Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

FiniteSet

class sympy.sets.sets.**FiniteSet**(**args*, ***kwargs*)

Represents a finite set of Sympy expressions.

Examples

```
>>> from sympy import FiniteSet, Symbol, Interval, Naturals0
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True
>>> FiniteSet(1, (1, 2), Symbol('x'))
{1, x, (1, 2)}
>>> FiniteSet(Interval(1, 2), Naturals0, {1, 2})
FiniteSet({1, 2}, Interval(1, 2), Naturals0)
>>> members = [1, 2, 3, 4]
>>> f = FiniteSet(*members)
>>> f
{1, 2, 3, 4}
>>> f - FiniteSet(2)
{1, 3, 4}
>>> f + FiniteSet(2, 5)
{1, 2, 3, 4, 5}
```

References

[R748]

as_relational(*symbol*)

Rewrite a FiniteSet in terms of equalities and logic operators.

Compound Sets

Union

class sympy.sets.sets.**Union**(**args*, ***kwargs*)

Represents a union of sets as a *Set* (page 1185).

Examples

```
>>> from sympy import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
Union(Interval(1, 2), Interval(3, 4))
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
Interval(1, 3)
```

See also:

Intersection (page 1198)

References

[R749]

as_relational(*symbol*)

Rewrite a Union in terms of equalities and logic operators.

Intersection

class sympy.sets.sets.**Intersection**(**args*, ***kwargs*)

Represents an intersection of sets as a *Set* (page 1185).

Examples

```
>>> from sympy import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
Interval(2, 3)
```

We often use the .intersect method

```
>>> Interval(1,3).intersect(Interval(2,4))
Interval(2, 3)
```

See also:

Union (page 1197)

References

[R750]

as_relational(*symbol*)

Rewrite an Intersection in terms of equalities and logic operators

ProductSet

class sympy.sets.sets.**ProductSet**(**sets*, ***assumptions*)

Represents a Cartesian Product of Sets.

Explanation

Returns a Cartesian product given several sets as either an iterable or individual arguments.

Can use * operator on any sets for convenient shorthand.

Examples

```
>>> from sympy import Interval, FiniteSet, ProductSet
>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
ProductSet(Interval(0, 5), {1, 2, 3})
```

```
>>> (2, 2) in ProductSet(I, S)
True
```

```
>>> Interval(0, 1) * Interval(0, 1) # The unit square
ProductSet(Interval(0, 1), Interval(0, 1))
```

```
>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
{(H, H), (H, T), (T, H), (T, T)}
```

The Cartesian product is not commutative or associative e.g.:

```
>>> I*S == S*I
False
>>> (I*I)*I == I*(I*I)
False
```

Notes

- Passes most operations down to the argument sets

References

[R751]

property is_iterable

A property method which tests whether a set is iterable or not. Returns True if set is iterable, otherwise returns False.

Examples

```
>>> from sympy import FiniteSet, Interval
>>> I = Interval(0, 1)
>>> A = FiniteSet(1, 2, 3, 4, 5)
>>> I.is_iterable
False
>>> A.is_iterable
True
```

Complement

class sympy.sets.sets.**Complement**(*a*, *b*, *evaluate=True*)

Represents the set difference or relative complement of a set with another set.

$$A - B = \{x \in A \mid x \notin B\}$$

Examples

```
>>> from sympy import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}
```

See also:

Intersection (page 1198), *Union* (page 1197)

References

[R752]

as_relational(*symbol*)
Rewrite a complement in terms of equalities and logic operators

static reduce(*A*, *B*)
Simplify a *Complement* (page 1200).

SymmetricDifference

class sympy.sets.sets.**SymmetricDifference**(*a*, *b*, *evaluate=True*)
Represents the set of elements which are in either of the sets and not in their intersection.

Examples

```
>>> from sympy import SymmetricDifference, FiniteSet
>>> SymmetricDifference(FiniteSet(1, 2, 3), FiniteSet(3, 4, 5))
{1, 2, 4, 5}
```

See also:

Complement (page 1200), *Union* (page 1197)

References

[R753]

as_relational(*symbol*)
Rewrite a symmetric_difference in terms of equalities and logic operators

DisjointUnion

class sympy.sets.sets.**DisjointUnion**(**sets*)
Represents the disjoint union (also known as the external disjoint union) of a finite number of sets.

Examples

```
>>> from sympy import DisjointUnion, FiniteSet, Interval, Union, Symbol
>>> A = FiniteSet(1, 2, 3)
>>> B = Interval(0, 5)
>>> DisjointUnion(A, B)
DisjointUnion({1, 2, 3}, Interval(0, 5))
>>> DisjointUnion(A, B).rewrite(Union)
Union(ProductSet({1, 2, 3}, {0}), ProductSet(Interval(0, 5), {1}))
```

(continues on next page)

(continued from previous page)

```
>>> C = FiniteSet(Symbol('x'), Symbol('y'), Symbol('z'))
>>> DisjointUnion(C, C)
DisjointUnion({x, y, z}, {x, y, z})
>>> DisjointUnion(C, C).rewrite(Union)
ProductSet({x, y, z}, {0, 1})
```

References

https://en.wikipedia.org/wiki/Disjoint_union

Singleton Sets

EmptySet

class sympy.sets.sets.**EmptySet**

Represents the empty set. The empty set is available as a singleton as `S.EmptySet`.

Examples

```
>>> from sympy import S, Interval
>>> S.EmptySet
EmptySet
```

```
>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet
```

See also:

UniversalSet (page 1202)

References

[R754]

UniversalSet

class sympy.sets.sets.**UniversalSet**

Represents the set of all things. The universal set is available as a singleton as `S.UniversalSet`.

Examples

```
>>> from sympy import S, Interval
>>> S.UniversalSet
UniversalSet
```

```
>>> Interval(1, 2).intersect(S.UniversalSet)
Interval(1, 2)
```

See also:

EmptySet (page 1202)

References

[R755]

Special Sets

Rationals

class sympy.sets.fancysets.**Rationals**

Represents the rational numbers. This set is also available as the singleton `S.Rationals`.

Examples

```
>>> from sympy import S
>>> S.Half in S.Rationals
True
>>> iterable = iter(S.Rationals)
>>> [next(iterable) for i in range(12)]
[0, 1, -1, 1/2, 2, -1/2, -2, 1/3, 3, -1/3, -3, 2/3]
```

Naturals

class sympy.sets.fancysets.**Naturals**

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the singleton `S.Naturals`.

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```

See also:

Naturals0 **(page 1204)**
non-negative integers (i.e. includes 0, too)

Integers **(page 1204)**
also includes negative integers

Naturals0

class sympy.sets.fancysets.**Naturals0**

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

Naturals **(page 1203)**
positive integers; does not include 0

Integers **(page 1204)**
also includes the negative integers

Integers

class sympy.sets.fancysets.**Integers**

Represents all integers: positive, negative and zero. This set is also available as the singleton `S.Integers`.

Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
>>> next(iterable)
2
```

```
>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

See also:

Naturals0 **(page 1204)**
non-negative integers

Integers **(page 1204)**
positive and negative integers and zero

Reals

class sympy.sets.fancysets.**Reals**

Represents all real numbers from negative infinity to positive infinity, including all integer, rational and irrational numbers. This set is also available as the singleton `S.Reals`.

Examples

```
>>> from sympy import S, Rational, pi, I
>>> 5 in S.Reals
True
>>> Rational(-1, 2) in S.Reals
True
>>> pi in S.Reals
True
>>> 3*I in S.Reals
False
>>> S.Reals.contains(pi)
True
```

See also:

ComplexRegion (page 1209)

Complexes

class sympy.sets.fancysets.**Complexes**

The Set of all complex numbers

Examples

```
>>> from sympy import S, I
>>> S.Complexes
Complexes
>>> 1 + I in S.Complexes
True
```

See also:

Reals (page 1205), *ComplexRegion* (page 1209)

ImageSet

class sympy.sets.fancysets.**ImageSet**(*flambda*, **sets*)

Image of a set under a mathematical function. The transformation must be given as a Lambda function which has as many arguments as the elements of the set upon which it operates, e.g. 1 argument when acting on the set of integers or 2 arguments when acting on a complex region.

This function is not normally called directly, but is called from `imageset`.

Examples

```
>>> from sympy import Symbol, S, pi, Dummy, Lambda
>>> from sympy import FiniteSet, ImageSet, Interval
```

```
>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>>> 4 in squares
True
>>> 5 in squares
False
```

```
>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
{1, 4, 9}
```

```
>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
```

(continues on next page)

(continued from previous page)

```
9
16
```

If you want to get value for x = 2, 1/2 etc. (Please check whether the x value is in base_set or not before passing it as args)

```
>>> squares.lamda(2)
4
>>> squares.lamda(S(1)/2)
1/4
```

```
>>> n = Dummy('n')
>>> solutions = ImageSet(Lambda(n, n*pi), S.Integers) # solutions of␣
↪sin(x) = 0
>>> dom = Interval(-1, 1)
>>> dom.intersect(solutions)
{0}
```

See also:

sympy.sets.sets.imageset (page 1193)

Range

class sympy.sets.fancysets.**Range**(**args*)

Represents a range of integers. Can be called as `Range(stop)`, `Range(start, stop)`, or `Range(start, stop, step)`; when `step` is not given it defaults to 1.

`Range(stop)` is the same as `Range(0, stop, 1)` and the stop value (just as for Python ranges) is not included in the Range values.

```
>>> from sympy import Range
>>> list(Range(3))
[0, 1, 2]
```

The step can also be negative:

```
>>> list(Range(10, 0, -2))
[10, 8, 6, 4, 2]
```

The stop value is made canonical so equivalent ranges always have the same args:

```
>>> Range(0, 10, 3)
Range(0, 12, 3)
```

Infinite ranges are allowed. `oo` and `-oo` are never included in the set (`Range` is always a subset of `Integers`). If the starting point is infinite, then the final value is `stop - step`. To iterate such a range, it needs to be reversed:

```
>>> from sympy import oo
>>> r = Range(-oo, 1)
>>> r[-1]
```

(continues on next page)

(continued from previous page)

```
0
>>> next(iter(r))
Traceback (most recent call last):
...
TypeError: Cannot iterate over Range with infinite start
>>> next(iter(r.reversed))
0
```

Although Range is a Set (and supports the normal set operations) it maintains the order of the elements and can be used in contexts where range would be used.

```
>>> from sympy import Interval
>>> Range(0, 10, 2).intersect(Interval(3, 7))
Range(4, 8, 2)
>>> list(_)
[4, 6]
```

Although slicing of a Range will always return a Range – possibly empty – an empty set will be returned from any intersection that is empty:

```
>>> Range(3)[:0]
Range(0, 0, 1)
>>> Range(3).intersect(Interval(4, oo))
EmptySet
>>> Range(3).intersect(Range(4, oo))
EmptySet
```

Range will accept symbolic arguments but has very limited support for doing anything other than displaying the Range:

```
>>> from sympy import Symbol, pprint
>>> from sympy.abc import i, j, k
>>> Range(i, j, k).start
i
>>> Range(i, j, k).inf
Traceback (most recent call last):
...
ValueError: invalid method for symbolic range
```

Better success will be had when using integer symbols:

```
>>> n = Symbol('n', integer=True)
>>> r = Range(n, n + 20, 3)
>>> r.inf
n
>>> pprint(r)
{n, n + 3, ..., n + 18}
```

as_relational(x)

Rewrite a Range in terms of equalities and logic operators.

property reversed

Return an equivalent Range in the opposite order.

Examples

```
>>> from sympy import Range
>>> Range(10).reversed
Range(9, -1, -1)
```

ComplexRegion

class sympy.sets.fancysets.**ComplexRegion**(*sets*, *polar=False*)

Represents the Set of all Complex Numbers. It can represent a region of Complex Plane in both the standard forms Polar and Rectangular coordinates.

- Polar Form Input is in the form of the ProductSet or Union of ProductSets of the intervals of `r` and `theta`, and use the flag `polar=True`.

$$Z = \{z \in \mathbb{C} \mid z = r \times (\cos(\theta) + I\sin(\theta)), r \in [\mathtt{r}], \theta \in [\mathtt{theta}]\}$$

- Rectangular Form Input is in the form of the ProductSet or Union of ProductSets of interval of x and y, the real and imaginary parts of the Complex numbers in a plane. Default input type is in rectangular form.

$$Z = \{z \in \mathbb{C} \mid z = x + Iy, x \in [\mathrm{re}(z)], y \in [\mathrm{im}(z)]\}$$

Examples

```
>>> from sympy import ComplexRegion, Interval, S, I, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 6)
>>> c1 = ComplexRegion(a*b)  # Rectangular Form
>>> c1
CartesianComplexRegion(ProductSet(Interval(2, 3), Interval(4, 6)))
```

- c1 represents the rectangular region in complex plane surrounded by the coordinates (2, 4), (3, 4), (3, 6) and (2, 6), of the four vertices.

```
>>> c = Interval(1, 8)
>>> c2 = ComplexRegion(Union(a*b, b*c))
>>> c2
CartesianComplexRegion(Union(ProductSet(Interval(2, 3), Interval(4, 6)),␣
↪ProductSet(Interval(4, 6), Interval(1, 8))))
```

- c2 represents the Union of two rectangular regions in complex plane. One of them surrounded by the coordinates of c1 and other surrounded by the coordinates (4, 1), (6, 1), (6, 8) and (4, 8).

```
>>> 2.5 + 4.5*I in c1
True
>>> 2.5 + 6.5*I in c1
False
```

```
>>> r = Interval(0, 1)
>>> theta = Interval(0, 2*S.Pi)
>>> c2 = ComplexRegion(r*theta, polar=True)  # Polar Form
>>> c2  # unit Disk
PolarComplexRegion(ProductSet(Interval(0, 1), Interval.Ropen(0, 2*pi)))
```

- c2 represents the region in complex plane inside the Unit Disk centered at the origin.

```
>>> 0.5 + 0.5*I in c2
True
>>> 1 + 2*I in c2
False
```

```
>>> unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, 2*S.Pi),␣
↪polar=True)
>>> upper_half_unit_disk = ComplexRegion(Interval(0, 1)*Interval(0, S.
↪Pi), polar=True)
>>> intersection = unit_disk.intersect(upper_half_unit_disk)
>>> intersection
PolarComplexRegion(ProductSet(Interval(0, 1), Interval(0, pi)))
>>> intersection == upper_half_unit_disk
True
```

See also:

CartesianComplexRegion (page 1212), *PolarComplexRegion* (page 1212), *Complexes* (page 1206)

property a_interval

Return the union of intervals of x when, self is in rectangular form, or the union of intervals of r when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.a_interval
Interval(2, 3)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.a_interval
Union(Interval(2, 3), Interval(4, 5))
```

property b_interval

Return the union of intervals of y when, self is in rectangular form, or the union of intervals of $theta$ when self is in polar form.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.b_interval
Interval(4, 5)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.b_interval
Interval(1, 7)
```

classmethod from_real(*sets*)

Converts given subset of real numbers to a complex region.

Examples

```
>>> from sympy import Interval, ComplexRegion
>>> unit = Interval(0,1)
>>> ComplexRegion.from_real(unit)
CartesianComplexRegion(ProductSet(Interval(0, 1), {0}))
```

property psets

Return a tuple of sets (ProductSets) input of the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.psets
(ProductSet(Interval(2, 3), Interval(4, 5)),)
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.psets
(ProductSet(Interval(2, 3), Interval(4, 5)), ProductSet(Interval(4,␣
↪5), Interval(1, 7)))
```

property sets

Return raw input sets to the self.

Examples

```
>>> from sympy import Interval, ComplexRegion, Union
>>> a = Interval(2, 3)
>>> b = Interval(4, 5)
>>> c = Interval(1, 7)
>>> C1 = ComplexRegion(a*b)
>>> C1.sets
ProductSet(Interval(2, 3), Interval(4, 5))
>>> C2 = ComplexRegion(Union(a*b, b*c))
>>> C2.sets
Union(ProductSet(Interval(2, 3), Interval(4, 5)),␣
↪ProductSet(Interval(4, 5), Interval(1, 7)))
```

class sympy.sets.fancysets.**CartesianComplexRegion**(*sets*)

Set representing a square region of the complex plane.

$$Z = \{z \in \mathbb{C} \mid z = x + Iy, x \in [\mathrm{re}(z)], y \in [\mathrm{im}(z)]\}$$

Examples

```
>>> from sympy import ComplexRegion, I, Interval
>>> region = ComplexRegion(Interval(1, 3) * Interval(4, 6))
>>> 2 + 5*I in region
True
>>> 5*I in region
False
```

See also:

ComplexRegion (page 1209), *PolarComplexRegion* (page 1212), *Complexes* (page 1206)

class sympy.sets.fancysets.**PolarComplexRegion**(*sets*)

Set representing a polar region of the complex plane.

$$Z = \{z \in \mathbb{C} \mid z = r \times (\cos(\theta) + I\sin(\theta)), r \in [\mathsf{r}], \theta \in [\mathsf{theta}]\}$$

Examples

```
>>> from sympy import ComplexRegion, Interval, oo, pi, I
>>> rset = Interval(0, oo)
>>> thetaset = Interval(0, pi)
>>> upper_half_plane = ComplexRegion(rset * thetaset, polar=True)
>>> 1 + I in upper_half_plane
True
>>> 1 - I in upper_half_plane
False
```

See also:

ComplexRegion (page 1209), *CartesianComplexRegion* (page 1212), *Complexes* (page 1206)

sympy.sets.fancysets.**normalize_theta_set**(*theta*)

Normalize a Real Set $theta$ in the interval $[0, 2\pi)$. It returns a normalized value of theta in the Set. For Interval, a maximum of one cycle $[0, 2\pi]$, is returned i.e. for theta equal to $[0, 10\pi]$, returned normalized value would be $[0, 2\pi)$. As of now intervals with end points as non-multiples of `pi` is not supported.

Raises

NotImplementedError

The algorithms for Normalizing theta Set are not yet implemented.

ValueError

The input is not valid, i.e. the input is not a real set.

RuntimeError

It is a bug, please report to the github issue tracker.

Examples

```
>>> from sympy.sets.fancysets import normalize_theta_set
>>> from sympy import Interval, FiniteSet, pi
>>> normalize_theta_set(Interval(9*pi/2, 5*pi))
Interval(pi/2, pi)
>>> normalize_theta_set(Interval(-3*pi/2, pi/2))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-pi/2, pi/2))
Union(Interval(0, pi/2), Interval.Ropen(3*pi/2, 2*pi))
>>> normalize_theta_set(Interval(-4*pi, 3*pi))
Interval.Ropen(0, 2*pi)
>>> normalize_theta_set(Interval(-3*pi/2, -pi/2))
Interval(pi/2, 3*pi/2)
>>> normalize_theta_set(FiniteSet(0, pi, 3*pi))
{0, pi}
```

Power sets

PowerSet

class sympy.sets.powerset.**PowerSet**(*arg*, *evaluate=None*)

A symbolic object representing a power set.

Parameters

arg : Set

The set to take power of.

evaluate : bool

The flag to control evaluation.

If the evaluation is disabled for finite sets, it can take advantage of using subset test as a membership test.

Notes

Power set $\mathcal{P}(S)$ is defined as a set containing all the subsets of S.

If the set S is a finite set, its power set would have $2^{|S|}$ elements, where $|S|$ denotes the cardinality of S.

Examples

```
>>> from sympy import PowerSet, S, FiniteSet
```

A power set of a finite set:

```
>>> PowerSet(FiniteSet(1, 2, 3))
PowerSet({1, 2, 3})
```

A power set of an empty set:

```
>>> PowerSet(S.EmptySet)
PowerSet(EmptySet)
>>> PowerSet(PowerSet(S.EmptySet))
PowerSet(PowerSet(EmptySet))
```

A power set of an infinite set:

```
>>> PowerSet(S.Reals)
PowerSet(Reals)
```

Evaluating the power set of a finite set to its explicit form:

```
>>> PowerSet(FiniteSet(1, 2, 3)).rewrite(FiniteSet)
FiniteSet(EmptySet, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3})
```

References

[R756], [R757]

Iteration over sets

For set unions, $\{a, b\} \cup \{x, y\}$ can be treated as $\{a, b, x, y\}$ for iteration regardless of the distinctiveness of the elements, however, for set intersections, assuming that $\{a, b\} \cap \{x, y\}$ is $\varnothing$ or $\{a, b\}$ would not always be valid, since some of a, b, x or y may or may not be the elements of the intersection.

Iterating over the elements of a set involving intersection, complement, or symmetric difference yields (possibly duplicate) elements of the set provided that all elements are known to be the elements of the set. If any element cannot be determined to be a member of a set then the iteration gives `TypeError`. This happens in the same cases where `x in y` would give an error.

There are some reasons to implement like this, even if it breaks the consistency with how the python set iterator works. We keep in mind that sympy set comprehension like

`FiniteSet(*s)` from a existing sympy sets could be a common usage. And this approach would make `FiniteSet(*s)` to be consistent with any symbolic set processing methods like `FiniteSet(*simplify(s))`.

Condition Sets

ConditionSet

class sympy.sets.conditionset.**ConditionSet**(*sym*, *condition*, *base_set=UniversalSet*)

Set of elements which satisfies a given condition.

$$\{x \mid \text{condition}(x) = \texttt{True}, x \in S\}$$

Examples

```
>>> from sympy import Symbol, S, ConditionSet, pi, Eq, sin, Interval
>>> from sympy.abc import x, y, z
```

```
>>> sin_sols = ConditionSet(x, Eq(sin(x), 0), Interval(0, 2*pi))
>>> 2*pi in sin_sols
True
>>> pi/2 in sin_sols
False
>>> 3*pi in sin_sols
False
>>> 5 in ConditionSet(x, x**2 > 4, S.Reals)
True
```

If the value is not in the base set, the result is false:

```
>>> 5 in ConditionSet(x, x**2 > 4, Interval(2, 4))
False
```

Notes

Symbols with assumptions should be avoided or else the condition may evaluate without consideration of the set:

```
>>> n = Symbol('n', negative=True)
>>> cond = (n > 0); cond
False
>>> ConditionSet(n, cond, S.Integers)
EmptySet
```

Only free symbols can be changed by using $subs$:

```
>>> c = ConditionSet(x, x < 1, {x, z})
>>> c.subs(x, y)
ConditionSet(x, x < 1, {y, z})
```

To check if `pi` is in `c` use:

```
>>> pi in c
False
```

If no base set is specified, the universal set is implied:

```
>>> ConditionSet(x, x < 1).base_set
UniversalSet
```

Only symbols or symbol-like expressions can be used:

```
>>> ConditionSet(x + 1, x + 1 < 1, S.Integers)
Traceback (most recent call last):
...
ValueError: non-symbol dummy not recognized in condition
```

When the base set is a ConditionSet, the symbols will be unified if possible with preference for the outermost symbols:

```
>>> ConditionSet(x, x < y, ConditionSet(z, z + y < 2, S.Integers))
ConditionSet(x, (x < y) & (x + y < 2), Integers)
```

Relations on sets

class sympy.sets.conditionset.**Contains**(*x*, *s*)

Asserts that x is an element of the set S.

Examples

```
>>> from sympy import Symbol, Integer, S, Contains
>>> Contains(Integer(2), S.Integers)
True
>>> Contains(Integer(-2), S.Naturals)
False
>>> i = Symbol('i', integer=True)
>>> Contains(i, S.Naturals)
Contains(i, Naturals)
```

References

[R758]

SetKind

class sympy.sets.conditionset.**SetKind**(*element_kind=None*)

SetKind is kind for all Sets

Every instance of Set will have kind `SetKind` parametrised by the kind of the elements of the `Set`. The kind of the elements might be `NumberKind`, or `TupleKind` or something else. When not all elements have the same kind then the kind of the elements will be given as `UndefinedKind`.

Parameters

element_kind: Kind (optional)

The kind of the elements of the set. In a well defined set all elements will have the same kind. Otherwise the kind should *sympy.core.kind.UndefinedKind* (page 1074). The `element_kind` argument is optional but should only be omitted in the case of `EmptySet` whose kind is simply `SetKind()`

Examples

```
>>> from sympy import Interval
>>> Interval(1, 2).kind
SetKind(NumberKind)
>>> Interval(1,2).kind.element_kind
NumberKind
```

See also:

sympy.core.kind.NumberKind (page 1074), *sympy.matrices.common.MatrixKind* (page 1360), *sympy.core.containers.TupleKind* (page 1069)

5.8.4 Matrices

Contents

Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first `Matrix` object:

```
>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False, wrap_line=False)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1  0  0]
[       ]
[0  0  0]
>>> Matrix([M, (0, 0, -1)])
[1  0  0 ]
[        ]
[0  0  0 ]
[        ]
[0  0  -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
[ ]
[2]
[ ]
[3]
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])
[1  2  3]
[       ]
[4  5  6]
```

More interesting (and useful), is the ability to use a 2-variable function (or `lambda`) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):
...     if i == j:
...         return 1
...     else:
...         return 0
...
>>> Matrix(4, 4, f)
[1  0  0  0]
[           ]
[0  1  0  0]
[           ]
```

(continues on next page)

(continued from previous page)

```
[0  0  1  0]
[           ]
[0  0  0  1]
```

Finally let's use `lambda` to create a 1-line matrix with 1's in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1  0  1  0]
[           ]
[0  1  0  1]
[           ]
[1  0  1  0]
```

There are also a couple of special constructors for quick matrix construction: `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively, and `diag` to put matrices or elements along the diagonal:

```
>>> eye(4)
[1  0  0  0]
[           ]
[0  1  0  0]
[           ]
[0  0  1  0]
[           ]
[0  0  0  1]
>>> zeros(2)
[0  0]
[    ]
[0  0]
>>> zeros(2, 5)
[0  0  0  0  0]
[              ]
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[       ]
[1  1  1]
[       ]
[1  1  1]
>>> ones(1, 3)
[1  1  1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1  0  0]
[       ]
[0  1  2]
[       ]
[0  3  4]
```

Basic Manipulation

While learning to work with matrices, let's choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```
>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5
```

Since this is Python we're also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```
>>> M[0:2, 0:2]
[1  2]
[    ]
[4  5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1  2  3]
[       ]
[4  5  6]
```

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing `M2` didn't change `M`. Since we can slice, we can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1   2   3   4 ]
[               ]
[5   6   7   8 ]
[               ]
[9   10  11  12]
[               ]
[13  14  15  16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1   2   3   0 ]
[               ]
[5   6   7   8 ]
[               ]
[9   10  0   12]
[               ]
[13  14  15  16]
```

as well as assign slices:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1   2   3  4]
[             ]
[5   6   7  8]
[             ]
[9   10  0  0]
[             ]
[13  14  0  0]
```

All the standard arithmetic operations are supported:

```
>>> M = Matrix(([1,2,3],[4,5,6],[7,8,9]))
>>> M - M
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
```

(continues on next page)

(continued from previous page)

```
>>> M + M
[2   4   6 ]
[          ]
[8   10  12]
[          ]
[14  16  18]
>>> M * M
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[  ]
[29]
[  ]
[47]
>>> M**2
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)
>>> M
[4  5  6]
[       ]
[7  8  9]
>>> M.col_del(1)
>>> M
[4  6]
[    ]
[7  9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall that the `row_del()` and `col_del()` operations don't return a value - they simply change the matrix object. We can also ''glue'' together matrices of the appropriate size:

```
>>> M1 = eye(3)
```

(continues on next page)

(continued from previous page)

```
>>> M2 = zeros(3, 4)
>>> M1.row_join(M2)
[1  0  0  0  0  0  0]
[                   ]
[0  1  0  0  0  0  0]
[                   ]
[0  0  1  0  0  0  0]
>>> M3 = zeros(4, 3)
>>> M1.col_join(M3)
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
[       ]
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
```

Operations on entries

We are not restricted to having multiplication between two matrices:

```
>>> M = eye(3)
>>> 2*M
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
>>> 3*M
[3  0  0]
[       ]
[0  3  0]
[       ]
[0  0  3]
```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
```

If you want to extract a common factor from a matrix you can do so by applying `gcd` to the data of the matrix:

```
>>> from sympy.abc import x, y
>>> from sympy import gcd
>>> m = Matrix([[x, y], [1, x*y]]).inv('ADJ'); m
[  x*y       -y    ]
[--------  --------]
[ 2         2      ]
[x *y - y  x *y - y]
[                  ]
[  -1         x    ]
[--------  --------]
[ 2         2      ]
[x *y - y  x *y - y]
>>> gcd(tuple(_))
   1
--------
 2
x *y - y
>>> m/_
[x*y  -y]
[       ]
[-1   x ]
```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x  0  0]
[       ]
[0  x  0]
[       ]
[0  0  x]
>>> M.subs(x, 4)
[4  0  0]
[       ]
[0  4  0]
[       ]
[0  0  4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y  0  0]
[       ]
[0  y  0]
[       ]
[0  0  y]
```

Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```
>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0
```

Another common operation is the inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but we can specify it be done by LU decomposition as well:

```
>>> M2.inv()
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M2.inv(method="LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M.inv(method="LU")
[-3/14  1/14  1/2 ]
[                 ]
[-1/28  5/28  -1/4]
[                 ]
[ 3/7   -1/7   0  ]
>>> M * M.inv(method="LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
```

We can perform a QR factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[  ___    ___     ___ ]
[\/ 6   -\/ 3    -\/ 2  ]
[-----  -------  -------]
[  6       3        2   ]
[                       ]
```

(continues on next page)

(continued from previous page)

```
[  ___      ___       ___  ]
[\/ 6   -\/ 3     \/ 2   ]
[-----  -------   -----  ]
[  6       3        2    ]
[                        ]
[  ___     ___           ]
[\/ 6    \/ 3            ]
[-----   -----      0     ]
[  3       3              ]
>>> R
[                  ___   ]
[  ___    4*\/ 6        ___]
[\/ 6    -------   2*\/ 6 ]
[           3              ]
[                          ]
[           ___            ]
[         \/ 3             ]
[  0      -----      0     ]
[           3              ]
[                          ]
[                    ___   ]
[  0        0      \/ 2    ]
>>> Q*R
[1  1  1]
[       ]
[1  1  3]
[       ]
[2  3  4]
```

In addition to the solvers in the `solver.py` file, we can solve the system Ax=b by passing the b vector to the matrix A's LUsolve function. Here we'll cheat a little choose A and x then multiply to get b. Then we can solve for x and check that it's correct:

```
>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]
```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```
>>> L = [Matrix([2,3,5]), Matrix([3,6,2]), Matrix([8,3,6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)
```

Let's take a look at the vectors:

```
>>> for i in out1:
...     print(i)
...
Matrix([[2], [3], [5]])
Matrix([[23/19], [63/19], [-47/19]])
Matrix([[1692/353], [-1551/706], [-423/706]])
>>> for i in out2:
...      print(i)
...
Matrix([[sqrt(38)/19], [3*sqrt(38)/38], [5*sqrt(38)/38]])
Matrix([[23*sqrt(6707)/6707], [63*sqrt(6707)/6707], [-47*sqrt(6707)/6707]])
Matrix([[12*sqrt(706)/353], [-11*sqrt(706)/706], [-3*sqrt(706)/706]])
```

We can spot-check their orthogonality with dot() and their normality with norm():

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```


So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the `matrices.py` file for all functionality.

MatrixDeterminant Class Reference

class sympy.matrices.matrices.**MatrixDeterminant**

Provides basic matrix determinant operations. Should not be instantiated directly. See `determinant.py` for their implementations.

adjugate(*method='berkowitz'*)

Returns the adjugate, or classical adjoint, of a matrix. That is, the transpose of the matrix of cofactors.

https://en.wikipedia.org/wiki/Adjugate

Parameters

method : string, optional

Method to use to find the cofactors, can be "bareiss", "berkowitz" or "lu".

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.adjugate()
Matrix([
[ 4, -2],
[-3,  1]])
```

See also:

cofactor_matrix (page 1229), *sympy.matrices.common.MatrixCommon.transpose* (page 1355)

charpoly(*x='lambda', simplify=<function _simplify>*)

Computes characteristic polynomial det(x*I - M) where I is the identity matrix.

A PurePoly is returned, so using different variables for x does not affect the comparison or the polynomials:

Parameters

x : string, optional

Name for the "lambda" variable, defaults to "lambda".

simplify : function, optional

Simplification function to use on the characteristic polynomial calculated. Defaults to `simplify`.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[1, 3], [2, 0]])
>>> M.charpoly()
PurePoly(lambda**2 - lambda - 6, lambda, domain='ZZ')
>>> M.charpoly(x) == M.charpoly(y)
True
>>> M.charpoly(x) == M.charpoly(y)
True
```

Specifying x is optional; a symbol named `lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> M.charpoly().as_expr()
lambda**2 - lambda - 6
```

And if x clashes with an existing symbol, underscores will be prepended to the name to make it unique:

```
>>> M = Matrix([[1, 2], [x, 0]])
>>> M.charpoly(x).as_expr()
_x**2 - _x - 2*x
```

Whether you pass a symbol or not, the generator can be obtained with the gen attribute since it may not be the same as the symbol that was passed:

```
>>> M.charpoly(x).gen
_x
>>> M.charpoly(x).gen == x
False
```

Notes

The Samuelson-Berkowitz algorithm is used to compute the characteristic polynomial efficiently and without any division operations. Thus the characteristic polynomial over any commutative ring without zero divisors can be computed.

If the determinant det(x*I - M) can be found out easily as in the case of an upper or a lower triangular matrix, then instead of Samuelson-Berkowitz algorithm, eigenvalues are computed and the characteristic polynomial with their help.

See also:

det (page 1230)

cofactor(*i*, *j*, *method='berkowitz'*)

Calculate the cofactor of an element.

Parameters

method : string, optional

Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.cofactor(0, 1)
-3
```

See also:

cofactor_matrix (page 1229), *minor* (page 1231), *minor_submatrix* (page 1231)

cofactor_matrix(*method='berkowitz'*)

Return a matrix containing the cofactor of each element.

Parameters

method : string, optional

Method to use to find the cofactors, can be “bareiss”, “berkowitz” or “lu”.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.cofactor_matrix()
Matrix([
[ 4, -3],
[-2,  1]])
```

See also:

cofactor (page 1229), *minor* (page 1231), *minor_submatrix* (page 1231)

det(*method='bareiss'*, *iszerofunc=None*)

Computes the determinant of a matrix if `M` is a concrete matrix object otherwise return an expressions `Determinant(M)` if `M` is a `MatrixSymbol` or other expression.

Parameters

method : string, optional

Specifies the algorithm used for computing the matrix determinant.

If the matrix is at most 3x3, a hard-coded formula is used and the specified method is ignored. Otherwise, it defaults to `'bareiss'`.

Also, if the matrix is an upper or a lower triangular matrix, determinant is computed by simple multiplication of diagonal elements, and the specified method is ignored.

If it is set to `'domain-ge'`, then Gaussian elimination method will be used via using DomainMatrix.

If it is set to `'bareiss'`, Bareiss' fraction-free algorithm will be used.

If it is set to `'berkowitz'`, Berkowitz' algorithm will be used.

Otherwise, if it is set to `'lu'`, LU decomposition will be used.

Note: For backward compatibility, legacy keys like "bareis" and "det_lu" can still be used to indicate the corresponding methods. And the keys are also case-insensitive for now. However, it is suggested to use the precise keys for specifying the method.

---

iszerofunc : FunctionType or None, optional

If it is set to `None`, it will be defaulted to `_iszero` if the method is set to `'bareiss'`, and `_is_zero_after_expand_mul` if the method is set to `'lu'`.

It can also accept any user-specified zero testing function, if it is formatted as a function which accepts a single symbolic argument and returns `True` if it is tested as zero and `False` if it tested as non-zero, and also `None` if it is undecidable.

Returns

det : Basic

Result of determinant.

Raises

ValueError

If unrecognized keys are given for `method` or `iszerofunc`.

NonSquareMatrixError

If attempted to calculate determinant from a non-square matrix.

Examples

```
>>> from sympy import Matrix, eye, det
>>> I3 = eye(3)
>>> det(I3)
1
>>> M = Matrix([[1, 2], [3, 4]])
>>> det(M)
-2
>>> det(M) == M.det()
True
>>> M.det(method="domain-ge")
-2
```

minor(*i*, *j*, *method='berkowitz'*)

Return the (i,j) minor of `M`. That is, return the determinant of the matrix obtained by deleting the $i^{th} row and j^{th} column from M$.

Parameters

i, j : int

The row and column to exclude to obtain the submatrix.

method : string, optional

Method to use to find the determinant of the submatrix, can be "bareiss", "berkowitz" or "lu".

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor(1, 1)
-12
```

See also:

minor_submatrix (page 1231), *cofactor* (page 1229), *det* (page 1230)

minor_submatrix(*i*, *j*)

Return the submatrix obtained by removing the $i^{th} row and j^{th} column from M$ (works with Pythonic negative indices).

Parameters

i, j : int

The row and column to exclude to obtain the submatrix.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.minor_submatrix(1, 1)
Matrix([
[1, 3],
[7, 9]])
```

See also:

minor (page 1231), *cofactor* (page 1229)

per()

Returns the permanent of a matrix. Unlike determinant, permanent is defined for both square and non-square matrices.

For an m x n matrix, with m less than or equal to n, it is given as the sum over the permutations s of size less than or equal to m on [1, 2, ... n] of the product from i = 1 to m of M[i, s[i]]. Taking the transpose will not affect the value of the permanent.

In the case of a square matrix, this is the same as the permutation definition of the determinant, but it does not take the sign of the permutation into account. Computing the permanent with this definition is quite inefficient, so here the Ryser formula is used.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M.per()
450
>>> M = Matrix([1, 5, 7])
>>> M.per()
13
```

References

[R566], [R567], [R568], [R569]

MatrixReductions Class Reference

class sympy.matrices.matrices.**MatrixReductions**

Provides basic matrix row/column operations. Should not be instantiated directly. See `reductions.py` for some of their implementations.

echelon_form(*iszerofunc=<function _iszero>*, *simplify=False*, *with_pivots=False*)

Returns a matrix row-equivalent to `M` that is in echelon form. Note that echelon form of a matrix is *not* unique, however, properties like the row space and the null space are preserved.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2], [3, 4]])
>>> M.echelon_form()
Matrix([
[1,  2],
[0, -2]])
```

elementary_col_op(*op='n->kn', col=None, k=None, col1=None, col2=None*)

Performs the elementary column operation op.

op may be one of

- "n->kn" (column n goes to k*n)
- "n<->m" (swap column n and column m)
- "n->n+km" (column n goes to column n + k*column m)

Parameters

op : string; the elementary row operation

col : the column to apply the column operation

k : the multiple to apply in the column operation

col1 : one column of a column swap

col2 : second column of a column swap or column "m" in the column operation

"n->n+km"

elementary_row_op(*op='n->kn', row=None, k=None, row1=None, row2=None*)

Performs the elementary row operation op.

op may be one of

- "n->kn" (row n goes to k*n)
- "n<->m" (swap row n and row m)
- "n->n+km" (row n goes to row n + k*row m)

Parameters

op : string; the elementary row operation

row : the row to apply the row operation

k : the multiple to apply in the row operation

row1 : one row of a row swap

row2 : second row of a row swap or row "m" in the row operation

"n->n+km"

property is_echelon

Returns $True$ if the matrix is in echelon form. That is, all rows of zeros are at the bottom, and below each leading non-zero in a row are exclusively zeros.

rank(*iszerofunc=<function _iszero>*, *simplify=False*)

Returns the rank of a matrix.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

rref(*iszerofunc=<function _iszero>*, *simplify=False*, *pivots=True*, *normalize_last=True*)

Return reduced row-echelon form of matrix and indices of pivot vars.

Parameters

iszerofunc : Function

A function used for detecting whether an element can act as a pivot. `lambda x: x.is_zero` is used by default.

simplify : Function

A function used to simplify elements when looking for a pivot. By default SymPy's `simplify` is used.

pivots : True or False

If `True`, a tuple containing the row-reduced matrix and a tuple of pivot columns is returned. If `False` just the row-reduced matrix is returned.

normalize_last : True or False

If `True`, no pivots are normalized to `1` until after all entries above and below each pivot are zeroed. This means the row reduction algorithm is fraction free until the very last step. If `False`, the naive row reduction procedure is used where each pivot is normalized to be `1` before row operations are used to zero above and below the pivot.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), (0, 1))
>>> rref_matrix, rref_pivots = m.rref()
>>> rref_matrix
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
>>> rref_pivots
(0, 1)
```

`iszerofunc` can correct rounding errors in matrices with float values. In the following example, calling `rref()` leads to floating point errors, incorrectly row reducing the matrix. `iszerofunc= lambda x: abs(x)<1e-9` sets sufficiently small numbers to zero, avoiding this error.

```
>>> m = Matrix([[0.9, -0.1, -0.2, 0], [-0.8, 0.9, -0.4, 0], [-0.1, -0.
↪8, 0.6, 0]])
>>> m.rref()
(Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0]]), (0, 1, 2))
>>> m.rref(iszerofunc=lambda x:abs(x)<1e-9)
(Matrix([
[1, 0, -0.301369863013699, 0],
[0, 1, -0.712328767123288, 0],
[0, 0,          0,          0]]), (0, 1))
```

Notes

The default value of `normalize_last=True` can provide significant speedup to row reduction, especially on matrices with symbols. However, if you depend on the form row reduction algorithm leaves entries of the matrix, set `noramlize_last=False`

MatrixSubspaces Class Reference

class sympy.matrices.matrices.**MatrixSubspaces**

Provides methods relating to the fundamental subspaces of a matrix. Should not be instantiated directly. See `subspaces.py` for their implementations.

columnspace(*simplify=False*)

Returns a list of vectors (Matrix objects) that span columnspace of `M`

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([
[ 1,  3, 0],
[-2, -6, 0],
[ 3,  9, 6]])
```

(continues on next page)

(continued from previous page)

```
>>> M.columnspace()
[Matrix([
[ 1],
[-2],
[ 3]]), Matrix([
[0],
[0],
[6]])]
```

See also:

nullspace (page 1236), *rowspace* (page 1237)

nullspace(*simplify=False*, *iszerofunc=<function _iszero>*)

Returns list of vectors (Matrix objects) that span nullspace of `M`

Examples

```
>>> from sympy import Matrix
>>> M = Matrix(3, 3, [1, 3, 0, -2, -6, 0, 3, 9, 6])
>>> M
Matrix([
[ 1,  3, 0],
[-2, -6, 0],
[ 3,  9, 6]])
>>> M.nullspace()
[Matrix([
[-3],
[ 1],
[ 0]])]
```

See also:

columnspace (page 1235), *rowspace* (page 1237)

classmethod orthogonalize(**vecs*, ***kwargs*)

Apply the Gram-Schmidt orthogonalization procedure to vectors supplied in `vecs`.

Parameters

vecs

vectors to be made orthogonal

normalize : bool

If `True`, return an orthonormal basis.

rankcheck : bool

If `True`, the computation does not stop when encountering linearly dependent vectors.

If `False`, it will raise `ValueError` when any zero or linearly dependent vectors are found.

Returns

list