tt

Release 0.6.4

May 11, 2020

Contents

Welcome to the documentation site for tt!

Warning: tt is heavily tested and fully usable, but is still pre-1.0/stable software with **no guarantees** of avoiding breaking API changes until hitting version 1.0.

CHAPTER 1

Synopsis

tt (**t**ruth **t**able) is a library aiming to provide a Pythonic toolkit for working with Boolean expressions and truth tables. Please see the project site for guides and documentation, or check out bool.tools for a simple web application powered by this library.

CHAPTER 2

Installation

tt is tested on CPython 3.6, 3.7, and 3.8. You can get the latest release from PyPI with:

```
pip install ttable
```

CHAPTER 3

Features

Parse expressions:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A impl not (B nand C)')
>>> b.tokens
['A', 'impl', 'not', '(', 'B', 'nand', 'C', ')']
>>> print(b.tree)
impl
`----A
`----not
     `----nand
          `----B
          `----C
```

Evaluate expressions:

```
>>> b = BooleanExpression('(A /\ B) -> (C \/ D)')
>>> b.evaluate(A=1, B=1, C=0, D=0)
False
>>> b.evaluate(A=1, B=1, C=1, D=0)
True
```

Interact with expression structure:

```
>>> b = BooleanExpression('(A and ~B and C) or (~C and D) or E')
>>> b.is_dnf
True
>>> for clause in b.iter_dnf_clauses():
...     print(clause)
...
A and ~B and C
~C and D
E
```

Apply expression transformations:

```
>>> from tt import to_primitives, to_cnf
>>> to_primitives('A xor B')
<BooleanExpression "(A and not B) or (not A and B)">
>>> to_cnf('(A nand B) impl (C or D)')
<BooleanExpression "(A or C or D) and (B or C or D)">
```

Or create your own:

```
>>> from tt import tt_compose, apply_de_morgans, coalesce_negations, twice
>>> b = BooleanExpression('not (not (A or B))')
>>> f = tt_compose(apply_de_morgans, twice)
>>> f(b)
<BooleanExpression "not not A or not not B">
>>> g = tt_compose(f, coalesce_negations)
>>> g(b)
<BooleanExpression "A or B">
```

Exhaust SAT solutions:

```
>>> b = BooleanExpression('~(A or B) xor C')
>>> for sat_solution in b.sat_all():
...     print(sat_solution)
...
A=0, B=1, C=1
A=1, B=0, C=1
A=1, B=1, C=1
A=0, B=0, C=0
```

Find just a few:

```
>>> with b.constrain(A=1):
...     for sat_solution in b.sat_all():
...         print(sat_solution)
...
A=1, B=0, C=1
A=1, B=1, C=1
```

Or just one:

```
>>> b.sat_one()
<BooleanValues [A=0, B=1, C=1]>
```

Build truth tables:

```
>>> from tt import TruthTable
>>> t = TruthTable('A iff B')
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 1 |
+---+---+---+
| 0 | 1 | 0 |
+---+---+---+
| 1 | 0 | 0 |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+
```

And much more!

CHAPTER 4

License

tt uses the MIT License.

4.1 User Guide

Exploring the topics listed below should give you an idea of how to use the tools provided in this library. If anything remains unclear, please feel free to open an issue on GitHub or reach out to *the author*.

4.1.1 Expression basics

At tt's core is the concept of the Boolean expression, encapsulated in this library with the *BooleanExpression* class. Let's take look at what we can do with expressions.

Creating an expression object

The top-level class for interacting with boolean expressions in tt is, fittingly named, *BooleanExpression*. Let's start by importing it:

```
>>> from tt import BooleanExpression
```

This class accepts boolean expressions as strings and provides the interface for parsing and tokenizing string expressions into a sequence of tokens and symbols, as we see here:

```
>>> b = BooleanExpression('(A nand B) or (C and D)')
>>> b.tokens
['(', 'A', 'nand', 'B', ')', 'or', '(', 'C', 'and', 'D', ')']
>>> b.symbols
['A', 'B', 'C', 'D']
```

We can also always retrieve the original string we passed in via the `raw_expr` attribute:

```
>>> b.raw_expr
'(A nand B) or (C and D)'
```

During initialization, the *BooleanExpression* also does some work behind the scenes to build a basic understanding of the expression's structure. It re-orders the tokens into postfix order, and uses this representation to build a *ExpressionTreeNode*. We can see this with:

```
>>> b.postfix_tokens
['A', 'B', 'nand', 'C', 'D', 'and', 'or']
>>> print(b.tree)
or
`----nand
|    `----A
|    `----B
`----and
     `----C
     `----D
```

This expression tree represents tt's understanding of the structure of your expression. If you are receiving an unexpected error for a more complicated expression, inspecting the `tree` attribute on the *BooleanExpression* instance can be a good starting point for debugging the issue.

Evaluating expressions

Looking at expression symbols and tokens is nice, but we need some real functionality for our expressions; a natural starting point is the ability to evaluate expressions. A *BooleanExpression* object provides an interface to this evaluation functionality; use it like this:

```
>>> b.evaluate(A=True, B=False, C=True, D=False)
True
>>> b.evaluate(A=1, B=0, C=1, D=0)
True
```

Notice that we can use `0` or `False` to represent low values and `1` or `True` to represent high values. tt makes sure that only valid Boolean-esque values are accepted for evaluation. For example, if we tried something like:

```
>>> b.evaluate(A=1, B='not a Boolean value', C=0, D=0)
Traceback (most recent call last):
    ...
tt.errors.evaluation.InvalidBooleanValueError: "not a Boolean value" passed as value␣
↪for "B" is not a valid Boolean value
```

or if we didn't include a value for each of the symbols:

```
>>> b.evaluate(A=1, B=0, C=0)
Traceback (most recent call last):
    ...
tt.errors.symbols.MissingSymbolError: Did not receive value for the following␣
↪symbols: "D"
```

These exceptions can be nice if you aren't sure about your input, but if you think this safety is just adding overhead for you, there's a way to skip those extra checks:

```
>>> b.evaluate_unchecked(A=0, B=0, C=1, D=0)
True
```

Handling malformed expressions

So far, we've only seen one example of a *BooleanExpression* instance, and we passed a valid expression string to it. What happens when we pass in a malformed expression? And what does tt even consider to be a malformed expression?

While there is no explicit grammar for expressions in tt, using your best judgement will work most of the time. Most well-known Boolean expression operators are available in plain-English and symbolic form. You can see the list of available operators like so:

```
>>> from tt import OPERATOR_MAPPING
>>> print(', '.join(sorted(OPERATOR_MAPPING.keys())))
!, &, &&, ->, /\, <->, AND, IFF, IMPL, NAND, NOR, NOT, NXOR, OR, XNOR, XOR, \/, and,␣
↪iff, impl, nand, nor, not, nxor, or, xnor, xor, |, ||, ~
```

Another possible source of errors in your expressions will be invalid symbol names. Due to some functionality based on accessing symbol names from `namedtuple`-like objects, symbol names must meet the following criteria:

1. Must be a valid Python identifiers.
2. Cannot be a Python keyword.
3. Cannot begin with an underscore

An exception will be raised if a symbol name in your expression does not meet the above criteria. Fortunately, tt provides a way for us to check if our symbols are valid. Let's take a look:

```
>>> from tt import is_valid_identifier
>>> is_valid_identifier('False')
False
>>> is_valid_identifier('_bad')
False
>>> is_valid_identifier('not$good')
False
>>> is_valid_identifier('a_good_symbol_name')
True
>>> b = BooleanExpression('_A or B')
Traceback (most recent call last):
    ...
tt.errors.grammar.InvalidIdentifierError: Invalid operand name "_A"
```

As we saw in the above example, we caused an error from the `tt.errors.grammar` module. If you play around with invalid expressions, you'll notice that all of these errors come from that module; that's because errors in this logical group are all descendants of *GrammarError*. This is the type of error that lexical expression errors will fall under:

```
>>> from tt import GrammarError
>>> invalid_expressions = ['A xor or B', 'A or ((B nand C)', 'A or B B']
>>> for expr in invalid_expressions:
...     try:
...             b = BooleanExpression(expr)
...     except Exception as e:
...             print(type(e))
...             print(isinstance(e, GrammarError))
...
<class 'tt.errors.grammar.ExpressionOrderError'>
True
<class 'tt.errors.grammar.UnbalancedParenError'>
True
```

(continues on next page)

(continued from previous page)

```
<class 'tt.errors.grammar.ExpressionOrderError'>
True
```

GrammarError is a unique type of exception in tt, as it provides attributes for accessing the specific position in the expression string that caused an error. This is best illustrated with an example:

```
>>> try:
...     b = BooleanExpression('A or or B')
... except GrammarError as e:
...     print("Here's what happened:")
...     print(e.message)
...     print("Here's where it happened:")
...     print(e.expr_str)
...     print(' '*e.error_pos + '^')
...
Here's what happened:
Unexpected binary operator "or"
Here's where it happened:
A or or B
     ^
```

4.1.2 Table basics

Truth tables are a nice way of showing the behavior of an expression for each permutation of possible inputs and are nice tool to pair with expressions. Let's examine the interface provided by tt for working with truth tables.

Creating a table object from an expression

Surprisingly, the top-level class for dealing with truth tables in tt is called *TruthTable*. Let's begin by importing it:

```
>>> from tt import TruthTable
```

There are a few ways we can fill up a truth table in tt. One of them is to pass in an expression, either as an already-created *BooleanExpression* object or as a string:

```
>>> t = TruthTable('A xor B')
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
```

As we we in the above example, printing tables produces a nicely-formatted text table. Tables will scale to fit the size of the symbol names, too:

```
>>> t = TruthTable('operand_1 and operand_2')
>>> print(t)
+-----------+-----------+---+
| operand_1 | operand_2 |   |
+-----------+-----------+---+
|     0     |     0     | 0 |
+-----------+-----------+---+
|     0     |     1     | 0 |
+-----------+-----------+---+
|     1     |     0     | 0 |
+-----------+-----------+---+
|     1     |     1     | 1 |
+-----------+-----------+---+
```

By default, tt will order the symbols in the top row of of the table to match the order of their appearance in the original expression; however, you can impose your own order, too:

```
>>> t = TruthTable('A xor B', ordering=['B', 'A'])
>>> print(t)
+---+---+---+
| B | A |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
```

Creating a table object from values

The tables we looked at above were populated by evaluating the expression for each combination of input values, but let's say that you already have the values you want in your truth table. You'd populate your table like this:

```
>>> t = TruthTable(from_values='00x1')
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 0 |
+---+---+---+
| 1 | 0 | x |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+
```

Notice that populating tables like this allows for *don't cares* (indicated by `'x'`) to be present in your table. Additionally, we can see that symbol names were automatically generated for us. That's nice sometimes, but what if we want to specify them ourselves? We return to the `ordering` keyword argument:

```
>>> t = TruthTable(from_values='1x01', ordering=['op1', 'op2'])
>>> print(t)
+-----+-----+---+
| op1 | op2 |   |
+-----+-----+---+
|  0  |  0  | 1 |
+-----+-----+---+
|  0  |  1  | x |
+-----+-----+---+
|  1  |  0  | 0 |
+-----+-----+---+
|  1  |  1  | 1 |
+-----+-----+---+
```

Accessing values from a table

So far, we've only been able to examine the results stored in our tables by printing them. This is nice for looking at an end result, but we need programmatic methods of accessing the values in our tables. There's a few ways to do this in tt; one such example is the *results* attribute present on *TruthTable* objects, which stores all results in the table:

```
>>> t = TruthTable('!A && B')
>>> t.results
[False, True, False, False]
```

Results in the table are also available by indexing the table:

```
>>> t[0], t[1], t[2], t[3]
(False, True, False, False)
```

Accessing results by index is also an intuitive time to use binary literals:

```
>>> t[0b00], t[0b01], t[0b10], t[0b11]
(False, True, False, False)
```

Tables in tt are also iterable. There are a couple of important items to note. First, iterating through the entries in a table will skip over the entries that would have appeared as `None` in the *results* list. Second, in addition to the result, each iteration through the table yields a `namedtuple`-like object representing the inputs associated with that result. Let's take a look:

```
>>> for inputs, result in t:
...     inputs.A, inputs.B
...     str(inputs), result
...
(False, False)
('A=0, B=0', False)
(False, True)
('A=0, B=1', True)
(True, False)
('A=1, B=0', False)
(True, True)
('A=1, B=1', False)
```

Partially filling tables

Up to this point, we've only taken a look at tables with all of their results filled in, but we don't have to completely fill up our tables to start working with them. Here's an example of iteratively filling a table:

```
>>> t = TruthTable('A nor B', fill_all=False)
>>> t.is_full
False
>>> print(t)
Empty!
>>> t.fill(A=0)
>>> t.is_full
False
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 1 |
+---+---+---+
| 0 | 1 | 0 |
+---+---+---+
>>> t.fill()
>>> t.is_full
True
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 1 |
+---+---+---+
| 0 | 1 | 0 |
+---+---+---+
| 1 | 0 | 0 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
```

Empty slots in the table will be represented with a corresponding `None` entry for their result:

```
>>> t = TruthTable('A or B', fill_all=False)
>>> t.results
[None, None, None, None]
>>> t.fill(B=0)
>>> t.results
[False, None, True, None]
```

Make sure not to try to keep filling an already-full table, though:

```
>>> t = TruthTable(from_values='0110')
>>> t.is_full
True
>>> t.fill()
Traceback (most recent call last):
    ...
tt.errors.state.AlreadyFullTableError: Cannot fill an already-full table
```

Logical equivalence

Another neat feature provided by tt's tables is the checking of logical equivalence:

```
>>> t1 = TruthTable('A xor B')
>>> t2 = TruthTable(from_values='0110')
>>> t1.equivalent_to(t2)
True
>>> t1.equivalent_to('C xor D')
True
```

Note that this equivalence comparison looks only at the result values of the tables and doesn't examine at the symbols of either table.

Next, let's examine how *don't cares* function within tt's concept of logical equivalence. *Don't cares* in the calling table will be considered to equal to any value in the comparison table, but any explicity value in the calling table must be matched in the comparison table to be considered equal.

In this sense, a fully-specified table (i.e., one without any *don't cares*) will never be logically equivalent to one which contains *don't cares*, but the converse may be true. Let's see an example:

```
>>> t1 = TruthTable('C nand D')
>>> t2 = TruthTable(from_values='xx10')
>>> t1.equivalent_to(t2)
False
>>> t2.equivalent_to(t1)
True
```

The user guide is a work in progress, with more to come soon!

4.2 Release Notes

Check below for new features added in each release. Please note that release notes were not recorded before version 0.5.0.

4.2.1 0.6.x

Features in the 0.6.x series of releases are focused on expanding functionality to include expression satisfiability and transformations.

0.6.4

- Introduce the *transformations.utils* module, including the *RepeatableAction*, *ComposedTransformation*, AbstractTransformationModifier classes; the *repeat*, *twice*, and *forever* factory classes; and the *tt_compose* utility function
- Publicly expose the *ensure_bexpr* in the *transformations.utils* module
- Drop support for all Python versions except 3.6, 3.7, and 3.8

0.6.3

- Add *non_negated_symbol_set* and *negated_symbol_set* to *ExpressionTreeNode*

- Add *apply_idempotent_law*, *apply_identity_law*, and *apply_inverse_law* transformations to *ExpressionTreeNode*
- Add *apply_idempotent_law*, *apply_identity_law*, and *apply_inverse_law* top-level transformation functions
- Add functionality to the *coalesce_negations* transformation to apply negations on constant operands
- Update *to_cnf* to incorporate new transformations, leading to more condense CNF transformed expressions

0.6.2

- Remove class `BooleanExpressionTree` in favor of working exclusively with instances of *ExpressionTreeNode*
- Add *AlreadyConstrainedSymbolError*
- Add *sat_all* to *picosat* interface
- Add *constrain*, *sat_one*, and *sat_all* to *BooleanExpression*
- Move the implementation logic of the *to_cnf* transformation to the *to_cnf* method of the *ExpressionTreeNode* class

0.6.1

- Add iff (`iff`, `->`) and implies (`impl`, `<->`) Boolean operators
- Add *is_cnf* and *is_dnf* attributes to *BooleanExpression*
- Add functionality to initialize *BooleanExpression* objects from instances of *ExpressionTreeNode* or `BooleanExpressionTree`
- Update __str__ and __repr__ for *BooleanExpression*
- Add *is_really_unary* attribute to *ExpressionTreeNode*
- Add *iter_clauses*, *iter_cnf_clauses*, and *iter_dnf_clauses* to *ExpressionTreeNode*
- Add *iter_clauses*, *iter_cnf_clauses*, and *iter_dnf_clauses* to *BooleanExpression*
- Add *RequiresNormalFormError*
- Add attributes *default_symbol_str* and *default_plain_english_str* to *BooleanOperator*, in place of removed `name` attribute
- Add *to_primitives*, *coalesce_negations*, *distribute_ands*, *distribute_ors*, and *apply_de_morgans* to *ExpressionTreeNode*
- Introduce high-level *transformations* interface, including transformation functions *to_primitives*, *coalesce_negations*, *distribute_ands*, *distribute_ors*, *to_cnf*, and *apply_de_morgans*
- Add *BINARY_OPERATORS* and *NON_PRIMITIVE_OPERATORS* sets to *definitions* module
- Add `__eq__` and `__ne__` implementations for *BooleanExpression* and derivatives of *ExpressionTreeNode*

0.6.0

- Add *is_valid_identifier* helper method for checking if symbol names are valid
- Add checking of valid symbol names to *BooleanExpression* and *TruthTable* initalization logic, with corresponding new exception type *InvalidIdentifierError*
- Add *boolean_variables_factory* helper for generating more intuitive collections of symbol inputs
- Update __iter__ in *TruthTable* to yield inputs as a `namedtuple`-like object rather than a plain `tuple`
- Re-organize *User Guide* into different sections instead of one long page
- Remove PyPy support, due to addition of C-extensions
- Add OS X builds to Travis
- Include both 32-bit and 64-bit builds on AppVeyor
- Add initial wrapper around PicoSAT library for future satisfiability interface; namely, the *sat_one* method
- Add automated deployment to PyPI on tagged commits from CI services

4.2.2 0.5.x

Features in the 0.5.x series of releases were focused on expanding the top-level interface and improving optimizations under the hood. See below for specific features and fixes.

0.5.1

- Add `from_values` option to the *TruthTable* initializer, allowing for table creation directly from values
- Add ability to store *don't cares* in a *TruthTable*
- Add *equivalent_to* method to *TruthTable* to check for equivalence of sources of truth
- Convert *generate_symbols* and *input_combos* to be static methods of the *TruthTable* class
- Add *is_full* to *TruthTable*
- Add __iter__ and __getitem__ functionality to *TruthTable*
- Add nice-looking __str__ to *BooleanExpression*
- Add new exception types: *AlreadyFullTableError*, *ConflictingArgumentsError*, and *RequiredArgumentError*
- Re-organize exception hierarchy so each group of exceptions extends from the same base class
- Re-organize the test file structure into more-focused files
- Add *User Guide*, acting as tutorial-style documentation
- Remove CLI example from the README
- Update documentation color palette

0.5.0

- Added the Release Notes section to the project's documentation (how fitting for this page)
- Publically exposed the *input_combos* method in the *TruthTable* class
- Added test coverage for the CPython 3.6, PyPy, and PyPy3 runtimes
- Migrated all documentation to from Napoleon docstrings to standard Sphinx docstrings
- Added doctest tests to the documentation
- Added type-checking to the *BooleanExpression* class's initialization
- Fixed a bug in the handling of empty expressions in the CLI

4.2.3 pre-0.5

Unfortunately, release notes were not kept before the 0.5.0 release.

4.3 Development

If you'd like to help out with the development of tt, we'd love to have you. Below are some helpful tips for working on this library. Feel free to *reach out* with any questions about getting involved in this project.

4.3.1 Managing with `ttasks.py`

tt ships with a script `ttasks.py` (tt + tasks = ttasks) in the project's top-level directory, used to manage common project tasks such as running tests, building the docs, and serving the docs via a live-reload server. You will see this script referenced below.

4.3.2 Dependencies

All development requirements for tt are stored in the `dev-requirements.txt` file in the project's top-level directory. You can install all of these dependencies with:

```
pip install -r dev-requirements.txt
```

4.3.3 Testing

Testing is done with Python's unittest and doctest modules. All tests can be run using the `ttasks.py` script:

```
python ttasks.py test
```

Note that while doc tests are used, they are mainly to make sure the documentation examples are valid. The true behavior of the library and its public contract are enforced through the unit tests.

Local cross-Python version testing is achieved through tox. To run changes against the reference and style tests, simply invoke `tox .` from the top-level directory of the project; tox will run the unit tests against the compatible CPython runtimes. Additionally, the source is run through the Flake8 linter. Similar configurations are used on AppVeyor (for Windows builds) and Travis CI. (for Mac and Linux builds).

4.3.4 Coding Style

tt aims to be strictly PEP8 compliant, enforcing this compliance via Flake8. This project also includes an editorconfig file to help with formatting issues.

4.3.5 Documentation

To build the docs from source, run the following:

```
python ttasks.py build-docs
```

If you're going to be working for a little bit, it's usually more convenient to boot up a live-reload server that will re-build the docs on any source file change. To run one on port 5000 of your machine, run:

```
python ttasks.py serve-docs
```

4.3.6 Building C-extensions

tt contains some C-extensions that need to be built before the library is fully usable. They can be built and installed in a development environment by running:

```
python setup.py build
python setup.py develop
```

from the project's top-level directory. There are some dependencies required for compiling these extensions, which can be a little difficult to get up and running on Windows. Depending on what CPython version you are targeting, you may need to install several different compilers. The following list contains information for all entries corresponding to Python versions that are either currently or were once supported by this project:

- Microsoft Visual C++ 9.0 (for Python 2.7)
- Microsoft Visual C++ 10.0 (for Python 3.3 and 3.4)
- Microsoft Visual C++ 14.0 (for Python 3.5, 3.6, 3.7, and 3.8)

For reference, check out this comprehensive list of Windows compilers necessary for building Python and C-extensions. You may have some trouble installing the 7.1 SDK (which contains Visual C++ 10.0). This stackoverflow answer provides some possible solutions.

4.3.7 Releases

Work for each release is done in a branch off of develop following the naming convention v{major}.{minor}.{micro}. When work for a version is complete, its branch is merged back into develop, which is subsequently merged into master. The master branch is then tagged with the release version number, following the scheme {major}.{minor}.{micro}.

Wheels for Windows environments are provided for the library's users on PyPI. To download the built wheels from the latest build on AppVeyor, make sure you have the `APPVEYOR_TOKEN` environment variable set and run:

```
python ttasks.py pull-latest-win-wheels
```

Additionally, when packaging for a release, make sure to include a source bundle:

```
python setup.py sdist
```

Now, all of our wheels and the source tarball should be in the `dist` folder in the top-level directory of the project. You can upload these files to PyPI with:

```
twine upload dist/*
```

4.4 Prior Art

There are some great projects operating in the same problem space as tt and might be worth a look. Many of tt's design and feature choices were inspired by the libraries listed on this page. If you think that your library should be listed here, please let me know or submit a PR.

4.4.1 General purpose EDA/Boolean logic

- boolean.py
- PyEDA
- LogicNG (Java)
- BoolExpr (C++)
- EvalEx (Java)

4.4.2 Satisfiability

- PyEDA
- pycosat
- SATisPy

4.5 Special Thanks

A lot of free services and open source libraries have helped this project become possible. This page aims to give credit where its due; if you were left out, I'm sorry! Please let me know!

4.5.1 Contributors

Thank you to the following people who generously contributed their time and brainpower towards writing code that was merged into the library.

- Thomas Applencourt
- Florian Kromer

4.5.2 Services

Thank you to the free hosting provided by these services!

- GitHub
- Travis CI

- AppVeyor
- Read the Docs

4.5.3 Design Resources

Thank you to Matthew Beckler, who designed the logic gate SVGs present in tt's logo.

4.5.4 Third Party Libraries Shipped with tt

Thank you to the developers of the following third party libraries that are wrapped in and shipped with tt. Your hard work drives some of the most powerful functionality of tt.

- PicoSAT

4.5.5 Inspiration

Thanks goes to the developers of the PyEDA and pycosat libraries, whose interface and design are inspirations behind a lot of the functionality packed in tt.

4.5.6 Open Source Projects & Libraries

tt relies on some well-written and well-documented projects and libraries for its development, listed below. Thank you!

- Alabaster
- Babel
- Colorama
- Docutils
- Flake8
- imagesize
- Jinja2
- MarkupSafe
- McCabe
- pep8
- pluggy
- py
- pyenv
- pyflakes
- Pygments
- Python
- pytz
- Requests

- six
- snowballstemmer
- Sphinx
- tox
- twine
- virtualenv

4.6 Author

tt is written by Brian Welch. If you'd like to discuss anything about this library, Python, or software engineering in general, please feel free to reach out via one of the below channels.

- Personal website
- Github

CHAPTER 5

Want to learn more?

If you're just getting started and looking for tutorial-style documentation, head on over to the *User Guide*. If you would prefer a comprehensive view of this library's functionality, check out the API docs:

5.1 `cli`

tt's command-line interface.

5.1.1 `cli.core` module

Core command-line interface for tt.

`tt.cli.core.`**`get_parsed_args`**(*args=None*)
Get the parsed command line arguments.

Parameters **`args`** (*`List[str], optional`*) – The command-line args to parse; if omitted, `sys.argv` will be used.

Returns The `Namespace` object holding the parsed args.

Return type `argparse.Namespace`

`tt.cli.core.`**`main`**(*args=None*)
The main routine to run the tt command-line interface.

Parameters **`args`** (*`List[str], optional`*) – The command-line arguments.

Returns The exit code of the program.

Return type int

5.1.2 cli.utils module

Utilities for the tt command-line interface.

tt.cli.utils.**print_err**(**args*, ***kwargs*)
A thin wrapper around print, explicitly printing to stderr.

tt.cli.utils.**print_info**(**args*, ***kwargs*)
A thin wrapper around print, explicitly printing to stdout.

5.2 definitions

Definitions for tt's expression grammar, operands, and operators.

5.2.1 definitions.grammar module

Definitions related to expression grammar.

tt.definitions.grammar.**CONSTANT_VALUES = {'0', '1'}**
Set of tokens that act as constant values in expressions.

Type Set[str]

tt.definitions.grammar.**DELIMITERS = {' ', '(', ')'}**
Set of tokens that act as delimiters in expressions.

Type Set[str]

5.2.2 definitions.operands module

Definitions related to operands.

tt.definitions.operands.**BOOLEAN_VALUES = {0, 1}**
Set of truthy values valid to submit for evaluation.

Type Set[int, bool]

tt.definitions.operands.**DONT_CARE_VALUE = 'x'**
The don't care string identifier.

Type str

tt.definitions.operands.**boolean_variables_factory**(*symbols*)
Returns a class for namedtuple-like objects for holding boolean values.

Parameters **symbols** (List[str]) – A list of the symbol names for which instances of this class will hold an entry.

Returns An object where the passed symbols can be accessed as attributes.

Return type namedtuple-like object

This functionality is best demonstrated with an example:

```
>>> from tt import boolean_variables_factory
>>> factory = boolean_variables_factory(['op1', 'op2', 'op3'])
>>> instance = factory(op1=True, op2=False, op3=False)
>>> instance.op1
```

(continues on next page)

(continued from previous page)

```
True
>>> instance.op2
False
>>> print(instance)
op1=1, op2=0, op3=0
>>> instance = factory(op1=0, op2=0, op3=1)
>>> instance.op3
1
>>> print(instance)
op1=0, op2=0, op3=1
```

It should be noted that this function is used internally within functionality where the validity of inputs is already checked. As such, this class won't enforce the Boolean-ness of input values:

```
>>> factory = boolean_variables_factory(['A', 'B'])
>>> instance = factory(A=-1, B='value')
>>> print(instance)
A=-1, B=value
```

Instances produced from the generated factory are descendants of `namedtuple` generated classes; some of the inherited attributes may be useful:

```
>>> instance = factory(A=True, B=False)
>>> instance._fields
('A', 'B')
>>> dict(instance._asdict())
{'A': True, 'B': False}
```

`tt.definitions.operands.`**`is_valid_identifier`**(*identifier_name*)

Returns whether the string is a valid symbol identifier.

Valid identifiers are those that follow Python variable naming conventions, are not Python keywords, and do not begin with an underscore.

Parameters **`identifier_name`** (`str`) – The string to test.

Returns True if the passed string is valid identifier, otherwise False.

Return type `bool`

Raises

- ***`InvalidArgumentTypeError`*** – If `identifier_name` is not a string.
- ***`InvalidArgumentValueError`*** – If `identifier_name` is an empty string.

As an example:

```
>>> from tt import is_valid_identifier
>>> is_valid_identifier('$var')
False
>>> is_valid_identifier('va#r')
False
>>> is_valid_identifier('for')
False
>>> is_valid_identifier('False')
False
>>> is_valid_identifier('var')
True
```

(continues on next page)

(continued from previous page)

```
>>> is_valid_identifier('')
Traceback (most recent call last):
    ...
tt.errors.arguments.InvalidArgumentValueError: identifier_name cannot be empty
>>> is_valid_identifier(None)
Traceback (most recent call last):
    ...
tt.errors.arguments.InvalidArgumentTypeError: identifier_name must be a string
```

5.2.3 `definitions.operators` module

Definitions for tt's built-in Boolean operators.

`tt.definitions.operators.`**`BINARY_OPERATORS = {<BooleanOperator "impl">, <BooleanOperator "x`**

The set of all binary operators available in tt.

Type Set{*BooleanOperator*}

class `tt.definitions.operators.`**`BooleanOperator`**(*precedence*, *eval_func*, *default_symbol_str*, *default_plain_english_str*)

Bases: `object`

A thin wrapper around a Boolean operator.

`__init__`(*precedence*, *eval_func*, *default_symbol_str*, *default_plain_english_str*)

Initialize self. See help(type(self)) for accurate signature.

`__repr__`()

Return repr(self).

`__str__`()

Return str(self).

`default_plain_english_str`

The default plain English string representation of this operator.

Unlike *default_symbol_str*, this attribute should never be `None`.

Type `str`

```
>>> from tt.definitions import TT_AND_OP, TT_NAND_OP
>>> print(TT_AND_OP.default_plain_english_str)
and
>>> print(TT_NAND_OP.default_plain_english_str)
nand
```

`default_symbol_str`

The default symbolic string representation of this operator.

Some operators may not have a recognized symbol str, in which case this attribute will be `None`.

Type `str` or `None`

```
>>> from tt.definitions import TT_AND_OP, TT_NAND_OP
>>> print(TT_AND_OP.default_symbol_str)
/\
>>> print(TT_NAND_OP.default_symbol_str)
None
```

eval_func
The evaluation function wrapped by this operator.

Type Callable

```
>>> from tt.definitions import TT_XOR_OP
>>> TT_XOR_OP.eval_func(0, 0)
False
>>> TT_XOR_OP.eval_func(True, False)
True
```

precedence
Precedence of this operator, relative to other operators.

Type int

```
>>> from tt.definitions import TT_AND_OP, TT_OR_OP
>>> TT_AND_OP.precedence > TT_OR_OP.precedence
True
```

tt.definitions.operators.**MAX_OPERATOR_STR_LEN = 4**
The length of the longest operator from *OPERATOR_MAPPING*.

Type int

tt.definitions.operators.**NON_PRIMITIVE_OPERATORS = {<BooleanOperator "impl">, <BooleanOpera**
The set of non-primitive operators available in tt.

This includes all binary operators other than AND and OR.

Type Set{*BooleanOperator*}

tt.definitions.operators.**OPERATOR_MAPPING = {'!': <BooleanOperator "not">, '&': <BooleanO**
A mapping of all available Boolean operators.

This dictionary is the concatentation of the *PLAIN_ENGLISH_OPERATOR_MAPPING* and *SYMBOLIC_OPERATOR_MAPPING* dictionaries.

Type Dict{str: *BooleanOperator*}

tt.definitions.operators.**PLAIN_ENGLISH_OPERATOR_MAPPING = {'AND': <BooleanOperator "and">,**
A mapping of Boolean operators.

This mapping includes the plain-English variants of the available Boolean operators.

Type Dict{str: *BooleanOperator*}

tt.definitions.operators.**SYMBOLIC_OPERATOR_MAPPING = {'!': <BooleanOperator "not">, '&':**
A mapping of Boolean operators.

This mapping includes the symbolic variants of the available Boolean operators.

Type Dict{str: *BooleanOperator*}

tt.definitions.operators.**TT_AND_OP = <BooleanOperator "and">**
tt's operator implementation of a Boolean AND.

Type *BooleanOperator*

tt.definitions.operators.**TT_IMPL_OP = <BooleanOperator "impl">**
tt's operator implementation of a Boolean IMPLIES.

Type *BooleanOperator*

tt.definitions.operators.**TT_NAND_OP = <BooleanOperator "nand">**
tt's operator implementation of a Boolean NAND.

Type *BooleanOperator*

tt.definitions.operators.**TT_NOR_OP = <BooleanOperator "nor">**
tt's operator implementation of a Boolean NOR.

Type *BooleanOperator*

tt.definitions.operators.**TT_NOT_OP = <BooleanOperator "not">**
tt's operator implementation of a Boolean NOT.

Type *BooleanOperator*

tt.definitions.operators.**TT_OR_OP = <BooleanOperator "or">**
tt's operator implementation of a Boolean OR.

Type *BooleanOperator*

tt.definitions.operators.**TT_XNOR_OP = <BooleanOperator "xnor">**
tt's operator implementation of a Boolean XNOR.

Type *BooleanOperator*

tt.definitions.operators.**TT_XOR_OP = <BooleanOperator "xor">**
tt's operator implementation of a Boolean XOR.

Type *BooleanOperator*

5.3 errors

tt error types.

5.3.1 errors.base module

The base tt exception type.

exception tt.errors.base.**TtError**(*message*, **args*)
Bases: Exception

Base exception type for tt errors. This exception type should be sub-classed and is not meant to be raised explicitly.

__init__(*message*, **args*)
Initialize self. See help(type(self)) for accurate signature.

message
A helpful message intended to be shown to the end user.

Type str

5.3.2 errors.arguments module

Generic exception types.

exception tt.errors.arguments.**ArgumentError**(*message*, **args*)
Bases: *tt.errors.base.TtError*

An exception type for invalid arguments. This exception type should be sub-classed and is not meant to be raised explicitly.

exception tt.errors.arguments.**ConflictingArgumentsError**(*message*, **args*)

Bases: *tt.errors.arguments.ArgumentError*

An exception type for two or more conflicting arguments.

This error type can be seen in action by passing both an expression and a set of values to the *TruthTable* class:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', from_values='1111')
Traceback (most recent call last):
    ...
tt.errors.arguments.ConflictingArgumentsError: `expr` and `from_values` are␣
↪mutually exclusive arguments
```

exception tt.errors.arguments.**InvalidArgumentTypeError**(*message*, **args*)

Bases: *tt.errors.arguments.ArgumentError*

An exception type for invalid argument types.

To illustrate this error type, let's try passing an invalid argument when creating a *TruthTable*:

```
>>> from tt import TruthTable
>>> t = TruthTable(7)
Traceback (most recent call last):
    ...
tt.errors.arguments.InvalidArgumentTypeError: Arg `expr` must be of type `str` or␣
↪`BooleanExpression`
```

exception tt.errors.arguments.**InvalidArgumentValueError**(*message*, **args*)

Bases: *tt.errors.arguments.ArgumentError*

An exception type for invalid argument values.

Here's an example where we pass a non-power of 2 number of values when attempting to create a *TruthTable*:

```
>>> from tt import TruthTable
>>> t = TruthTable(from_values='01x')
Traceback (most recent call last):
    ...
tt.errors.arguments.InvalidArgumentValueError: Must specify a number of input␣
↪values that is a power of 2
```

exception tt.errors.arguments.**RequiredArgumentError**(*message*, **args*)

Bases: *tt.errors.arguments.ArgumentError*

An exception for when a required argument is missing.

Let's try an example where we omit all arguments when attempting to make a new *TruthTable* object:

```
>>> from tt import TruthTable
>>> t = TruthTable()
Traceback (most recent call last):
    ...
tt.errors.arguments.RequiredArgumentError: Must specify either `expr` or `from_
↪values`
```

5.3.3 errors.evaluation module

Exception type definitions related to expression evaluation.

exception tt.errors.evaluation.**EvaluationError**(*message*, **args*)

Bases: *tt.errors.base.TtError*

An exception type for errors occurring in expression evaluation. This exception type should be sub-classed and is not meant to be raised explicitly.

exception tt.errors.evaluation.**InvalidBooleanValueError**(*message*, **args*)

Bases: *tt.errors.evaluation.EvaluationError*

An exception for when an invalid truth or don't care value is passed.

Here's an example where we attempt to evaluate a *BooleanExpression* with an invalid value passed through `kwargs`:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or B')
>>> b.evaluate(A=1, B='brian')
Traceback (most recent call last):
    ...
tt.errors.evaluation.InvalidBooleanValueError: "brian" passed as value for "B" is␣
↪not a valid Boolean value
```

exception tt.errors.evaluation.**NoEvaluationVariationError**(*message*, **args*)

Bases: *tt.errors.evaluation.EvaluationError*

An exception type for when evaluation of an expression will not vary.

Let's see an example where we attempt to make a *TruthTable* from an expression that has no symbols nor variation in its results:

```
>>> from tt import TruthTable
>>> t = TruthTable('1 or 0')
Traceback (most recent call last):
    ...
tt.errors.evaluation.NoEvaluationVariationError: This expression is composed only␣
↪of constant values
```

5.3.4 errors.grammar module

Exception type definitions related to expression grammar and parsing.

exception tt.errors.grammar.**BadParenPositionError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.grammar.GrammarError*

An exception type for unexpected parentheses.

Here's a quick and dirty example:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or B (')
Traceback (most recent call last):
    ...
tt.errors.grammar.BadParenPositionError: Unexpected parenthesis
```

exception tt.errors.grammar.**EmptyExpressionError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.grammar.GrammarError*

An exception type for when an empty expression is received.

Let's take a brief look:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('')
Traceback (most recent call last):
    ...
tt.errors.grammar.EmptyExpressionError: Empty expression is invalid
```

exception tt.errors.grammar.**ExpressionOrderError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.grammar.GrammarError*

An exception type for unexpected operands or operators.

Here's an example with an unexpected operator:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or or B')
Traceback (most recent call last):
    ...
tt.errors.grammar.ExpressionOrderError: Unexpected binary operator "or"
```

exception tt.errors.grammar.**GrammarError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.base.TtError*

Base type for errors that occur in the handling of expression. This exception type should be sub-classed and is not meant to be raised explicitly.

__init__(*message*, *expr_str=None*, *error_pos=None*, **args*)
Initialize self. See help(type(self)) for accurate signature.

error_pos
The position in the expression where the error occurred.

If this property is left as `None`, it can be assumed that there is no specific location in the expression causing the exception.

Type `int`

expr_str
The expression in which the exception occurred.

If this property is left as `None`, the expression will not be propagated with the exception.

Type `str`

exception tt.errors.grammar.**InvalidIdentifierError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.grammar.GrammarError*

An exception type for invalid operand names. Invalid operand names are determined via the *is_valid_identifier* function.

Here are a couple of examples, for both expressions and tables:

```
>>> from tt import BooleanExpression, TruthTable
>>> b = BooleanExpression('__A xor B')
Traceback (most recent call last):
    ...
tt.errors.grammar.InvalidIdentifierError: Invalid operand name "__A"
>>> t = TruthTable(from_values='0x11', ordering=['for', 'operand'])
Traceback (most recent call last):
    ...
tt.errors.grammar.InvalidIdentifierError: "for" in ordering is not a     valid␣
↪symbol name
```

exception tt.errors.grammar.**UnbalancedParenError**(*message*, *expr_str=None*, *error_pos=None*, **args*)

Bases: *tt.errors.grammar.GrammarError*

An exception type for unbalanced parentheses.

Here's a short example:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or B or C)')
Traceback (most recent call last):
    ...
tt.errors.grammar.UnbalancedParenError: Unbalanced parenthesis
```

5.3.5 errors.state module

Exception type definitions related to invalid operations based on state.

exception tt.errors.state.**AlreadyConstrainedSymbolError**(*message*, **args*)

Bases: *tt.errors.state.StateError*

An exception to be raised when trying to doubly constrain a symbol.

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or B or C')
>>> with b.constrain(C=1):
...     with b.constrain(C=0):
...         pass
...
Traceback (most recent call last):
tt.errors.state.AlreadyConstrainedSymbolError: Symbol "C" cannot be constrained␣
↪multiple times
```

exception tt.errors.state.**AlreadyFullTableError**(*message*, **args*)

Bases: *tt.errors.state.StateError*

An exception to be raised when attempting to fill an already-full table.

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> t.fill()
>>> t.is_full
True
>>> t.fill()
Traceback (most recent call last):
tt.errors.state.AlreadyFullTableError: Cannot fill an already-full table
```

exception tt.errors.state.**RequiresFullTableError**(*message*, **args*)
Bases: *tt.errors.state.StateError*

An exception to be raised when a full table is required.

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> t.equivalent_to('A or B')
Traceback (most recent call last):
tt.errors.state.RequiresFullTableError: Equivalence can only be checked on full␣
↪truth tables
```

exception tt.errors.state.**RequiresNormalFormError**(*message*, **args*)
Bases: *tt.errors.state.StateError*

An exception to be raised when expression normal form is required.

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A nand (B or C)')
>>> b.is_cnf or b.is_dnf
False
>>> for clause in b.iter_clauses():
...     print(clause)
...
Traceback (most recent call last):
tt.errors.state.RequiresNormalFormError: Must be in conjunctive or disjunctive␣
↪normal form to iterate clauses
```

exception tt.errors.state.**StateError**(*message*, **args*)
Bases: *tt.errors.base.TtError*

Base exception type for errors involving invalid state.

5.3.6 errors.symbols module

Exception types related to symbol processing.

exception tt.errors.symbols.**DuplicateSymbolError**(*message*, **args*)
Bases: *tt.errors.symbols.SymbolError*

An exception type for user-specified duplicate symbols.

Here's an example where we try to pass duplicate symbols to the `ordering` property of the *TruthTable* class:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', ordering=['A', 'A', 'B'])
Traceback (most recent call last):
    ...
tt.errors.symbols.DuplicateSymbolError: Received duplicate symbols
```

exception tt.errors.symbols.**ExtraSymbolError**(*message*, **args*)
Bases: *tt.errors.symbols.SymbolError*

An exception for a passed token that is not a parsed symbol.

Here's a quick table example:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', ordering=['A', 'B', 'C'])
Traceback (most recent call last):
    ...
tt.errors.symbols.ExtraSymbolError: Received unexpected symbols: "C"
```

exception tt.errors.symbols.**MissingSymbolError**(*message*, **args*)

Bases: *tt.errors.symbols.SymbolError*

An exception type for a missing token value in evaluation.

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A and B')
>>> b.evaluate(A=1)
Traceback (most recent call last):
    ...
tt.errors.symbols.MissingSymbolError: Did not receive value for the following␣
↪symbols: "B"
```

exception tt.errors.symbols.**SymbolError**(*message*, **args*)

Bases: *tt.errors.base.TtError*

An exception for errors occurring in symbol processing. This exception type should be sub-classed and is not meant to be raised explicitly.

5.4 expressions

Tools for working with Boolean expressions.

5.4.1 expressions.bexpr module

Tools for interacting with Boolean expressions.

class tt.expressions.bexpr.**BooleanExpression**(*expr*)

Bases: object

An interface for interacting with a Boolean expression.

Instances of BooleanExpression are meant to be immutable and can be instantiated from a few different representations of expressions. The simplest way to make an expression object is from a string:

```
>>> from tt import BooleanExpression
>>> BooleanExpression('(A or B) iff (C and D)')
<BooleanExpression "(A or B) iff (C and D)">
```

If you already have an instance of *ExpressionTreeNode* laying around, you can make a new expression object from that, too:

```
>>> from tt import ExpressionTreeNode
>>> tree_root = ExpressionTreeNode.build_tree(
...     ['A', 'B', 'or',
...      'C', 'D', 'and',
...      'iff'])
>>> BooleanExpression(tree_root)
<BooleanExpression "(A or B) iff (C and D)">
```

Additionally, any sub-tree node can be used to build an expression object. Continuing from above, let's make a new expression object for each of the sub-expressions wrapped in parentheses:

```
>>> BooleanExpression(tree_root.l_child)
<BooleanExpression "A or B">
>>> BooleanExpression(tree_root.r_child)
<BooleanExpression "C and D">
```

Expressions also implement the equality and inequality operators (== and !=). Equality is determined by the same semantic structure and the same operand names; the string used to represent the operators in two expressions is not taken into account. Here's a few examples:

```
>>> from tt import BooleanExpression as be
>>> be('A or B or C') == be('A or B or C')
True
>>> be('A or B or C') == be('A || B || C')
True
>>> be('A or B or C') == be('A or C or B')
False
```

Parameters **expr** (*str* or *ExpressionTreeNode*) – The expression representation from which this object is derived.

Raises

- ***BadParenPositionError*** – If the passed expression contains a parenthesis in an invalid position.
- ***EmptyExpressionError*** – If the passed expressions contains nothing other than whitespace.
- ***ExpressionOrderError*** – If the expression contains invalid consecutive operators or operands.
- ***InvalidArgumentTypeError*** – If `expr` is not an acceptable type.
- ***InvalidIdentifierError*** – If any parsed variable symbols in the expression are invalid identifiers.
- ***UnbalancedParenError*** – If any parenthesis pairs remain unbalanced.

It is important to note that aside from *InvalidArgumentTypeError*, all exceptions raised in expression initialization will be descendants of *GrammarError*.

__eq__(*other*)
Return self==value.

__init__(*expr*)
Initialize self. See help(type(self)) for accurate signature.

__ne__(*other*)
Return self!=value.

__repr__()
Return repr(self).

__str__()
Return str(self).

constrain(***kwargs*)
A context manager to impose satisfiability constraints.

This is the interface for adding assumptions to the satisfiability solving functionality provided through the *sat_one()* and *sat_all()* methods.

It should be noted that this context manager is only designed to work with the satisfiability-related functionality of this class. Constrained symbol values will not have an effect on non-sat methods of this class. For example:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A or B) and (C or D)')
>>> with b.constrain(A=1):
...     b.evaluate(A=0, B=1, C=1, D=0)
...
True
```

This context manager returns a reference to the same object upon which it is called. This behavior is designed with the following use case in mind:

```
>>> from tt import BooleanExpression
>>> with BooleanExpression('A or B').constrain(A=1, B=0) as b:
...     b.sat_one()
...
<BooleanValues [A=1, B=0]>
```

Parameters **kwargs** – Keys are names of symbols in this expression; the specified value for each of these keys will be added to the `constraints` attribute of this object for the duration of the context manager.

Returns A reference to the same object that called this method (i.e., `self` in the context of this method).

Return type *BooleanExpression*

Raises

- ***AlreadyConstrainedSymbolError*** – If trying to constrain this expression with multiple context managers.
- ***ExtraSymbolError*** – If a symbol not in this expression is passed through `kwargs`.
- ***InvalidArgumentValueError*** – If no contraints are specified (i.e., `kwargs` is empty).
- ***InvalidBooleanValueError*** – If any values from `kwargs` are not valid Boolean inputs.

evaluate(***kwargs*)

Evaluate the Boolean expression for the passed keyword arguments.

This is a checked wrapper around the *evaluate_unchecked()* function.

Parameters **kwargs** – Keys are names of symbols in this expression; the specified value for each of these keys will be substituted into the expression for evaluation.

Returns The result of evaluating the expression.

Return type `bool`

Raises

- ***ExtraSymbolError*** – If a symbol not in this expression is passed through `kwargs`.

- ***MissingSymbolError*** – If any symbols in this expression are not passed through `kwargs`.
- ***InvalidBooleanValueError*** – If any values from `kwargs` are not valid Boolean inputs.
- ***InvalidIdentifierError*** – If any symbol names are invalid identifiers.

Usage:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A or B')
>>> b.evaluate(A=0, B=0)
False
>>> b.evaluate(A=1, B=0)
True
```

evaluate_unchecked(***kwargs*)

Evaluate the Boolean expression without checking the input.

This is used for evaluation by the *evaluate()* method, which validates the input `kwargs` before passing them to this method.

Parameters **kwargs** – Keys are names of symbols in this expression; the specified value for each of these keys will be substituted into the expression for evaluation.

Returns The Boolean result of evaluating the expression.

Return type `bool`

is_cnf

Whether this expression is in conjunctive norma form or not.

Type `bool`

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A or ~B) and (~C or D or E) and F')
>>> b.is_cnf
True
>>> b = BooleanExpression('A nand B')
>>> b.is_cnf
False
```

is_dnf

Whether this expression is in conjunctive normal form or not.

Type `bool`

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A and B) or (~C and D)')
>>> b.is_dnf
True
>>> b = BooleanExpression('(op1 or !op2) and (op3 or op4)')
>>> b.is_dnf
False
```

iter_clauses()

Iterate over the clauses in this expression.

An expression must be in conjunctive normal form (CNF) or disjunctive normal form (DNF) in order to iterate over its clauses. Here's a simple example:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(~A or B) and (C or D) and (~E or ~F)')
>>> for clause in b.iter_clauses():
...     clause
...
<BooleanExpression "~A or B">
<BooleanExpression "C or D">
<BooleanExpression "~E or ~F">
```

In the case of an ambiguous expression form (between CNF and DNF), the clauses will be interpreted to be in CNF form. For example:

```
>>> b = BooleanExpression('A and ~B and C')
>>> b.is_cnf
True
>>> b.is_dnf
True
>>> print(', '.join(str(clause) for clause in b.iter_clauses()))
A, ~B, C
```

If you want to enforce a specific CNF or DNF interpretation of the clauses, take a look at *iter_cnf_clauses()* and *iter_dnf_clauses()*.

Returns An iterator of expression objects, each representing a separate clause of this expression.

Return type Iterator[*BooleanExpression*]

Raises ***RequiresNormalFormError*** – If this expression is not in conjunctive or disjunctive normal form.

iter_cnf_clauses()

Iterate over the CNF clauses in this expression.

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A or B) and ~C')
>>> for clause in b.iter_cnf_clauses():
...     print(clause)
...
A or B
~C
```

Returns An iterator of expression objects, each representing a separate CNF clause of this expression.

Return type Iterator[*BooleanExpression*]

Raises ***RequiresNormalFormError*** – If this expression is not in conjunctive normal form.

iter_dnf_clauses()

Iterate over the DNF clauses in this expression.

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A and ~B) or (C and D and E)')
>>> for clause in b.iter_dnf_clauses():
...     print(clause)
...
A and ~B
C and D and E
```

Returns An iterator of expression objects, each representing a separate DNF clause of this expression.

Return type Iterator[*BooleanExpression*]

Raises ***RequiresNormalFormError*** – If this expression is not in disjunctive normal form.

postfix_tokens

Similar to the `tokens` attribute, but in postfix order.

Type List[`str`]

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A xor (B or C)')
>>> b.postfix_tokens
['A', 'B', 'C', 'or', 'xor']
```

raw_expr

The raw string expression, parsed upon initialization.

This is what you pass into the `BooleanExpression` constructor; it is kept on the object as an attribute for convenience.

Type `str`

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A nand B')
>>> b.raw_expr
'A nand B'
```

sat_all()

Find all combinations of inputs that satisfy this expression.

Under the hood, this method is using the functionality exposed in tt's *satisfiability.picosat* module.

Here's a simple example of iterating through a few SAT solutions:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('(A xor B) and (C xor D)')
>>> for solution in b.sat_all():
...     print(solution)
...
A=1, B=0, C=1, D=0
A=1, B=0, C=0, D=1
A=0, B=1, C=0, D=1
A=0, B=1, C=1, D=0
```

We can also constrain away a few of those solutions:

```
>>> with b.constrain(A=1, C=0):
...     for solution in b.sat_all():
...         print(solution)
...
A=1, B=0, C=0, D=1
```

Returns An iterator of `namedtuple`-like objects representing satisfying combinations of inputs; if no satisfying solutions exist, the iterator will be empty.

Return type Iterator[`namedtuple` -like objects]

Raises *NoEvaluationVariationError* – If this is an expression of only constants.

sat_one()

Find a combination of inputs that satisfies this expression.

Under the hood, this method is using the functionality exposed in tt's *satisfiability.picosat* module.

Here's a simple example of satisfying an expression:

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A xor 1')
>>> b.sat_one()
<BooleanValues [A=0]>
```

Don't forget about the utility provided by the *constrain()* context manager:

```
>>> b = BooleanExpression('(A nand B) iff C')
>>> with b.constrain(A=1, C=1):
...     b.sat_one()
...
<BooleanValues [A=1, B=0, C=1]>
```

Finally, here's an example when the expression cannot be satisfied:

```
>>> with BooleanExpression('A xor 1').constrain(A=1) as b:
...     b.sat_one() is None
...
True
```

Returns namedtuple-like object representing a satisfying set of values (see *boolean_variables_factory* for more information about the type of object returned); None will be returned if no satisfiable set of inputs exists.

Return type namedtuple-like object or None

Raises *NoEvaluationVariationError* – If this is an expression of only constants.

symbols

The list of unique symbols present in this expression.

The order of the symbols in this list matches the order of symbol appearance in the original expression.

Type List[str]

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A xor (B or C)')
>>> b.symbols
['A', 'B', 'C']
```

tokens

The parsed, non-whitespace tokens of an expression.

Type List[str]

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A xor (B or C)')
>>> b.tokens
['A', 'xor', '(', 'B', 'or', 'C', ')']
```

tree
The tree node representing the root of the tree of this expression.

Type *ExpressionTreeNode*

```
>>> from tt import BooleanExpression
>>> b = BooleanExpression('A xor (B or C)')
>>> print(b.tree)
xor
`----A
`----or
     `----B
     `----C
```

5.5 satisfiability

Functionality for determining logic satisfiasbility.

5.5.1 satisfiability.picosat module

Python wrapper around the _clibs PicoSAT extension.

tt.satisfiability.picosat.**sat_all**(*clauses*, *assumptions=None*)
Find all solutions that satisfy the specified clauses and assumptions.

This provides a light Python wrapper around the same method in the PicoSAT C-extension. While completely tested and usable, this method is probably not as useful as the interface provided through the *sat_all* method in the *BooleanExpression* class.

Parameters

- **clauses** (List[List[int]]) – CNF (AND of ORs) clauses; positive integers represent non-negated terms and negative integers represent negated terms.
- **assumptions** (List[int]) – Assumed terms; same negation logic from `clauses` applies here. Note that assumptions *cannot* be an empty list; leave it as `None` if there are no assumptions to include.

Returns An iterator of solutions; if no satisfiable solutions exist, the iterator will be empty.

Return type Iterator[List[int]]

Raises

- ***InvalidArgumentTypeError*** – If `clauses` is not a list of lists of ints or `assumptions` is not a list of ints.
- ***InvalidArgumentValueError*** – If any literal ints are equal to zero.

Here's an example showing the basic usage:

```
>>> from tt import picosat
>>> for solution in picosat.sat_all([[1], [2, 3, 4], [2, 3]]):
...     print(solution)
...
[1, 2, 3, 4]
[1, 2, 3, -4]
[1, 2, -3, 4]
```

(continues on next page)

(continued from previous page)

```
[1, 2, -3, -4]
[1, -2, 3, 4]
[1, -2, 3, -4]
```

We can cut down on some of the above solutions by including an assumption:

```
>>> for solution in picosat.sat_all([[1], [2, 3, 4], [2, 3]],
...                                 assumptions=[-3]):
...     print(solution)
...
[1, 2, -3, 4]
[1, 2, -3, -4]
```

tt.satisfiability.picosat.**sat_one**(*clauses*, *assumptions=None*)

Find a solution that satisfies the specified clauses and assumptions.

This provides a light Python wrapper around the same method in the PicoSAT C-extension. While completely tested and usable, this method is probably not as useful as the interface provided through the *sat_one* method in the *BooleanExpression* class.

Parameters

- **clauses** (List[List[int]]) – CNF (AND of ORs) clauses; positive integers represent non-negated terms and negative integers represent negated terms.
- **assumptions** (List[int]) – Assumed terms; same negation logic from clauses applies here. Note that assumptions *cannot* be an empty list; leave it as None if there are no assumptions to include.

Returns If solution is found, a list of ints representing the terms of the solution; otherwise, if no solution found, None.

Return type List[int] or None

Raises

- ***InvalidArgumentTypeError*** – If clauses is not a list of lists of ints or assumptions is not a list of ints.
- ***InvalidArgumentValueError*** – If any literal ints are equal to zero.

Let's look at a simple example with no satisfiable solution:

```
>>> from tt import picosat
>>> picosat.sat_one([[1], [-1]]) is None
True
```

Here's an example where a solution exists:

```
>>> picosat.sat_one([[1, 2, 3], [-2, -3], [1, -2], [2, -3], [-2]])
[1, -2, -3]
```

Finally, here's an example using assumptions:

```
>>> picosat.sat_one([[1, 2, 3], [2, 3]], assumptions=[-1, -3])
[-1, 2, -3]
```

5.6 tables

Tools for working with truth tables.

5.6.1 tables.truth_table module

Implementation of a truth table.

class tt.tables.truth_table.**TruthTable**(*expr=None*, *from_values=None*, *fill_all=True*, *ordering=None*)

Bases: object

A class representing a truth table.

There are two ways to fill a table: either populated from an expression or by specifying the values yourself.

An existing BooleanExpression expression can be used, or you can just pass in a string:

```
>>> from tt import TruthTable
>>> t = TruthTable('A xor B')
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
```

When manually specifying the values tt can generate the symbols for you:

```
>>> from tt import TruthTable
>>> t = TruthTable(from_values='0110')
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
```

You can also specify the symbol names yourself, if you'd like:

```
>>> from tt import TruthTable
>>> t = TruthTable(from_values='0110', ordering=['tt', 'rocks'])
>>> print(t)
+----+-------+---+
| tt | rocks |   |
```

(continues on next page)

(continued from previous page)

```
+---+-------+---+
| 0 |   0   | 0 |
+---+-------+---+
| 0 |   1   | 1 |
+---+-------+---+
| 1 |   0   | 1 |
+---+-------+---+
| 1 |   1   | 0 |
+---+-------+---+
```

Parameters

- **expr** (str or *BooleanExpression*) – The expression with which to populate this truth table. If this argument is omitted, then the from_values argument must be properly set.
- **from_values** (str) – A string of 1's, 0's, and x's representing the values to be stored in the table; the length of this string must be a power of 2 and is the complete set of values (in sequential order) to be stored in table.
- **fill_all** (bool, optional) – A flag indicating whether the entirety of the table should be filled on initialization; defaults to True.
- **ordering** (List[str], optional) – An input that maps to this class's *ordering* property. If omitted, the ordering of symbols in the table will match that of the symbols' appearance in the original expression.

Raises

- ***ConflictingArgumentsError*** – If both expr and from_values are specified in the initalization; a table can only be instantiated from one or the other.
- ***DuplicateSymbolError*** – If multiple symbols of the same name are passed into the ordering list.
- ***ExtraSymbolError*** – If a symbol not present in the expression is passed into the ordering list.
- ***MissingSymbolError*** – If a symbol present in the expression is omitted from the ordering list.
- ***InvalidArgumentTypeError*** – If an unexpected parameter type is encountered.
- ***InvalidArgumentValueError*** – If the number of values specified via from_values is not a power of 2 or the ordering list (when filling the table using from_values) is empty.
- ***InvalidIdentifierError*** – If any symbol names specified in ordering are not valid identifiers.
- ***NoEvaluationVariationError*** – If an expression without any unqiue symbols (i.e., one merely composed of constant operands) is specified.
- ***RequiredArgumentError*** – If neither the expr or from_values arguments are specified.

__init__(*expr=None*, *from_values=None*, *fill_all=True*, *ordering=None*)
Initialize self. See help(type(self)) for accurate signature.

__str__()
Return str(self).

equivalent_to(*other*)

Return whether this table is equivalent to another source of truth.

Parameters **other** (*TruthTable*, str, or *BooleanExpression*) – The other source of truth with which to compare logical equivalence.

Returns True if the other expression is logically equivalent to this one, otherwise False.

Return type bool

Raises

- ***InvalidArgumentTypeError*** – If the other argument is not one of the acceptable types.
- ***RequiresFullTableError*** – If either the calling table or other source of truth represents an unfilled table.

It is important to note that the concept of equivalence employed here is only concerned with the corresponding outputs between this table and the other provided source of truth. For example, the ordering of symbols is not taken into consideration when computing equivalence:

```
>>> from tt import TruthTable
>>> t1 = TruthTable('op1 or op2')
>>> t2 = TruthTable('A or B')
>>> t1.equivalent_to(t2)
True
>>> t2.equivalent_to(t1)
True
```

Another area of possible ambiguity here is the role of the don't care value in equivalence. When comparing tables, don't cares in the caller will allow for any corresponding value in other, but the reverse is not true. For example:

```
>>> from tt import TruthTable
>>> t1 = TruthTable(from_values='0x11')
>>> t2 = TruthTable(from_values='0011')
>>> t1.equivalent_to(t2)
True
>>> t2.equivalent_to(t1)
False
```

Additionally, only full tables are valid for equivalence checks. The appropriate error will be raised if you attempt to check the equivalence of partially filled tables:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> t.fill(A=0)
>>> try:
...     t.equivalent_to('A or B')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.state.RequiresFullTableError'>
```

expr

The BooleanExpression object represented by this table.

This attribute will be None if this table was not derived from an expression (i.e., the user provided the values).

Type *BooleanExpression*

fill(***kwargs*)

Fill the table with results, based on values specified by kwargs.

Parameters **kwargs** – Filter which entries in the table are filled by specifying symbol values through the keyword args.

Raises

- ***AlreadyFullTableError*** – If the table is already full when this method is called.
- ***ExtraSymbolError*** – If a symbol not in the expression is passed as a keyword arg.
- ***InvalidBooleanValueError*** – If a non-Boolean value is passed as a value for one of the keyword args.

An example of iteratively filling a table:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> print(t)
Empty!
>>> t.fill(A=0)
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
>>> t.fill(A=1)
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+
```

static generate_symbols(*num_symbols*)

Generate a list of symbols for a specified number of symbols.

Generated symbol names are permutations of a properly-sized number of uppercase alphabet letters.

Parameters **num_symbols** (*int*) – The number of symbols to generate.

Returns A list of strings of length *num_symbols*, containing auto-generated symbols.

Return type List[*str*]

A simple example:

```
>>> from tt import TruthTable
>>> TruthTable.generate_symbols(3)
['A', 'B', 'C']
```

(continues on next page)

(continued from previous page)

```
>>> TruthTable.generate_symbols(7)
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

static input_combos(*combo_len*)

Get an iterator of Boolean input combinations for this expression.

Parameters **combo_len** (int, optional) – The length of each combination in the returned iterator.

Returns An iterator of tuples containing permutations of Boolean inputs.

Return type itertools.product

A simple example:

```
>>> from tt import TruthTable
>>> for tup in TruthTable.input_combos(2):
...     print(tup)
...
(False, False)
(False, True)
(True, False)
(True, True)
```

is_full

A Boolean flag indicating whether this table is full or not.

Type bool

Attempting to further fill an already-full table will raise an *AlreadyFullTableError*:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> t.is_full
False
>>> t.fill()
>>> t.is_full
True
>>> try:
...     t.fill()
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.state.AlreadyFullTableError'>
```

ordering

The order in which the symbols should appear in the truth table.

Type List[str]

Here's a short example of alternative orderings of a partially-filled, three-symbol table:

```
>>> from tt import TruthTable
>>> t = TruthTable('(A or B) and C', fill_all=False)
>>> t.fill(A=0, B=0)
>>> print(t)
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

(continues on next page)

(continued from previous page)

```
| 0 | 0 | 0 | 0 |
+---+---+---+---+
| 0 | 0 | 1 | 0 |
+---+---+---+---+
>>> t = TruthTable('(A or B) and C',
...                fill_all=False, ordering=['C', 'B', 'A'])
>>> t.fill(A=0, B=0)
>>> print(t)
+---+---+---+---+
| C | B | A |   |
+---+---+---+---+
| 0 | 0 | 0 | 0 |
+---+---+---+---+
| 1 | 0 | 0 | 0 |
+---+---+---+---+
```

results

A list containing the results of each possible set of inputs.

Type List[bool, str]

In the case that the table is not completely filled, spots in this list that do not yet have a computed result will hold the None value.

Regardless of the filled status of this table, all positions in the results list are allocated at initialization and subsequently filled as computed. This is illustrated in the below example:

```
>>> from tt import TruthTable
>>> t = TruthTable('A or B', fill_all=False)
>>> t.results
[None, None, None, None]
>>> t.fill(A=0)
>>> t.results
[False, True, None, None]
>>> t.fill()
>>> t.results
[False, True, True, True]
```

If the table is filled upon initialization via the from_values parameter, don't care strings could be present in the result list:

```
>>> from tt import TruthTable
>>> t = TruthTable(from_values='1xx0')
>>> t.results
[True, 'x', 'x', False]
```

5.7 transformations

Interfaces for transforming representations of expressions.

5.7.1 transformations.bexpr module

Transformation functions for expressions.

tt.transformations.bexpr.**apply_de_morgans**(*expr*)

Convert an expression to a form with De Morgan's Law applied.

Returns A new expression object, transformed so that De Morgan's Law has been applied to negated *ANDs* and *ORs*.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

Here's a couple of simple examples showing De Morgan's Law being applied to a negated AND and a negated OR:

```
>>> from tt import apply_de_morgans
>>> apply_de_morgans('~(A /\ B)')
<BooleanExpression "~A \/ ~B">
>>> apply_de_morgans('~(A \/ B)')
<BooleanExpression "~A /\ ~B">
```

tt.transformations.bexpr.**apply_idempotent_law**(*expr*)

Convert an expression to a form with the Idempotent Law applied.

Returns A new expression object, transformed so that the Idempotent Law has been applied to applicable clauses.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid data type.

This transformation will apply the Idempotent Law to clauses of *AND* and *OR* operators containing redundant operands. Here are a couple of simple examples:

```
>>> from tt import apply_idempotent_law
>>> apply_idempotent_law('A and A')
<BooleanExpression "A">
>>> apply_idempotent_law('B or B')
<BooleanExpression "B">
```

This transformation will consider similarly-negated operands to be redundant; for example:

```
>>> from tt import apply_idempotent_law
>>> apply_idempotent_law('~A and ~~~A')
<BooleanExpression "~A">
>>> apply_idempotent_law('B or ~B or ~~B or ~~~B or ~~~~B or ~~~~~B')
<BooleanExpression "B or ~B">
```

Let's also take a quick look at this transformation's ability to prune redundant operands from CNF and DNF clauses:

```
>>> from tt import apply_idempotent_law
>>> apply_idempotent_law('(A and B and C and C and B) or (A and A)')
<BooleanExpression "(A and B and C) or A">
```

Of important note is that this transformation will not recursively apply the Idempotent Law to operands that bubble up. Here's an example illustrating this case:

```
>>> from tt import apply_idempotent_law
>>> apply_idempotent_law('(A or A) and (A or A)')
<BooleanExpression "A and A">
```

tt.transformations.bexpr.**apply_identity_law**(*expr*)
Convert an expression to a form with the Identity Law applied.

It should be noted that this transformation will also annihilate terms when possible. One such case where this would be applicable is the expression `A and 0`, which would be transformed to the constant value `0`.

Returns A new expression object, transformed so that the Identity Law has been applied to applicable *ANDs* and *ORs*.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

Here are a few simple examples showing the behavior of this transformation across all two-operand scenarios:

```
>>> from tt import apply_identity_law
>>> apply_identity_law('A and 1')
<BooleanExpression "A">
>>> apply_identity_law('A and 0')
<BooleanExpression "0">
>>> apply_identity_law('A or 0')
<BooleanExpression "A">
>>> apply_identity_law('A or 1')
<BooleanExpression "1">
```

tt.transformations.bexpr.**apply_inverse_law**(*expr*)
Convert an expression to a form with the Inverse Law applied.

Returns A new expression object, transformed so that the Inverse Law has been applied to applicable *ANDs* and *ORs*.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

This transformation will apply the Identity Law to simple binary expressions consisting of negated and non-negated forms of the same operand. Let's take a look:

```
>>> from tt.transformations import apply_inverse_law
>>> apply_inverse_law('A and ~A')
<BooleanExpression "0">
>>> apply_inverse_law('A or B or ~B or C')
<BooleanExpression "1">
```

This transformation will also apply the behavior expected of the Inverse Law when negated and non-negated forms of the same operand appear in the same CNF or DNF clause in an expression:

```
>>> from tt.transformations import apply_inverse_law
>>> apply_inverse_law('(A or B or ~A) -> (C and ~C)')
<BooleanExpression "1 -> 0">
>>> apply_inverse_law('(A or !!!A) xor (not C or not not C)')
<BooleanExpression "1 xor 1">
```

tt.transformations.bexpr.**coalesce_negations**(*expr*)
Convert an expression to a form with all negations condensed.

Returns A new expression object, transformed so that all "runs" of logical *NOTs* are condensed into the minimal equivalent number.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

Here's a simple example showing the basic premise of this transformation:

```
>>> from tt import coalesce_negations
>>> coalesce_negations('~~A or ~B or ~~~C or ~~~~D')
<BooleanExpression "A or ~B or ~C or D">
```

This transformation works on more complex expressions, too:

```
>>> coalesce_negations('!!(A -> not not B) or ~(~(A xor B))')
<BooleanExpression "(A -> B) or (A xor B)">
```

It should be noted that this transformation will also apply negations to constant operands, as well. The behavior for this functionality is as follows:

```
>>> coalesce_negations('~0')
<BooleanExpression "1">
>>> coalesce_negations('~1')
<BooleanExpression "0">
>>> coalesce_negations('~~~0 -> ~1 -> not 1')
<BooleanExpression "1 -> 0 -> 0">
```

tt.transformations.bexpr.**distribute_ands**(*expr*)

Convert an expression to distribute ANDs over ORed clauses.

Parameters **expr** (`str` or *BooleanExpression*) – The expression to transform.

Returns A new expression object, transformed to distribute ANDs over ORed clauses.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

Here's a couple of simple examples:

```
>>> from tt import distribute_ands
>>> distribute_ands('A and (B or C or D)')
<BooleanExpression "(A and B) or (A and C) or (A and D)">
>>> distribute_ands('(A or B) and C')
<BooleanExpression "(A and C) or (B and C)">
```

And an example involving distributing a sub-expression:

```
>>> distribute_ands('(A and B) and (C or D or E)')
<BooleanExpression "(A and B and C) or (A and B and D) or (A and B and E)">
```

tt.transformations.bexpr.**distribute_ors**(*expr*)

Convert an expression to distribute ORs over ANDed clauses.

Parameters **expr** (`str` or *BooleanExpression*) – The expression to transform.

Returns A new expression object, transformed to distribute ORs over ANDed clauses.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If `expr` is not a valid type.

Here's a couple of simple examples:

```
>>> from tt import distribute_ors
>>> distribute_ors('A or (B and C and D and E)')
<BooleanExpression "(A or B) and (A or C) and (A or D) and (A or E)">
```

(continues on next page)

(continued from previous page)

```
>>> distribute_ors('(A and B) or C')
<BooleanExpression "(A or C) and (B or C)">
```

And an example involving distributing a sub-expression:

```
>>> distribute_ors('(A or B) or (C and D)')
<BooleanExpression "(A or B or C) and (A or B or D)">
```

tt.transformations.bexpr.**to_cnf**(*expr*)

Convert an expression to conjunctive normal form (CNF).

This transformation only guarantees to produce an equivalent form of the passed expression in conjunctive normal form; the transformed expression may be an inefficent representation of the passed expression.

Parameters **expr** (str or *BooleanExpression*) – The expression to transform.

Returns A new expression object, transformed to be in CNF.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If expr is not a valid type.

Here are a few examples:

```
>>> from tt import to_cnf
>>> b = to_cnf('(A nor B) impl C')
>>> b
<BooleanExpression "A or B or C">
>>> b.is_cnf
True
>>> b = to_cnf(r'~(~(A /\ B) /\ C /\ D)')
>>> b
<BooleanExpression "(A \/ ~C \/ ~D) /\ (B \/ ~C \/ ~D)">
>>> b.is_cnf
True
```

tt.transformations.bexpr.**to_primitives**(*expr*)

Convert an expression to a form with only primitive operators.

All operators will be transformed equivalent form composed only of the logical AND, OR,and NOT operators. Symbolic operators in the passed expression will remain symbolic in the transformed expression and the same applies for plain English operators.

Parameters **expr** (str or *BooleanExpression*) – The expression to transform.

Returns A new expression object, transformed to contain only primitive operators.

Return type *BooleanExpression*

Raises ***InvalidArgumentTypeError*** – If expr is not a valid type.

Here's a simple transformation of exclusive-or:

```
>>> from tt import to_primitives
>>> to_primitives('A xor B')
<BooleanExpression "(A and not B) or (not A and B)">
```

And another example of if-and-only-if (using symbolic operators):

```
>>> to_primitives('A <-> B')
<BooleanExpression "(A /\ B) \/ (~A /\ ~B)">
```

5.7.2 `transformations.utils` module

Utilities for building more complex transformations.

class tt.transformations.utils.**ComposedTransformation**(*fn*, *next_transformation=None*, *times=1*)

Bases: *tt.transformations.utils.RepeatableAction*

An encapsulation of composed transformation functions.

This class opens up a world of functionality consisting of buildable (i.e., composed) transformation functions. While instances of this class will work when manually initialized by the user, it will likely be easier to compose functions using the *tt_compose()* method from this module.

Transformation functions, held within the `fn` attribute of this class, are intended to be pure functions that both receive and produce an instance of *BooleanExpression*.

When called, instances of this class will repeatedly apply the `fn` callable to the passed argument. The repeated application of the `fn` callable will continue until either the specified number of `times` is met or the callable produces no change to the expression during the transformation.

Let's take a look at a simple example, where all we do is compose two fairly basic transformations:

```
>>> from tt import coalesce_negations, to_primitives, tt_compose
>>> f = tt_compose(to_primitives, coalesce_negations)
>>> f
<ComposedTransformation [to_primitives -> coalesce_negations]>
>>> to_primitives('~A <-> ~B')
<BooleanExpression "(~A /\ ~B) \/ (~~A /\ ~~B)">
>>> f('~A <-> ~B')
<BooleanExpression "(~A /\ ~B) \/ (A /\ B)">
```

This fairly simple example gives us an idea of how to compose functions using the *tt_compose()* helper. A few operators make manual composition of instances of this class a little more intuitive, too. Let's take a look at how we would make the same composition from above using the >> operator:

```
>>> from tt import ComposedTransformation
>>> one = ComposedTransformation(to_primitives)
>>> two = ComposedTransformation(coalesce_negations)
>>> one >> two
<ComposedTransformation [to_primitives -> coalesce_negations]>
```

The >> and << operators shown above are just shallow wrappers around the core *compose* function.

It is important to note that instances of this class are immutable and hashable; consequently, they support == and != equality checks. We can see this by continuing our example from above:

```
>>> three = ComposedTransformation(to_primitives)
>>> one == two
False
>>> one == three
True
>>> two == three
False
```

The hash of instances of this class is computed at initialization and never updated, so meddling with `ComposedTransformation` instances will likely have unintended consequences for you.

Parameters

- **fn** (Callable) – The callable transformation function wrapped by this class.
- **next_transformation** (*ComposedTransformation*) – The next transformation in the constructed composed sequence of transformation functions.
- **times** (Typically an int) – The number of times the wrapped function is to be repeatedly applied to its passed argument when called.

Raises

- ***InvalidArgumentTypeError*** – If the passed fn argument is not a callable.
- ***InvalidArgumentValueError*** – If times is not valid, as per the *RepeatableAction* initialization logic.

__call__(*expr*)
Call self as a function.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

__init__(*fn*, *next_transformation=None*, *times=1*)
Initialize self. See help(type(self)) for accurate signature.

__ne__(*other*)
Return self!=value.

__repr__()
Return repr(self).

__str__()
Return str(self).

compose(*other*)
Compose this transformation with another.

Parameters **other** (A Callable, instance of *ComposedTransformation*, or instance of AbstractTransformationModifier.) – The callable transformation function, composed transformation object, or modifier object to either be composed with or modify this object.

Returns A new composed transformation instance, with the intended composition or modification applied.

Return type *ComposedTransformation*

Raises **InvalidArgumentType** – If the other argument is not of an expected type.

fn
The callable transformation function that this class wraps.

This callable should both accept as an argument and produce as its result an instance of the *BooleanExpression* class.

Type Callable

```
>>> from tt import tt_compose, apply_de_morgans, twice
>>> f = tt_compose(apply_de_morgans, twice)
>>> f.fn.__name__
'apply_de_morgans'
```

next_transformation
The next transformation that this object's result will be passed to.

The next transformation function in the chain of composed functions. A value of `None` indicates that this is the last function in the composition.

Type *ComposedTransformation*

class `tt.transformations.utils.`**RepeatableAction**(*times=1*)
Bases: `object`

A mixin for describing actions that can be repeated.

This class is meant to be used as a mixin when simple access to a `times` attribute is needed, presumably to perform some action or task multiple times. Here's a simple look at the class:

```
>>> from tt import RepeatableAction
>>> r = RepeatableAction(5)
>>> print(r)
5 times
>>> r
<RepeatableAction [5 times]>
>>> r.times
5
```

The passed `times` argument to this class must be a value that implements `__lt__` that is not less than 1. Here's an example:

```
>>> r = RepeatableAction(-1)
Traceback (most recent call last):
    ...
tt.errors.arguments.InvalidArgumentValueError: `times` must be at least 1
```

Instances of *RepeatableAction* are immutable, hashable, and implement all rich comparison operators. Let's take a look:

```
>>> r1, r2 = RepeatableAction(3), RepeatableAction(4)
>>> hash(r1)
3
>>> hash(r2)
4
>>> r1 < r2
True
>>> r1 == r2
False
>>> r1 > r2
False
>>> r3 = RepeatableAction(4)
>>> r2 == r3
True
```

Parameters **times** (Typically an `int`) – The number of times that this action would be repeated when executed.

Raises ***InvalidArgumentValueError*** – If `times` is less than 1.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

__init__(*times=1*)
Initialize self. See help(type(self)) for accurate signature.

__lt__(*other*)
Return self<value.

__repr__()
Return repr(self).

__str__()
Return str(self).

times
The number of times the action is to be repeated.

Type int

```
>>> from tt import RepeatableAction
>>> r = RepeatableAction(3)
>>> r.times
3
>>> r = RepeatableAction(float('inf'))
>>> r.times
inf
```

tt.transformations.utils.**ensure_bexpr**(*expr*)
Return an expression object or raise an InvalidArgumentTypeError.

Parameters **expr** (*BooleanExpression* or str) – The expression whose type is being checked.

Raises ***InvalidArgumentTypeError*** – If expr is not of a valid type.

tt.transformations.utils.**forever = <RepeatableAction [inf times]>**
A repeating modifier to perform a transformation forever.

Type *repeat*

class tt.transformations.utils.**repeat**(*times*)
Bases: tt.transformations.utils.AbstractTransformationModifier, *tt.transformations.utils.RepeatableAction*

Factory for a repeating transformation modifier.

This factory method is largely meant to provide repeating modifier for the *tt_compose()* function. As an example, let's compose a transformation that will be applied 7 times to expressions passed to it:

```
>>> from tt import tt_compose, coalesce_negations, repeat
>>> tt_compose(coalesce_negations, repeat(7))
<ComposedTransformation [coalesce_negations (7 times)]>
```

Check out the *twice* and *forever* modifiers for some pre-made utilities that may come in handy.

__init__(*times*)
Initialize self. See help(type(self)) for accurate signature.

modify(*other*)
Modify a transformation composition or other modifier.

This method must be implemented by descendants of this class.

Parameters **other** (*ComposedTransformation* or AbstractTransformationModifier) – A transformation composition or

Returns A modified composition or modifier.

Return type The same type as other

tt.transformations.utils.**tt_compose**(**fns*)

Compose multiple transformations into a new callable transformation.

This function will compose multiple transformations and transformation modifiers into a single callable. When called, this new transformation will apply the composition to generate a transformed expression.

Parameters **fns** (Callable, *ComposedTransformation*, or AbstractTransformationModifier) – A sequence of callable transformation functions or transformation modifiers from which a single composed transformation will be constructed.

Returns The callable composition of all functions in fn, which will return a *BooleanExpression* object when called.

Return type Callable

Raises

- ***InvalidArgumentTypeError*** – If a modifier is ordered incorrectly or a non-callable function is included in the sequence.
- ***InvalidArgumentValueError*** – If an insufficient number of arguments is provided (must be at least 2).

Let's say we wanted a transformation that would first convert all operators in our expression to their equivalent primitive form, and then apply De Morgan's Law twice:

```
>>> from tt.transformations import *
>>> f = tt_compose(
...     to_primitives,
...     apply_de_morgans, twice
... )
>>> f
<ComposedTransformation [to_primitives -> apply_de_morgans (2 times)]>
>>> f('~A <-> ~B')
<BooleanExpression "(~A /\ ~B) \/ (~~A /\ ~~B)">
```

Composed transformations can be nested, too. Let's add some functionality to our composed transformation so that all redundant negations are coalesced:

```
>>> g = tt_compose(f, coalesce_negations)
>>> g
<ComposedTransformation [to_primitives -> apply_de_morgans (2 times) -> coalesce_
↪negations]>
>>> g('~A <-> ~B')
<BooleanExpression "(~A /\ ~B) \/ (A /\ B)">
```

tt.transformations.utils.**twice = <RepeatableAction [2 times]>**

A repeating modifier to perform a transformation twice.

Type *repeat*

5.8 trees

Tools for working with Boolean expression trees.

It should be noted that virtually all of the functionality within this module is presented with an easier-to-use interface in the *expressions* module.

5.8.1 trees.tree_node module

A node, and related classes, for use in expression trees.

class tt.trees.tree_node.**BinaryOperatorExpressionTreeNode**(*operator_str*, *l_child*, *r_child*)

Bases: *tt.trees.tree_node.ExpressionTreeNode*

An expression tree node for binary operators.

__eq__(*other*)
Return self==value.

__init__(*operator_str*, *l_child*, *r_child*)
Initialize self. See help(type(self)) for accurate signature.

apply_de_morgans()
Return a transformed node, with De Morgan's Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all negated AND and OR operators transformed, following De Morgan's Law.

Return type *ExpressionTreeNode*

apply_idempotent_law()
Returns a transformed node, with the Idempotent Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects

Returns An expression tree node with the Idempotent Law applied to *AND* and *OR* operators.

Return type *ExpressionTreeNode*

This transformation will apply the Idempotent Law to *AND* and *OR* expressions involving repeated operands. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and A').tree
>>> print(tree.apply_idempotent_law())
A
>>> tree = BooleanExpression('~B or ~~~B').tree
>>> print(tree.apply_idempotent_law())
~
`----B
```

In the latter of the two above examples, we see that this transformation will compare operands with negations condensed. This transformation will also prune redundant operands from CNF and DNF clauses. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and B and B and C and ~C and ~~C and D').tree
>>> print(tree.apply_idempotent_law())
and
`----and
|    `----and
|    |    `----and
|    |    |    `----A
|    |    |    `----B
|    |    `----C
|    `----~
|         `----C
`----D
```

apply_identity_law()

Return a transformed node, with the Identity Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

This transformation will achieve the following effects by applying the Inverse Law to the *AND* and *OR* operators:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 1').tree
>>> print(tree.apply_identity_law())
A
>>> tree = BooleanExpression('0 or B').tree
>>> print(tree.apply_identity_law())
B
```

It should also be noted that this transformation will also apply the annihilator properties of the logical *AND* and *OR* operators. For example:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 0').tree
>>> print(tree.apply_identity_law())
0
>>> tree = BooleanExpression('1 or B').tree
>>> print(tree.apply_identity_law())
1
```

Returns An expression tree node with AND and OR identities simplified.

Return type *ExpressionTreeNode*

apply_inverse_law()

Return a transformed node, with the Inverse Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with the Inverse Law applied to applicable clauses.

Return type *ExpressionTreeNode*

This transformation will apply the Inverse Law to *AND* and *OR* expressions involving the negated and non-negated forms of a variable. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('~A and A').tree
>>> print(tree.apply_inverse_law())
0
>>> tree = BooleanExpression('B or !B').tree
>>> print(tree.apply_inverse_law())
1
```

Note that this transformation will **not** reduce expressions of constants; the transformation *apply_identity_law* will probably do what you want in this case, though.

This transformation will also reduce expressions in CNF or DNF that contain negated and non-negated forms of the same symbol. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A or B or C or ~B').tree
>>> print(tree.apply_inverse_law())
1
>>> tree = BooleanExpression('A and B and C and !B').tree
>>> print(tree.apply_inverse_law())
0
```

coalesce_negations()
Return a transformed node, with consecutive negations coalesced.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all consecutive negations compressed into the minimal number of equivalent negations (either one or none).

Return type *ExpressionTreeNode*

distribute_ands()
Return a transformed nodes, with ANDs recursively distributed across ORed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable AND operators distributed across ORed sub-expressions.

Return type *ExpressionTreeNode*

distribute_ors()
Return a transformed nodes, with ORs recursively distributed across ANDed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable OR operators distributed across ANDed sub-expressions.

Return type *ExpressionTreeNode*

evaluate(*input_dict*)
Recursively evaluate this node.

This is an interface that should be defined in sub-classes. Node evaluation does no checking of the validity of inputs; they should be check before being passed here.

Parameters **input_dict** (Dict{str: truthy) – A dictionary mapping expression symbols to the value for which they should be subsituted in expression evaluation.

Returns The evaluation of the tree rooted at this node.

Return type bool

operator
The actual operator object wrapped in this node.

Type *BooleanOperator*

to_primitives()
Return a transformed node, containing only NOTs, ANDs, and ORs.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all operators transformed to consist only of NOTs, ANDs, and ORs.

Return type *ExpressionTreeNode*

class tt.trees.tree_node.**ExpressionTreeNode**(*symbol_name*, *l_child=None*, *r_child=None*)

Bases: object

A base class for expression tree nodes.

This class is extended within tt and is not meant to be used directly.

If you plan to extend it, note that descendants of this class must compute the _is_cnf, _is_dnf, and _is_really_unary boolean attributes and the _non_negated_symbol_set and _negated_symbol_set set attributes within their initialization. Additionally, descendants of this class must implemented the __eq__ magic method (but not __ne__) as well as the private _copy transformation.

__eq__(*other*)
Return self==value.

__init__(*symbol_name*, *l_child=None*, *r_child=None*)
Initialize self. See help(type(self)) for accurate signature.

__ne__(*other*)
Return self!=value.

__str__()
Return str(self).

apply_de_morgans()
Return a transformed node, with De Morgan's Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all negated AND and OR operators transformed, following De Morgan's Law.

Return type *ExpressionTreeNode*

apply_idempotent_law()
Returns a transformed node, with the Idempotent Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects

Returns An expression tree node with the Idempotent Law applied to *AND* and *OR* operators.

Return type *ExpressionTreeNode*

This transformation will apply the Idempotent Law to *AND* and *OR* expressions involving repeated operands. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and A').tree
>>> print(tree.apply_idempotent_law())
A
>>> tree = BooleanExpression('~B or ~~~B').tree
>>> print(tree.apply_idempotent_law())
~
`----B
```

In the latter of the two above examples, we see that this transformation will compare operands with negations condensed. This transformation will also prune redundant operands from CNF and DNF clauses. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and B and B and C and ~C and ~~C and D').tree
>>> print(tree.apply_idempotent_law())
and
`----and
|    `----and
|    |    `----and
|    |    |    `----A
|    |    |    `----B
|    |    `----C
|    `-----~
|         `----C
`----D
```

apply_identity_law()

Return a transformed node, with the Identity Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

This transformation will achieve the following effects by applying the Inverse Law to the *AND* and *OR* operators:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 1').tree
>>> print(tree.apply_identity_law())
A
>>> tree = BooleanExpression('0 or B').tree
>>> print(tree.apply_identity_law())
B
```

It should also be noted that this transformation will also apply the annihilator properties of the logical *AND* and *OR* operators. For example:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 0').tree
>>> print(tree.apply_identity_law())
0
>>> tree = BooleanExpression('1 or B').tree
>>> print(tree.apply_identity_law())
1
```

Returns An expression tree node with AND and OR identities simplified.

Return type *ExpressionTreeNode*

apply_inverse_law()
Return a transformed node, with the Inverse Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with the Inverse Law applied to applicable clauses.

Return type *ExpressionTreeNode*

This transformation will apply the Inverse Law to *AND* and *OR* expressions involving the negated and non-negated forms of a variable. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('~A and A').tree
>>> print(tree.apply_inverse_law())
0
>>> tree = BooleanExpression('B or !B').tree
>>> print(tree.apply_inverse_law())
1
```

Note that this transformation will **not** reduce expressions of constants; the transformation *apply_identity_law* will probably do what you want in this case, though.

This transformation will also reduce expressions in CNF or DNF that contain negated and non-negated forms of the same symbol. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A or B or C or ~B').tree
>>> print(tree.apply_inverse_law())
1
>>> tree = BooleanExpression('A and B and C and !B').tree
>>> print(tree.apply_inverse_law())
0
```

static build_tree(*postfix_tokens*)
Build a tree from a list of expression tokens in postfix order.

This method does not check that the tokens are indeed in postfix order; undefined behavior will ensue if you pass tokens in an order other than postfix.

Parameters **postfix_tokens** (List[str]) – A list of string tokens from which to construct the tree of expression nodes.

Returns The root node of the constructed tree.

Return type *ExpressionTreeNode*

Raises

- ***InvalidArgumentTypeError*** – If `postfix_tokens` is not a list of strings.
- ***InvalidArgumentValueError*** – If `postfix_tokens` is empty.

coalesce_negations()
Return a transformed node, with consecutive negations coalesced.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all consecutive negations compressed into the minimal number of equivalent negations (either one or none).

Return type *ExpressionTreeNode*

distribute_ands()
Return a transformed nodes, with ANDs recursively distributed across ORed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable AND operators distributed across ORed sub-expressions.

Return type *ExpressionTreeNode*

distribute_ors()
Return a transformed nodes, with ORs recursively distributed across ANDed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable OR operators distributed across ANDed sub-expressions.

Return type *ExpressionTreeNode*

evaluate(*input_dict*)
Recursively evaluate this node.

This is an interface that should be defined in sub-classes. Node evaluation does no checking of the validity of inputs; they should be check before being passed here.

Parameters **input_dict** (Dict{str: truthy) – A dictionary mapping expression symbols to the value for which they should be subsituted in expression evaluation.

Returns The evaluation of the tree rooted at this node.

Return type bool

is_cnf
Whether the tree rooted at this node is in conjunctive normal form.

Type bool

is_dnf
Whether the tree rooted at this node is in disjunctive normal form.

Type bool

is_really_unary
Whether the tree rooted at this node contains no binary operators.

Type bool

iter_clauses()
Iterate the clauses in the expression tree rooted at this node.

If the normal form of the expression is ambiguous, then precedence will be given to conjunctive normal form.

Returns Iterator of each CNF or DNF clause, rooted by a tree node, contained within the expression tree rooted at this node.

Return type Iterator[*ExpressionTreeNode*]

Raises ***RequiresNormalFormError*** – If this expression is not in conjunctive or disjunctive normal form.

iter_cnf_clauses()
Iterate the clauses in conjunctive normal form order.

Returns Iterator of each CNF clause, rooted by a tree node, contained within the expression tree rooted at this node.

Return type Iterator[*ExpressionTreeNode*]

Raises ***RequiresNormalFormError*** – If the expression tree rooted at this node is not in conjunctive normal form.

iter_dnf_clauses()
Iterate the clauses in disjunctive normal form order.

Returns Iterator of each DNF clause, rooted by a tree node, contained within the expression tree rooted at this node.

Return type Iterator[*ExpressionTreeNode*]

Raises ***RequiresNormalFormError*** – If the expression tree rooted at this node is not in disjunctive normal form.

l_child
This node's left child; `None` indicates the absence of a child.

Type *ExpressionTreeNode* or `None`

negated_symbol_set
A set of the negated symbols present in the tree rooted here.

Type Set[`str`]

non_negated_symbol_set
A set of the non-negated symbols present in the tree rooted here.

Type Set[`str`]

r_child
This node's left child; `None` indicates the absence of a child.

Type *ExpressionTreeNode* or `None`

symbol_name
The string operator/operand name wrapped in this node.

Type `str`

to_cnf()
Return a transformed node, in conjunctive normal form.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all operators transformed to consist only of NOTs, ANDs, and ORs.

Return type *ExpressionTreeNode*

to_primitives()
Return a transformed node, containing only NOTs, ANDs, and ORs.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all operators transformed to consist only of NOTs, ANDs, and ORs.

Return type *ExpressionTreeNode*

class tt.trees.tree_node.**OperandExpressionTreeNode**(*operand_str*)

Bases: *tt.trees.tree_node.ExpressionTreeNode*

An expression tree node for operands.

Nodes of this type will always be leaves in an expression tree.

__eq__(*other*)

Return self==value.

__init__(*operand_str*)

Initialize self. See help(type(self)) for accurate signature.

apply_de_morgans()

Return a transformed node, with De Morgan's Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all negated AND and OR operators transformed, following De Morgan's Law.

Return type *ExpressionTreeNode*

apply_idempotent_law()

Returns a transformed node, with the Idempotent Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects

Returns An expression tree node with the Idempotent Law applied to *AND* and *OR* operators.

Return type *ExpressionTreeNode*

This transformation will apply the Idempotent Law to *AND* and *OR* expressions involving repeated operands. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and A').tree
>>> print(tree.apply_idempotent_law())
A
>>> tree = BooleanExpression('~B or ~~~B').tree
>>> print(tree.apply_idempotent_law())
~
`----B
```

In the latter of the two above examples, we see that this transformation will compare operands with negations condensed. This transformation will also prune redundant operands from CNF and DNF clauses. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and B and B and C and ~C and ~~C and D').tree
>>> print(tree.apply_idempotent_law())
and
`----and
|    `----and
|    |    `----and
|    |    |    `----A
|    |    |    `----B
|    |    `----C
|    `-----~
|         `----C
`----D
```

apply_identity_law()

Return a transformed node, with the Identity Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

This transformation will achieve the following effects by applying the Inverse Law to the *AND* and *OR* operators:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 1').tree
>>> print(tree.apply_identity_law())
A
>>> tree = BooleanExpression('0 or B').tree
>>> print(tree.apply_identity_law())
B
```

It should also be noted that this transformation will also apply the annihilator properties of the logical *AND* and *OR* operators. For example:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 0').tree
>>> print(tree.apply_identity_law())
0
>>> tree = BooleanExpression('1 or B').tree
>>> print(tree.apply_identity_law())
1
```

Returns An expression tree node with AND and OR identities simplified.

Return type *ExpressionTreeNode*

apply_inverse_law()

Return a transformed node, with the Inverse Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with the Inverse Law applied to applicable clauses.

Return type *ExpressionTreeNode*

This transformation will apply the Inverse Law to *AND* and *OR* expressions involving the negated and non-negated forms of a variable. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('~A and A').tree
>>> print(tree.apply_inverse_law())
0
>>> tree = BooleanExpression('B or !B').tree
>>> print(tree.apply_inverse_law())
1
```

Note that this transformation will **not** reduce expressions of constants; the transformation *apply_identity_law* will probably do what you want in this case, though.

This transformation will also reduce expressions in CNF or DNF that contain negated and non-negated forms of the same symbol. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A or B or C or ~B').tree
```

(continues on next page)

(continued from previous page)

```
>>> print(tree.apply_inverse_law())
1
>>> tree = BooleanExpression('A and B and C and !B').tree
>>> print(tree.apply_inverse_law())
0
```

coalesce_negations()
Return a transformed node, with consecutive negations coalesced.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all consecutive negations compressed into the minimal number of equivalent negations (either one or none).

Return type *ExpressionTreeNode*

distribute_ands()
Return a transformed nodes, with ANDs recursively distributed across ORed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable AND operators distributed across ORed sub-expressions.

Return type *ExpressionTreeNode*

distribute_ors()
Return a transformed nodes, with ORs recursively distributed across ANDed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable OR operators distributed across ANDed sub-expressions.

Return type *ExpressionTreeNode*

evaluate(*input_dict*)
Recursively evaluate this node.

This is an interface that should be defined in sub-classes. Node evaluation does no checking of the validity of inputs; they should be check before being passed here.

Parameters **input_dict** (Dict{str: truthy) – A dictionary mapping expression symbols to the value for which they should be subsituted in expression evaluation.

Returns The evaluation of the tree rooted at this node.

Return type bool

to_primitives()
Return a transformed node, containing only NOTs, ANDs, and ORs.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all operators transformed to consist only of NOTs, ANDs, and ORs.

Return type *ExpressionTreeNode*

class tt.trees.tree_node.**UnaryOperatorExpressionTreeNode**(*operator_str*, *l_child*)
Bases: *tt.trees.tree_node.ExpressionTreeNode*

An expression tree node for unary operators.

__eq__(*other*)
Return self==value.

__init__(*operator_str*, *l_child*)
Initialize self. See help(type(self)) for accurate signature.

apply_de_morgans()
Return a transformed node, with De Morgan's Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all negated AND and OR operators transformed, following De Morgan's Law.

Return type *ExpressionTreeNode*

apply_idempotent_law()
Returns a transformed node, with the Idempotent Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects

Returns An expression tree node with the Idempotent Law applied to *AND* and *OR* operators.

Return type *ExpressionTreeNode*

This transformation will apply the Idempotent Law to *AND* and *OR* expressions involving repeated operands. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and A').tree
>>> print(tree.apply_idempotent_law())
A
>>> tree = BooleanExpression('~B or ~~~B').tree
>>> print(tree.apply_idempotent_law())
~
`----B
```

In the latter of the two above examples, we see that this transformation will compare operands with negations condensed. This transformation will also prune redundant operands from CNF and DNF clauses. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and B and B and C and ~C and ~~C and D').tree
>>> print(tree.apply_idempotent_law())
and
`----and
|    `----and
|    |    `----and
|    |    |    `----A
|    |    |    `----B
|    |    `----C
|    `----~
|         `----C
`----D
```

apply_identity_law()
Return a transformed node, with the Identity Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

This transformation will achieve the following effects by applying the Inverse Law to the *AND* and *OR* operators:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 1').tree
>>> print(tree.apply_identity_law())
A
>>> tree = BooleanExpression('0 or B').tree
>>> print(tree.apply_identity_law())
B
```

It should also be noted that this transformation will also apply the annihilator properties of the logical *AND* and *OR* operators. For example:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A and 0').tree
>>> print(tree.apply_identity_law())
0
>>> tree = BooleanExpression('1 or B').tree
>>> print(tree.apply_identity_law())
1
```

Returns An expression tree node with AND and OR identities simplified.

Return type *ExpressionTreeNode*

apply_inverse_law()

Return a transformed node, with the Inverse Law applied.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with the Inverse Law applied to applicable clauses.

Return type *ExpressionTreeNode*

This transformation will apply the Inverse Law to *AND* and *OR* expressions involving the negated and non-negated forms of a variable. Here are a few examples:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('~A and A').tree
>>> print(tree.apply_inverse_law())
0
>>> tree = BooleanExpression('B or !B').tree
>>> print(tree.apply_inverse_law())
1
```

Note that this transformation will **not** reduce expressions of constants; the transformation *apply_identity_law* will probably do what you want in this case, though.

This transformation will also reduce expressions in CNF or DNF that contain negated and non-negated forms of the same symbol. Let's take a look:

```
>>> from tt import BooleanExpression
>>> tree = BooleanExpression('A or B or C or ~B').tree
>>> print(tree.apply_inverse_law())
1
>>> tree = BooleanExpression('A and B and C and !B').tree
>>> print(tree.apply_inverse_law())
0
```

coalesce_negations()

Return a transformed node, with consecutive negations coalesced.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all consecutive negations compressed into the minimal number of equivalent negations (either one or none).

Return type *ExpressionTreeNode*

distribute_ands()
Return a transformed nodes, with ANDs recursively distributed across ORed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable AND operators distributed across ORed sub-expressions.

Return type *ExpressionTreeNode*

distribute_ors()
Return a transformed nodes, with ORs recursively distributed across ANDed sub-expressions.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all applicable OR operators distributed across ANDed sub-expressions.

Return type *ExpressionTreeNode*

evaluate(*input_dict*)
Recursively evaluate this node.

This is an interface that should be defined in sub-classes. Node evaluation does no checking of the validity of inputs; they should be check before being passed here.

Parameters **input_dict** (Dict{str: truthy) – A dictionary mapping expression symbols to the value for which they should be subsituted in expression evaluation.

Returns The evaluation of the tree rooted at this node.

Return type bool

operator
The actual operator object wrapped in this node.

Type *BooleanOperator*

to_primitives()
Return a transformed node, containing only NOTs, ANDs, and ORs.

Since nodes are immutable, the returned node, and all descendants, are new objects.

Returns An expression tree node with all operators transformed to consist only of NOTs, ANDs, and ORs.

Return type *ExpressionTreeNode*

Python Module Index

t

Index

Symbols

A

B

C

D

L

M

N

O

P

R

S

T

U