tt

Release 0.5.0

Mar 09, 2017

Contents

1 Synopsis **3**

2 Installation **5**

3 Basic Usage **7**
- 3.1 As a Library 7
- 3.2 From the Command Line 8

4 License **11**
- 4.1 Development 11
- 4.2 Release Notes 12
- 4.3 Prior Art 13
- 4.4 Special Thanks 13
- 4.5 Author 14

5 API Docs **15**
- 5.1 `cli` 15
- 5.2 `definitions` 16
- 5.3 `errors` 18
- 5.4 `expressions` 21
- 5.5 `tables` 23
- 5.6 `trees` 26
- 5.7 `utils` 28

Python Module Index **33**

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
```

CHAPTER 3

Basic Usage

Below are a couple of examples to show you the kind of things tt can do. For more examples and further documentation, take a look at the project site.

As a Library

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('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 |
+---+---+---+
```

From the Command Line

tt also provides a command-line interface for working with expressions. Here are a couple of examples:

```
$ tt tokens "(op1 nand op2) xnor op3"
(
op1
nand
op2
)
xnor
op3

$ tt table A or B
+---+---+---+
| A | B |   |
+---+---+---+
| 0 | 0 | 0 |
+---+---+---+
| 0 | 1 | 1 |
+---+---+---+
| 1 | 0 | 1 |
+---+---+---+
| 1 | 1 | 1 |
+---+---+---+

$ tt tree A or or B
```

```
Error! Unexpected binary operator "or":
A or or B
     ^
```

CHAPTER 4

License

tt uses the MIT License.

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.

0.5.0

- 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

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

CHAPTER 5

API Docs

Feel free to peruse through the source, or take a look through the auto-generated api docs below.

`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`]

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 OR>, 'nxor': <BooleanOperator XNOR>, '**
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.evaluation` module

Exception type definitions related to expression evaluation.

exception `tt.errors.evaluation.`**`DuplicateSymbolError`**(*message*, **args*)

Bases: *`tt.errors.evaluation.EvaluationError`*

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.evaluation.DuplicateSymbolError'>
```

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.`**`ExtraSymbolError`**(*message*, **args*)

Bases: *`tt.errors.evaluation.EvaluationError`*

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.evaluation.ExtraSymbolError'>
```

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

Bases: *`tt.errors.evaluation.EvaluationError`*

An exception for an invalid truth value passed in evaluation.

```
>>> 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.`**`MissingSymbolError`**(*message*, **args*)

Bases: *`tt.errors.evaluation.EvaluationError`*

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.evaluation.MissingSymbolError'>
```

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.generic` module

Generic exception types.

exception `tt.errors.generic.`**`InvalidArgumentTypeError`**(*message*, **args*)

Bases: *`tt.errors.base.TtError`*

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.generic.InvalidArgumentTypeError'>
```

`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'>
```

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*, *fill_all=True*, *ordering=None*)

Bases: object

A class representing a truth table.

Parameters

- **expr** (str or *BooleanExpression*) – The expression with which to populate this truth 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

- ***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.
- ***NoEvaluationVariationError*** – If an expression without any unqiue symbols (i.e., one merely composed of constant operators) is specified.

Note: See *assert_iterable_contains_all_expr_symbols* for more information about the exceptions raised by this class's initializer.

expr

The `BooleanExpression` object represented by this table.

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

`input_combos`(*combo_len=None*)

Get an iterator of Boolean input combinations for this expression.

Parameters **`combo_len`** (`int`, optional) – The length of each combination in the returned iterator. If omitted, this defaults to the number of symbols in the expression.

Returns An iterator of tuples containing permutations of Boolean inputs.

Return type `itertools.product`

The length of each tuple of combinations is the same as the number of symbols in this expression if no `combo_len` value is specified; otherwise, the specified value is used.

Iterating through the returned value, without fiddling with the `combo_len` input, will yield every combination of inputs for this expression.

A simple example:

```
>>> from tt import TruthTable
>>> t = TruthTable('A and B')
>>> for tup in t.input_combos():
...     print(tup)
...
(False, False)
(False, True)
(True, False)
(True, True)
```

`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]

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]
```

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.

`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.evaluation.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.evaluation.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.evaluation.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.evaluation.MissingSymbolError'>
```

Python Module Index

t

tt.cli, 15
tt.cli.core, 15
tt.cli.utils, 15
tt.definitions, 16
tt.definitions.grammar, 16
tt.definitions.operands, 16
tt.definitions.operators, 16
tt.errors, 18
tt.errors.base, 18
tt.errors.evaluation, 18
tt.errors.generic, 19
tt.errors.grammar, 20
tt.expressions, 21
tt.expressions.bexpr, 21
tt.tables, 23
tt.tables.truth_table, 23
tt.trees, 26
tt.trees.expr_tree, 26
tt.trees.tree_node, 27
tt.utils, 28
tt.utils.assertions, 28

Index

A

assert_all_valid_keys() (in module tt.utils.assertions), 28
assert_iterable_contains_all_expr_symbols() (in module tt.utils.assertions), 29

B

BadParenPositionError, 20
BinaryOperatorExpressionTreeNode (class in tt.trees.tree_node), 27
BOOLEAN_VALUES (in module tt.definitions.operands), 16
BooleanExpression (class in tt.expressions.bexpr), 21
BooleanExpressionTree (class in tt.trees.expr_tree), 26
BooleanOperator (class in tt.definitions.operators), 16

C

CONSTANT_VALUES (in module tt.definitions.grammar), 16

D

DELIMITERS (in module tt.definitions.grammar), 16
DuplicateSymbolError, 18

E

EmptyExpressionError, 20
error_pos (tt.errors.grammar.GrammarError attribute), 20
eval_func (tt.definitions.operators.BooleanOperator attribute), 16
evaluate() (tt.expressions.bexpr.BooleanExpression method), 21
evaluate() (tt.trees.expr_tree.BooleanExpressionTree method), 26
evaluate() (tt.trees.tree_node.ExpressionTreeNode method), 27
evaluate_unchecked() (tt.expressions.bexpr.BooleanExpression method), 22
EvaluationError, 18
expr (tt.tables.truth_table.TruthTable attribute), 24
expr_str (tt.errors.grammar.GrammarError attribute), 21
ExpressionOrderError, 20
ExpressionTreeNode (class in tt.trees.tree_node), 27
ExtraSymbolError, 18

F

fill() (tt.tables.truth_table.TruthTable method), 24

G

get_parsed_args() (in module tt.cli.core), 15
GrammarError, 20

I

input_combos() (tt.tables.truth_table.TruthTable method), 25
InvalidArgumentTypeError, 19
InvalidBooleanValueError, 19

L

l_child (tt.trees.tree_node.ExpressionTreeNode attribute), 28

M

main() (in module tt.cli.core), 15
MAX_OPERATOR_STR_LEN (in module tt.definitions.operators), 17
message (tt.errors.base.TtError attribute), 18
MissingSymbolError, 19

N

name (tt.definitions.operators.BooleanOperator attribute), 16
NoEvaluationVariationError, 19

O

OperandExpressionTreeNode (class in tt.trees.tree_node), 28
operator (tt.trees.tree_node.BinaryOperatorExpressionTreeNode attribute), 27

operator (tt.trees.tree_node.UnaryOperatorExpressionTreeNode attribute), 28
OPERATOR_MAPPING (in module tt.definitions.operators), 17
ordering (tt.tables.truth_table.TruthTable attribute), 25

P

postfix_tokens (tt.expressions.bexpr.BooleanExpression attribute), 22
postfix_tokens (tt.trees.expr_tree.BooleanExpressionTree attribute), 27
precedence (tt.definitions.operators.BooleanOperator attribute), 17
print_err() (in module tt.cli.utils), 15
print_info() (in module tt.cli.utils), 16

R

r_child (tt.trees.tree_node.ExpressionTreeNode attribute), 28
raw_expr (tt.expressions.bexpr.BooleanExpression attribute), 22
results (tt.tables.truth_table.TruthTable attribute), 26
root (tt.trees.expr_tree.BooleanExpressionTree attribute), 27

S

symbol_name (tt.trees.tree_node.ExpressionTreeNode attribute), 28
symbols (tt.expressions.bexpr.BooleanExpression attribute), 22

T

tokens (tt.expressions.bexpr.BooleanExpression attribute), 23
tree (tt.expressions.bexpr.BooleanExpression attribute), 23
TruthTable (class in tt.tables.truth_table), 23
tt.cli (module), 15
tt.cli.core (module), 15
tt.cli.utils (module), 15
tt.definitions (module), 16
tt.definitions.grammar (module), 16
tt.definitions.operands (module), 16
tt.definitions.operators (module), 16
tt.errors (module), 18
tt.errors.base (module), 18
tt.errors.evaluation (module), 18
tt.errors.generic (module), 19
tt.errors.grammar (module), 20
tt.expressions (module), 21
tt.expressions.bexpr (module), 21
tt.tables (module), 23
tt.tables.truth_table (module), 23
tt.trees (module), 26
tt.trees.expr_tree (module), 26
tt.trees.tree_node (module), 27
tt.utils (module), 28
tt.utils.assertions (module), 28
TT_AND_OP (in module tt.definitions.operators), 17
TT_NAND_OP (in module tt.definitions.operators), 17
TT_NOR_OP (in module tt.definitions.operators), 17
TT_NOT_OP (in module tt.definitions.operators), 17
TT_OR_OP (in module tt.definitions.operators), 17
TT_XNOR_OP (in module tt.definitions.operators), 17
TT_XOR_OP (in module tt.definitions.operators), 17
TtError, 18

U

UnaryOperatorExpressionTreeNode (class in tt.trees.tree_node), 28
UnbalancedParenError, 21