
tt

Release 0.5.1

Apr 06, 2017

Contents

1	Synopsis	3
2	Installation	5
3	Basic Usage	7
4	License	9
4.1	User Guide	9
4.2	Development	14
4.3	Release Notes	15
4.4	Prior Art	16
4.5	Special Thanks	17
4.6	Author	18
5	Want to learn more?	19
5.1	cli	19
5.2	definitions	20
5.3	errors	22
5.4	expressions	27
5.5	tables	29
5.6	trees	35
5.7	utils	37
	Python Module Index	41

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 as well as PyPy. tt is written in pure Python with no dependencies, so it only requires a compatible Python installation to run. You can get the latest release from PyPI with:

```
pip install ttable
```


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')
>>> for inputs, result in t:
...     inputs, result
...
((False, False), False)
```

```
((False, True), True)
((True, False), 'x')
((True, True), 'x')
>>> t.equivalent_to(b)
True
```

tt uses the MIT License.

User Guide

The below code samples 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

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']
```

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

```

Under the hood, this expression tree is used for evaluation of the expression. The expression 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"

```

we see that we get an exception. This brings up one of the key design philosophies of tt: exceptions are specific and abundant. This library does its best to make sure you use its top-level interfaces as designed, and will raise exceptions if something isn't right.

Exceptions in tt are organized in a hierarchy, with each category of exceptions grouped under a base exception type. For example, a common group of exceptions you'll deal with if you might handle malformed expressions is *GrammarError*. *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:

```

>>> from tt import GrammarError
>>> 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

Now that we've gotten the basic idea of how expressions work, we can use them to fill some truth tables. 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 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 saw in the above example, printing tables produces a nicely-formatted text table. While we have stuck to simple variable names (A, B, C, etc.) so far, these tables will scale to fit the size of the symbol names:

```
>>> 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 |
+---+---+---+
```

These tables are populated by evaluating the expression for each combination of input values. Let's say that you already have the values you want in your truth table and would rather skip over re-evaluating ; 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* (indicating 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 |
+-----+-----+---+
```

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, like these:

```
>>> t = TruthTable('!A && B')
>>> t.results
[False, True, False, False]
>>> for inputs, result in t:
...     inputs, result
...
((False, False), False)
((False, True), True)
((True, False), False)
((True, True), False)
>>> t[0], t[1], t[2], t[3]
(False, True, False, False)
```

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

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

Up to this point, we've only taken a look at tables with all 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
```

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 look 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 explicit 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
```

Development

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. You will see it 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 public contract is enforced through 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 and also adds the ability to test on the `PyPy` runtime.

Style

tt aims to be strictly [PEP8](#) compliant, enforcing this compliance via [Flake8](#). This project includes an `editorconfig` file to help with formatting issues, as well.

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}`.

After these steps, make sure you update the release notes, publish on Read the Docs, and publish on PyPI.

Long Term Development Goals

Below are features I'd like to add eventually, roughly ordered in anticipated schedule of completion. A new release will be cut every so often down the list.

- For the CLI
 - Functional testing, capturing `stdout/stderr`
 - Option for interfacing with the truth table's `fill` method
 - Option for interfacing with the truth table's `ordering` attribute
 - Option for specifying output delimiters for token-listing commands
- For the project as a whole
 - A *Getting Started* section for the docs, with a tutorial-style guide to the library and CLI
 - Karnaugh map support
 - Interface for substituting/transforming expression symbols
 - Functionality for optimizing/simplifying expressions (`pos`, `sop`, `espresso`, etc.)

Release Notes

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

0.5.x

Features in the 0.5.x series of releases were focused on expanding the top-level interfacing 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

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

Prior Art

There are some great projects operating in the same problem space as *tt*. Most of the listed libraries are more mature and feature-rich than *tt*, so they may be a better choice for the problems you're working on. If you think that your library should be listed here, please let me know or submit a PR.

Python libraries

- *boolean.py*
- *PyEDA*

Other languages

- [LogicNG](#) (Java)
- [BoolExpr](#) (C++)
- [EvalEx](#) (Java)

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](#)

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](#)
- [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

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 it 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 = {'1', '0'}
    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 :class'str <python:str>'
```

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 = {'xor': <BooleanOperator XOR>, 'NAND': <BooleanOperator NA`

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.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.

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
```

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 initialization; 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.
- ***NoEvaluationVariationError*** – If an expression without any unique 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

tt.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 substituted 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.

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'>
```


t

- `tt.cli`, 19
- `tt.cli.core`, 19
- `tt.cli.utils`, 20
- `tt.definitions`, 20
- `tt.definitions.grammar`, 20
- `tt.definitions.operands`, 20
- `tt.definitions.operators`, 20
- `tt.errors`, 22
- `tt.errors.arguments`, 22
- `tt.errors.base`, 22
- `tt.errors.evaluation`, 23
- `tt.errors.grammar`, 24
- `tt.errors.state`, 25
- `tt.errors.symbols`, 26
- `tt.expressions`, 27
- `tt.expressions.bexpr`, 27
- `tt.tables`, 29
- `tt.tables.truth_table`, 29
- `tt.trees`, 35
- `tt.trees.expr_tree`, 35
- `tt.trees.tree_node`, 36
- `tt.utils`, 37
- `tt.utils.assertions`, 37

A

AlreadyFullTableError, 25
 ArgumentError, 22
 assert_all_valid_keys() (in module tt.utils.assertions), 37
 assert_iterable_contains_all_expr_symbols() (in module tt.utils.assertions), 38

B

BadParenPositionError, 24
 BinaryOperatorExpressionTreeNode (class in tt.trees.tree_node), 36
 BOOLEAN_VALUES (in module tt.definitions.operands), 20
 BooleanExpression (class in tt.expressions.bexpr), 27
 BooleanExpressionTree (class in tt.trees.expr_tree), 35
 BooleanOperator (class in tt.definitions.operators), 20

C

ConflictingArgumentsError, 22
 CONSTANT_VALUES (in module tt.definitions.grammar), 20

D

DELIMITERS (in module tt.definitions.grammar), 20
 DONT_CARE_VALUE (in module tt.definitions.operands), 20
 DuplicateSymbolError, 26

E

EmptyExpressionError, 24
 equivalent_to() (tt.tables.truth_table.TruthTable method), 31
 error_pos (tt.errors.grammar.GrammarError attribute), 25
 eval_func (tt.definitions.operators.BooleanOperator attribute), 20
 evaluate() (tt.expressions.bexpr.BooleanExpression method), 27
 evaluate() (tt.trees.expr_tree.BooleanExpressionTree method), 35

evaluate() (tt.trees.tree_node.ExpressionTreeNode method), 36
 evaluate_unchecked() (tt.expressions.bexpr.BooleanExpression method), 28
 EvaluationError, 23
 expr (tt.tables.truth_table.TruthTable attribute), 32
 expr_str (tt.errors.grammar.GrammarError attribute), 25
 ExpressionOrderError, 24
 ExpressionTreeNode (class in tt.trees.tree_node), 36
 ExtraSymbolError, 26

F

fill() (tt.tables.truth_table.TruthTable method), 32

G

generate_symbols() (tt.tables.truth_table.TruthTable static method), 33
 get_parsed_args() (in module tt.cli.core), 19
 GrammarError, 25

I

input_combos() (tt.tables.truth_table.TruthTable static method), 33
 InvalidArgumentTypeError, 22
 InvalidArgumentValueError, 23
 InvalidBooleanValueError, 23
 is_full (tt.tables.truth_table.TruthTable attribute), 33

L

l_child (tt.trees.tree_node.ExpressionTreeNode attribute), 36

M

main() (in module tt.cli.core), 19
 MAX_OPERATOR_STR_LEN (in module tt.definitions.operators), 21
 message (tt.errors.base.TtError attribute), 22
 MissingSymbolError, 27

N

name (tt.definitions.operators.BooleanOperator attribute), 21
 NoEvaluationVariationError, 24

O

OperandExpressionTreeNode (class in tt.trees.tree_node), 36
 operator (tt.trees.tree_node.BinaryOperatorExpressionTreeNode attribute), 36
 operator (tt.trees.tree_node.UnaryOperatorExpressionTreeNode attribute), 37
 OPERATOR_MAPPING (in module tt.definitions.operators), 21
 ordering (tt.tables.truth_table.TruthTable attribute), 34

P

postfix_tokens (tt.expressions.bexpr.BooleanExpression attribute), 28
 postfix_tokens (tt.trees.expr_tree.BooleanExpressionTree attribute), 35
 precedence (tt.definitions.operators.BooleanOperator attribute), 21
 print_err() (in module tt.cli.utils), 20
 print_info() (in module tt.cli.utils), 20

R

r_child (tt.trees.tree_node.ExpressionTreeNode attribute), 36
 raw_expr (tt.expressions.bexpr.BooleanExpression attribute), 28
 RequiredArgumentError, 23
 RequiresFullTableError, 26
 results (tt.tables.truth_table.TruthTable attribute), 34
 root (tt.trees.expr_tree.BooleanExpressionTree attribute), 36

S

StateError, 26
 symbol_name (tt.trees.tree_node.ExpressionTreeNode attribute), 36
 SymbolError, 27
 symbols (tt.expressions.bexpr.BooleanExpression attribute), 28

T

tokens (tt.expressions.bexpr.BooleanExpression attribute), 29
 tree (tt.expressions.bexpr.BooleanExpression attribute), 29
 TruthTable (class in tt.tables.truth_table), 29
 tt.cli (module), 19
 tt.cli.core (module), 19

tt.cli.utils (module), 20
 tt.definitions (module), 20
 tt.definitions.grammar (module), 20
 tt.definitions.operands (module), 20
 tt.definitions.operators (module), 20
 tt.errors (module), 22
 tt.errors.arguments (module), 22
 tt.errors.base (module), 22
 tt.errors.evaluation (module), 23
 tt.errors.grammar (module), 24
 tt.errors.state (module), 25
 tt.errors.symbols (module), 26
 tt.expressions (module), 27
 tt.expressions.bexpr (module), 27
 tt.tables (module), 29
 tt.tables.truth_table (module), 29
 tt.trees (module), 35
 tt.trees.expr_tree (module), 35
 tt.trees.tree_node (module), 36
 tt.utils (module), 37
 tt.utils.assertions (module), 37
 TT_AND_OP (in module tt.definitions.operators), 21
 TT_NAND_OP (in module tt.definitions.operators), 21
 TT_NOR_OP (in module tt.definitions.operators), 21
 TT_NOT_OP (in module tt.definitions.operators), 21
 TT_OR_OP (in module tt.definitions.operators), 21
 TT_XNOR_OP (in module tt.definitions.operators), 21
 TT_XOR_OP (in module tt.definitions.operators), 22
 TtError, 22

U

UnaryOperatorExpressionTreeNode (class in tt.trees.tree_node), 37
 UnbalancedParenError, 25