NaN

class sympy.core.numbers.**NaN**

Not a Number.

Explanation

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as `0/0` or `oo - oo` produce NaN. Two exceptions are ``0**0`` and `oo**0`, which all produce `1` (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a TypeError. This is in contrast with floating point nan where all inequalities are false.

NaN is a singleton, and can be accessed by `S.NaN`, or can be imported as `nan`.

Examples

```
>>> from sympy import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)   # mathematical equality
False
>>> nan == nan     # structural equality
True
```

References

[R117]

Infinity

class sympy.core.numbers.**Infinity**

Positive infinite quantity.

Explanation

In real analysis the symbol ∞ denotes an unbounded limit: $x \to \infty$ means that x grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `S.Infinity`, or can be imported as `oo`.

Examples

```
>>> from sympy import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```


See also:

NegativeInfinity (page 998), *NaN* (page 997)

References

[R118]

NegativeInfinity

class sympy.core.numbers.**NegativeInfinity**

Negative infinite quantity.

NegativeInfinity is a singleton, and can be accessed by `S.NegativeInfinity`.

See also:

Infinity (page 998)

ComplexInfinity

class sympy.core.numbers.**ComplexInfinity**

Complex infinity.

Explanation

In complex analysis the symbol $\tilde\infty$, called "complex infinity", represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by `S.ComplexInfinity`, or can be imported as `zoo`.

Examples

```
>>> from sympy import zoo
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

See also:

Infinity (page 998)

Exp1

class sympy.core.numbers.**Exp1**

The e constant.

Explanation

The transcendental number $e = 2.718281828\ldots$ is the base of the natural logarithm and of the exponential function, $e = \exp(1)$. Sometimes called Euler's number or Napier's constant.

Exp1 is a singleton, and can be accessed by `S.Exp1`, or can be imported as `E`.

Examples

```
>>> from sympy import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

References

[R119]

ImaginaryUnit

class sympy.core.numbers.**ImaginaryUnit**

The imaginary unit, $i = \sqrt{-1}$.

I is a singleton, and can be accessed by `S.I`, or can be imported as `I`.

Examples

```
>>> from sympy import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

References

[R120]

Pi

class sympy.core.numbers.**Pi**

The π constant.

Explanation

The transcendental number $\pi = 3.141592654\ldots$ represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be accessed by `S.Pi`, or can be imported as `pi`.

Examples

```
>>> from sympy import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
True
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)
```


References

[R121]

EulerGamma

class sympy.core.numbers.**EulerGamma**

The Euler-Mascheroni constant.

Explanation

$\gamma = 0.5772157\ldots$ (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n\to\infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \ln n\right)$$

EulerGamma is a singleton, and can be accessed by `S.EulerGamma`.

Examples

```
>>> from sympy import S
>>> S.EulerGamma.is_irrational
>>> S.EulerGamma > 0
True
>>> S.EulerGamma > 1
False
```

References

[R122]

Catalan

class sympy.core.numbers.**Catalan**

Catalan's constant.

Explanation

$G = 0.91596559\ldots$ is given by the infinite series

$$G = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

Catalan is a singleton, and can be accessed by `S.Catalan`.

Examples

```
>>> from sympy import S
>>> S.Catalan.is_irrational
>>> S.Catalan > 0
True
>>> S.Catalan > 1
False
```

References

[R123]

GoldenRatio

class sympy.core.numbers.**GoldenRatio**

The golden ratio, ϕ.

Explanation

$\phi = \frac{1+\sqrt{5}}{2}$ is an algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

GoldenRatio is a singleton, and can be accessed by `S.GoldenRatio`.

Examples

```
>>> from sympy import S
>>> S.GoldenRatio > 1
True
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```


References

[R124]

TribonacciConstant

class sympy.core.numbers.**TribonacciConstant**

The tribonacci constant.

Explanation

The tribonacci numbers are like the Fibonacci numbers, but instead of starting with two predetermined terms, the sequence starts with three predetermined terms and each term afterwards is the sum of the preceding three terms.

The tribonacci constant is the ratio toward which adjacent tribonacci numbers tend. It is a root of the polynomial $x^3 - x^2 - x - 1 = 0$, and also satisfies the equation $x + x^{-3} = 2$.

TribonacciConstant is a singleton, and can be accessed by `S.TribonacciConstant`.

Examples

```
>>> from sympy import S
>>> S.TribonacciConstant > 1
True
>>> S.TribonacciConstant.expand(func=True)
1/3 + (19 - 3*sqrt(33))**(1/3)/3 + (3*sqrt(33) + 19)**(1/3)/3
>>> S.TribonacciConstant.is_irrational
True
>>> S.TribonacciConstant.n(20)
1.8392867552141611326
```

References

[R125]

mod_inverse

sympy.core.numbers.**mod_inverse**(*a*, *m*)

Return the number c such that, $a \times c = 1 \pmod{m}$ where c has the same sign as m. If no such value exists, a ValueError is raised.

Examples

```
>>> from sympy import mod_inverse, S
```


Suppose we wish to find multiplicative inverse x of 3 modulo 11. This is the same as finding x such that $3x = 1 \pmod{11}$. One value of x that satisfies this congruence is 4. Because $3 \times 4 = 12$ and $12 = 1 \pmod{11}$. This is the value returned by `mod_inverse`:

```
>>> mod_inverse(3, 11)
4
>>> mod_inverse(-3, 11)
7
```

When there is a common factor between the numerators of a and m the inverse does not exist:

```
>>> mod_inverse(2, 4)
Traceback (most recent call last):
...
ValueError: inverse of 2 mod 4 does not exist
```

```
>>> mod_inverse(S(2)/7, S(5)/2)
7/2
```

References

[R126], [R127]

power

Pow

class sympy.core.power.**Pow**(*b*, *e*, *evaluate=None*)

Defines the expression x**y as "x raised to a power y"

Deprecated since version 1.7: Using arguments that aren't subclasses of *Expr* (page 947) in core operators (*Mul* (page 1009), *Add* (page 1013), and *Pow* (page 1005)) is deprecated. See *Core operators no longer accept non-Expr args* (page 174) for details.

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

expr	value	reason
z**0	1	Although arguments over 0**0 exist, see [2].
z**1	z	
(-oo)**(-1)	0	
(-1)**-1	-1	
S.Zero**-1	zoo	This is not strictly true, as 0**-1 may be undefined, but is convenient in some contexts where the base is assumed to be positive.
1**-1	1	
oo**-1	0	
0**oo	0	Because for all complex numbers z near 0, z**oo -> 0.
0**-oo	zoo	This is not strictly true, as 0**oo may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
1**oo 1**-oo	nan	Because there are various cases where lim(x(t),t)=1, lim(y(t),t)=oo (or -oo), but lim(x(t)**y(t), t) != 1. See [3].
b**zoo	nan	Because b**z has no limit as z -> zoo
(-1)**oo (-1)**(-oo)	nan	Because of oscillations in the limit.
oo**oo	oo	
oo**-oo	0	
(-oo)**oo (-oo)**-oo	nan	
oo**I (-oo)**I	nan	oo**e could probably be best thought of as the limit of x**e for real x as x tends to oo. If e is I, then the limit does not exist and nan is used to indicate that.
oo**(1+I) (-oo)**(1+I)	zoo	If the real part of e is positive, then the limit of abs(x**e) is oo. So the limit value is zoo.
oo**(-1+I) -oo**(-1+I)	0	If the real part of e is negative, then the limit is 0.

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

See also:

sympy.core.numbers.Infinity (page 998), *sympy.core.numbers.NegativeInfinity* (page 998), *sympy.core.numbers.NaN* (page 997)

References

[R128], [R129], [R130]

as_base_exp()

Return base and exp of self.

Explanation

If base is 1/Integer, then return Integer, -exp. If this extra processing is not needed, the base and exp properties will give the raw arguments

Examples

```
>>> from sympy import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)
```

as_content_primitive(*radical=False*, *clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))
```

```
>>> from sympy import expand_power_base, powsimp, Mul
>>> from sympy.abc import x, y
```

```
>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**(y/2))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content_primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
```

(continues on next page)

(continued from previous page)

```
>>> powsimp(Mul(*_))
3**(2*x + 2)
```

```
>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)
```

See docstring of Expr.as_content_primitive for more examples.

integer_nthroot

sympy.core.power.**integer_nthroot**(*y*, *n*)

Return a tuple containing x = floor(y**(1/n)) and a boolean indicating whether the result is exact (that is, whether x**n == y).

Examples

```
>>> from sympy import integer_nthroot
>>> integer_nthroot(16, 2)
(4, True)
>>> integer_nthroot(26, 2)
(5, False)
```

To simply determine if a number is a perfect square, the is_square function should be used:

```
>>> from sympy.ntheory.primetest import is_square
>>> is_square(26)
False
```

See also:

sympy.ntheory.primetest.is_square (page 1514), *integer_log* (page 1008)

integer_log

sympy.core.power.**integer_log**(*y*, *x*)

Returns (e, bool) where e is the largest nonnegative integer such that $|y| \geq |x^e|$ and bool is True if $y = x^e$.

Examples

```
>>> from sympy import integer_log
>>> integer_log(125, 5)
(3, True)
>>> integer_log(17, 9)
(1, False)
>>> integer_log(4, -2)
(2, True)
>>> integer_log(-125,-5)
(3, True)
```

See also:

integer_nthroot (page 1008), *sympy.ntheory.primetest.is_square* (page 1514), *sympy.ntheory.factor_.multiplicity* (page 1488), *sympy.ntheory.factor_.perfect_power* (page 1488)

mul

Mul

class sympy.core.mul.**Mul**(**args, evaluate=None, _sympify=True*)

Expression representing multiplication operation for algebraic field.

Deprecated since version 1.7: Using arguments that aren't subclasses of *Expr* (page 947) in core operators (*Mul* (page 1009), *Add* (page 1013), and *Pow* (page 1005)) is deprecated. See *Core operators no longer accept non-Expr args* (page 174) for details.

Every argument of `Mul()` must be `Expr`. Infix operator * on most scalar objects in SymPy calls this class.

Another use of `Mul()` is to represent the structure of abstract multiplication so that its arguments can be substituted to return different class. Refer to examples section for this.

`Mul()` evaluates the argument unless `evaluate=False` is passed. The evaluation logic includes:

1. **Flattening**
   `Mul(x, Mul(y, z))` -> `Mul(x, y, z)`
2. **Identity removing**
   `Mul(x, 1, y)` -> `Mul(x, y)`
3. **Exponent collecting by `.as_base_exp()`**
   `Mul(x, x**2)` -> `Pow(x, 3)`
4. **Term sorting**
   `Mul(y, x, 2)` -> `Mul(2, x, y)`

Since multiplication can be vector space operation, arguments may have the different *sympy.core.kind.Kind()* (page 1073). Kind of the resulting object is automatically inferred.

Examples

```
>>> from sympy import Mul
>>> from sympy.abc import x, y
>>> Mul(x, 1)
x
>>> Mul(x, x)
x**2
```

If `evaluate=False` is passed, result is not evaluated.

```
>>> Mul(1, 2, evaluate=False)
1*2
>>> Mul(x, x, evaluate=False)
x*x
```

`Mul()` also represents the general structure of multiplication operation.

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 2,2)
>>> expr = Mul(x,y).subs({y:A})
>>> expr
x*A
>>> type(expr)
<class 'sympy.matrices.expressions.matmul.MatMul'>
```

See also:

MatMul (page 1373)

as_coeff_Mul(*rational=False*)
Efficiently extract the coefficient of a product.

as_content_primitive(*radical=False, clear=True*)
Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(1 - sqrt(2)))
```

See docstring of Expr.as_content_primitive for more examples.

as_ordered_factors(*order=None*)
Transform an expression into an ordered list of factors.

Examples

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

```
>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as_coef_mul() which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use self.as_coeff_add()[0]

Examples

```
>>> from sympy.abc import x, y
>>> (3*x*y).as_two_terms()
(3, x*y)
```


classmethod flatten(*seq*)

Return commutative, noncommutative and order arguments by combining related terms.

Notes

- In an expression like `a*b*c`, Python process this through SymPy as `Mul(Mul(a, b), c)`. This can have undesirable consequences.
  - Sometimes terms are not combined as one would like: {c.f. https://github.com/sympy/sympy/issues/4596}

```
>>> from sympy import Mul, sqrt
>>> from sympy.abc import x, y, z
>>> 2*(x + 1) # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)*2
2*y*(x + 1)
>>> 2*(x + 1)*y # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y) # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y) # parentheses can control this behavior
2*y*(x + 1)
```

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases. {c.f. https://github.com/sympy/sympy/issues/5728}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a,a,a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)
```

- If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of a, b and c were *Mul* (page 1009) expression, then a*b*c (or building up the product with *=) will process all the arguments of a and b twice: once when a*b is computed and again when c is multiplied.

  Using Mul(a, b, c) will process all arguments once.

- The results of Mul are cached according to arguments, so flatten will only be called once for Mul(a, b, c). If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a Mul, M, that you wished to divide by d[i] and multiply by n[i] and you suspect there are many repeats in n. It would be better to compute M*n[i]/d[i] rather than M/d[i]*n[i] since every time n[i] is a repeat, the product, M*n[i] will be returned without flattening – the cached value will be returned. If you divide by the d[i] first (and those are more unique than the n[i]) then that will create a new Mul, M/d[i] the args of which will be traversed again when it is multiplied by n[i].

  {c.f. https://github.com/sympy/sympy/issues/5706}

  This consideration is moot if the cache is turned off.

Nb

The validity of the above notes depends on the implementation details of Mul and flatten which may change at any time. Therefore, you should only consider them when your code is highly performance sensitive.

Removal of 1 from the sequence is already handled by AssocOp.__new__.

prod

sympy.core.mul.**prod**(*a*, *start=1*)

Return product of elements of a. Start with int 1 so if only
ints are included then an int result is returned.

Examples

```
>>> from sympy import prod, S
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([S(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```
>>> prod([1, 2], 3)
6
```

add

Add

class sympy.core.add.**Add**(**args*, *evaluate=None*, *_sympify=True*)

Expression representing addition operation for algebraic group.

Deprecated since version 1.7: Using arguments that aren't subclasses of *Expr* (page 947) in core operators (*Mul* (page 1009), *Add* (page 1013), and *Pow* (page 1005)) is deprecated. See *Core operators no longer accept non-Expr args* (page 174) for details.

Every argument of `Add()` must be `Expr`. Infix operator + on most scalar objects in SymPy calls this class.

Another use of `Add()` is to represent the structure of abstract addition so that its arguments can be substituted to return different class. Refer to examples section for this.

`Add()` evaluates the argument unless `evaluate=False` is passed. The evaluation logic includes:

1. **Flattening**
   `Add(x, Add(y, z)) -> Add(x, y, z)`
2. **Identity removing**
   `Add(x, 0, y) -> Add(x, y)`
3. **Coefficient collecting by `.as_coeff_Mul()`**
   `Add(x, 2*x) -> Mul(3, x)`
4. **Term sorting**
   `Add(y, x, 2) -> Add(2, x, y)`

If no argument is passed, identity element 0 is returned. If single element is passed, that element is returned.

Note that `Add(*args)` is more efficient than `sum(args)` because it flattens the arguments. `sum(a, b, c, ...)` recursively adds the arguments as `a + (b + (c + ...))`, which has quadratic complexity. On the other hand, `Add(a, b, c, d)` does not assume nested structure, making the complexity linear.

Since addition is group operation, every argument should have the same *`sympy.core.kind.Kind()`* (page 1073).

Examples

```
>>> from sympy import Add, I
>>> from sympy.abc import x, y
>>> Add(x, 1)
x + 1
>>> Add(x, x)
2*x
>>> 2*x**2 + 3*x + I*y + 2*y + 2*x/5 + 1.0*y + 1
2*x**2 + 17*x/5 + 3.0*y + I*y + 1
```

If `evaluate=False` is passed, result is not evaluated.

```
>>> Add(1, 2, evaluate=False)
1 + 2
>>> Add(x, x, evaluate=False)
x + x
```

`Add()` also represents the general structure of addition operation.

```
>>> from sympy import MatrixSymbol
>>> A,B = MatrixSymbol('A', 2,2), MatrixSymbol('B', 2,2)
>>> expr = Add(x,y).subs({x:A, y:B})
>>> expr
A + B
>>> type(expr)
<class 'sympy.matrices.expressions.matadd.MatAdd'>
```

Note that the printers do not display in args order.

```
>>> Add(x, 1)
x + 1
```

(continues on next page)

(continued from previous page)

```
>>> Add(x, 1).args
(1, x)
```

See also:

MatAdd (page 1372)

as_coeff_Add(*rational=False, deps=None*)

Efficiently extract the coefficient of a summation.

as_coeff_add(**deps*)

Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

Examples

```
>>> from sympy.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

as_content_primitive(*radical=False, clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

Examples

```
>>> from sympy import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

See docstring of Expr.as_content_primitive for more examples.

as_numer_denom()

Decomposes an expression to its numerator part and its denominator part.

Examples

```
>>> from sympy.abc import x, y, z
>>> (x*y/z).as_numer_denom()
(x*y, z)
>>> (x*(y + 1)/y**7).as_numer_denom()
(x*(y + 1), y**7)
```

See also:

sympy.core.expr.Expr.as_numer_denom (page 955)

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

returns a tuple representing a complex number

Examples

```
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as_coef_add() which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use self.as_coeff_mul()[0]

```
>>> from sympy.abc import x, y
>>> (3*x - 2*y + 5).as_two_terms()
(5, 3*x - 2*y)
```

classmethod class_key()

Nice order of classes

extract_leading_order(*symbols*, *point=None*)

Returns the leading term and its order.

Examples

```
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), O(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, O(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, O(x)),)
```

classmethod flatten(*seq*)

Takes the sequence "seq" of nested Adds and returns a flatten list.

Returns: (commutative_part, noncommutative_part, order_symbols)

Applies associativity, all terms are commutable with respect to addition.

NB: the removal of 0 is already handled by AssocOp.__new__

See also:

sympy.core.mul.Mul.flatten (page 1011)

primitive()

Return `(R, self/R)` where R` is the Rational GCD of `self`.

`R` is collected only from the leading coefficient of each term.

Examples

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

```
>>> (2*x + 4*y).primitive()
(2, x + 2*y)
```

```
>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)
```

```
>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)
```

No subprocessing of term factors is performed:

```
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```

Recursive processing can be done with the `as_content_primitive()` method:

```
>>> ((2 + 2*x)*x + 2).as_content_primitive()
(2, x*(x + 1) + 1)
```

See also: primitive() function in polytools.py

mod

Mod

class sympy.core.mod.**Mod**(*p*, *q*)

Represents a modulo operation on symbolic expressions.

Parameters

p : Expr

Dividend.

q : Expr

Divisor.

Notes

The convention used is the same as Python's: the remainder always has the same sign as the divisor.

Examples

```
>>> from sympy.abc import x, y
>>> x**2 % y
Mod(x**2, y)
>>> _.subs({x: 5, y: 6})
1
```


relational

Rel

class sympy.core.relational.**Relational**(*lhs*, *rhs*, *rop=None*, ***assumptions*)

Base class for all relation types.

Parameters

rop : str or None

Indicates what subclass to instantiate. Valid values can be found in the keys of Relational.ValidRelationOperator.

Explanation

Subclasses of Relational should generally be instantiated directly, but Relational can be instantiated with a valid `rop` value to dispatch to the appropriate subclass.

Examples

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x + x**2, '==')
Eq(y, x**2 + x)
```

A relation's type can be defined upon creation using `rop`. The relation type of an existing expression can be obtained using its `rel_op` property. Here is a table of all the relation types, along with their `rop` and `rel_op` values:

Relation	rop	rel_op
Equality	== or eq or None	==
Unequality	!= or ne	!=
GreaterThan	>= or ge	>=
LessThan	<= or le	<=
StrictGreaterThan	> or gt	>
StrictLessThan	< or lt	<

For example, setting `rop` to == produces an `Equality` relation, `Eq()`. So does setting `rop` to `eq`, or leaving `rop` unspecified. That is, the first three `Rel()` below all produce the same result. Using a `rop` from a different row in the table produces a different relation type. For example, the fourth `Rel()` below using `lt` for `rop` produces a `StrictLessThan` inequality:

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x + x**2, '==')
    Eq(y, x**2 + x)
>>> Rel(y, x + x**2, 'eq')
    Eq(y, x**2 + x)
>>> Rel(y, x + x**2)
    Eq(y, x**2 + x)
>>> Rel(y, x + x**2, 'lt')
    y < x**2 + x
```

To obtain the relation type of an existing expression, get its `rel_op` property. For example, `rel_op` is == for the `Equality` relation above, and < for the strict less than inequality above:

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> my_equality = Rel(y, x + x**2, '==')
>>> my_equality.rel_op
    '=='
>>> my_inequality = Rel(y, x + x**2, 'lt')
```

(continues on next page)

(continued from previous page)

```
>>> my_inequality.rel_op
'<'
```

property canonical

Return a canonical form of the relational by putting a number on the rhs, canonically removing a sign or else ordering the args canonically. No other simplification is attempted.

Examples

```
>>> from sympy.abc import x, y
>>> x < 2
x < 2
>>> _.reversed.canonical
x < 2
>>> (-y < x).canonical
x > -y
>>> (-y > x).canonical
x < -y
>>> (-y < -x).canonical
x < y
```

The canonicalization is recursively applied:

```
>>> from sympy import Eq
>>> Eq(x < y, y > x).canonical
True
```

equals(*other, failing_expression=False*)

Return True if the sides of the relationship are mathematically identical and the type of relationship is the same. If failing_expression is True, return the expression whose truth value was unknown.

property lhs

The left-hand side of the relation.

property negated

Return the negated relationship.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.negated
Ne(x, 1)
>>> x < 1
x < 1
```

(continues on next page)

(continued from previous page)

```
>>> _.negated
x >= 1
```

Notes

This works more or less identical to ~/Not. The difference is that negated returns the relationship even if evaluate=False. Hence, this is useful in code when checking for e.g. negated relations to existing ones as it will not be affected by the *evaluate* flag.

property reversed

Return the relationship with sides reversed.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversed
Eq(1, x)
>>> x < 1
x < 1
>>> _.reversed
1 > x
```


property reversedsign

Return the relationship with signs reversed.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversedsign
Eq(-x, -1)
>>> x < 1
x < 1
>>> _.reversedsign
-x > -1
```

property rhs

The right-hand side of the relation.

property strict

return the strict version of the inequality or self

Examples

```
>>> from sympy.abc import x
>>> (x <= 1).strict
x < 1
>>> _.strict
x < 1
```

property weak
return the non-strict version of the inequality or self

Examples

```
>>> from sympy.abc import x
>>> (x < 1).weak
x <= 1
>>> _.weak
x <= 1
```

sympy.core.relational.**Rel**
alias of *Relational* (page 1018)

Eq

sympy.core.relational.**Eq**
alias of *Equality* (page 1023)

Ne

sympy.core.relational.**Ne**
alias of *Unequality* (page 1031)

Lt

sympy.core.relational.**Lt**
alias of *StrictLessThan* (page 1035)

Le

sympy.core.relational.**Le**
alias of *LessThan* (page 1028)

Gt

sympy.core.relational.**Gt**

alias of *StrictGreaterThan* (page 1032)

Ge

sympy.core.relational.**Ge**

alias of *GreaterThan* (page 1024)

Equality

class sympy.core.relational.**Equality**(*lhs, rhs=None, **options*)

An equal relation between two objects.

Explanation

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the simplify function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument evaluate=False can be used to prevent any evaluation.

Examples

```
>>> from sympy import Eq, simplify, exp, cos
>>> from sympy.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
False
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
False
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(exp(x), sinh(x) + cosh(x))
>>> simplify(_)
True
```

Notes

Python treats 1 and True (and 0 and False) as being equal; SymPy does not. And integer will always compare as unequal to a Boolean:

```
>>> Eq(True, 1), True == 1
(False, True)
```

This class is not the same as the == operator. The == operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `_eval_Eq` method, it can be used in place of the default algorithm. If `lhs._eval_Eq(rhs)` or `rhs._eval_Eq(lhs)` returns anything other than None, that return value will be substituted for the Equality. If None is returned by `_eval_Eq`, an Equality object will be created as usual.

Since this object is already an expression, it does not respond to the method `as_expr` if one tries to create $x - y$ from `Eq(x, y)`. This can be done with the `rewrite(Add)` method.

Deprecated since version 1.5: `Eq(expr)` with a single argument is a shorthand for `Eq(expr, 0)`, but this behavior is deprecated and will be removed in a future version of SymPy.

See also:

sympy.logic.boolalg.Equivalent **(page 1171)**
for representing equality between two boolean expressions

as_poly(**gens, **kwargs*)
Returns lhs-rhs as a Poly

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x**2, 1).as_poly(x)
Poly(x**2 - 1, x, domain='ZZ')
```

integrate(**args, **kwargs*)
See the integrate function in sympy.integrals

GreaterThan

class sympy.core.relational.**GreaterThan**(*lhs, rhs, **options*)
Class representations of inequalities.

Explanation

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$lhs \geq rhs$

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	>=
LessThan	<=
StrictGreaterThan	>
StrictLessThan	<

All classes take two arguments, lhs and rhs.

Signature Example	Math Equivalent
GreaterThan(lhs, rhs)	$lhs \geq rhs$
LessThan(lhs, rhs)	$lhs \leq rhs$
StrictGreaterThan(lhs, rhs)	$lhs > rhs$
StrictLessThan(lhs, rhs)	$lhs < rhs$

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the "less than side" and "greater than side" of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of "gotchas" to be aware of when using Python's operators.

The first is that what your write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that x is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, x > 1 and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) "sympify" the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving == or !=. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R131], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the '&' operator, it cannot be done with the 'and' operarator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

LessThan

class sympy.core.relational.**LessThan**(*lhs*, *rhs*, ***options*)

Class representations of inequalities.

Explanation

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs $\geq$ rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	>=
LessThan	<=
StrictGreaterThan	>
StrictLessThan	<

All classes take two arguments, lhs and rhs.

Signature Example	Math Equivalent
GreaterThan(lhs, rhs)	lhs $\geq$ rhs
LessThan(lhs, rhs)	lhs $\leq$ rhs
StrictGreaterThan(lhs, rhs)	lhs $>$ rhs
StrictLessThan(lhs, rhs)	lhs $<$ rhs

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the "less than side" and "greater than side" of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of "gotchas" to be aware of when using Python's operators.

The first is that what your write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that x is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) "sympify" the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```


The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving == or !=. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R132], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the '&' operator, it cannot be done with the 'and' operarator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```


Unequality

class sympy.core.relational.**Unequality**(*lhs*, *rhs*, ***options*)

An unequal relation between two objects.

Explanation

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

Examples

```
>>> from sympy import Ne
>>> from sympy.abc import x, y
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```

Notes

This class is not the same as the != operator. The != operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of Equality. As such, it uses the same algorithms, including any available $_eval_Eq$ methods.

See also:

Equality (page 1023)

StrictGreaterThan

class sympy.core.relational.**StrictGreaterThan**(*lhs*, *rhs*, ***options*)

Class representations of inequalities.

Explanation

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs $\geq$ rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	>=
LessThan	<=
StrictGreaterThan	>
StrictLessThan	<

All classes take two arguments, lhs and rhs.

Signature Example	Math Equivalent
GreaterThan(lhs, rhs)	lhs $\geq$ rhs
LessThan(lhs, rhs)	lhs $\leq$ rhs
StrictGreaterThan(lhs, rhs)	lhs $>$ rhs
StrictLessThan(lhs, rhs)	lhs $<$ rhs

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the "less than side" and "greater than side" of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```


However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of "gotchas" to be aware of when using Python's operators.

The first is that what your write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that x is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) "sympify" the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving == or !=. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R133], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the '&' operator, it cannot be done with the 'and' operarator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```


StrictLessThan

class sympy.core.relational.**StrictLessThan**(*lhs*, *rhs*, ***options*)

Class representations of inequalities.

Explanation

The *Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs $\geq$ rhs

In total, there are four *Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	>=
LessThan	<=
StrictGreaterThan	>
StrictLessThan	<

All classes take two arguments, lhs and rhs.

Signature Example	Math Equivalent
GreaterThan(lhs, rhs)	lhs $\geq$ rhs
LessThan(lhs, rhs)	lhs $\leq$ rhs
StrictGreaterThan(lhs, rhs)	lhs $>$ rhs
StrictLessThan(lhs, rhs)	lhs $<$ rhs

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the "less than side" and "greater than side" of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```


Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> for f in [Ge, Gt, Le, Lt]:  # convenience wrappers
...     print(f(x, 2))
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (>=, >, <=, <) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more "mathematical looking" statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for 'gotcha', below).

```
>>> x >= 2
x >= 2
>>> _ == Ge(x, 2)
True
```

However, it is also perfectly valid to instantiate a *Than class less succinctly and less conveniently: