

robert.truxler



CS511 – Object Oriented Software Principles

# logic.py:

a Python Module for Logical Validation

Rob Truxler  
rtruxler [ at ] gmail dot com

May 2011



## A Python Module to ...

1. Parse and evaluate statements of *Propositional Logic*
2. Dynamic context for *assumptions* and statement revalidation
3. Evaluation of *First-Order* logical assertions



## logic.py

The module, **logic.py**, is fairly simple. It defines a number of classes that are used internally to parse and evaluate logical statements, but there are very few steps a user needs to take in order to get started:

```
import logic
print str(logic.validate("[P] or not [P]"))
# prints out "True"
```

For the simplest invocation, just call `logic.validate()`



## logic.py

For more complex invocations, create a **Context** object, and validate with the same method.

```
import logic
context = logic.Context()
print str(context.validate("[P] or not [P]"))
# prints out "True"
```

We'll see the benefits of a **Context** after discussing the propositional logic functionality...



# Propositional Logic

Propositional logic statements of the following grammar are acceptable:

$\langle \text{atom} \rangle ::= [\text{any string in these brackets}]$   
 $\langle \text{formula} \rangle ::=$  |  $\langle \text{formula} \rangle \text{ implies } \langle \text{formula} \rangle$   
|  $\langle \text{formula} \rangle \text{ therefore } \langle \text{formula} \rangle$   
|  $\langle \text{formula} \rangle \text{ and } \langle \text{formula} \rangle$   
|  $\langle \text{formula} \rangle \text{ or } \langle \text{formula} \rangle$   
|  $\text{not } \langle \text{formula} \rangle$   
|  $\{ \langle \text{formula} \rangle \}$   
|  $\langle \text{atom} \rangle$



# Propositional Logic

`<atom>` ::= [any string in these brackets]  
`<formula>` ::= | `<formula>` implies `<formula>`  
| `<formula>` therefore `<formula>`  
| `<formula>` and `<formula>`  
| `<formula>` or `<formula>`  
| not `<formula>`  
| {`<formula>`}  
| `<atom>`

- **and, or, implies**, are the binary logical operators in propositional logic
  - **therefore** is synonymous with **implies**
- **not** is the only unary logical operator in propositional logic
- **{ }** are grouping, or precedence operators



# Propositional Logic Examples

Simple truth

*[P] or not [P]*

Simple contradiction

*[P] and not [P]*

Simple implication with many atoms

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

Truth with **therefore**

*not [P] or not [Q] therefore not {[P] and [Q]}*



# Context Implementing Assumptions

The implication with many atoms

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

Could be more simply written as a **Context** with 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"
```



# Truth Tables

Truth tables can be constructed for any sentence that is validated (either with `context.printTruthTable()`, or just by passing a flag to `validate`)

```
import logic
context = logic.Context()
context.assume("[P] and [Q] and [R]")
context.assume("[S] and [T]")
context.validate("[Q] and [S]", True)
# prints out a tab-delimited truth table
# by default, but the delimiter can also be set
```



# How does it work?

- Propositional Logic formulas are validated by first building a truth table, and considering all possible truth assignments to all of the propositional atoms that are found in the sentence.
- If the last column in the truth table is all True, then the sentence is a Tautology.
- Assumptions are validated by constructing a new sentence where the conjunction of all of the assumptions implies the original sentence. This new sentence is then validated in the same way.



# First-Order Logic

First-order is supported in logic.py. Support for first-order logic expands the supported grammar in three ways:

- A domain for terms can be defined on the **context**
- Implicitly and explicitly definable predicate functions (with finite arity)
- Quantifier unary operators (“exists x <formula>” and “all x <formula>”) added



# First-Order Logic - Terms

There is so much support for quickly building lists in Python. Lists can support sequences of strings, scalars, and any other object. For this reason, they are a good candidate for internal representation of the term domain for first-order logic. The following are valid domain assignments:

```
context = logic.Context()
context.setTermDomain(range(10))
# any integer between 0-9 inclusive
context.setTermDomain(["apple", "orange", "banana"])
# resets the domain to be these three fruits
```



# Predicate Functions

Predicate functions must have a positive arity (meaning they take at least one argument). 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)]



# Implicitly Defined Predicate Functions

An **Implicitly Defined** Predicate Function is one that actually does not require a formal function definition. Instead, the values in the domain that make the predicate true are specified through assumptions:

```
context = logic.Context()
context.setTermDomain(["apple", "orange", "banana"])
# This will provide a context for truth assignments to the
# model
context.assume("[peel before eating(orange)] and [peel before eating(banana)]")
# instead of writing a full sentence, we can explicitly call
# out individual invocations of predicates that are true.
# With assumePredicate() we allow the validation
# algorithm to run more optimally
context.assumePredicate("peel before eating(orange)")
```



# Explicitly Defined Predicate Functions

Implicitly Defined Predicate Functions allow the user to quickly prototype models, but more practical use can be found from **explicitly defined** predicate functions, that is, predicate functions that get their truth evaluated by a user-definable Python 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
```



# First-Order Quantifiers: exists

**Exists**, the first-order quantifier that binds a variable to at least one term in the domain that makes a statement true is supportable in `logic.py` and can be combined with predicate functions:

```
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
```



# First-Order Quantifiers: all

**For all**, or just **“All”** represents the the first-order quantifier that binds a variable to all terms in the domain in order to satisfy a statement is also representable in logic.py:

```
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
```



# Testing Shuffle and QuickSort

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.



# Testing Shuffle

But there are various ways to shuffle lists and there is no guarantee that our shuffle algorithm is itself correct. So might start by actually testing the shuffle algorithm.

First, the algorithm:

```
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
```



# Testing Shuffle

```
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
```



# Testing Quick Sort

```
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) # we now know that this works.. so it's out of order
quicksort(list, 0, len(list)-1) # commenting this out causes the output to be False
# Verify the proper ordering
isTrue = c.validate("all x {all y {[follows(x,y)] implies [greaterThan(x,y)]}")
print str(isTrue) # prints True
```



# Limitations

1. For predicate functions, the model checking algorithm has to pre-compute truth values. This leads to  $O(n^a)$  where  $a$  is the arity of the predicate. With dynamic programming we can improve on this complexity slightly, but ideally we would only get this in the worst-case, and not always.
2. Term functions are not supported. That is, the user can't define an  $f(x) \rightarrow y$ , such that  $y$  is in the domain. This would improve usability, although technically it does not improve expressiveness. The same can be achieved with predicate functions alone, but maybe at added complexity.



# Future Work

The current implementation for model checking against statements with quantifiers does nothing to simplify the logical statement and move quantifiers to the prefix. By doing this we might isolate the quantifiers and thus define a search problem that is described by the sub-formula. With a single, isolated search problem, we can explore different iteration techniques, as opposed to linear iteration through objects in the model. Possible areas to explore are whether different iteration procedures, like those in simulated annealing, or genetic algorithms might be possible here.

Of course these types of local optimization search techniques would be difficult in a logic environment since the feedback is purely binary: true or false.



# Conclusion

The logic.py module built here provides a significant first-step in providing logical assertion testing for software written in Python. I can imagine adapting the capabilities contained in here to various programming languages to augment conventional testing strategies.

Much of the time proper software testing coverage through unit tests and integration tests is poor due to the difficulty in setting up tests. The capabilities implemented here, especially with the first-order logic quantifiers and user-definable predicates, could greatly improve the effectiveness and efficiency of test engineers' work.