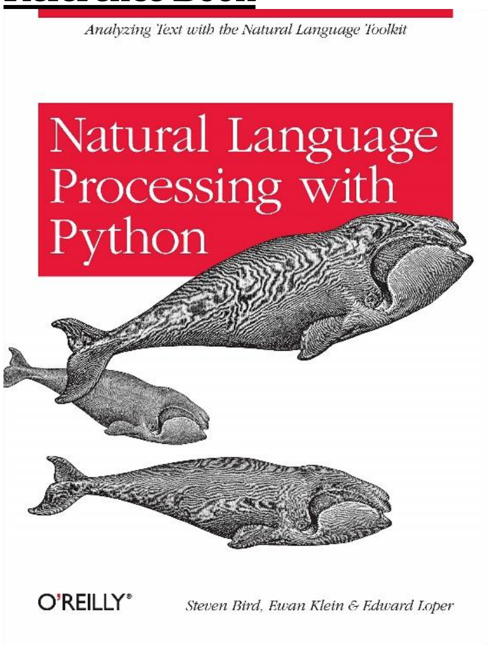
Reference Book


Analyzing Text with the Natural Language Toolkit
Natural Language Processing with Python
O'REILLY
Steven Bird, Ewan Klein & Edward Loper

V. Analyzing the Meaning of Sentences

Natural Language Understanding

- Querying a Database

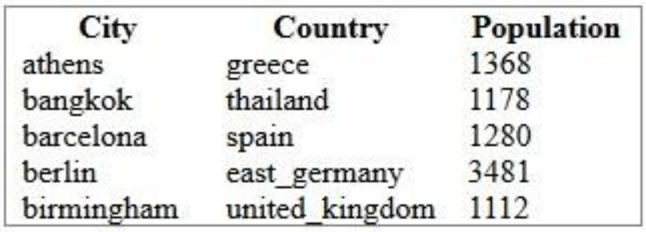
Suppose we have a program that lets us type in a natural language question and gives us back the right answer:

(1) a. Which country is Athens in?
b. Greece.

How hard is it to write such a program? To be concrete, we will use a database table whose first few rows are shown in Table VII.

The obvious way to retrieve answers from this tabular data involves writing queries in a database query language such as SQL.

Table VII. city_table.



City	Country	Population
athens	greece	1368
bangkok	thailand	1178
barcelona	spain	1280
berlin	east_germany	3481
birmingham	united_kingdom	1112

The grammar sql0.fcfg illustrates how to assemble a meaning representation for a sentence in tandem with parsing the sentence. Each phrase structure rule is supplemented with a recipe for constructing a value for the feature sem. You can see that these recipes are extremely simple; in each case, we use the string concatenation operation + to splice the values for the child constituents to make a value for the parent constituent. This allows us to parse a query into SQL.

```
import nltk
from nltk import load_parser
print(nltk.data.show_cfg('grammars/book_grammars/sql0.fcfg'))

cp = load_parser('grammars/book_grammars/sql0.fcfg')
query = 'What cities are located in China'
trees = list(cp.parse(query.split()))
answer = trees[0].label()['SEM']
answer = [s for s in answer if s]
q = ' '.join(answer)
print(q)
```

Finally, we execute the query over the database city.db and retrieve some results.

```
from nltk.sem import chat80
rows = chat80.sql_query('corpora/city_database/city.db', q)
for r in rows:
    print(r[0], end=" ")
```

To summarize, we have defined a task where the computer returns useful data in response to a natural language query, and we implemented this by translating a small subset of English into SQL. We can say that our NLTK code already "understands" SQL, given that Python is able to execute SQL queries against a database, and by extension it also "understands" queries such as What cities are located in China.

Propositional Logic

A logical language is designed to make reasoning formally explicit. As a result, it can capture aspects of natural language which determine whether a set of sentences is consistent. As part of this approach, we need to develop logical representations of a sentence φ which formally capture the truth-conditions of φ. We'll start off with a simple example:

(1) [Tom chased Mary] and [Mary ran away].

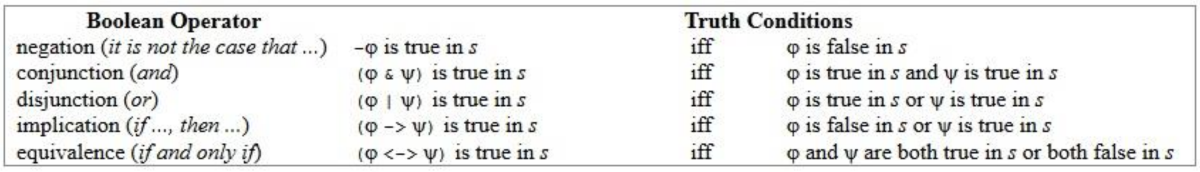
Let's replace the two sub-sentences in (1) by φ and ψ respectively, and put & for the logical operator corresponding to the English word and: φ & ψ.

Propositional logic allows us to represent just those parts of linguistic structure which correspond to certain sentential connectives. We have just looked at and. Other such connectives are not, or and if..., then.... In the formalization of propositional logic, the counterparts of such connectives are sometimes called boolean operators. The basic expressions of propositional logic are propositional symbols, often written as P, Q, R, etc. There are varying conventions for representing boolean operators. Since we will be focusing on ways of exploring logic within NLTK, we will stick to the following ASCII versions of the operators:

```
print(nltk.boolean_ops())
```

From the propositional symbols and the boolean operators we can build an infinite set of well formed formulas (or just formulas, for short) of propositional logic. First, every propositional letter is a formula. Then if φ is a formula, so is -φ. And if φ and ψ are formulas, then so are (φ & ψ) (φ | ψ) (φ -> ψ) (φ <-> ψ).

Table VIII. Truth conditions for the Boolean Operators in Propositional Logic.



Boolean Operator	Truth Conditions		
negation (*it is not the case that ...*)	$-\varphi$ is true in *s*	iff	φ is false in *s*
conjunction (*and*)	$(\varphi \,\&\, \psi)$ is true in *s*	iff	φ is true in *s* and ψ is true in *s*
disjunction (*or*)	$(\varphi \mid \psi)$ is true in *s*	iff	φ is true in *s* or ψ is true in *s*
implication (*if ..., then ...*)	$(\varphi \text{ -> } \psi)$ is true in *s*	iff	φ is false in *s* or ψ is true in *s*
equivalence (*if and only if*)	$(\varphi \text{ <-> } \psi)$ is true in *s*	iff	φ and ψ are both true in *s* or both false in *s*

First-Order Logic

In the remainder of this chapter, we will represent the meaning of natural language expressions by translating them into first-order logic. Not all of natural language semantics can be expressed in first-order logic. But it is a good choice for computational semantics because it is expressive enough to represent a good deal, and on the other hand, there are excellent systems available off the shelf for carrying out automated inference in first order logic.

- Syntax

First-order logic keeps all the boolean operators of Propositional Logic. But it adds some important new mechanisms. To start with, propositions are analyzed into predicates and arguments, which takes us a step closer to the structure of natural languages. The standard construction rules for first-order logic recognize terms such as individual variables and individual constants, and predicates which take differing numbers of arguments. For example, Angus walks might be formalized as *walk(angus)* and Angus sees Bertie as *see(angus, bertie)*. We will call walk a unary predicate, and see a binary predicate.

Consider (2) and (3). You will see that a result of (2) is semantically equivalent to (3).

(2) He disappeared.

(3) Cyril disappeared.

Corresponding to (4a), we can construct an open formula (4b) with two occurrences of the variable x. (We ignore tense to simplify exposition.)

(4) a. He is a dog and he disappeared.
b. $dog(x) \wedge disappear(x)$

By placing an existential quantifier ∃x ('for some x') in front of (4b), we can bind these variables, as in (5a), which means (5b) or, more idiomatically, (5c).

(5) a. $\exists x.(dog(x) \wedge disappear(x))$

b. At least one entity is a dog and disappeared.

c. A dog disappeared.

The NLTK rendering of (5a):

(6) exists x.(dog(x) & disappear(x))

In addition to the existential quantifier, first-order logic offers us the universal quantifier ∀x ('for all x'), illustrated in (7).

(7) a. ∀x.(dog(x) → disappear(x))

b. Everything has the property that if it is a dog, it disappears.

c. Every dog disappeared.

The NLTK syntax for (7a):

(7) all x.(dog(x) -> disappear(x))

- <u>Truth in Model</u>

We have looked at the syntax of first-order logic, we need to give a truth-conditional semantics to first-order logic.

Given a first-order logic language L, a model M for L is a pair ⟨D, Val⟩, where D is an nonempty set called the domain of the model, and Val is a function called the valuation function which assigns values from D to expressions of L.

Relations are represented semantically in NLTK in the standard set-theoretic way: as sets of tuples. We will use the utility function Valuation.fromstring() to convert a list of strings of the form symbol => value into a Valuation object.

```
import nltk
v = """
    bertie => b
    olive => o
    cyril => c
    boy => {b}
    girl => {o}
    dog => {c}
    walk => {o, c}
    see => {(b, o), (c, b), (o, c)}"""
val = nltk.Valuation.fromstring(v)
print(val)
```

So according to this valuation, the value of see is a set of tuples such that Bertie sees Olive, Cyril sees Bertie, and Olive sees Cyril.

```
print(('o', 'c') in val['see'])
print(('b',) in val['boy'])
```

- <u>Individual Variables and Assignments</u>

In our models, the counterpart of a context of use is a variable assignment. This is a mapping from individual variables to entities in the domain. Assignments are created using the Assignment constructor, which also takes the model's domain of discourse as a parameter. We are not required to actually enter any bindings, but if we do, they are in a (variable, value) format similar to what we saw earlier for valuations.

```
dom = {'b', 'o', 'c'}
g = nltk.Assignment(dom, [('x', 'o'), ('y', 'c')])
print(g)
```

Let's now look at how we can evaluate an atomic formula of first-order logic. First, we create a model, then we call the evaluate() method to compute the truth value.

```
m = nltk.Model(dom, val)
print(m.evaluate('see(olive, y)', g))
```

What's happening here? We are evaluating a formula which is similar to our earlier examplle, see(olive, cyril). However, when the interpretation function encounters the variable y, rather than checking for a value in val, it asks the variable assignment g to come up with a value:

```
print(g['y'])
```

Since we already know that individuals o and c stand in the see relation, the value True is what we expected. In this case, we can say that assignment g satisfies the formula see(olive, y). By contrast, the following formula evaluates to False relative to g — check that you see why this is.

```
print(m.evaluate('see(y, x)', g))
```

In our approach (though not in standard first-order logic), variable assignments are partial. For example, g says nothing about any variables apart from x and y. The method purge() clears all bindings from an assignment.

```
print(g.purge())
```

If we now try to evaluate a formula such as see(olive, y) relative to g, it is like trying to interpret a sentence containing a him when we don't know what him refers to. In this case, the evaluation function fails to deliver a truth value.

```
print(m.evaluate('see(olive, y)', g))
```

Since our models already contain rules for interpreting boolean operators, arbitrarily complex formulas can be composed and evaluated.

```
print(m.evaluate('see(bertie, olive) & boy(bertie)
& -walk(bertie)', g))
```

- Quantification

One of the crucial insights of modern logic is that the notion of variable satisfaction can be used to provide an interpretation to quantified formulas. Let's use (8) as an example.

(8) exists x.(girl(x) & walk(x))

When is it true? Let's think about all the individuals in our domain, i.e., in dom. We want to check whether any of these individuals have the property of being a girl and walking.
Consider the following:

```
print(m.evaluate('exists x.(girl(x) & walk(x))',
g))
```

One useful tool offered by NLTK is the satisfiers() method. This returns a set of all the individuals that satisfy an open formula. The method parameters are a parsed formula, a variable, and an assignment. Here are a few examples:

```
#NLTKs Expression object can process logical
expressions into various subclasses of Expression
read_expr = nltk.sem.Expression.fromstring
fmla1 = read_expr('girl(x) | boy(x)')
print(m.satisfiers(fmla1, 'x', g))
fmla2 = read_expr('girl(x) -> walk(x)')
print(m.satisfiers(fmla2, 'x', g))
fmla3 = read_expr('walk(x) -> girl(x)')
print(m.satisfiers(fmla3, 'x', g))
```

The Semantics of English Sentences

NLTK provides some utilities to make it easier to derive and inspect semantic interpretations. The function interpret_sents() is intended for interpretation of a list of input sentences. It builds a dictionary d where for each sentence sent in the input, d[sent] is a list of pairs (*synrep*, *semrep*) consisting of trees and semantic representations for sent. The value is a list since sent may be syntactically ambiguous; in the following example, however, there is only one parse tree per sentence in the list.

```
sents = ['Irene walks', 'Cyril bites an ankle']
grammar_file = 'grammars/book_grammars/simple-
sem.fcfg'
for results in nltk.interpret_sents(sents,
grammar_file):
    for (synrep, semrep) in results:
        print(synrep)
```

We have seen now how to convert English sentences into logical forms, and earlier we saw how logical forms could be checked as true or false in a model. Putting these two mappings together, we can check the truth value of English sentences in a given model. Let's take model m as defined above. The utility evaluate_sents() resembles interpret_sents() except that we need to pass a model and a variable assignment as parameters. The output is a triple (*synrep*, *semrep*, *value*) where *synrep*, *semrep* are as before, and *value* is a truth value. For simplicity, the following example only processes a single sentence.

```
v = """
    bertie => b
    olive => o
    cyril => c
    boy => {b}
    girl => {o}
    dog => {c}
    walk => {o, c}
    see => {(b, o), (c, b), (o, c)} """
val = nltk.Valuation.fromstring(v)
g = nltk.Assignment(val.domain)
m = nltk.Model(val.domain, val)
sent = 'Cyril sees every boy'
grammar_file = 'grammars/book_grammars/simple-
sem.fcfg'
results = nltk.evaluate_sents([sent],
grammar_file, m, g)[0]
for (syntree, semrep, value) in results:
    print(semrep)
    print(value)
```