Sets

↑ Back to top

Basic Sets

Set

class sympy.sets.sets.**Set**(*args) [source]

The base class for any kind of set.

Explanation

This is not meant to be used directly as a container of items. It does not behave like the builtin `set`; see `FiniteSet` for that.

Real intervals are represented by the `Interval` class and unions of sets by the `Union` class. The empty set is represented by the `EmptySet` class and available as a singleton as `S.EmptySet`.

property boundary

The boundary or frontier of a set.

Explanation

A point x is on the boundary of a set S if

1. x is in the closure of S. I.e. Every neighborhood of x contains a point in S.
2. x is not in the interior of S. I.e. There does not exist an open set centered on x contained entirely within S.

There are the points on the outer rim of S. If S is open then these points need not actually be contained within S.

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

property closure

Property method which returns the closure of a set. The closure is defined as the union of the set itself and its boundary. ↑ Back to top

Examples

```
>>> from sympy import S, Interval
>>> S.Reals.closure
Reals
>>> Interval(0, 1).closure
Interval(0, 1)
```

complement(universe) [source]

The complement of 'self' w.r.t the given universe.

Examples

```
>>> from sympy import Interval, S
>>> Interval(0, 1).complement(S.Reals)
Union(Interval.open(-oo, 0), Interval.open(1, oo))
```

```
>>> Interval(0, 1).complement(S.UniversalSet)
Complement(UniversalSet, Interval(0, 1))
```

contains(other) [source]

Returns a SymPy value indicating whether `other` is contained in `self`: `true` if it is, `false` if it is not, else an unevaluated `Contains` expression (or, as in the case of ConditionSet and a union of FiniteSet/Intervals, an expression indicating the conditions for containment).

Examples

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

```
>>> Interval(0, 1).contains(0.5)
True
```

As a shortcut it is possible to use the `in` operator, but that will raise an error unless an affirmative true or false is not obtained.

```
>>> Interval(0, 1).contains(x)
(0 <= x) & (x <= 1)
>>> x in Interval(0, 1)
Traceback (most recent call last):
```

```
...
TypeError: did not evaluate to a bool: None
```

The result of 'in' is a bool, not a Sy ↑ Back to top

```
>>> 1 in Interval(0, 2)
True
>>> _ is S.true
False
```

property inf

The infimum of `self`.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

property interior

Property method which returns the interior of a set. The interior of a set S consists all points of S that do not belong to the boundary of S.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).interior
Interval.open(0, 1)
>>> Interval(0, 1).boundary.interior
EmptySet
```

intersect(other) [source]

Returns the intersection of 'self' and 'other'.

Examples

```
>>> from sympy import Interval
```

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

```
>>> from sympy import imageset, Lambda, symbols, S
>>> n, m = symbols('n m')
```

```
>>> a = imageset(Lambda(n, 2*n), S.Integers)
>>> a.intersect(imageset(Lambda(m, 2*m + 1), S.Integers))
EmptySet
```

↑ Back to top

intersection(other) [source]

Alias for `intersect()`

property is_closed

A property method to check whether a set is closed.

Explanation

A set is closed if its complement is an open set. The closedness of a subset of the reals is determined with respect to R and its standard topology.

Examples

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

is_disjoint(other) [source]

Returns True if `self` and `other` are disjoint.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

References

[R744]

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

property is_open

Property method to check whether a set is open.

Explanation

A set is open if and only if it has an empty intersection with its boundary. In particular, a subset A of the reals is open if and only if each one of its points is contained in an open interval that is a subset of A.

Examples

```
>>> from sympy import S
>>> S.Reals.is_open
True
>>> S.Rationals.is_open
False
```

↑ Back to top

is_proper_subset(other) [source]

Returns True if `self` is a proper subset of `other`.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

is_proper_superset(other) [source]

Returns True if `self` is a proper superset of `other`.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

is_subset(other) [source]

Returns True if `self` is a subset of `other`.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(other) [source]

Returns True if `self` is a superset of `other`.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superse 1, left_open=True))
True
```

↑ Back to top

isdisjoint(other) [source]

Alias for `is_disjoint()`

issubset(other) [source]

Alias for `is_subset()`

issuperset(other) [source]

Alias for `is_superset()`

property kind

The kind of a Set

Explanation

Any `Set` will have kind `SetKind` which is parametrised by the kind of the elements of the set. For example most sets are sets of numbers and will have kind `SetKind(NumberKind)`. If elements of sets are different in kind than their kind will `SetKind(UndefinedKind)`. See `sympy.core.kind.Kind` for an explanation of the kind system.

Examples

```
>>> from sympy import Interval, Matrix, FiniteSet, EmptySet, ProductSet, PowerSet
```

```
>>> FiniteSet(Matrix([1, 2])).kind
SetKind(MatrixKind(NumberKind))
```

```
>>> Interval(1, 2).kind
SetKind(NumberKind)
```

```
>>> EmptySet.kind
SetKind()
```

A `sympy.sets.powerset.PowerSet` is a set of sets:

```
>>> PowerSet({1, 2, 3}).kind
SetKind(SetKind(NumberKind))
```

A `ProductSet` represents the set of tuples of elements of other sets. Its kind is `sympy.core.containers.TupleKind` parametrised by the kinds of the elements of those sets:

↑ Back to top

```
>>> p = ProductSet(FiniteSet(1, 2), FiniteSet(3, 4))
>>> list(p)
[(1, 3), (2, 3), (1, 4), (2, 4)]
>>> p.kind
SetKind(TupleKind(NumberKind, NumberKind))
```

When all elements of the set do not have same kind, the kind will be returned as `SetKind(UndefinedKind)`:

```
>>> FiniteSet(0, Matrix([1, 2])).kind
SetKind(UndefinedKind)
```

The kind of the elements of a set are given by the `element_kind` attribute of `SetKind`:

```
>>> Interval(1, 2).kind.element_kind
NumberKind
```

See also

`NumberKind`, `sympy.core.kind.UndefinedKind`, `sympy.core.containers.TupleKind`, `MatrixKind`, `sympy.matrices.expressions.sets.MatrixSet`, `sympy.sets.conditionset.ConditionSet`, `Rationals`, `Naturals`, `Integers`, `sympy.sets.fancysets.ImageSet`, `sympy.sets.fancysets.Range`, `sympy.sets.fancysets.ComplexRegion`, `sympy.sets.powerset.PowerSet`, `sympy.sets.sets.ProductSet`, `sympy.sets.sets.Interval`, `sympy.sets.sets.Union`, `sympy.sets.sets.Intersection`, `sympy.sets.sets.Complement`, `sympy.sets.sets.EmptySet`, `sympy.sets.sets.UniversalSet`, `sympy.sets.sets.FiniteSet`, `sympy.sets.sets.SymmetricDifference`, `sympy.sets.sets.DisjointUnion`

property measure

The (Lebesgue) measure of `self`.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset() [source]

Find the Power set of `self`.

Examples

↑ Back to top

```
>>> from sympy import EmptySet, FiniteSet, Interval
```

A power set of an empty set:

```
>>> A = EmptySet
>>> A.powerset()
{EmptySet}
```

A power set of a finite set:

```
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet)
True
```

A power set of an interval:

```
>>> Interval(1, 2).powerset()
PowerSet(Interval(1, 2))
```

References

[R745]

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

property sup

The supremum of `self`.

Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

symmetric_difference(other) [source]

Returns symmetric difference of `self` and `other`.

Examples

```
>>> from sympy import Interval, S
>>> Interval(1, 3).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(3, oo))
>>> Interval(1, 10).symmetric_difference(S.Reals)
Union(Interval.open(-oo, 1), Interval.open(10, oo))
```

↑ Back to top

```
>>> from sympy import S, EmptySet
>>> S.Reals.symmetric_difference(EmptySet)
Reals
```

References

[R746]

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

union(other) [source]

Returns the union of `self` and `other`.

Examples

As a shortcut it is possible to use the `+` operator:

```
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(0, 1) + Interval(2, 3)
Union(Interval(0, 1), Interval(2, 3))
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
Union({3}, Interval.Lopen(1, 2))
```

Similarly it is possible to use the `-` operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
Interval.Lopen(1, 2)
>>> Interval(1, 3) - FiniteSet(2)
Union(Interval.Ropen(1, 2), Interval.Lopen(2, 3))
```

sympy.sets.sets.**imageset(*args)** [source]

Return an image of the set under transformation `f`.

Explanation

If this function cannot compute the image, it returns an unevaluated ImageSet object.

$$\{f(x) \mid x \in \text{self}\}$$

Examples

```
>>> from sympy import S, Interval, imageset, sin, Lambda
>>> from sympy.abc import x
```

↑ Back to top

```
>>> imageset(x, 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(lambda x: 2*x, Interval(0, 2))
Interval(0, 4)
```

```
>>> imageset(Lambda(x, sin(x)), Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
```

```
>>> imageset(sin, Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), Interval(-2, 1))
>>> imageset(lambda y: x + y, Interval(-2, 1))
ImageSet(Lambda(y, x + y), Interval(-2, 1))
```

Expressions applied to the set of Integers are simplified to show as few negatives as possible and linear expressions are converted to a canonical form. If this is not desirable then the unevaluated ImageSet should be used.

```
>>> imageset(x, -2*x + 5, S.Integers)
ImageSet(Lambda(x, 2*x + 1), Integers)
```

> **See also**
>
> `sympy.sets.fancysets.ImageSet`

Elementary Sets

Interval

class sympy.sets.sets.**Interval**(*start*, *end*, *left_open=False*, *right_open=False*) [source]

Represents a real interval as a Set.

Usage:

Returns an interval with end points `start` and `end`.

For `left_open=True` (default `left_open` is `False`) the interval will be open on the left. Similarly, for `right_open=True` the interval will be open on the right.

Examples

```
>>> from sympy import Symbol, Interval
>>> Interval(0, 1)
Interval(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Ropen(0, 1)
Interval.Ropen(0, 1)
>>> Interval.Lopen(0, 1)
Interval.Lopen(0, 1)
>>> Interval.open(0, 1)
Interval.open(0, 1)
```

↑ Back to top

```
>>> a = Symbol('a', real=True)
>>> Interval(0, a)
Interval(0, a)
```

Notes

- Only real end points are supported
- `Interval(a, b)` with $a > b$ will return the empty set
- Use the `evalf()` method to turn an Interval into an mpmath `mpi` interval instance

References

[R747]

https://en.wikipedia.org/wiki/Interval_%28mathematics%29

classmethod Lopen(a, b) [source]

Return an interval not including the left boundary.

classmethod Ropen(a, b) [source]

Return an interval not including the right boundary.

as_relational(x) [source]

Rewrite an interval in terms of inequalities and logic operators.

property end

The right end point of the interval.

This property takes the same value as the `sup` property.

Examples

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

↑ Back to top

property is_left_unbounded

Return `True` if the left endpoint is negative infinity.

property is_right_unbounded

Return `True` if the right endpoint is positive infinity.

property left_open

True if interval is left-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False
```

classmethod open(a, b) [source]

Return an interval including neither boundary.

property right_open

True if interval is right-open.

Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, right_open=True).right_open
True
>>> Interval(0, 1, right_open=False).right_open
False
```

property start

The left end point of the interval.

This property takes the same value as the `inf` property.

Examples

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

```
0
```

FiniteSet

↑ Back to top

class sympy.sets.sets.**FiniteSet**(*args, **kwargs) [source]

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]

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

as_relational(symbol) [source]

Rewrite a FiniteSet in terms of equalities and logic operators.

Compound Sets

Union

class sympy.sets.sets.**Union**(*args, **kwargs) [source]

Represents a union of sets as a `Set`.

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:

↑ Back to top

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

See also

`Intersection`

References

[R749]

https://en.wikipedia.org/wiki/Union_%28set_theory%29

as_relational(symbol) [source]

Rewrite a Union in terms of equalities and logic operators.

Intersection

class sympy.sets.sets.Intersection(*args, **kwargs) [source]

Represents an intersection of sets as a `Set`.

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`

References

[R750]

https://en.wikipedia.org/wiki/Intersection_%28set_theory%29

`as_relational(symbol)` [source]

Rewrite an Intersection in terms of equalities and logic operators

↑ Back to top

ProductSet

`class sympy.sets.sets.ProductSet(*sets, **assumptions)` [source]

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]

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

property is_iterable

↑ Back to top

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)** [source]

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`, `Union`

References

[R752]

http://mathworld.wolfram.com/ComplementSet.html

as_relational(symbol) [source]

Rewrite a complement in terms of equalities and logic operators

static reduce(A, B) [source]

Simplify a `Complement`.

SymmetricDifference

class sympy.sets.sets.SymmetricDi↑ Back to top b, evaluate=True) [source]

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`, `Union`

References

[R753]

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

as_relational(symbol) [source]

Rewrite a symmetric_difference in terms of equalities and logic operators

DisjointUnion

class sympy.sets.sets.DisjointUnion(*sets) [source]

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}))
>>> 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

↑ Back to top

EmptySet

class sympy.sets.sets.**EmptySet** [source]

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`

References

[R754]

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

UniversalSet

class sympy.sets.sets.**UniversalSet** [source]

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`

↑ Back to top

References

[R755]

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

Special Sets

Rationals

class sympy.sets.fancysets.**Rationals** [source]

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** [source]

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`
: non-negative integers (i.e. includ

`Integers`
: also includes negative integers

Naturals0

class sympy.sets.fancysets.**Naturals0** [source]

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

See also

`Naturals`
: positive integers; does not include 0

`Integers`
: also includes the negative integers

Integers

class sympy.sets.fancysets.**Integers** [source]

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`
non-negative integers

`Integers`
positive and negative integers and zero

↑ Back to top

Reals

class sympy.sets.fancysets.**Reals** [source]

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`

Complexes

class sympy.sets.fancysets.**Complexes** [source]

The `Set` of all complex numbers

Examples

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

See also

`Reals`, `ComplexRegion`

↑ Back to top

ImageSet

class sympy.sets.fancysets.**ImageSet**(flambda, *sets) [source]

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
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}
```

↑ Back to top

See also

`sympy.sets.sets.imageset`

Range

class sympy.sets.fancysets.**Range**(*args) [source]

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]
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 ↑ Back to top t 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) [source]

Rewrite a Range in terms of equalities and logic operators.

property reversed

Return an equivalent Range in the opposite order.

Examples

↑ Back to top

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

ComplexRegion

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

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`.

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

- 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(In
```

- 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
```

↑ Back to top

```
>>> 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`, `PolarComplexRegion`, `Complexes`

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))
```

↑ Back to top

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)** [source]

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
```

property **sets**

Return raw input sets to the self.

Examples

↑ Back to top

```
>>> 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), Inter
```

class sympy.sets.fancysets.**CartesianComplexRegion(sets)** [source]

Set representing a square region of the complex plane.

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

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`, `PolarComplexRegion`, `Complexes`

class sympy.sets.fancysets.**PolarComplexRegion(sets)** [source]

Set representing a polar region of the complex plane.

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

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
```

↑ Back to top

See also

`ComplexRegion`, `CartesianComplexRegion`, `Complexes`

`sympy.sets.fancysets.normalize_theta_set(theta)` [source]

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)` [source]

A symbolic object representing a power set.

↑ Back to top

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]

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

[R757]

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

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) [source]

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
True
```

↑ Back to top

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:

↑ Back to top

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

Relations on sets

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

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]

https://en.wikipedia.org/wiki/Element_%28mathematics%29

SetKind

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

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`. 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
```

↑ Back to top

See also

`sympy.core.kind.NumberKind`, `sympy.matrices.common.MatrixKind`, `sympy.core.containers.TupleKind`