# Python EDA Documentation

***Release 0.25.0***

**Chris Drake**

June 29, 2014

Contents

**Release** 0.25.0

**Date** June 29, 2014

PyEDA is a Python library for electronic design automation.

Fork PyEDA: https://github.com/cjdrake/pyeda

Features:

- Symbolic Boolean algebra with a selection of function representations:
  - Logic expressions
  - Truth tables, with three output states (0, 1, "don't care")
  - Reduced, ordered binary decision diagrams (ROBDDs)
- SAT solvers:
  - Backtracking
  - DPLL
  - PicoSAT
- Espresso logic minimization
- Formal equivalence
- Multi-dimensional bit vectors
- DIMACS CNF/SAT parsers
- Logic expression parser

CHAPTER 1

# Contents

## 1.1 Overview

### 1.1.1 What is Electronic Design Eutomation (EDA)?

The Intel 4004, the world's first commercially available microprocessor, was built from approximately 2300 transistors, and had a clock frequency of 740 kilohertz (thousands of cycles per second) [1] [2]. A modern Intel microprocessor can contain over 1.5 billion transistors, and will typically have a clock frequency ranging from two to four gigahertz (billions of cycles per second).

In 1971 it took less than one hundred people to manufacture the 4004. That is approximately 23 transistors per employee. If that ratio stayed the same between 1971 and 2012, Intel would need to employ about 65 *million* people just to produce the latest Core i7 processor. That is **one fifth** the entire population of the United States!

Clearly, companies that design and manufacture integrated circuits have found ways to be more productive since then.

Simply stated, electronic design automation (EDA) is the science of optimizing productivity in the design and manufacture of electronic components.

### 1.1.2 Goals

After reading the previous section, EDA sounds like a vast field. The way we have defined it covers everything from controlling robotic arms in the fabrication plant to providing free coffee to keep interns busy. We need to narrow our focus a bit.

PyEDA is primarily concerned with implementing the data structures and algorithms necessary for performing logic synthesis and verification. These tools form the theoretical foundation for the implementation of CAD tools for designing VLSI (Very Large Scale Integrated circuit).

PyEDA is a hobby project, and is very unlikely to ever be a competitor to state-of-the-art EDA industry technology. It should be useful for academic exploration and experimentation. If you use PyEDA, please email the author with your success/failure stories.

### 1.1.3 Free Software

PyEDA is free software; you can use it or redistribute it under the terms of the "two-clause" BSD License.

[1] Wikipedia: Intel 4004
[2] The Story of the Intel 4004

### 1.1.4 Repository

View the PyEDA source code on GitHub.

## 1.2 Installing PyEDA

This page describes how to procure your very own, shiny copy of PyEDA. It is a primary goal of the PyEDA project to be a mainstream Python package, and adhere to the majority of conventions observed by the community.

### 1.2.1 Supported Platforms

PyEDA supports Windows, and any platform with a C compiler. The author does most development and testing on Xubuntu Linux.

### 1.2.2 Supported Python Versions

Starting with version `0.15`, PyEDA will only work with Python 3.2+. Starting with version `0.23`, PyEDA will only work with Python 3.3+. There were several reasons to drop support for Python 2:

- Python 3 is the future of the language.
- Almost all scientific software either has already been ported, or is in the process of being ported to Python 3.
- Only Python 3 has support for the `def f(*args, kw1=val1, ...)` syntax, used to great effect by logic expression factory functions.
- It is too arduous to research and support all the C API changes from version 2 to version 3. Preprocessor is evil.

### 1.2.3 Distutils / Virtualenv

The latest PyEDA release is hosted on PyPI.

To get PyEDA with pip:

```
$ pip3 install pyeda
```

---

**Note:** If you are using the Linux system distribution of pip, most likely `pip` will be part of Python-2.x, which won't work. It's safer to always use `pip3`.

---

---

**Note:** If you are using a Windows wheel distribution, you may need to install the [Visual Studio 2012 Redistributable](http://www.microsoft.com/en-us/download/details.aspx?id=30679).

---

We *strongly* recommend that you also install an excellent Python tool called IPython. For interactive use, it is vastly superior to using the standard Python interpreter. To install IPython into your virtual environment:

```
$ pip3 install ipython
```

### 1.2.4 Getting the Source

The PyEDA repository is hosted on GitHub. If you want the bleeding-edge source code, here is how to get it:

```
$ git clone https://github.com/cjdrake/pyeda.git
$ cd pyeda
# $PREFIX is the root of the installation area
$ python setup.py install --prefix $PREFIX
```

If you want to build the documentation, you must have the excellent Sphinx documentaton system installed.

```
$ make html
```

If you want to run the tests, you must have the excellent Nose unit testing framework installed.

```
$ make test
.................................
----------------------------------------------------------------------
Ran 72 tests in 15.123s

OK
```

## 1.3 Boolean Algebra

Boolean Algebra is a cornerstone of electronic design automation, and fundamental to several other areas of computer science and engineering. PyEDA has an extensive library for the creation and analysis of Boolean functions.

This document describes how to explore Boolean algebra using PyEDA. We will be using some mathematical language here and there, but please do not run away screaming in fear. This document assumes very little background knowledge.

### 1.3.1 What is Boolean Algebra?

All great stories have a beginning, so let's start with the basics. You probably took a class called "algebra" in (junior) high school. So when you started reading this document you were already confused. Algebra is just algebra, right? You solve for $x$, find the intersection of two lines, and you're done, right?

As it turns out, the high school algebra you are familiar with just scratches the surface. There are many algebras with equally many theoretical and practical uses. An algebra is the combination of two things:

1. a collection of mathematical objects, and
2. a collection of rules to manipulate those objects

For example, in high school algebra, you have numbers such as $\{1, 3, 5, \frac{1}{2}, .337\}$, and operators such as $\{+, -, \cdot, \div\}$. The numbers are the mathematical objects, and the operators are the rules for how to manipulate them. Except in very extreme circumstances (division by zero), whenever you add, subtract, or divide two numbers, you get another number.

Algebras are a big part of the "tools of the trade" for a mathematician. A plumber has a wrench, a carpenter has a saw, and a mathematician has algebras. To each his own.

A *Boolean* algebra defines the rules for working with the set $\{0, 1\}$. So unlike in normal algebra class where you have more numbers than you can possibly imagine, in Boolean Algebra you only have two.

Even though it is possible to define a Boolean Algebra using different operators, by far the most common operators are complement, sum, and product.

### Complement Operator

The complement operator is a *unary* operator, which means it acts on a single Boolean input: $x$. The Boolean complement of $x$ is usually written as $x'$, $\overline{x}$, or $\neg x$.

The output of the Boolean complement is defined by:

$$\overline{0} = 1$$
$$\overline{1} = 0$$

### Sum Operator

The sum (or disjunction) operator is a *binary* operator, which means it acts on two Boolean inputs: $(x, y)$. The Boolean sum of $x$ and $y$ is usually written as $x + y$, or $x \vee y$.

The output of the Boolean sum is defined by:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

This looks familiar so far except for the $1 + 1 = 1$ part. The Boolean sum operator is also called **OR** because the output of $x$ *or* $y$ equals 1 *if and only if* $x = 1$, *or* $y = 1$, *or* both.

### Product Operator

The product (or conjunction) operator is also a *binary* operator. The Boolean product of $x$ and $y$ is usually written as $x \cdot y$, or $x \wedge y$.

The output of the Boolean product is defined by:

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

As you can see, the product operator looks exactly like normal multiplication. The Boolean product is also called **AND** because the output of $x$ *and* $y$ equals 1 *if and only if* both $x = 1$, *and* $y = 1$.

### Other Binary Operators

For reference, here is a table of all binary Boolean operators:

| $f$ | $g$ | 0 | $f \downarrow g$ | $f < g$ | $f'$ | $f > g$ | $g'$ | $f \neq g$ | $f \uparrow g$ | $f \cdot g$ | $f = g$ | $g$ | $f \leq g$ | $f$ | $f \geq g$ | $f + g$ | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Some additional notes:

- $f \downarrow g$ is the binary **NOR** (not or) operator.

- $f \uparrow g$ is the binary **NAND** (not and) operator.
- $f \leq g$ is commonly written using the binary implication operator $f \implies g$.
- $f = g$ is commonly written using either the binary equivalence operator $f \iff g$, or the binary **XNOR** (exclusive nor) operator $f \odot g$.
- $f \neq g$ is commonly written using the binary **XOR** (exclusive or) operator $f \oplus g$.

#### Additional Perspective

You are probably thinking this is all very nice, but what can you possibly do with an algebra that only concerns itself with 0, 1, **NOT**, **OR**, and **AND**?

In 1937, Claude Shannon realized that electronic circuits have two-value switches that can be combined into networks capable of solving any logical or numeric relationship. A transistor is nothing but an electrical switch. Similar to a light bulb, it has two states: off (0), and on (1). Wiring transistors together in serial imitates the **AND** operator, and wiring them together in parallel imitates the **OR** operator. If you wire a few thousand transistors together in interesting ways, you can build a computer.

### 1.3.2 Import Symbols from PyEDA

All examples in this document require that you execute the following statements in your interpreter:

```
>>> from pyeda.inter import *
```

If you want to see all the symbols you import with this statement, look into `pyeda/inter.py`.

---

**Note:** Using the `from ... import *` syntax is generally frowned upon for Python programming, but is *extremely* convenient for interactive use.

---

### 1.3.3 Built-in Python Boolean Operations

Python has a built-in Boolean data type, `bool`. You can think of the `False` keyword as an alias for the number 0, and the `True` keyword as an alias for the number 1.

```
>>> int(False)
0
>>> int(True)
1
>>> bool(0)
False
>>> bool(1)
True
```

The keywords for complement, sum, and product are `not`, `or`, `and`.

```
>>> not True
False
>>> True or False
True
>>> True and False
False
```

You can use the Python interpreter to evaluate complex expressions:

```
>>> (True and False) or not (False or True)
False
```

PyEDA recognizes `False`, `0`, and `'0'` as Boolean zero (0), and `True`, `1`, and `'1'` as Boolean one (1). You can use the `int` function to manually convert the `bool` and `str` data types to integers:

```
>>> int(True)
1
>>> int('0')
0
```

### 1.3.4 Boolean Variables

Okay, so we already know what Boolean Algebra is, and Python can already do everything we need, right?

Just like in high school algebra, things start to get interesting when we introduce a few *variables*.

A Boolean variable is an abstract numerical quantity that may assume any value in the set $B = \{0, 1\}$.

For example, if we flip a coin, the result will either be "heads" or "tails". Let's say we assign tails the value 0, and heads the value 1. Now divide all of time into two periods: 1) before the flip, and 2) after the flip.

Before you flip the coin, imagine the possibility of either "tails" (0) or "heads" (1). The abstract concept in your mind about a coin that may land in one of two ways is the *variable*. Normally, we will give the abstract quantity a name to distinguish it from other abstract quantities we might be simultaneously considering. The most familiar name for an arbitrary algebraic variable is $x$.

After you flip the coin, you can see the result in front of you. The coin flip is no longer an imaginary variable; it is a known constant.

#### Creating Variable Instances

Let's create a few Boolean expression variables using the `exprvar` method:

```
>>> a, b, c, d = map(exprvar, 'abcd')
>>> a.name
a
>>> b.name
b
```

By default, all variables go into a global namespace. Also, all variable instances are singletons. That is, only one variable is allowed to exist per name. Verify this fact with the following:

```
>>> a = exprvar('a')
>>> _a = exprvar('a')
>>> id(a) == id(_a)
True
```

> **Warning:** We recommend that you never do something crazy like assigning `a` and `_a` to the same variable instance.

#### Indexing Variables

> "There are only two hard things in Computer Science: cache invalidation and naming things."
>
> —Tim Bray

Consider the coin-flipping example from before. But this time, instead of flipping one coin, we want to flip a hundred coins. You could start naming your variables by assigning the first flip to $x$, followed by $y$, and so on. But there are only twenty-six letters in the English alphabet, so unless we start resorting to other alphabets, we will hit some limitations with this system very quickly.

For cases like these, it is convenient to give variables an *index*. Then, you can name the variable for the first coin flip $x[0]$, followed by $x[1]$, $x[2]$, and so on.

Here is how to give variables indices using the `exprvar` function:

```
>>> x_0 = exprvar('x', 0)
>>> x_1 = exprvar('x', 1)
>>> x_0, x_1
(x[0], x[1])
```

You can even give variables multiple indices by using a tuple:

```
>>> x_0_1_2_3 = exprvar('x', (0, 1, 2 ,3))
>>> x_0_1_2_3
x[0,1,2,3]
```

Assigning individual variables names like this is a bit cumbersome. It is much easier to just use the `exprvars` factory function:

```
>>> X = exprvars('x', 8)
>>> X
[x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]]
>>> X[3]
x[3]
>>> X[2:5]
[x[2], x[3], x[4]]
>>> X[:5]
[x[0], x[1], x[2], x[3], x[4]]
>>> X[5:]
[x[5], x[6], x[7]]
>>> X[-1]
x[7]
```

Similary for multi-dimensional bit vectors:

```
>>> X = exprvars('x', 4, 4)
>>> X
farray([[x[0,0], x[0,1], x[0,2], x[0,3]],
        [x[1,0], x[1,1], x[1,2], x[1,3]],
        [x[2,0], x[2,1], x[2,2], x[2,3]],
        [x[3,0], x[3,1], x[3,2], x[3,3]]])
>>> X[2]
farray([x[2,0], x[2,1], x[2,2], x[2,3]])
>>> X[2,2]
x[2,2]
>>> X[1:3]
farray([[x[1,0], x[1,1], x[1,2], x[1,3]],
        [x[2,0], x[2,1], x[2,2], x[2,3]]])
>>> X[1:3,2]
farray([x[1,2], x[2,2]])
>>> X[2,1:3]
farray([x[2,1], x[2,2]])
>>> X[-1,-1]
x[3,3]
```

### 1.3.5 Points in Boolean Space

Before we talk about Boolean functions, it will be useful to discuss the nature of Boolean space.

In high school algebra, you started with functions that looked like $f(x) = 2x + 3$. Later, you probably investigated slightly more interesting functions such as $f(x) = x^2$, $f(x) = sin(x)$, and $f(x) = e^x$. All of these are functions of a single variable. That is, the domain of these functions is the set of all values the variable $x$ can take. In all these cases, that domain is $[-\infty, +\infty]$.

Remember that variables in Boolean algebra can only take values of 0 or 1. So to create interesting functions in Boolean algebra, you use many variables.

Let's revisit the coin-flipping example again. This time we will flip the coin exactly twice. Create a variable $x$ to represent the result of the first flip, and a variable $y$ to represent the result of the second flip. Use zero (0) to represent "tails", and one (1) to represent "heads".

The number of variables you use is called the **dimension**. All the possible outcomes of this experiment is called the **space**. Each possible outcome is called a **point**.

If you flip the coin twice, and the result is (heads, tails), that result is point $(1, 0)$ in a 2-dimensional Boolean space.

Use the `iter_points` generator to iterate through all possible points in an N-dimensional Boolean space:

```
>>> list(iter_points([x, y]))
[{x: 0, y: 0}, {x: 1, y: 0}, {x: 0, y: 1}, {x: 1, y: 1}]
```

PyEDA uses a dictionary to represent a point. The keys of the dictionary are the variable instances, and the values are numbers in $0, 1$.

Try doing the experiment with three coin flips. Use the variable $z$ to represent the result of the third flip.

```
>>> list(iter_points([z, y, x]))
[{x: 0, y: 0, z: 0},
 {x: 0, y: 0, z: 1},
 {x: 0, y: 1, z: 0},
 {x: 0, y: 1, z: 1},
 {x: 1, y: 0, z: 0},
 {x: 1, y: 0, z: 1},
 {x: 1, y: 1, z: 0},
 {x: 1, y: 1, z: 1}]
```

The observant reader will notice that this is equivalent to:

- generating all bit-strings of length $N$
- counting from 0 to 7 in the binary number system

### 1.3.6 Boolean Functions

A Boolean function is a rule that maps points in an $N$-dimensional Boolean space to an element in $\{0, 1\}$. In formal mathematical lingo, $f : B^N \Rightarrow B$, where $B^N$ means the Cartesian product of $N$ Boolean variables, $v \in \{0, 1\}$. For example, if you have three input variables, $a, b, c$, then $B^3 = a \times b \times c = \{(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}$. $B^3$ is the **domain** of the function (the input part), and $B = \{0, 1\}$ is the **range** of the function (the output part).

In the relevant literature, you might see a Boolean function described as a type of **relation**, or geometrically as a **cube**. These are valid descriptions, but we will use a familiar analogy for better understanding.

A Boolean function is somewhat like a game, where the player takes $N$ binary input turns (eg heads/tails), and there is a binary result determined by a rule (eg win/loss). Let's revisit the coin-flipping example. One possible game is that we will flip the coin three times, and it will be considered a "win" if heads comes up all three.

Summarize the game using a **truth table**.

| $x_0$ | $x_1$ | $x_2$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

You can create the equivalent truth table with PyEDA like so:

```
>>> X = exprvars('x', 3)
>>> f = truthtable(X, "00000001")
>>> f
inputs: x[2] x[1] x[0]
000 0
001 0
010 0
011 0
100 0
101 0
110 0
111 1
```

Don't be alarmed that the inputs are displayed most-significant-bit first. That can actually come in handy sometimes.

The game from the previous example can be expressed as the expression $f(x_0, x_1, x_2) = x_0 \cdot x_1 \cdot x_2$. It is generally not convenient to list all the input variables, so we will normally shorten that statement to just $f = x_0 \cdot x_1 \cdot x_2$.

```
>>> truthtable2expr(f)
And(x[0], x[1], x[2])
```

Let's define another game with a slightly more interesting rule: "you win if the majority of flips come up heads".

| $x_0$ | $x_1$ | $x_2$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

This is a three-variable form of the "majority" function. You can express it as a truth table:

```
>>> f = truthtable(X, "00010111")
>>> f
inputs: x[2] x[1] x[0]
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
```

or as an expression:

```
>>> truthtable2expr(f)
Or(And(~x[0], x[1], x[2]), And(x[0], ~x[1], x[2]), And(x[0], x[1], ~x[2]), And(x[0], x[1], x[2]))
```

### 1.3.7 PyEDA Variable/Function Base Classes

Now that we have a better understanding of Boolean variables and functions, we will dive into how PyEDA models them.

We have already seen a glance of the type of data structure used to represent Boolean functions (tables and expressions). There are actually several of these representations, including (but not limited to):

- Truth tables
- Implicant tables
- Logic expressions
- Decision diagrams, including:
  - Binary decision diagrams (BDD)
  - Reduced, ordered binary decisions diagrams (ROBDD)
  - Zero-suppressed decision diagrams (ZDD)
- And inverter graphs (AIG)

Each data type has strengths and weaknesses. For example, ROBDDs are a canonical form, which make proofs of formal equivalence very cheap. On the other hand, ROBDDs can be exponential in size in many cases, which makes them memory-constrained.

The following sections show the abstract base classes for Boolean variables and functions defined in `pyeda.boolalg.boolfunc`.

#### Boolean Variables

In order to easily support algebraic operations on Boolean functions, each function representation has a corresponding variable representation. For example, truth table variables are instances of `TruthTableVariable`, and expression variables are instances of `ExpressionVariable`, both of which inherit from `Variable`.

**class** `pyeda.boolalg.boolfunc.`**`Variable`**(*names*, *indices*)

Base class for a symbolic Boolean variable.

A Boolean *variable* is an abstract numerical quantity that may assume any value in the set $B = \{0, 1\}$.

---

**Note:** Do **NOT** instantiate a `Variable` directly. Instead, use one of the concrete implementations: `pyeda.boolalg.bdd.bddvar()`, `pyeda.boolalg.expr.exprvar()`, `pyeda.boolalg.table.ttvar()`.

---

A variable is defined by one or more *names*, and zero or more *indices*. Multiple names establish hierarchical namespaces, and multiple indices group several related variables.

Each variable has a unique, positive integer called the *uniqid*. This integer may be used to identify a variable that is represented by more than one data structure. For example, `bddvar('a', 0)` and `exprvar('a', 0)` will refer to two different Variable instances, but both will share the same uniqid.

All variables are implicitly ordered. If two variable names are identical, compare their indices. Otherwise, do a string comparison of their names. This is only useful where variable order matters, and is not meaningful in any algebraic sense.

For example, the following statements are true:

- `a == a`
- `a < b`
- `a[0] < a[1]`
- `a[0][1] < a[0][2]`

**name**
Return the innermost variable name.

**qualname**
Return the fully qualified name.

### Boolean Functions

This is the abstract base class for Boolean function representations.

In addition to the methods and properties listed below, classes inheriting from `Function` should also overload the `__invert__`, `__or__`, `__and__`, and `__xor__` magic methods. This makes it possible to perform symbolic, algebraic manipulations using a Python interpreter.

**class** `pyeda.boolalg.boolfunc.`**Function**
Abstract base class that defines an interface for a symbolic Boolean function of $N$ variables.

**support**
Return the support set of a function.

Let $f(x_1, x_2, \ldots, x_n)$ be a Boolean function of $N$ variables.

The unordered set $\{x_1, x_2, \ldots, x_n\}$ is called the *support* of the function.

**usupport**
Return the untyped support set of a function.

**inputs**
Return the support set in name/index order.

**top**
Return the first variable in the ordered support set.

**degree**
Return the degree of a function.

A function from $B^N \Rightarrow B$ is called a Boolean function of *degree* $N$.

**cardinality**
Return the cardinality of the relation $B^N \Rightarrow B$.

Always equal to $2^N$.

**iter_domain**`()`
Iterate through all points in the domain.

**iter_image**`()`
Iterate through all elements in the image.

**iter_relation**()
Iterate through all (point, element) pairs in the relation.

**restrict**(*point*)
Restrict a subset of support variables to $\{0, 1\}$.

Returns a new function: $f(x_i, \ldots)$

$f \mid x_i = b$

**vrestrict**(*vpoint*)
Expand all vectors in *vpoint* before applying `restrict`.

**compose**(*mapping*)
Substitute a subset of support variables with other Boolean functions.

Returns a new function: $f(g_i, \ldots)$

$f_1 \mid x_i = f_2$

**satisfy_one**()
If this function is satisfiable, return a satisfying input point. A tautology *may* return a zero-dimensional point; a contradiction *must* return None.

**satisfy_all**()
Iterate through all satisfying input points.

**satisfy_count**()
Return the cardinality of the set of all satisfying input points.

**iter_cofactors**(*vs=None*)
Iterate through the cofactors of a function over N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $f_{x_i} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i'$ is: $f_{x_i'} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$

**cofactors**(*vs=None*)
Return a tuple of the cofactors of a function over N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $f_{x_i} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i'$ is: $f_{x_i'} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$

**smoothing**(*vs=None*)
Return the smoothing of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *smoothing* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $S_{x_i}(f) = f_{x_i} + f_{x_i'}$

This is the same as the existential quantification operator: $\exists\{x_1, x_2, \ldots\}\, f$

**consensus**(*vs=None*)
Return the consensus of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *consensus* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $C_{x_i}(f) = f_{x_i} \cdot f_{x_i'}$

This is the same as the universal quantification operator: $\forall\{x_1, x_2, \ldots\}\, f$

**derivative**(*vs=None*)
Return the derivative of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *derivative* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $\frac{\partial}{\partial x_i} f = f_{x_i} \oplus f_{x_i'}$

This is also known as the Boolean *difference*.

**is_zero**()
Return whether this function is zero.

---

**Note:** This method will only look for a particular "zero form", and will **NOT** do a full search for a contradiction.

---

**is_one**()
Return whether this function is one.

---

**Note:** This method will only look for a particular "one form", and will **NOT** do a full search for a tautology.

---

**static box**(*obj*)
Convert primitive types to a Function.

**unbox**()
Return integer 0 or 1 if possible, otherwise return the Function.

## 1.4 Binary Decision Diagrams

A binary decision diagram is a directed acyclic graph used to represent a Boolean function. They were originally introduced by Lee [3], and later by Akers [4]. In 1986, Randal Bryant introduced the reduced, ordered BDD (ROBDD) [5].

The ROBDD is a *canonical* form, which means that given an identical ordering of input variables, equivalent Boolean functions will always reduce to the same ROBDD. This is a very desirable property for determining formal equivalence. Also, it means that unsatisfiable functions will be reduced to zero, making SAT/UNSAT calculations trivial. Due to these auspicious properties, the term BDD almost always refers to some minor variation of the ROBDD devised by Bryant.

In this chapter, we will discuss how to construct and visualize ROBDDs using PyEDA.

The code examples in this chapter assume that you have already prepared your terminal by importing all interactive symbols from PyEDA:

```
>>> from pyeda.inter import *
```

### 1.4.1 Constructing BDDs

There are two ways to construct a BDD:

1. Convert an expression
2. Use operators on existing BDDs

[3] C.Y. Lee, *Representation of Switching Circuits by Binary-Decision Programs*, Bell System Technical Journal, Vol. 38, July 1959, pp. 985-999.
[4] S.B. Akers, *Binary Decision Diagrams*, IEEE Transactions on Computers, Vol. C-27, No. 6, June 1978, pp. 509-516.
[5] Randal E. Bryant *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, 1986 http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

### Convert an Expression

Since the Boolean expression is PyEDA's central data type, you can use the `expr2bdd` function to convert arbitrary expressions to BDDs:

```
>>> f = expr("a & b | a & c | b & c")
>>> f
Or(And(a, b), And(a, c), And(b, c))
>>> f = expr2bdd(f)
>>> f
<pyeda.boolalg.bdd.BinaryDecisionDiagram at 0x7f556874ed68>
```

As you can see, the BDD does not have such a useful string representation. More on this subject later.

### Using Operators

Just like expressions, BDDs have their own `Variable` implementation. You can use the `bddvar` function to construct them:

```
>>> a, b, c = map(bddvar, 'abc')
>>> type(a)
pyeda.boolalg.bdd.BDDVariable
>>> isinstance(a, BinaryDecisionDiagram)
True
```

Creating indexed variables with namespaces always works just like expressions:

```
>>> a0 = bddvar('a', 0)
>>> a0
a[0]
>>> b_a_0_1 = bddvar(('a', 'b'), (0, 1))
b.a[0,1]
```

Also, the `bddvars` function can be used to create multi-dimensional arrays of indexed variables:

```
>>> X = bddvars('x', 4, 4)
>>> X
farray([[x[0,0], x[0,1], x[0,2], x[0,3]],
        [x[1,0], x[1,1], x[1,2], x[1,3]],
        [x[2,0], x[2,1], x[2,2], x[2,3]],
        [x[3,0], x[3,1], x[3,2], x[3,3]]])
```

From variables, you can use Python's `~ | & ^` operators to construct arbitrarily large BDDs.

Here is the simple majority function again:

```
>>> f = a & b | a & c | b & c
>>> f
<pyeda.boolalg.bdd.BinaryDecisionDiagram at 0x7f556874ed68>
```

This time, we can see the benefit of having variables available:

```
>>> f.restrict({a: 0})
<pyeda.boolalg.bdd.BinaryDecisionDiagram at 0x7f556874eb38>
>>> f.restrict({a: 1, b: 0})
c
>>> f.restrict({a: 1, b: 1})
1
```

### 1.4.2 BDD Visualization with IPython and GraphViz

If you have GraphViz installed on your machine, and the `dot` executable is available on your shell's path, you can use the `gvmagic` IPython extension to visualize BDDs.

```
In [1]: %install_ext https://raw.github.com/cjdrake/ipython-magic/master/gvmagic.py

In [2]: %load_ext gvmagic
```

For example, here is the majority function in three variables as a BDD:

```
In [3]: a, b, c = map(bddvar, 'abc')

In [4]: f = a & b | a & c | b & c

In [5]: %dotobj f
```

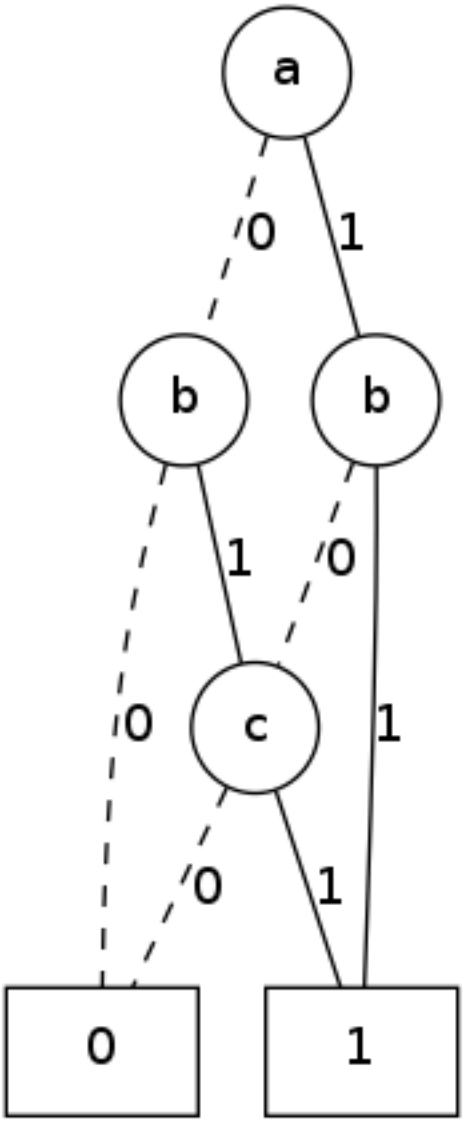



Figure 1.1: BDD of Three-Input Majority Function

The way this works is that the `%dotobj` extension internally calls the `to_dot` method on `f`:

```
In [6]: f.to_dot()
'graph BDD {
    n139865543613912 [label=0,shape=box];
    n139865543728208 [label=1,shape=box];
    n139865543169752 [label="c",shape=circle];
    n139865552542296 [label="b",shape=circle];
    n139865543169864 [label="b",shape=circle];
    n139865543170312 [label="a",shape=circle];
    n139865543169752 -- n139865543613912 [label=0,style=dashed];
    n139865543169752 -- n139865543728208 [label=1];
    n139865552542296 -- n139865543613912 [label=0,style=dashed];
    n139865552542296 -- n139865543169752 [label=1];
    n139865543169864 -- n139865543169752 [label=0,style=dashed];
    n139865543169864 -- n139865543728208 [label=1];
    n139865543170312 -- n139865552542296 [label=0,style=dashed];
    n139865543170312 -- n139865543169864 [label=1]; }'
```

### 1.4.3 Satisfiability

Like we mentioned in the introduction, BDDs are a canonical form. That means that all unsatisfiable functions will reduce to zero, and all tautologies will reduce to one. If you simply want to check whether a function is SAT or UNSAT, just construct a BDD, and test whether it is zero/one.

```
>>> a, b = map(bddvar, 'ab')
>>> f = ~a & ~b | ~a & b | a & ~b | a & b
>>> f
1
>>> f.is_one()
True
>>> g = (~a | ~b) & (~a | b) & (a | ~b) & (a | b)
>>> g
0
>>> g.is_zero()
True
```

If you need one or more satisfying input points, use the `satisfy_one` and `satisfy_all` functions. The algorithm that implements SAT is very simple and elegant; it just finds a path from the function's root node to one.

```
>>> a, b, c = map(bddvar, 'abc')
>>> f = a ^ b ^ c
>>> f.satisfy_one()
{b: 0, a: 0, c: 1}
>>> list(f.satisfy_all())
[{a: 0, b: 0, c: 1},
 {a: 0, b: 1, c: 0},
 {a: 1, b: 0, c: 0},
 {a: 1, b: 1, c: 1}]
```

Trace all that paths from the top node to `1` to verify.

### 1.4.4 Formal Equivalence

Because BDDs are a canonical form, functional equivalence is trivial.

Here is an example where we define the XOR function by using 1) the XOR operator, and 2) OR/AND/NOT operators.

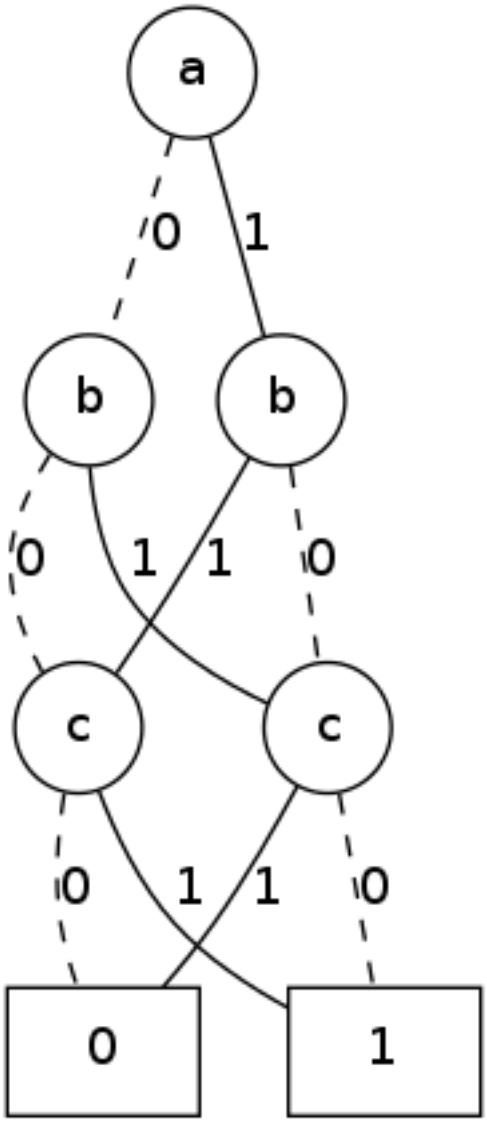



Figure 1.2: BDD of Three-Input XOR Function

```
>>> a, b, c = map(bddvar, 'abc')
>>> f1 = a ^ b ^ c
>>> f2 = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c
```

Just like expressions, BDDs have an `equivalent` method:

```
>>> f1.equivalent(f2)
True
```

However, this isn't required. PyEDA maintains a unique table of BDD nodes and their function pointers, so you can just test for equality using the Python `is` operator:

```
>>> f1 is f2
True
```

### 1.4.5 Variable Ordering

The size of a BDD is very sensitive to the order of variable decomposition. For example, here is a BDD that uses an ideal variable order:

```
In [1]: X = bddvars('x', 8)

In [2]: f1 = X[0] & X[1] | X[2] & X[3] | X[4] & X[5]

In [3]: %dotobj f1
```

And here is the same function, with a bad variable order:

```
In [2]: f2 = X[0] & X[3] | X[1] & X[4] | X[2] & X[5]

In [3]: %dotobj f2
```

The previous example was used by Bryant [3] to demonstrate this concept. When you think of the definition of a BDD, it becomes clear why some orderings are superior to others. What you want in a variable ordering is to decide as much of the function at every decision level as you traverse towards `0` and `1`.

PyEDA implicitly orders all variables. It is therefore not possible to create a new BDD by reordering its inputs. You can, however, rename the variables using the `compose` method to achieve the desired result.

For example, to optimize the previous BDD:

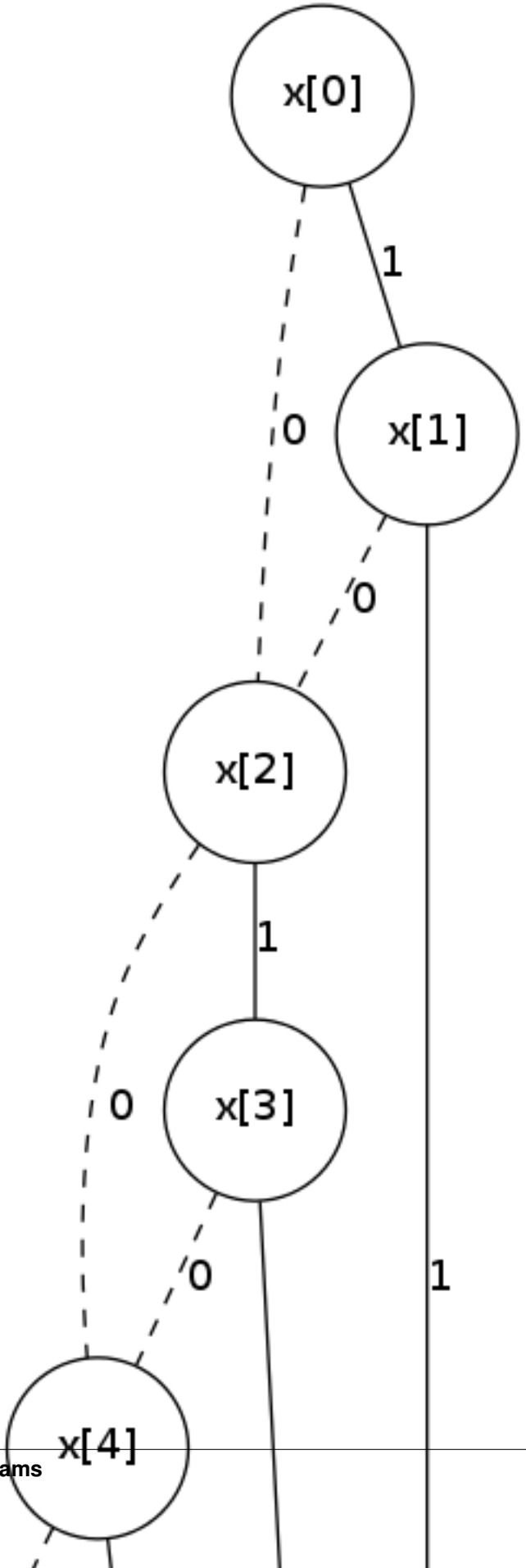
```
In [4]: g2 = f2.compose({X[0]: Y[0], X[1]: Y[2], X[2]: Y[4],
                         X[3]: Y[1], X[4]: Y[3], X[5]: Y[5]})

In [5]: %dotobj g2
```

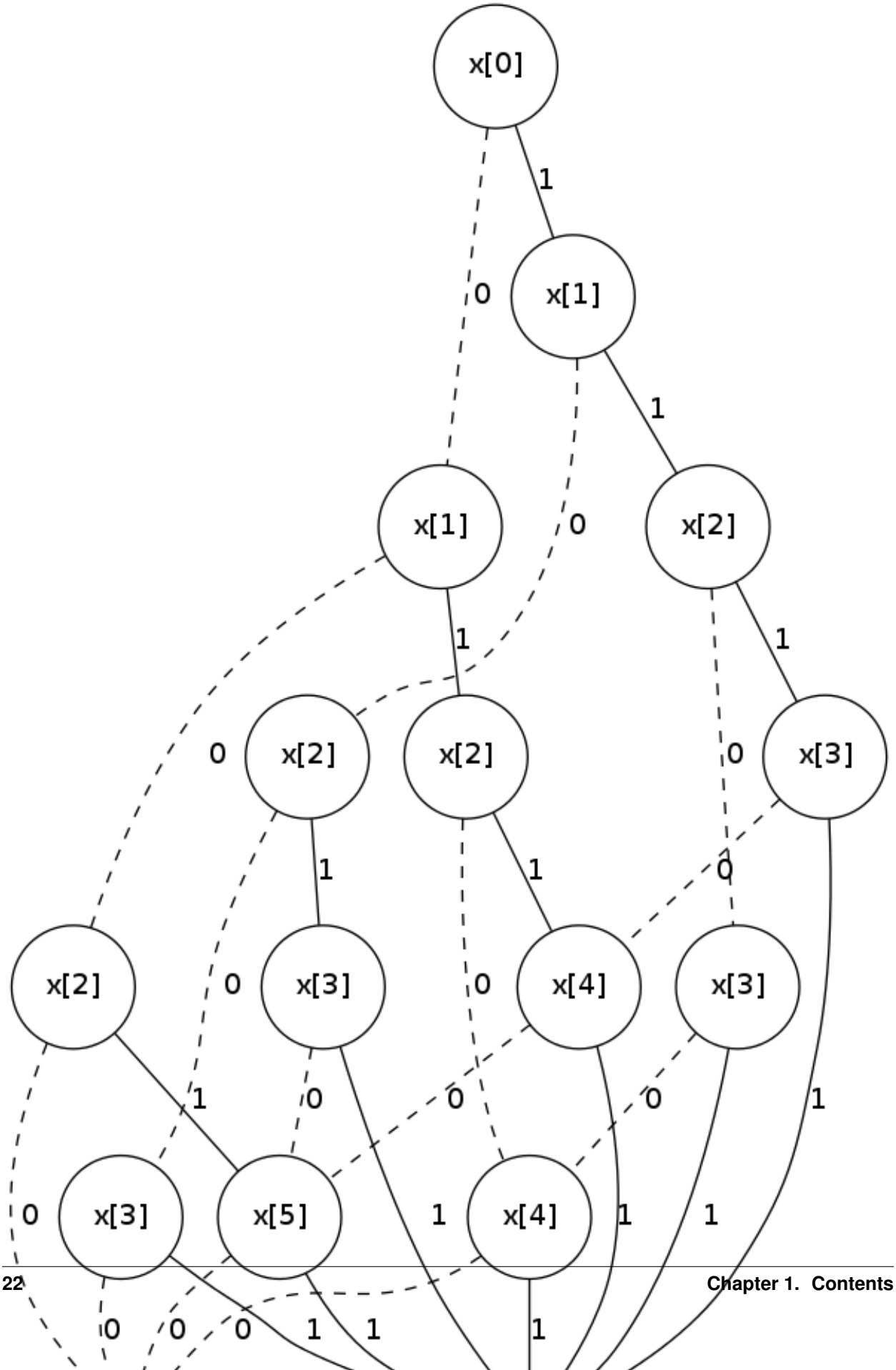
### 1.4.6 Garbage Collection

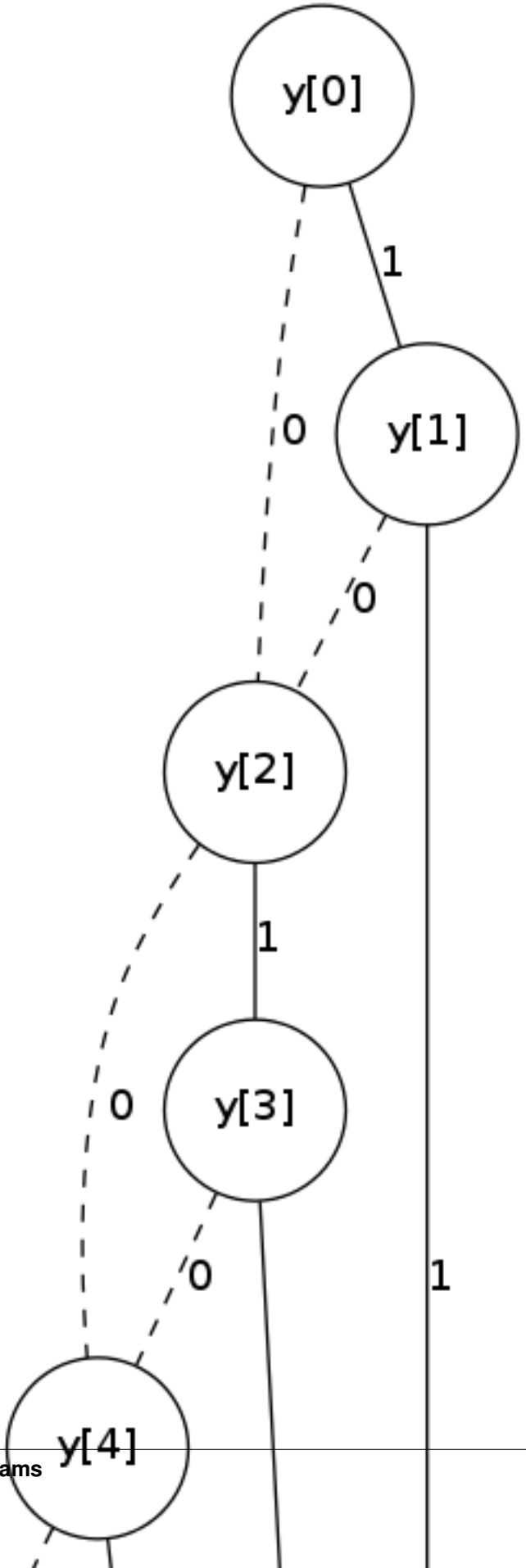
Since BDDs are a memory-constrained data structure, the subject of garbage collection is very important.

PyEDA uses the Python standard library's weakref module to automatically garbage collect BDD nodes when they are no longer needed. The BDD function contains a reference to a node, which contains references to it's children, and so on until you get to zero/one. When a function's name is either deleted or it goes out of scope, it may initiate a corresponding cascade of node deletions.

This is best illustrated with an example. If you look directly into the `pyeda.boolalg.bdd` module, you can find the memory structure that holds BDD nodes:


x[0]
1
0
x[1]
0
x[2]
1
0
x[3]
0
1
x[4]


x[0]
1
0
x[1]
1
x[1]
0
x[2]
0
1
1
0
x[2]
x[2]
0
x[3]
1
1
0
x[2]
0
x[3]
0
x[4]
x[3]
1
0
0
0
1
0
x[3]
x[5]
1
x[4]
1
1
0
0
0
0
1
1
1


y[0]
1
0
y[1]
0
y[2]
1
0
y[3]
0
1
y[4]

```
>>> from pyeda.boolalg.bdd import _BDDNODES
>>> len(_BDDNODES)
2
```

The table contains two static nodes: zero and one. Let's define a few variables:

```
>>> from pyeda.inter import *
>>> a, b = map(bddvar, 'ab')
>>> len(_BDDNODES)
4
```

Now define three simple BDDs:

```
>>> f1 = a | b
>>> len(_BDDNODES)
5
>>> f2 = a & b
>>> len(_BDDNODES)
6
>>> f3 = a ^ b
>>> len(_BDDNODES)
8
```

Now there are eight nodes. Let's count the remaining nodes as we delete functions:

```
>>> del f1
>>> len(_BDDNODES)
7
>>> del f2
>>> len(_BDDNODES)
6
>>> del f3
>>> len(_BDDNODES)
4
```

### 1.4.7 References

## 1.5 Boolean Expressions

Expressions are a very powerful and flexible way to represent Boolean functions. They are the central data type of PyEDA's symbolic Boolean algebra engine. This chapter will explain how to construct and manipulate Boolean expressions.

The code examples in this chapter assume that you have already prepared your terminal by importing all interactive symbols from PyEDA:

```
>>> from pyeda.inter import *
```

### 1.5.1 Expression Constants

You can represent 0 and 1 as expressions:

```
>>> zero = expr(0)
>>> one = expr(1)
```

We will describe the `expr` function in more detail later. For now, let's just focus on the representation of constant values.

All of the following conversions from 0 have the same effect:

```
>>> zeroA = expr(0)
>>> zeroB = expr(False)
>>> zeroC = expr("0")
```

All three "zero" objects are the same:

```
>>> zeroA == zeroB == zeroC
True
```

Similarly for 1:

```
>>> oneA = expr(1)
>>> oneB = expr(True)
>>> oneC = expr("1")
```

All three "one" objects are the same:

```
>>> oneA == oneB == oneC
True
```

However, these results might come as a surprise:

```
>>> expr(0) == 0
False
>>> expr(1) == True
False
```

PyEDA evaluates all zero-like and one-like objects, and stores them internally using a module-level singleton in `pyeda.boolalg.expr`:

```
>>> type(expr(0))
pyeda.boolalg.expr._ExprZero
>>> type(expr(1))
pyeda.boolalg.expr._ExprOne
```

Once you have converted zero/one to expressions, they implement the full Boolean Function API.

For example, constants have an empty support set:

```
>>> one.support
frozenset()
```

Also, apparently zero is not satisfiable:

```
>>> zero.satisfy_one() is None
True
```

This fact might seem underwhelming, but it can have some neat applications. For example, here is a sneak peak of Shannon expansions:

```
>>> a, b = map(exprvar, 'ab')
>>> zero.expand([a, b], conj=True)
And(Or(a, b), Or(a, ~b), Or(~a, b), Or(~a, ~b))
>>> one.expand([a, b])
Or(And(~a, ~b), And(~a, b), And(a, ~b), And(a, b))
```

### 1.5.2 Expression Literals

A Boolean *literal* is defined as a variable or its complement. The expression variable and complement data types are the primitives of Boolean expressions.

#### Variables

To create expression variables, use the `exprvar` function.

For example, let's create a variable named $a$, and assign it to a Python object named `a`:

```
>>> a = exprvar('a')
>>> type(a)
pyeda.boolalg.expr.ExprVariable
```

One efficient method for creating multiple variables is to use Python's builtin `map` function:

```
>>> a, b, c = map(exprvar, 'abc')
```

The primary benefit of using the `exprvar` function rather than a class constructor is to ensure that variable instances are unique:

```
>>> a = exprvar('a')
>>> a_new = exprvar('a')
>>> id(a) == id(a_new)
True
```

You can name a variable pretty much anything you want, though we recommend standard identifiers:

```
>>> foo = exprvar('foo')
>>> holy_hand_grenade = exprvar('holy_hand_grenade')
```

By default, all variables go into a global namespace. You can assign a variable to a specific namespace by passing a tuple of strings as the first argument:

```
>>> a = exprvar('a')
>>> c_b_a = exprvar(('a', 'b', 'c'))
>>> a.names
('a', )
>>> c_b_a.names
('a', 'b', 'c')
```

Notice that the default representation of a variable will dot all the names together starting with the most significant index of the tuple on the left:

```
>>> str(c_b_a)
'c.b.a'
```

Since it is very common to deal with grouped variables, you may assign indices to variables as well. Each index is a new dimension.

To create a variable with a single index, use an integer argument:

```
>>> a42 = exprvar('a', 42)
>>> str(a42)
a[42]
```

To create a variable with multiple indices, use a tuple argument:

```
>>> a_1_2_3 = exprvar('a', (1, 2, 3))
>>> str(a_1_2_3)
a[1,2,3]
```

Finally, you can combine multiple namespaces and dimensions:

```
>>> c_b_a_1_2_3 = exprvar(('a', 'b', 'c'), (1, 2, 3))
>>> str(c_b_a_1_2_3)
c.b.a[1,2,3]
```

---

**Note:** The previous syntax is starting to get a bit cumbersome. For a more powerful method of creating multi-dimensional arrays, use the `exprvars` function.

---

#### Complements

A complement is defined as the inverse of a variable. That is:

$$a + a' = 1$$
$$a \cdot a' = 0$$

One way to create a complement from a pre-existing variable is to simply apply Python's ~ unary negate operator.

For example, let's create a variable and its complement:

```
>>> a = exprvar('a')
>>> ~a
~a
>>> type(~a)
pyeda.boolalg.expr.ExprComplement
```

All complements created from the same variable instance are not just identical, they all refer to the same object:

```
>>> id(~a) == id(~a)
True
```

### 1.5.3 Constructing Expressions

Expressions are defined recursively as being composed of primitives (constants, variables), and expressions joined by Boolean operators.

Now that we are familiar with all of PyEDA's Boolean primitives, we will learn how to construct more interesting expressions.

#### From Constants, Variables, and Python Operators

PyEDA overloads Python's ~, |, ^, and & operators with NOT, OR, XOR, and AND, respectively.

Let's jump in by creating a full adder:

```
>>> a, b, ci = map(exprvar, "a b ci".split())
>>> s = ~a & ~b & ci | ~a & b & ~ci | a & ~b & ~ci | a & b & ci
>>> co = a & b | a & ci | b & ci
```

Using XOR looks a lot nicer for the sum output:

```
>>> s = a ^ b ^ ci
```

You can use the `expr2truthtable` function to do a quick check that the sum logic is correct:

```
>>> expr2truthtable(s)
inputs: ci b a
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
```

Similarly for the carry out logic:

```
>>> expr2truthtable(co)
inputs: ci b a
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
```

### From Factory Functions

Python does not have enough builtin operators to handle all interesting Boolean functions we can represent directly as an expression. Also, binary operators are limited to two operands at a time, whereas several Boolean operators are N-ary (arbitrary many operands). This section will describe all the factory functions that can be used to create arbitrary Boolean expressions.

The general form of these functions is `OP(arg [, arg], simplify=True)`. The function is an operator name, followed by one or more arguments, followed by the `simplify` parameter. Some functions also have a `conj` parameter, which selects between conjunctive (`conj=True`) and disjunctive (`conj=False`) formats.

One advantage of using these functions is that you do not need to create variable instances prior to passing them as arguments. You can just pass string identifiers, and PyEDA will automatically parse and convert them to variables.

For example, the following two statements are equivalent:

```
>>> Not('a[0]')
~a[0]
```

and:

```
>>> a0 = exprvar('a', 0)
>>> Not(a0)
~a[0]
```

#### Primary Operators

Since NOT, OR, and AND form a complete basis for a Boolean algebra, these three operators are *primary*.

**Not** (*arg*, *simplify=True*)
Return an expression that is the inverse of the input.

**Or** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if *any* inputs are 1.

**And** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if *all* inputs are 1.

Example of full adder logic using `Not`, `Or`, and `And`:

```
>>> s = Or(And(Not('a'), Not('b'), 'ci'), And(Not('a'), 'b', Not('ci')), And('a', Not('b'), Not('ci'
>>> co = Or(And('a', 'b'), And('a', 'ci'), And('b', 'ci'))
```

### Secondary Operators

A *secondary* operator is a Boolean operator that can be natively represented as a PyEDA expression, but contains more information than the primary operators. That is, these expressions always increase in tree size when converted to primary operators.

**Nor** (**args*, *simplify=True*)
Return an expression that evaluates to 0 if and only if *any* inputs are 1. The inverse of *Or*.

**Nand** (**args*, *simplify=True*)
Return an expression that evaluates to 0 if an only if *all* inputs are 1. The inverse of *And*.

**Xor** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if the input parity is odd.

The word **parity** in this context refers to whether the number of ones in the input is even (divisible by two). For example, the input string "0101" has *even* parity (2 ones), and the input string "0001" has *odd* parity (1 ones).

The full adder circuit has a more dense representation when you use the `Xor` operator:

```
>>> s = Xor('a', 'b', 'ci')
>>> co = Or(And('a', 'b'), And('a', 'ci'), And('b', 'ci'))
```

**Xnor** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if the input parity is even.

**Equal** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if all inputs are equivalent.

**Unequal** (**args*, *simplify=True*)
Return an expression that evaluates to 1 if and only if *not* all inputs are equivalent.

**Implies** (*p*, *q*, *simplify=True*)
Return an expression that implements Boolean implication ($p \implies q$).

| $p$ | $q$ | $p \implies q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Note that this truth table is equivalent to $p \leq q$, but Boolean implication is by far the more common form due to its use in propositional logic.

**ITE** (*s*, *d1*, *d0*, *simplify=True*)
Return an expression that implements the Boolean "if, then, else" operator. If $s = 1$, then the output equals $d_0$. Otherwise ($s = 0$), the output equals $d_1$.

| $s$ | $d_1$ | $d_0$ | $ite(s, d_1, d_0)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

### High Order Operators

A *high order* operator is a Boolean operator that can NOT be natively represented as a PyEDA expression. That is, these factory functions will always return expressions composed from primary and/or secondary operators.

**OneHot0** (**args*, *simplify=True*, *conj=True*)
Return an expression that evaluates to 1 if and only if the number of inputs equal to 1 is at most 1. That is, return true when at most one input is "hot".

**OneHot** (**args*, *simplify=True*, *conj=True*)
Return an expression that evaluates to 1 if and only if exactly one input is equal to 1. That is, return true when exactly one input is "hot".

**Majority** (**args*, *simplify=True*, *conj=False*)
Return an expression that evaluates to 1 if and only if the majority of inputs equal 1.

The full adder circuit has a much more dense representation when you use both the `Xor` and `Majority` operators:

```
>>> s = Xor('a', 'b', 'ci')
>>> co = Majority('a', 'b', 'ci')
```

**AchillesHeel** (**args*, *simplify=True*)
Return the Achille's Heel function, defined as $\prod_{i=0}^{N-1} (x_{i/2} + x_{i/2+1})$.

The `AchillesHeel` function has $N$ literals in its CNF form, but $N/2 \times 2^{N/2}$ literals in its DNF form. It is an interesting demonstration of tradeoffs when choosing an expression representation. For example:

```
>>> f =  AchillesHeel('a', 'b', 'c', 'd', 'w', 'x', 'y', 'z')
>>> f
And(Or(a, b), Or(c, d), Or(w, x), Or(y, z))
>>> f.to_dnf()
Or(And(a, c, w, y), And(a, c, w, z), And(a, c, x, y), And(a, c, x, z), And(a, d, w, y), And(a, d, w,
```

**Mux** (*fs*, *sel*, *simplify=True*)
Return an expression that multiplexes a sequence of input functions over a sequence of select functions.

For example:

```
>>> X = exprvars('x', 4)
>>> S = exprvars('s', 2)
>>> Mux(X, S)
Or(And(~s[0], ~s[1], x[0]), And(s[0], ~s[1], x[1]), And(~s[0], s[1], x[2]), And(s[0], s[1], x[3]))
```

### From the `expr` Function

**expr** (*arg*, *simplify=True*)

The `expr` function is very special. It will attempt to convert the input argument to an `Expression` object.

We have already seen how the `expr` function converts a Python `bool` input to a constant expression:

```
>>> expr(False)
0
```

Now let's pass a `str` as the input argument:

```
>>> expr('0')
0
```

If given an input string, the `expr` function will attempt to parse the input string and return an expression.

Examples of input expressions:

```
>>> expr("~a & b | ~c & d")
Or(And(~a, b), And(~c, d))
>>> expr("a | b ^ c & d")
Or(a, Xor(b, And(c, d)))
>>> expr("p => q")
Implies(p, q)
>>> expr("p <=> q")
Equal(p, q)
>>> expr("s ? d[1] : d[0]")
ITE(s, d[1], d[0])
>>> expr("Or(a, b, Not(c))")
Or(a, b, ~c)
>>> expr("Majority(a, b, c)")
Or(And(a, b), And(a, c), And(b, c))
```

Operator Precedence Table (lowest to highest):

| Operator | Description |
|---|---|
| `s ? d1 : d0` | If Then Else |
| `=> <=>` | Binary Implies, Binary Equal |
| `\|` | Binary OR |
| `^` | Binary XOR |
| `&` | Binary AND |
| `~x` | Unary NOT |
| `(expr ...) OP(expr ...)` | Parenthesis, Explicit operators |

The full list of valid operators accepted by the expression parser:

- `Or(...)`
- `And(...)`
- `Xor(...)`
- `Xnor(...)`
- `Equal(...)`
- `Unequal(...)`
- `Nor(...)`
- `Nand(...)`
- `OneHot0(...)`
- `OneHot(...)`
- `Majority(...)`

- `AchillesHeel(...)`
- `ITE(s, d1, d0)`
- `Implies(p, q)`
- `Not(a)`

### 1.5.4 Expression Types

This section will cover the hierarchy of Boolean expression types.

#### Unsimplified

An unsimplified expression consists of the following components:

- Constants
- Expressions that can *easily* be converted to constants (eg $x + x' = 1$)
- Literals
- Primary operators: `Not`, `Or`, `And`
- Secondary operators

Also, an unsimplified expression does not automatically join adjacent, associative operators. For example, $a + (b + c)$ is equivalent to $a + b + c$. The depth of the unsimplified expression is two:

```
>>> f = Or('a', Or('b', 'c'), simplify=False)
>>> f.args
(a, Or(b, c))
>>> f.depth
2
```

The depth of the simplified expression is one:

```
>>> g = f.simplify()
>>> g.args
(b, a, c)
>>> g.depth
1
```

If the `simplify=False` usage is too verbose, you can use the `Expression` subclasses directly to create unsimplified expressions:

```
>>> ExprAnd(a, ~a)
And(~a, a)
>>> ExprOr(a, ExprOr(b, c))
Or(a, Or(b, c))
```

Notice that there are no unsimplified representations for:

- degenerate forms
- negated literals
- double negation

For example:

```
>>> ExprOr()
0
>>> ExprNot(a)
~a
>>> ExprNot(ExprNand(a, b))
And(a, b)
```

### Simplifed

A simplified expression consists of the following components:

- Literals
- Primary operators: `Not`, `Or`, `And`
- Secondary operators

Also, 0 and 1 are considered simplified by themselves.

That is, the act of *simplifying* an expression eliminates constants, and all sub-expressions that can be easily converted to constants.

All expressions constructed using overloaded operatiors are automatically simplified:

```
>>> a | 0
a
>>> a | 1
1
>>> a | b & ~b
a
```

Unsimplified expressions are not very useful, so the factory functions also simplify by default:

```
>>> Or(a, And(b, ~b))
a
```

To simplify an expression, use the `simplify` method:

```
>>> f = Or(a, 0, simplify=False)
>>> f
Or(0, a)
>>> g = f.simplify()
>>> g
a
```

You can check whether an expression is simplified using the `simplified` attribute:

```
>>> f.simplified
False
>>> g.simplified
True
```

### Factored

A factored expression consists of the following components:

- Literals
- Primary operators: `Or`, `And`

That is, the act of *factoring* an expression converts all secondary operators to primary operators, and uses DeMorgan's transform to eliminate `Not` operators.

You can factor all secondary operators:

```
>>> Xor(a, b, c).factor()
Or(And(~a, ~b, c), And(~a, b, ~c), And(a, ~b, ~c), And(a, b, c))
>>> Implies(p, q).factor()
Or(~p, q)
>>> Equal(a, b, c).factor()
Or(And(~a, ~b, ~c), And(a, b, c))
>>> ITE(s, a, b).factor()
Or(And(b, Or(a, b, ~ci), Or(a, ~b, ci)), And(a, Or(And(~a, ~b, ci), And(~a, b, ~ci))))
```

Factoring also eliminates all `Not` operators, by using DeMorgan's law:

```
>>> Not(a | b).factor()
And(~a, ~b)
>>> Not(a & b).factor()
Or(~a, ~b)
```

### Normal Form

A normal form expression is a factored expression with depth less than or equal to two.

That is, a normal form expression has been factored, and *flattened*.

There are two types of normal forms:

- disjunctive normal form (SOP: sum of products)
- conjunctive normal form (POS: product of sums)

The preferred method for creating normal form expressions is to use the `to_dnf` and `to_cnf` methods:

```
>>> f = Xor(a, Implies(b, c))
>>> f.to_dnf()
Or(And(~a, ~b), And(~a, c), And(a, b, ~c))
>>> f.to_cnf()
And(Or(~a, b), Or(~a, ~c), Or(a, ~b, c))
```

### Canonical Normal Form

A canonical normal form expression is a normal form expression with the additional property that all terms in the expression have the same degree as the expression itself.

That is, a canonical normal form expression has been factored, flattened, and *reduced*.

The preferred method for creating canonical normal form expressions is to use the `to_cdnf` and `to_ccnf` methods.

Using the same function from the previous section as an example:

```
>>> f = Xor(a, Implies(b, c))
>>> f.to_cdnf()
Or(And(~a, ~b, ~c), And(~a, ~b, c), And(~a, b, c), And(a, b, ~c))
>>> f.to_ccnf()
And(Or(a, ~b, c), Or(~a, b, c), Or(~a, b, ~c), Or(~a, ~b, ~c))
```

### 1.5.5 Depth

Expressions are a type of tree data structure. The depth of a tree is defined recursively:

1. A leaf node (constant or literal) has zero depth.
2. A branch node (operator) has depth equal to the maximum depth of its children (arguments) plus one.

You can think of the depth as the maximum number of operators between the expression's inputs to its output.

For example:

```
>>> a.depth
0
>>> (a & b).depth
1
>>> Xor(a, Implies(b, c)).depth
2
```

### 1.5.6 Satisfiability

Expressions have smart support for Boolean satisfiability.

They inherit both the `satisfy_one` and `satisfy_all` methods from the `Function` interface.

For example:

```
>>> f = Xor('a', Implies('b', 'c'))
>>> f.satisfy_one()
{a: 0, b: 0}
>>> list(f.satisfy_all())
[{a: 0, b: 0}, {a: 0, b: 1, c: 1}, {a: 1, b: 1, c: 0}]
```

By default, Boolean expressions use a very naive "backtracking" algorithm to solve for satisfying input points. Since SAT is an NP-complete algorithm, you should always use care when preparing your inputs.

A conjunctive normal form expression will automatically use the PicoSAT C extension. This is an industrial-strength SAT solver, and can be used to solve very non-trivial problems.

```
>>> g = f.to_cnf()
>>> g.satisfy_one()
{a: 0, b: 0, c: 1}
>>> list(g.satisfy_all())
[{a: 0, b: 1, c: 1},
 {a: 0, b: 0, c: 0},
 {a: 0, b: 0, c: 1},
 {a: 1, b: 1, c: 0}]
```

---

**Note:** Future versions of PyEDA might support additional C/C++ extensions for SAT solving. This is an active area of research, and no single solver is ideal for all cases.

---

#### Assumptions

A common pattern in SAT-solving is to setup a large database of clauses, and attempt to solve the CNF several times with different simplifying assumptions. This is equivalent to adding unit clauses to the database, which can be easily eliminated by boolean constraint propagation.

The `Expression` data type supports applying assumptions using the `with` statement. Here is an example of creating a nested context of literals that assumes `a=1` and `b=0`:

```
>>> f = Xor(a, b, c)
>>> with a, ~b:
...     print(f.satisfy_one())
{a: 1, b: 0, c: 0}
```

There are four satisfying solutions to this function, but the return value will always correspond to the input assumptions.

`And` terms of literals (clauses) may also be used as assumptions:

```
>>> with a & ~b:
...     print(f.satisfy_one())
{a: 1, b: 0, c: 0}
```

Note that it is an error to assume conflicting values for a literal:

```
>>> with a, ~a:
...     print(f.satisfy_one())
ValueError: conflicting constraints: a, ~a
```

### PicoSAT Script

Starting with version `0.23.0`, PyEDA includes a script that implements some of the functionality of the PicoSAT command-line utility.

```
$ picosat -h
usage: picosat [-h] [--version] [-n] [-a LIT] [-v] [-i {0,1,2,3}] [-P LIMIT]
               [-l LIMIT] [--all]
               [file]

PicoSAT solver

positional arguments:
  file          CNF input file (default: stdin)

optional arguments:
  -h, --help    show this help message and exit
  --version     print version and exit
  ...
```

Here is an example of a basic SAT problem encoded using the well-known DIMACS CNF format. See http://www.satcompetition.org/ for details on expected inputs and outputs. The function is a one-hot function on four input variables.

```
$ cat onehot4.cnf
c Comments start with a 'c' character
c The problem 'p' line is followed by clauses
p cnf 4 7
1 2 3 4 0
-4 -3 0
-4 -2 0
-4 -1 0
-3 -2 0
-3 -1 0
-2 -1 0
```

To get one satisfying solution:

```
$ picosat onehot4.cnf
s SATISFIABLE
v -1 -2 -3 4 0
```

To get all satisfying solutions:

```
$ picosat --all onehot4.cnf
s SATISFIABLE
v -1 -2 -3 4 0
s SATISFIABLE
v -1 -2 3 -4 0
s SATISFIABLE
v -1 2 -3 -4 0
s SATISFIABLE
v 1 -2 -3 -4 0
s SOLUTIONS 4
```

You can also constrain the solver with one or more assumptions:

```
$ picosat -a 1 -a -2 onehot4.cnf
s SATISFIABLE
v 1 -2 -3 -4 0
$ picosat -a 1 -a 2 onehot4.cnf
s UNSATISFIABLE
```

### 1.5.7 Tseitin's Encoding

To take advantage of the PicoSAT solver, you need an expression that is in conjunctive normal form. Some expressions (especially Xor) have exponentially large size when you expand them to a CNF.

One way to work around this limitation is to use Tseitin's encoding. To convert an expression to Tseitin's encoding, use the `tseitin` method:

```
>>> f = Xor('a', Implies('b', 'c'))
>>> tf = f.tseitin()
>>> tf
And(Or(a, ~aux[0]), Or(~a, aux[0], ~b, c), Or(~a, ~aux[2]), Or(a, ~aux[1], aux[2]), Or(aux[0], aux[2
```

As you can see, Tseitin's encoding introduces several "auxiliary" variables into the expression.

You can change the name of the auxiliary variable by using the `auxvarname` parameter:

```
>>> f = Xor('a', Implies('b', 'c'))
>>> f.tseitin(auxvarname='z')
And(Or(a, ~z[0]), Or(~a, ~b, c, z[0]), Or(~a, ~z[2]), Or(a, ~z[1], z[2]), Or(b, z[1]), Or(~b, c, ~z[1
```

You will see the auxiliary variables in the satisfying points:

```
>>> tf.satisfy_one()
{a: 0, aux[0]: 0, aux[1]: 1, aux[2]: 1, b: 0, c: 1}
>>> list(tf.satisfy_all())
[{a: 1, aux[0]: 0, aux[1]: 0, aux[2]: 1, b: 1, c: 0},
 {a: 0, aux[0]: 1, aux[1]: 1, aux[2]: 0, b: 1, c: 1},
 {a: 0, aux[0]: 1, aux[1]: 1, aux[2]: 0, b: 0, c: 0},
 {a: 0, aux[0]: 1, aux[1]: 1, aux[2]: 0, b: 0, c: 1}]
```

Just filter them out to get the answer you're looking for:

```
>>> [{v: val for v, val in point.items() if v.name != 'aux'} for point in tf.satisfy_all()]
[{a: 1, b: 1, c: 0},
 {a: 0, b: 1, c: 1},
 {a: 0, b: 0, c: 0},
 {a: 0, b: 0, c: 1}]
```

### 1.5.8 Formal Equivalence

Two Boolean expressions $f$ and $g$ are formally equivalent if $f \oplus g$ is not satisfiable.

Boolean expressions have an `equivalent` method that implements this basic functionality. It uses the naive backtracking SAT, because it is difficult to determine whether any particular expression can be converted efficiently to a CNF.

Let's test whether bit 6 of a ripple carry adder is equivalent to bit 6 of a Kogge Stone adder:

```
>>> from pyeda.logic.addition import ripple_carry_add, kogge_stone_add
>>> A = exprvars('a', 16)
>>> B = exprvars('b', 16)
>>> S1, C1 = ripple_carry_add(A, B)
>>> S2, C2 = kogge_stone_add(A, B)
>>> S1[6].equivalent(S2[6])
True
```

Note that this is the same as the following:

```
>>> Xor(S1[6], S2[6]).satisfy_one() is None
True
```

## 1.6 Function Arrays

When dealing with several related Boolean functions, it is usually convenient to index the inputs and outputs. For this purpose, PyEDA includes a multi-dimensional array (MDA) data type, called an `farray` (function array).

The most pervasive example is computation involving any numeric data type. For example, let's say you want to add two numbers $A$, and $B$. If these numbers are 32-bit integers, there are 64 total inputs, not including a carry-in. The conventional way of labeling the input variables is $a_0, a_1, a_2, \ldots$, and $b_0, b_1, b_2, \ldots$.

Furthermore, you can extend the symbolic algebra of Boolean functions to arrays. For example, the element-wise XOR or $A$ and $B$ is also an array

This chapter will explain how to construct and manipulate arrays of Boolean functions. Typical examples will be one-dimensional vectors, but you can shape your data into several dimensions if you like. We have purposefully adopted many of the conventions used by the numpy.array module.

The code examples in this chapter assume that you have already prepared your terminal by importing all interactive symbols from PyEDA:

```
>>> from pyeda.inter import *
```

### 1.6.1 Construction

There are three ways to construct `farray` instances:

1. Use the `farray` class constructor.

2. Use one of the zeros/ones/vars factory functions.
3. Use the unsigned/signed integer conversion functions.

### Constructor

First, create a few expression variables:

```
>>> a, b, c, d = map(exprvar, 'abcd')
```

To store these variables in an array, invoke the `farray` constructor on a sequence:

```
>>> vs = farray([a, b, c, d])
>>> vs
farray([a, b, c, d])
>>> vs[2]
c
```

Now let's store some arbitrary functions into an array. Create six expressions using the secondary operators:

```
>>> f0 = Nor(a, b, c)
>>> f1 = Nand(a, b, c)
>>> f2 = Xor(a, b, c)
>>> f3 = Xnor(a, b, c)
>>> f4 = Equal(a, b, c)
>>> f5 = Unequal(a, b, c)
```

To neatly store these six functions in an `farray`, invoke the constructor like we did before:

```
>>> F = farray([f0, f1, f2, f3, f4, f5])
>>> F
farray([Nor(a, b, c), Nand(a, b, c), Xor(a, b, c), Xnor(a, b, c), Equal(a, b, c), Unequal(a, b, c)])
```

This is sufficient for 1-D arrays, but the `farray` constructor can create mult-dimensional arrays as well. You can apply shape to the input array by either using a nested sequence, or manually using the *shape* parameter.

To create a 2x3 array using a nested sequence:

```
>>> F = farray([[f0, f2, f4], [f1, f3, f5]])
>>> F
farray([[Nor(a, b, c), Xor(a, b, c), Equal(a, b, c)],
        [Nand(a, b, c), Xnor(a, b, c), Unequal(a, b, c)]])
>>> F.shape
((0, 2), (0, 3))
>>> F.size
6
>>> F[0,1]
Xor(a, b, c)
```

Similarly for a 3x2 array:

```
>>> F = farray([[f0, f1], [f2, f3], [f4, f5]])
>>> F
farray([[Nor(a, b, c), Nand(a, b, c)],
        [Xor(a, b, c), Xnor(a, b, c)],
        [Equal(a, b, c), Unequal(a, b, c)]])
>>> F.shape
((0, 3), (0, 2))
>>> F.size
6
```

```
>>> F[0,1]
Nand(a, b, c)
```

Use the *shape* parameter to manually impose a shape. It takes a tuple of *dimension* specs, which are `(start, stop)` tuples.

```
>>> F = farray([f0, f1, f2, f3, f4, f5], shape=((0, 2), (0, 3)))
>>> F
farray([[Nor(a, b, c), Xor(a, b, c), Equal(a, b, c)],
        [Nand(a, b, c), Xnor(a, b, c), Unequal(a, b, c)]])
```

Internally, function arrays are stored in a flat list. You can retrieve the items by using the *flat* iterator:

```
>>> list(F.flat)
[Nor(a, b, c),
 Nand(a, b, c),
 Xor(a, b, c),
 Xnor(a, b, c),
 Equal(a, b, c),
 Unequal(a, b, c)]
```

Use the *reshape* method to return a new `farray` with the same contents and size, but with different dimensions:

```
>>> F.reshape(3, 2)
farray([[Nor(a, b, c), Nand(a, b, c)],
        [Xor(a, b, c), Xnor(a, b, c)],
        [Equal(a, b, c), Unequal(a, b, c)]])
```

#### Empty Arrays

It is possible to create an empty `farray`, but only if you supply the *ftype* parameter. That parameter is not necessary for non-empty arrays, because it can be automatically determined.

For example:

```
>>> empty = farray([], ftype=Expression)
>>> empty
farray([])
>>> empty.shape
((0, 0),)
>>> empty.size
0
```

#### Irregular Shapes

Without the *shape* parameter, array dimensions will be created with start index zero. This is fine for most applications, but in digital design it is sometimes useful to create arrays with irregular starting points.

For example, let's say you are designing the load/store unit of a CPU. A computer's memory is addressed in *bytes*, but data is accessed from memory in *cache lines*. If the size of your machine's cache line is 64 bits, data will be retrieved from memory eight bytes at a time. The lower 3 bits of the address bus going from the load/store unit to main memory are not necessary. Therefore, your load/store unit will output an address with one dimension bounded by `(3, 32)`, i.e. all address bits starting from 3, up to but not including 32.

Going back to the previous example, let's say for some reason we wanted a shape of `((7, 9), (13, 16))`. Just change the *shape* parameter:

```
>>> F = farray([f0, f1, f2, f3, f4, f5], shape=((7, 9), (13, 16)))
>>> F
farray([[Nor(a, b, c), Xor(a, b, c), Equal(a, b, c)],
        [Nand(a, b, c), Xnor(a, b, c), Unequal(a, b, c)]])
```

The *size* property is still the same:

```
>>> F.size
6
```

However, the slices now have different bounds:

```
>>> F.shape
((7, 9), (13, 16))
>>> F[7,14]
Nand(a, b, c)
```

### Factory Functions

For convenience, PyEDA provides factory functions for producing arrays with arbitrary shape initialized to all zeros, all ones, or all variables with incremental indices.

The functions for creating arrays of zeros are:

- pyeda.boolalg.bfarray.bddzeros()
- pyeda.boolalg.bfarray.exprzeros()
- pyeda.boolalg.bfarray.ttzeros()

For example, to create a 4x4 farray of expression zeros:

```
>>> zeros = exprzeros(4, 4)
>>> zeros
farray([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

For example:

```
>>> zeros = bddzeros((1, 3), (2, 4), 2)
>>> zeros
farray([[[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]]])
```

Similarly for creating arrays of ones:

- pyeda.boolalg.bfarray.bddones()
- pyeda.boolalg.bfarray.exprones()
- pyeda.boolalg.bfarray.ttones()

The functions for creating arrays of variables are:

- pyeda.boolalg.bfarray.bddvars()
- pyeda.boolalg.bfarray.exprvars()
- pyeda.boolalg.bfarray.ttvars()

These functions behave similarly to the zeros/ones functions, but take a *name* argument as well.

For example, to create a 4x4 farray of expression variables:

```
>>> A = exprvars('a', 4, 4)
>>> A
farray([[a[0,0], a[0,1], a[0,2], a[0,3]],
        [a[1,0], a[1,1], a[1,2], a[1,3]],
        [a[2,0], a[2,1], a[2,2], a[2,3]],
        [a[3,0], a[3,1], a[3,2], a[3,3]]])
```

The *name* argument accepts a tuple of names, just like the `exprvar` function, and the variadic *dims* input also supports irregular shapes:

```
>>> A = exprvars(('a', 'b', 'c'), (1, 3), (2, 4), 2)
>>> A
farray([[[c.b.a[1,2,0], c.b.a[1,2,1]],
         [c.b.a[1,3,0], c.b.a[1,3,1]]],

        [[c.b.a[2,2,0], c.b.a[2,2,1]],
         [c.b.a[2,3,0], c.b.a[2,3,1]]]])
```

### Integer Conversion

The previous section discussed ways to initialize arrays to all zeros or ones. It is also possible to create one-dimensional arrays that represent integers using either the unsigned or twos-complement notations.

The following functions convert an unsigned integer to a 1-D `farray`:

- pyeda.boolalg.bfarray.uint2bdds()
- pyeda.boolalg.bfarray.uint2exprs()
- pyeda.boolalg.bfarray.uint2tts()

The following functions convert a signed integer to a 1-D `farray`:

- pyeda.boolalg.bfarray.int2bdds()
- pyeda.boolalg.bfarray.int2exprs()
- pyeda.boolalg.bfarray.int2tts()

The signature for these functions are all identical. The *num* argument is the `int` to convert, and the *length* parameter is optional. Unsigned conversion will always zero-extend to the provided length, and signed conversion will always sign-extend.

Here are a few examples of converting integers to expressions:

```
>>> uint2exprs(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
>>> int2exprs(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
# A nifty one-liner to verify the previous conversions
>>> bin(42)[2:].zfill(8)[::-1]
'01010100'
```

```
>>> int2exprs(-42, 8)
farray([0, 1, 1, 0, 1, 0, 1, 1])
```

Function arrays also have `to_uint` and `to_int` methods to perform the reverse computation. They do not, however, have any property to indicate whether the array represents signed data. So always know what the encoding is ahead of time. For example:

```
>>> int2exprs(-42, 8).to_int()
-42
>>> int2exprs(-42, 8).to_uint()
214
```

### 1.6.2 Slicing

The `farray` type accepts two types of slice arguments:

- Integral indices
- Muliplexor selects

#### Integral Indices

Function arrays support a slice syntax that mostly follows the numpy ndarray data type. The primary difference is that `farray` supports nonzero start indices.

To demonstrate the various capabilities, let's create some arrays. For simplicity, we will only use zero indexing.

```
>>> A = exprvars('a', 4)
>>> B = exprvars('b', 4, 4, 4)
```

Using a single integer index will *collapse* an array dimension. For 1-D arrays, this means returning an item.

```
>>> A[2]
a[2]
>>> B[2]
farray([[b[2,0,0], b[2,0,1], b[2,0,2], b[2,0,3]],
        [b[2,1,0], b[2,1,1], b[2,1,2], b[2,1,3]],
        [b[2,2,0], b[2,2,1], b[2,2,2], b[2,2,3]],
        [b[2,3,0], b[2,3,1], b[2,3,2], b[2,3,3]]])
>>> B[2].shape
((0, 4), (0, 4))
```

The colon `:` slice syntax *shrinks* a dimension:

```
>>> A[:]
farray([a[0], a[1], a[2], a[3]])
>>> A[1:]
farray([a[1], a[2], a[3]])
>>> A[:3]
farray([a[0], a[1], a[2]])
>>> B[1:3]
farray([[[b[1,0,0], b[1,0,1], b[1,0,2], b[1,0,3]],
         [b[1,1,0], b[1,1,1], b[1,1,2], b[1,1,3]],
         [b[1,2,0], b[1,2,1], b[1,2,2], b[1,2,3]],
         [b[1,3,0], b[1,3,1], b[1,3,2], b[1,3,3]]],

        [[b[2,0,0], b[2,0,1], b[2,0,2], b[2,0,3]],
         [b[2,1,0], b[2,1,1], b[2,1,2], b[2,1,3]],
```

```
         [b[2,2,0], b[2,2,1], b[2,2,2], b[2,2,3]],
         [b[2,3,0], b[2,3,1], b[2,3,2], b[2,3,3]]])
```

For N-dimensional arrays, the slice accepts up to N indices separated by a comma. Unspecified slices at the end will default to `:`.

```
>>> B[1,2,3]
b[1,2,3]
>>> B[:,2,3]
farray([b[0,2,3], b[1,2,3], b[2,2,3], b[3,2,3]])
>>> B[1,:,3]
farray([b[1,0,3], b[1,1,3], b[1,2,3], b[1,3,3]])
>>> B[1,2,:]
farray([b[1,2,0], b[1,2,1], b[1,2,2], b[1,2,3]])
>>> B[1,2]
farray([b[1,2,0], b[1,2,1], b[1,2,2], b[1,2,3]])
```

The `...` syntax will fill available indices left to right with `:`. Only one ellipsis will be recognized per slice.

```
>>> B[...,1]
farray([[b[0,0,1], b[0,1,1], b[0,2,1], b[0,3,1]],
        [b[1,0,1], b[1,1,1], b[1,2,1], b[1,3,1]],
        [b[2,0,1], b[2,1,1], b[2,2,1], b[2,3,1]],
        [b[3,0,1], b[3,1,1], b[3,2,1], b[3,3,1]]])
>>> B[1,...]
farray([[b[1,0,0], b[1,0,1], b[1,0,2], b[1,0,3]],
        [b[1,1,0], b[1,1,1], b[1,1,2], b[1,1,3]],
        [b[1,2,0], b[1,2,1], b[1,2,2], b[1,2,3]],
        [b[1,3,0], b[1,3,1], b[1,3,2], b[1,3,3]]])
```

Function arrays support negative indices. Arrays with a zero start index follow Python's usual conventions.

For example, here is the index guide for `A[0:4]`:

```
 +------+------+------+------+
 | a[0] | a[1] | a[2] | a[3] |
 +------+------+------+------+
 0      1      2      3      4
-4     -3     -2     -1
```

And example usage:

```
>>> A[-1]
a[3]
>>> A[-3:-1]
farray([a[1], a[2]])
```

Arrays with non-zero start indices also support negative indices. For example, here is the index guide for `A[3:7]`:

```
 +------+------+------+------+
 | a[3] | a[4] | a[5] | a[6] |
 +------+------+------+------+
 3      4      5      6      7
-4     -3     -2     -1
```

### Multiplexor Selects

A special feature of array slicing is the ability to multiplex array items over a select input. For a 2:1 mux, the *select* may be a single function. Otherwise, it must be an `farray` with enough bits.

For example, to create a simple 8:1 mux:

```
>>> X = exprvars('x', 8)
>>> sel = exprvars('s', 3)
>>> X[sel]
Or(And(~s[0], ~s[1], ~s[2], x[0]),
   And( s[0], ~s[1], ~s[2], x[1]),
   And(~s[0],  s[1], ~s[2], x[2]),
   And( s[0],  s[1], ~s[2], x[3]),
   And(~s[0], ~s[1],  s[2], x[4]),
   And( s[0], ~s[1],  s[2], x[5]),
   And(~s[0],  s[1],  s[2], x[6]),
   And( s[0],  s[1],  s[2], x[7]))
```

This works for multi-dimensional arrays as well:

```
>>> s = exprvar('s')
>>> Y = exprvars('y', 2, 2, 2)
>>> Y[:,s,:]
farray([[Or(And(~s, y[0,0,0]),
            And( s, y[0,1,0])),
         Or(And(~s, y[0,0,1]),
            And( s, y[0,1,1]))],

        [Or(And(~s, y[1,0,0]),
            And( s, y[1,1,0])),
         Or(And(~s, y[1,0,1]),
            And( s, y[1,1,1]))]])
```

### 1.6.3 Operators

Function arrays support several operators for algebraic manipulation. Some of these operators overload Python's operator symbols. This section will describe how you can use the `farray` data type and the Python interpreter to perform powerful symbolic computations.

#### Unary Reduction

A common operation is to reduce the entire contents of an array to a single function. This is supported by the OR, AND, and XOR operators because they are 1) variadic, and 2) associative.

Unfortunately, Python has no appropriate symbols available, so unary operators are supported by the following `farray` methods:

- `pyeda.boolalg.bfarray.farray.uor()`
- `pyeda.boolalg.bfarray.farray.unor()`
- `pyeda.boolalg.bfarray.farray.uand()`
- `pyeda.boolalg.bfarray.farray.unand()`
- `pyeda.boolalg.bfarray.farray.uxor()`
- `pyeda.boolalg.bfarray.farray.uxnor()`

For example, to OR the contents of an eight-bit array:

```
>>> X = exprvars('x', 8)
>>> X.uor()
Or(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])
```

One well-known usage of unary reduction is conversion from a binary-reflected gray code (BRGC) back to binary. In the following example, `B` is a 3-bit array that contains logic to convert the contents of `G` from gray code to binary. See the Wikipedia Gray Code article for background.

```
>>> G = exprvars('g', 3)
>>> B = farray([G[i:].uxor() for i, _ in enumerate(G)])
>>> graycode = ['000', '100', '110', '010', '011', '111', '101', '001']
>>> for gs in graycode:
...     print(B.vrestrict({X: gs}).to_uint())
0
1
2
3
4
5
6
7
```

### Bit-wise Logic

Arrays are an algebraic data type. They overload several of Python's operators to perform bit-wise logic.

First, let's create a few arrays:

```
>>> A = exprvars('a', 4)
>>> B = exprvars('b', 4)
>>> C = exprvars('c', 2, 2)
>>> D = exprvars('d', 2, 2)
```

To invert the contents of `A`:

```
>>> ~A
farray([~a[0], ~a[1], ~a[2], ~a[3]])
```

Inverting a multi-dimensional array will retain its shape:

```
>>> ~C
farray([[~c[0,0], ~c[0,1]],
        [~c[1,0], ~c[1,1]]])
```

The binary OR, AND, and XOR operators work for arrays with equal size:

```
>>> A | B
farray([Or(a[0], b[0]), Or(a[1], b[1]), Or(a[2], b[2]), Or(a[3], b[3])])
>>> A & B
farray([And(a[0], b[0]), And(a[1], b[1]), And(a[2], b[2]), And(a[3], b[3])])
>>> C ^ D
farray([[Xor(c[0,0], d[0,0]), Xor(c[0,1], d[0,1])],
        [Xor(c[1,0], d[1,0]), Xor(c[1,1], d[1,1])]])
```

Mismatched sizes will raise an exception:

```
>>> A & B[2:]
Traceback (most recent call last):
```

```
    ...
ValueError: expected operand sizes to match
```

For arrays of the same size but different shape, the resulting shape is ambiguous so by default the result is flattened:

```
>>> Y = ~A | C
>>> Y
farray([Or(~a[0], c[0,0]), Or(~a[1], c[0,1]), Or(~a[2], c[1,0]), Or(~a[3], c[1,1])])
>>> Y.size
4
>>> Y.shape
((0, 4),)
```

### Shifts

Function array have three shift methods:

- `pyeda.boolalg.bfarray.farray.lsh()`: logical left shift
- `pyeda.boolalg.bfarray.farray.rsh()`: logical right shift
- `pyeda.boolalg.bfarray.farray.arsh()`: arithmetic right shift

The logical left/right shift operators shift out *num* items from the array, and optionally shift in values from a *cin* (carry-in) parameter. The output is a two-tuple of the shifted array, and the "carry-out".

The "left" direction in `lsh` shifts towards the most significant bit. For example:

```
>>> X = exprvars('x', 8)
>>> X.lsh(4)
(farray([0, 0, 0, 0, x[0], x[1], x[2], x[3]]),
 farray([x[4], x[5], x[6], x[7]]))
>>> X.lsh(4, exprvars('y', 4))
(farray([y[0], y[1], y[2], y[3], x[0], x[1], x[2], x[3]]),
 farray([x[4], x[5], x[6], x[7]]))
```

Similarly, the "right" direction in `rsh` shifts towards the least significant bit. For example:

```
>>> X.rsh(4)
(farray([x[4], x[5], x[6], x[7], 0, 0, 0, 0]),
 farray([x[0], x[1], x[2], x[3]]))
>>> X.rsh(4, exprvars('y', 4))
(farray([x[4], x[5], x[6], x[7], y[0], y[1], y[2], y[3]]),
 farray([x[0], x[1], x[2], x[3]]))
```

You can use the Python overloaded << and >> operators for `lsh`, and `rsh`, respectively. The only difference is that they do not produce a carry-out. For example:

```
>>> X << 4
farray([0, 0, 0, 0, x[0], x[1], x[2], x[3]])
>>> X >> 4
farray([x[4], x[5], x[6], x[7], 0, 0, 0, 0])
```

Using a somewhat awkward `(num, farray)` syntax, you can use these operators with a carry-in. For example:

```
>>> X << (4, exprvars('y', 4))
farray([y[0], y[1], y[2], y[3], x[0], x[1], x[2], x[3]])
>>> X >> (4, exprvars('y', 4))
farray([x[4], x[5], x[6], x[7], y[0], y[1], y[2], y[3]])
```

An *arithmetic* right shift automatically sign-extends the array. Therefore, it does not take a carry-in. For example:

```
>>> X.arsh(4)
(farray([x[4], x[5], x[6], x[7], x[7], x[7], x[7], x[7]]),
 farray([x[0], x[1], x[2], x[3]]))
```

Due to its importance in digital design, Verilog has a special >>> operator for an arithmetic right shift. Sadly, Python has no such indulgence. If you really want to use a symbol, you can use the *cin* parameter to achieve the same effect with >>:

```
>>> num = 4
>>> X >> (num, num * X[-1])
farray([x[4], x[5], x[6], x[7], x[7], x[7], x[7], x[7]])
```

### Concatenation and Repetition

Two very important operators in hardware description languages are concatenation and repetition of logic vectors. For example, in this implementation of the `xtime` function from the AES standard, `xtime[6:0]` is concatenated with `1'b0`, and `xtime[7]` is repeated eight times before being AND'ed with `8'h1b`.

```
function automatic logic [7:0]
xtime(logic [7:0] b, int n);
    xtime = b;
    for (int i = 0; i < n; i++)
        xtime = {xtime[6:0], 1'b0}       // concatenation
              ^ (8'h1b & {8{xtime[7]}}); // repetition
endfunction
```

The `farray` data type resembles the Python `tuple` for these operations.

To concatenate two arrays, use the + operator:

```
>>> X = exprvars('x', 4)
>>> Y = exprvars('y', 4)
>>> X + Y
farray([x[0], x[1], x[2], x[3], y[0], y[1], y[2], y[3]])
```

It is also possible to prepend or append single functions:

```
>>> a, b = map(exprvar, 'ab')
>>> a + X
farray([a, x[0], x[1], x[2], x[3]])
>>> X + b
farray([x[0], x[1], x[2], x[3], b])
>>> a + X + b
farray([a, x[0], x[1], x[2], x[3], b])
>>> a + b
farray([a, b])
```

Even `0` (or `False`) and `1` (or `True`) work:

```
>>> 0 + X
farray([0, x[0], x[1], x[2], x[3]])
>>> X + True
farray([x[0], x[1], x[2], x[3], 1])
```

To repeat arrays, use the * operator:

```
>>> X * 2
farray([x[0], x[1], x[2], x[3], x[0], x[1], x[2], x[3]])
>>> 0 * X
farray([])
```

Similarly, this works for single functions as well:

```
>>> a * 3
farray([a, a, a])
>>> 2 * a + b * 3
farray([a, a, b, b, b])
```

Multi-dimensional arrays are automatically flattened during either concatenation or repetition:

```
>>> Z = exprvars('z', 2, 2)
>>> X + Z
farray([x[0], x[1], x[2], x[3], z[0,0], z[0,1], z[1,0], z[1,1]])
>>> Z * 2
farray([z[0,0], z[0,1], z[1,0], z[1,1], z[0,0], z[0,1], z[1,0], z[1,1]])
```

If you require a more subtle treatment of the shapes, use the `reshape` method to unflatten things:

```
>>> (Z*2).reshape(2, 4)
farray([[z[0,0], z[0,1], z[1,0], z[1,1]],
        [z[0,0], z[0,1], z[1,0], z[1,1]]])
```

Function arrays also support the "in-place" += and *= operators. The `farray` behaves like an immutable object. That is, it behaves more like the Python `tuple` than a `list`.

For example, when you concatenate/repeat an `farray`, it returns a new `farray`:

```
>>> A = exprvars('a', 4)
>>> B = exprvars('b', 4)
>>> id(A)
3050928972
>>> id(B)
3050939660
>>> A += B
>>> id(A)
3050939948
>>> B *= 2
>>> id(B)
3050940716
```

The `A += B` implementation is just syntactic sugar for:

```
>>> A = A + B
```

And the `A *= 2` implementation is just syntactic sugar for:

```
>>> A = A * 2
```

To wrap up, let's re-write the `xtime` function using PyEDA function arrays.

```
def xtime(b, n):
    """Return b^n using polynomial multiplication in GF(2^8)."""
    for _ in range(n):
        b = exprzeros(1) + b[:7] ^ uint2exprs(0x1b, 8) & b[7]*8
    return b
```

## 1.7 Two-level Logic Minimization

This chapter will explain how to use PyEDA to minimize two-level "sum-of-products" forms of Boolean functions.

Logic minimization is known to be an NP-complete problem. It is equivalent to finding a minimal-cost set of subsets of a set $S$ that *covers* $S$. This is sometimes called the "paving problem", because it is conceptually similar to finding the cheapest configuration of tiles that cover a floor. Due to the complexity of this operation, PyEDA uses a C extension to the famous Berkeley Espresso library [6].

All examples in this chapter assume you have interactive symbols imported:

```
>>> from pyeda.inter import *
```

### 1.7.1 Minimize Boolean Expressions

Consider the three-input function $f_1 = a \cdot b' \cdot c' + a' \cdot b' \cdot c + a \cdot b' \cdot c + a \cdot b \cdot c + a \cdot b \cdot c'$

```
>>> a, b, c = map(exprvar, 'abc')
>>> f1 = ~a & ~b & ~c | ~a & ~b & c | a & ~b & c | a & b & c | a & b & ~c
```

To use Espresso to perform minimization:

```
>>> f1m, = espresso_exprs(f1)
>>> f1m
Or(And(~a, ~b), And(a, b), And(~b, c))
```

Notice that the `espresso_exprs` function returns a tuple. The reason is that this function can minimize multiple input functions simultaneously. To demonstrate, let's create a second function $f_2 = a' \cdot b' \cdot c + a \cdot b' \cdot c$.

```
>>> f2 = ~a & ~b & c | a & ~b & c
>>> f1m, f2m = espresso_exprs(f1, f2)
>>> f1m
Or(And(~a, ~b), And(a, b), And(~b, c))
>>> f2m
And(~b, c)
```

It's easy to verify that the minimal functions are equivalent to the originals:

```
>>> f1.equivalent(f1m)
True
>>> f2.equivalent(f2m)
True
```

### 1.7.2 Minimize Truth Tables

An expression is a *completely* specified function. Sometimes, instead of minimizing an existing expression, you instead start with only a truth table that maps inputs in $0, 1$ to outputs in $0, 1, *$, where $*$ means "don't care". For this type of *incompletely* specified function, you may use the `espresso_tts` function to find a low-cost, equivalent Boolean expression.

Consider the following truth table with four inputs and two outputs:

[6] R. Brayton, G. Hatchel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, MA, 1984.

| Inputs | | | | Outputs | |
|---|---|---|---|---|---|
| x3 | x2 | x1 | x0 | f1 | f2 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | X | X |
| 1 | 0 | 1 | 1 | X | X |
| 1 | 1 | 0 | 0 | X | X |
| 1 | 1 | 0 | 1 | X | X |
| 1 | 1 | 1 | 0 | X | X |
| 1 | 1 | 1 | 1 | X | X |

The `espresso_tts` function takes a sequence of input truth table functions, and returns a sequence of DNF expression instances.

```
>>> X = ttvars('x', 4)
>>> f1 = truthtable(X, "0000011111------")
>>> f2 = truthtable(X, "0001111100------")
>>> f1m, f2m = espresso_tts(f1, f2)
>>> f1m
Or(x[3], And(x[0], x[2]), And(x[1], x[2]))
>>> f2m
Or(x[2], And(x[0], x[1]))
```

You can test whether the resulting expressions are equivalent to the original truth tables by visual inspection (or some smarter method):

```
>>> expr2truthtable(f1m)
inputs: x[3] x[2] x[1] x[0]
0000 0
0001 0
0010 0
0011 0
0100 0
0101 1
0110 1
0111 1
1000 1
1001 1
1010 1
1011 1
1100 1
1101 1
1110 1
1111 1
>>> expr2truthtable(f2m)
inputs: x[2] x[1] x[0]
000 0
001 0
010 0
011 1
```

```
100 1
101 1
110 1
111 1
```

### 1.7.3 Espresso Script

Starting with version `0.20`, PyEDA includes a script that implements some of the functionality of the original Espresso command-line utility.

```
$ espresso -h
usage: espresso [-h] [-e {fast,ness,nirr,nunwrap,onset,strong}] [--fast]
                [--no-ess] [--no-irr] [--no-unwrap] [--onset] [--strong]
                [file]

Minimize a PLA file

positional arguments:
  file                  PLA file (default: stdin)

optional arguments:
  -h, --help            show this help message and exit
  ...
```

Here is an example of a simple PLA file that is part of the BOOM benchmark suite. This function has 50 input variables, 5 output variables, and 50 product terms. Also, 20% of the literals in the implicants are "don't care".

```
$ cat extension/espresso/test/bb_all/bb_50x5x50_20%_0.pla
.i 50
.o 5
.p 50
.ilb x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24 x25 x2
.ob y0 y1 y2 y3 y4
.type fr
001011110--00--0100-0010-10-01010101-010--01011111 00011
0-1-1010-01000100--11011-0110001010-10001010-1-1-0 00100
011101011111010110-1001011010100100011111----1-011 10000
01-0010011-001110--000-011--11--1-0100-01101--1000 00001
001011010-1100000001101--10100-010-0011001111110010 00101
011-01010-10-1101110-00-1-11001-1-0000--1-1-00-000 00011
0000000011001-0000010-000110-11011001110--100-1-10 00110
00-111111-00100-100111101000-11101100--0-1110-1-10 10001
-1-10011011000011--00-0011011101-1-1101110--1-001- 11100
-0101100-110111-010-01110-0110011-1-1---1001011111 11000
0-111011101000-11-1--10-0001001101000010-11-111101 11001
11000000-1-01--1111-10111111----0010--1-0--1--0111 01010
00-101000011-1-10101101-1101011-0101000-111111011- 11011
1-00-11111-0010-0---000--0110-0010111--000001-0001 11011
-1-1100100001--00--00001000-1-1--100-0111-00011011 11000
0-0000-010-11-1100-101-00101000111-01--11-0010-011 10000
11-1-0001100101-10-0-1-0-1100010101111-1111000-101 11101
10-01--10011111-11011-0010011011001100100010-000-0- 01110
1-11010-00011101-010--101010--0111010101-11-101--1 00111
11--111-111-111-000-11000-101-1-011--1000--1111100 01111
0---0-10011101000--11001-1100-10-000011-0100011100 11110
-01--11-010-1001011-0-101000100000--10111---100-1- 11101
11-1-000010--00110-011101--11-10-1-0000110100-1101 11010
```

```
-0111110-100-11-110001001100001-100011110001001100 11110
11--00100-01--00-10---11-0001-00011101001011-01110 00000
1--010011-001-0000--0-11-001010001110-00-01-110-11 01101
100011--0101--1-1-0-101--001-0-101-1-1011101011-01 00111
0--0-01-10101-11-0100111111000-1-1011100-111-01111 10100
0-0110010--11101-0---1001-1001--001-110000---1011- 00100
0-1000-0--00000010-0--1011-1001011-01-00-011001111 10000
111-1101111-01001101-111--00-01011111000001-001001 11100
0--100111-1010001-0111-0-000--00-0111101111-101100 11000
00001101100-001001-1010010010011-1101-110-10-110-1 01011
0101-01-0100101000010111--0011-0011011110-111100-0 00100
000-1--100-00-1001-10-000000100-001100-10101010001 10000
10001001-0001011-1-1-0-00101110-10100---0010001--- 10111
0101100000100100000---1--11-0001011111101-01-1011 01111
1--01--00100110001-110-0-00001011---01001000110--- 10010
0-0001--01--11101010100000000010011001000-01100001 00011
0-0100110-00111100-001--11--00-1001-00-0-11-1-0-1- 00100
101-1-100-001001-010111-01--010-1-1011-01101001001 11000
0110-111011--1-010101-011-1-00100110-00-1111000-11 11001
011001011---010011-10-00-11-0010000001011101101-0-1 00100
1001111-1-1111-1001-000111010-100--0111110011000-1 10111
1-1010-1-100111110010-101011-1001000111-0000--11-1 11000
-0011000100001000001010001001O-0-0-100-1-0111011-1 00101
1110-01100111111-1-1-110-0-110--011--01-11-0000-01 00000
-01010101010-1-1-00-1111010100-1001111110110--0-00 11011
110-10000001--0-0-01001111-0011-0110110100010--111 11111
101-10111000011110000-1001-001-01111-011-0001-0100 00100
.e
```

After running the input file through the `espresso` script, it minimizes the function to 26 implicants with significantly fewer literals.

```
$ espresso extension/espresso/test/bb_all/bb_50x5x50_20%_0.pla
.i 50
.o 5
.ilb x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24 x25 x2
.ob y0 y1 y2 y3 y4
.p 26
----------------------0-----1---0------0---------- 01110
------0-----------------------------1-----1--0--0- 11100
--------------------------1--------1---0------0-- 01011
-------------------------------------11-1--------1 10000
0------------------------1---------1-1----------- 00110
--------0--------1----------0-------0------------- 00010
-------------0---------0-0-0--------------------1- 10001
--------------0-00-----------------0-------1----- 00101
-------------------------0-0---0---------0------- 00011
-----------0----0-------------------------0---0--- 10000
-------------------------1---0---1---------------1 10000
-----------01-----------------------------0---1--1 11000
-------------00------------------------11--------- 11000
----------------------------0-1---------0-0---1--- 11110
-------------------------1----1--0------------1-- 00100
-----1-----------------------------1-1---1-------- 00111
------1---------------0-------1--1---0------------ 11001
------0----0--------------------------------1-1-0--- 01000
-1---1--------1-----------------------------------0 00001
-----------------1----------------------0-----0-1- 01010
```

```
--------------------0--------1---------1------0--- 00100
--------------------------11-------------1-0-1--- 10010
-------------------------1-----0----10----------- 00100
------0------0---------0---------0-----------0---- 00101
-------0----------0---1--1--0---0----------------- 11011
--------------------0-----------0-----------1-0--- 00100
.e
```

### 1.7.4 References

## 1.8 Using PyEDA to Solve Sudoku

According to Peter Norvig in his fantastic essay on solving every Sudoku puzzle using Python, security expert Ben Laurie once stated that "Sudoku is a denial of service attack on human intellect". I can personally attest to that.

In this example, I will explain how to use PyEDA's Boolean expressions and satisfiability engine to create a general-purpose Sudoku solver.

First, let's get a few ground rules straight:

1. There are lots of Sudoku solvers on the Internet; I make no claims of novelty.
2. PyEDA is pretty fast, but unlikely to win any speed competitions.
3. Let's face it–this is a pretty awesome waste of time.

### 1.8.1 Getting Started

First, import all the standard symbols from PyEDA.

```
>>> from pyeda.inter import *
```

Let's also define a variable "DIGITS" that makes it easier to access the Sudoku square values.

```
>>> DIGITS = "123456789"
```

### 1.8.2 Setting Up the Puzzle Grid

A Sudoko puzzle is a 9x9 **grid**. Each square in the grid may contain any number in the digits 1-9. The following example grid was generated by Web Sudoku.

To represent the state of the grid coordinate, create a 3-dimensional variable, `X`.

```
>>> X = exprvars('x', (1, 10), (1, 10), (1, 10))
```

`exprvars` is a versatile function for returning arrays of arbitrary dimension. The first argument is the variable name. Each additional argument adds an additional dimension with a non-negative start and stop index. If you provide only a number instead of a two-tuple, the start index is zero. For example, we could also have used `exprvars('x', 9, 9, 9)`, but that would have given `X[0:9, 0:9, 0:9]` instead of `X[1:10, 1:10, 1:10]`.

The variable `X` is a 9x9x9 bit vector, indexed as `X[row, column, value]`. So for the *Example Sudoku Grid*, since row 5, column 3 has value '8', we would represent this by setting `X[5,3,8] = 1`.

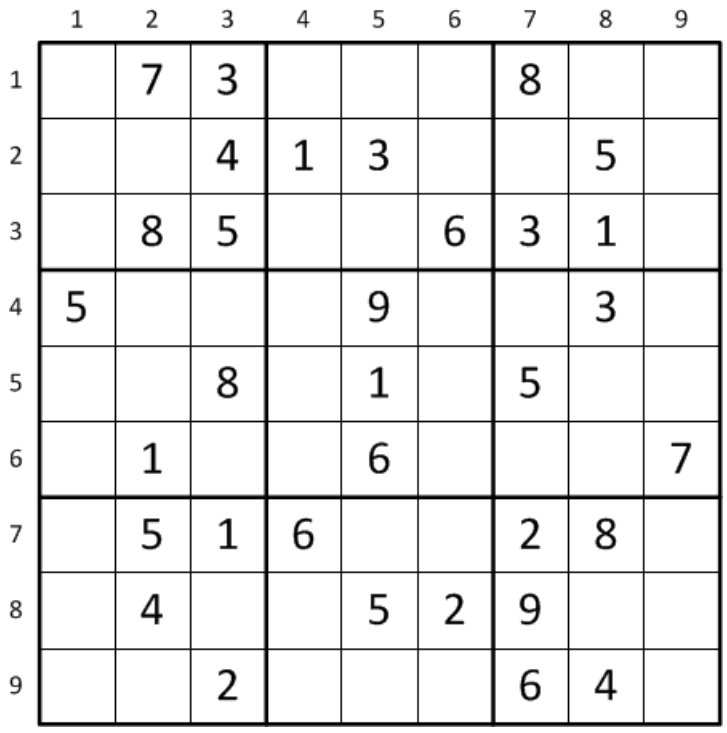



Figure 1.6: Example Sudoku Grid

### 1.8.3 Constraints

Now that we have a variable `X[r,c,v]` to represent the state of the Sudoku board, we need to program the constraints. We will use the familiar Boolean `And` function, and the `OneHot` function. In case you are not familiar with the `OneHot` function, we will describe it here.

#### One Hot Function

Let's say I have three variables, **a**, **b**, and **c**.

```
>>> a, b, c = map(exprvar, 'abc')
```

I want to write a Boolean formula that guarantees that only one of them is true at any given moment.

```
>>> f = OneHot(a, b, c)
```

You can use PyEDA to automatically produce the truth table.

```
>>> expr2truthtable(f)
inputs: c b a
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 0
```

By default, the `OneHot` function returns a formula in conjunctive normal (product-of-sums) form. Roughly translated, this formula says that "no two variables can both be true, and at least one must be true".

```
>>> f
And(Or(~a, ~b), Or(~a, ~c), Or(~b, ~c), Or(a, b, c))
```

In disjunctive normal (sum-of-products) form, the function looks like this:

```
>>> f.to_dnf()
Or(And(~a, ~b, c), And(~a, b, ~c), And(a, ~b, ~c))
```

### Value Constraints

You probably already noticed that if the square at (5, 3) has value '8', it is not allowed to have any other value. That is, if `X[5,3,8] = 1`, then `X[5,3,1:10] == [0, 0, 0, 0, 0, 0, 0, 1, 0]`.

We need to write a constraint formula that says "every square on the board can assume only one value." With PyEDA, you can write this formula as follows:

```
>>> V = And(*[
...           And(*[
...               OneHot(*[ X[r, c, v]
...                   for v in range(1, 10) ])
...               for c in range(1, 10) ])
...           for r in range(1, 10) ])
```

### Row and Column Constraints

Next, we need to write formulas that say "every square in each row is unique", and "every square in each column is unique", respectively.

```
>>> R = And(*[
...           And(*[
...               OneHot(*[ X[r, c, v]
...                   for c in range(1, 10) ])
...               for v in range(1, 10) ])
...           for r in range(1, 10) ])

>>> C = And(*[
...           And(*[
...               OneHot(*[ X[r, c, v]
...                   for r in range(1, 10) ])
...               for v in range(1, 10) ])
...           for c in range(1, 10) ])
```

### Box Constraints

The box constraints are a little tricker. We need a formula that says "every square in a box is unique". The key to understanding how to write this formula is to think of the grid as consisting of 3x3 boxes. Now instead of iterating over the nine squares in a row or column, we will iterate over the 3 rows and 3 columns of the 3x3 boxes.

```
>>> B = And(*[
...           And(*[
...               OneHot(*[ X[3*br+r, 3*bc+c, v]
...                   for r in range(1, 4) for c in range(1, 4) ])
```

```
...             for v in range(1, 10) ])
...         for br in range(3) for bc in range(3) ])
```

#### Putting It All Together

Now that we have the value, row, column, and box constraints, we need to combine them all into a single formula. We will use the `And` function to join the constraints, because all constraints must be true for the puzzle to be solved.

```
>>> S = And(V, R, C, B)
>>> len(S.args)
10530
```

As you can see, the constraints formula is *quite* large.

### 1.8.4 Preparing the Input

We now have the generic constraints for the rules of Sudoku, but when you sit down to solve a puzzle, you are always given a set of known values. These are the *inputs*, and they will further constrain the solution.

Here is a function to parse an input string, and produce the input constraints. Any character in the set 1-9 will be taken as an assignment, the values '0' and '.' (period) will be taken as an unknown, and all other characters will be ignored. This function also returns a CNF data type.

```
>>> def parse_grid(grid):
...     chars = [c for c in grid if c in DIGITS or c in "0."]
...     assert len(chars) == 9 ** 2
...     return And(*[ X[i // 9 + 1, i % 9 + 1, int(c)]
...                   for i, c in enumerate(chars) if c in DIGITS ])
```

The example grid above can be written like this:

```
>>> grid = ( ".73|...|8.."
...          "..4|13.|.5."
...          ".85|..6|31."
...          "---+---+---"
...          "5..|.9.|.3."
...          "..8|.1.|5.."
...          ".1.|.6.|..7"
...          "---+---+---"
...          ".51|6..|28."
...          ".4.|.52|9.."
...          "..2|...|64." )
```

### 1.8.5 Display Methods

To display the solution, we will need some methods. The PyEDA SAT solver returns a dictionary that represents a "point" in an N-dimensional Boolean space. That is, it maps N Boolean variables (in our case 729) onto their values in $\{0, 1\}$.

```
>>> def get_val(point, r, c):
...     for v in range(1, 10):
...         if point[X[r, c, v]]:
...             return DIGITS[v-1]
...     return "X"
```

```
>>> def display(point):
...     chars = list()
...     for r in range(1, 10):
...         for c in range(1, 10):
...             if c in (4, 7):
...                 chars.append("|")
...             chars.append(get_val(point, r, c))
...         if r != 9:
...             chars.append("\n")
...             if r in (3, 6):
...                 chars.append("---+---+---\n")
...     print("".join(chars))
```

### 1.8.6 Finding the Solution

Without further ado, let's use the PicoSAT fast SAT solver to crunch the numbers.

```
>>> def solve(grid):
...     with parse_grid(grid):
...         return S.satisfy_one()
```

Here is the solution to the *Example Sudoku Grid*:

```
>>> display(solve(grid))
173|529|864
694|138|752
285|476|319
---+---+---
567|294|138
428|713|596
319|865|427
---+---+---
951|647|283
846|352|971
732|981|645
```

That example was actually a pretty easy puzzle. Let's see how the Sudoku solver handles a few harder puzzles.

```
>>> grid = ( "6..|3.2|..."
...          ".5.|...|.1."
...          "...|...|..."
...          "---+---+---"
...          "7.2|6..|..."
...          "...|...|.54"
...          "3..|...|..."
...          "---+---+---"
...          ".8.|15.|..."
...          "...|.4.|2.."
...          "...|...|7.." )

>>> display(solve(grid))
614|382|579
953|764|812
827|591|436
---+---+---
742|635|198
168|279|354
```

```
395|418|627
---+---+---
286|157|943
579|843|261
431|926|785

>>> grid = ( "38.|6..|..."
...          "..9|...|..."
...          ".2.|.3.|51."
...          "---+---+---"
...          "...|..5|..."
...          ".3.|.1.|.6."
...          "...|4..|..."
...          "---+---+---"
...          ".17|.5.|.8."
...          "...|...|9.."
...          "...|..7|.32" )

>>> display(solve(grid))
385|621|497
179|584|326
426|739|518
---+---+---
762|395|841
534|812|769
891|476|253
---+---+---
917|253|684
243|168|975
658|947|132
```

## 1.9 All Solutions To The Eight Queens Puzzle

The **eight queens** puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens attack each other. It is a classic demonstration of finding the solutions to a constraint problem.

In this essay we will use the PyEDA SAT solver to find all solutions to the eight queens puzzle.

### 1.9.1 Getting Started

First, import all the standard symbols from PyEDA.

```
>>> from pyeda.inter import *
```

### 1.9.2 Setting Up the Chess Board

A chess board is an 8x8 **grid**. Each square on the grid either has a queen on it, or doesn't. Therefore, we can represent the board using a two-dimensional bit vector, `X`.

```
>>> X = exprvars('x', 8, 8)
```

### 1.9.3 Constraints

#### Row and Column Constraints

Rather than start with the constraint that $\sum X_{i,j} = 8$, we will instead start with a simplifying observation. In order to place eight queens on the board, since there are exactly eight rows and eight columns on the board itself, it is obvious that exactly one queen must be placed on each row, and each column.

First, we write a constraint that says "exactly one queen must be placed on each row".

```
>>> R = And(*[OneHot(*[X[r,c] for c in range(8)]) for r in range(8)])
```

Next, we write a constraint that says "exactly one queen must be placed on each column".

```
>>> C = And(*[OneHot(*[X[r,c] for r in range(8)]) for c in range(8)])
```

#### Diagonal Constraints

Diagonal constraints are easy to visualize, but slightly trickier to specify mathematically. We will break down the diagonal constraints into two separate sets:

- left-to-right
- right-to-left

In both cases, the diagonal is always oriented "bottom-to-top".

In both cases, we need to write a constraint that says "at most one queen can be located on each diagonal".

```
>>> starts = [(i, 0) for i in range(8-2, 0, -1)] + [(0, i) for i in range(8-1)]
>>> lrdiags = []
>>> for r, c in starts:
...     lrdiags.append([])
...     ri, ci = r, c
...     while ri < 8 and ci < 8:
...         lrdiags[-1].append((ri, ci))
...         ri += 1
...         ci += 1
...
>>> DLR = And(*[OneHot0(*[X[r,c] for r, c in diag]) for diag in lrdiags])

>>> starts = [(i, 8-1) for i in range(8-2, -1, -1)] + [(0, i) for i in range(8-2, 0, -1)]
>>> rldiags = []
>>> for r, c in starts:
...     rldiags.append([])
...     ri, ci = r, c
...     while ri < 8 and ci >= 0:
...         rldiags[-1].append((ri, ci))
...         ri += 1
...         ci -= 1
...
>>> DRL = And(*[OneHot0(*[X[r,c] for r, c in diag]) for diag in rldiags])
```

#### Putting It All Together

Now that we have constraints for rows, columns, and diagonals, we have successfully defined all rules for solving the puzzle. Put them all together using the And function, because all constraints must simultaneously be valid.

```
>>> S = R & C & DLR & DRL
```

Verify the formula is in CNF form, and show how large it is:

```
>>> S.is_cnf()
True
>>> len(S.args)
744
```

### 1.9.4 Display Method

For convenience, let's define a function display to conveniently convert a solution point to ASCII:

```
>>> def display(point):
...     chars = list()
...     for r in range(8):
...         for c in range(8):
...             if point[X[r,c]]:
...                 chars.append("Q")
...             else:
...                 chars.append(".")
...         if r != 7:
...             chars.append("\n")
...     print("".join(chars))
```

### 1.9.5 Find a Single Solution

Find a single solution to the puzzle using the *satisfy_one* method:

```
>>> display(S.satisfy_one())

.......Q
...Q....
Q.......
..Q.....
.....Q..
.Q......
......Q.
....Q...
```

### 1.9.6 Find All Solutions

Part of the challenge of the eight queens puzzle is to not just find a single solution, but find all solutions. Use the *satisfy_all* method to iterate through all solutions:

```
>>> for i, soln in enumerate(S.satisfy_all()):
...     print("Solution", i+1, end="\n\n")
...     display(soln)
...     print("")

Solution 1

.......Q
...Q....
Q.......
```

```
..Q.....
.....Q..
.Q......
......Q.
....Q...

...
```

It is easy to verify that there are exactly 92 distinct solutions to the puzzle:

```
>>> S.satisfy_count()
92
>>> len(list(S.satisfy_all()))
92
```

## 1.10 Release Notes

### 1.10.1 Version 0.25

This is a small, incremental release. I recently changed jobs and moved, so development will definitely slow down for a while.

Function array concatenation and repetition for MDAs is now a bit smarter (Issue 96). Rather than simply flattening, the operators will attempt to retain the shape of the MDAs if possible. For example, a `2x6x7 + 2x6x7` concatenation will return `4x6x7`, and `2x6x7 * 2` repetition will return `4x6x7`.

Got rid of `a[0][1][2]` expression parsing syntax. Use `a[0,1,2]` instead. Also got rid of the `bitvec` function. Use the `exprvars` function (or `bddvars`, `ttvars`) instead. Finally all vestiges of the legacy `BitVector` MDA methodology is gone.

Everything else was just miscellaneous code/test/documentation cleanup.

### 1.10.2 Version 0.24

Variables names are now required to be C-style identifiers. I.e., `[a-zA-Z_][a-zA-Z0-9_]*`.

The expression parser now handles both `a[1][2][3]` and `a[1,2,3]` syntaxes (Issue 91). The `a[1][2][3]` is deprecated.

Got rid of expression `is_neg_unate`, `is_pos_unate`, and `is_binate` functions. I haven't been able to find an *efficient* algorithm for this, so just convert expressions and BDDs to truth tables first. If your function is too big to fit in a truth table, it's probably also too big to expand to a canonical expression.

`Not(Not(...))` double negation is now automatically reduced, just like `Not(Nand(...))`, etc.

Cleaned up the definition of expression depth (Issue 92). This is not backwards compatible.

Fixed Issue 93, picosat script fails with trivial zero input:

```
$ picosat
p cnf 0 1
0
```

Changed `RegexLexer` to yield `EndToken` at the end of a token stream. This makes parsing nicer, avoiding catching `StopIteration` everywhere.

Got rid of `factor=False` on expression factory functions. This was overly designed UI.

The expression `restrict` method is a little faster now. Especially for big functions.

Added *lots* of new reference documentation.

Added new `farray` documentation chapter. Fixed several little issues with function arrays during this process. The constructor now takes an `ftype=None` parameter. Negative indices make more sense now. Slices behave more like Python tuple slices. Fixed several inconsistencies with empty arrays.

Deprecated `bitvec` function.

### 1.10.3 Version 0.23

This version introduces a new `picosat` script. Now you can solve DIMACS CNF files from the command-line. See http://pyeda.readthedocs.org/en/latest/expr.html#picosat-script for details.

Finally there is a proper documentation chapter for binary decision diagrams! While writing this documentation, I noticed, and fixed some obscure bugs related to incorrect usage of weak references to BDD nodes.

Made some minor changes to the public interface of the `bdd` module.

Replaced the `traverse` method with three options for BDD iteration:

- `bdd.dfs_preorder` - Depth-first search pre-order traversal
- `bdd.dfs_postorder` - Depth-first search post-order traversal
- `bdd.bfs()` - Breadth-first search

Got rid of the deprecated `uint2bv` and `int2bv` functions. Use the `uint2exprs`, `int2exprs` functions instead.

Changed the `pyeda.parsing.parse_pla` function so it takes a string input. This makes it much easier to test.

Deprecated the `is_neg_unate`, `is_pos_unate`, `is_binate` methods for expressions. I haven't found a correct algorithm that is better than just 1) converting to a truth table, and 2) checking for monotonicity in the cofactors.

As of this release, I will be dropping support for Python 3.2.

### 1.10.4 Version 0.22

A couple features, and some good bug-fixes in this release.

Fixed Issue 80. Apparently, I forgot to implement the right-side version of XOR operator: `0 ^ x`.

Fixed Issue 81. I continue finding bugs with degenerate forms. This particular one comes up when you try to do something similar to `Or(Or(a, b))`. The `__new__` method was implemented incorrectly, so I moved the `Or(a) = a` (and similar) rules to the `simplify` method.

To match the notation used by Univ of Illinois VLSI class, I changed BDD low/high nodes to "lo", and "hi".

Got rid of the "minus" operator, `a - b`. This was previously implemented as `a | ~b`, but I don't think it has merit anymore.

The `farray` type now uses the `+` operator for concatenation, and `*` for repetition. These are very important features in SystemVerilog. See Issue 77 for details.

Implemented the `farray.__setitem__` method. It is very useful to instantiate an `farray` using `exprzeros`, and then programmatically assign indices one-by-one. See Issue 78 for details.

To demonstrate some of the fancy, new `farray` features, I added the AES algorithm to the `logic` package. It manages to complete all the logic assignments, but I haven't been able to test its correctness yet, because it explodes the memory on my machine. At a bare minimum, it will be a nice test case for performance optimizations necessary to handle large designs.

### 1.10.5 Version 0.21

Important bug fix! Issue 75. Harnesser pointed out that Espresso was returning some goofy results for degenerate inputs (a literal or `AND(lit, lit, ...)`).

The major new feature in this release is the `farray` mult-dimensional array (MDA) data type. The implementation of `BitVector` was a kludge – it was built around the `Expression` function type, and didn't support all the fancy things you could do with numpy slices. All usage of the old `Slicer` and `BitVector` types has been eliminated, and replaced by `farray`. This includes the `bitvec`, `uint2bv`, and `int2bv` functions, and the contents of the `pyeda.logic` package (addition, Sudoku, etc).

Both `uint2bv` and `int2bv` are deprecated, superceded by `uint2exprs` and `int2exprs` (or `uint2bdds`, etc). So far I haven't deprecated `bitvec`, because it's a very commonly-used function.

See Issue 68 for some details on the `farray` type. My favorite part is the ability to multiplex an `farray` using Python's slice syntax:

```
>>> xs = exprvars('x', 4)
>>> sel = exprvars('s', 2)
>>> xs[sel]
Or(And(~s[0], ~s[1], x[0]), And(s[0], ~s[1], x[1]), And(~s[0], s[1], x[2]), And(s[0], s[1], x[3]))
```

This even works with MDAs:

```
>>> xs = exprvars('x', 4, 4)
>>> sel = exprvars('s', 2)
>>> xs[0,sel]
Or(And(~s[0], ~s[1], x[0][0]), And(s[0], ~s[1], x[0][1]), And(~s[0], s[1], x[0][2]), And(s[0], s[1],
```

Added `AchillesHeel` function to expression parsing engine.

Eliminated the + and * operators for Boolean OR, AND, respectively. This is annoying, but I need these operators for Issue 77. Sorry for any trouble, but that's what major version zero is for :).

### 1.10.6 Version 0.20

Probably the most useful feature in this release is the `espresso` script:

```
$ espresso -h
usage: espresso [-h] [-e {fast,ness,nirr,nunwrap,onset,strong}] [--fast]
                [--no-ess] [--no-irr] [--no-unwrap] [--onset] [--strong]
                [file]

Minimize a PLA file

positional arguments:
  file                  PLA file (default: stdin)

optional arguments:
  ...
```

This script implements a subset of the functionality of the original `Espresso` command-line program. It uses the new `parse_pla` function in the `pyeda.parsing.pla` module to parse common PLA files. Note that the script only intends to implement basic truth-table functionality at the moment. It doesn't support multiple-valued variables, and various other Espresso built-in features.

Added Espresso `get_config` and `set_config` functions, to manipulate global configuration

New `Bitvector` methods:

- `unor` - unary nor
- `unand` - unary nand
- `uxnor` - unary xnor

Made `BitVector` an immutable type. As a result, dropped item assignment `X[0] = a`, zero extension `X.zext(4)`, sign extension `X.sext(4)`, and `append` method.

The `BitVector` type now supports more overloaded operators:

- `X + Y` concatenate two bit vectors
- `X << n` return the bit vector left-shifted by `n` places
- `X >> n` return the bit vector right-shifted by `n` places

Both left shift and right shift are simple shifts–they use the default "carry-in" of zero.

Got rid of `boolify` utility function. It had been replaced over time by more sophisticated techniques.

There is a new `Mux` factory function, for multiplexing arbitrarily many input functions.

Update to PicoSAT 959. Check the homepage for details, but it looks like the only changes were related to header file documentation.

Added a neat capability to specify assumptions for SAT-solving using a `with` statement. It supports both literal and product-term forms:

```
>>> f = Xor(a, b, c)
>>> with a, ~b:
...     print(f.satisfy_one())
{a: 1, b: 0, c: 0}
>>> with a & ~b:
...     print(f.satisfy_one())
{a: 1, b: 0, c: 0}
```

At the moment, this only works for the `satisfy_one` method, because it is so handy and intuitive.

### 1.10.7 Version 0.19

#### Release 0.19.3

Enhanced error handling in the Espresso C extension.

#### Release 0.19.2

Added the `espresso_tts` function, which allows you to run Espresso on one or more `TruthTable` instances.

#### Release 0.19.1

Fixed a bone-headed mistake: leaving `espresso.h` out of the source distribution. One of these days I will remember to test the source distribution for all the necessary files before releasing it.

#### Release 0.19.0

This is a very exciting release! After much hard work, PyEDA now has a C extension to the famous Espresso logic minimization software from Berkeley! See the new chapter on two-level logic minimization for usage information.

Also, after some feedback from users, it became increasingly obvious that using the `-+*` operators for NOT, OR, AND was a limitation. Now, just like Sympy, PyEDA uses the `~|&^` operators for symbolic algebra. For convenience, the legacy operators will issue deprecation warnings for now. In some upcoming release, they will no longer work.

After other feedback from users, I changed the way `Expression` string representation works. Now, the `__str__` method uses `Or`, `And`, etc, instead of ascii characters. The idea is that the string representation now returns valid Python that can be parsed by the `expr` function (or the Python interpreter). To provide support for fancy formatting in IPython notebook, I added the new `to_unicode` and `to_latex` methods. These methods also return fancy string representations.

For consistency, the `uint2vec` and `int2vec` functions have been renamed to `uint2bv` and `int2bv`, respectively.

Since `is_pos_unate`, `is_neg_unate`, and `is_binate` didn't seem like fundamental operations, I remove them from the `Function` base class.

### 1.10.8 Version 0.18

#### Release 0.18.1

Three minor tweaks in this release:

- `expr`/`bdd` `to_dot` methods now return undirected graphs.
- Added `AchillesHeel` factory function to `expr`.
- Fixed a few obscure bugs with simplification of `Implies` and `ITE`.

#### Release 0.18.0

New stuff in this release:

- Unified the `Expression` and `Normalform` expression types, getting rid of the need for the `nfexpr` module.
- Added `to_dot` methods to both `Expression` and `BinaryDecisionDiagram` data types.

Mostly incremental changes this time around. My apologies to anybody who was using the `nfexpr` module. Lately, `Expression` has gotten quite fast, especially with the addition of the PicoSAT C extension. The normal form data type as `set(frozenset(int))` was not a proper implementation of the `Function` class, so finally did away with it in favor of the new "encoded" representation that matches the Dimacs CNF convention of mapping an index 1..N to each variable, and having the negative index correspond to the complement. So far this is only useful for CNF SAT-solving, but may also come in handy for any future, fast operations on 2-level covers.

Also, somewhat awesome is the addition of the `to_dot` methods. I was playing around with IPython extensions, and eventually hacked up a neat solution for drawing BDDs into the notebook. The magic functions are published in my ipython-magic repo. See the usage notes. Using `subprocess` is probably not the best way to interface with Graphviz, but it works well enough without any dependencies.

### 1.10.9 Version 0.17

#### Release 0.17.1

Got rid of the `assumptions` parameter from `boolalg.picosat.satisfy_all` function, because it had no effect. Read through `picosat.h` to figure out what happened, and you need to re-apply assumptions for every call to `picosat_sat`. For now, the usage model seems a little dubious, so just got rid of it.

#### Release 0.17.0

New stuff in this release:

- Added `assumptions=None` parameter to PicoSAT `satisfy_one` and `satisfy_all` functions. This produces a *very* nice speedup in some situations.
- Got rid of extraneous `picosat.py` Python wrapper module. Now the PicoSAT Python interface is implemented by `picosatmodule.c`.
- Updated Nor/Nand operators to secondary status. That is, they now can be natively represented by symbolic expressions.
- Added a Brent-Kung adder to logic.addition module
- Lots of other miscellaneous cleanup and better error handling

### 1.10.10 Version 0.16

#### Release 0.16.3

Fixed bug: absorption algorithm not returning a fully simplified expression.

#### Release 0.16.2

Significantly enhance the performance of the absorption algorithm

#### Release 0.16.1

Fixed bug: PicoSAT module compilation busted on Windows

#### Release 0.16.0

New stuff in this release:

- Added Expression `complete_sum` method, to generate a normal form expression that contains all prime implicants.
- Unicode expression symbols, because it's awesome
- Added new Expression ForEach, Exists factory functions.
- Changed `frozenset` implementation of `OrAnd` and `EqualBase` arguments back to `tuple`. The simplification aspects had an unfortunate performance penalty. Use `absorb` to get rid of duplicate terms in DNF/CNF forms.

- Added flatten=False/True to Expression to_dnf, to_cdnf, to_cnf, to_ccnf methods. Often, flatten=False is faster at reducing to a normal form.
- Simplified absorb algorithm using Python sets.
- Expression added a new splitvar property, which implements a common heuristic to find a good splitting variable.

### 1.10.11 Version 0.15

#### Release 0.15.1

- Thanks to Christoph Gohlke, added build support for Windows platforms.

#### Release 0.15.0

This is probably the most exciting release of PyEDA yet! Integration of the popular PicoSAT fast C SAT solver makes PyEDA suitable for industrial-strength applications. Unfortunately, I have no idea how to make this work on Windows yet.

Here are the full release notes:

- Drop support for Python 2.7. Will only support Python 3.2+ going forward.
- Integrate PicoSAT, a compact SAT solver written in C.
- Added *lots* of new capabilities to Boolean expression parsing:
  - `s ? d1 : d0` (ITE), `p => q` (Implies), and `p <=> q` (Equal) symbolic operators.
  - Full complement of explicit form Boolean operators: `Or`, `And`, `Xor`, `Xnor`, `Equal`, `Unequal`, `Nor`, `Nand`, `OneHot0`, `OneHot`, `Majority`, `ITE`, `Implies`, `Not`
  - The `expr` function now simplifies by default, and has `simplify=True`, and `factor=False` parameters.
- New `Unequal` expression operator.
- New `Majority` high-order expression operator.
- `OneHot0`, `OneHot`, and `Majority` all have both disjunctive (`conj=False`) and conjunctive (`conj=True`) forms.
- Add new `Expression.to_ast` method. This might replace the `expr2dimacssat` function in the future,
- Fixed bug: `Xor.factor(conj=True)` returns non-equivalent expression.
- Changed the meaning of `conj` parameter in `Expression.factor` method. Now it is only used by the top-level, and not passed recursively.
- Normal form expression no longer inherit from `Function`. They didn't implement the full interface, so this just made sense.
- Replaced `pyeda.expr.expr2dimacscnf` with a new `pyeda.expr.DimacsCNF` class. This might be unified with normal form expressions in the future.

### 1.10.12 Version 0.14

#### Release 0.14.2

Fixed Issue #42.

There was a bug in the implementation of `OrAnd`, due to the new usage of a *frozenset* to represent the argument container.

With `0.14.1`, you could get this:

```
>>> And('a', 'b', 'c') == Or('a', 'b', 'c')
True
```

Now:

```
>>> And('a', 'b', 'c') == Or('a', 'b', 'c')
False
```

The == operator is only used by PyEDA for hashing, and is not overloaded by `Expression`. Therefore, this could potentially cause some serious issues with `Or`/`And` expressions that prune arguments incorrectly.

#### Release 0.14.1

Fixed Issue #41. Basically, the package metadata in the `0.14.0` release was incomplete, so the source distribution only contained a few modules. Whoops.

#### Release 0.14.0

This release reorganizes the PyEDA source code around quite a bit, and introduces some awesome new parsing utilities.

Probably the most important new feature is the addition of the `pyeda.boolalg.expr.expr` function. This function takes `int` or `str` as an input. If the input is a `str` instance, the function *parses the input string*, and returns an `Expression` instance. This makes it easy to form symbolic expression without even having to declare variables ahead of time:

```
>>> from pyeda.boolalg.expr import expr
>>> f = expr("-a * b + -b * c")
>>> g = expr("(-x[0] + x[1]) * (-x[1] + x[2])")
```

The return value of `expr` function is **not** simplified by default. This allows you to represent arbitrary expressions, for example:

```
>>> h = expr("a * 0")
>>> h
0 * a
>>> h.simplify()
0
```

- Reorganized source code:
  - Moved all Boolean algebra (functions, vector functions) into a new package, `pyeda.boolalg`.
  - Split `arithmetic` into `addition` and `gray_code` modules.
  - Moved all logic functions (addition, gray code) into a new package, `pyeda.logic`.
  - Created new Sudoku module under `pyeda.logic`.
- Awesome new regex-based lexical analysis class, `pyeda.parsing.RegexLexer`.

- Reorganized the DIMACS parsing code:
  - Refactored parsing code to use `RegexLexer`.
  - Parsing functions now return an abstract syntax tree, to be used by `pyeda.boolalg.ast2expr` function.
  - Changed `dimacs.load_cnf` to `pyeda.parsing.dimacs.parse_cnf`.
  - Changed `dimacs.load_sat` to `pyeda.parsing.dimacs.parse_sat`.
  - Changed `dimacs.dump_cnf` to `pyeda.boolalg.expr2dimacscnf`.
  - Changed `dimacs.dump_sat` to `pyeda.boolalg.expr2dimacssat`.
- Changed constructors for `Variable` factories. Unified `namespace` as just a part of the `name`.
- Changed interactive usage. Originally was `from pyeda import *`. Now use `from pyeda.inter import *`.
- Some more miscellaneous refactoring on logic expressions:
  - Fixed weirdness with `Expression.simplified` implementation.
  - Added new private class `_ArgumentContainer`, which is now the parent of `ExprOrAnd`, `ExprExclusive`, `ExprEqual`, `ExprImplies`, `ExprITE`.
  - Changed `ExprOrAnd` argument container to a `frozenset`, which has several nice properties for simplification of AND/OR expressions.
- Got rid of `pyeda.alphas` module.
- Preliminary support for logic expression `complete_sum` method, for generating the set of prime implicants.
- Use a "computed table" cache in BDD `restrict` method.
- Use weak references to help with BDD garbage collection.
- Replace distutils with setuptools.
- Preliminary support for Tseitin encoding of logic expressions.
- Rename `pyeda.common` to `pyeda.util`.

### 1.10.13 Version 0.13

Wow, this release took a huge leap from version 0.12. We're probably not ready to declare a "1.0", but it is definitely time to take a step back from API development, and start focusing on producing useful documentation.

This is not a complete list of changes, but here are the highlights.

- Binary Decision Diagrams! The recursive algorithms used to implement this datatype are awesome.
- Unification of all Variable subclasses by using separate factory functions (`exprvar`, `ttvar`, `bddvar`), but a common integer "uniqid".
- New "untyped point" is an immutable 2-tuple of variable uniqids assigned to zero and one. Also a new `urestrict` method to go along with it. Most important algorithms now use untyped points internally, because the set operations are very elegant and avoid dealing with which type of variable you are using.
- Changed the Variable's `namespace` argument to a tuple of strings.
- Restricting a function to a 0/1 state no longer returns an integer. Now every function representation has its own zero/one representations.
- Now using the fantastic Logilab PyLint program!

- Truth tables now use the awesome stdlib array.array for internal representation.
- Changed the names of almost all Expression sublasses to ExprSomething. the Or/And/Not operators are now functions. This simplified lots of crummy `__new__` magic.
- Expression instances to not automatically simplify, but they do if you use Or/And/Not/etc with default `**kwargs`.
- Got rid of `constant` and `binop` modules, of dubious value.
- Added `is_zero`, `is_one`, `box`, and `unbox` to Function interface.
- Removed `reduce`, `iter_zeros`, and `iter_ones` from Function interface.
- Lots of refactoring of SAT methodology.
- Finally implemented `unate` methods correctly for Expressions.

### 1.10.14 Version 0.12

- Lots of work in `pyeda.table`:
  - Now two classes, `TruthTable`, and `PCTable` (for positional-cube format, which allows `X` outputs).
  - Implemented *most* of the `boolfunc.Function` API.
  - Tables now support `-`, `+`, `*`, and `xor` operators.
- Using a set container for And/Or/Xor argument simplification results in about 30% speedup of unit tests.
- Renamed `boolfunc.iter_space` to `boolfunc.iter_points`.
- New `boolfunc.iter_terms` generator.
- Changed `dnf=True` to `conf=False` on several methods that give the option of returnin an expression in conjunctive or disjunctive form.
- Added `conj=False` argument to all expression `factor` methods.
- New `Function.iter_domain` and `Function.iter_image` iterators.
- Renamed `Function.iter_outputs` to `Function.iter_relation`.
- Add `pyeda.alphas` module for a convenience way to grab all the a, b, c, d, ... variables.
- `Xor.factor` now returns a flattened form, instead of nested.

### 1.10.15 Version 0.11

#### Release 0.11.1

- Fixed bug #16: `Function.reduce` only implemented by Variable

#### Release 0.11.0

- In `pyeda.dimacs` changed `parse_cnf` method name to `load_cnf`
- In `pyeda.dimacs` changed `parse_sat` method name to `load_sat`
- In `pyeda.dimacs` added new method `dump_cnf`, to convert expressions to CNF-formatted strings.
- In `pyeda.dimacs` added new method `dump_sat`, to convert expressions to SAT-formatted strings.

- Variables now have a `qualname` attribute, to allow referencing a variable either by its local name or its fully-qualified name.
- Function gained a `reduce` method, to provide a standard interface to reduce Boolean function implementations to their canonical forms.
- Expressions gained a `simplify` parameter, to allow constructing unsimplified expressions.
- Expressions gained an `expand` method, to implement Shannon expansion.
- New if-then-else (ITE) expression type.
- NormalForm expressions now both support $-$, $+$, and $*$ operators.

## 1.11 Reference

### 1.11.1 `pyeda.util` — Utilities

The `pyeda.util` module contains top-level utilities, such as fundamental functions and decorators.

Interface Functions:

- `bit_on()` — Return the value of a number's bit position
- `clog2()` — Return the ceiling log base two of an integer
- `parity()` — Return the parity of an integer

Decorators:

- `cached_property()`

#### Interface Functions

`pyeda.util.`**`bit_on`**(*num: int*, *bit: int*) → int

Return the value of a number's bit position.

For example, since $42 = 2^1 + 2^3 + 2^5$, this function will return 1 in bit positions 1, 3, 5:

```
>>> [bit_on(42, i) for i in range(clog2(42))]
[0, 1, 0, 1, 0, 1]
```

`pyeda.util.`**`clog2`**(*num: int*) → int

Return the ceiling log base two of an integer $\geq 1$.

This function tells you the minimum dimension of a Boolean space with at least N points.

For example, here are the values of `clog2(N)` for $1 \leq N < 18$:

```
>>> [clog2(n) for n in range(1, 18)]
[0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5]
```

This function is undefined for non-positive integers:

```
>>> clog2(0)
Traceback (most recent call last):
    ...
ValueError: expected num >= 1
```

pyeda.util.**parity**(*num: int*) → int
Return the parity of a non-negative integer.

For example, here are the parities of the first ten integers:

```
>>> [parity(n) for n in range(10)]
[0, 1, 1, 0, 1, 0, 0, 1, 1, 0]
```

This function is undefined for negative integers:

```
>>> parity(-1)
Traceback (most recent call last):
    ...
ValueError: expected num >= 0
```

### Decorators

pyeda.util.**cached_property**(*func*)
Return a cached property calculated by the input function.

Unlike the `property` decorator builtin, this decorator will cache the return value in order to avoid repeated calculations. This is particularly useful when the property involves some non-trivial computation.

For example, consider a class that models a right triangle. The hypotenuse `c` will only be calculated once.

```
import math

class RightTriangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @cached_property
    def c(self):
        return math.sqrt(self.a**2 + self.b**2)
```

## 1.11.2 pyeda.boolalg.boolfunc — Boolean Functions

The `pyeda.boolalg.boolfunc` module implements the fundamentals of Boolean space, variables and functions.

Data Types:

**point** A dictionary of `Variable => {0, 1}` mappings. For example, `{a: 0, b: 1, c: 0, d: 1}`. An N-dimensional *point* corresponds to one vertex of an N-dimensional *cube*.

**untyped point** An untyped, immutable representation of a *point*, represented as `(frozenset([int]), frozenset([int]))`. The integers are Variable uniqids. Index zero contains a frozenset of variables mapped to zero, and index one contains a frozenset of variables mapped to one. This representation is easier to manipulate than the *point*, and it is hashable for caching purposes.

**term** A tuple of $N$ Boolean functions, intended to be used as the input of either an OR or AND function. An OR term is called a *maxterm* (product of sums), and an AND term is called a *minterm* (sum of products).

Interface Functions:

- `num2point()` — Convert an integer into a point in an N-dimensional Boolean space
- `num2upoint()` — Convert an integer into an untyped point in an N-dimensional Boolean space.
- `num2term()` — Convert an integer into a min/max term in an N-dimensional Boolean space

- point2upoint() — Convert a point into an untyped point
- point2term() — Convert a point into a min/max term
- iter_points() — Iterate through all points in an N-dimensional Boolean space
- iter_upoints() — Iterate through all untyped points in an N-dimensional Boolean space
- iter_terms() — Iterate through all min/max terms in an N-dimensional Boolean space
- vpoint2point() — Convert a vector point into a point in an N-dimensional Boolean space

Interface Classes:

- Variable
- Function

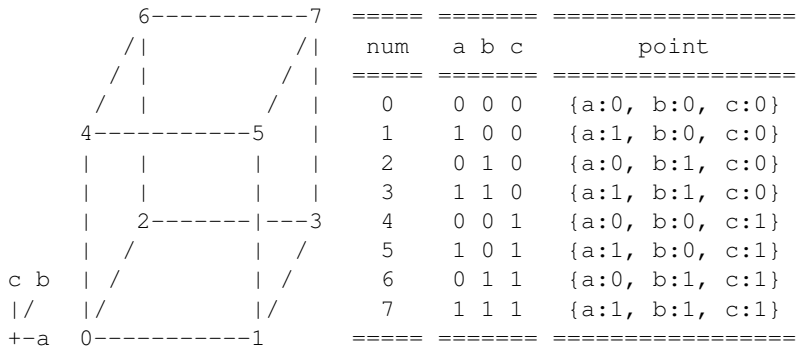
### Interface Functions

pyeda.boolalg.boolfunc.**num2point**(*num*, *vs*)
Convert *num* into a point in an N-dimensional Boolean space.

The *vs* argument is a sequence of $N$ Boolean variables. There are $2^N$ points in the corresponding Boolean space. The dimension number of each variable is its index in the sequence.

The *num* argument is an int in range $[0, 2^N)$.

For example, consider the 3-dimensional space formed by variables $a$, $b$, $c$. Each vertex corresponds to a 3-dimensional point as summarized by the table:




| num | a b c | point |
|---|---|---|
| 0 | 0 0 0 | {a:0, b:0, c:0} |
| 1 | 1 0 0 | {a:1, b:0, c:0} |
| 2 | 0 1 0 | {a:0, b:1, c:0} |
| 3 | 1 1 0 | {a:1, b:1, c:0} |
| 4 | 0 0 1 | {a:0, b:0, c:1} |
| 5 | 1 0 1 | {a:1, b:0, c:1} |
| 6 | 0 1 1 | {a:0, b:1, c:1} |
| 7 | 1 1 1 | {a:1, b:1, c:1} |

**Note:** The a b c column is the binary representation of *num* written in little-endian order.

pyeda.boolalg.boolfunc.**num2upoint**(*num*, *vs*)
Convert *num* into an untyped point in an N-dimensional Boolean space.

The *vs* argument is a sequence of $N$ Boolean variables. There are $2^N$ points in the corresponding Boolean space. The dimension number of each variable is its index in the sequence.

The *num* argument is an int in range $[0, 2^N)$.

See num2point() for a description of how *num* maps onto an N-dimensional point. This function merely converts the output to an immutable (untyped) form.

pyeda.boolalg.boolfunc.**num2term**(*num*, *fs*, *conj=False*)
Convert *num* into a min/max term in an N-dimensional Boolean space.

The *fs* argument is a sequence of $N$ Boolean functions. There are $2^N$ points in the corresponding Boolean space. The dimension number of each function is its index in the sequence.

The *num* argument is an int in range $[0, 2^N)$.

If *conj* is `False`, return a minterm. Otherwise, return a maxterm.

For example, consider the 3-dimensional space formed by functions $f$, $g$, $h$. Each vertex corresponds to a min/max term as summarized by the table:

```
          6-----------7  ===== ======= ========== ==========
         /|          /|   num   f g h    minterm    maxterm
        / |         / |  ===== ======= ========== ==========
       /  |        /  |    0    0 0 0   f' g' h'    f  g  h
      4-----------5   |    1    1 0 0   f  g' h'    f' g  h
      |   |       |   |    2    0 1 0   f' g  h'    f  g' h
      |   |       |   |    3    1 1 0   f  g  h'    f' g' h
      |   2-------|---3    4    0 0 1   f' g' h     f  g  h'
      |  /        |  /     5    1 0 1   f  g' h     f' g  h'
h g   | /         | /      6    0 1 1   f' g  h     f  g' h'
|/    |/          |/       7    1 1 1   f  g  h     f' g' h'
+-f   0-----------1      ===== ======= ========== ==========
```

---

**Note:** The `f g h` column is the binary representation of *num* written in little-endian order.

---

`pyeda.boolalg.boolfunc.`**`point2upoint`**(*point*)
Convert *point* into an untyped point.

`pyeda.boolalg.boolfunc.`**`point2term`**(*point*, *conj=False*)
Convert *point* into a min/max term.

If *conj* is `False`, return a minterm. Otherwise, return a maxterm.

`pyeda.boolalg.boolfunc.`**`iter_points`**(*vs*)
Iterate through all points in an N-dimensional Boolean space.

The *vs* argument is a sequence of $N$ Boolean variables.

`pyeda.boolalg.boolfunc.`**`iter_upoints`**(*vs*)
Iterate through all untyped points in an N-dimensional Boolean space.

The *vs* argument is a sequence of $N$ Boolean variables.

`pyeda.boolalg.boolfunc.`**`iter_terms`**(*fs*, *conj=False*)
Iterate through all min/max terms in an N-dimensional Boolean space.

The *fs* argument is a sequence of $N$ Boolean functions.

If *conj* is `False`, yield minterms. Otherwise, yield maxterms.

`pyeda.boolalg.boolfunc.`**`vpoint2point`**(*vpoint*)
Convert *vpoint* into a point in an N-dimensional Boolean space.

The *vpoint* argument is a mapping from multi-dimensional arrays of variables to matching arrays of $0, 1$. Elements from the values array will be converted to $0, 1$ using the *int* builtin function.

For example:

```
>>> from pyeda.boolalg.expr import exprvar
>>> a = exprvar('a')
>>> b00, b01, b10, b11 = map(exprvar, "b00 b01 b10 b11".split())
>>> vpoint = {a: 0, ((b00, b01), (b10, b11)): ["01", "01"]}
>>> point = vpoint2point(vpoint)
>>> point[a], point[b00], point[b01], point[b10], point[b11]
(0, 0, 1, 0, 1)
```

The vpoint mapping is more concise if B is an farray:

```
>>> from pyeda.boolalg.expr import exprvar
>>> from pyeda.boolalg.bfarray import exprvars
>>> a = exprvar('a')
>>> B = exprvars('b', 2, 2)
>>> vpoint = {a: 0, B: ["01", "01"]}
>>> point = vpoint2point(vpoint)
>>> point[a], point[B[0,0]], point[B[0,1]], point[B[1,0]], point[B[1,1]]
(0, 0, 1, 0, 1)
```

The shape of the array must match the shape of its values:

```
>>> from pyeda.boolalg.bfarray import exprvars
>>> X = exprvars('x', 2, 2)
>>> vpoint = {X: "0101"}
>>> point = vpoint2point(vpoint)
Traceback (most recent call last):
    ...
ValueError: expected 1:1 mapping from Variable => {0, 1}
```

### Interface Classes

**class** pyeda.boolalg.boolfunc.**Variable**(*names*, *indices*)

Base class for a symbolic Boolean variable.

A Boolean *variable* is an abstract numerical quantity that may assume any value in the set $B = \{0, 1\}$.

---

**Note:** Do **NOT** instantiate a Variable directly. Instead, use one of the concrete implementations: pyeda.boolalg.bdd.bddvar() pyeda.boolalg.expr.exprvar(), pyeda.boolalg.table.ttvar().

---

A variable is defined by one or more *names*, and zero or more *indices*. Multiple names establish hierarchical namespaces, and multiple indices group several related variables.

Each variable has a unique, positive integer called the *uniqid*. This integer may be used to identify a variable that is represented by more than one data structure. For example, bddvar('a', 0) and exprvar('a', 0) will refer to two different Variable instances, but both will share the same uniqid.

All variables are implicitly ordered. If two variable names are identical, compare their indices. Otherwise, do a string comparison of their names. This is only useful where variable order matters, and is not meaningful in any algebraic sense.

For example, the following statements are true:

- a == a
- a < b
- a[0] < a[1]
- a[0][1] < a[0][2]

**name**

Return the innermost variable name.

**qualname**

Return the fully qualified name.

**class** pyeda.boolalg.boolfunc.**Function**

Abstract base class that defines an interface for a symbolic Boolean function of $N$ variables.

**__invert__**()

Boolean negation operator

| $f$ | $f'$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**__or__**($g$)

Boolean disjunction (sum, OR) operator

| $f$ | $g$ | $f + g$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**__and__**($g$)

Boolean conjunction (product, AND) operator

| $f$ | $g$ | $f \cdot g$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**__xor__**($g$)

Boolean exclusive or (XOR) operator

| $f$ | $g$ | $f \oplus g$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**__add__**(*other*)

Concatenation operator

The *other* argument may be a Function or an farray.

**__mul__**(*num*)

Repetition operator

**support**

Return the support set of a function.

Let $f(x_1, x_2, \ldots, x_n)$ be a Boolean function of $N$ variables.

The unordered set $\{x_1, x_2, \ldots, x_n\}$ is called the *support* of the function.

**usupport**

Return the untyped support set of a function.

**inputs**

Return the support set in name/index order.

**top**

Return the first variable in the ordered support set.

**degree**

Return the degree of a function.

A function from $B^N \Rightarrow B$ is called a Boolean function of *degree* $N$.

**cardinality**
Return the cardinality of the relation $B^N \Rightarrow B$.

Always equal to $2^N$.

**iter_domain**()
Iterate through all points in the domain.

**iter_image**()
Iterate through all elements in the image.

**iter_relation**()
Iterate through all (point, element) pairs in the relation.

**restrict**(*point*)
Restrict a subset of support variables to $\{0, 1\}$.

Returns a new function: $f(x_i, \ldots)$

$f \mid x_i = b$

**vrestrict**(*vpoint*)
Expand all vectors in *vpoint* before applying `restrict`.

**compose**(*mapping*)
Substitute a subset of support variables with other Boolean functions.

Returns a new function: $f(g_i, \ldots)$

$f_1 \mid x_i = f_2$

**satisfy_one**()
If this function is satisfiable, return a satisfying input point. A tautology *may* return a zero-dimensional point; a contradiction *must* return None.

**satisfy_all**()
Iterate through all satisfying input points.

**satisfy_count**()
Return the cardinality of the set of all satisfying input points.

**iter_cofactors**(*vs=None*)
Iterate through the cofactors of a function over N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $f_{x_i} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i'$ is: $f_{x_i'} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$

**cofactors**(*vs=None*)
Return a tuple of the cofactors of a function over N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $f_{x_i} = f(x_1, x_2, \ldots, 1, \ldots, x_n)$

The *cofactor* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i'$ is: $f_{x_i'} = f(x_1, x_2, \ldots, 0, \ldots, x_n)$

**smoothing**(*vs=None*)
Return the smoothing of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *smoothing* of $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to variable $x_i$ is: $S_{x_i}(f) = f_{x_i} + f_{x_i'}$

This is the same as the existential quantification operator: $\exists\{x_1, x_2, \dots\}\ f$

**consensus**(*vs=None*)
Return the consensus of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *consensus* of $f(x_1, x_2, \dots, x_i, \dots, x_n)$ with respect to variable $x_i$ is: $C_{x_i}(f) = f_{x_i} \cdot f_{x_i'}$

This is the same as the universal quantification operator: $\forall\{x_1, x_2, \dots\}\ f$

**derivative**(*vs=None*)
Return the derivative of a function over a sequence of N variables.

The *vs* argument is a sequence of $N$ Boolean variables.

The *derivative* of $f(x_1, x_2, \dots, x_i, \dots, x_n)$ with respect to variable $x_i$ is: $\frac{\partial}{\partial x_i} f = f_{x_i} \oplus f_{x_i'}$

This is also known as the Boolean *difference*.

**is_zero**()
Return whether this function is zero.

---

**Note:** This method will only look for a particular "zero form", and will **NOT** do a full search for a contradiction.

---

**is_one**()
Return whether this function is one.

---

**Note:** This method will only look for a particular "one form", and will **NOT** do a full search for a tautology.

---

**static box**(*obj*)
Convert primitive types to a Function.

**unbox**()
Return integer 0 or 1 if possible, otherwise return the Function.

### 1.11.3 pyeda.boolalg.bdd — Binary Decision Diagrams

The pyeda.boolalg.bdd module implements Boolean functions represented as binary decision diagrams.

Interface Functions:

- bddvar() — Return a unique BDD variable
- expr2bdd() — Convert an expression into a binary decision diagram
- bdd2expr() — Convert a binary decision diagram into an expression
- upoint2bddpoint() — Convert an untyped point into a BDD point
- ite() — BDD if-then-else operator

Interface Classes:

- BDDNode
- BinaryDecisionDiagram
  - BDDConstant
  - BDDVariable

### Interface Functions

pyeda.boolalg.bdd.**bddvar**(*name*, *index=None*)
Return a unique BDD variable.

A Boolean *variable* is an abstract numerical quantity that may assume any value in the set $B = \{0, 1\}$. The `bddvar` function returns a unique Boolean variable instance represented by a binary decision diagram. Variable instances may be used to symbolically construct larger BDDs.

A variable is defined by one or more *names*, and zero or more *indices*. Multiple names establish hierarchical namespaces, and multiple indices group several related variables. If the `name` parameter is a single `str`, it will be converted to `(name, )`. The `index` parameter is optional; when empty, it will be converted to an empty tuple `()`. If the `index` parameter is a single `int`, it will be converted to `(index, )`.

Given identical names and indices, the `bddvar` function will always return the same variable:

```
>>> bddvar('a', 0) is bddvar('a', 0)
True
```

To create several single-letter variables:

```
>>> a, b, c, d = map(bddvar, 'abcd')
```

To create variables with multiple names (inner-most first):

```
>>> fifo_push = bddvar(('push', 'fifo'))
>>> fifo_pop = bddvar(('pop', 'fifo'))
```

**See also:**

For creating arrays of variables with incremental indices, use the `pyeda.boolalg.bfarray.bddvars()` function.

pyeda.boolalg.bdd.**expr2bdd**(*expr*)
Convert an expression into a binary decision diagram.

pyeda.boolalg.bdd.**bdd2expr**(*bdd*, *conj=False*)
Convert a binary decision diagram into an expression.

This function will always return an expression in two-level form. If *conj* is `False`, return a sum of products (SOP). Otherwise, return a product of sums (POS).

For example:

```
>>> a, b = map(bddvar, 'ab')
>>> bdd2expr(~a & b | a & ~b)
Or(And(~a, b), And(a, ~b))
```

pyeda.boolalg.bdd.**upoint2bddpoint**(*upoint*)
Convert an untyped point into a BDD point.

**See also:**

For definitions of points an untyped points, see the `pyeda.boolalg.boolfunc` module.

pyeda.boolalg.bdd.**ite**(*f*, *g*, *h*)
BDD if-then-else (ITE) operator

The *f*, *g*, and *h* arguments are BDDs.

The ITE(f, g, h) operator means "if *f* is true, return *g*, else return *h*".

It is equivalent to:

- DNF form: `f & g | ~f & h`
- CNF form: `(~f | g) & (f | h)`

### Interface Classes

**class** `pyeda.boolalg.bdd.`**`BDDNode`**(*root*, *lo*, *hi*)
Binary decision diagram node

A BDD node represents one cofactor in the decomposition of a Boolean function. Nodes are uniquely identified by a `root` integer, `lo` child node, and `hi` child node:

- `root` is the cofactor variable's `uniqid` attribute
- `lo` is the zero cofactor node
- `hi` is the one cofactor node

The `root` of the zero node is -2, and the `root` of the one node is -1. Both zero/one nodes have `lo=None` and `hi=None`.

Do **NOT** create BDD nodes using the `BDDNode` constructor. BDD node instances are managed internally.

**class** `pyeda.boolalg.bdd.`**`BinaryDecisionDiagram`**(*node*)
Boolean function represented by a binary decision diagram

**See also:**

This is a subclass of `pyeda.boolalg.boolfunc.Function`

BDDs have a single attribute, `node`, that points to a node in the managed unique table.

There are two ways to construct a BDD:

- Convert an expression using the `expr2bdd` function.
- Use operators on existing BDDs.

Use the `bddvar` function to create BDD variables, and use the Python `~ | & ^` operators for NOT, OR, AND, XOR.

For example:

```
>>> a, b, c, d = map(bddvar, 'abcd')
>>> f = ~a | b & c ^ d
```

The `BinaryDecisionDiagram` class is useful for type checking, e.g. `isinstance(f, BinaryDecisionDiagram)`.

Do **NOT** create a BDD using the `BinaryDecisionDiagram` constructor. BDD instances are managed internally, and you will not be able to use the Python `is` operator to establish formal equivalence with manually constructed BDDs.

**`dfs_preorder`**()
Iterate through nodes in depth first search (DFS) pre-order.

**`dfs_postorder`**()
Iterate through nodes in depth first search (DFS) post-order.

**`bfs`**()
Iterate through nodes in breadth first search (BFS) order.

**equivalent**(*other*)
Return whether this BDD is equivalent to *other*.

You can also use Python's `is` operator for BDD equivalency testing.

For example:

```
>>> a, b, c = map(bddvar, 'abc')
>>> f1 = a ^ b ^ c
>>> f2 = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c
>>> f1 is f2
True
```

**to_dot**(*name='BDD'*)
Convert to DOT language representation.

See the DOT language reference for details.

**class** pyeda.boolalg.bdd.**BDDConstant**(*node*)
Binary decision diagram constant zero/one

The `BDDConstant` class is useful for type checking, e.g. `isinstance(f, BDDConstant)`.

Do **NOT** create a BDD using the `BDDConstant` constructor. BDD instances are managed internally, and the BDD zero/one instances are singletons.

**class** pyeda.boolalg.bdd.**BDDVariable**(*bvar*)
Binary decision diagram variable

The `BDDVariable` class is useful for type checking, e.g. `isinstance(f, BDDVariable)`.

Do **NOT** create a BDD using the `BDDVariable` constructor. Use the `bddvar()` function instead.

### 1.11.4 pyeda.boolalg.expr — Expressions

The `pyeda.boolalg.expr` module implements Boolean functions represented as expressions.

Data Types:

**abstract syntax tree** A nested tuple of entries that represents an expression. It is defined recursively:

```
ast := ('const', bool)
     | ('var', names, indices)
     | ('not', ast)
     | ('implies', ast, ast)
     | ('ite', ast, ast, ast)
     | (func, ast, ...)

bool := 0 | 1

names := (name, ...)

indices := (index, ...)

func := 'or'
      | 'nor'
      | 'and'
      | 'nand'
      | 'xor'
      | 'xnor'
      | 'equal'
```

```
        | 'unequal'
        | 'onehot0'
        | 'onehot'
        | 'majority'
        | 'achillesheel'
```

Interface Functions:

- `exprvar()` — Return a unique expression variable
- `expr()` — Convert an arbitrary object into an Expression
- `ast2expr()` — Convert an abstract syntax tree to an Expression
- `expr2dimacscnf()` — Convert an expression into an equivalent DIMACS CNF
- `upoint2exprpoint()` — Convert an untyped point to an Expression point
- `Not()` — Expression negation operator
- `Or()` — Expression disjunction (sum, OR) operator
- `And()` — Expression conjunction (product, AND) operator
- `Nor()` — Expression NOR (not OR) operator
- `Nand()` — Expression NAND (not AND) operator
- `Xor()` — Expression exclusive or (XOR) operator
- `Xnor()` — Expression exclusive nor (XNOR) operator
- `Equal()` — Expression equality operator
- `Unequal()` — Expression inequality operator
- `Implies()` — Expression implication operator
- `ITE()` — Expression If-Then-Else (ITE) operator
- `OneHot0()`
- `OneHot()`
- `Majority()`
- `AchillesHeel()`
- `Mux()`

Interface Classes:

- `Expression`
  - `ExprConstant`
  - `ExprLiteral`
    - `ExprVariable`
    - `ExprComplement`
  - `ExprNot`
  - `ExprOrAnd`
    - `ExprOr`
    - `ExprAnd`

### Interface Functions

pyeda.boolalg.expr.**exprvar**(*name*, *index=None*)

Return a unique Expression variable.

A Boolean *variable* is an abstract numerical quantity that may assume any value in the set $B = \{0, 1\}$. The `exprvar` function returns a unique Boolean variable instance represented by an expression. Variable instances may be used to symbolically construct larger expressions.

A variable is defined by one or more *names*, and zero or more *indices*. Multiple names establish hierarchical namespaces, and multiple indices group several related variables. If the `name` parameter is a single `str`, it will be converted to `(name, )`. The `index` parameter is optional; when empty, it will be converted to an empty tuple `()`. If the `index` parameter is a single `int`, it will be converted to `(index, )`.

Given identical names and indices, the `exprvar` function will always return the same variable:

```
>>> exprvar('a', 0) is exprvar('a', 0)
True
```

To create several single-letter variables:

```
>>> a, b, c, d = map(exprvar, 'abcd')
```

To create variables with multiple names (inner-most first):

```
>>> fifo_push = exprvar(('push', 'fifo'))
>>> fifo_pop = exprvar(('pop', 'fifo'))
```

**See also:**

For creating arrays of variables with incremental indices, use the `pyeda.boolalg.bfarray.exprvars()` function.

pyeda.boolalg.expr.**expr**(*obj*, *simplify=True*)

Convert an arbitrary object into an Expression.

pyeda.boolalg.expr.**ast2expr**(*ast*)
Convert an abstract syntax tree to an Expression.

pyeda.boolalg.expr.**expr2dimacscnf**(*expr*)
Convert an expression into an equivalent DIMACS CNF.

pyeda.boolalg.expr.**upoint2exprpoint**(*upoint*)
Convert an untyped point to an Expression point.

#### Operators

**Primary Operators**

pyeda.boolalg.expr.**Not**(*arg*, *simplify=True*)
Expression negation operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Or**(**args*, *simplify=True*)
Expression disjunction (sum, OR) operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**And**(**args*, *simplify=True*)
Expression conjunction (product, AND) operator

If *simplify* is `True`, return a simplified expression.

**Secondary Operators**

pyeda.boolalg.expr.**Nor**(**args*, *simplify=True*)
Expression NOR (not OR) operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Nand**(**args*, *simplify=True*)
Expression NAND (not AND) operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Xor**(**args*, *simplify=True*)
Expression exclusive or (XOR) operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Xnor**(**args*, *simplify=True*)
Expression exclusive nor (XNOR) operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Equal**(**args*, *simplify=True*)
Expression equality operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Unequal**(**args*, *simplify=True*)
Expression inequality operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Implies**(*p*, *q*, *simplify=True*)
Expression implication operator

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**ITE**(*s*, *d1*, *d0*, *simplify=True*)
Expression If-Then-Else (ITE) operator

If *simplify* is `True`, return a simplified expression.

**High Order Operators**

pyeda.boolalg.expr.**OneHot0**(**args*, *simplify=True*, *conj=True*)
Return an expression that means "at most one input function is true".

If *simplify* is `True`, return a simplified expression.

If *conj* is `True`, return a CNF. Otherwise, return a DNF.

pyeda.boolalg.expr.**OneHot**(**args*, *simplify=True*, *conj=True*)
Return an expression that means "exactly one input function is true".

If *simplify* is `True`, return a simplified expression.

If *conj* is `True`, return a CNF. Otherwise, return a DNF.

pyeda.boolalg.expr.**Majority**(**args*, *simplify=True*, *conj=False*)
Return an expression that means "the majority of input functions are true".

If *simplify* is `True`, return a simplified expression.

If *conj* is `True`, return a CNF. Otherwise, return a DNF.

pyeda.boolalg.expr.**AchillesHeel**(**args*, *simplify=True*)
Return the Achille's Heel function, defined as: $\prod_{i=0}^{n/2-1} X_{2i} + X_{2i+1}$.

If *simplify* is `True`, return a simplified expression.

pyeda.boolalg.expr.**Mux**(*fs*, *sel*, *simplify=True*)
Return an expression that multiplexes a sequence of input functions over a sequence of select functions.

## Interface Classes

### Expression Tree Nodes

**class** pyeda.boolalg.expr.**Expression**
Boolean function represented by a logic expression

**See also:**

This is a subclass of pyeda.boolalg.boolfunc.Function

An expression is a tree data structure with operators at the branches, and constants/literals at the leaves. A literal is a variable or its complement.

There are two ways to construct an expression:

- Convert another function representation.
- Use operators on existing expressions.

Use the `exprvar` function to create expression variables, and use the Python `~ | & ^` operators for NOT, OR, AND, XOR. Additionally, expressions overload `>>` for IMPLIES.

For example:

```
>>> a, b, c, d, p, q = map(exprvar, 'abcdpq')
>>> f = ~a | b & c ^ d
>>> g = p >> q
```

To create unsimplified expressions, use class constructors:

```
>>> ExprOr(0, a)
Or(0, a)
```

To create simplified expressions, use function form operators with `simplify=True`, or Python `~|&^` operators:

```
>>> Or(0, a)
a
>>> 0 | a
a
```

**to_unicode**()
Return a string representation using Unicode symbols.

**to_latex**()
Return a string representation using Latex.

**__rshift__**(*g*)
Boolean implication operator

| $f$ | $g$ | $f \implies g$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**invert**()
Return an inverted expression.

**simplify**()
Return a simplified expression.

**simplified**
Return True if the expression is simplified.

**factor**(*conj=False*)
Return a factored expression.

A factored expression is one and only one of the following:

- A literal.
- A disjunction / conjunction of factored expressions.

Parameters

**conj** [bool] Always choose a conjunctive form when there's a choice

**depth**
Return the depth of the expression tree.

Expression depth is defined recursively:

1. A leaf node (constant or literal) has zero depth.
2. A branch node (operator) has depth equal to the maximum depth of its children (arguments) plus one.

**to_ast**()
Return the expression converted to an abstract syntax tree.

**expand**(*vs=None*, *conj=False*)
Return the Shannon expansion with respect to a list of variables.

**flatten**(*op*)
Return a factored, flattened expression.

Use the distributive law to flatten all nested expressions:

$$a + (b \cdot c) = (a + b) \cdot (a + c)$$
$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

**op** [ExprOr or ExprAnd] The operator you want to flatten. For example, if you want to produce an ExprOr expression, flatten all the nested ExprAnds.

**to_dnf**(*flatten=True*)
Return the expression in disjunctive normal form.

**to_cdnf**(*flatten=True*)
Return the expression in canonical disjunctive normal form.

**to_cnf**(*flatten=True*)
Return the expression in conjunctive normal form.

**to_ccnf**(*flatten=True*)
Return the expression in canonical conjunctive normal form.

**cover**
Return the DNF expression as a cover of cubes.

**absorb**()
Return the DNF/CNF expression after absorption.

$$a + (a \cdot b) = a$$
$$a \cdot (a + b) = a$$

**reduce**()
Reduce the DNF/CNF expression to a canonical form.

**encode_inputs**()
Return a compact encoding for input variables.

**encode_dnf**()
Encode as a compact DNF.

**encode_cnf**()
Encode as a compact CNF.

**tseitin**(*auxvarname='aux'*)
Convert the expression to Tseitin's encoding.

**complete_sum**()
Return a DNF that contains all prime implicants.

**equivalent**(*other*)
Return True if this expression is equivalent to other.

**to_dot**(*name='EXPR'*)
Convert to DOT language representation.

**class** pyeda.boolalg.expr.**ExprConstant**
Expression constant

**class** pyeda.boolalg.expr.**ExprLiteral**
An instance of a variable or of its complement

**class** pyeda.boolalg.expr.**ExprVariable**(*bvar*)
Expression variable

**class** pyeda.boolalg.expr.**ExprComplement**(*exprvar*)
Expression complement

**class** pyeda.boolalg.expr.**ExprNot**(*arg*)
Expression NOT operator

**class** pyeda.boolalg.expr.**ExprOrAnd**(**args*)
Base class for Expression OR/AND expressions

**class** pyeda.boolalg.expr.**ExprOr**(**args*)
Expression OR operator

**class** pyeda.boolalg.expr.**ExprAnd**(**args*)
Expression AND operator

**class** pyeda.boolalg.expr.**ExprNorNand**(**args*)
Base class for Expression NOR/NAND expressions

**class** pyeda.boolalg.expr.**ExprNor**(**args*)
Expression NOR operator

**class** pyeda.boolalg.expr.**ExprNand**(**args*)
Expression NAND operator

**class** pyeda.boolalg.expr.**ExprExclusive**(**args*)
Expression exclusive (XOR, XNOR) operator

**class** pyeda.boolalg.expr.**ExprXor**(**args*)
Expression Exclusive OR (XOR) operator

**class** pyeda.boolalg.expr.**ExprXnor**(**args*)
Expression Exclusive NOR (XNOR) operator

**class** pyeda.boolalg.expr.**ExprEqualBase**(**args*)
Expression equality (EQUAL, UNEQUAL) operators

**class** pyeda.boolalg.expr.**ExprEqual**(**args*)
Expression EQUAL operator

**class** pyeda.boolalg.expr.**ExprUnequal**(**args*)
Expression UNEQUAL operator

**class** pyeda.boolalg.expr.**ExprImplies**(*p*, *q*)
Expression implication operator

**class** pyeda.boolalg.expr.**ExprITE**(*s*, *d1*, *d0*)
Expression if-then-else ternary operator

#### Normal Forms

**class** pyeda.boolalg.expr.**NormalForm**(*nvars*, *clauses*)
Normal form expression

**class** pyeda.boolalg.expr.**DisjNormalForm**(*nvars*, *clauses*)
Disjunctive normal form expression

**class** pyeda.boolalg.expr.**ConjNormalForm**(*nvars*, *clauses*)
Conjunctive normal form expression

**class** pyeda.boolalg.expr.**DimacsCNF**(*nvars*, *clauses*)
Wrapper class for a DIMACS CNF representation

### 1.11.5 pyeda.boolalg.bfarray — Boolean Function Arrays

The pyeda.boolalg.bfarray module implements multi-dimensional arrays of Boolean functions.

Interface Functions:

- bddzeros() — Return a multi-dimensional array of BDD zeros
- bddones() — Return a multi-dimensional array of BDD ones
- bddvars() — Return a multi-dimensional array of BDD variables
- exprzeros() — Return a multi-dimensional array of expression zeros
- exprones() — Return a multi-dimensional array of expression ones
- exprvars() — Return a multi-dimensional array of expression variables
- ttzeros() — Return a multi-dimensional array of truth table zeros
- ttones() — Return a multi-dimensional array of truth table ones
- ttvars() — Return a multi-dimensional array of truth table variables
- uint2bdds() — Convert unsigned *num* to an array of BDDs
- uint2exprs() — Convert unsigned *num* to an array of expressions
- uint2tts() — Convert unsigned *num* to an array of truth tables
- int2bdds() — Convert *num* to an array of BDDs
- int2exprs() — Convert *num* to an array of expressions
- int2tts() — Convert *num* to an array of truth tables
- fcat() — Concatenate a sequence of farrays

Interface Classes:

- farray()

#### Interface Functions

pyeda.boolalg.bfarray.**bddzeros**(**dims*)
Return a multi-dimensional array of BDD zeros.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single int, it will be converted to (0, stop).

The dimension starts at index start, and increments by one up to, but not including, stop. This follows the Python slice convention.

For example, to create a 4x4 array of BDD zeros:

```
>>> zeros = bddzeros(4, 4)
>>> zeros
farray([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

pyeda.boolalg.bfarray.**bddones**(**dims*)
Return a multi-dimensional array of BDD ones.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of BDD ones:

```
>>> ones = bddones(4, 4)
>>> ones
farray([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]])
```

pyeda.boolalg.bfarray.**bddvars**(*name*, **dims*)
Return a multi-dimensional array of BDD variables.

The *name* argument is passed directly to the `pyeda.boolalg.bdd.bddvar()` function, and may be either a `str` or tuple of `str`.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of BDD variables:

```
>>> vs = bddvars('a', 4, 4)
>>> vs
farray([[a[0,0], a[0,1], a[0,2], a[0,3]],
        [a[1,0], a[1,1], a[1,2], a[1,3]],
        [a[2,0], a[2,1], a[2,2], a[2,3]],
        [a[3,0], a[3,1], a[3,2], a[3,3]]])
```

pyeda.boolalg.bfarray.**exprzeros**(**dims*)
Return a multi-dimensional array of expression zeros.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of expression zeros:

```
>>> zeros = exprzeros(4, 4)
>>> zeros
farray([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

pyeda.boolalg.bfarray.**exprones**(**dims*)
Return a multi-dimensional array of expression ones.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of expression ones:

```
>>> ones = exprones(4, 4)
>>> ones
farray([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]])
```

pyeda.boolalg.bfarray.**exprvars**(*name*, **dims*)

Return a multi-dimensional array of expression variables.

The *name* argument is passed directly to the `pyeda.boolalg.expr.exprvar()` function, and may be either a `str` or tuple of `str`.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of expression variables:

```
>>> vs = exprvars('a', 4, 4)
>>> vs
farray([[a[0,0], a[0,1], a[0,2], a[0,3]],
        [a[1,0], a[1,1], a[1,2], a[1,3]],
        [a[2,0], a[2,1], a[2,2], a[2,3]],
        [a[3,0], a[3,1], a[3,2], a[3,3]]])
```

pyeda.boolalg.bfarray.**ttzeros**(**dims*)

Return a multi-dimensional array of truth table zeros.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of truth table zeros:

```
>>> zeros = ttzeros(4, 4)
>>> zeros
farray([[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]])
```

pyeda.boolalg.bfarray.**ttones**(**dims*)

Return a multi-dimensional array of truth table ones.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of truth table ones:

```
>>> ones = ttones(4, 4)
>>> ones
farray([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]])
```

pyeda.boolalg.bfarray.**ttvars**(*name*, **dims*)

Return a multi-dimensional array of truth table variables.

The *name* argument is passed directly to the `pyeda.boolalg.table.ttvar()` function, and may be either a `str` or tuple of `str`.

The variadic *dims* input is a sequence of dimension specs. A dimension spec is a two-tuple: (start index, stop index). If a dimension is given as a single `int`, it will be converted to `(0, stop)`.

The dimension starts at index `start`, and increments by one up to, but not including, `stop`. This follows the Python slice convention.

For example, to create a 4x4 array of truth table variables:

```
>>> vs = ttvars('a', 4, 4)
>>> vs
farray([[a[0,0], a[0,1], a[0,2], a[0,3]],
        [a[1,0], a[1,1], a[1,2], a[1,3]],
        [a[2,0], a[2,1], a[2,2], a[2,3]],
        [a[3,0], a[3,1], a[3,2], a[3,3]]])
```

pyeda.boolalg.bfarray.**uint2bdds**(*num*, *length=None*)

Convert unsigned *num* to an array of BDDs.

The *num* argument is a non-negative integer.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be zero-extended to match *length*.

For example, to convert the byte 42 (binary `0b00101010`):

```
>>> uint2bdds(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
```

pyeda.boolalg.bfarray.**uint2exprs**(*num*, *length=None*)

Convert unsigned *num* to an array of expressions.

The *num* argument is a non-negative integer.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be zero-extended to match *length*.

For example, to convert the byte 42 (binary `0b00101010`):

```
>>> uint2exprs(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
```

pyeda.boolalg.bfarray.**uint2tts**(*num*, *length=None*)

Convert unsigned *num* to an array of truth tables.

The *num* argument is a non-negative integer.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be zero-extended to match *length*.

For example, to convert the byte 42 (binary `0b00101010`):

```
>>> uint2tts(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
```

pyeda.boolalg.bfarray.**int2bdds**(*num*, *length=None*)

Convert *num* to an array of BDDs.

The *num* argument is an `int`. Negative numbers will be converted using twos-complement notation.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be sign-extended to match *length*.

For example, to convert the bytes 42 (binary `0b00101010`), and -42 (binary `0b11010110`):

```
>>> int2bdds(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
>>> int2bdds(-42, 8)
farray([0, 1, 1, 0, 1, 0, 1, 1])
```

pyeda.boolalg.bfarray.**int2exprs**(*num*, *length=None*)

Convert *num* to an array of expressions.

The *num* argument is an `int`. Negative numbers will be converted using twos-complement notation.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be sign-extended to match *length*.

For example, to convert the bytes 42 (binary `0b00101010`), and -42 (binary `0b11010110`):

```
>>> int2exprs(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
>>> int2exprs(-42, 8)
farray([0, 1, 1, 0, 1, 0, 1, 1])
```

pyeda.boolalg.bfarray.**int2tts**(*num*, *length=None*)

Convert *num* to an array of truth tables.

The *num* argument is an `int`. Negative numbers will be converted using twos-complement notation.

If no *length* parameter is given, the return value will have the minimal required length. Otherwise, the return value will be sign-extended to match *length*.

For example, to convert the bytes 42 (binary `0b00101010`), and -42 (binary `0b11010110`):

```
>>> int2tts(42, 8)
farray([0, 1, 0, 1, 0, 1, 0, 0])
>>> int2tts(-42, 8)
farray([0, 1, 1, 0, 1, 0, 1, 1])
```

pyeda.boolalg.bfarray.**fcat**(**fs*)

Concatenate a sequence of farrays.

The variadic *fs* input is a homogeneous sequence of functions or arrays.

### Interface Classes

**class** pyeda.boolalg.bfarray.**farray**(*objs*, *shape=None*, *ftype=None*)

Multi-dimensional array of Boolean functions

The *objs* argument is a nested sequence of homogeneous Boolean functions. That is, both [a, b, c, d] and [[a, b], [c, d]] are valid inputs.

The optional *shape* parameter is a tuple of dimension specs, which are `(int, int)` tuples. It must match the volume of *objs*. The shape can always be automatically determined from *objs*, but you can supply it to automatically reshape a flat input.

The optional *ftype* parameter is a proper subclass of `Function`. It must match the homogeneous type of *objs*. In most cases, *ftype* can automatically be determined from *objs*. The one exception is that you must provide *ftype* for `objs=[]` (an empty array).

**__invert__**()
Bit-wise NOT operator

**__or__**(*other*)
Bit-wise OR operator

**__and__**(*other*)
Bit-wise AND operator

**__xor__**(*other*)
Bit-wise XOR operator

**__lshift__**(*obj*)
Left shift operator

The *obj* argument may either be an `int`, or `(int, farray)`. The `int` argument is *num*, and the `farray` argument is *cin*.

**See also:**

`lsh()`

**__rshift__**(*obj*)
Right shift operator

The *obj* argument may either be an `int`, or `(int, farray)`. The `int` argument is *num*, and the `farray` argument is *cin*.

**See also:**

`rsh()`

**__add__**(*other*)
Concatenation operator

The *other* argument may be a Function or farray.

**__mul__**(*num*)
Repetition operator

**restrict**(*point*)
Apply the `restrict` method to all functions.

Returns a new farray.

**vrestrict**(*vpoint*)
Expand all vectors in *vpoint* before applying `restrict`.

**compose**(*mapping*)
Apply the `compose` method to all functions.

Returns a new farray.

**size**
Return the size of the array.

The *size* of a multi-dimensional array is the product of the sizes of its dimensions.

**offsets**
Return a tuple of dimension offsets.

**ndim**
Return the number of dimensions.

**reshape**(**dims*)
Return an equivalent farray with a modified shape.

**flat**
Return a 1D iterator over the farray.

**to_uint**()
Convert vector to an unsigned integer, if possible.

This is only useful for arrays filled with zero/one entries.

**to_int**()
Convert vector to an integer, if possible.

This is only useful for arrays filled with zero/one entries.

**zext**(*num*)
Zero-extend this farray by *num* bits.

Returns a new farray.

**sext**(*num*)
Sign-extend this farray by *num* bits.

Returns a new farray.

**uor**()
Unary OR reduction operator

**unor**()
Unary NOR reduction operator

**uand**()
Unary AND reduction operator

**unand**()
Unary NAND reduction operator

**uxor**()
Unary XOR reduction operator

**uxnor**()
Unary XNOR reduction operator

**lsh**(*num*, *cin=None*)
Left shift the farray by *num* places.

The *num* argument must be a non-negative `int`.

If the *cin* farray is provided, it will be shifted in. Otherwise, the carry-in is zero.

Returns a two-tuple (farray fs, farray cout), where *fs* is the shifted vector, and *cout* is the "carry out".

Returns a new farray.

**rsh**(*num*, *cin=None*)
Right shift the farray by *num* places.

The *num* argument must be a non-negative `int`.

If the *cin* farray is provided, it will be shifted in. Otherwise, the carry-in is zero.

Returns a two-tuple (farray fs, farray cout), where *fs* is the shifted vector, and *cout* is the "carry out".

Returns a new farray.

**arsh**(*num*)
Arithmetically right shift the farray by *num* places.

The *num* argument must be a non-negative `int`.

The carry-in will be the value of the most significant bit.

Returns a new farray.

**decode**()
Return a $N \to 2^N$ decoder.

Example Truth Table for a 2:4 decoder:

| $A_1$ | $A_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

### 1.11.6 `pyeda.boolalg.minimization` — Logic Minimization

The `pyeda.boolalg.minimization` module contains interface functions for two-level logic minimization.

Interface Functions:

- `espresso_exprs()`
- `espresso_tts()`

#### Interface Functions

pyeda.boolalg.minimization.**espresso_exprs**(**exprs*)
Return a tuple of expressions optimized using Espresso.

The variadic *exprs* argument is a sequence of expressions.

For example:

```
>>> from pyeda.boolalg.expr import exprvar
>>> a, b, c = map(exprvar, 'abc')
>>> f1 = ~a & ~b & ~c | ~a & ~b & c | a & ~b & c | a & b & c | a & b & ~c
>>> f2 = f2 = ~a & ~b & c | a & ~b & c
>>> f1m, f2m = espresso_exprs(f1, f2)
>>> f1m
Or(And(~a, ~b), And(a, b), And(~b, c))
>>> f2m
And(~b, c)
```

pyeda.boolalg.minimization.**espresso_tts**(**tts*)
Return a tuple of expressions optimized using Espresso.

The variadic *tts* argument is a sequence of truth tables.

For example:

```
>>> from pyeda.boolalg.bfarray import ttvars
>>> from pyeda.boolalg.table import truthtable
>>> X = ttvars('x', 4)
>>> f1 = truthtable(X, "0000011111------")
>>> f2 = truthtable(X, "0001111100------")
>>> f1m, f2m = espresso_tts(f1, f2)
>>> f1m
Or(x[3], And(x[0], x[2]), And(x[1], x[2]))
>>> f2m
Or(x[2], And(x[0], x[1]))
```

### 1.11.7 `pyeda.boolalg.picosat` — PicoSAT C Extension

Interface to PicoSAT SAT solver C extension

#### Constants

**VERSION**
PicoSAT version string

**COPYRIGHT**
PicoSAT copyright statement

#### Exceptions

**exception PicosatError**
An error happened inside PicoSAT. Examples include initialization errors, and unexpected return codes.

#### Interface Functions

**satisfy_one**(*nvars*, *clauses*, *assumptions*, *verbosity=0*, *default_phase=2*, *propagation_limit=-1*, *decision_limit=-1*)
If the input CNF is satisfiable, return a satisfying input point. A contradiction will return None.

Parameters

**nvars** [posint] Number of variables in the CNF

**clauses** [iter of iter of (nonzero) int] The CNF clauses

**assumptions** [iter of (nonzero) int] Add assumptions (unit clauses) to the CNF

**verbosity** [int, optional] Set verbosity level. A verbosity level of 1 and above prints more and more detailed progress reports to stdout.

default_phase : {0, 1, 2, 3}

**Set default initial phase:**

- 0 = false
- 1 = true
- 2 = Jeroslow-Wang (default)
- 3 = random

**progagation_limit** [int] Set a limit on the number of propagations. A negative value sets no propagation limit.

**decision_limit** [int] Set a limit on the number of decisions. A negative value sets no decision limit.

Returns

**tuple of {-1, 0, 1}**

- -1 : zero
- 0 : dont-care
- 1 : one

**satisfy_all**(*nvars*, *clauses*, *verbosity=0*, *default_phase=2*, *propagation_limit=-1*, *decision_limit=-1*)

Iterate through all satisfying input points.

Parameters

**nvars** [posint] Number of variables in the CNF

**clauses** [iter of iter of (nonzero) int] The CNF clauses

**verbosity** [int, optional] Set verbosity level. A verbosity level of 1 and above prints more and more detailed progress reports to stdout.

**default_phase** [{0, 1, 2, 3}]

**Set default initial phase:**

- 0 = false
- 1 = true
- 2 = Jeroslow-Wang (default)
- 3 = random

**progagation_limit** [int] Set a limit on the number of propagations. A negative value sets no propagation limit.

**decision_limit** [int] Set a limit on the number of decisions. A negative value sets no decision limit.

Returns

**iter of tuple of {-1, 0, 1}**

- -1 : zero
- 0 : dont-care
- 1 : one

### 1.11.8 pyeda.boolalg.espresso — Espresso C Extension

#### Constants

**FTYPE**

**DTYPE**

**RTYPE**

### Exceptions

**exception** **EspressoError**
An error happened inside Espresso.

### Interface Functions

**get_config**()
Return a dict of Espresso global configuration values.

**set_config**(*single_expand=0*, *remove_essential=0*, *force_irredundant=0*, *unwrap_onset=0*, *recompute_onset=0*, *use_super_gasp=0*, *skip_make_sparse=0*)
Set Espresso global configuration values.

**espresso**(*ninputs*, *noutputs*, *cover*, *intype=FTYPE|DTYPE*)
Return a logically equivalent, (near) minimal cost set of product-terms to represent the ON-set and optionally minterms that lie in the DC-set, without containing any minterms of the OFF-set.

Parameters

**ninputs** [posint] Number of inputs in the implicant in-part vector.

**noutputs** [posint] Number of outputs in the implicant out-part vector.

**cover** [iter(((int), (int)))] The iterator over multi-output implicants. A multi-output implicant is a pair of row vectors of dimension *ninputs*, and *noutputs*, respectively. The input part contains integers in positional cube notation, and the output part contains entries in {0, 1, 2}.

- '0' means 0 for R-type covers, otherwise has no meaning.
- '1' means 1 for F-type covers, otherwise has no meaning.
- '2' means "don't care" for D-type covers, otherwise has no meaning.

**intype** [int] A flag field that indicates the type of the input cover. F-type = 1, D-type = 2, R-type = 4

Returns

set of implicants in the same format as the input cover

CHAPTER 2

# Indices and Tables

- *genindex*
- *modindex*
- *search*

## p