Representing any formal boolean logical statement using Python comments

Gábor Funk

September 2026

Abstract

I propose ways and prove that any boolean logical statement can be represented using the Python's multiline string literal ''' comment and the single-line comment #. The program will also need exactly 2 print calls but that will only serve as a test to seeing the output of the construct.

Contents

1 Introduction and definitions

The 3 most important thing to define is the TRUE, FALSE and the measurement unit. TRUE is defined as a literal that can be a part of a literal enclosure:

```
'''
```

FALSE is defined as a literal that can not be a part of a literal enclosure or enclosed:

```
#'''
```

It is trivial to see that FALSE does not act. It is merely the emptiness and inaction.

The last thing is the measurement unit. The code consists of a logical statement X, then the measurement block will act as the output generator:

```
X

print("FALSE")
'''
print("TRUE")
''' # '''
```

I will denote further input statements with large alphabetic characters such as X,Y,Z,...W. Proving the correctness of the gates and ultimately my first proposition will involve checking all outputs of the measurement block.

Let us see the first base cases:

```
'''
print("FALSE")
'''
print("TRUE")
''' # '''
```

Plugging in TRUE will print TRUE. It will collapse the first print. Similarly plugging FALSY value will print the FALSE because it does not close the first branch. So the base cases are proven right.

To prove that this lexical logic system is functionally complete, we must demonstrate that it can evaluate a functionally complete set of boolean operators. We will map the Python lexer's parsing behavior to a finite state automaton and prove that it can natively compute logical **NOT** and logical **AND**. By establishing $\{AND, NOT\}$, the system is proven to be functionally complete.

Note, that the 2 print calls are solely there for getting a runtime output. They are not part of the circuit, they just demonstrate the correctness of the branching logic.

2 Lemma 1: NOT gate

A NOT gate is natively supported by this expression:

```
X
'''
```

Where X is the sole input of the NOT gate.

Seeing the correctness is trivial. Plugging TRUE:

```
'''
'''
```

Then this can be simplified to FALSE.

Plugging FALSE just keeps the TRUE expression and forwards it.

3 Lemma 2: AND gate

An AND gate can be constructed by the following expression:

```
X
'''
Y
```

Where X and Y are the 2 inputs of the AND gate.

Plugging in TRUE to both X and Y will simplify to TRUE. The other cases are somewhat non-trivial. Plugging in X=FALSE and Y=TRUE is somewhat simple. The two FALSES will cancel out and thus, we will measure FALSE. Plugging in X=TRUE and Y=FALSE will behave similarly. Plugging in FALSE as both X and Y will yield FALSE again because of a weird trick. X contributes nothing, but note that as of now the Y altough contains a `#` it will not fully cancel. It contains a string termination for fixed the multiline string literal so this will too be evaluated to FALSE.

4 Lemma 3: Composability and N-way AND

For this system to represent arbitrary boolean circuits, the gates must be composable; the evaluation of one gate must leave the lexer in a valid state to be consumed by the next gate without parity misalignment.

Because the lexical accumulator natively propagates the FALSE state, we can chain an arbitrary number of inputs into an N-way AND gate by simply interleaving inputs with NOT gates.

Let $X_1, X_2, \ldots, X_n$ be a sequence of boolean inputs. The expression for $X_1 \wedge X_2 \wedge \cdots \wedge X_n$ is constructed as:

```
X_1
'''
X_2
'''
...
'''
X_n
```

Proof. If all inputs X_i are TRUE, the sequence consists of exactly $2n-1$ single quote tokens. Because an odd number of toggles applied to Parity 0 results in Parity 1, the measurer will correctly output TRUE.

If any input X_k is FALSE, we evaluate its effect based on the preceding state. If the running parity before X_k is 1, the `#` is swallowed and the `'''` resets the parity to 0. If the parity is already 0, the `#` turns the line into a comment, maintaining Parity 0. Once the parity reaches 0 on a FALSE input, all subsequent TRUE and FALSE pairs will continuously oscillate the parity to 1 and immediately back to 0 (swallowing the `#`). Thus, the FALSE state propagates infinitely to the measurer, proving flawless composability. □

5 Functional Completeness

Theorem 1. *The sequential logic system utilizing only the Python lexical tokens* `''' and #' ' ' is functionally complete.`

Unfortunately Lemma 1,2 and 3 is not sufficient for this as the gates I proposed here are non-local, especially the NAND gate. An OR gate is yet to be found (or some mechanism which acts as an OR). If it exists, it must be composed from more than 448*4 expressions. This has been verified using exhaustive search. I tried to make an OR gate from ANDs and NOTs but the standard De Morgan identities fail at some cases but not at all. I hope someone will either prove or disprove this as I am really curious if an OR gate ever exsists or can be produced.