_root
```

```
>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*(-1)**(1/3)
>>> real_root(_)
-2
```

If one creates a non-principal root and applies real_root, the result will not be real (so use with caution):

```
>>> root(-8, 3, 2)
-2*(-1)**(2/3)
>>> real_root(_)
-2*(-1)**(2/3)
```

See also:

sympy.polys.rootoftools.rootof (page 2431), *sympy.core.power.integer_nthroot* (page 1008), *root* (page 422), *sqrt* (page 424)

Combinatorial

This module implements various combinatorial functions.

bell

class sympy.functions.combinatorial.numbers.**bell**(*n*, *k_sym=None*, *symbols=None*)

Bell numbers / Bell polynomials

The Bell numbers satisfy $B_0 = 1$ and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by $B_0(x) = 1$ and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called "partial" Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_2+\cdots=k \\ j_1+2j_2+3j_2+\cdots=n}} \frac{n!}{j_1! j_2! \cdots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \cdots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- bell(n) gives the n^{th} Bell number, B_n.
- bell(n, x) gives the n^{th} Bell polynomial, $B_n(x)$.
- bell(n, k, (x1, x2, ...)) gives Bell polynomials of the second kind, $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$.

Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

Examples

```
>>> from sympy import bell, Symbol, symbols
```

```
>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
>>> bell(30)
846749014511809332450147
>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t
>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

See also:

bernoulli (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R193], [R194], [R195]

bernoulli

class sympy.functions.combinatorial.numbers.**bernoulli**(*n*, *sym=None*)

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^{n} \binom{n+1}{k} B_k$$

They are also commonly defined by their exponential generating function, which is $\frac{x}{e^x - 1}$. For odd indices > 1, the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = \frac{A(n) - S(n)}{\binom{n+3}{n}}$$

where:

$$A(n) = \begin{cases} \frac{n+3}{3} & n \equiv 0 \text{ or } 2 \pmod{6} \\ -\frac{n+3}{6} & n \equiv 4 \pmod{6} \end{cases}$$

and:

$$S(n) = \sum_{k=1}^{[n/6]} \binom{n+3}{n-6k} B_{n-6k}$$

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the nth Bernoulli number, B_n
- `bernoulli(n, x)` gives the nth Bernoulli polynomial in x, $B_n(x)$

Examples

```
>>> from sympy import bernoulli
```

```
>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]
>>> bernoulli(1000001)
0
```

See also:

bell (page 427), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R196], [R197], [R198], [R199]

binomial

class sympy.functions.combinatorial.factorials.**binomial**(*n*, *k*)

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ or } \binom{n}{k} = \frac{(n)_k}{k!}$$

First, in a strict combinatorial sense it defines the number of ways we can choose k elements from a set of n elements. In this case both arguments are nonnegative integers and binomial is computed using an efficient algorithm based on prime factorization.

The other definition is generalization for arbitrary n, however k must also be nonnegative. This case is very useful when evaluating summations.

For the sake of convenience, for negative integer k this function will return zero no matter the other argument.

To expand the binomial when n is a symbol, use either `expand_func()` or `expand(func=True)`. The former will keep the polynomial in factored form while the latter will expand the polynomial itself. See examples for details.

Examples

```
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)
```

```
>>> binomial(15, 8)
6435
```

```
>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
...     print([binomial(N, i) for i in range(N + 1)])
...
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```
>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128
```

```
>>> binomial(n, 3)
binomial(n, 3)
```

```
>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3
```

```
>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```

References

[R200]

catalan

class sympy.functions.combinatorial.numbers.**catalan**(*n*)

Catalan numbers

The n^{th} catalan number is given by:

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

- catalan(n) gives the n^{th} Catalan number, C_n

Examples

```
>>> from sympy import (Symbol, binomial, gamma, hyper,
...     catalan, diff, combsimp, Rational, I)
```

```
>>> [catalan(i) for i in range(1,10)]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> n = Symbol("n", integer=True)
```

```
>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)
```

```
>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```
>>> catalan(n).rewrite(hyper)
hyper((1 - n, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1, 2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers C(n) interpreted as a continuous real function in n:

```
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)
4**I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

See also:

bell (page 427), *bernoulli* (page 428), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438), *sympy.functions.combinatorial.factorials.binomial* (page 430)

References

[R201], [R202], [R203], [R204]

euler

class `sympy.functions.combinatorial.numbers.euler(m, sym=None)`

Euler numbers / Euler polynomials

The Euler numbers are given by:

$$E_{2n} = I \sum_{k=1}^{2n+1} \sum_{j=0}^{k} \binom{k}{j} \frac{(-1)^j (k-2j)^{2n+1}}{2^k I^k k}$$

$$E_{2n+1} = 0$$

Euler numbers and Euler polynomials are related by

$$E_n = 2^n E_n \left(\frac{1}{2}\right).$$

We compute symbolic Euler polynomials using [R209]

$$E_n(x) = \sum_{k=0}^{n} \binom{n}{k} \frac{E_k}{2^k} \left(x - \frac{1}{2}\right)^{n-k}.$$

However, numerical evaluation of the Euler polynomial is computed more efficiently (and more accurately) using the mpmath library.

- `euler(n)` gives the n^{th} Euler number, E_n.
- `euler(n, x)` gives the n^{th} Euler polynomial, $E_n(x)$.

Examples

```
>>> from sympy import euler, Symbol, S
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n + 2*n)
euler(3*n)
```

```
>>> x = Symbol("x")
>>> euler(n, x)
euler(n, x)
```

```
>>> euler(0, x)
1
>>> euler(1, x)
x - 1/2
>>> euler(2, x)
x**2 - x
>>> euler(3, x)
x**3 - 3*x**2/2 + 1/4
>>> euler(4, x)
x**4 - 2*x**3 + x
```

```
>>> euler(12, S.Half)
2702765/4096
>>> euler(12)
2702765
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R205], [R206], [R207], [R208], [R209]

factorial

class sympy.functions.combinatorial.factorials.**factorial**(*n*)

Implementation of factorial function over nonnegative integers. By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which n objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact $n! = gamma(n + 1)$ for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments a precomputed look up table is used. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes $n!$ via prime factorization of special class of numbers, called here the 'Swing Numbers'.

Examples

```
>>> from sympy import Symbol, factorial, S
>>> n = Symbol('n', integer=True)
```

```
>>> factorial(0)
1
```

```
>>> factorial(7)
5040
```

```
>>> factorial(-2)
zoo
```

```
>>> factorial(n)
factorial(n)
```

```
>>> factorial(2*n)
factorial(2*n)
```

```
>>> factorial(S(1)/2)
factorial(1/2)
```

See also:

factorial2 (page 436), *RisingFactorial* (page 443), *FallingFactorial* (page 436)

subfactorial

class sympy.functions.combinatorial.factorials.**subfactorial**(*arg*)

The subfactorial counts the derangements of n items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(!(n-1) + !(n-2)) & n > 1 \end{cases}$$

It can also be written as `int(round(n!/exp(1)))` but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following [R211]

$$!x = \Gamma(x+1, -1)/e$$

which is valid for non-negative integers x. The above formula is not very useful in case of non-integers. $\Gamma(x+1, -1)$ is single-valued only for integral arguments x, elsewhere on the positive real axis it has an infinite number of branches none of which are real.

Examples

```
>>> from sympy import subfactorial
>>> from sympy.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

See also:

factorial (page 434), *uppergamma* (page 466), *sympy.utilities.iterables.generate_derangements* (page 2076)

References

[R210], [R211]

factorial2 / double factorial

class sympy.functions.combinatorial.factorials.**factorial2**(*arg*)

The double factorial $n!!$, not to be confused with $(n!)!$

The double factorial is defined for nonnegative integers and for odd negative integers as:

$$n!! = \begin{cases} 1 & n = 0 \\ n(n-2)(n-4)\cdots 1 & n \text{ positive odd} \\ n(n-2)(n-4)\cdots 2 & n \text{ positive even} \\ (n+2)!!/(n+2) & n \text{ negative odd} \end{cases}$$

Examples

```
>>> from sympy import factorial2, var
>>> n = var('n')
>>> n
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

See also:

factorial (page 434), *RisingFactorial* (page 443), *FallingFactorial* (page 436)

References

[R212]

FallingFactorial

class sympy.functions.combinatorial.factorials.**FallingFactorial**(*x*, *k*)

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

$$\texttt{ff(x, k)} = (x)_k = x \cdot (x-1) \cdots (x-k+1)$$

where x can be arbitrary expression and k is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or [R213].