# Preface: Learning NLP with the Natural Language Toolkit

**Teaching NLP:** Natural Language Processing (NLP) is often taught within the confines of a single-semester course, either at advanced undergraduate level, or at postgraduate level. Unfortunately, it turns out to be rather difficult to cover both the theoretical and practical sides of the subject in such a short span of time. Some courses focus on theory to the exclusion of practical exercises, and deprive students of the challenge and excitement of writing programs to automatically process natural language. Other courses are simply designed to teach programming for linguists, and do not manage to cover any significant NLP content. NLTK was developed to address this very problem, making it feasible to cover a substantial amount of theory and practice within a single-semester course.

A significant fraction of any NLP course is made up of fundamental data structures and algorithms. These are usually taught with the help of formal notations and complex diagrams. Large trees and charts are copied onto the board and edited in tedious slow motion, or laboriously prepared for presentation slides. A more effective method is to use live demonstrations in which those diagrams are generated and updated automatically. NLTK provides interactive graphical user interfaces, making it possible to view program state and to study program execution step-by-step. Most NLTK components have a demonstration mode, and will perform an interesting task without requiring any special input from the user. It is even possible to make minor modifications to programs in response to "what if" questions. In this way, students learn the mechanics of NLP quickly, gain deeper insights into the data structures and algorithms, and acquire new problem-solving skills.

NLTK supports assignments of varying difficulty and scope. In the simplest assignments, students experiment with existing components to perform a wide variety of NLP tasks. This may involve no programming at all, in the case of the existing demonstrations, or simply changing a line or two of program code. As students become more familiar with the toolkit they can be asked to modify existing components or to create complete systems out of existing components. NLTK also provides students with a flexible framework for advanced projects, such as developing a multi-component system, by integrating and extending NLTK components, and adding on entirely new components. Here NLTK helps by providing standard implementations of all the basic data structures and algorithms, interfaces to standard corpora, substantial corpus samples, and a flexible and extensible architecture. Thus, as we have seen, NLTK offers a fresh approach to NLP pedagogy, in which theoretical content is tightly integrated with application.

The *Natural Language Toolkit (NLTK)* was originally created as part of a computational linguistics course in the Department of Computer and Information Science at the University of Pennsylvania in 2001. Since then it has been developed and expanded with the help of dozens of contributors. It has now been adopted in courses in dozens of universities, and serves as the basis of many research projects. In this section we will discuss some of the benefits of learning (and teaching) NLP using NLTK.

**Note on NLTK-Lite:** Recently, the NLTK developers have been creating a lightweight version NLTK, called NLTK-Lite. NLTK-Lite is simpler and faster than NLTK. Once it is complete, NLTK-Lite will provide all the same functionality as NLTK. However, unlike NLTK, NLTK-Lite does not impose such a heavy burden on the programmer. Wherever possible, standard Python objects are used

instead of custom NLP versions, so that students learning to program for the first time will be learning to program in Python with some useful libraries, rather than learning to program in NLTK.

**The Design of NLTK:** NLTK was designed with six requirements in mind:

1. *Ease of use:* The primary purpose of the toolkit is to allow students to concentrate on building natural language processing systems. The more time students must spend learning to use the toolkit, the less useful it is. We have provided software distributions for several platforms, along with platform-specific instructions, to make the toolkit easy to install.

2. *Consistency:* We have made a significant effort to ensure that all the data structures and interfaces are consistent, making it easy to carry out a variety of tasks using a uniform framework.

3. *Extensibility:* The toolkit easily accommodates new components, whether those components replicate or extend existing functionality. Moreover, the toolkit is organized so that it is usually obvious where extensions would fit into the toolkit's infrastructure.

4. *Simplicity:* We have tried to provide an intuitive and appealing framework along with substantial building blocks, for students to gain a practical knowledge of NLP without getting bogged down in the tedious house-keeping usually associated with processing annotated language data.

5. *Modularity:* The interaction between different components of the toolkit is minimized, and uses simple, well-defined interfaces. It is possible to complete individual projects using small parts of the toolkit, without needing to understand how they interact with the rest of the toolkit. This allows students to learn how to use the toolkit incrementally throughout a course. Modularity also makes it easier to change and extend the toolkit.

6. *Well-Documented:* The toolkit comes with substantial documentation, including nomenclature, data structures, and implementations.

Contrasting with these requirements are three non-requirements, potentially useful features that we have deliberately avoided. First, while the toolkit provides a wide range of functions, it is not intended to be encyclopedic. There should be a wide variety of ways in which students can extend the toolkit. Second, while the toolkit should be efficient enough that students can use their NLP systems to perform meaningful tasks, it does not need to be highly optimized for runtime performance. Such optimizations often involve more complex algorithms, and sometimes require the use of C or C++, making the toolkit less accessible, and harder to install. Third, we have avoided clever programming tricks, since clear implementations are far preferable to ingenious yet indecipherable ones.

**NLTK Organization:** NLTK is organized into a collection of task-specific components. Each module is a combination of data structures for representing a particular kind of information such as trees, and implementations of standard algorithms involving those structures such as parsers. This approach is a standard feature of *object-oriented design*, in which components encapsulate both the resources and methods needed to accomplish a particular task.

The most fundamental NLTK components are for identifying and manipulating individual words of text. These include: **`tokenize`**, for breaking up strings of characters into word tokens; **`tag`**, for adding part-of-speech tags, including regular-expression taggers, n-gram taggers and Brill taggers; and the Porter stemmer.

The second kind of module is for creating and manipulating structured linguistic information. These components include: **`tree`**, for representing and processing parse trees; **`featurestructure`**, for

building and unifying nested feature structures (or attribute-value matrices); **`cfg`**, for specifying free grammars; and **`parse`**, for creating parse trees over input text, including chart parsers, chunk parsers and probabilistic parsers.

Several utility components are provided to facilitate processing and visualization. These include: **`draw`**, to visualize NLP structures and processes; **`probability`**, to count and collate events, and perform statistical estimation; and **`corpora`**, to access tagged linguistic corpora.

A further group of components is not part of NLTK proper. These are a wide selection of third-party contributions, often developed as student projects at various institutions where NLTK is used, and distributed in a separate package called *NLTK Contrib*. Several of these student contributions, such as the Brill tagger and the HMM module, have now been incorporated into NLTK. Although these components are not maintained, they may serve as a useful starting point for future student projects. In general, they do not work with the current version of NLTK.

In addition to software and documentation, NLTK provides substantial corpus samples, listed below. Many of these can be accessed using the **`corpora`** module, avoiding the need to write specialized file parsing code before you can do NLP tasks.

| **Corpora and Corpus Samples Distributed with NLTK (starred items with NLTK-Lite)** | | |
|---|---|---|
| **Corpus** | **Compiler** | **Contents** |
| Brown Corpus | Francis, Kucera | 15 genres, 1.15M words, tagged |
| CoNLL 2000 Chunking Data | Tjong Kim Sang | 270k words, tagged and chunked |
| Genesis Corpus | Misc web sources | 6 texts, 200k words, 6 languages |
| Project Gutenberg (sel) | Hart, Newby, et al | 14 texts, 1.7M words |
| NIST 1999 Info Extr (sel) | Garofolo | 63k words, newswire and named-entity SGML markup |
| Lexicon Corpus | | Words, tags and frequencies from Brown Corpus and WSJ |
| Names Corpus | Kantrowitz, Ross | 8k male and female names |
| PP Attachment Corpus | Ratnaparkhi | 28k prepositional phrases, tagged as noun or verb modifiers |
| Roget's Thesaurus | Project Gutenberg | 200k words, formatted text |
| SEMCOR | Rus, Mihalcea | 880k words, part-of-speech and sense tagged |
| SENSEVAL 2 Corpus | Ted Pedersen | 600k words, part-of-speech and sense tagged |
| Stopwords Corpus | Porter et al | 2,400 stopwords for 11 languages |
| Penn Treebank (sel) | LDC | 40k words, tagged and parsed |
| TIMIT Corpus (sel) | NIST/LDC | audio files and transcripts for 16 speakers |
| Wordlist Corpus | OpenOffice.org et al | 960k words and 20k affixes for 8 languages |

**NLTK Website:** All software, corpora, and documentation are freely downloadable from **`http://nltk.sourcefo`** Distributions are provided for Windows, Macintosh and Unix platforms. An ISO CD-ROM image, containing all NLTK distributions, plus Python and WordNet distributions, is also downloadable.

**Relationship to Other NLP Textbooks:** A variety of excellent NLP textbooks are available. What sets these materials apart from the others is the tight coupling of the chapters and exercises

with a toolkit, giving students -- even those with no prior programming experience -- a practical introduction to NLP. Once completing these materials, students will be ready to attempt the more advanced textbook *Foundations of Statistical Natural Language Processing*, by Manning and Schütze (MIT Press, 2000). Two other recent textbooks cover NLP together with speech processing: *Speech and Language Processing*, by Jurafsky and Martin (Prentice Hall, 2000), and *Introducing Speech and Language Processing* by Coleman (Cambridge, 2005). While impressive for their coverage, neither provides a uniform computational framework so important for newcomers to NLP. Hammond's book *Programming for Linguists: Perl for Language Researchers*, (Blackwell, 2003) and a Java version, cover elementary programming but do not address NLP. There are many older textbooks, which opened the field to earlier generations of students; these are mostly of historical interest: *Natural Language Understanding* (Allen, Addison Wesley, 1995); *Statistical Language Learning* (Charniak, MIT Press, 1993); *Natural Language Processing for Prolog Programmers* (Covington, 1993); *Natural Language Processing in Prolog* (Gazdar and Mellish, Addison Wesley, 1989) *Prolog and Natural-Language Analysis* (Pereira and Shieber, CSLI, 1987) *Computational Linguistics* (Grishman, Cambridge, 1986).

**NLP in Python vs other Programming Languages:** Many programming languages have been used for NLP. As we will explain in more detail in the introductory chapter, we have chosen Python because we believe it is well-suited to the special requirements of NLP. Here we present a brief survey of several programming languages, for the simple task of reading a text and printing the words that end with **`ing`**. We begin with the Python version, which we believe is readily interpretable, even by non Python programmers:

```
import sys
for line in sys.stdin.readlines():
    for word in line.split():
        if word.endswith('ing'):
            print word
```

Like Python, Perl is a scripting language. However, its syntax is obscure. For instance, it is difficult to guess what kind of entities are represented by: **`<>`**, **`$`**, **`my`**, and **`split`**, in the following program:

```
while (<>) {
    foreach my $word (split) {
        if ($word =~ /ing$/) {
            print "$word\n";
        }
    }
}
```

We agree that "it is quite easy in Perl to write programs that simply look like raving gibberish, even to experienced Perl programmers" (Hammond 2003:47). Having used Perl ourselves in research and teaching since the 1980s, we have found that Perl programs of any size are inordinately difficult to maintain and re-use. Therefore we believe Perl is not an optimal choice of programming language for linguists or for language processing.

Prolog is a logic programming language which has been popular for developing natural language parsers and feature-based grammars, given the inbuilt support for search and the *unification* operation which combines two feature structures into one. Unfortunately Prolog is not easy to use for string processing or input/output, as the following program code demonstrates:

```
main :-
    current_input(InputStream),
```

```
    read_stream_to_codes(InputStream, Codes),
    codesToWords(Codes, Words),
    maplist(string_to_list, Words, Strings),
    filter(endsWithIng, Strings, MatchingStrings),
    writeMany(MatchingStrings),
    halt.

codesToWords([], []).
codesToWords([Head | Tail], Words) :-
    ( char_type(Head, space) ->
        codesToWords(Tail, Words)
    ;
        getWord([Head | Tail], Word, Rest),
        codesToWords(Rest, Words0),
        Words = [Word | Words0]
    ).

getWord([], [], []).
getWord([Head | Tail], Word, Rest) :-
    (
        ( char_type(Head, space) ; char_type(Head, punct) )
    ->  Word = [], Tail = Rest
    ;   getWord(Tail, Word0, Rest), Word = [Head | Word0]
    ).

filter(Predicate, List0, List) :-
    ( List0 = [] -> List = []
    ;   List0 = [Head | Tail],
        ( apply(Predicate, [Head]) ->
            filter(Predicate, Tail, List1),
            List = [Head | List1]
        ;   filter(Predicate, Tail, List)
        )
    ).

endsWithIng(String) :- sub_string(String, _Start, _Len, 0, 'ing').

writeMany([]).
writeMany([Head | Tail]) :- write(Head), nl, writeMany(Tail).
```

Java is an object-oriented language incorporating native support for the internet, that was originally designed to permit the same executable program to be run on most computer platforms. Java has replaced COBOL as the standard language for business enterprise software:

```
import java.io.*;
public class IngWords {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
          InputStreamReader(
                 System.in));
        String line = in.readLine();
        while (line != null) {
            for (String word : line.split(" ")) {
```

```
                if (word.endsWith("ing"))
                    System.out.println(word);
            }
            line = in.readLine();
        }
    }
}
```

The C programming language is a highly-efficient low-level language that is popular for operating system and networking software:

```
#include <sys/types.h>
#include <regex.h>
#include <stdio.h>
#define BUFFER_SIZE 1024

int main(int argc, char **argv) {
     regex_t space_pat, ing_pat;
     char buffer[BUFFER_SIZE];
     regcomp(&space_pat, "[, \t\n]+", REG_EXTENDED);
     regcomp(&ing_pat, "ing$", REG_EXTENDED | REG_ICASE);

     while (fgets(buffer, BUFFER_SIZE, stdin) != NULL) {
         char *start = buffer;
         regmatch_t space_match;
         while (regexec(&space_pat, start, 1, &space_match, 0) == 0) {
             if (space_match.rm_so > 0) {
                 regmatch_t ing_match;
                 start[space_match.rm_so] = '\0';
                 if (regexec(&ing_pat, start, 1, &ing_match, 0) == 0)
                     printf("%s\n", start);
             }
             start += space_match.rm_eo;
         }
     }
     regfree(&space_pat);
     regfree(&ing_pat);

     return 0;
}
```

LISP is a so-called functional programming language, in which all objects are lists, and all operations are performed by (nested) functions of the form **`(function arg1 arg2 ...)`**. Many of the earliest NLP systems were implemented in LISP:

```
(defpackage "REGEXP-TEST" (:use "LISP" "REGEXP"))
(in-package "REGEXP-TEST")

(defun has-suffix (string suffix)
  "Open a file and look for words ending in _ing."
  (with-open-file (f string)
     (with-loop-split (s f " ")
        (mapcar #'(lambda (x) (has_suffix suffix x)) s))))
```

```
(defun has_suffix (suffix string)
  (let* ((suffix_len (length suffix))
   (string_len (length string))
    (base_len (- string_len suffix_len)))
    (if (string-equal suffix string :start1 0 :end1 NIL :start2 base_len :end2 NIL)
        (print string))))

(has-suffix "test.txt" "ing")
```

Haskell is another functional programming language which permits a much more compact solution of our simple task:

```
module Main
  where main = interact (unlines.(filter ing).(map (filter isAlpha)).words)
    where ing = (=="gni").(take 3).reverse
```

(We are grateful to the following people for furnishing us with these program samples: Tim Baldwin, Trevor Cohn, Rod Farmer, Edward Ivanovic, Olivia March, and Lars Yencken.)

---

**About the Authors:**

| | | | |
|---|---|---|---|
|  |  |  |  |
| Steven Bird | James Curran | Ewan Klein | Edward Loper |

**Steven Bird** is an Associate Professor in the Department of Computer Science and Software Engineering at the University of Melbourne, and a Senior Research Associate in the Linguistic Data Consortium at the University of Pennsylvania. After completing a PhD at the University of Edinburgh on computational phonology (1990), Steven moved to Cameroon to conduct fieldwork on tone and orthography. Later he spent four years as Associate Director of the Linguistic Data Consortium where he developed models and tools for linguistic annotation. His current research interests are in linguistic databases and query languages.

**James Curran** is a Postdoctoral Fellow in the School of Information Technologies at the University of Sydney. ...

**Ewan Klein** is a Professor in the School of Informatics at the University of Edinburgh. ...

**Edward Loper** is a doctoral student in the Department of Computer and Information Sciences at the University of Pennsylvania. ...