Algorithms (sympy.codegen.algorithms)

sympy.codegen.algorithms.**newtons_method**(*expr*, *wrt*, *atol=1e-12*, *delta=None*, *debug=False*, *itermax=None*, *counter=None*)

Generates an AST for Newton-Raphson method (a root-finding algorithm).

Parameters

expr : expression

wrt : Symbol

With respect to, i.e. what is the variable.

atol : number or expr

Absolute tolerance (stopping criterion)

delta : Symbol

Will be a `Dummy` if `None`.

debug : bool

Whether to print convergence information during iterations

itermax : number or expr

Maximum number of iterations.

counter : Symbol

Will be a `Dummy` if `None`.

Explanation

Returns an abstract syntax tree (AST) based on `sympy.codegen.ast` for Netwon's method of root-finding.

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import Assignment
>>> from sympy.codegen.algorithms import newtons_method
>>> x, dx, atol = symbols('x dx atol')
>>> expr = cos(x) - x**3
>>> algo = newtons_method(expr, x, atol, dx)
>>> algo.has(Assignment(dx, -expr/expr.diff(x)))
True
```

References

[R38]

sympy.codegen.algorithms.**newtons_method_function**(*expr*, *wrt*, *params=None*, *func_name='newton'*, *attrs=()*, ***, *delta=None*, ***kwargs*)

Generates an AST for a function implementing the Newton-Raphson method.

Parameters

expr : expression

wrt : Symbol

With respect to, i.e. what is the variable

params : iterable of symbols

Symbols appearing in expr that are taken as constants during the iterations (these will be accepted as parameters to the generated function).

func_name : str

Name of the generated function.

attrs : Tuple

Attribute instances passed as `attrs` to `FunctionDefinition`.

****kwargs :**

Keyword arguments passed to *sympy.codegen.algorithms.newtons_method()* (page 1157).

Examples

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.algorithms import newtons_method_function
>>> from sympy.codegen.pyutils import render_as_module
>>> x = symbols('x')
>>> expr = cos(x) - x**3
>>> func = newtons_method_function(expr, x)
>>> py_mod = render_as_module(func)  # source code as string
>>> namespace = {}
>>> exec(py_mod, namespace, namespace)
>>> res = eval('newton(0.5)', namespace)
>>> abs(res - 0.865474033102) < 1e-12
True
```

See also:

sympy.codegen.algorithms.newtons_method (page 1157)

Python utilities (sympy.codegen.pyutils)

sympy.codegen.pyutils.**render_as_module**(*content*, *standard='python3'*)

Renders Python code as a module (with the required imports).

Parameters

standard :

See the parameter `standard` in *sympy.printing.pycode.pycode()* (page 2176)

C utilities (sympy.codegen.cutils)

sympy.codegen.cutils.**render_as_source_file**(*content*, *Printer=<class 'sympy.printing.c.C99CodePrinter'>*, *settings=None*)

Renders a C source file (with required #include statements)

Fortran utilities (sympy.codegen.futils)

sympy.codegen.futils.**render_as_module**(*definitions*, *name*, *declarations=()*, *printer_settings=None*)

Creates a `Module` instance and renders it as a string.

This generates Fortran source code for a module with the correct `use` statements.

Parameters

definitions : iterable

Passed to *sympy.codegen.fnodes.Module* (page 1151).

name : str

Passed to *sympy.codegen.fnodes.Module* (page 1151).

declarations : iterable

Passed to *sympy.codegen.fnodes.Module* (page 1151). It will be extended with use statements, 'implicit none' and public list generated from `definitions`.

printer_settings : dict

Passed to `FCodePrinter` (default: `{'standard': 2003, 'source_format': 'free'}`).

5.8.3 Logic

Contents

Logic

Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

Forming logical expressions

You can build Boolean expressions with the standard python operators & (*And* (page 1166)), | (*Or* (page 1167)), ~ (*Not* (page 1167)):

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
y | (x & y)
>>> x | y
x | y
>>> ~x
~x
```

You can also form implications with >> and <<:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in SymPy, Boolean expressions inherit from *Basic* (page 927):

```
>>> (y & x).subs({x: True, y: True})
True
>>> (x | y).atoms()
{x, y}
```

The logic module also includes the following functions to derive boolean expressions from their truth tables:

`sympy.logic.boolalg.`**`SOPform`**(*variables*, *minterms*, *dontcares=None*)

The SOPform function uses simplified_pairs and a redundant group- eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest sum-of-products form.

The variables must be given as the first argument.

Return a logical *Or* (page 1167) function (i.e., the "sum of products" or "SOP" form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

Examples

```
>>> from sympy.logic import SOPform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
...             [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

The terms can also be represented as integers:

```
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(y & z) | (~w & ~x)
```

They can also be specified using dicts, which does not have to be fully specified:

```
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> SOPform([w, x, y, z], minterms)
(x & ~w) | (y & z & ~x)
```

Or a combination:

```
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w : 0, x : 0, y: 0}, 5]
>>> SOPform([w, x, y, z], minterms, dontcares)
(w & y & z) | (~w & ~y) | (x & z & ~w)
```

References

[R557]

sympy.logic.boolalg.**POSform**(*variables*, *minterms*, *dontcares=None*)

The POSform function uses simplified_pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest product-of-sums form.

The variables must be given as the first argument.

Return a logical *And* (page 1166) function (i.e., the "product of sums" or "POS" form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

Examples

```
>>> from sympy.logic import POSform
>>> from sympy import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1],
...             [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

The terms can also be represented as integers:

```
>>> minterms = [1, 3, 7, 11, 15]
>>> dontcares = [0, 2, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
z & (y | ~w)
```

They can also be specified using dicts, which does not have to be fully specified:

```
>>> minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
>>> POSform([w, x, y, z], minterms)
(x | y) & (x | z) & (~w | ~x)
```

Or a combination:

```
>>> minterms = [4, 7, 11, [1, 1, 1, 1]]
>>> dontcares = [{w : 0, x : 0, y: 0}, 5]
>>> POSform([w, x, y, z], minterms, dontcares)
(w | x) & (y | ~w) & (z | ~y)
```

References

[R558]

sympy.logic.boolalg.**ANFform**(*variables*, *truthvalues*)

The ANFform function converts the list of truth values to Algebraic Normal Form (ANF).

The variables must be given as the first argument.

Return True, False, logical *And* (page 1166) funciton (i.e., the "Zhegalkin monomial") or logical *Xor* (page 1168) function (i.e., the "Zhegalkin polynomial"). When True and False are represented by 1 and 0, respectively, then *And* (page 1166) is multiplication and *Xor* (page 1168) is addition.

Formally a "Zhegalkin monomial" is the product (logical And) of a finite set of distinct variables, including the empty set whose product is denoted 1 (True). A "Zhegalkin polynomial" is the sum (logical Xor) of a set of Zhegalkin monomials, with the empty set denoted by 0 (False).

Parameters

variables : list of variables

truthvalues : list of 1's and 0's (result column of truth table)

Examples

```
>>> from sympy.logic.boolalg import ANFform
>>> from sympy.abc import x, y
>>> ANFform([x], [1, 0])
x ^ True
>>> ANFform([x, y], [0, 1, 1, 1])
x ^ y ^ (x & y)
```

References

[R559]

Boolean functions

class sympy.logic.boolalg.**Boolean**(**args*)

A Boolean object is an object for which logic operations make sense.

as_set()

Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

equals(*other*)

Returns True if the given formulas have the same truth table. For two formulas to be equal they must have the same literals.

Examples

```
>>> from sympy.abc import A, B, C
>>> from sympy import And, Or, Not
>>> (A >> B).equals(~B >> ~A)
True
>>> Not(And(A, B, C)).equals(And(Not(A), Not(B), Not(C)))
False
>>> Not(And(A, Not(A))).equals(Or(B, Not(B)))
False
```

class sympy.logic.boolalg.**BooleanTrue**

SymPy version of `True`, a singleton that can be accessed via `S.true`.

This is the SymPy version of `True`, for use in the logic module. The primary advantage of using `true` instead of `True` is that shorthand Boolean operations like ~ and >> will work as expected on this class, whereas with True they act bitwise on 1. Functions in the logic module will return this class when they evaluate to true.

Notes

There is liable to be some confusion as to when `True` should be used and when `S.true` should be used in various contexts throughout SymPy. An important thing to remember is that `sympify(True)` returns `S.true`. This means that for the most part, you can just use `True` and it will automatically be converted to `S.true` when necessary, similar to how you can generally use 1 instead of `S.One`.

The rule of thumb is:

"If the boolean in question can be replaced by an arbitrary symbolic `Boolean`, like `Or(x, y)` or `x > 1`, use `S.true`. Otherwise, use `True`"

In other words, use `S.true` only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the `.args` of any expression, then it must necessarily be `S.true` instead of `True`, as elements of `.args` must be `Basic`. On the other hand, `==` is not a symbolic operation in SymPy, since it always returns `True` or `False`, and does so in terms of structural equality rather than mathematical, so it should return `True`. The assumptions system should use `True` and `False`. Aside from not satisfying the above rule of thumb, the assumptions system uses a three-valued logic (`True`, `False`, `None`), whereas `S.true` and `S.false` represent a two-valued logic. When in doubt, use `True`.

"`S.true == True is True`."

While "`S.true is True`" is `False`, "`S.true == True`" is `True`, so if there is any doubt over whether a function or expression will return `S.true` or `True`, just use `==` instead of `is` to do the comparison, and it will work in either case. Finally, for boolean flags, it's better to just use `if x` instead of `if x is True`. To quote PEP 8:

Do not compare boolean values to `True` or `False` using `==`.

- Yes: `if greeting:`
- No: `if greeting == True:`
- Worse: `if greeting is True:`

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(True)
True
>>> _ is True, _ is true
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for true but a bitwise result for True

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

Python operators give a boolean result for true but a bitwise result for True

```
>>> ~true, ~True
(False, -2)
>>> true >> true, True >> True
(True, 0)
```

See also:

sympy.logic.boolalg.BooleanFalse (page 1165)

as_set()

Rewrite logic operators and relationals in terms of real sets.

Examples

```
>>> from sympy import true
>>> true.as_set()
UniversalSet
```

class sympy.logic.boolalg.**BooleanFalse**

SymPy version of `False`, a singleton that can be accessed via `S.false`.

This is the SymPy version of `False`, for use in the logic module. The primary advantage of using `false` instead of `False` is that shorthand Boolean operations like ~ and >> will work as expected on this class, whereas with `False` they act bitwise on 0. Functions in the logic module will return this class when they evaluate to false.

Notes

See the notes section in *sympy.logic.boolalg.BooleanTrue* (page 1163)

Examples

```
>>> from sympy import sympify, true, false, Or
>>> sympify(False)
False
>>> _ is False, _ is false
(False, True)
```

```
>>> Or(true, false)
True
>>> _ is true
True
```

Python operators give a boolean result for false but a bitwise result for False

```
>>> ~false, ~False
(True, -1)
>>> false >> false, False >> False
(True, 0)
```

See also:

sympy.logic.boolalg.BooleanTrue (page 1163)

as_set()

Rewrite logic operators and relationals in terms of real sets.

Examples

```
>>> from sympy import false
>>> false.as_set()
EmptySet
```

class sympy.logic.boolalg.**And**(**args*)

Logical AND function.

It evaluates its arguments in order, returning false immediately when an argument is false and true if they are all true.

Examples

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

Notes

The & operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, `And(a, b)` and `a & b` will produce different results if `a` and `b` are integers.

```
>>> And(x, y).subs(x, 1)
y
```

class sympy.logic.boolalg.**Or**(**args*)

Logical OR function

It evaluates its arguments in order, returning true immediately when an argument is true, and false if they are all false.

Examples

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

Notes

The `|` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, `Or(a, b)` and `a | b` will return different things if `a` and `b` are integers.

```
>>> Or(x, y).subs(x, 0)
y
```

class sympy.logic.boolalg.**Not**(*arg*)

Logical Not function (negation)

Returns `true` if the statement is `false` or `False`. Returns `false` if the statement is `true` or `True`.

Examples

```
>>> from sympy import Not, And, Or
>>> from sympy.abc import x, A, B
>>> Not(True)
False
>>> Not(False)
True
>>> Not(And(True, False))
True
>>> Not(Or(True, False))
False
>>> Not(And(And(True, x), Or(x, False)))
```

(continues on next page)

(continued from previous page)

```
~x
>>> ~x
~x
>>> Not(And(Or(A, B), Or(~A, ~B)))
~((A | B) & (~A | ~B))
```

Notes

- The ~ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise not. In particular, `~a` and `Not(a)` will be different if `a` is an integer. Furthermore, since bools in Python subclass from `int`, `~True` is the same as `~1` which is `-2`, which has a boolean value of True. To avoid this issue, use the SymPy boolean types `true` and `false`.

```
>>> from sympy import true
>>> ~True
-2
>>> ~true
False
```

class sympy.logic.boolalg.**Xor**(**args*)

Logical XOR (exclusive OR) function.

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

Examples

```
>>> from sympy.logic.boolalg import Xor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xor(True, False)
True
>>> Xor(True, True)
False
>>> Xor(True, False, True, True, False)
True
>>> Xor(True, False, True, False)
False
>>> x ^ y
x ^ y
```

Notes

The ^ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, `a ^ b` and `Xor(a, b)` will be different if `a` and `b` are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

class sympy.logic.boolalg.**Nand**(**args*)

Logical NAND function.

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

Examples

```
>>> from sympy.logic.boolalg import Nand
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
True
>>> Nand(True, True)
False
>>> Nand(x, y)
~(x & y)
```


class sympy.logic.boolalg.**Nor**(**args*)

Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

Examples

```
>>> from sympy.logic.boolalg import Nor
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```
>>> Nor(True, False)
False
>>> Nor(True, True)
False
>>> Nor(False, True)
False
>>> Nor(False, False)
True
```

(continues on next page)

(continued from previous page)

```
>>> Nor(x, y)
~(x | y)
```

class sympy.logic.boolalg.**Xnor**(**args*)

Logical XNOR function.

Returns False if an odd number of the arguments are True and the rest are False.

Returns True if an even number of the arguments are True and the rest are False.

Examples

```
>>> from sympy.logic.boolalg import Xnor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xnor(True, False)
False
>>> Xnor(True, True)
True
>>> Xnor(True, False, True, True, False)
False
>>> Xnor(True, False, True, False)
True
```

class sympy.logic.boolalg.**Implies**(**args*)

Logical implication.

A implies B is equivalent to if A then B. Mathematically, it is written as $A \Rightarrow B$ and is equivalent to $\neg A \vee B$ or `~A | B`.

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

Examples

```
>>> from sympy.logic.boolalg import Implies
>>> from sympy import symbols
>>> x, y = symbols('x y')
```

```
>>> Implies(True, False)
False
>>> Implies(False, False)
True
>>> Implies(True, True)
True
>>> Implies(False, True)
True
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

Notes

The >> and << operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a, b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python considers `True` and `False` to be integers, `True >> True` will be the same as `1 >> 1`, i.e., 0, which has a truth value of False. To avoid this issue, use the SymPy objects `true` and `false`.

```
>>> from sympy import true, false
>>> True >> False
1
>>> true >> false
False
```

class sympy.logic.boolalg.**Equivalent**(**args*)

Equivalence relation.

`Equivalent(A, B)` is True iff A and B are both True or both False.

Returns True if all of the arguments are logically equivalent. Returns False otherwise.

For two arguments, this is equivalent to *Xnor* (page 1170).

Examples

```
>>> from sympy.logic.boolalg import Equivalent, And
>>> from sympy.abc import x
>>> Equivalent(False, False, False)
True
>>> Equivalent(True, False, False)
False
>>> Equivalent(x, And(x, True))
True
```

class sympy.logic.boolalg.**ITE**(**args*)

If-then-else clause.

`ITE(A, B, C)` evaluates and returns the result of B if A is true else it returns the result of C. All args must be Booleans.

From a logic gate perspective, ITE corresponds to a 2-to-1 multiplexer, where A is the select signal.

Examples

```
>>> from sympy.logic.boolalg import ITE, And, Xor, Or
>>> from sympy.abc import x, y, z
>>> ITE(True, False, True)
False
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
True
```

(continues on next page)

(continued from previous page)

```
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
>>> ITE(x, y, y)
y
```

Trying to use non-Boolean args will generate a TypeError:

```
>>> ITE(True, [], ())
Traceback (most recent call last):
...
TypeError: expecting bool, Boolean or ITE, not `[]`
```

class sympy.logic.boolalg.**Exclusive**(**args*)

True if only one or no argument is true.

`Exclusive(A, B, C)` is equivalent to `~(A & B) & ~(A & C) & ~(B & C)`.

For two arguments, this is equivalent to *Xor* (page 1168).

Examples

```
>>> from sympy.logic.boolalg import Exclusive
>>> Exclusive(False, False, False)
True
>>> Exclusive(False, True, False)
True
>>> Exclusive(False, True, True)
False
```

The following functions can be used to handle Algebraic, Conjunctive, Disjunctive, and Negated Normal forms:

sympy.logic.boolalg.**to_anf**(*expr*, *deep=True*)

Converts expr to Algebraic Normal Form (ANF).

ANF is a canonical normal form, which means that two equivalent formulas will convert to the same ANF.

A logical expression is in ANF if it has the form

$$1 \oplus a \oplus b \oplus ab \oplus abc$$

i.e. it can be:

- purely true,
- purely false,
- conjunction of variables,
- exclusive disjunction.

The exclusive disjunction can only contain true, variables or conjunction of variables. No negations are permitted.

If `deep` is `False`, arguments of the boolean expression are considered variables, i.e. only the top-level expression is converted to ANF.

Examples

```
>>> from sympy.logic.boolalg import And, Or, Not, Implies, Equivalent
>>> from sympy.logic.boolalg import to_anf
>>> from sympy.abc import A, B, C
>>> to_anf(Not(A))
A ^ True
>>> to_anf(And(Or(A, B), Not(C)))
A ^ B ^ (A & B) ^ (A & C) ^ (B & C) ^ (A & B & C)
>>> to_anf(Implies(Not(A), Equivalent(B, C)), deep=False)
True ^ ~A ^ (~A & (Equivalent(B, C)))
```

sympy.logic.boolalg.**to_cnf**(*expr*, *simplify=False*, *force=False*)

Convert a propositional logical sentence `expr` to conjunctive normal form: `((A | ~B | ...) & (B | C | ...) & ...)`. If `simplify` is `True`, `expr` is evaluated to its simplest CNF form using the Quine-McCluskey algorithm; this may take a long time. If there are more than 8 variables the *force*' flag must be set to `True` to simplify (default is `False`).

Examples

```
>>> from sympy.logic.boolalg import to_cnf
>>> from sympy.abc import A, B, D
>>> to_cnf(~(A | B) | D)
(D | ~A) & (D | ~B)
>>> to_cnf((A | B) & (A | ~A), True)
A | B
```

sympy.logic.boolalg.**to_dnf**(*expr*, *simplify=False*, *force=False*)

Convert a propositional logical sentence `expr` to disjunctive normal form: `((A & ~B & ...) | (B & C & ...) | ...)`. If `simplify` is `True`, `expr` is evaluated to its simplest DNF form using the Quine-McCluskey algorithm; this may take a long time. If there are more than 8 variables, the `force` flag must be set to `True` to simplify (default is `False`).

Examples

```
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy.abc import A, B, C
>>> to_dnf(B & (A | C))
(A & B) | (B & C)
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
A | C
```

sympy.logic.boolalg.**to_nnf**(*expr*, *simplify=True*)

Converts `expr` to Negation Normal Form (NNF).

A logical expression is in NNF if it contains only *And* (page 1166), *Or* (page 1167) and *Not* (page 1167), and *Not* (page 1167) is applied only to literals. If `simplify` is `True`, the result contains no redundant clauses.

Examples

```
>>> from sympy.abc import A, B, C, D
>>> from sympy.logic.boolalg import Not, Equivalent, to_nnf
>>> to_nnf(Not((~A & ~B) | (C & D)))
(A | B) & (~C | ~D)
>>> to_nnf(Equivalent(A >> B, B >> A))
(A | ~B | (A & ~B)) & (B | ~A | (B & ~A))
```

sympy.logic.boolalg.**is_anf**(*expr*)

Checks if `expr` is in Algebraic Normal Form (ANF).

A logical expression is in ANF if it has the form

$$1 \oplus a \oplus b \oplus ab \oplus abc$$

i.e. it is purely true, purely false, conjunction of variables or exclusive disjunction. The exclusive disjunction can only contain true, variables or conjunction of variables. No negations are permitted.

Examples

```
>>> from sympy.logic.boolalg import And, Not, Xor, true, is_anf
>>> from sympy.abc import A, B, C
>>> is_anf(true)
True
>>> is_anf(A)
True
>>> is_anf(And(A, B, C))
True
>>> is_anf(Xor(A, Not(B)))
False
```

sympy.logic.boolalg.**is_cnf**(*expr*)

Test whether or not an expression is in conjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_cnf
>>> from sympy.abc import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

sympy.logic.boolalg.**is_dnf**(*expr*)

Test whether or not an expression is in disjunctive normal form.

Examples

```
>>> from sympy.logic.boolalg import is_dnf
>>> from sympy.abc import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

sympy.logic.boolalg.**is_nnf**(*expr*, *simplified=True*)

Checks if `expr` is in Negation Normal Form (NNF).

A logical expression is in NNF if it contains only *And* (page 1166), *Or* (page 1167) and *Not* (page 1167), and *Not* (page 1167) is applied only to literals. If `simplified` is `True`, checks if result contains no redundant clauses.

Examples

```
>>> from sympy.abc import A, B, C
>>> from sympy.logic.boolalg import Not, is_nnf
>>> is_nnf(A & B | ~C)
True
>>> is_nnf((A | ~A) & (B | C))
False
>>> is_nnf((A | ~A) & (B | C), False)
True
>>> is_nnf(Not(A & B) | C)
False
>>> is_nnf((A >> B) & (B >> A))
False
```

sympy.logic.boolalg.**gateinputcount**(*expr*)

Return the total number of inputs for the logic gates realizing the Boolean expression.

Returns

int

Number of gate inputs

Note

Not all Boolean functions count as gate here, only those that are considered to be standard gates. These are: *And* (page 1166), *Or* (page 1167), *Xor* (page 1168), *Not* (page 1167), and *ITE* (page 1171) (multiplexer). *Nand* (page 1169), *Nor* (page 1169), and *Xnor* (page 1170) will be evaluated to `Not(And())` etc.

Examples

```
>>> from sympy.logic import And, Or, Nand, Not, gateinputcount
>>> from sympy.abc import x, y, z
>>> expr = And(x, y)
>>> gateinputcount(expr)
2
>>> gateinputcount(Or(expr, z))
4
```

Note that `Nand` is automatically evaluated to `Not(And())` so

```
>>> gateinputcount(Nand(x, y, z))
4
>>> gateinputcount(Not(And(x, y, z)))
4
```

Although this can be avoided by using `evaluate=False`

```
>>> gateinputcount(Nand(x, y, z, evaluate=False))
3
```

Also note that a comparison will count as a Boolean variable:

```
>>> gateinputcount(And(x > z, y >= 2))
2
```

As will a symbol: >>> gateinputcount(x) 0

Simplification and equivalence-testing

`sympy.logic.boolalg.`**`simplify_logic`**(*expr*, *form=None*, *deep=True*, *force=False*)

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an *Or* (page 1167) or *And* (page 1166) object in SymPy.

Parameters

expr : Boolean expression

form : string (`'cnf'` or `'dnf'`) or `None` (default).

If `'cnf'` or `'dnf'`, the simplest expression in the corresponding normal form is returned; if `None`, the answer is returned according to the form with fewest args (in CNF by default).

deep : bool (default `True`)

Indicates whether to recursively simplify any non-boolean functions contained within the input.

force : bool (default `False`)

As the simplifications require exponential time in the number of variables, there is by default a limit on expressions with 8 variables. When the expression has more than 8 variables only symbolical simplification (controlled by `deep`) is made. By setting `force` to `True`, this limit is removed. Be aware that this can lead to very long simplification times.

Examples

```
>>> from sympy.logic import simplify_logic
>>> from sympy.abc import x, y, z
>>> from sympy import S
>>> b = (~x & ~y & ~z) | ( ~x & ~y & z)
>>> simplify_logic(b)
~x & ~y
```

```
>>> S(b)
(z & ~x & ~y) | (~x & ~y & ~z)
>>> simplify_logic(_)
~x & ~y
```

SymPy's *simplify()* (page 661) function can also be used to simplify logic expressions to their simplest forms.

`sympy.logic.boolalg.`**`bool_map`**`(`*bool1*, *bool2*`)`

Return the simplified version of *bool1*, and the mapping of variables that makes the two expressions *bool1* and *bool2* represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned.

For example, `And(x, y)` is logically equivalent to `And(a, b)` for the mapping `{x: a, y: b}` or `{x: b, y: a}`. If no such mapping exists, return `False`.

Examples

```
>>> from sympy import SOPform, bool_map, Or, And, Not, Xor
>>> from sympy.abc import w, x, y, z, a, b, c, d
>>> function1 = SOPform([x, z, y],[[1, 0, 1], [0, 0, 1]])
>>> function2 = SOPform([a, b, c],[[1, 0, 1], [1, 0, 0]])
>>> bool_map(function1, function2)
(y & ~z, {y: a, z: b})
```

The results are not necessarily unique, but they are canonical. Here, `(w, z)` could be `(a, d)` or `(d, a)`:

```
>>> eq =  Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))
>>> bool_map(eq, eq2)
((x & y) | (w & ~y) | (z & ~y), {w: a, x: b, y: c, z: d})
>>> eq = And(Xor(a, b), c, And(c,d))
>>> bool_map(eq, eq.subs(c, x))
(c & d & (a | b) & (~a | ~b), {a: a, b: b, c: d, d: x})
```

Manipulating expressions

The following functions can be used to manipulate Boolean expressions:

sympy.logic.boolalg.**distribute_and_over_or**(*expr*)

Given a sentence `expr` consisting of conjunctions and disjunctions of literals, return an equivalent sentence in CNF.

Examples

```
>>> from sympy.logic.boolalg import distribute_and_over_or, And, Or, Not
>>> from sympy.abc import A, B, C
>>> distribute_and_over_or(Or(A, And(Not(B), Not(C))))
(A | ~B) & (A | ~C)
```

sympy.logic.boolalg.**distribute_or_over_and**(*expr*)

Given a sentence `expr` consisting of conjunctions and disjunctions of literals, return an equivalent sentence in DNF.

Note that the output is NOT simplified.

Examples

```
>>> from sympy.logic.boolalg import distribute_or_over_and, And, Or, Not
>>> from sympy.abc import A, B, C
>>> distribute_or_over_and(And(Or(Not(A), B), C))
(B & C) | (C & ~A)
```

sympy.logic.boolalg.**distribute_xor_over_and**(*expr*)

Given a sentence `expr` consisting of conjunction and exclusive disjunctions of literals, return an equivalent exclusive disjunction.

Note that the output is NOT simplified.

Examples

```
>>> from sympy.logic.boolalg import distribute_xor_over_and, And, Xor,␣
↪Not
>>> from sympy.abc import A, B, C
>>> distribute_xor_over_and(And(Xor(Not(A), B), C))
(B & C) ^ (C & ~A)
```

sympy.logic.boolalg.**eliminate_implications**(*expr*)

Change *Implies* (page 1170) and *Equivalent* (page 1171) into *And* (page 1166), *Or* (page 1167), and *Not* (page 1167). That is, return an expression that is equivalent to expr, but has only &, |, and ~ as logical operators.

Examples

```
>>> from sympy.logic.boolalg import Implies, Equivalent,
↪eliminate_implications
>>> from sympy.abc import A, B, C
>>> eliminate_implications(Implies(A, B))
B | ~A
>>> eliminate_implications(Equivalent(A, B))
(A | ~B) & (B | ~A)
>>> eliminate_implications(Equivalent(A, B, C))
(A | ~C) & (B | ~A) & (C | ~B)
```

Truth tables and related functions

It is possible to create a truth table for a Boolean function with

sympy.logic.boolalg.**truth_table**(*expr*, *variables*, *input=True*)

Return a generator of all possible configurations of the input variables, and the result of the boolean expression for those values.

Parameters

expr : Boolean expression

variables : list of variables

input : bool (default True)

Indicates whether to return the input combinations.

Examples

```
>>> from sympy.logic.boolalg import truth_table
>>> from sympy.abc import x,y
>>> table = truth_table(x >> y, [x, y])
>>> for t in table:
...     print('{0} -> {1}'.format(*t))
[0, 0] -> True
[0, 1] -> True
[1, 0] -> False
[1, 1] -> True
```

```
>>> table = truth_table(x | y, [x, y])
>>> list(table)
[([0, 0], False), ([0, 1], True), ([1, 0], True), ([1, 1], True)]
```

If `input` is `False`, `truth_table` returns only a list of truth values. In this case, the corresponding input values of variables can be deduced from the index of a given output.

```
>>> from sympy.utilities.iterables import ibin
>>> vars = [y, x]
>>> values = truth_table(x >> y, vars, input=False)
>>> values = list(values)
>>> values
[True, False, True, True]
```

```
>>> for i, value in enumerate(values):
...     print('{0} -> {1}'.format(list(zip(
...     vars, ibin(i, len(vars)))), value))
[(y, 0), (x, 0)] -> True
[(y, 0), (x, 1)] -> False
[(y, 1), (x, 0)] -> True
[(y, 1), (x, 1)] -> True
```

For mapping between integer representations of truth table positions, lists of zeros and ones and symbols, the following functions can be used:

`sympy.logic.boolalg.`**`integer_to_term`**(*n*, *bits=None*, *str=False*)

Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If bits is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0,..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for bits, e.g. `'all'`.

If the bit *string* is desired pass `str=True`.

Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
```

If all lists corresponding to 0 to 2**n - 1, pass a non-integer for bits:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use str=True:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1]  # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

sympy.logic.boolalg.**term_to_integer**(*term*)

Return an integer corresponding to the base-2 digits given by *term*.

Parameters

term : a string or list of ones and zeros

Examples

```
>>> from sympy.logic.boolalg import term_to_integer
>>> term_to_integer([1, 0, 0])
4
>>> term_to_integer('100')
4
```

sympy.logic.boolalg.**bool_maxterm**(*k*, *variables*)

Return the k-th maxterm.

Each maxterm is assigned an index based on the opposite conventional binary encoding used for minterms. The maxterm convention assigns the value 0 to the direct form and 1 to the complemented form.

Parameters

k : int or list of 1's and 0's (complementation pattern)

variables : list of variables

Examples

```
>>> from sympy.logic.boolalg import bool_maxterm
>>> from sympy.abc import x, y, z
>>> bool_maxterm([1, 0, 1], [x, y, z])
y | ~x | ~z
>>> bool_maxterm(6, [x, y, z])
z | ~x | ~y
```

References

[R560]

sympy.logic.boolalg.**bool_minterm**(*k*, *variables*)

Return the k-th minterm.

Minterms are numbered by a binary encoding of the complementation pattern of the variables. This convention assigns the value 1 to the direct form and 0 to the complemented form.

Parameters

k : int or list of 1's and 0's (complementation patter)

variables : list of variables

Examples

```
>>> from sympy.logic.boolalg import bool_minterm
>>> from sympy.abc import x, y, z
>>> bool_minterm([1, 0, 1], [x, y, z])
x & z & ~y
>>> bool_minterm(6, [x, y, z])
x & y & ~z
```

References

[R561]

sympy.logic.boolalg.**bool_monomial**(*k*, *variables*)

Return the k-th monomial.

Monomials are numbered by a binary encoding of the presence and absences of the variables. This convention assigns the value 1 to the presence of variable and 0 to the absence of variable.

Each boolean function can be uniquely represented by a Zhegalkin Polynomial (Algebraic Normal Form). The Zhegalkin Polynomial of the boolean function with n variables can contain up to 2^n monomials. We can enumarate all the monomials. Each monomial is fully specified by the presence or absence of each variable.

For example, boolean function with four variables `(a, b, c, d)` can contain up to $2^4 = 16$ monomials. The 13-th monomial is the product `a & b & d`, because 13 in binary is 1, 1, 0, 1.

Parameters

k : int or list of 1's and 0's

variables : list of variables

Examples

```
>>> from sympy.logic.boolalg import bool_monomial
>>> from sympy.abc import x, y, z
>>> bool_monomial([1, 0, 1], [x, y, z])
x & z
>>> bool_monomial(6, [x, y, z])
x & y
```

sympy.logic.boolalg.**anf_coeffs**(*truthvalues*)

Convert a list of truth values of some boolean expression to the list of coefficients of the polynomial mod 2 (exclusive disjunction) representing the boolean expression in ANF (i.e., the "Zhegalkin polynomial").

There are 2^n possible Zhegalkin monomials in n variables, since each monomial is fully specified by the presence or absence of each variable.

We can enumerate all the monomials. For example, boolean function with four variables `(a, b, c, d)` can contain up to $2^4 = 16$ monomials. The 13-th monomial is the product `a & b & d`, because 13 in binary is 1, 1, 0, 1.

A given monomial's presence or absence in a polynomial corresponds to that monomial's coefficient being 1 or 0 respectively.

Examples

```
>>> from sympy.logic.boolalg import anf_coeffs, bool_monomial, Xor
>>> from sympy.abc import a, b, c
>>> truthvalues = [0, 1, 1, 0, 0, 1, 0, 1]
>>> coeffs = anf_coeffs(truthvalues)
>>> coeffs
[0, 1, 1, 0, 0, 0, 1, 0]
>>> polynomial = Xor(*[
...     bool_monomial(k, [a, b, c])
...     for k, coeff in enumerate(coeffs) if coeff == 1
... ])
>>> polynomial
b ^ c ^ (a & b)
```

sympy.logic.boolalg.**to_int_repr**(*clauses*, *symbols*)

Takes clauses in CNF format and puts them into an integer representation.

Examples

```
>>> from sympy.logic.boolalg import to_int_repr
>>> from sympy.abc import x, y
>>> to_int_repr([x | y, y], [x, y]) == [{1, 2}, {2}]
True
```

Inference

This module implements some inference routines in propositional logic.

The function satisfiable will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence `True`.

For example, the expression `x & ~x` is not satisfiable, since there are no values for `x` that make this sentence `True`. On the other hand, `(x | y) & (x | ~y) & (~x | y)` is satisfiable with both `x` and `y` being `True`.

```
>>> from sympy.logic.inference import satisfiable
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> satisfiable(x & ~x)
False
>>> satisfiable((x | y) & (x | ~y) & (~x | y))
{x: True, y: True}
```

As you see, when a sentence is satisfiable, it returns a model that makes that sentence `True`. If it is not satisfiable it will return `False`.

sympy.logic.inference.**satisfiable**(*expr*, *algorithm=None*, *all_models=False*, *minimal=False*)

Check satisfiability of a propositional sentence. Returns a model when it succeeds. Returns {true: true} for trivially true expressions.

On setting all_models to True, if given expr is satisfiable then returns a generator of models. However, if expr is unsatisfiable then returns a generator containing the single element False.

Examples

```
>>> from sympy.abc import A, B
>>> from sympy.logic.inference import satisfiable
>>> satisfiable(A & ~B)
{A: True, B: False}
>>> satisfiable(A & ~A)
False
>>> satisfiable(True)
{True: True}
>>> next(satisfiable(A & ~A, all_models=True))
False
```

(continues on next page)

(continued from previous page)

```
>>> models = satisfiable((A >> B) & B, all_models=True)
>>> next(models)
{A: False, B: True}
>>> next(models)
{A: True, B: True}
>>> def use_models(models):
...     for model in models:
...         if model:
...             # Do something with the model.
...             print(model)
...         else:
...             # Given expr is unsatisfiable.
...             print("UNSAT")
>>> use_models(satisfiable(A >> ~A, all_models=True))
{A: False}
>>> use_models(satisfiable(A ^ A, all_models=True))
UNSAT
```

Sets

Basic Sets

Set

class sympy.sets.sets.**Set**(**args*)

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* (page 1197) for that.

Real intervals are represented by the *Interval* (page 1194) class and unions of sets by the *Union* (page 1197) class. The empty set is represented by the *EmptySet* (page 1202) 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.

Examples

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

complement(*universe*)

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*)

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 SymPy value

```
>>> 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*)

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
```

intersection(*other*)

Alias for *intersect()* (page 1187)

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*)

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]

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
```

is_proper_subset(*other*)

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*)

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*)

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*)

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_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(*other*)

Alias for *is_disjoint()* (page 1188)

issubset(*other*)

Alias for *is_subset()* (page 1189)

issuperset(*other*)

Alias for *is_superset()* (page 1190)

property kind

The kind of a Set

Explanation

Any *Set* (page 1185) will have kind *SetKind* (page 1217) 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* (page 1073) 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* (page 1213) is a set of sets:

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

A *ProductSet* (page 1199) represents the set of tuples of elements of other sets. Its kind is *sympy.core.containers.TupleKind* (page 1069) parametrised by the kinds of the elements of those sets:

```
>>> 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 (page 1074), *sympy.core.kind.UndefinedKind* (page 1074), *sympy.core.containers.TupleKind* (page 1069), *MatrixKind* (page 1360), *sympy.matrices.expressions.sets.MatrixSet* (page 1379), *sympy.sets.conditionset.ConditionSet* (page 1215), *Rationals* (page 1203), *Naturals* (page 1203), *Integers* (page 1204), *sympy.sets.fancysets.ImageSet* (page 1206), *sympy.sets.fancysets.Range* (page 1207), *sympy.sets.fancysets.ComplexRegion* (page 1209), *sympy.sets.powerset.PowerSet* (page 1213), *sympy.sets.sets.ProductSet* (page 1199), *sympy.sets.sets.Interval* (page 1194), *sympy.sets.sets.Union* (page 1197), *sympy.sets.sets.Intersection* (page 1198), *sympy.sets.sets.Complement* (page 1200), *sympy.sets.sets.EmptySet* (page 1202), *sympy.sets.sets.UniversalSet* (page 1202), *sympy.sets.sets.FiniteSet* (page 1197), *sympy.sets.sets.SymmetricDifference* (page 1201), *sympy.sets.sets.DisjointUnion* (page 1201)

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()

Find the Power set of `self`.

Examples

```
>>> 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]

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*)

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))
```

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

References

[R746]

union(*other*)

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*)

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
```

```
>>> 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 (page 1206)

Elementary Sets

Interval

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

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)
```

```
>>> 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]

classmethod Lopen(*a*, *b*)

Return an interval not including the left boundary.

classmethod Ropen(*a*, *b*)

Return an interval not including the right boundary.

as_relational(*x*)

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
```

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*)

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.