

logic.py: a Python Module for Logical Validation

Rob Truxler

May 6, 2011

Contents

1	Overview	2
2	Usage	3
2.1	Install	3
2.2	Propositional Logic	3
2.2.1	Grammar	3
2.2.2	logic.validate()	4
2.2.3	logic.satisfiable()	4
2.2.4	The Context object	4
2.3	First-Order Logic	5
2.3.1	Terms	5
2.3.2	Implicitly Defined Predicate Functions	6
2.3.3	Explicitly Defined Predicate Functions	6
2.3.4	Grammar with Quantifiers	7
3	Case Study: Testing Quick Sort	9
3.1	Testing Shuffle	9
3.2	Testing Quick Sort	10
4	Under the Hood	11
4.1	Parsing with PLY	11
4.2	Evaluation	11
4.3	First-order Predicate Invocations	11
5	References	12

1 Overview

Simple software verification can be achieved through use of this Python module, `logic.py`. A basic scenario is when you wish to confirm that some state is achieved after running a block of code. Typically this is done through unit tests. The trouble with unit tests is that they require much effort to set up and typically only test as many cases as the programmer's time can afford. Unit tests can be augmented with a simple model that enumerates a representative set of inputs so that a logical statement can be validated with a single call to `logic.validate()` or `Context.validate()`.

The simplest use of the logic module is to check the validity of propositional logic statements. To evaluate propositional logic statements, see `logic.validate()` or `Context.validate()`. Truth tables can also be printed with `Context.printTruthTable()`.

Unless you're testing very simple conditions and testing your logic, you probably want to use first-order logic so that you can specify sets, define relationships (between objects in a domain), and test using quantifiers (that a statement is true for at least one object in the domain, or all objects). Statements are verified by use of the `Context.validate()` function. A domain of objects can be set on the context with `Context.setTermDomain()`, which takes a list.

Predicate functions can be defined either implicitly or explicitly. Implicit predicate functions are defined through a call to `Context.assume(sentence)` or `Context.assumePredicate(predicate)`. `Context.assume(sentence)` will infer which predicate invocations must be true in order to make sentence true. `Context.assumePredicate(predicate)` will tell the logic engine that predicate is necessarily true without requiring inferencing. Using this method leads to more optimal performance in the `validate()` function. Finally Python functions can be used to define predicate functions. You can tell a Context to use a Python function by passing the function in as an argument to `Context.setPredicateFunction(function, arity)`.

2 Usage

2.1 Install

`logic.py` can be downloaded from <http://rob.truxler.com>. Once downloaded, unzip it into the same directory as your source code and add `import logic` to your code. For execution, the only necessary Python files are

- `logic.py`
- `yacc.py`
- `lex.py`

Note: `logic_examples.py` is simply a self-documenting Python module full of example usage.

2.2 Propositional Logic

Propositional logic statements can be validated using `logic.py`, meaning their truth can be easily computed and returned as a Python `bool`. Depending on the situation, the user may wish to test that a logical statement is true for all valuations of all possible atoms (ie. that the statement is a tautology), that it is true for at least one valuation (ie. that the statement is satisfiable), or in cases where the statement is not a tautology, to print out a truth table showing the valuations that make the statement false.

2.2.1 Grammar

All public functions take a string as an argument that defines the statement to be validated. This logic string must conform to the syntax specified by `logic.py`. The grammar is designed in such a way to make logical statements simple to construct and easy to read. For this reason propositional atoms are allowed to be any string that begins with a `[` and ends with a `]`.

The full grammar for propositional logic is specified by the following table:

```
<atom> ::=      [any string in these brackets]

<formula> ::=    <formula> implies <formula>
                 | <formula> therefore <formula>
                 | <formula> or <formula>
                 | <formula> and <formula>
                 | not <formula>
                 | {<formula>}
                 | <atom>
```

As with the usual precedence rules of propositional logic, the order of operations is `not`, `and`, `or`, `implies`, where `therefore` is synonymous with `implies` and is added as syntactic sugar to aid readability of logical sentences that would include the symbol \vdash .

We can translate a traditional logical statement to our more human and machine-friendly syntax:

$\neg P \rightarrow (R \wedge Q) \vdash R \vee P = \text{not } [P] \text{ implies } \{[R] \text{ and } [Q]\} \text{ therefore } [R] \text{ or } [P]$

2.2.2 `logic.validate()`

The `logic.validate()` method is used to evaluate a string that represents a logical statement specified in the above syntax. This function can be called directly from the `logic` module without instantiating any class objects.

`logic.validate()` takes a string as an argument and returns a Python `bool` that is `True` if the logical statement is a tautology (meaning all possible valuations of propositional atoms lead to the statement being true), and `False` otherwise.

2.2.3 `logic.satisfiable()`

Unlike `logic.validate()`, `logic.satisfiable()` returns `True` if there is at least one valuation of the propositional atoms in the logical statement that lead to the statement being true. This method returns a `bool`, so in order to determine details about the valuation for the atoms that led to this statement's truth, you must call `Context.printTruthTable()`.

2.2.4 The Context object

The `Context` object contains the state information that gets created in setting up or evaluating a logical statement. It also provides further control. With the `Context` object you can do perform two additional functions: printing a truth table, and specifying assumptions before calling `validate()`.

The `Context` object is created with the one line call

```
c = logic.Context()
```

At this point you can print a truth table, which will simply produce a column for each propositional atom and one for the formula that the user passes in as an argument. It will not produce columns for any sub-formulas. As an optional second argument, the `printTruthTable()` function will take a string that will serve as the column separator in the printed output. By default this separator is a tab character.

```
c = logic.Context()
c.printTruthTable("[P] and [Q] or [P] and [Q]", ",")
```

```
# This prints out the following:

# Q,P,{{{[P] and [Q]} or [P]} and [Q]}
# False,False,False
# True,False,False
# False,True,False
# True,True,True
```

The `Context` object will also accept a number of logical formulas as assumptions against which subsequent calls to `validate()` will be evaluated. For example, the statement

```
{{{[P] and [Q]} and [R]} and {[S] and [T]}} implies {[Q] and [S]}
```

can be simplified with the use of assumptions:

```

import logic
context = logic.Context()
context.assume("[P] and [Q] and [R]")
context.assume("[S] and [T]")
print str(context.validate("[Q] and [S]"))
# prints out "True"

```

With the last call to `context.validate()`, the `Context` object will reconstruct the longer sentence that includes the implication out of all of the assumptions that it knows about. In other words, all assumptions are composed in a conjunction. The formula in the `validate()` method will then be combined with an implication, and the resulting formula will be evaluated, with its truth being returned by `validate()`.

2.3 First-Order Logic

Formulas in first-order logic are also supported by `logic.py`. First-order logic is an important feature as it allows the user to test pre and post-conditions of her code using predicate functions.

Predicate functions are extensions of propositional atoms. They are like propositional atoms but they are allowed to take a finite number of arguments. These arguments are objects (strings, integers, or any object that can have a unique id) and they must belong to a user-specified domain. The objects, or terms, may be relevant to the code that is in test. For instance, if the user were to test that a sorting algorithm worked correctly over a list, he might set the domain to be all objects in the list and check to see that for all items in the domain, their relative positions in the list are based on their ordering.

In order to use first-order logical validation, there are three features that need to be understood: terms, predicate functions, and the grammar for using quantifiers.

2.3.1 Terms

As previously stated, terms are objects that can serve as an argument to a predicate function. Terms belong to a finite list that determines the pool of all possible values. This is called the domain. The domain for terms can be specified on a `Context` object with a call to `Context.setTermDomain()`.

`Context.setTermDomain()` takes just a single argument, a Python list.

For instance,

```
c.setTermDomain(["apple", "orange", "banana", "carrot"])
```

sets the domain to be four fruits and vegetables. But since the `Context` can take any Python list, it is not confined to just strings. A finite range for integers can be easily created using the Python `range()` function:

```
c.setTermDomain(range(20))
```

which sets the domain to be all integers from 0-19 inclusive.

Note that while the current version (1.0) of `logic.py` supports setting a domain with a list of any type, strange behavior can occur if the objects in the domain list are not unique when cast as strings.

2.3.2 Implicitly Defined Predicate Functions

Propositional atoms are still supported in first-order logic, but they aren't treated as predicate functions. A predicate function has the same syntax as a propositional atom, but with a parenthetical, comma-separated list of arguments:

```
[is multiple of(16, 4)]  
[reachable from(node_a, node_b)]  
[four-legged creature(giraffe)]
```

Consider the example in the previous section, where we set the term to be a list of strings that represent fruits. In this particular case we could define a predicate function whose purpose is to define the set of food items that are orange colored. One can easily think of how to write a python function to return just the strings that represent these cases (orange and carrot), but it would involve first defining a function, then writing a switch statement and returning true or false based on the value of the food item in question. This is a lot of work for something that is relatively simple. Instead we can implicitly define the values which make a predicate function true by reusing the `assume()` function:

```
context.assume("[orange colored(orange)] and [orange colored(carrot)]")
```

In just one line we have defined the terms that make the predicate “orange colored” true. While such a concise definition has its benefits, breaking this out into two lines leads to faster computation when it comes to calling `validate()`:

```
context.assumePredicate("orange colored(orange)")  
context.assumePredicate("orange colored(carrot)")
```

Here we have restated the previous assumption in two calls to `assumePredicate()`. This tells the `Context` object to remember that “orange colored” is true for an orange and a carrot. Using the regular `assume()` method requires that the `validate()` function do some extra work to infer which values in the domain must be true in order to make the assumption true.

Note that with `assumePredicate()` you omit the square brackets.

2.3.3 Explicitly Defined Predicate Functions

Sometimes it's too costly and tedious to enumerate all terms in the domain that make a particular predicate function true. This is especially the case for something simple like the predicate `is odd(x)`. Considering a possible domain of 100 values, this would require 50 assumptions to be manually entered by the user, whereas a Python function could be defined in one line that determines if `x` is odd. For this reason, there are explicitly defined predicate functions.

Any Python function can be set on the `Context` object. In order to do this, the Python function must first be defined, and its arity known. The function and its arity are passed to the `Context.setPredicateFunction()` function:

```
context = logic.Context()  
# define our predicate function, cast all arguments to ints!  
# the validator will stringify them before passing them here  
def greater_than(x, y):  
    return int(x) > int(y)
```

```

context.setTermDomain(range(10))
# tell the context about our predicate function, with arity 2
context.setPredicateFunction(greater_than, 2)

print str(context.validate("[greater_than(5,2)]")) # prints True

```

This example shows how predicate functions with a positive arity can be used to explain the relationship between two numbers. Beware of using functions with too-high of an arity. Existential and universal quantifiers can be evaluated at roughly $O(n)$ for a domain of length n and a predicate function with arity 1. If a predicate function has arity a however, this becomes $O(n^a)$, so a seemingly harmless domain defined by `c.setTermDomain(range(100))` with a predicate of arity 3 now leads to 1,000,000 invocations of that predicate function. In practice the `validate()` method isn't quite so inefficient. It will at least not call the predicate function a second time if it has already been executed with the same arguments.

2.3.4 Grammar with Quantifiers

Finally, the universal (`all x`) and existential (`exists x`) quantifiers allow to shine all of the infrastructure that we have set up. You can evaluate logical statements that check if there is a term in the domain that makes a formula true at least once (through `exists x`) or if a formula is true for all terms in the domain (with `all x`). The first-order quantifiers can be combined and ordered to express a greater number of possibilities.

The full grammar with quantifiers included is:

```

<atom> ::= [any string in these brackets]

<function> ::= [any string(comma separated list)]

<variable> ::= a single word (no spaces)

<formula> ::= <formula> implies <formula>
| <formula> therefore <formula>
| <formula> or <formula>
| <formula> and <formula>
| exists <variable> <formula>
| all <variable> <formula>
| not <formula>
| {<formula>}
| <function>
| <atom>

```

This grammar is somewhat loosely defined here, with `<function>`, `<atom>`, and `<variable>` as human readable strings instead of the conventional Backus-Naur Form.

The `exists` operator is exemplified here:

```
context = logic.Context()
```

```

# define our predicate function, cast all arguments to ints!
# the validator will stringify them before passing them here
def greater_than(x, y):
    return int(x) > int(y)

context.setTermDomain(range(10))
# tell the context about our predicate function, with arity 2
context.setPredicateFunction(greater_than,2)

print str(context.validate("exists x {[greater_than(x,5)]}")) # prints True

```

Here a domain is set up that includes all integers from 0-9 inclusive. A predicate function, `greater_than()` is explicitly defined and registered with the context. Finally in the last line we print out the validity of a statement which tests whether there is a variable, x in the domain such that $x > 5$. This statement is trivially true since the domain includes 6.

An example of the universal quantifier in use:

```

context = logic.Context()

context.assumePredicate("is a fruit(banana)")
context.assumePredicate("is a fruit(apple)")

context.setTermDomain(["banana", "apple", "carrot"])

print str(context.validate("all x {[is a fruit(x)]}")) # prints False

```

Here we set the term domain to include three food items and implicitly define a predicate, `is a fruit` that is true for only “banana” and “apple”. Finally the statement `all x {[is a fruit(x)]}` is validated and returns `False`.

3 Case Study: Testing Quick Sort

Putting it all together, we can use first-order validation to test our implementation of a couple simple list algorithms. If we wanted to test quick sort, we would first want to be able to create a list, and make sure that for unsorted lists, quick sort sorts them. To perform this test, two algorithms have to be implemented: shuffle, which takes a sorted list and randomly shuffles it, and quick sort, which of course sorts the list.

3.1 Testing Shuffle

```
def shuffle(list):
    import random
    numSwaps = len(list)
    for i in range(numSwaps):
        # perform a swap
        idx1 = random.randrange(0, len(list))
        idx2 = random.randrange(0, len(list))
        temp = list[idx1]
        list[idx1] = list[idx2]
        list[idx2] = temp
```

This algorithm is a rather simple shuffle algorithm. It takes a list with length n and determines that it will perform n swaps. For each swap it determines two indices, i_1 and i_2 that are randomly chosen from all possible indices. It then swaps the values at each of these indices. One should note that it's possible, though improbable, for this algorithm to perform n swaps and still end up with the same list. For simplicity here we'll only test that shuffle works for a single call.

To test that shuffle reorders an ordered list we can do the following:

```
c = logic.Context()
list = range(20)
c.setTermDomain(list)

def greaterThan(x, y):
    return int(x) > int(y)

def follows(x, y):
    return list.index(int(x)) == list.index(int(y))+1

c.setPredicateFunction(greaterThan, 2)
c.setPredicateFunction(follows, 2)

# perform the actual shuffle
shuffle(list) # commenting this out causes the output to be False
isTrue = c.validate("exists x {exists y {not [greaterThan(x,y)] and [follows (x,y)]")
print str(isTrue) # prints True
```

This code defines two predicate functions, `greaterThan` and `follows` which are important for testing an ordering on the list `greaterThan(x, y)` takes two arguments and returns `True` if the first is strictly

greater than the second. `follows` takes two arguments and returns `True` if the first immediately follows the second argument in the list. We can check that the shuffle algorithm worked then by asking if there is some pair of indices which defy a well-ordering. That is, are there two members of the domain, x , and y , such that $x < y$ and x immediately follows y :

Validating the statement `exists x exists y {not [greaterThan(x,y)] and [follows (x,y)]}` returns `True`, validating that shuffle worked.

3.2 Testing Quick Sort

Now that we know that shuffle works we can first shuffle the list and test that quick sort works. Instead of trying to find an x and y such that x is less than y although it follows y , here we want to verify that for all x and y , if x follows y , then it must be greater than y . Or in our syntax:

```
exists x exists y {not [greaterThan(x,y)] and [follows (x,y)]}
```

The full test environment is similar to the shuffle experiment:

```
c = logic.Context()
list = range(20)
c.setTermDomain(list)

def greaterThan(x, y):
    return int(x) > int(y)

def follows(x,y):
    return list.index(int(x)) == list.index(int(y))+1

c.setPredicateFunction(greaterThan, 2)
c.setPredicateFunction(follows, 2)

# perform the actual shuffle
shuffle(list) # commenting this out causes the output to be False
isTrue = c.validate("exists x {exists y {not [greaterThan(x,y)] and [follows (x,y)]}")
print str(isTrue) # prints True
```

and indeed, the validate function returns `True`.

4 Under the Hood

4.1 Parsing with PLY

At the heart of `logic.py` is the logical syntax parser. To perform this task, `logic.py` uses PLY (Python Lex and Yacc), a Look-Ahead LR Parser. The parser defines the grammar stated in this document and builds a parse tree for each logical statement passed to it. Each operator in the logical structure of the statement is represented by a subclass of the `BinaryOp` or `UnaryOp` classes. As such, all of these operators has a single very important function: `isTrue()` which is recursively defined based on its children. The only exception to this rule is the `Predicate` class which represents propositional atoms and predicate functions. These objects have values that are either true or false.

4.2 Evaluation

The `validate()` and `satisfiable()` methods both first parse the statement in question to produce a parse tree. With the parse tree these methods then build a truth table. If there are n propositional atoms recorded during parsing, then there are 2^n rows in the truth table. When building the truth table, the validation methods loop through the range from 0 to 2^n and alter the registered propositional atoms to hold the truth value of the n^{th} bit in the row counter. Finally once the row is initialized, the parse tree's root node is asked if it is true.

4.3 First-order Predicate Invocations

Quantifying operators have to do a little more magic to compute whether they are true or not. Before each validation, the `Context` object builds a list of all true predicate function invocations from the domain. That is, it loops through all possible ways a predicate function can be invoked (all possible n^a argument tuples for a predicate function with arity a) and if the invocation leads to a true result, a string identifier is kept so that this predicate is never invoked again until the context has changed. Finally, the `exists` and `all` operators can be evaluated during the recursive `isTrue()` call by finding a single valuation of its bound variable that makes the statement true (for `exists`) or false (for `all`, meaning that it is not true if a counter example was found).

5 References

- PLY (Python Lex-Yacc), David M. Beazley, available from <http://www.dabeaz.com/ply/>
- Aartifact: Automated Assistance for Formal Reasoning, Andrei Lapets, available from <http://aartifact.org/>