### Examples

```
>>> from sympy import Predicate, Q
>>> Q.P = Predicate('P')
>>> Q.P.func
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P.name
Str('P')
```

sympy.assumptions.assume.**assuming**(**assumptions*)

Context manager for assumptions.

### Examples

```
>>> from sympy import assuming, Q, ask
>>> from sympy.abc import x, y
>>> print(ask(Q.integer(x + y)))
None
>>> with assuming(Q.integer(x), Q.integer(y)):
...     print(ask(Q.integer(x + y)))
True
```

## Refine

sympy.assumptions.refine.**refine**(*expr*, *assumptions=True*)

Simplify an expression using assumptions.

### Explanation

Unlike *simplify()* (page 661) which performs structural simplification without any assumption, this function transforms the expression into the form which is only valid under certain assumptions. Note that `simplify()` is generally not done in refining process.

Refining boolean expression involves reducing it to `S.true` or `S.false`. Unlike *ask()* (page 191), the expression will not be reduced if the truth value cannot be determined.

### Examples

```
>>> from sympy import refine, sqrt, Q
>>> from sympy.abc import x
>>> refine(sqrt(x**2), Q.real(x))
Abs(x)
>>> refine(sqrt(x**2), Q.positive(x))
x
```

```
>>> refine(Q.real(x), Q.positive(x))
True
>>> refine(Q.positive(x), Q.real(x))
Q.positive(x)
```

**See also:**

*sympy.simplify.simplify.simplify* **(page 661)**
Structural simplification without assumptions.

*sympy.assumptions.ask.ask* **(page 191)**
Query for boolean expressions using assumptions.

sympy.assumptions.refine.**refine_Pow**(*expr*, *assumptions*)
Handler for instances of Pow.

### Examples

```
>>> from sympy import Q
>>> from sympy.assumptions.refine import refine_Pow
>>> from sympy.abc import x,y,z
>>> refine_Pow((-1)**x, Q.real(x))
>>> refine_Pow((-1)**x, Q.even(x))
1
>>> refine_Pow((-1)**x, Q.odd(x))
-1
```

For powers of -1, even parts of the exponent can be simplified:

```
>>> refine_Pow((-1)**(x+y), Q.even(x))
(-1)**y
>>> refine_Pow((-1)**(x+y+z), Q.odd(x) & Q.odd(z))
(-1)**y
>>> refine_Pow((-1)**(x+y+2), Q.odd(x))
(-1)**(y + 1)
>>> refine_Pow((-1)**(x+3), True)
(-1)**(x + 1)
```

sympy.assumptions.refine.**refine_abs**(*expr*, *assumptions*)
Handler for the absolute value.

### Examples

```
>>> from sympy import Q, Abs
>>> from sympy.assumptions.refine import refine_abs
>>> from sympy.abc import x
>>> refine_abs(Abs(x), Q.real(x))
>>> refine_abs(Abs(x), Q.positive(x))
x
>>> refine_abs(Abs(x), Q.negative(x))
-x
```

sympy.assumptions.refine.**refine_arg**(*expr*, *assumptions*)

Handler for complex argument

### Explanation

```
>>> from sympy.assumptions.refine import refine_arg
>>> from sympy import Q, arg
>>> from sympy.abc import x
>>> refine_arg(arg(x), Q.positive(x))
0
>>> refine_arg(arg(x), Q.negative(x))
pi
```

sympy.assumptions.refine.**refine_atan2**(*expr*, *assumptions*)

Handler for the atan2 function.

### Examples

```
>>> from sympy import Q, atan2
>>> from sympy.assumptions.refine import refine_atan2
>>> from sympy.abc import x, y
>>> refine_atan2(atan2(y,x), Q.real(y) & Q.positive(x))
atan(y/x)
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.negative(x))
atan(y/x) - pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.negative(x))
atan(y/x) + pi
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.negative(x))
pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.zero(x))
pi/2
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.zero(x))
-pi/2
>>> refine_atan2(atan2(y,x), Q.zero(y) & Q.zero(x))
nan
```

sympy.assumptions.refine.**refine_im**(*expr*, *assumptions*)

Handler for imaginary part.

### Explanation

```
>>> from sympy.assumptions.refine import refine_im
>>> from sympy import Q, im
>>> from sympy.abc import x
>>> refine_im(im(x), Q.real(x))
0
>>> refine_im(im(x), Q.imaginary(x))
-I*x
```

sympy.assumptions.refine.**refine_matrixelement**(*expr, assumptions*)

Handler for symmetric part.

### Examples

```
>>> from sympy.assumptions.refine import refine_matrixelement
>>> from sympy import MatrixSymbol, Q
>>> X = MatrixSymbol('X', 3, 3)
>>> refine_matrixelement(X[0, 1], Q.symmetric(X))
X[0, 1]
>>> refine_matrixelement(X[1, 0], Q.symmetric(X))
X[0, 1]
```

sympy.assumptions.refine.**refine_re**(*expr, assumptions*)

Handler for real part.

### Examples

```
>>> from sympy.assumptions.refine import refine_re
>>> from sympy import Q, re
>>> from sympy.abc import x
>>> refine_re(re(x), Q.real(x))
x
>>> refine_re(re(x), Q.imaginary(x))
0
```

sympy.assumptions.refine.**refine_sign**(*expr, assumptions*)

Handler for sign.

### Examples

```
>>> from sympy.assumptions.refine import refine_sign
>>> from sympy import Symbol, Q, sign, im
>>> x = Symbol('x', real = True)
>>> expr = sign(x)
>>> refine_sign(expr, Q.positive(x) & Q.nonzero(x))
1
>>> refine_sign(expr, Q.negative(x) & Q.nonzero(x))
-1
>>> refine_sign(expr, Q.zero(x))
0
>>> y = Symbol('y', imaginary = True)
>>> expr = sign(y)
>>> refine_sign(expr, Q.positive(im(y)))
I
>>> refine_sign(expr, Q.negative(im(y)))
-I
```

### Predicates

### Common

### Tautological

**class** sympy.assumptions.predicates.common.**IsTruePredicate**(**args*, ***kwargs*)

Generic predicate.

#### Explanation

`ask(Q.is_true(x))` is true iff `x` is true. This only makes sense if `x` is a boolean object.

#### Examples

```
>>> from sympy import ask, Q
>>> from sympy.abc import x, y
>>> ask(Q.is_true(True))
True
```

Wrapping another applied predicate just returns the applied predicate.

```
>>> Q.is_true(Q.even(x))
Q.even(x)
```

Wrapping binary relation classes in SymPy core returns applied binary relational predicates.

```
>>> from sympy import Eq, Gt
>>> Q.is_true(Eq(x, y))
Q.eq(x, y)
>>> Q.is_true(Gt(x, y))
Q.gt(x, y)
```

#### Notes

This class is designed to wrap the boolean objects so that they can behave as if they are applied predicates. Consequently, wrapping another applied predicate is unnecessary and thus it just returns the argument. Also, binary relation classes in SymPy core have binary predicates to represent themselves and thus wrapping them with `Q.is_true` converts them to these applied predicates.

#### Handler

Multiply dispatched method: IsTrueHandler

Wrapper allowing to query the truth value of a boolean expression.

**handler = <dispatched IsTrueHandler>**

### Commutative

**class** sympy.assumptions.predicates.common.**CommutativePredicate**(**args*, ***kwargs*)

Commutative predicate.

#### Explanation

`ask(Q.commutative(x))` is true iff `x` commutes with any other object with respect to multiplication operation.

#### Handler

Multiply dispatched method: CommutativeHandler

Handler for key 'commutative'.

**handler = <dispatched CommutativeHandler>**

### Calculus

### Finite

**class** sympy.assumptions.predicates.calculus.**FinitePredicate**(**args*, ***kwargs*)

Finite number predicate.

#### Explanation

`Q.finite(x)` is true if `x` is a number but neither an infinity nor a `NaN`. In other words, `ask(Q.finite(x))` is true for all numerical `x` having a bounded absolute value.

#### Examples

```
>>> from sympy import Q, ask, S, oo, I, zoo
>>> from sympy.abc import x
>>> ask(Q.finite(oo))
False
>>> ask(Q.finite(-oo))
False
>>> ask(Q.finite(zoo))
```

(continues on next page)

(continued from previous page)

```
False
>>> ask(Q.finite(1))
True
>>> ask(Q.finite(2 + 3*I))
True
>>> ask(Q.finite(x), Q.positive(x))
True
>>> print(ask(Q.finite(S.NaN)))
None
```

#### Handler

Multiply dispatched method: FiniteHandler

Handler for Q.finite. Test that an expression is bounded respect to all its variables.

#### References

[R7]

**handler = <dispatched FiniteHandler>**

### Infinite

**class** sympy.assumptions.predicates.calculus.**InfinitePredicate**(**args*, ***kwargs*)

Infinite number predicate.

`Q.infinite(x)` is true iff the absolute value of `x` is infinity.

#### Handler

Multiply dispatched method: InfiniteHandler

Handler for Q.infinite key.

**handler = <dispatched InfiniteHandler>**

### Matrix

### Symmetric

**class** sympy.assumptions.predicates.matrices.**SymmetricPredicate**(**args*, ***kwargs*)

Symmetric matrix predicate.

### Explanation

`Q.symmetric(x)` is true iff `x` is a square matrix and is equal to its transpose. Every square diagonal matrix is a symmetric matrix.

### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.symmetric(X*Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(X + Z), Q.symmetric(X) & Q.symmetric(Z))
True
>>> ask(Q.symmetric(Y))
False
```

### Handler

Multiply dispatched method: SymmetricHandler

Handler for Q.symmetric.

### References

[R8]

**handler = <dispatched SymmetricHandler>**

## Invertible

**class** sympy.assumptions.predicates.matrices.**InvertiblePredicate**(**args*, ***kwargs*)

Invertible matrix predicate.

### Explanation

`Q.invertible(x)` is true iff `x` is an invertible matrix. A square matrix is called invertible only if its determinant is 0.

### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.invertible(X*Y), Q.invertible(X))
False
>>> ask(Q.invertible(X*Z), Q.invertible(X) & Q.invertible(Z))
True
>>> ask(Q.invertible(X), Q.fullrank(X) & Q.square(X))
True
```

### Handler

Multiply dispatched method: InvertibleHandler

Handler for Q.invertible.

### References

[R9]

**handler = <dispatched InvertibleHandler>**

## Orthogonal

**class** sympy.assumptions.predicates.matrices.**OrthogonalPredicate**(**args*, ***kwargs*)

Orthogonal matrix predicate.

### Explanation

`Q.orthogonal(x)` is true iff `x` is an orthogonal matrix. A square matrix `M` is an orthogonal matrix if it satisfies `M^TM = MM^T = I` where `M^T` is the transpose matrix of `M` and `I` is an identity matrix. Note that an orthogonal matrix is necessarily invertible.

### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.orthogonal(Y))
False
>>> ask(Q.orthogonal(X*Z*X), Q.orthogonal(X) & Q.orthogonal(Z))
True
```

(continues on next page)

(continued from previous page)

```
>>> ask(Q.orthogonal(Identity(3)))
True
>>> ask(Q.invertible(X), Q.orthogonal(X))
True
```

#### Handler

Multiply dispatched method: OrthogonalHandler

Handler for key 'orthogonal'.

#### References

[R10]

**handler = <dispatched OrthogonalHandler>**

### Unitary

**class** sympy.assumptions.predicates.matrices.**UnitaryPredicate**(**args*, ***kwargs*)

Unitary matrix predicate.

#### Explanation

`Q.unitary(x)` is true iff `x` is a unitary matrix. Unitary matrix is an analogue to orthogonal matrix. A square matrix `M` with complex elements is unitary if :math:`M^TM = MM^T= I` where :math:`M^T` is the conjugate transpose matrix of `M`.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.unitary(Y))
False
>>> ask(Q.unitary(X*Z*X), Q.unitary(X) & Q.unitary(Z))
True
>>> ask(Q.unitary(Identity(3)))
True
```

### Handler

Multiply dispatched method: UnitaryHandler

Handler for key ‘unitary’.

### References

[R11]

**handler = <dispatched UnitaryHandler>**

## Positive Definite

**class** sympy.assumptions.predicates.matrices.**PositiveDefinitePredicate**(**args*, ***kwargs*)

Positive definite matrix predicate.

### Explanation

If $M$ is a $n \times n$ symmetric real matrix, it is said to be positive definite if $Z^T M Z$ is positive for every non-zero column vector $Z$ of $n$ real numbers.

### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('Y', 2, 3)
>>> Z = MatrixSymbol('Z', 2, 2)
>>> ask(Q.positive_definite(Y))
False
>>> ask(Q.positive_definite(Identity(3)))
True
>>> ask(Q.positive_definite(X + Z), Q.positive_definite(X) &
...      Q.positive_definite(Z))
True
```

### Handler

Multiply dispatched method: PositiveDefiniteHandler

Handler for key ‘positive_definite’.

#### References

[R12]

**handler = <dispatched PositiveDefiniteHandler>**

### Upper triangular

**class** sympy.assumptions.predicates.matrices.**UpperTriangularPredicate**(**args*, ***kwargs*)

Upper triangular matrix predicate.

#### Explanation

A matrix $M$ is called upper triangular matrix if $M_{ij}=0$ for $i<j$.

#### Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.upper_triangular(Identity(3)))
True
>>> ask(Q.upper_triangular(ZeroMatrix(3, 3)))
True
```

#### Handler

Multiply dispatched method: UpperTriangularHandler

Handler for key 'upper_triangular'.

#### References

[R13]

**handler = <dispatched UpperTriangularHandler>**

### Lower triangular

**class** sympy.assumptions.predicates.matrices.**LowerTriangularPredicate**(**args*, ***kwargs*)

Lower triangular matrix predicate.

#### Explanation

A matrix $M$ is called lower triangular matrix if $M_{ij} = 0$ for $i > j$.

#### Examples

```
>>> from sympy import Q, ask, ZeroMatrix, Identity
>>> ask(Q.lower_triangular(Identity(3)))
True
>>> ask(Q.lower_triangular(ZeroMatrix(3, 3)))
True
```

#### Handler

Multiply dispatched method: LowerTriangularHandler

Handler for key 'lower_triangular'.

#### References

[R14]

**handler = <dispatched LowerTriangularHandler>**

### Diagonal

**class** sympy.assumptions.predicates.matrices.**DiagonalPredicate**(**args*, ***kwargs*)

Diagonal matrix predicate.

#### Explanation

`Q.diagonal(x)` is true iff `x` is a diagonal matrix. A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.diagonal(ZeroMatrix(3, 3)))
True
>>> ask(Q.diagonal(X), Q.lower_triangular(X) &
...      Q.upper_triangular(X))
True
```

#### Handler

Multiply dispatched method: DiagonalHandler

Handler for key 'diagonal'.

#### References

[R15]

**handler = <dispatched DiagonalHandler>**

### Full rank

**class** sympy.assumptions.predicates.matrices.**FullRankPredicate**(**args*, ***kwargs*)

Fullrank matrix predicate.

#### Explanation

`Q.fullrank(x)` is true iff `x` is a full rank matrix. A matrix is full rank if all rows and columns of the matrix are linearly independent. A square matrix is full rank iff its determinant is nonzero.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> ask(Q.fullrank(X.T), Q.fullrank(X))
True
>>> ask(Q.fullrank(ZeroMatrix(3, 3)))
False
>>> ask(Q.fullrank(Identity(3)))
True
```

#### Handler

Multiply dispatched method: FullRankHandler

Handler for key 'fullrank'.

**handler = <dispatched FullRankHandler>**

### Square

**class** sympy.assumptions.predicates.matrices.**SquarePredicate**(**args*, ***kwargs*)

Square matrix predicate.

#### Explanation

`Q.square(x)` is true iff `x` is a square matrix. A square matrix is a matrix with the same number of rows and columns.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol, ZeroMatrix, Identity
>>> X = MatrixSymbol('X', 2, 2)
>>> Y = MatrixSymbol('X', 2, 3)
>>> ask(Q.square(X))
True
>>> ask(Q.square(Y))
False
>>> ask(Q.square(ZeroMatrix(3, 3)))
True
>>> ask(Q.square(Identity(3)))
True
```

#### Handler

Multiply dispatched method: SquareHandler

Handler for Q.square.

#### References

[R16]

**handler = <dispatched SquareHandler>**

### Integer elements

**class** sympy.assumptions.predicates.matrices.**IntegerElementsPredicate**(**args*, ***kwargs*)

Integer elements matrix predicate.

#### Explanation

`Q.integer_elements(x)` is true iff all the elements of `x` are integers.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.integer(X[1, 2]), Q.integer_elements(X))
True
```

#### Handler

Multiply dispatched method: IntegerElementsHandler

Handler for key 'integer_elements'.

**handler = <dispatched IntegerElementsHandler>**

### Real elements

**class** sympy.assumptions.predicates.matrices.**RealElementsPredicate**(**args*, ***kwargs*)

Real elements matrix predicate.

#### Explanation

`Q.real_elements(x)` is true iff all the elements of `x` are real numbers.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.real(X[1, 2]), Q.real_elements(X))
True
```

#### Handler

Multiply dispatched method: RealElementsHandler

Handler for key 'real_elements'.

**handler = <dispatched RealElementsHandler>**

### Complex elements

**class** sympy.assumptions.predicates.matrices.**ComplexElementsPredicate**(**args*, ***kwargs*)

Complex elements matrix predicate.

#### Explanation

`Q.complex_elements(x)` is true iff all the elements of `x` are complex numbers.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.complex(X[1, 2]), Q.complex_elements(X))
True
>>> ask(Q.complex_elements(X), Q.integer_elements(X))
True
```

#### Handler

Multiply dispatched method: ComplexElementsHandler

Handler for key ‘complex_elements’.

**handler = <dispatched ComplexElementsHandler>**

### Singular

**class** sympy.assumptions.predicates.matrices.**SingularPredicate**(**args*, ***kwargs*)

Singular matrix predicate.

A matrix is singular iff the value of its determinant is 0.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.singular(X), Q.invertible(X))
False
>>> ask(Q.singular(X), ~Q.invertible(X))
True
```

#### Handler

Multiply dispatched method: SingularHandler

Predicate fore key ‘singular’.

#### References

[R17]

**handler = <dispatched SingularHandler>**

### Normal

**class** sympy.assumptions.predicates.matrices.**NormalPredicate**(**args*, ***kwargs*)

Normal matrix predicate.

A matrix is normal if it commutes with its conjugate transpose.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.normal(X), Q.unitary(X))
True
```

#### Handler

Multiply dispatched method: NormalHandler

Predicate fore key ‘normal’.

#### References

[R18]

**handler = <dispatched NormalHandler>**

### Triangular

**class** sympy.assumptions.predicates.matrices.**TriangularPredicate**(**args*, ***kwargs*)

Triangular matrix predicate.

#### Explanation

`Q.triangular(X)` is true if `X` is one that is either lower triangular or upper triangular.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.upper_triangular(X))
True
>>> ask(Q.triangular(X), Q.lower_triangular(X))
True
```

#### Handler

Multiply dispatched method: TriangularHandler

Predicate fore key 'triangular'.

#### References

[R19]

**handler = <dispatched TriangularHandler>**

### Unit triangular

**class** sympy.assumptions.predicates.matrices.**UnitTriangularPredicate**(**args*, ***kwargs*)

Unit triangular matrix predicate.

#### Explanation

A unit triangular matrix is a triangular matrix with 1s on the diagonal.

#### Examples

```
>>> from sympy import Q, ask, MatrixSymbol
>>> X = MatrixSymbol('X', 4, 4)
>>> ask(Q.triangular(X), Q.unit_triangular(X))
True
```

#### Handler

Multiply dispatched method: UnitTriangularHandler

Predicate fore key ‘unit_triangular’.

**handler = <dispatched UnitTriangularHandler>**

### Number Theory

### Even

**class** sympy.assumptions.predicates.ntheory.**EvenPredicate**(**args*, ***kwargs*)

Even number predicate.

#### Explanation

`ask(Q.even(x))` is true iff `x` belongs to the set of even integers.

#### Examples

```
>>> from sympy import Q, ask, pi
>>> ask(Q.even(0))
True
>>> ask(Q.even(2))
True
>>> ask(Q.even(3))
False
>>> ask(Q.even(pi))
False
```

#### Handler

Multiply dispatched method: EvenHandler

Handler for key ‘even’.

**handler = <dispatched EvenHandler>**

### Odd

**class** sympy.assumptions.predicates.ntheory.**OddPredicate**(**args*, ***kwargs*)

Odd number predicate.

#### Explanation

`ask(Q.odd(x))` is true iff `x` belongs to the set of odd numbers.

#### Examples

```
>>> from sympy import Q, ask, pi
>>> ask(Q.odd(0))
False
>>> ask(Q.odd(2))
False
>>> ask(Q.odd(3))
True
>>> ask(Q.odd(pi))
False
```

#### Handler

Multiply dispatched method: OddHandler

Handler for key 'odd'. Test that an expression represents an odd number.

**handler = <dispatched OddHandler>**

### Prime

**class** sympy.assumptions.predicates.ntheory.**PrimePredicate**(**args*, ***kwargs*)

Prime number predicate.

#### Explanation

`ask(Q.prime(x))` is true iff `x` is a natural number greater than 1 that has no positive divisors other than 1 and the number itself.

#### Examples

```
>>> from sympy import Q, ask
>>> ask(Q.prime(0))
False
>>> ask(Q.prime(1))
False
>>> ask(Q.prime(2))
True
>>> ask(Q.prime(20))
False
>>> ask(Q.prime(-3))
False
```

### Handler

Multiply dispatched method: PrimeHandler

Handler for key 'prime'. Test that an expression represents a prime number. When the expression is an exact number, the result (when True) is subject to the limitations of isprime() which is used to return the result.

**handler = <dispatched PrimeHandler>**

## Composite

**class** sympy.assumptions.predicates.ntheory.**CompositePredicate**(**args*, ***kwargs*)

Composite number predicate.

### Explanation

`ask(Q.composite(x))` is true iff `x` is a positive integer and has at least one positive divisor other than 1 and the number itself.

### Examples

```
>>> from sympy import Q, ask
>>> ask(Q.composite(0))
False
>>> ask(Q.composite(1))
False
>>> ask(Q.composite(2))
False
>>> ask(Q.composite(20))
True
```

### Handler

Multiply dispatched method: CompositeHandler

Handler for key 'composite'.

**handler = <dispatched CompositeHandler>**

### Order

### Positive

**class** sympy.assumptions.predicates.order.**PositivePredicate**(**args*, ***kwargs*)

Positive real number predicate.

#### Explanation

`Q.positive(x)` is true iff `x` is real and $x > 0$, that is if `x` is in the interval $(0, \infty)$. In particular, infinity is not positive.

A few important facts about positive numbers:

- **Note that `Q.nonpositive` and `~Q.positive` are *not* the same thing.** `~Q.positive(x)` simply means that `x` is not positive, whereas `Q.nonpositive(x)` means that `x` is real and not positive, i.e., `Q.nonpositive(x)` is logically equivalent to $Q.negative(x) | Q.zero(x)$. So for example, `~Q.positive(I)` is true, whereas `Q.nonpositive(I)` is false.
- **See the documentation of `Q.real` for more information about** related facts.

#### Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.positive(x), Q.real(x) & ~Q.negative(x) & ~Q.zero(x))
True
>>> ask(Q.positive(1))
True
>>> ask(Q.nonpositive(I))
False
>>> ask(~Q.positive(I))
True
```

#### Handler

Multiply dispatched method: PositiveHandler

Handler for key 'positive'. Test that an expression is strictly greater than zero.

**handler = <dispatched PositiveHandler>**

### Negative

**class** sympy.assumptions.predicates.order.**NegativePredicate**(**args*, ***kwargs*)

Negative number predicate.

#### Explanation

`Q.negative(x)` is true iff `x` is a real number and $x < 0$, that is, it is in the interval $(-\infty, 0)$. Note in particular that negative infinity is not negative.

A few important facts about negative numbers:

- **Note that `Q.nonnegative` and `~Q.negative` are *not* the same thing.** `~Q.negative(x)` simply means that `x` is not negative, whereas `Q.nonnegative(x)` means that `x` is real and not negative, i.e., `Q.nonnegative(x)` is logically equivalent to `Q.zero(x) | Q.positive(x)`. So for example, `~Q.negative(I)` is true, whereas `Q.nonnegative(I)` is false.
- **See the documentation of `Q.real` for more information about** related facts.

#### Examples

```
>>> from sympy import Q, ask, symbols, I
>>> x = symbols('x')
>>> ask(Q.negative(x), Q.real(x) & ~Q.positive(x) & ~Q.zero(x))
True
>>> ask(Q.negative(-1))
True
>>> ask(Q.nonnegative(I))
False
>>> ask(~Q.negative(I))
True
```

#### Handler

Multiply dispatched method: NegativeHandler

Handler for Q.negative. Test that an expression is strictly less than zero.

**handler = <dispatched NegativeHandler>**

### Zero

**class** sympy.assumptions.predicates.order.**ZeroPredicate**(**args*, ***kwargs*)

Zero number predicate.

#### Explanation

`ask(Q.zero(x))` is true iff the value of `x` is zero.

#### Examples

```
>>> from sympy import ask, Q, oo, symbols
>>> x, y = symbols('x, y')
>>> ask(Q.zero(0))
True
>>> ask(Q.zero(1/oo))
True
>>> print(ask(Q.zero(0*oo)))
None
>>> ask(Q.zero(1))
False
>>> ask(Q.zero(x*y), Q.zero(x) | Q.zero(y))
True
```

#### Handler

Multiply dispatched method: ZeroHandler

Handler for key ‘zero’.

**handler = <dispatched ZeroHandler>**

### Nonzero

**class** sympy.assumptions.predicates.order.**NonZeroPredicate**(**args*, ***kwargs*)

Nonzero real number predicate.

#### Explanation

`ask(Q.nonzero(x))` is true iff `x` is real and `x` is not zero. Note in particular that `Q.nonzero(x)` is false if `x` is not real. Use `~Q.zero(x)` if you want the negation of being zero without any real assumptions.

A few important facts about nonzero numbers:

- `Q.nonzero` is logically equivalent to `Q.positive | Q.negative`.
- **See the documentation of `Q.real` for more information about** related facts.

### Examples

```
>>> from sympy import Q, ask, symbols, I, oo
>>> x = symbols('x')
>>> print(ask(Q.nonzero(x), ~Q.zero(x)))
None
>>> ask(Q.nonzero(x), Q.positive(x))
True
>>> ask(Q.nonzero(x), Q.zero(x))
False
>>> ask(Q.nonzero(0))
False
>>> ask(Q.nonzero(I))
False
>>> ask(~Q.zero(I))
True
>>> ask(Q.nonzero(oo))
False
```

### Handler

Multiply dispatched method: NonZeroHandler

Handler for key ‘zero’. Test that an expression is not identically zero.

**handler = <dispatched NonZeroHandler>**

## Nonpositive

**class** sympy.assumptions.predicates.order.**NonPositivePredicate**(**args*, ***kwargs*)

Nonpositive real number predicate.

### Explanation

`ask(Q.nonpositive(x))` is true iff `x` belongs to the set of negative numbers including zero.

- **Note that `Q.nonpositive` and `~Q.positive` are *not* the same thing.** `~Q.positive(x)` simply means that `x` is not positive, whereas `Q.nonpositive(x)` means that `x` is real and not positive, i.e., `Q.nonpositive(x)` is logically equivalent to $Q.negative(x) | Q.zero(x)$‘. So for example, `~Q.positive(I)` is true, whereas `Q.nonpositive(I)` is false.

### Examples

```
>>> from sympy import Q, ask, I
```

```
>>> ask(Q.nonpositive(-1))
True
>>> ask(Q.nonpositive(0))
True
>>> ask(Q.nonpositive(1))
False
>>> ask(Q.nonpositive(I))
False
>>> ask(Q.nonpositive(-I))
False
```

### Handler

Multiply dispatched method: NonPositiveHandler

Handler for key 'nonpositive'.

**handler = <dispatched NonPositiveHandler>**

## Nonnegative

**class** sympy.assumptions.predicates.order.**NonNegativePredicate**(**args*, ***kwargs*)

Nonnegative real number predicate.

### Explanation

`ask(Q.nonnegative(x))` is true iff `x` belongs to the set of positive numbers including zero.

- **Note that `Q.nonnegative` and `~Q.negative` are *not* the same thing.** `~Q.negative(x)` simply means that `x` is not negative, whereas `Q.nonnegative(x)` means that `x` is real and not negative, i.e., `Q.nonnegative(x)` is logically equivalent to `Q.zero(x) | Q.positive(x)`. So for example, `~Q.negative(I)` is true, whereas `Q.nonnegative(I)` is false.

### Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.nonnegative(1))
True
>>> ask(Q.nonnegative(0))
True
>>> ask(Q.nonnegative(-1))
False
```

(continues on next page)

(continued from previous page)

```
>>> ask(Q.nonnegative(I))
False
>>> ask(Q.nonnegative(-I))
False
```

#### Handler

Multiply dispatched method: NonNegativeHandler

Handler for Q.nonnegative.

**handler = <dispatched NonNegativeHandler>**

### Sets

### Integer

**class** sympy.assumptions.predicates.sets.**IntegerPredicate**(**args*, ***kwargs*)

Integer predicate.

#### Explanation

`Q.integer(x)` is true iff `x` belongs to the set of integer numbers.

#### Examples

```
>>> from sympy import Q, ask, S
>>> ask(Q.integer(5))
True
>>> ask(Q.integer(S(1)/2))
False
```

#### Handler

Multiply dispatched method: IntegerHandler

Handler for Q.integer.

Test that an expression belongs to the field of integer numbers.

### References

[R20]

**handler = <dispatched IntegerHandler>**

## Rational

**class** sympy.assumptions.predicates.sets.**RationalPredicate**(**args*, ***kwargs*)

Rational number predicate.

### Explanation

`Q.rational(x)` is true iff `x` belongs to the set of rational numbers.

### Examples

```
>>> from sympy import ask, Q, pi, S
>>> ask(Q.rational(0))
True
>>> ask(Q.rational(S(1)/2))
True
>>> ask(Q.rational(pi))
False
```

### Handler

Multiply dispatched method: RationalHandler

Handler for Q.rational.

Test that an expression belongs to the field of rational numbers.

### References

[R21]

**handler = <dispatched RationalHandler>**

### Irrational

**class** sympy.assumptions.predicates.sets.**IrrationalPredicate**(**args*, ***kwargs*)

Irrational number predicate.

#### Explanation

`Q.irrational(x)` is true iff `x` is any real number that cannot be expressed as a ratio of integers.

#### Examples

```
>>> from sympy import ask, Q, pi, S, I
>>> ask(Q.irrational(0))
False
>>> ask(Q.irrational(S(1)/2))
False
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(I))
False
```

#### Handler

Multiply dispatched method: IrrationalHandler

Handler for Q.irrational.

Test that an expression is irrational numbers.

#### References

[R22]

**handler = <dispatched IrrationalHandler>**

### Real

**class** sympy.assumptions.predicates.sets.**RealPredicate**(**args*, ***kwargs*)

Real number predicate.

### Explanation

`Q.real(x)` is true iff `x` is a real number, i.e., it is in the interval $(-\infty, \infty)$. Note that, in particular the infinities are not real. Use `Q.extended_real` if you want to consider those as well.

A few important facts about reals:

- **Every real number is positive, negative, or zero. Furthermore,**
  because these sets are pairwise disjoint, each real number is exactly one of those three.
- Every real number is also complex.
- Every real number is finite.
- Every real number is either rational or irrational.
- Every real number is either algebraic or transcendental.
- **The facts `Q.negative`, `Q.zero`, `Q.positive`,**
  `Q.nonnegative`, `Q.nonpositive`, `Q.nonzero`, `Q.integer`, `Q.rational`, and `Q.irrational` all imply `Q.real`, as do all facts that imply those facts.
- **The facts `Q.algebraic`, and `Q.transcendental` do not imply**
  `Q.real`; they imply `Q.complex`. An algebraic or transcendental number may or may not be real.
- **The "non" facts (i.e., `Q.nonnegative`, `Q.nonzero`,**
  `Q.nonpositive` and `Q.noninteger`) are not equivalent to not the fact, but rather, not the fact *and* `Q.real`. For example, `Q.nonnegative` means `~Q.negative & Q.real`. So for example, `I` is not nonnegative, nonzero, or nonpositive.

### Examples

```
>>> from sympy import Q, ask, symbols
>>> x = symbols('x')
>>> ask(Q.real(x), Q.positive(x))
True
>>> ask(Q.real(0))
True
```

### Handler

Multiply dispatched method: RealHandler

Handler for Q.real.

Test that an expression belongs to the field of real numbers.

### References

[R23]

**handler = <dispatched RealHandler>**

## Extended real

**class** sympy.assumptions.predicates.sets.**ExtendedRealPredicate**(**args*, ***kwargs*)

Extended real predicate.

### Explanation

`Q.extended_real(x)` is true iff `x` is a real number or $\{-\infty, \infty\}$.

See documentation of `Q.real` for more information about related facts.

### Examples

```
>>> from sympy import ask, Q, oo, I
>>> ask(Q.extended_real(1))
True
>>> ask(Q.extended_real(I))
False
>>> ask(Q.extended_real(oo))
True
```

### Handler

Multiply dispatched method: ExtendedRealHandler

Handler for Q.extended_real.

Test that an expression belongs to the field of extended real

numbers, that is real numbers union {Infinity, -Infinity}.

**handler = <dispatched ExtendedRealHandler>**

## Hermitian

**class** sympy.assumptions.predicates.sets.**HermitianPredicate**(**args*, ***kwargs*)

Hermitian predicate.

#### Explanation

`ask(Q.hermitian(x))` is true iff `x` belongs to the set of Hermitian operators.

#### Handler

Multiply dispatched method: HermitianHandler

Handler for Q.hermitian.

Test that an expression belongs to the field of Hermitian operators.

#### References

[R24]

**handler = <dispatched HermitianHandler>**

### Complex

**class** sympy.assumptions.predicates.sets.**ComplexPredicate**(**args*, ***kwargs*)

Complex number predicate.

#### Explanation

`Q.complex(x)` is true iff `x` belongs to the set of complex numbers. Note that every complex number is finite.

#### Examples

```
>>> from sympy import Q, Symbol, ask, I, oo
>>> x = Symbol('x')
>>> ask(Q.complex(0))
True
>>> ask(Q.complex(2 + 3*I))
True
>>> ask(Q.complex(oo))
False
```

### Handler

Multiply dispatched method: ComplexHandler

Handler for Q.complex.

Test that an expression belongs to the field of complex numbers.

### References

[R25]

**handler = <dispatched ComplexHandler>**

## Imaginary

**class** sympy.assumptions.predicates.sets.**ImaginaryPredicate**(**args*, ***kwargs*)

Imaginary number predicate.

### Explanation

`Q.imaginary(x)` is true iff `x` can be written as a real number multiplied by the imaginary unit `I`. Please note that `0` is not considered to be an imaginary number.

### Examples

```
>>> from sympy import Q, ask, I
>>> ask(Q.imaginary(3*I))
True
>>> ask(Q.imaginary(2 + 3*I))
False
>>> ask(Q.imaginary(0))
False
```

### Handler

Multiply dispatched method: ImaginaryHandler

Handler for Q.imaginary.

Test that an expression belongs to the field of imaginary numbers,

that is, numbers in the form x*I, where x is real.

#### References

[R26]

**handler = <dispatched ImaginaryHandler>**

### Antihermitian

**class** sympy.assumptions.predicates.sets.**AntihermitianPredicate**(**args*, ***kwargs*)

Antihermitian predicate.

#### Explanation

`Q.antihermitian(x)` is true iff `x` belongs to the field of antihermitian operators, i.e., operators in the form `x*I`, where `x` is Hermitian.

#### Handler

Multiply dispatched method: AntiHermitianHandler

Handler for Q.antihermitian.

Test that an expression belongs to the field of anti-Hermitian

operators, that is, operators in the form x*I, where x is Hermitian.

#### References

[R27]

**handler = <dispatched AntiHermitianHandler>**

### Algebraic

**class** sympy.assumptions.predicates.sets.**AlgebraicPredicate**(**args*, ***kwargs*)

Algebraic number predicate.

#### Explanation

`Q.algebraic(x)` is true iff `x` belongs to the set of algebraic numbers. `x` is algebraic if there is some polynomial in `p(x)\in \mathbb\{Q\}[x]` such that `p(x) = 0`.

#### Examples

```
>>> from sympy import ask, Q, sqrt, I, pi
>>> ask(Q.algebraic(sqrt(2)))
True
>>> ask(Q.algebraic(I))
True
>>> ask(Q.algebraic(pi))
False
```

#### Handler

Multiply dispatched method: AskAlgebraicpredicateHandler

Handler for key AskAlgebraicpredicateHandler

#### References

[R28]

**AlgebraicHandler = <dispatched AlgebraicHandler>**

**handler = <dispatched AskAlgebraicpredicateHandler>**

### Transcendental

**class** sympy.assumptions.predicates.sets.**TranscendentalPredicate**(**args*, ***kwargs*)

Transcedental number predicate.

#### Explanation

`Q.transcendental(x)` is true iff `x` belongs to the set of transcendental numbers. A transcendental number is a real or complex number that is not algebraic.

#### Handler

Multiply dispatched method: Transcendental

Handler for Q.transcendental key.

**handler = <dispatched Transcendental>**

### Performance improvements

On queries that involve symbolic coefficients, logical inference is used. Work on improving satisfiable function (sympy.logic.inference.satisfiable) should result in notable speed improvements.

Logic inference used in one ask could be used to speed up further queries, and current system does not take advantage of this. For example, a truth maintenance system (https://en.wikipedia.org/wiki/Truth_maintenance_system) could be implemented.

### Misc

You can find more examples in the form of tests in the directory `sympy/assumptions/tests/`

### Calculus

Calculus-related methods.

This module implements a method to find Euler-Lagrange Equations for given Lagrangian.

`sympy.calculus.euler.`**`euler_equations`**(*L*, *funcs=()*, *vars=()*)

Find the Euler-Lagrange equations [R29] for a given Lagrangian.

**Parameters**

**L** : Expr

The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

For example, in the case of two functions $f(x,y)$, $g(x,y)$ and two independent variables $x$, $y$ the Lagrangian has the form:

$$L\left(f(x,y), g(x,y), \frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}, \frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}, x, y\right)$$

In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this cannot be done).

**funcs** : Function or an iterable of Functions

The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

**vars** : Symbol or an iterable of Symbols

The Symbols that are the independent variables of the functions.

**Returns**

**eqns** : list of Eq

The list of differential equations, one for each function.

### Examples

```
>>> from sympy import euler_equations, Symbol, Function
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), (t, 2)), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), (t, 2)) + Derivative(u(t, x), (x, 2)), 0)]
```

### References

[R29]

## Singularities

This module implements algorithms for finding singularities for a function and identifying types of functions.

The differential calculus methods in this module include methods to identify the following function types in the given `Interval`: - Increasing - Strictly Increasing - Decreasing - Strictly Decreasing - Monotonic

`sympy.calculus.singularities.`**`is_decreasing`**(*expression*, *interval=Reals*, *symbol=None*)

Return whether the function is decreasing in the given interval.

**Parameters**

**expression** : Expr

The target function which is being checked.

**interval** : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if `expression` is decreasing (either strictly decreasing or constant) in the given `interval`, False otherwise.

### Examples

```
>>> from sympy import is_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_decreasing(1/(x**2 - 3*x), Interval.open(S(3)/2, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, 1.5))
False
>>> is_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

sympy.calculus.singularities.**is_increasing**(*expression*, *interval=Reals*, *symbol=None*)

Return whether the function is increasing in the given interval.

**Parameters**

**expression** : Expr

The target function which is being checked.

**interval** : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if `expression` is increasing (either strictly increasing or constant) in the given `interval`, False otherwise.

### Examples

```
>>> from sympy import is_increasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_increasing(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_increasing(-x**2, Interval(-oo, 0))
True
>>> is_increasing(-x**2, Interval(0, oo))
```

(continues on next page)

(continued from previous page)

```
False
>>> is_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval(-2, 3))
False
>>> is_increasing(x**2 + y, Interval(1, 2), x)
True
```

sympy.calculus.singularities.**is_monotonic**(*expression*, *interval=Reals*, *symbol=None*)

Return whether the function is monotonic in the given interval.

**Parameters**

**expression** : Expr

The target function which is being checked.

**interval** : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if `expression` is monotonic in the given `interval`, False otherwise.

**Raises**

**NotImplementedError**

Monotonicity check has not been implemented for the queried function.

### Examples

```
>>> from sympy import is_monotonic
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(S(3)/2, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.open(1.5, 3))
True
>>> is_monotonic(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_monotonic(x**3 - 3*x**2 + 4*x, S.Reals)
True
>>> is_monotonic(-x**2, S.Reals)
False
>>> is_monotonic(x**2 + y + 1, Interval(1, 2), x)
True
```