tt

Release 0.6.0

Apr 30, 2017

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 is a Python library and command-line tool for working with Boolean expressions. Please check out the project site for more information.

CHAPTER 2

Installation

tt is tested on CPython 2.7, 3.3, 3.4, 3.5, and 3.6. You can get the latest release from PyPI with:

```
pip install ttable
```

CHAPTER 3

Basic Usage

tt aims to provide a Pythonic interface for working with Boolean expressions. Here are some simple examples from the REPL:

```
>>> from tt import BooleanExpression, TruthTable
>>> b = BooleanExpression('A xor (B and 1)')
>>> b.tokens
['A', 'xor', '(', 'B', 'and', '1', ')']
>>> b.symbols
['A', 'B']
>>> print(b.tree)
xor
`----A
`----and
     `----B
     `----1
>>> b.evaluate(A=True, B=False)
True
>>> t = TruthTable(b)
>>> print(t)
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 0 |
+---+---+---+
>>> t = TruthTable(from_values='01xx')
>>> t.ordering
['A', 'B']
>>> for inputs, result in t:
...     inputs, result
```

```
...
(<BooleanValues [A=0, B=0]>, False)
(<BooleanValues [A=0, B=1]>, True)
(<BooleanValues [A=1, B=0]>, 'x')
(<BooleanValues [A=1, B=1]>, 'x')
>>> t.equivalent_to(b)
True
```

CHAPTER 4

License

tt uses the MIT License.

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*.

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 *BooleanExpressionTree*. 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, NAND, NOR, NOT, NXOR, OR, XNOR, XOR, \/, and, 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
```

```
<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
     ^
```

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
```

Release Notes

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

0.6.x

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

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

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

pre-0.5

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

Development

If you'd like to help out with tt, we'd love to have you. Below are some helpful tips to development. Feel free to *reach out* with any questions about development or getting involved.

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.

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
```

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, this is mostly just to make sure the documentation examples are valid. The true behavior of the library and its public contract are enforced through the unit tests.

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. Whenever new code is pushed to the repo, this same set of tox tests is run on AppVeyor (for Windows builds). A separate configuration is used for Travis CI, which tests on Linux.

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.

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 changes. To run one on port 5000 of your machine, run:

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

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. You will need to install several different compilers:

- 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 and 3.6)

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.

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}. On tagged commits, the CI services will automatically build wheels and publish them to PyPI.

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.

General purpose EDA/Boolean logic

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

Satisfiability

- PyEDA
- pycosat
- SATisPy

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!

Services

Thank you to the free hosting provided by these services!

- Github
- Travis CI
- AppVeyor
- Read the Docs

Design Resources

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

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

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
- virtualenv

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 the tools tt provided, check out the autogenerated API docs:

cli

tt's command-line interface.

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

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.

definitions

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

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]

definitions.operands module

Definitions related to operands.

tt.definitions.operands.**BOOLEAN_VALUES** = {False, True}
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
```

```
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')
>>> instance._asdict()
OrderedDict([('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
>>> 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
```

.

definitions.operators module

Definitions for tt's built-in Boolean operators.

class tt.definitions.operators.**BooleanOperator**(*precedence*, *eval_func*, *name*)

Bases: object

A wrapper around a Boolean operator.

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
```

name

The human-readable name of this operator.

Type str

```
>>> from tt.definitions import TT_NOT_OP, TT_XOR_OP
>>> TT_NOT_OP.name
'NOT'
>>> TT_XOR_OP.name
'XOR'
```

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.**OPERATOR_MAPPING = {'~': <BooleanOperator NOT>, 'XOR': <BooleanOperator XOR>,**

A mapping of Boolean operators.

This mapping serves to define all valid operator strings and maps them to the appropriate *BooleanOperator* object defining the operator behavior.

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_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*

errors

tt error types.

errors.base module

The base tt exception type.

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

Base exception type for tt errors.

Note: This exception type should be sub-classed and is not meant to be raised explicitly.

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

Type str

errors.arguments module

Generic exception types.

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

An exception type for invalid arguments.

Note: 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.

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable('A or B', from_values='1111')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.arguments.ConflictingArgumentsError'>
```

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

An exception type for invalid argument types.

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable(7)
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.arguments.InvalidArgumentTypeError'>
```

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

An exception type for invalid argument values.

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable(from_values='01x')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.arguments.InvalidArgumentValueError'>
```

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

An exception for when a required argument is missing.

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable()
... except Exception as e:
```

```
...     print(type(e))
...
<class 'tt.errors.arguments.RequiredArgumentError'>
```

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.

Note: 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 an invalid truth or don't care value passed.

```
>>> from tt import BooleanExpression
>>> try:
...     b = BooleanExpression('A or B')
...     b.evaluate(A=1, B='brian')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.evaluation.InvalidBooleanValueError'>
```

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

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

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable('1 or 0')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.evaluation.NoEvaluationVariationError'>
```

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.

```
>>> from tt import BooleanExpression
>>> try:
...     b = BooleanExpression(') A or B')
```

```
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.grammar.BadParenPositionError'>
```

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.

```
>>> from tt import BooleanExpression
>>> try:
...     b = BooleanExpression('')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.grammar.EmptyExpressionError'>
```

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.

```
>>> from tt import BooleanExpression
>>> try:
...     b = BooleanExpression('A or or B')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.grammar.ExpressionOrderError'>
```

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.

Note: This exception type should be sub-classed and is not meant to be raised explicitly.

---

error_pos

The position in the expression where the error occurred.

Note: This may be left as `None`, in which case there is no specific location in the expression causing the exception.

Type `int`

expr_str

The expression in which the exception occurred.

Note: This may be left as `None`, in which case 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.

```
>>> from tt import BooleanExpression, TruthTable
>>> b = BooleanExpression('%A or B')
Traceback (most recent call last):
    ...
tt.errors.grammar.InvalidIdentifierError: Invalid operand name "%A"
>>> t = TruthTable(from_values='0x11', ordering=['A', 'while'])
Traceback (most recent call last):
    ...
tt.errors.grammar.InvalidIdentifierError: "while" 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.

```
>>> from tt import BooleanExpression
>>> try:
...     b = BooleanExpression('A or ((B)')
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.grammar.UnbalancedParenError'>
```

errors.state module

Exception type definitions related to invalid operations based on state.

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()
>>> try:
...     t.fill()
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.state.AlreadyFullTableError'>
```

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)
>>> try:
...     print(t.equivalent_to('A or B'))
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.state.RequiresFullTableError'>
```

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

Bases: *tt.errors.base.TtError*

An exception type for errors occurring in expression evaluation.

Note: This exception type should be sub-classed and is not meant to be raised explicitly.

---

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.

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable('A or B', ordering=['A', 'A', 'B'])
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.DuplicateSymbolError'>
```

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

Bases: *tt.errors.symbols.SymbolError*

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

```
>>> from tt import TruthTable
>>> try:
...     t = TruthTable('A or B', ordering=['A', 'B', 'C'])
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.ExtraSymbolError'>
```

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
>>> try:
...     b = BooleanExpression('A and B')
...     b.evaluate(A=1)
... except Exception as e:
```

```
...     print(type(e))
...
<class 'tt.errors.symbols.MissingSymbolError'>
```

exception `tt.errors.symbols.`**`SymbolError`**(*message*, **args*)
Bases: *tt.errors.base.TtError*

An exception for errors occurring in symbol processing.

Note: This exception type should be sub-classed and is not meant to be raised explicitly.

expressions

Tools for working with Boolean expressions.

expressions.bexpr module

Tools for interacting with Boolean expressions.

class `tt.expressions.bexpr.`**`BooleanExpression`**(*raw_expr*)
Bases: `object`

An interface for interacting with a Boolean expression.

Instances of `BooleanExpression` are meant to be immutable.

`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 identifier.

Note: See *assert_all_valid_keys* and *assert_iterable_contains_all_expr_symbols* for more information about the exceptions raised by this method.

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`

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'
```

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 expression tree representing this Boolean expression.

Type *BooleanExpressionTree*

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

satisfiability

Functionality for determining logic satisfiasbility.

satisfiability.picosat module

Python wrapper around the _clibs PicoSAT extension.

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

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.

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.

Example usage:

```
>>> from tt import picosat
>>> # Return None when no solution possible
... picosat.sat_one([[1], [-1]]) is None
```

```
True
>>> # Compute a satisfying solution
... picosat.sat_one([[1, 2, 3], [-2, -3], [-3]])
[1, -2, -3]
>>> # Include assumptions
... picosat.sat_one([[1, 2, 3], [2, 3]], assumptions=[-3])
[1, 2, -3]
```

tables

Tools for working with truth tables.

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 |   |
+----+-------+---+
| 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.

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

- ***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.

Note: See *assert_all_valid_keys* for more information about the exceptions raised by this method.

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']
>>> 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 |   |
+---+---+---+---+
| 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]
```

trees

Tools for working with Boolean expression trees.

trees.expr_tree module

An expression tree implementation for Boolean expressions.

class tt.trees.expr_tree.**BooleanExpressionTree**(*postfix_tokens*)

Bases: object

An expression tree for Boolean expressions.

This class expects any input it receives to be well-formed; any tokenized lists you pass it directly (instead of from the attribute of the *BooleanExpression* class) will not be checked.

evaluate(*input_dict*)

Evaluate the expression held in this tree for specified inputs.

Parameters **input_dict** (Dict{str: truthy}) – A dict mapping string variable names to the values for which they should be evaluated.

Returns The result of the expression tree evaluation.

Return type bool

Note: This function does not check to ensure the validity of the input_dict argument in any way.

While you would normally evaluate expressions through the interface provided by the *BooleanExpression* class, this interface is still exposed for your use if you want to avoid any overhead introduced by the extra layer of abstraction. For example:

```
>>> from tt import BooleanExpressionTree
>>> bet = BooleanExpressionTree(['A', 'B', 'xor'])
>>> bet.evaluate({'A': 1, 'B': 0})
True
>>> bet.evaluate({'A': 1, 'B': 1})
False
```

postfix_tokens

The tokens, in postfix order, from which this tree was built.

Type List[str]

root

The root of the tree; this is None for an empty tree.

Type *ExpressionTreeNode*

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.

operator

The actual operator object wrapped in this node.

Type *BooleanOperator*

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

A base class for expression tree nodes.

evaluate(*input_dict*)
Recursively evaluate this node.

This is an interface that should be defined in sub-classes.

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

Note: Node evaluation does no checking of the validity of inputs; they should be check before being passed here.

Returns The evaluation of the tree rooted at this node.

Return type bool

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

Type *ExpressionTreeNode*, optional

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

Type *ExpressionTreeNode*, optional

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

Type str

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.

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

An expression tree node for unary operators.

operator
The actual operator object wrapped in this node.

Type *BooleanOperator*

utils

Utilities for use under the hood.

tt.utils.assertions module

Utilities for asserting inputs and states.

tt.utils.assertions.**assert_all_valid_keys**(*symbol_input_dict*, *symbol_set*)

Assert that all keys in the passed input dict are valid.

Valid keys are considered those that are present in the passed set of symbols and that map to valid Boolean values. Dictionaries cannot have duplicate keys, so no duplicate checking is necessary.

Parameters

- **symbol_input_dict** (Dict{str: truthy}) – A dict containing symbol names mapping to what should be Boolean values.
- **symbol_set** (Set[str]) – A set of the symbol names expected to be present as keys in symbol_input_dict.

Raises

- ***ExtraSymbolError*** – If any keys in the passed input dict are not present in the passed set of symbols.
- ***InvalidBooleanValueError*** – If any values in the passed input dict are not valid Boolean values (1, 0, True, or False).

This assert is used for validation of user-specified kwargs which map symbols to expected values. Below are some example uses.

Valid input:

```
>>> from tt.utils.assertions import assert_all_valid_keys
>>> try:
...     assert_all_valid_keys({'A': True, 'B': False},
...                           {'A', 'B'})
... except Exception as e:
...     print(type(e))
... else:
...     print('All good!')
...
All good!
```

Producing an *ExtraSymbolError*:

```
>>> from tt.utils.assertions import assert_all_valid_keys
>>> try:
...     assert_all_valid_keys({'A': 1, 'B': 0}, {'A'})
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.ExtraSymbolError'>
```

Producing an *InvalidBooleanValueError*:

```
>>> from tt.utils.assertions import assert_all_valid_keys
>>> try:
...     assert_all_valid_keys({'A': 'brian', 'B': True},
...                           {'A', 'B'})
... except Exception as e:
...     print(type(e))
```

```
...
<class 'tt.errors.evaluation.InvalidBooleanValueError'>
```

tt.utils.assertions.**assert_iterable_contains_all_expr_symbols**(*iter_of_strs*, *reference_set*)

Assert a one-to-one presence of all symbols in the passed iterable.

Parameters

- **iter_of_strs** (Iterable[str]) – An iterable of strings to assert.
- **reference_set** (Set[str]) – A set of strings, each of which will be asserted to be present in the passed iterable.

Note: This function will consume `iter_of_strs`.

Raises

- ***DuplicateSymbolError*** – If the passed iterable contains more than one of a given symbol.
- ***ExtraSymbolError*** – If the passed iterable contains symbols not present in the reference set.
- ***MissingSymbolError*** – If the passed iterable is missing symbols present in the reference set.

This assertion is used for validation of user-specified sets of symbols. Below are some example uses.

Valid input:

```
>>> from tt.utils.assertions import (
...     assert_iterable_contains_all_expr_symbols)
>>> try:
...     assert_iterable_contains_all_expr_symbols(
...             ('A', 'B', 'C',),
...             {'A', 'B', 'C'},
...     )
... except Exception as e:
...     print(type(e))
... else:
...     print('All good!')
...
All good!
```

Producing a *DuplicateSymbolError*:

```
>>> from tt.utils.assertions import (
...     assert_iterable_contains_all_expr_symbols)
>>> try:
...     assert_iterable_contains_all_expr_symbols(
...             ('A', 'A', 'B',),
...             {'A', 'B'}
...     )
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.DuplicateSymbolError'>
```

Producing an *ExtraSymbolError*:

```
>>> from tt.utils.assertions import (
...     assert_iterable_contains_all_expr_symbols)
>>> try:
...     assert_iterable_contains_all_expr_symbols(
...             ('A', 'B', 'C',),
...             {'A', 'B'}
...     )
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.ExtraSymbolError'>
```

Producing a *MissingSymbolError*:

```
>>> from tt.utils.assertions import (
...     assert_iterable_contains_all_expr_symbols)
>>> try:
...     assert_iterable_contains_all_expr_symbols(
...             ('A', 'B',),
...             {'A', 'B', 'C'}
...     )
... except Exception as e:
...     print(type(e))
...
<class 'tt.errors.symbols.MissingSymbolError'>
```

Python Module Index

t

Index