#01: Working with Boolean Functions

Boolean functions are the building blocks of Boolean network models used to represent gene regulatory networks, signaling pathways, and other biological control systems. Understanding how to create and analyze individual Boolean functions is essential before studying network-level dynamics.

In this tutorial, we explore the `BooleanFunction` class — the foundation of BoolForge. Boolean functions form the regulatory rules in Boolean network models of gene regulation, so understanding their structure is essential before studying networks.

What you will learn

In this tutorial you will:

- create Boolean functions from truth tables and from textual expressions,
- inspect core attributes such as degree, variable names, and stored properties,
- compute basic structural properties (essential variables, Hamming weight, bias),
- convert Boolean functions into logical and polynomial representations,
- and interface with CANA objects.

---

0. Setup

```
import boolforge
```

---

1. Create a Boolean function

Boolean functions can be described in logical form, as polynomials, or as truth tables. BoolForge treats Boolean functions as binary vectors of length 2^n, where n is the number of inputs. The vectors describe the *right side* of the truth table. The left side of the truth table is not stored because it is the same for any function with n inputs. For example, the function

$$f(A, B) = A \wedge B$$

is stored as `[0, 0, 0, 1]`, corresponding to:

A	B	f(A,B)
0	0	0

A	B	f(A,B)
0	1	0
1	0	0
1	1	1

1.1 Create Boolean functions from a truth table

An instance of `BooleanFunction` can be generated by specifying the right side of the truth table, i.e., by providing a binary vector of length 2^n for any $n \geq 0$. For example, to create the AND function above, we can write

```
f = boolforge.BooleanFunction([0, 0, 0, 1], name="f_AND") #name is optional
print("f:", f)
print("Truth table of f:\n", f.to_truth_table().to_string())
```

```
f: [0 0 0 1]
Truth table of f:
    x0  x1  f_AND
0   0   0       0
1   0   1       0
2   1   0       0
3   1   1       1
```

Any Boolean function is stored as right side of the truth table. That is, the outputs are ordered by the binary representation of inputs:

- Position 0 → (A,B) = (0,0)
- Position 1 → (A,B) = (0,1)
- Position 2 → (A,B) = (1,0)
- Position 3 → (A,B) = (1,1)

1.2 Create Boolean functions from text

Boolean functions can also be created from textual expressions. For example, to define the same function as f, we can write

```
f2 = boolforge.BooleanFunction("A and B")
print("f2:", f2)
```

```
f2: [0 0 0 1]
```

The text processor is fairly versatile. For example, we can define the same function as f also by writing

```
f3 = boolforge.BooleanFunction("A + B > 1")
print("f3:", f3)
```

```
f3: [0 0 0 0]
```

Some examples of more complicated functions include:

```
g = boolforge.BooleanFunction("(A AND B) OR (NOT A AND C)")
h = boolforge.BooleanFunction("(x + y + z) % 2 == 0")
k = boolforge.BooleanFunction("(-1) * x + y + z > 0")

labels = ["g", "h", "k"]
boolforge.display_truth_table(g, h, k, labels=labels)
```

```
x1      x2      x3      |       g       h       k
-------------------------------------------------
0       0       0       |       0       1       0
0       0       1       |       1       0       1
0       1       0       |       0       0       1
0       1       1       |       1       0       1
1       0       0       |       0       0       0
1       0       1       |       0       0       0
1       1       0       |       1       0       0
1       1       1       |       1       0       1
```

1.3 Combining BooleanFunction objects

New Boolean functions can be constructed by combining existing ones using Boolean algebra operations. This is useful when building larger rules from simpler components.

Supported operations include:

- `~` NOT
- `&` AND
- `|` OR
- `^` XOR

```
a = boolforge.BooleanFunction("X + Y == 1")
b = boolforge.BooleanFunction("X OR Y")

not_a = ~a
a_and_b = a & b
a_or_b = a | b
a_xor_b = a ^ b

labels = ["a", "b", "NOT a", "a AND b", "a OR b", "a XOR b"]
boolforge.display_truth_table(a, b, not_a, a_and_b, a_or_b, a_xor_b, labels=labels)
```

```
x1      x2      |       a       b       NOT a   a AND b a OR b  a XOR b
-------------------------------------------------------------------------
0       0       |       0       0       1       0       0       0
0       1       |       1       1       0       1       1       0
1       0       |       1       1       0       1       1       0
1       1       |       1       1       0       1       1       0
```

---

2. Attributes of BooleanFunction

Each `BooleanFunction` instance has the following attributes:

attribute	type	description
`f`	`np.ndarray`	truth table (right side)
`n`	`int`	number of variables
`variables`	`np.ndarray`	variable names
`name`	`str`	optional name
`properties`	`dict`	cached properties

```
print("f.f:", f.f)
print("f.n:", f.n)
print("f.variables:", f.variables)
print("f.name:", f.name)
print("f.properties:", f.properties)
```

```
f.f: [0 0 0 1]
f.n: 2
f.variables: ['x0' 'x1']
f.name: f_AND
f.properties: {}
```

When a function is created from a truth table, variable names default to `x0, x1, ...`. When created from text, variable names are inferred.

```
print("f2.variables:", f2.variables)
print("f3.variables:", f3.variables)
print("g.variables:", g.variables)
print("h.variables:", h.variables)
```

```
f2.variables: ['A' 'B']
f3.variables: ['A' 'B']
g.variables: ['A' 'B' 'C']
h.variables: ['x' 'y' 'z']
```

The variable order is determined by first occurrence in the expression. See e.g.,

```
print(boolforge.BooleanFunction("(x + y + z) % 2 == 0").variables)
print(boolforge.BooleanFunction("(y + z + x) % 2 == 0").variables)
```

```
['x' 'y' 'z']
['y' 'z' 'x']
```

The variable order determines how the truth table is indexed. For example, for variables [x,y,z], the entry in position i corresponds to the binary expansion of i over (x,y,z). Therefore, the same expression with a different variable order results in a different right-side truth table ordering. This becomes important when combining functions inside networks or importing networks from text files.

---

3. Basic properties of Boolean functions

We can inspect various properties of a Boolean function. The degree, i.e., the number of inputs, is readily available via 'f.n'. Other properties can be computed.

- '.is_constant()' checks if the function is constant,
- '.is_degenerate()' checks if the function contains non-essential variables,
- '.get_essential_variables()' provides the indices (Python: starting at 0!) of the essential variables,
- '.get_type_of_inputs()' describes the type of each input ('positive', 'negative', 'conditional', or 'non-essential').
- The Hamming weight is the number of 1s in the right side of the truth table.
- The absolute bias is $|\#\text{ones} - \#\text{zeros}|/2^n$. It equals 1 for constant functions and 0 for unbiased functions.

```
print("Number of variables:", f.n)
print("Is constant?", f.is_constant())
print("Is degenerate?", f.is_degenerate())
print("Essential variables:", f.get_essential_variables())
print("Type of inputs:", f.get_type_of_inputs())
print("Hamming weight:", f.get_hamming_weight())
print("Absolute bias:", f.get_absolute_bias())
```

```
Number of variables: 2
Is constant? False
Is degenerate? False
Essential variables: [0, 1]
Type of inputs: ['positive' 'positive']
Hamming weight: 1
Absolute bias: 0.5
```

Repeating this for `g` illustrates how properties differ.

```
print("Number of variables:", g.n)
print("Is constant?", g.is_constant())
print("Is degenerate?", g.is_degenerate())
print("Essential variables:", g.get_essential_variables())
print("Type of inputs:", g.get_type_of_inputs())
print("Hamming weight:", g.get_hamming_weight())
print("Absolute bias:", g.get_absolute_bias())
```

```
Number of variables: 3
Is constant? False
Is degenerate? False
Essential variables: [0, 1, 2]
Type of inputs: ['positive' 'positive' 'conditional']
Hamming weight: 4
Absolute bias: 0.0
```

The `.summary()` method prints a human-readable overview of basic properties. If more advanced properties have already been computed, e.g., by `.get_layer_structure()` or

`get_type_of_inputs()`, they are also displayed (or if the optional keyword `COMPUTE_ALL` is set to True, default False).

```
f = boolforge.BooleanFunction("(A and B) OR NOT C")
print(f.summary())
print()

# Trigger computation of more advanced properties
print(f.summary(compute_all=True))
```

```
BooleanFunction summary
----------------------------------------
Number of variables:        3
Hamming Weight:             5
Bias:                       0.625
Absolute bias:              0.250
Variables:                  ['A' 'B' 'C']

BooleanFunction summary
----------------------------------------
Number of variables:        3
Hamming Weight:             5
Bias:                       0.625
Absolute bias:              0.250
Variables:                  ['A' 'B' 'C']
CanalizingDepth:            3
NumberOfLayers:             2
CanalizingInputs:           [0 0 0]
CanalizedOutputs:           [1 0 0]
CoreFunction:               [1]
OrderOfCanalizingVariables: [2 0 1]
LayerStructure:             [1, 2]
InputTypes:                 ['negative' 'positive' 'positive']
```

---

4. Logical and polynomial representations

While Boolean functions are stored as truth tables, they can be expressed in logical and polynomial format.

```
print(f"Logical form of {f.name}:", f.to_logical(and_op=" ∧ ", or_op=" ∨ ", not_op=" ¬
print(f"Polynomial form of {f.name}:", f.to_polynomial())
```

```
Logical form of : (( ¬C)) ∨ (A ∧ B)
Polynomial form of : (1 - A) * (1 - B) * (1 - C) + (1 - A) * B * (1 - C) + A
* (1 - B) * (1 - C) + A * B * (1 - C) + A * B * C
```

In addition, an instance of `BooleanFunction` can be turned into an instance of `BooleanNode` from the CANA package. This requires the optional CANA package to be installed.

```
cana_object = f.to_cana()
print(type(cana_object))
```

```
<class 'cana.boolean_node.BooleanNode'>
```

---

5. Summary of Key Concepts

Before moving on to more advanced topics, here is a short summary of the fundamental ideas introduced in this tutorial:

Boolean functions

A Boolean function maps a set of binary inputs (0/1) to a single binary output. BoolForge represents Boolean functions internally by their truth table, i.e., the list of outputs in lexicographic order of the input combinations.

Representations of Boolean functions

Boolean functions can be created from:

- a truth table (list of 0s and 1s),
- a logical expression written in Python syntax,
- algebraic combinations of existing BooleanFunction objects using operations such as `+` (OR), `*` (AND), `^` (XOR), and other supported Boolean operations.

Each representation produces an equivalent internal truth-table-based object.

Variable names and ordering

BoolForge automatically infers variable names from the order of first appearance in expressions. This order determines the indexing of the truth table and therefore affects how the function interacts with larger Boolean networks.

Basic properties of Boolean functions

BoolForge can compute structural properties, including:

- the number of variables (`n`),
- the Hamming weight (number of 1s in the truth table),
- absolute bias (imbalance between 0s and 1s),
- essential and non-essential variables,
- positive/negative influence of each input.

These properties help characterize the function’s behavior and are used throughout later tutorials.

Conversions and interoperability

BoolForge supports conversion between representations (e.g., truth table to/from polynomial form) and is compatible with external packages such as **CANA** for advanced analysis.
This makes it easy to move between analytical frameworks and reuse models.

Together, these concepts provide the foundation for understanding canalization, random Boolean function generation, and eventually the construction and analysis of full Boolean networks.

Frequently Asked Questions (FAQ)

1. Why does the order of variables matter?

The order in which variables appear determines the ordering of the truth table. For a function with variables `[A, B, C]`, the entry at position `i` corresponds to the binary representation of `i` over `(A, B, C)`.

If two equivalent expressions list variables in different orders, their truth tables will be indexed differently.

To ensure reproducibility, always use consistent variable names and ordering.

2. How do I choose between defining a function via a truth table or via an expression?

Short answer: It does not matter. Both methods produce identical internal representations.

Slightly longer answer: Use a **textual expression** if:

- you know the natural logical description of your function (e.g., `A and B`),
- the function is part of a Boolean network stored in some text file.

Use a **truth table** if:

- you generated the table programmatically (e.g., using `boolforge.random_function`).

3. What is the difference between `get_type_of_inputs()` and monotonicity?

The method `get_type_of_inputs()` classifies each input variable individually according to how it influences the output:

- positively increasing,
- negatively increasing,
- conditional,

- or non-essential.

Monotonicity, by contrast, is a **global property** of the Boolean function. A function is monotone if **all** essential variables influence the output in a consistent direction.

A function can therefore be non-monotone even if individual inputs have a well-defined influence type.

4. Quick Reference:

Task	Example
Create from truth table	`BooleanFunction([0, 0, 0, 1])`
Create from expression	`BooleanFunction("A and B")`
Combine with operations	`f & g, f | g, ~f, f ^ g`
Check properties	`f.is_constant()`, `f.is_degenerate()`
Get variable info	`f.variables`, `f.n`
Convert representations	`f.to_logical()`, `f.to_polynomial()`