## CONTENTS

1 Solvers  
2 Converters  
3 Highlevel Interfaces  
4 Indices and Tables  
Bibliography  
Python Module Index  
Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Solvers</td>
<td>3</td>
</tr>
<tr>
<td>2 Converters</td>
<td>19</td>
</tr>
<tr>
<td>3 Highlevel Interfaces</td>
<td>27</td>
</tr>
<tr>
<td>4 Indices and Tables</td>
<td>31</td>
</tr>
<tr>
<td>Bibliography</td>
<td>33</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>35</td>
</tr>
<tr>
<td>Index</td>
<td>37</td>
</tr>
</tbody>
</table>
Sage supports solving clauses in Conjunctive Normal Form (see Wikipedia article Conjunctive_normal_form), i.e., SAT solving, via an interface inspired by the usual DIMACS format used in SAT solving [SG09]. For example, to express that:

\[ x_1 \lor x_2 \lor \neg x_3 \]

should be true, we write:

\[(1, 2, -3)\]

**Warning:** Variable indices **must** start at one.
By default, Sage solves SAT instances as an Integer Linear Program (see sage.numerical.mip), but any SAT solver supporting the DIMACS input format is easily interfaced using the sage.sat.solvers.dimacs. DIMACS blueprint. Sage ships with pre-written interfaces for RSat [RS] and Glucose [GL]. Furthermore, Sage provides an interface to the CryptoMiniSat [CMS] SAT solver which can be used interchangeably with DIMACS-based solvers. For this last solver, the optional CryptoMiniSat package must be installed, this can be accomplished by typing the following in the shell:

```
sage -i cryptominisat sagelib
```

We now show how to solve a simple SAT problem.

\[(x_1 \text{ OR } x_2 \text{ OR } x_3) \text{ AND } (x_1 \text{ OR } x_2 \text{ OR } \neg x_3)\]

In Sage's notation:

```
sage: solver = SAT()
sage: solver.add_clause( ( 1, 2, 3) )
sage: solver.add_clause( ( 1, 2, -3) )
sage: solver()  # random
(None, True, True, False)
```

**Note:** `add_clause()` creates new variables when necessary. When using CryptoMiniSat, it creates all variables up to the given index. Hence, adding a literal involving the variable 1000 creates up to 1000 internal variables.

DIMACS-base solvers can also be used to write DIMACS files:

```
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: solver.add_clause( ( 1, 2, 3) )
sage: solver.add_clause( ( 1, 2, -3) )
sage: _ = solver.write()
sage: for line in open(fn).readlines():
    print(line)
p cnf 3 2
1 2 3 0
1 2 -3 0
```

Alternatively, there is `sage.sat.solvers.dimacs.DIMACS.clauses()`.
These files can then be passed external SAT solvers.

### 1.1 Details on Specific Solvers

#### 1.1.1 Abstract SAT Solver

All SAT solvers must inherit from this class.

**Note:** Our SAT solver interfaces are 1-based, i.e., literals start at 1. This is consistent with the popular DIMACS format for SAT solving but not with Python’s 0-based convention. However, this also allows to construct clauses using simple integers.

**AUTHORS:**

- Martin Albrecht (2012): first version

```python
sage.sat.solvers.satsolver.SAT(solver=None, **args, **kwds)
```

Return a `SatSolver` instance.

Through this class, one can define and solve SAT problems.

**INPUT:**

- `solver` (string) – select a solver. Admissible values are:
  - "cryptominisat" – note that the cryptominisat package must be installed.
  - "picosat" – note that the picosat package must be installed.
  - "glucose" – note that the glucose package must be installed.
  - "glucose-syrup" – note that the glucose package must be installed.
  - "LP" – use `SatLP` to solve the SAT instance.
  - None (default) – use CryptoMiniSat if available, else PicoSAT if available, and a LP solver otherwise.

**EXAMPLES:**

```python
sage: SAT(solver="LP")
```

an ILP-based SAT Solver

```python
class sage.sat.solvers.satsolver.SatSolver
Bases: object
```
add_clause\((lits)\)
Add a new clause to set of clauses.

INPUT:
\begin{itemize}
\item lits - a tuple of integers \neq 0
\end{itemize}

**Note:** If any element \(e\) in \(lits\) has \(|e|\) greater than the number of variables generated so far, then new variables are created automatically.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.add_clause( (1, -2 , 3) )
Traceback (most recent call last):
  ... 
NotImplementedError
```

clauses\((filename=\text{None})\)
Return original clauses.

INPUT:
\begin{itemize}
\item filename - if not \text{None} clauses are written to filename in DIMACS format (default: None)
\end{itemize}

OUTPUT:
If filename is \text{None} then a list of lits, is_xor, rhs tuples is returned, where lits is a tuple of literals, is_xor is always False and rhs is always \text{None}.

If filename points to a writable file, then the list of original clauses is written to that file in DIMACS format.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.clauses()
Traceback (most recent call last):
  ... 
NotImplementedError
```

conflict_clause()
Return conflict clause if this instance is UNSAT and the last call used assumptions.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.conflict_clause()
Traceback (most recent call last):
  ... 
NotImplementedError
```

learnt_clauses\((\text{unitary_only}=\text{False})\)
Return learnt clauses.

**INPUT:**
1.1. Details on Specific Solvers
• unitary_only - return only unitary learnt clauses (default: False)

EXAMLES:

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.learnt_clauses()
Traceback (most recent call last):
... NotImplmmentedError
sage: solver.learnt_clauses(unitary_only=True)
Traceback (most recent call last):
... NotImplmmentedError
```

**nvars()**

Return the number of variables.

EXAMLES:

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
sage: solver.nvars()
Traceback (most recent call last):
... NotImplmmentedError
```

**read(filename)**

Reads DIMAC files.


The differences were summarized in the discussion on the ticket trac ticket #16924. This method assumes the following DIMACS format:

• Any line starting with “c” is a comment
• Any line starting with “p” is a header
• Any variable 1-n can be used
• Every line containing a clause must end with a “0”

The format is extended to allow lines starting with “x” defining xor clauses, with the notation introduced in cryptominisat, see https://www.msoos.org/xor-clauses/

INPUT:

• filename - The name of a file as a string or a file object

EXAMLES:

```python
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file.
p cnf 3 2
1 -3 0
2 3 -1 0")
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.read(file_object)
```

(continues on next page)
With xor clauses:

```python
sage: from io import StringIO
sage: file_object = StringIO("c A sample .cnf file with xor clauses.
\n p cnf 3\n \n 1 2 0\n 3 0\n x1 2 3 0")
```

```python
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
    # optional - cryptominisat
sage: solver = CryptoMiniSat()
    # optional - cryptominisat
sage: solver.read(file_object)
    # optional - cryptominisat
sage: solver.clauses()
    # optional - cryptominisat
```

```
[((1, 2), False, None), ((3,), False, None), ((1, 2, 3), True, True)]
```

```python
sage: solver()
```

```
(None, True, True, True)
```

trait_names()

Allow alias to appear in tab completion.

EXAMPLES:

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
```

```python
sage: solver.trait_names()
```

```
['gens']
```

var(decision=None)

Return a new variable.

INPUT:

- `decision` - is this variable a decision variable?

EXAMPLES:

```python
sage: from sage.sat.solvers.satsolver import SatSolver
sage: solver = SatSolver()
```

```python
sage: solver.var()
```

```
Traceback (most recent call last):
 ...
NotImplementedError
```
1.1.2 SAT-Solvers via DIMACS Files

Sage supports calling SAT solvers using the popular DIMACS format. This module implements infrastructure to make it easy to add new such interfaces and some example interfaces.

Currently, interfaces to **RSat** and **Glucose** are included by default.

**Note:** Our SAT solver interfaces are 1-based, i.e., literals start at 1. This is consistent with the popular DIMACS format for SAT solving but not with Python’s 0-based convention. However, this also allows to construct clauses using simple integers.

**AUTHORS:**
- Martin Albrecht (2012): first version

**Classes and Methods**

```python
class sage.sat.solvers.dimacs.DIMACS(command=None, filename=None, verbosity=0, **kwds):
    Bases: sage.sat.solvers.satsolver.SatSolver
    Generic DIMACS Solver.

    __init__(command=None, filename=None, verbosity=0, **kwds)
    Construct a new generic DIMACS solver.

    INPUT:
    - command - a named format string with the command to run. The string must contain {input} and may contain {output} if the solvers writes the solution to an output file. For example “sat-solver {input}” is a valid command. If None then the class variable command is used. (default: None)
    - filename - a filename to write clauses to in DIMACS format, must be writable. If None a temporary filename is chosen automatically. (default: None)
    - verbosity - a verbosity level, where zero means silent and anything else means verbose output. (default: 0)
    - **kwds - accepted for compatibility with other solves, ignored.

    __call__(assumptions=None)
    Run ‘command’ and collect output.

    INPUT:
    - assumptions - ignored, accepted for compatibility with other solves (default: None)

    add_clause(lits)
    Add a new clause to set of clauses.

    INPUT:
    - lits - a tuple of integers != 0
```
Note: If any element $e$ in lits has $\text{abs}(e)$ greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var()
1
sage: solver.var(decision=True)
2
sage: solver.add_clause( (1, -2, 3) )

sage: solver
DIMACS Solver: ''
```

### clauses (filename=None)

Return original clauses.

**INPUT:**

- filename - if not None clauses are written to filename in DIMACS format (default: None)

**OUTPUT:**

If filename is None then a list of lits, is_xor, rhs tuples is returned, where lits is a tuple of literals, is_xor is always False and rhs is always None.

If filename points to a writable file, then the list of original clauses is written to that file in DIMACS format.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, 2, 3) )
sage: solver.clauses()
[[(1, 2, 3), False, None]]
sage: solver.add_clause( (1, 2, -3) )
sage: solver.clauses(fn)
sage: print(open(fn).read())
p cnf 3 2
1 2 3 0
1 2 -3 0
```

### nvars ()

Return the number of variables.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.var()
1
sage: solver.var(decision=True)
2
sage: solver.nvars()
2
```
static render_dimacs(clauses, filename, nlits)
Produce DIMACS file filename from clauses.

INPUT:

• clauses - a list of clauses, either in simple format as a list of literals or in extended format for
  CryptoMiniSat: a tuple of literals, is_xor and rhs.
• filename - the file to write to
• nlits -- the number of literals appearing in `clauses

EXAMPLES:

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, 2, -3) )
sage: DIMACS.render_dimacs(solver.clauses(), fn, solver.nvars())
sage: print(open(fn).read())
p cnf 3 1
 1 2 -3 0
```

This is equivalent to:

```python
sage: solver.clauses(fn)
sage: print(open(fn).read())
p cnf 3 1
 1 2 -3 0
```

This function also accepts a “simple” format:

```python
sage: DIMACS.render_dimacs([ (1,2), (1,2,-3) ], fn, 3)
sage: print(open(fn).read())
p cnf 3 2
 1 2 0
 1 2 -3 0
```

var(decision=None)
Return a new variable.

INPUT:

• decision - accepted for compatibility with other solvers, ignored.

EXAMPLES:

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: solver = DIMACS()
sage: solver.var()
1
```

write(filename=None)
Write DIMACS file.

INPUT:

• filename - if None default filename specified at initialization is used for writing to (default: None)

EXAMPLES:
```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: solver.add_clause( (1, -2 , 3) )
_ = solver.write()
sage: for line in open(fn).readlines():
  ....:  print(line)
p cnf 3 1
  1 -2 3 0
```

```python
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS()
sage: solver.add_clause( (1, -2 , 3) )
_ = solver.write(fn)
sage: for line in open(fn).readlines():
  ....:  print(line)
p cnf 3 1
  1 -2 3 0
```

```python
class sage.sat.solvers.dimacs.Glucose(command=None, filename=None, verbosity=0, **kwds)
Bases: sage.sat.solvers.dimacs.DIMACS

An instance of the Glucose solver.

For information on Glucose see: http://www.labri.fr/perso/lsimon/glucose/

EXAMPLES:
```n
sage: from sage.sat.solvers import Glucose
sage: solver = Glucose()
sage: solver
DIMACS Solver: 'glucose -verb=2 {input} {output}'
sage: solver.add_clause( (1, 2, 3) )
sage: solver.add_clause( (-1,) )
sage: solver.add_clause( (-2,) )
sage: solver()  # optional - glucose
(Non, False, False, True)
```

```python
class sage.sat.solvers.dimacs.GlucoseSyrup(command=None, filename=None, verbosity=0, **kwds)
Bases: sage.sat.solvers.dimacs.DIMACS

An instance of the Glucose-syrup parallel solver.

For information on Glucose see: http://www.labri.fr/perso/lsimon/glucose/
```

```python
class sage.sat.solvers.dimacs.RSat(command=None, filename=None, verbosity=0, **kwds)
Bases: sage.sat.solvers.dimacs.DIMACS

An instance of the RSat solver.

For information on RSat see: http://reasoning.cs.ucla.edu/rsat/
```
1.1.3 PicoSAT Solver

This solver relies on the pycosat Python bindings to PicoSAT.

The pycosat package should be installed on your Sage installation.

AUTHORS:

```python
class sage.sat.solvers.picosat.PicoSAT(verbosity=0, prop_limit=0)
    Bases: sage.sat.solvers.satsolver.SatSolver
    PicoSAT Solver.

INPUT:
- verbosity – an integer between 0 and 2 (default: 0); verbosity
- prop_limit – an integer (default: 0); the propagation limit

EXAMPLES:
```
```python
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat

add_clause(lits)
Add a new clause to set of clauses.

INPUT:
- lits – a tuple of nonzero integers

Note: If any element e in lits has abs(e) greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:
```
```python
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat
sage: solver.add_clause((1, -2, 3))  # optional - pycosat

clauses(filename=None)
Return original clauses.

INPUT:
- filename – (optional) if given, clauses are written to filename in DIMACS format

OUTPUT:
If filename is None then a list of lits is returned, where lits is a list of literals.

If filename points to a writable file, then the list of original clauses is written to that file in DIMACS format.

EXAMPLES:
```
```python
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat
sage: solver.add_clause((1, -2, 3))  # optional - pycosat
```
sage: solver.clauses()  # optional - pycosat
[[1, 2, 3, 4, 5, 6, 7, 8, -9]]

DIMACS format output:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat
sage: solver.add_clause((1, 2, 4))  # optional - pycosat
sage: solver.add_clause((1, 2, -4))  # optional - pycosat
sage: fn = tmp_filename()  # optional - pycosat
sage: solver.clauses(fn)  # optional - pycosat
sage: print(open(fn).read())  # optional - pycosat
p cnf 4 2
1 2 4 0
1 2 -4 0
```

**nvars()**

Return the number of variables. Note that for compatibility with DIMACS convention, the number of variables corresponds to the maximal index of the variables used.

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat
sage: solver.nvars()  # optional - pycosat
0
```

If a variable with intermediate index is not used, it is still considered as a variable:

```
sage: solver.add_clause((1,-2,4))  # optional - pycosat
sage: solver.nvars()  # optional - pycosat
4
```

**var**(decision=None)

Return a new variable.

INPUT:

* decision – ignored; accepted for compatibility with other solvers

EXAMPLES:

```
sage: from sage.sat.solvers.picosat import PicoSAT
sage: solver = PicoSAT()  # optional - pycosat
sage: solver.var()  # optional - pycosat
1
```

```
sage: solver.add_clause((-1,2,-4))  # optional - pycosat
sage: solver.var()  # optional - pycosat
5
```
1.1.4 Solve SAT problems Integer Linear Programming

The class defined here is a SatSolver that solves its instance using MixedIntegerLinearProgram. Its performance can be expected to be slower than when using CryptoMiniSat.

```python
class sage.sat.solvers.sat_lp.SatLP(solver=None, verbose=0):
    Bases: sage.sat.solvers.satsolver.SatSolver
```

Initializes the instance

**INPUT:**

- `solver` – (default: None) Specify a Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class MixedIntegerLinearProgram.

- `verbose` – integer (default: 0). Sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.

**EXAMPLES:**

```python
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
```

**add_clause(lits)**

Add a new clause to set of clauses.

**INPUT:**

- `lits` - a tuple of integers != 0

**Note:** If any element \( e \) in \( lits \) has \( \text{abs}(e) \) greater than the number of variables generated so far, then new variables are created automatically.

**EXAMPLES:**

```python
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
sage: for u,v in graphs.CycleGraph(6).edges(labels=False):
....:     u,v = u+1,v+1
....:     S.add_clause((u,v))
....:     S.add_clause((-u,-v))
```

**nvars()**

Return the number of variables.

**EXAMPLES:**

```python
sage: S=SAT(solver="LP"); S
an ILP-based SAT Solver
sage: S.var()
1
sage: S.var()
2
sage: S.nvars()
2
```

**var()**

Return a new variable.
1.1.5 CryptoMiniSat Solver

This solver relies on Python bindings provided by upstream cryptominisat.

The cryptominisat package should be installed on your Sage installation.

AUTHORS:

• Thierry Monteil (2017): complete rewrite, using upstream Python bindings, works with cryptominisat 5.
• Martin Albrecht (2012): first version, as a cython interface, works with cryptominisat 2.

```python
class sage.sat.solvers.cryptominisat.CryptoMiniSat(verbosity=0, confl_limit=None, threads=None):
    Bases: sage.sat.solvers.satsolver.SatSolver

CryptoMiniSat Solver.

INPUT:

• verbosity – an integer between 0 and 15 (default: 0). Verbosity.
• confl_limit – an integer (default: None). Abort after this many conflicts. If set to None, never aborts.
• threads – an integer (default: None). The number of thread to use. If set to None, the number of threads used corresponds to the number of cpus.

EXAMPLES:

```python
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
```

`add_clause(lits)`

Add a new clause to set of clauses.

INPUT:

• lits – a tuple of nonzero integers.

Note: If any element e in lits has abs(e) greater than the number of variables generated so far, then new variables are created automatically.

EXAMPLES:

```python
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
sage: solver.add_clause((1, -2, 3))  # optional - cryptominisat
```
**add_xor_clause** *(lits, rhs=True)*

Add a new XOR clause to set of clauses.

**INPUT:**

- **lits** – a tuple of positive integers.
- **rhs** – boolean (default: True). Whether this XOR clause should be evaluated to True or False.

**EXAMPLES:**

```python
sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
sage: solver.add_xor_clause((1, 2, 3), False)  # optional - cryptominisat
```

**clauses** *(filename=None)*

Return original clauses.

**INPUT:**

- **filename** – if not None clauses are written to filename in DIMACS format (default: None)

**OUTPUT:**

If filename is None then a list of lits, is_xor, rhs tuples is returned, where lits is a tuple of literals, is_xor is always False and rhs is always None.

If filename points to a writable file, then the list of original clauses is written to that file in DIMACS format.

**EXAMPLES:**

```python
sage: from sage.sat.solvers import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
sage: solver.add_clause((1, 2, 4))  # optional - cryptominisat
sage: solver.add_clause((1, 2, -4))  # optional - cryptominisat
sage: fn = tmp_filename()  # optional - cryptominisat
sage: solver.clauses(fn)  # optional - cryptominisat
sage: print(open(fn).read())  # optional - cryptominisat
p cnf 4 2
1 2 4 0
1 2 -4 0
```

Note that in cryptominisat, the DIMACS standard format is augmented with the following extension: having an x in front of a line makes that line an XOR clause:
sage: solver.add_xor_clause((1, 2, 3), rhs=True)  # optional - cryptominisat
sage: solver.clauses(fn)  # optional - cryptominisat
p cnf 4 3
1 2 4 0
1 2 -4 0
x1 2 3 0

Note that inverting an xor-clause is equivalent to inverting one of the variables:

sage: solver.add_xor_clause((1, 2, 5), rhs=False)  # optional - cryptominisat
sage: solver.clauses(fn)  # optional - cryptominisat
p cnf 5 4
1 2 4 0
1 2 -4 0
x1 2 3 0
x1 2 -5 0

nvars()
Return the number of variables. Note that for compatibility with DIMACS convention, the number of
variables corresponds to the maximal index of the variables used.

EXAMPLES:

sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
sage: solver.nvars()  # optional - cryptominisat
0

If a variable with intermediate index is not used, it is still considered as a variable:

sage: solver.add_clause((-1, 2, -4))  # optional - cryptominisat
sage: solver.nvars()  # optional - cryptominisat
4

var(decision=None)
Return a new variable.

INPUT:

• decision – accepted for compatibility with other solvers, ignored.

EXAMPLES:

sage: from sage.sat.solvers.cryptominisat import CryptoMiniSat
sage: solver = CryptoMiniSat()  # optional - cryptominisat
sage: solver.var()  # optional - cryptominisat
1
sage: solver.add_clause((-1, 2, -4))  # optional - cryptominisat

sage: solver.var()
cryptominisat 5

# optional
Sage supports conversion from Boolean polynomials (also known as Algebraic Normal Form) to Conjunctive Normal Form:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
```
```
p cnf 3 2
-2 0
1 0
```

### 2.1 Details on Specific Converters

#### 2.1.1 An ANF to CNF Converter using a Dense/Sparse Strategy

This converter is based on two converters. The first one, by Martin Albrecht, was based on [CB2007], this is the basis of the “dense” part of the converter. It was later improved by Mate Soos. The second one, by Michael Brickenstein, uses a reduced truth table based approach and forms the “sparse” part of the converter.

**AUTHORS:**

- Martin Albrecht - (2008-09) initial version of ‘anf2cnf.py’
- Michael Brickenstein - (2009) ‘cnf.py’ for PolyBoRi
- Mate Soos - (2010) improved version of ‘anf2cnf.py’
- Martin Albrecht - (2012) unified and added to Sage
Classes and Methods

```python
class sage.sat.converters.polybori.CNFEncoder(solver, ring, max_vars_sparse=6, use_xor_clauses=None, cutting_number=6, random_seed=16):
    Bases: sage.sat.converters.anf2cnf.ANF2CNFConverter

ANF to CNF Converter using a Dense/Sparse Strategy. This converter distinguishes two classes of polynomials.

1. Sparse polynomials are those with at most `max_vars_sparse` variables. Those are converted using reduced truth-tables based on PolyBoRi’s internal representation.

2. Polynomials with more variables are converted by introducing new variables for monomials and by converting these linearised polynomials.

Linearised polynomials are converted either by splitting XOR chains — into chunks of length `cutting_number` — or by constructing XOR clauses if the underlying solver supports it. This behaviour is disabled by passing `use_xor_clauses=False`.

```python
__init__(solver, ring, max_vars_sparse=6, use_xor_clauses=None, cutting_number=6, random_seed=16)
```

Construct ANF to CNF converter over `ring` passing clauses to `solver`.

**INPUT:**

- `solver` - a SAT-solver instance
- `ring` - a `sage.rings.polynomial.pbori.BooleanPolynomialRing`
- `max_vars_sparse` - maximum number of variables for direct conversion
- `use_xor_clauses` - use XOR clauses; if `None` use if `solver` supports it. (default: `None`)
- `cutting_number` - maximum length of XOR chains after splitting if XOR clauses are not supported (default: `6`)
- `random_seed` - the direct conversion method uses randomness, this sets the seed (default: `16`)

**EXAMPLES:**

We compare the sparse and the dense strategies, sparse first:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
-2 0
1 0
sage: e.phi
[None, a, b, c]
```

Now, we convert using the dense strategy:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
```

(continues on next page)
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_dense(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())

p cnf 4 5
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 0
4 1 0

sage: e.phi
[None, a, b, c, a*b]

Note: This constructor generates SAT variables for each Boolean polynomial variable.

__call__(F)

Encode the boolean polynomials in \( F \).

INPUT:

• \( F \) - an iterable of \texttt{sage.rings.polynomial.pbori.BooleanPolynomial}

OUTPUT: An inverse map int \( \rightarrow \) variable

EXAMPLES:

sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: e([a*b + a + 1, a*b+ a + c])
[None, a, b, c, a*b]
sage: _ = solver.write()
sage: print(open(fn).read())

p cnf 4 9
-2 0
1 0
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 -3 0
4 1 -3 0
4 -1 3 0
-4 1 3 0

sage: e.phi
[None, a, b, c, a*b]

clauses(f)

Convert \( f \) using the sparse strategy if \( f.nvariables() \) is at most \texttt{max-vars-sparse} and the dense strategy otherwise.

INPUT:
clauses_dense (f)
Convert f using the dense strategy.

INPUT:

• f - a sage.rings.polynomial.pbori.BooleanPolynomial

EXAMPLES:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
data: from sage.sat.converters.polybori import DIMACS
data: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: e.clauses(a*b + a + c)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 4 7
1 -4 0
2 -4 0
4 -1 -2 0
-4 -1 -3 0
4 1 -3 0
4 -1 3 0
-4 1 3 0
sage: e.phi
[None, a, b, c, a*b]
```

(continues on next page)
clauses_sparse(f)

Convert f using the sparse strategy.

INPUT:

- f - a sage.rings.polynomial.pbori.BooleanPolynomial

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_sparse(a*b + a + 1)
sage: _ = solver.write()
sage: print(open(fn).read())
p cnf 3 2
  -2 0
  1 0
sage: e.phi
[None, a, b, c]
```

monomial(m)

Return SAT variable for m.

INPUT:

- m - a monomial.

OUTPUT: An index for a SAT variable corresponding to m.

EXAMPLES:

```
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B)
sage: e.clauses_dense(a*b + a + 1)
sage: e.phi
[None, a, b, c, a*b]
sage: e.monomial(a*b*c)
5
sage: e.phi
[None, a, b, c, a*b, a*b*c]
```

2.1. Details on Specific Converters
If monomial is called on a monomial that was queried before, the index of the old variable is returned and no new variable is created:

```python
sage: e.monomial(a*b)
4
sage: e.phi
[None, a, b, c, a*b, a*b*c]
```

.. note::

   For correctness, this function is cached.

.. function:: permutations(length, equal_zero)

   Return permutations of length `length` which are equal to zero if `equal_zero` and equal to one otherwise.

   A variable is false if the integer in its position is smaller than zero and true otherwise.

   **INPUT:**

   - `length` – the number of variables
   - `equal_zero` – should the sum be equal to zero?

   **EXAMPLES:**

```python
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dhmacs import DIMACS
sage: B.<a,b,c> = BooleanPolynomialRing()
```

```python
sage: ce = CNFEncoder(DIMACS(), B)
```

```python
sage: ce.permutations(3, True)
[[[-1, -1, -1], [1, 1, -1], [1, -1, 1], [-1, 1, 1]]

sage: ce.permutations(3, False)
[[[1, -1, -1], [-1, 1, -1], [-1, -1, 1], [1, 1, 1]]
```

.. function:: phi

   Map SAT variables to polynomial variables.

   **EXAMPLES:**

```python
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dhmacs import DIMACS
sage: B.<a,b,c> = BooleanPolynomialRing()
```

```python
sage: ce = CNFEncoder(DIMACS(), B)
```

```python
sage: ce.var()  
4
sage: ce.phi
[None, a, b, c, None]
```

.. function:: split_xor(monomial_list, equal_zero)

   Split XOR chains into subchains.

   **INPUT:**

   - `monomial_list` - a list of monomials
   - `equal_zero` - is the constant coefficient zero?

   **EXAMPLES:**
```python
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: ce = CNFEncoder(DIMACS(), B, cutting_number=3)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 7], False], [[7, 2, 8], True], [[8, 3, 9], True], [[9, 4, 10], True],
  [[10, 5, 11], True]]
sage: ce = CNFEncoder(DIMACS(), B, cutting_number=4)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 2, 7], False], [[7, 3, 4, 8], True], [[8, 5, 6], True]]
sage: ce = CNFEncoder(DIMACS(), B, cutting_number=5)
sage: ce.split_xor([1,2,3,4,5,6], False)
[[[1, 2, 3, 7], False], [[7, 4, 5, 6], True]]
```

### to_polynomial(c)

Convert clause to `sage.rings.polynomial.pbori.BooleanPolynomial`

**INPUT:**

- `c` - a clause

**EXAMPLES:**

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: fn = tmp_filename()
sage: solver = DIMACS(filename=fn)
sage: e = CNFEncoder(solver, B, max_vars_sparse=2)
sage: _ = e([a*b + a + 1, a*b+ a + c])
sage: e.to_polynomial((1,-2,3))
a*b*c + a*b + b*c + b
```

### var(m=None, decision=None)

Return a new variable.

This is a thin wrapper around the SAT-solvers function where we keep track of which SAT variable cor-responds to which monomial.

**INPUT:**

- `m` - something the new variables maps to, usually a monomial
- `decision` - is this variable a decision variable?

**EXAMPLES:**

```python
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: ce = CNFEncoder(DIMACS(), B)
sage: ce.var()  # None
4
```

### zero_blocks(f)

Divide the zero set of `f` into blocks.

**EXAMPLES:**

```python
```

---

2.1. Details on Specific Converters 25
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: from sage.sat.converters.polybori import CNFEncoder
sage: from sage.sat.solvers.dimacs import DIMACS
sage: e = CNFEncoder(DIMACS(), B)
sage: sorted(sorted(d.items()) for d in e.zero_blocks(a*b*c))
[[{c, 0}], [{b, 0}], [{a, 0}]]

Note: This function is randomised.
Sage provides various highlevel functions which make working with Boolean polynomials easier. We construct a very small-scale AES system of equations and pass it to a SAT solver:

```python
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True)

sage: F,s = sr.polynomial_system()

sage: from sage.sat.boolean_polynomials import solve as solve_sat

# optional - cryptominisat

sage: s = solve_sat(F)

# optional - cryptominisat

sage: F.subs(s[0])

# optional - cryptominisat
```

Polynomial Sequence with 36 Polynomials in 0 Variables

3.1 Details on Specific Highlevel Interfaces

3.1.1 SAT Functions for Boolean Polynomials

These highlevel functions support solving and learning from Boolean polynomial systems. In this context, “learning” means the construction of new polynomials in the ideal spanned by the original polynomials.

**AUTHOR:**

- Martin Albrecht (2012): initial version

**Functions**

```python
sage.sat.boolean_polynomials.learn(F, converter=None, solver=None, max_learnt_length=3, interreduction=False, **kwds)
```

Learn new polynomials by running SAT-solver `solver` on SAT-instance produced by `converter` from `F`.

**INPUT:**

- `F` - a sequence of Boolean polynomials
- `converter` - an ANF to CNF converter class or object. If `converter` is `None` then `sage.sat.converters.polybori.CNFEncoder` is used to construct a new converter. (default: `None`)
- `solver` - a SAT-solver class or object. If `solver` is `None` then `sage.sat.solvers.cryptominisat.CryptoMiniSat` is used to construct a new converter. (default: `None`)
- `max_learnt_length` - only clauses of length $\leq$ `max_length_learnt` are considered and converted to polynomials. (default: `3`)
• **interreduction** - inter-reduce the resulting polynomials (default: False)

**Note:** More parameters can be passed to the converter and the solver by prefixing them with `c_` and `s_` respectively. For example, to increase CryptoMiniSat’s verbosity level, pass `s_verbosity=1`.

**OUTPUT:**

A sequence of Boolean polynomials.

**EXAMPLES:**

```python
sage: from sage.sat.boolean_polynomials import learn as learn_sat
# optional - cryptominisat

We construct a simple system and solve it:

```python
define

sage: set_random_seed(2300) # optional - cryptominisat
sage: sr = mq.SR(1,2,2,4,gf2=True,polybori=True) # optional - cryptominisat
sage: F,s = sr.polynomial_system() # optional - cryptominisat
sage: H = learn_sat(F) # optional - cryptominisat
sage: H[-1] # optional - cryptominisat
k033 + 1
```

**sage.sat.boolean_polynomials.solve**(*F*, *converter=None*, *solver=None*, *n=1*, **kwds)
Solve system of Boolean polynomials *F* by solving the SAT-problem – produced by *converter* – using *solver*.

**INPUT:**

- **F** - a sequence of Boolean polynomials
- **n** - number of solutions to return. If *n* is +infinity then all solutions are returned. If *n* < infinity then *n* solutions are returned if *F* has at least *n* solutions. Otherwise, all solutions of *F* are returned. (default: 1)
- **converter** - an ANF to CNF converter class or object. If *converter* is None then *sage.sat.converters.polybori.CNFEncoder* is used to construct a new converter. (default: None)
- **solver** - a SAT-solver class or object. If *solver* is None then *sage.sat.solvers.cryptominisat.CryptoMiniSat* is used to construct a new converter. (default: None)
- **target_variables** - a list of variables. The elements of the list are used to exclude a particular combination of variable assignments of a solution from any further solution. Furthermore **target_variables** denotes which variable-value pairs appear in the solutions. If **target_variables** is None all variables appearing in the polynomials of *F* are used to construct exclusion clauses. (default: None)
- ****kwds** - parameters can be passed to the converter and the solver by prefixing them with `c_` and `s_` respectively. For example, to increase CryptoMiniSat’s verbosity level, pass `s_verbosity=1`.

**OUTPUT:**

A list of dictionaries, each of which contains a variable assignment solving *F*.

**EXAMPLES:**

We construct a very small-scale AES system of equations:

```python
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True)
sage: F,s = sr.polynomial_system()
```
and pass it to a SAT solver:

```python
sage: from sage.sat.boolean_polynomials import solve as solve_sat  # optional - cryptominisat
group 0: 0/0.0s
sage: s = solve_sat(F)  # optional - cryptominisat
  group 0: 0/0.0s
sage: F.subs(s[0])  # optional - cryptominisat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

This time we pass a few options through to the converter and the solver:

```python
sage: s = solve_sat(F, s_verbosity=1, c_max_vars_sparse=4, c_cutting_number=8)  # optional - cryptominisat
  group 0: 0/0.0s
c ...
...
sage: F.subs(s[0])  # optional - cryptominisat
Polynomial Sequence with 36 Polynomials in 0 Variables
```

We construct a very simple system with three solutions and ask for a specific number of solutions:

```python
sage: B.<a,b> = BooleanPolynomialRing()  # optional - cryptominisat
sage: f = a*b  # optional - cryptominisat
sage: l = solve_sat([f], n=1)  # optional - cryptominisat
  group 0: 0/0.0s
sage: len(l) == 1, f.subs(l[0])  # optional - cryptominisat
(True, 0)
sage: l = solve_sat([a*b], n=2)  # optional - cryptominisat
  group 0: 0/0.0s
sage: len(l) == 2, f.subs(l[0]), f.subs(l[1])  # optional - cryptominisat
(True, 0, 0)
sage: sorted((d[a], d[b]) for d in solve_sat([a*b], n=3))  # optional - cryptominisat
[(0, 0), (0, 1), (1, 0)]
sage: sorted((d[a], d[b]) for d in solve_sat([a*b], n=4))  # optional - cryptominisat
[(0, 0), (0, 1), (1, 0)]
sage: sorted((d[a], d[b]) for d in solve_sat([a*b], n=infinity))  # optional - cryptominisat
[(0, 0), (0, 1), (1, 0)]
```

In the next example we see how the `target_variables` parameter works:

```python
sage: from sage.sat.boolean_polynomials import solve as solve_sat  # optional - cryptominisat
sage: R.<a,b,c,d> = BooleanPolynomialRing()  # optional - cryptominisat
sage: F = [a+b, a+c+d]  # optional - cryptominisat
sage: sorted((D[a], D[b], D[c], D[d]) for D in solve_sat(F, n=infinity))  # optional - cryptominisat
[(0, 0, 0, 0), (0, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
```

First the normal use case:

```python
sage: sorted((D[a], D[b]) for D in solve_sat(F, n=infinity))  # optional - cryptominisat
[(0, 0), (0, 0, 1, 1), (1, 1, 0, 1), (1, 1, 1, 0)]
```

Now we are only interested in the solutions of the variables a and b:
Here, we generate and solve the cubic equations of the AES SBox (see trac ticket #26676):

```
sage: from sage.rings.polynomial.multi_polynomial_sequence import PolynomialSequence # optional - cryptominisat, long time
sage: from sage.sat.boolean_polynomials import solve as solve_sat # optional - cryptominisat, long time
sage: sr = sage.crypto.mq.SR(1, 4, 4, 8, allow_zero_inversions = True) # optional - cryptominisat, long time
sage: sb = sr.sbox() # optional - cryptominisat, long time
sage: eqs = sb.polynomials(degree = 3) # optional - cryptominisat, long time
sage: eqs = PolynomialSequence(eqs) # optional - cryptominisat, long time
sage: variables = map(str, eqs.variables()) # optional - cryptominisat, long time
sage: variables = ','.join(variables) # optional - cryptominisat, long time
sage: R = BooleanPolynomialRing(16, variables) # optional - cryptominisat, long time
sage: eqs = [R(eq) for eq in eqs] # optional - cryptominisat, long time
sage: sls_aes = solve_sat(eqs, n = infinity) # optional - cryptominisat, long time
sage: len(sls_aes) # optional - cryptominisat, long time
256
```

**Note:** Although supported, passing converter and solver objects instead of classes is discouraged because these objects are stateful.

REFERENCES:
INDICES AND TABLES

• Index
• Module Index
• Search Page
BIBLIOGRAPHY

S
sage.sat.boolean_polynomials, 27
sage.sat.converters.polybori, 19
sage.sat.solvers.cryptominisat, 15
sage.sat.solvers.dimacs, 8
sage.sat.solvers.picosat, 12
sage.sat.solvers.sat_lp, 14
sage.sat.solvers.satsolver, 4
Symbols

__call__() (sage.sat.converters.polybori.CNFEncoder method), 21
__call__() (sage.sat.solvers.dimacs.DIMACS method), 8
__init__() (sage.sat.converters.polybori.CNFEncoder method), 20
__init__() (sage.sat.solvers.dimacs.DIMACS method), 8

A
add_clause() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 15
add_clause() (sage.sat.solvers.dimacs.DIMACS method), 8
add_clause() (sage.sat.solvers.picosat.PicoSAT method), 12
add_clause() (sage.sat.solvers.sat_lp.SatLP method), 14
add_clause() (sage.sat.solvers.sat_solver.SatSolver method), 4
add_xor_clause() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 15

C
clauses() (sage.sat.converters.polybori.CNFEncoder method), 21
clauses() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 16
clauses() (sage.sat.solvers.dimacs.DIMACS method), 9
clauses() (sage.sat.solvers.picosat.PicoSAT method), 12
clauses() (sage.sat.solvers.sat_solver.SatSolver method), 5
clauses_dense() (sage.sat.converters.polybori.CNFEncoder method), 22
clauses_sparse() (sage.sat.converters.polybori.CNFEncoder method), 23
CNFEncoder (class in sage.sat.converters.polybori), 20
CNFEncoder.permutations() (in module sage.sat.converters.polybori), 24
conflict_clause() (sage.sat.solvers.sat_solver.SatSolver method), 5
CryptoMiniSat (class in sage.sat.solvers.cryptominisat), 15

D
DIMACS (class in sage.sat.solvers.dimacs), 8

G
Glucose (class in sage.sat.solvers.dimacs), 11
GlucoseSyrup (class in sage.sat.solvers.dimacs), 11

L
learn() (in module sage.sat.boolean_polynomials), 27
learnt_clauses() (sage.sat.solvers.sat_solver.SatSolver method), 5
M
module
    sage.sat.boolean_polynomials, 27
    sage.sat.converters.polybori, 19
    sage.sat.solvers.cryptominisat, 15
    sage.sat.solvers.dimacs, 8
    sage.sat.solvers.picosat, 12
    sage.sat.solvers.sat_lp, 14
    sage.sat.solvers.satsolver, 4
monomial() (sage.sat.converters.polybori.CNFEncoder method), 23

N
nvars() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 17
nvars() (sage.sat.solvers.dimacs.DIMACS method), 9
nvars() (sage.sat.solvers.picosat.PicoSAT method), 13
nvars() (sage.sat.solvers.sat_lp.SatLP method), 14
nvars() (sage.sat.solvers.satsolver.SatSolver method), 6

P
phi (sage.sat.converters.polybori.CNFEncoder attribute), 24
PicoSAT (class in sage.sat.solvers.picosat), 12

R
read() (sage.sat.solvers.satsolver.SatSolver method), 6
render_dimacs() (sage.sat.solvers.dimacs.DIMACS static method), 9
RSat (class in sage.sat.solvers.dimacs), 11

S
sage.sat.boolean_polynomials
    module, 27
sage.sat.converters.polybori
    module, 19
sage.sat.solvers.cryptominisat
    module, 15
sage.sat.solvers.dimacs
    module, 8
sage.sat.solvers.picosat
    module, 12
sage.sat.solvers.sat_lp
    module, 14
sage.sat.solvers.satsolver
    module, 4
SAT() (in module sage.sat.solvers.satsolver), 4
SatLP (class in sage.sat.solvers.sat_lp), 14
SatSolver (class in sage.sat.solvers.satsolver), 4
solve() (in module sage.sat.boolean_polynomials), 28
split_xor() (sage.sat.converters.polybori.CNFEncoder method), 24

T
to_polynomial() (sage.sat.converters.polybori.CNFEncoder method), 25
trait_names() (sage.sat.solvers.satsolver.SatSolver method), 7

V
var() (sage.sat.converters.polybori.CNFEncoder method), 25
var() (sage.sat.solvers.cryptominisat.CryptoMiniSat method), 17
var() (sage.sat.solvers.dimacs.DIMACS method), 10
var() (sage.sat.solvers.picosat.PicoSAT method), 13
var() (sage.sat.solvers.sat_lp.SatLP method), 14
var() (sage.sat.solvers.satsolver.SatSolver method), 7

W
write() (sage.sat.solvers.dimacs.DIMACS method), 10

Z
zero_blocks() (sage.sat.converters.polybori.CNFEncoder method), 25