When x is a $.Poly$ instance of degree ≥ 1 with single variable, $(x)_k = x(y) \cdot x(y-1) \cdots x(y-k+1)$, where y is the variable of x. This is as described in

```
>>> from sympy import ff, Poly, Symbol
>>> from sympy.abc import x
>>> n = Symbol('n', integer=True)
```

```
>>> ff(x, 0)
1
>>> ff(5, 5)
120
>>> ff(x, 5) == x*(x - 1)*(x - 2)*(x - 3)*(x - 4)
True
>>> ff(Poly(x**2, x), 2)
Poly(x**4 - 2*x**3 + x**2, x, domain='ZZ')
>>> ff(n, n)
factorial(n)
```

Rewriting is complicated unless the relationship between the arguments is known, but falling factorial can be rewritten in terms of gamma, factorial and binomial and rising factorial.

```
>>> from sympy import factorial, rf, gamma, binomial, Symbol
>>> n = Symbol('n', integer=True, positive=True)
>>> F = ff(n, n - 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...  F.rewrite(i)
...
RisingFactorial(3, n - 2)
FallingFactorial(n, n - 2)
factorial(n)/2
binomial(n, n - 2)*factorial(n - 2)
gamma(n + 1)/2
```

See also:

factorial (page 434), *factorial2* (page 436), *RisingFactorial* (page 443)

References

[R213], [R214]

fibonacci

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

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms $F_0 = 0$, $F_1 = 1$ and the two-term recurrence relation $F_n = F_{n-1} + F_{n-2}$. This definition extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by $F_1(x) = 1$, $F_2(x) = x$, and $F_n(x) = x * F_{n-1}(x) + F_{n-2}(x)$ for $n > 2$. For all positive integers n, $F_n(1) = F_n$.

- `fibonacci(n)` gives the n^{th} Fibonacci number, F_n
- `fibonacci(n, x)` gives the n^{th} Fibonacci polynomial in x, $F_n(x)$

Examples

```
>>> from sympy import fibonacci, Symbol
```

```
>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R216], [R217]

tribonacci

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

Tribonacci numbers / Tribonacci polynomials

The Tribonacci numbers are the integer sequence defined by the initial terms $T_0 = 0$, $T_1 = 1$, $T_2 = 1$ and the three-term recurrence relation $T_n = T_{n-1} + T_{n-2} + T_{n-3}$.

The Tribonacci polynomials are defined by $T_0(x) = 0$, $T_1(x) = 1$, $T_2(x) = x^2$, and $T_n(x) = x^2 T_{n-1}(x) + x T_{n-2}(x) + T_{n-3}(x)$ for $n > 2$. For all positive integers n, $T_n(1) = T_n$.

- `tribonacci(n)` gives the n^{th} Tribonacci number, T_n
- `tribonacci(n, x)` gives the n^{th} Tribonacci polynomial in x, $T_n(x)$

Examples

```
>>> from sympy import tribonacci, Symbol
```

```
>>> [tribonacci(x) for x in range(11)]
[0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149]
>>> tribonacci(5, Symbol('t'))
t**8 + 3*t**5 + 3*t**2
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442)

References

[R218], [R219], [R220]

harmonic

class sympy.functions.combinatorial.numbers.**harmonic**(*n*, *m=None*)

Harmonic numbers

The nth harmonic number is given by $\mathrm{H}_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.

More generally:

$$\mathrm{H}_{n,m} = \sum_{k=1}^{n} \frac{1}{k^m}$$

As $n \to \infty$, $\mathrm{H}_{n,m} \to \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the nth harmonic number, H_n
- `harmonic(n, m)` gives the nth generalized harmonic number of order m, $\mathrm{H}_{n,m}$, where `harmonic(n) == harmonic(n, 1)`

Examples

```
>>> from sympy import harmonic, oo
```

```
>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]
>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]
>>> harmonic(oo, 2)
pi**2/6
```

```
>>> from sympy import Symbol, Sum
>>> n = Symbol("n")
```

```
>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> from sympy import S, expand_func, simplify
>>> harmonic(8)
761/280
```

(continues on next page)

(continued from previous page)

```
>>> harmonic(11)
83711/27720
```

```
>>> H = harmonic(1/S(3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(_k*pi/3))*cos(2*_k*pi/3), (_k, 1,␣
↪1))
                       + 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3
>>> H = harmonic(25/S(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(2*pi/7)**(2*cos(16*pi/7))/(14*sin(pi/7)**(2*cos(pi/7))*cos(pi/
↪14)**(2*sin(pi/14)))) + pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/S(7)).n(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> from sympy import digamma, polygamma
>>> m = Symbol("m")
```

```
>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n,3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2
```

```
>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)
```

Integer offsets in the argument can be pulled out:

```
>>> from sympy import expand_func
```

```
>>> expand_func(harmonic(n+4))
harmonic(n) + 1/(n + 4) + 1/(n + 3) + 1/(n + 2) + 1/(n + 1)
```

```
>>> expand_func(harmonic(n-4))
harmonic(n) - 1/(n - 1) - 1/(n - 2) - 1/(n - 3) - 1/n
```

Some limits can be computed as well:

```
>>> from sympy import limit, oo
```

```
>>> limit(harmonic(n), n, oo)
oo
```

```
>>> limit(harmonic(n, 2), n, oo)
pi**2/6
```

```
>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we cannot compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals `zeta(m)` for `m > 1`.

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *lucas* (page 441), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R221], [R222], [R223]

lucas

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

Lucas numbers

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values $L_0 = 2$ and $L_1 = 1$.

- `lucas(n)` gives the n^{th} Lucas number

Examples

```
>>> from sympy import lucas
```

```
>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *genocchi* (page 442), *partition* (page 442), *tribonacci* (page 438)

References

[R224], [R225]

genocchi

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

Genocchi numbers

The Genocchi numbers are a sequence of integers G_n that satisfy the relation:

$$\frac{2t}{e^t + 1} = \sum_{n=1}^{\infty} \frac{G_n t^n}{n!}$$

Examples

```
>>> from sympy import genocchi, Symbol
>>> [genocchi(n) for n in range(1, 9)]
[1, -1, 0, 1, 0, -3, 0, 17]
>>> n = Symbol('n', integer=True, positive=True)
>>> genocchi(2*n + 1)
0
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *partition* (page 442), *tribonacci* (page 438)

References

[R226], [R227]

partition

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

Partition numbers

The Partition numbers are a sequence of integers p_n that represent the number of distinct ways of representing n as a sum of natural numbers (with order irrelevant). The generating function for p_n is given by:

$$\sum_{n=0}^{\infty} p_n x^n = \prod_{k=1}^{\infty} (1 - x^k)^{-1}$$

Examples

```
>>> from sympy import partition, Symbol
>>> [partition(n) for n in range(9)]
[1, 1, 2, 3, 5, 7, 11, 15, 22]
>>> n = Symbol('n', integer=True, negative=True)
>>> partition(n)
0
```

See also:

bell (page 427), *bernoulli* (page 428), *catalan* (page 431), *euler* (page 433), *fibonacci* (page 437), *harmonic* (page 439), *lucas* (page 441), *genocchi* (page 442), *tribonacci* (page 438)

References

[R228], [R229]

MultiFactorial

class sympy.functions.combinatorial.factorials.**MultiFactorial**(**args*)

RisingFactorial

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

Rising factorial (also called Pochhammer symbol [R230]) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by:

$$\texttt{rf(y, k)} = (x)^k = x \cdot (x+1) \cdots (x+k-1)$$

where x can be arbitrary expression and k is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit http://mathworld.wolfram.com/RisingFactorial.html page.

When x is a *.Poly* instance of degree ≥ 1 with a single variable, $(x)^k = x(y) \cdot x(y+1) \cdots x(y+k-1)$, where y is the variable of x. This is as described in [R231].

Examples

```
>>> from sympy import rf, Poly
>>> from sympy.abc import x
>>> rf(x, 0)
1
>>> rf(1, 5)
120
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
```

(continues on next page)

(continued from previous page)

```
True
>>> rf(Poly(x**3, x), 2)
Poly(x**6 + 3*x**5 + 3*x**4 + x**3, x, domain='ZZ')
```

Rewriting is complicated unless the relationship between the arguments is known, but rising factorial can be rewritten in terms of gamma, factorial, binomial, and falling factorial.

```
>>> from sympy import Symbol, factorial, ff, binomial, gamma
>>> n = Symbol('n', integer=True, positive=True)
>>> R = rf(n, n + 2)
>>> for i in (rf, ff, factorial, binomial, gamma):
...  R.rewrite(i)
...
RisingFactorial(n, n + 2)
FallingFactorial(2*n + 1, n + 2)
factorial(2*n + 1)/factorial(n - 1)
binomial(2*n + 1, n + 2)*factorial(n + 2)
gamma(2*n + 2)/gamma(n)
```

See also:

factorial (page 434), *factorial2* (page 436), *FallingFactorial* (page 436)

References

[R230], [R231]

stirling

sympy.functions.combinatorial.numbers.**stirling**(*n*, *k*, *d=None*, *kind=2*, *signed=False*)

Return Stirling number $S(n, k)$ of the first or second (default) kind.

The sum of all Stirling numbers of the second kind for $k = 1$ through n is `bell(n)`. The recurrence relationship for these numbers is:

$$\left\{ {0 \atop 0} \right\} = 1; \left\{ {n \atop 0} \right\} = \left\{ {0 \atop k} \right\} = 0;$$

$$\left\{ {n+1 \atop k} \right\} = j \left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\}$$

where j is:

n for Stirling numbers of the first kind, $-n$ for signed Stirling numbers of the first kind, k for Stirling numbers of the second kind.

The first kind of Stirling number counts the number of permutations of `n` distinct items that have `k` cycles; the second kind counts the ways in which `n` distinct items can be partitioned into `k` parts. If `d` is given, the "reduced Stirling number of the second kind" is returned: $S^d(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition n consecutive integers into k groups with no pairwise difference less than d. See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

Examples

```
>>> from sympy.functions.combinatorial.numbers import stirling, bell
>>> from sympy.combinatorics import Permutation
>>> from sympy.utilities.iterables import multiset_partitions,␣
↪permutations
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
>>> [stirling(4, i, kind=1) for i in range(5)]
[0, 6, 11, 6, 1]
```

First kind (signed):

```
>>> [stirling(4, i, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```

Second kind:

```
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```

Reduced second kind:

```
>>> from sympy import subsets, oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

See also:

sympy.utilities.iterables.multiset_partitions (page 2084)

References

[R232], [R233]

Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The k parameter indicates the number of elements to pick (or the number of partitions to make). When k is None, the sum of the enumeration for all k (from 0 through the number of items represented by n) is returned. A `replacement` parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.

```
>>> from sympy.functions.combinatorial.numbers import nC, nP, nT
>>> items = 'baby'
```

nC

sympy.functions.combinatorial.numbers.**nC**(*n*, *k=None*, *replacement=False*)

Return the number of combinations of n items taken k at a time.

Possible values for n:

integer - set of length n

sequence - converted to a multiset internally

multiset - {element: multiplicity}

If k is None then the total of all combinations of length 0 through the number of items represented in n will be returned.

If `replacement` is True then a given item can appear more than once in the k items. (For example, for 'ab' sets of 2 would include 'aa', 'ab', and 'bb'.) The multiplicity of elements in n is ignored when `replacement` is True but the total number of elements is considered since no element can appear more times than the number of elements in n.

Examples

```
>>> from sympy.functions.combinatorial.numbers import nC
>>> from sympy.utilities.iterables import multiset_combinations
>>> nC(3, 2)
3
>>> nC('abc', 2)
3
>>> nC('aab', 2)
2
```

When `replacement` is True, each item can have multiplicity equal to the length represented by n:

```
>>> nC('aabc', replacement=True)
35
>>> [len(list(multiset_combinations('aaaabbbbcccc', i))) for i in␣
↪range(5)]
[1, 3, 6, 10, 15]
>>> sum(_)
35
```

If there are k items with multiplicities `m_1, m_2, ..., m_k` then the total of all combinations of length 0 through k is the product, `(m_1 + 1)*(m_2 + 1)*...*(m_k + 1)`. When the multiplicity of each item is 1 (i.e., k unique items) then there are 2**k combinations. For example, if there are 4 unique items, the total number of combinations is 16:

```
>>> sum(nC(4, i) for i in range(5))
16
```

See also:

sympy.utilities.iterables.multiset_combinations (page 2083)

References

[R234], [R235]

nP

sympy.functions.combinatorial.numbers.**nP**(*n*, *k=None*, *replacement=False*)

Return the number of permutations of n items taken k at a time.

Possible values for n:

integer - set of length n

sequence - converted to a multiset internally

multiset - {element: multiplicity}

If k is None then the total of all permutations of length 0 through the number of items represented by n will be returned.

If `replacement` is True then a given item can appear more than once in the k items. (For example, for 'ab' permutations of 2 would include 'aa', 'ab', 'ba' and 'bb'.) The multiplicity of elements in n is ignored when `replacement` is True but the total number of elements is considered since no element can appear more times than the number of elements in n.

Examples

```
>>> from sympy.functions.combinatorial.numbers import nP
>>> from sympy.utilities.iterables import multiset_permutations, multiset
>>> nP(3, 2)
6
>>> nP('abc', 2) == nP(multiset('abc'), 2) == 6
True
>>> nP('aab', 2)
3
>>> nP([1, 2, 2], 2)
3
>>> [nP(3, i) for i in range(4)]
[1, 3, 6, 6]
>>> nP(3) == sum(_)
True
```

When `replacement` is True, each item can have multiplicity equal to the length represented by `n`:

```
>>> nP('aabc', replacement=True)
121
>>> [len(list(multiset_permutations('aaaabbbbcccc', i))) for i in␣
↪range(5)]
[1, 3, 9, 27, 81]
>>> sum(_)
121
```

See also:

sympy.utilities.iterables.multiset_permutations (page 2086)

References

[R236]

nT

`sympy.functions.combinatorial.numbers.`**nT**(*n*, *k=None*)

Return the number of `k`-sized partitions of `n` items.

Possible values for `n`:

integer - `n` identical items

sequence - converted to a multiset internally

multiset - {element: multiplicity}

Note: the convention for `nT` is different than that of `nC` and `nP` in that here an integer indicates `n` *identical* items instead of a set of length `n`; this is in keeping with the `partitions` function which treats its integer-`n` input like a list of `n` 1s. One can use `range(n)` for `n` to indicate `n` distinct items.

If k is None then the total number of ways to partition the elements represented in n will be returned.

Examples

```
>>> from sympy.functions.combinatorial.numbers import nT
```

Partitions of the given multiset:

```
>>> [nT('aabbc', i) for i in range(1, 7)]
[1, 8, 11, 5, 1, 0]
>>> nT('aabbc') == sum(_)
True
```

```
>>> [nT("mississippi", i) for i in range(1, 12)]
[1, 74, 609, 1521, 1768, 1224, 579, 197, 50, 9, 1]
```

Partitions when all items are identical:

```
>>> [nT(5, i) for i in range(1, 6)]
[1, 2, 2, 1, 1]
>>> nT('1'*5) == sum(_)
True
```

When all items are different:

```
>>> [nT(range(5), i) for i in range(1, 6)]
[1, 15, 25, 10, 1]
>>> nT(range(5)) == sum(_)
True
```

Partitions of an integer expressed as a sum of positive integers:

```
>>> from sympy import partition
>>> partition(4)
5
>>> nT(4, 1) + nT(4, 2) + nT(4, 3) + nT(4, 4)
5
>>> nT('1'*4)
5
```

See also:

sympy.utilities.iterables.partitions (page 2089), *sympy.utilities.iterables.multiset_partitions* (page 2084), *sympy.functions.combinatorial.numbers.partition* (page 442)

References

[R237]

Special

DiracDelta

class sympy.functions.special.delta_functions.**DiracDelta**(*arg*, *k=0*)

The DiracDelta function and its derivatives.

Explanation

DiracDelta is not an ordinary function. It can be rigorously defined either as a distribution or as a measure.

DiracDelta only makes sense in definite integrals, and in particular, integrals of the form `Integral(f(x)*DiracDelta(x - x0), (x, a, b))`, where it equals `f(x0)` if `a <= x0 <= b` and `0` otherwise. Formally, DiracDelta acts in some ways like a function that is `0` everywhere except at `0`, but in many ways it also does not. It can often be useful to treat DiracDelta in formal ways, building up and manipulating expressions with delta functions (which may eventually be integrated), but care must be taken to not treat it as a real function. SymPy's `oo` is similar. It only truly makes sense formally in certain contexts (such as integration limits), but SymPy allows its use everywhere, and it tries to be consistent with operations on it (like `1/oo`), but it is easy to get into trouble and get wrong results if `oo` is treated too much like a number. Similarly, if DiracDelta is treated too much like a function, it is easy to get wrong or nonsensical results.

DiracDelta function has the following properties:

1) $\frac{d}{dx}\theta(x) = \delta(x)$
2) $\int_{-\infty}^{\infty} \delta(x-a)f(x)\,dx = f(a)$ and $\int_{a-\epsilon}^{a+\epsilon} \delta(x-a)f(x)\,dx = f(a)$
3) $\delta(x) = 0$ for all $x \neq 0$
4) $\delta(g(x)) = \sum_i \frac{\delta(x-x_i)}{\|g'(x_i)\|}$ where x_i are the roots of g
5) $\delta(-x) = \delta(x)$

Derivatives of k-th order of DiracDelta have the following properties:

6) $\delta(x, k) = 0$ for all $x \neq 0$
7) $\delta(-x, k) = -\delta(x, k)$ for odd k
8) $\delta(-x, k) = \delta(x, k)$ for even k

Examples

```
>>> from sympy import DiracDelta, diff, pi
>>> from sympy.abc import x, y
```

```
>>> DiracDelta(x)
DiracDelta(x)
>>> DiracDelta(1)
0
>>> DiracDelta(-1)
0
>>> DiracDelta(pi)
0
>>> DiracDelta(x - 4).subs(x, 4)
DiracDelta(0)
>>> diff(DiracDelta(x))
DiracDelta(x, 1)
>>> diff(DiracDelta(x - 1), x, 2)
DiracDelta(x - 1, 2)
>>> diff(DiracDelta(x**2 - 1), x, 2)
2*(2*x**2*DiracDelta(x**2 - 1, 2) + DiracDelta(x**2 - 1, 1))
>>> DiracDelta(3*x).is_simple(x)
True
>>> DiracDelta(x**2).is_simple(x)
False
>>> DiracDelta((x**2 - 1)*y).expand(diracdelta=True, wrt=x)
DiracDelta(x - 1)/(2*Abs(y)) + DiracDelta(x + 1)/(2*Abs(y))
```

See also:

Heaviside (page 454), *sympy.simplify.simplify.simplify* (page 661), *is_simple* (page 453), *sympy.functions.special.tensor_functions.KroneckerDelta* (page 549)

References

[R304]

classmethod eval(*arg*, *k=0*)

Returns a simplified form or a value of DiracDelta depending on the argument passed by the DiracDelta object.

Parameters

k : integer

order of derivative

arg : argument passed to DiracDelta

Explanation

The `eval()` method is automatically called when the `DiracDelta` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import DiracDelta, S
>>> from sympy.abc import x
```

```
>>> DiracDelta(x)
DiracDelta(x)
```

```
>>> DiracDelta(-x, 1)
-DiracDelta(x, 1)
```

```
>>> DiracDelta(1)
0
```

```
>>> DiracDelta(5, 1)
0
```

```
>>> DiracDelta(0)
DiracDelta(0)
```

```
>>> DiracDelta(-1)
0
```

```
>>> DiracDelta(S.NaN)
nan
```

```
>>> DiracDelta(x - 100).subs(x, 5)
0
```

```
>>> DiracDelta(x - 100).subs(x, 100)
DiracDelta(0)
```

fdiff(*argindex=1*)

Returns the first derivative of a DiracDelta Function.

Parameters

argindex : integer

degree of derivative

Explanation

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the `Function` class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Examples

```
>>> from sympy import DiracDelta, diff
>>> from sympy.abc import x
```

```
>>> DiracDelta(x).fdiff()
DiracDelta(x, 1)
```

```
>>> DiracDelta(x, 1).fdiff()
DiracDelta(x, 2)
```

```
>>> DiracDelta(x**2 - 1).fdiff()
DiracDelta(x**2 - 1, 1)
```

```
>>> diff(DiracDelta(x, 1)).fdiff()
DiracDelta(x, 3)
```

is_simple(*x*)

Tells whether the argument(args[0]) of DiracDelta is a linear expression in *x*.

Parameters

x : can be a symbol

Examples

```
>>> from sympy import DiracDelta, cos
>>> from sympy.abc import x, y
```

```
>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True
```

```
>>> DiracDelta(x**2 + x - 2).is_simple(x)
False
```

```
>>> DiracDelta(cos(x)).is_simple(x)
False
```

See also:

sympy.simplify.simplify.simplify (page 661), *DiracDelta* (page 450)

Heaviside

class sympy.functions.special.delta_functions.**Heaviside**(*arg*, *H0=1 / 2*)

Heaviside step function.

Explanation

The Heaviside step function has the following properties:

1) $\frac{d}{dx}\theta(x) = \delta(x)$

2) $\theta(x) = \begin{cases} 0 & \text{for } x < 0 \\ \frac{1}{2} & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases}$

3) $\frac{d}{dx} \max(x, 0) = \theta(x)$

Heaviside(x) is printed as $\theta(x)$ with the SymPy LaTeX printer.

The value at 0 is set differently in different fields. SymPy uses 1/2, which is a convention from electronics and signal processing, and is consistent with solving improper integrals by Fourier transform and convolution.

To specify a different value of Heaviside at `x=0`, a second argument can be given. Using `Heaviside(x, nan)` gives an expression that will evaluate to nan for x=0.

Changed in version 1.9: `Heaviside(0)` now returns 1/2 (before: undefined)

Examples

```
>>> from sympy import Heaviside, nan
>>> from sympy.abc import x
>>> Heaviside(9)
1
>>> Heaviside(-9)
0
>>> Heaviside(0)
1/2
>>> Heaviside(0, nan)
nan
>>> (Heaviside(x) + 1).replace(Heaviside(x), Heaviside(x, 1))
Heaviside(x, 1) + 1
```

See also:

DiracDelta (page 450)

References

[R305], [R306]

classmethod eval(*arg*, *H0=1 / 2*)

Returns a simplified form or a value of Heaviside depending on the argument passed by the Heaviside object.

Parameters

arg : argument passed by Heaviside object

H0 : value of Heaviside(0)

Explanation

The `eval()` method is automatically called when the `Heaviside` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import Heaviside, S
>>> from sympy.abc import x
```

```
>>> Heaviside(x)
Heaviside(x)
```

```
>>> Heaviside(19)
1
```

```
>>> Heaviside(0)
1/2
```

```
>>> Heaviside(0, 1)
1
```

```
>>> Heaviside(-5)
0
```

```
>>> Heaviside(S.NaN)
nan
```

```
>>> Heaviside(x - 100).subs(x, 5)
0
```

```
>>> Heaviside(x - 100).subs(x, 105)
1
```

fdiff(*argindex=1*)

Returns the first derivative of a Heaviside Function.

Parameters

argindex : integer

order of derivative

Examples

```
>>> from sympy import Heaviside, diff
>>> from sympy.abc import x
```

```
>>> Heaviside(x).fdiff()
DiracDelta(x)
```

```
>>> Heaviside(x**2 - 1).fdiff()
DiracDelta(x**2 - 1)
```

```
>>> diff(Heaviside(x)).fdiff()
DiracDelta(x, 1)
```

property pargs

Args without default S.Half

Singularity Function

class sympy.functions.special.singularity_functions.**SingularityFunction**(*variable, offset, exponent*)

Singularity functions are a class of discontinuous functions.

Explanation

Singularity functions take a variable, an offset, and an exponent as arguments. These functions are represented using Macaulay brackets as:

SingularityFunction(x, a, n) := <x - a>^n

The singularity function will automatically evaluate to `Derivative(DiracDelta(x - a), x, -n - 1)` if `n < 0` and `(x - a)**n*Heaviside(x - a)` if `n >= 0`.

Examples

```
>>> from sympy import SingularityFunction, diff, Piecewise, DiracDelta,␣
↪Heaviside, Symbol
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> SingularityFunction(y, -10, n)
(y + 10)**n
>>> y = Symbol('y', negative=True)
>>> SingularityFunction(y, 10, n)
0
>>> SingularityFunction(x, 4, -1).subs(x, 4)
oo
>>> SingularityFunction(x, 10, -2).subs(x, 10)
oo
>>> SingularityFunction(4, 1, 5)
243
>>> diff(SingularityFunction(x, 1, 5) + SingularityFunction(x, 1, 4), x)
4*SingularityFunction(x, 1, 3) + 5*SingularityFunction(x, 1, 4)
>>> diff(SingularityFunction(x, 4, 0), x, 2)
SingularityFunction(x, 4, -2)
>>> SingularityFunction(x, 4, 5).rewrite(Piecewise)
Piecewise(((x - 4)**5, x > 4), (0, True))
>>> expr = SingularityFunction(x, a, n)
>>> y = Symbol('y', positive=True)
>>> n = Symbol('n', nonnegative=True)
>>> expr.subs({x: y, a: -10, n: n})
(y + 10)**n
```

The methods rewrite(DiracDelta), rewrite(Heaviside), and rewrite('HeavisideDiracDelta') returns the same output. One can use any of these methods according to their choice.

```
>>> expr = SingularityFunction(x, 4, 5) + SingularityFunction(x, -3, -1)␣
↪- SingularityFunction(x, 0, -2)
>>> expr.rewrite(Heaviside)
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite(DiracDelta)
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
>>> expr.rewrite('HeavisideDiracDelta')
(x - 4)**5*Heaviside(x - 4) + DiracDelta(x + 3) - DiracDelta(x, 1)
```

See also:

DiracDelta (page 450), *Heaviside* (page 454)

References

[R307]

classmethod eval(*variable, offset, exponent*)

Returns a simplified form or a value of Singularity Function depending on the argument passed by the object.

Explanation

The `eval()` method is automatically called when the `SingularityFunction` class is about to be instantiated and it returns either some simplified instance or the unevaluated instance depending on the argument passed. In other words, `eval()` method is not needed to be called explicitly, it is being called and evaluated once the object is called.

Examples

```
>>> from sympy import SingularityFunction, Symbol, nan
>>> from sympy.abc import x, a, n
>>> SingularityFunction(x, a, n)
SingularityFunction(x, a, n)
>>> SingularityFunction(5, 3, 2)
4
>>> SingularityFunction(x, a, nan)
nan
>>> SingularityFunction(x, 3, 0).subs(x, 3)
1
>>> SingularityFunction(4, 1, 5)
243
>>> x = Symbol('x', positive = True)
>>> a = Symbol('a', negative = True)
>>> n = Symbol('n', nonnegative = True)
>>> SingularityFunction(x, a, n)
(-a + x)**n
>>> x = Symbol('x', negative = True)
>>> a = Symbol('a', positive = True)
>>> SingularityFunction(x, a, n)
0
```

fdiff(*argindex=1*)

Returns the first derivative of a DiracDelta Function.

Explanation

The difference between `diff()` and `fdiff()` is: `diff()` is the user-level function and `fdiff()` is an object method. `fdiff()` is a convenience method available in the `Function` class. It returns the derivative of the function without considering the chain rule. `diff(function, x)` calls `Function._eval_derivative` which in turn calls `fdiff()` internally to compute the derivative of the function.

Gamma, Beta and related Functions

class sympy.functions.special.gamma_functions.**gamma**(*arg*)

The gamma function

$$\Gamma(x) := \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t.$$

Explanation

The `gamma` function implements the function which passes through the values of the factorial function (i.e., $\Gamma(n) = (n-1)!$ when n is an integer). More generally, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

Examples

```
>>> from sympy import S, I, pi, gamma
>>> from sympy.abc import x
```

Several special values are known:

```
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2
```

The `gamma` function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-
↪EulerGamma*pi**2/12 + polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.2880377953400324179595889090606233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

See also:

lowergamma **(page 468)**
Lower incomplete gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

trigamma **(page 465)**
Trigamma function.

beta **(page 470)**
Euler Beta function.

References

[R308], [R309], [R310], [R311]

class sympy.functions.special.gamma_functions.**loggamma**(z)

The loggamma function implements the logarithm of the gamma function (i.e., $\log\Gamma(x)$).

Examples

Several special values are known. For numerical integral arguments we have:

```
>>> from sympy import loggamma
>>> loggamma(-2)
oo
>>> loggamma(0)
oo
>>> loggamma(1)
0
```

(continues on next page)

(continued from previous page)

```
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

And for symbolic values:

```
>>> from sympy import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

For half-integral values:

```
>>> from sympy import S
>>> loggamma(S(5)/2)
log(3*sqrt(pi)/4)
>>> loggamma(n/2)
log(2**(1 - n)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

And general rational arguments:

```
>>> from sympy import expand_func
>>> L = loggamma(S(16)/3)
>>> expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
>>> L = loggamma(S(19)/4)
>>> expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
>>> L = loggamma(S(23)/7)
>>> expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The `loggamma` function has the following limits towards infinity:

```
>>> from sympy import oo
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The `loggamma` function obeys the mirror symmetry if $x \in \mathbb{C} \setminus \{-\infty, 0\}$:

```
>>> from sympy.abc import x
>>> from sympy import conjugate
>>> conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(loggamma(x), x, 0, 4).cancel()
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import I
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

See also:

***gamma* (page 459)**
Gamma function.

***lowergamma* (page 468)**
Lower incomplete gamma function.

***uppergamma* (page 466)**
Upper incomplete gamma function.

***polygamma* (page 462)**
Polygamma function.

***digamma* (page 464)**
Digamma function.

***trigamma* (page 465)**
Trigamma function.

***beta* (page 470)**
Euler Beta function.

References

[R312], [R313], [R314], [R315]

class sympy.functions.special.gamma_functions.**polygamma**(*n*, *z*)

The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`.

Explanation

It is a meromorphic function on $\mathbb{C}$ and defined as the $(n+1)$-th derivative of the logarithm of the gamma function:

$$\psi^{(n)}(z) := \frac{\mathrm{d}^{n+1}}{\mathrm{d}z^{n+1}} \log\Gamma(z).$$

Examples

Several special values are known:

```
>>> from sympy import S, polygamma
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, 1/S(2))
-2*log(2) - EulerGamma
>>> polygamma(0, 1/S(3))
-log(3) - sqrt(3)*pi/6 - EulerGamma - log(sqrt(3))
>>> polygamma(0, 1/S(4))
-pi/2 - log(4) - log(2) - EulerGamma
>>> polygamma(0, 2)
1 - EulerGamma
>>> polygamma(0, 23)
19093197/5173168 - EulerGamma
```

```
>>> from sympy import oo, I
>>> polygamma(0, oo)
oo
>>> polygamma(0, -oo)
oo
>>> polygamma(0, I*oo)
oo
>>> polygamma(0, -I*oo)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import Symbol, diff
>>> x = Symbol("x")
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
```

(continues on next page)

(continued from previous page)

```
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)
```

```
>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)
```

We can rewrite `polygamma` functions in terms of harmonic numbers:

```
>>> from sympy import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)
```

See also:

gamma **(page 459)**
Gamma function.

lowergamma **(page 468)**
Lower incomplete gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

trigamma **(page 465)**
Trigamma function.

beta **(page 470)**
Euler Beta function.

References

[R316], [R317], [R318], [R319]

class sympy.functions.special.gamma_functions.**digamma**(*z*)

The `digamma` function is the first derivative of the `loggamma` function

$$\psi(x) := \frac{\mathrm{d}}{\mathrm{d}z} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

In this case, `digamma(z) = polygamma(0, z)`.

Examples

```
>>> from sympy import digamma
>>> digamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> digamma(z)
polygamma(0, z)
```

To retain `digamma` as it is:

```
>>> digamma(0, evaluate=False)
digamma(0)
>>> digamma(z, evaluate=False)
digamma(z)
```

See also:

gamma **(page 459)**
Gamma function.

lowergamma **(page 468)**
Lower incomplete gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

trigamma **(page 465)**
Trigamma function.

beta **(page 470)**
Euler Beta function.

References

[R320], [R321], [R322]

class sympy.functions.special.gamma_functions.**trigamma**(*z*)

The `trigamma` function is the second derivative of the `loggamma` function

$$\psi^{(1)}(z) := \frac{\mathrm{d}^2}{\mathrm{d}z^2} \log \Gamma(z).$$

In this case, `trigamma(z) = polygamma(1, z)`.

Examples

```
>>> from sympy import trigamma
>>> trigamma(0)
zoo
>>> from sympy import Symbol
>>> z = Symbol('z')
>>> trigamma(z)
polygamma(1, z)
```

To retain `trigamma` as it is:

```
>>> trigamma(0, evaluate=False)
trigamma(0)
>>> trigamma(z, evaluate=False)
trigamma(z)
```

See also:

gamma **(page 459)**
Gamma function.

lowergamma **(page 468)**
Lower incomplete gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

beta **(page 470)**
Euler Beta function.

References

[R323], [R324], [R325]

class sympy.functions.special.gamma_functions.**uppergamma**(*a*, *z*)
The upper incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

$$\Gamma(s,x) := \int_x^\infty t^{s-1}e^{-t}\mathrm{d}t = \Gamma(s) - \gamma(s,x).$$

where $\gamma(s,x)$ is the lower incomplete gamma function, *lowergamma* (page 468). This can be shown to be the same as

$$\Gamma(s,x) = \Gamma(s) - \frac{x^s}{s}{}_1F_1\left(\begin{matrix} s \\ s+1 \end{matrix}\middle| -x\right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$\mathrm{E}_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt = x^{n-1}\Gamma(1-n,x).$$

Examples

```
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
2*(x**2/2 + x + 1)*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*erfc(sqrt(x)) + 2*exp(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

See also:

gamma **(page 459)**
Gamma function.

lowergamma **(page 468)**
Lower incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

trigamma **(page 465)**
Trigamma function.

beta **(page 470)**
Euler Beta function.

References

[R326], [R327], [R328], [R329], [R330], [R331]

class sympy.functions.special.gamma_functions.**lowergamma**(*a*, *x*)

The lower incomplete gamma function.

Explanation

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} \mathrm{d}t = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1\left(\begin{matrix} s \\ s+1 \end{matrix}\middle| -x\right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

Examples

```
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-2*(x**2/2 + x + 1)*exp(-x) + 2
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

See also:

gamma **(page 459)**
Gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

trigamma **(page 465)**
Trigamma function.

beta **(page 470)**
Euler Beta function.

References

[R332], [R333], [R334], [R335], [R336]

class sympy.functions.special.gamma_functions.**multigamma**(*x*, *p*)

The multivariate gamma function is a generalization of the gamma function

$$\Gamma_p(z) = \pi^{p(p-1)/4} \prod_{k=1}^{p} \Gamma[z + (1 - k)/2].$$

In a special case, `multigamma(x, 1) = gamma(x)`.

Parameters

p : order or dimension of the multivariate gamma function

Examples

```
>>> from sympy import S, multigamma
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> p = Symbol('p', positive=True, integer=True)
```

```
>>> multigamma(x, p)
pi**(p*(p - 1)/4)*Product(gamma(-_k/2 + x + 1/2), (_k, 1, p))
```

Several special values are known:

```
>>> multigamma(1, 1)
1
>>> multigamma(4, 1)
6
>>> multigamma(S(3)/2, 1)
sqrt(pi)/2
```

Writing `multigamma` in terms of the `gamma` function:

```
>>> multigamma(x, 1)
gamma(x)
```

```
>>> multigamma(x, 2)
sqrt(pi)*gamma(x)*gamma(x - 1/2)
```

```
>>> multigamma(x, 3)
pi**(3/2)*gamma(x)*gamma(x - 1)*gamma(x - 1/2)
```

See also:

gamma (page 459), *lowergamma* (page 468), *uppergamma* (page 466), *polygamma* (page 462), *loggamma* (page 460), *digamma* (page 464), *trigamma* (page 465), *beta* (page 470)

References

[R337]

class sympy.functions.special.beta_functions.**beta**(*x*, *y=None*)

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$\mathrm{B}(x,y)\int_0^1 t^{x-1}(1-t)^{y-1}\mathrm{d}t.$$

Explanation

The Beta function or Euler's first integral is closely associated with the gamma function. The Beta function is often used in probability theory and mathematical statistics. It satisfies properties like:

$$\mathrm{B}(a,1) = \frac{1}{a}$$
$$\mathrm{B}(a,b) = \mathrm{B}(b,a)$$
$$\mathrm{B}(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

Therefore for integral values of a and b:

$$\mathrm{B} = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

A special case of the Beta function when $x = y$ is the Central Beta function. It satisfies properties like:

$$\mathrm{B}(x) = 2^{1-2x}\mathrm{B}(x,\frac{1}{2})\mathrm{B}(x) = 2^{1-2x}cos(\pi x)\mathrm{B}(\frac{1}{2}-x,x)\mathrm{B}(x) = \int_0^1 \frac{t^x}{(1+t)^{2x}}dt\mathrm{B}(x) = \frac{2}{x}\prod_{n=1}^{\infty}\frac{n(n+2x)}{(n+x)^2}$$

Examples

```
>>> from sympy import I, pi
>>> from sympy.abc import x, y
```

The Beta function obeys the mirror symmetry:

```
>>> from sympy import beta, conjugate
>>> conjugate(beta(x, y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> from sympy import beta, diff
>>> diff(beta(x, y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```
>>> diff(beta(x, y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

```
>>> diff(beta(x), x)
2*(polygamma(0, x) - polygamma(0, 2*x))*beta(x, x)
```

We can numerically evaluate the Beta function to arbitrary precision for any complex numbers x and y:

```
>>> from sympy import beta
>>> beta(pi).evalf(40)
0.02671848900111377452242355235388489324562
```

```
>>> beta(1 + I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

See also:

gamma **(page 459)**
Gamma function.

uppergamma **(page 466)**
Upper incomplete gamma function.

lowergamma **(page 468)**
Lower incomplete gamma function.

polygamma **(page 462)**
Polygamma function.

loggamma **(page 460)**
Log Gamma function.

digamma **(page 464)**
Digamma function.

trigamma **(page 465)**
Trigamma function.

References

[R338], [R339], [R340]

Error Functions and Fresnel Integrals

class sympy.functions.special.error_functions.**erf**(*arg*)
The Gauss error function.

Explanation

This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t.$$

Examples

```
>>> from sympy import I, oo, erf
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erf(0)
0
>>> erf(oo)
1
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erf(z), z)
2*exp(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:

```
>>> erf(4).evalf(30)
0.999999984582742099719981147840
```

```
>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

See also:

erfc **(page 473)**
Complementary error function.

erfi **(page 474)**
Imaginary error function.

erf2 **(page 476)**
Two-argument error function.

erfinv **(page 477)**
Inverse error function.

erfcinv **(page 478)**
Inverse Complementary error function.

erf2inv **(page 479)**
Inverse two-argument error function.

References

[R341], [R342], [R343], [R344]

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

class sympy.functions.special.error_functions.**erfc**(*arg*)
Complementary Error Function.

Explanation

The function is defined as:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} \mathrm{d}t$$

Examples

```
>>> from sympy import I, oo, erfc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfc(z), z)
-2*exp(-z**2)/sqrt(pi)
```

It also follows

```
>>> erfc(-z)
2 - erfc(z)
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```
>>> erfc(4).evalf(30)
0.000000015417257900280018852159673486 9
```

```
>>> erfc(4*I).evalf(30)
1.0 - 1296959.730717639232152794095062*I
```

See also:

***erf* (page 471)**
Gaussian error function.

***erfi* (page 474)**
Imaginary error function.

***erf2* (page 476)**
Two-argument error function.

***erfinv* (page 477)**
Inverse error function.

***erfcinv* (page 478)**
Inverse Complementary error function.

***erf2inv* (page 479)**
Inverse two-argument error function.

References

[R345], [R346], [R347], [R348]

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

class sympy.functions.special.error_functions.**erfi**(*z*)
Imaginary error function.

Explanation

The function erfi is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} \mathrm{d}t$$

Examples

```
>>> from sympy import I, oo, erfi
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erfi(-z)
-erfi(z)
```

```
>>> from sympy import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfi(z), z)
2*exp(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```
>>> erfi(2).evalf(30)
18.5648024145755525987042919132
```

```
>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

See also:

erf **(page 471)**
Gaussian error function.

erfc **(page 473)**
Complementary error function.

erf2 **(page 476)**
Two-argument error function.

erfinv **(page 477)**
Inverse error function.

erfcinv **(page 478)**
Inverse Complementary error function.

erf2inv **(page 479)**
Inverse two-argument error function.

References

[R349], [R350], [R351]

class sympy.functions.special.error_functions.**erf2**(*x*, *y*)
Two-argument error function.

Explanation

This function is defined as:

$$\mathrm{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} \mathrm{d}t$$

Examples

```
>>> from sympy import oo, erf2
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
1 - erf(x)
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
>>> erf2(-oo, y)
erf(y) + 1
```

In general one can pull out factors of -1:

```
>>> erf2(-x, -y)
-erf2(x, y)
```