NLTK Documentation

Release 3.0

Steven Bird

October 10, 2015

Contents

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.

NLTK has been called "a wonderful tool for teaching, and working in, computational linguistics using Python," and "an amazing library to play with natural language."

Natural Language Processing with Python provides a practical introduction to programming for language processing. Written by the creators of NLTK, it guides the reader through the fundamentals of writing Python programs, working with corpora, categorizing text, analyzing linguistic structure, and more. The book is being updated for Python 3 and NLTK 3. (The original Python 2 version is still available at http://nltk.org/book_1ed.)

CHAPTER 1

Some simple things you can do with NLTK

Tokenize and tag some text:

```
>>> import nltk
>>> sentence = """At eight o'clock on Thursday morning
... Arthur didn't feel very good."""
>>> tokens = nltk.word_tokenize(sentence)
>>> tokens
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning',
'Arthur', 'did', "n't", 'feel', 'very', 'good', '.']
>>> tagged = nltk.pos_tag(tokens)
>>> tagged[0:6]
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

Identify named entities:

```
>>> entities = nltk.chunk.ne_chunk(tagged)
>>> entities
Tree('S', [('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'),
           ('on', 'IN'), ('Thursday', 'NNP'), ('morning', 'NN'),
       Tree('PERSON', [('Arthur', 'NNP')]),
           ('did', 'VBD'), ("n't", 'RB'), ('feel', 'VB'),
           ('very', 'RB'), ('good', 'JJ'), ('.', '.')])
```

Display a parse tree:

```
>>> from nltk.corpus import treebank
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> t.draw()
```

NB. If you publish work that uses NLTK, please cite the NLTK book as follows:

> Bird, Steven, Edward Loper and Ewan Klein (2009), *Natural Language Processing with Python*. O'Reilly Media Inc.

CHAPTER 2

Next Steps

- sign up for release announcements
- join in the discussion

CHAPTER 3

Contents

3.1 NLTK News

3.1.1 2015

NLTK 3.0.5 released [September 2015] New Twitter package; updates to IBM models 1-3, new models 4 and 5, minor bugfixes and enhancements For details see: https://github.com/nltk/nltk/blob/develop/ChangeLog

NLTK 3.0.4 released [July 2015] Minor bugfixes and enhancements.

NLTK 3.0.3 released [June 2015] PanLex Swadesh Corpus, tgrep tree search, minor bugfixes.

NLTK 3.0.2 released [March 2015] Senna, BLLIP, python-crfsuite interfaces, transition-based dependency parsers, dependency graph visualization, NKJP corpus reader, minor bugfixes and clean-ups.

NLTK 3.0.1 released [January 2015] Minor packaging update.

3.1.2 2014

NLTK 3.0.0 released [September 2014] Minor bugfixes.

NLTK 3.0.0b2 released [August 2014] Minor bugfixes and clean-ups.

NLTK Book Updates [July 2014] The NLTK book is being updated for Python 3 and NLTK 3 here. The original Python 2 edition is still available here.

NLTK 3.0.0b1 released [July 2014] FrameNet, SentiWordNet, universal tagset, misc efficiency improvements and bugfixes Several API changes, see https://github.com/nltk/nltk/wiki/Porting-your-code-to-NLTK-3.0

NLTK 3.0a4 released [June 2014] FrameNet, universal tagset, misc efficiency improvements and bugfixes Several API changes, see https://github.com/nltk/nltk/wiki/Porting-your-code-to-NLTK-3.0 For full details see: https://github.com/nltk/nltk/blob/develop/ChangeLog http://nltk.org/nltk3-alpha/

3.1.3 2013

NLTK Book Updates [October 2013] We are updating the NLTK book for Python 3 and NLTK 3; please see http://nltk.org/book3/

NLTK 3.0a2 released [July 2013] Misc efficiency improvements and bugfixes; for details see https://github.com/nltk/nltk/blob/develop/ChangeLog http://nltk.org/nltk3-alpha/

NLTK 3.0a1 released [February 2013] This version adds support for NLTK's graphical user interfaces. http://nltk.org/nltk3-alpha/

NLTK 3.0a0 released [January 2013] The first alpha release of NLTK 3.0 is now available for testing. This version of NLTK works with Python 2.6, 2.7, and Python 3. http://nltk.org/nltk3-alpha/

3.1.4 2012

Python Grant [November 2012] The Python Software Foundation is sponsoring Mikhail Korborov's work on porting NLTK to Python 3. http://pyfound.blogspot.hu/2012/11/grants-to-assist-kivy-nltk-in-porting.html

NLTK 2.0.4 released [November 2012] Minor fix to remove numpy dependency.

NLTK 2.0.3 released [September 2012] This release contains minor improvements and bugfixes. This is the final release compatible with Python 2.5. For details see https://github.com/nltk/nltk/blob/develop/ChangeLog

NLTK 2.0.2 released [July 2012] This release contains minor improvements and bugfixes. For details see https://github.com/nltk/nltk/blob/develop/ChangeLog

NLTK 2.0.1 released [May 2012] The final release of NLTK 2. For details see https://github.com/nltk/nltk/blob/develop/ChangeLog

NLTK 2.0.1rc4 released [February 2012] The fourth release candidate for NLTK 2.

NLTK 2.0.1rc3 released [January 2012] The third release candidate for NLTK 2.

3.1.5 2011

NLTK 2.0.1rc2 released [December 2011] The second release candidate for NLTK 2. For full details see the ChangeLog.

NLTK development moved to GitHub [October 2011] The development site for NLTK has moved from Google-Code to GitHub: http://github.com/nltk

NLTK 2.0.1rc1 released [April 2011] The first release candidate for NLTK 2. For full details see the ChangeLog.

3.1.6 2010

Python Text Processing with NLTK 2.0 Cookbook [December 2010] Jacob Perkins has written a 250-page cookbook full of great recipes for text processing using Python and NLTK, published by Packt Publishing. Some of the royalties are being donated to the NLTK project.

Japanese translation of NLTK book [November 2010] Masato Hagiwara has translated the NLTK book into Japanese, along with an extra chapter on particular issues with Japanese language process. See http://www.oreilly.co.jp/books/9784873114705/.

NLTK 2.0b9 released [July 2010] The last beta release before 2.0 final. For full details see the ChangeLog.

NLTK in Ubuntu 10.4 (Lucid Lynx) [February 2010] NLTK is now in the latest LTS version of Ubuntu, thanks to the efforts of Robin Munn. See http://packages.ubuntu.com/lucid/python/python-nltk

NLTK 2.0b? released [June 2009 - February 2010] Bugfix releases in preparation for 2.0 final. For full details see the ChangeLog.

3.1.7 2009

NLTK Book in second printing [December 2009] The second print run of Natural Language Processing with Python will go on sale in January. We've taken the opportunity to make about 40 minor corrections. The online version has been updated.

NLTK Book published [June 2009] Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper, has been published by O'Reilly Media Inc. It can be purchased in hardcopy, ebook, PDF or for online access, at http://oreilly.com/catalog/9780596516499/. For information about sellers and prices, see https://isbndb.com/d/book/natural_language_processing_with_python/prices.html.

Version 0.9.9 released [May 2009] This version finalizes NLTK's API ahead of the 2.0 release and the publication of the NLTK book. There have been dozens of minor enhancements and bugfixes. Many names of the form nltk.foo.Bar are now available as nltk.Bar. There is expanded functionality in the decision tree, collocations, and Toolbox modules. A new translation toy nltk.misc.babelfish has been added. A new module nltk.help gives access to tagset documentation. Fixed imports so NLTK will build and install without Tkinter (for running on servers). New data includes a maximum entropy chunker model and updated grammars. NLTK Contrib includes updates to the coreference package (Joseph Frazee) and the ISRI Arabic stemmer (Hosam Algasaier). The book has undergone substantial editorial corrections ahead of final publication. For full details see the ChangeLog.

Version 0.9.8 released [February 2009] This version contains a new off-the-shelf tokenizer, POS tagger, and named-entity tagger. A new metrics package includes inter-annotator agreement scores and various distance and word association measures (Tom Lippincott and Joel Nothman). There's a new collocations package (Joel Nothman). There are many improvements to the WordNet package and browser (Steven Bethard, Jordan Boyd-Graber, Paul Bone), and to the semantics and inference packages (Dan Garrette). The NLTK corpus collection now includes the PE08 Parser Evaluation data, and the CoNLL 2007 Basque and Catalan Dependency Treebanks. We have added an interface for dependency treebanks. Many chapters of the book have been revised in response to feedback from readers. For full details see the ChangeLog. NB some method names have been changed for consistency and simplicity. Use of old names will generate deprecation warnings that indicate the correct name to use.

3.1.8 2008

Version 0.9.7 released [December 2008] This version contains fixes to the corpus downloader (see instructions) enabling NLTK corpora to be released independently of the software, and to be stored in compressed format. There are improvements in the grammars, chart parsers, probability distributions, sentence segmenter, text classifiers and RTE classifier. There are many further improvements to the book. For full details see the ChangeLog.

Version 0.9.6 released [December 2008] This version has an incremental corpus downloader (see instructions) enabling NLTK corpora to be released independently of the software. A new WordNet interface has been developed by Steven Bethard (details). NLTK now has support for dependency parsing, developed by Jason Narad (sponsored by Google Summer of Code). There are many enhancements to the semantics and inference packages, contributed by Dan Garrette. The frequency distribution classes have new support for tabulation and plotting. The Brown Corpus reader has human readable category labels instead of letters. A new Swadesh Corpus containing comparative wordlists has been added. NLTK-Contrib includes a TIGERSearch implementation for searching treebanks (Torsten Marek). Most chapters of the book have been substantially revised.

The NLTK Project has moved [November 2008] The NLTK project has moved to Google Sites, Google Code and Google Groups. Content for users and the nltk.org domain is hosted on Google Sites. The home of NLTK development is now Google Code. All discussion lists are at Google Groups. Our old site at nltk.sourceforge.net will continue to be available while we complete this transition. Old releases are still available via our SourceForge release page. We're grateful to SourceForge for hosting our project since its inception in 2001.

Version 0.9.5 released [August 2008] This version contains several low-level changes to facilitate installation, plus updates to several NLTK-Contrib projects. A new text module gives easy access to text corpora for newcomers to NLP. For full details see the ChangeLog.

Version 0.9.4 released [August 2008] This version contains a substantially expanded semantics package contributed by Dan Garrette, improvements to the chunk, tag, wordnet, tree and feature-structure modules, Mallet interface, ngram language modeling, new GUI tools (WordNet? browser, chunking, POS-concordance). The data distribution includes the new NPS Chat Corpus. NLTK-Contrib includes the following new packages (still undergoing active development) NLG package (Petro Verkhogliad), dependency parsers (Jason Narad), coreference (Joseph Frazee), CCG parser (Graeme Gange), and a first order resolution theorem prover (Dan Garrette). For full details see the ChangeLog.

NLTK presented at ACL conference [June 2008] A paper on teaching courses using NLTK will be presented at the ACL conference: Multidisciplinary Instruction with the Natural Language Toolkit

Version 0.9.3 released [June 2008] This version contains an improved WordNet? similarity module using pre-built information content files (included in the corpus distribution), new/improved interfaces to Weka, MEGAM and Prover9/Mace4 toolkits, improved Unicode support for corpus readers, a BNC corpus reader, and a rewrite of the Punkt sentence segmenter contributed by Joel Nothman. NLTK-Contrib includes an implementation of incremental algorithm for generating referring expression contributed by Margaret Mitchell. For full details see the ChangeLog.

NLTK presented at LinuxFest Northwest [April 2008] Sean Boisen presented NLTK at LinuxFest Northwest, which took place in Bellingham, Washington. His presentation slides are available at: http://semanticbible.com/other/talks/2008/nltk/main.html

NLTK in Google Summer of Code [April 2008] Google Summer of Code will sponsor two NLTK projects. Jason Narad won funding for a project on dependency parsers in NLTK (mentored by Sebastian Riedel and Jason Baldridge). Petro Verkhogliad won funding for a project on natural language generation in NLTK (mentored by Robert Dale and Edward Loper).

Python Software Foundation adopts NLTK for Google Summer of Code application [March 2008] The Python Software Foundation has listed NLTK projects for sponsorship from the 2008 Google Summer of Code program. For details please see http://wiki.python.org/moin/SummerOfCode.

Version 0.9.2 released [March 2008] This version contains a new inference module linked to the Prover9/Mace4 theorem-prover and model checker (Dan Garrette, Ewan Klein). It also includes the VerbNet? and PropBank? corpora along with corpus readers. A bug in the Reuters corpus reader has been fixed. NLTK-Contrib includes new work on the WordNet? browser (Jussi Salmela). For full details see the ChangeLog

Youtube video about NLTK [January 2008] The video from of the NLTK talk at the Bay Area Python Interest Group last July has been posted at http://www.youtube.com/watch?v=keXW_5-llD0 (1h15m)

Version 0.9.1 released [January 2008] This version contains new support for accessing text categorization corpora, along with several corpora categorized for topic, genre, question type, or sentiment. It includes several new corpora: Question classification data (Li & Roth), Reuters 21578 Corpus, Movie Reviews corpus (Pang & Lee), Recognising Textual Entailment (RTE) Challenges. NLTK-Contrib includes expanded support for semantics (Dan Garrette), readability scoring (Thomas Jakobsen, Thomas Skardal), and SIL Toolbox (Greg Aumann). The book contains many improvements in early chapters in response to reader feedback. For full details see the ChangeLog.

3.1.9 2007

NLTK-Lite 0.9 released [October 2007] This version is substantially revised and expanded from version 0.8. The entire toolkit can be accessed via a single import statement "import nltk", and there is a more convenient naming scheme. Calling deprecated functions generates messages that help programmers update their code. The corpus, tagger, and classifier modules have been redesigned. All functionality of the old NLTK 1.4.3 is now covered by NLTK-Lite 0.9. The book has been revised and expanded. A new data package incorporates the existing corpus collection and contains new sections for pre-specified grammars and pre-computed models. Several new corpora have been added, including treebanks for Portuguese, Spanish, Catalan and Dutch. A Macintosh distribution is provided. For full details see the ChangeLog.

NLTK-Lite 0.9b2 released [September 2007] This version is substantially revised and expanded from version 0.8. The entire toolkit can be accessed via a single import statement "import nltk", and many common NLP functions accessed directly, e.g. nltk.PorterStemmer?, nltk.ShiftReduceParser?. The corpus, tagger, and classifier modules have been redesigned. The book has been revised and expanded, and the chapters have been reordered. NLTK has a new data package incorporating the existing corpus collection and adding new sections for pre-specified grammars and pre-computed models. The Floresta Portuguese Treebank has been added. Release 0.9b2 fixes several minor problems with 0.9b1 and removes the numpy dependency. It includes a new corpus and corpus reader for Brazilian Portuguese news text (MacMorphy?) and an improved corpus reader for the Sinica Treebank, and a trained model for Portuguese sentence segmentation.

NLTK-Lite 0.9b1 released [August 2007] This version is substantially revised and expanded from version 0.8. The entire toolkit can be accessed via a single import statement "import nltk", and many common NLP functions accessed directly, e.g. nltk.PorterStemmer?, nltk.ShiftReduceParser?. The corpus, tagger, and classifier modules have been redesigned. The book has been revised and expanded, and the chapters have been reordered. NLTK has a new data package incorporating the existing corpus collection and adding new sections for pre-specified grammars and pre-computed models. The Floresta Portuguese Treebank has been added. For full details see the ChangeLog?.

NLTK talks in São Paulo [August 2007] Steven Bird will present NLTK in a series of talks at the First Brazilian School on Computational Linguistics, at the University of São Paulo in the first week of September.

NLTK talk in Bay Area [July 2007] Steven Bird, Ewan Klein, and Edward Loper will present NLTK at the Bay Area Python Interest Group, at Google on Thursday 12 July.

NLTK-Lite 0.8 released [July 2007] This version is substantially revised and expanded from version 0.7. The code now includes improved interfaces to corpora, chunkers, grammars, frequency distributions, full integration with WordNet? 3.0 and WordNet? similarity measures. The book contains substantial revision of Part I (tokenization, tagging, chunking) and Part II (grammars and parsing). NLTK has several new corpora including the Switchboard Telephone Speech Corpus transcript sample (Talkbank Project), CMU Problem Reports Corpus sample, CONLL2002 POS+NER data, Patient Information Leaflet corpus sample, Indian POS-Tagged data (Bangla, Hindi, Marathi, Telugu), Shakespeare XML corpus sample, and the Universal Declaration of Human Rights corpus with text samples in 300+ languages.

NLTK features in Language Documentation and Conservation article [July 2007] An article Managing Fieldwork Data with Toolbox and the Natural Language Toolkit by Stuart Robinson, Greg Aumann, and Steven Bird appears in the inaugural issue of ''Language Documentation and Conservation''. It discusses several small Python programs for manipulating field data.

NLTK features in ACM Crossroads article [May 2007] An article Getting Started on Natural Language Processing with Python by Nitin Madnani will appear in ''ACM Crossroads'', the ACM Student Journal. It discusses NLTK in detail, and provides several helpful examples including an entertaining free word association program.

NLTK-Lite 0.7.5 released [May 2007] This version contains improved interfaces for WordNet 3.0 and WordNet-Similarity, the Lancaster Stemmer (contributed by Steven Tomcavage), and several new corpora including the Switchboard Telephone Speech Corpus transcript sample (Talkbank Project), CMU Problem Reports Corpus sample, CONLL2002 POS+NER data, Patient Information Leaflet corpus sample and WordNet 3.0 data files. With this distribution WordNet no longer needs to be separately installed.

NLTK-Lite 0.7.4 released [May 2007] This release contains new corpora and corpus readers for Indian POS-Tagged data (Bangla, Hindi, Marathi, Telugu), and the Sinica Treebank, and substantial revision of Part II of the book on structured programming, grammars and parsing.

NLTK-Lite 0.7.3 released [April 2007] This release contains improved chunker and PCFG interfaces, the Shakespeare XML corpus sample and corpus reader, improved tutorials and improved formatting of code samples, and categorization of problem sets by difficulty.

NLTK-Lite 0.7.2 released [March 2007] This release contains new text classifiers (Cosine, NaiveBayes?, Spearman), contributed by Sam Huston, simple feature detectors, the UDHR corpus with text samples in 300+ languages and

a corpus interface; improved tutorials (340 pages in total); additions to contrib area including Kimmo finite-state morphology system, Lambek calculus system, and a demonstration of text classifiers for language identification.

NLTK-Lite 0.7.1 released [January 2007] This release contains bugfixes in the WordNet? and HMM modules.

3.1.10 2006

NLTK-Lite 0.7 released [December 2006] This release contains: new semantic interpretation package (Ewan Klein), new support for SIL Toolbox format (Greg Aumann), new chunking package including cascaded chunking (Steven Bird), new interface to WordNet? 2.1 and Wordnet similarity measures (David Ormiston Smith), new support for Penn Treebank format (Yoav Goldberg), bringing the codebase to 48,000 lines; substantial new chapters on semantic interpretation and chunking, and substantial revisions to several other chapters, bringing the textbook documentation to 280 pages;

NLTK-Lite 0.7b1 released [December 2006] This release contains: new semantic interpretation package (Ewan Klein), new support for SIL Toolbox format (Greg Aumann), new chunking package including cascaded chunking, wordnet package updated for version 2.1 of Wordnet, and prototype wordnet similarity measures (David Ormiston Smith), bringing the codebase to 48,000 lines; substantial new chapters on semantic interpretation and chunking, and substantial revisions to several other chapters, bringing the textbook documentation to 270 pages;

NLTK-Lite 0.6.6 released [October 2006] This release contains bugfixes, improvements to Shoebox file format support, and expanded tutorial discussions of programming and feature-based grammars.

NLTK-Lite 0.6.5 released [July 2006] This release contains improvements to Shoebox file format support (by Stuart Robinson and Greg Aumann); an implementation of hole semantics (by Peter Wang); improvements to lambda calculus and semantic interpretation modules (by Ewan Klein); a new corpus (Sinica Treebank sample); and expanded tutorial discussions of trees, feature-based grammar, unification, PCFGs, and more exercises.

NLTK-Lite passes 10k download milestone [May 2006] We have now had 10,000 downloads of NLTK-Lite in the nine months since it was first released.

NLTK-Lite 0.6.4 released [April 2006] This release contains new corpora (Senseval 2, TIMIT sample), a clusterer, cascaded chunker, and several substantially revised tutorials.

3.1.11 2005

NLTK 1.4 no longer supported [December 2005] The main development has switched to NLTK-Lite. The latest version of NLTK can still be downloaded; see the installation page for instructions.

NLTK-Lite 0.6 released [November 2005] contains bug-fixes, PDF versions of tutorials, expanded fieldwork tutorial, PCFG grammar induction (by Nathan Bodenstab), and prototype concordance and paradigm display tools (by Peter Spiller and Will Hardy).

NLTK-Lite 0.5 released [September 2005] contains bug-fixes, improved tutorials, more project suggestions, and a pronunciation dictionary.

NLTK-Lite 0.4 released [September 2005] contains bug-fixes, improved tutorials, more project suggestions, and probabilistic parsers.

NLTK-Lite 0.3 released [August 2005] contains bug-fixes, documentation clean-up, project suggestions, and the chart parser demos including one for Earley parsing by Jean Mark Gawron.

NLTK-Lite 0.2 released [July 2005] contains bug-fixes, documentation clean-up, and some translations of tutorials into Brazilian Portuguese by Tiago Tresoldi.

NLTK-Lite 0.1 released [July 2005] substantially simplified and streamlined version of NLTK has been released

Brazilian Portuguese Translation [April 2005] top-level pages of this website have been translated into Brazilian Portuguese by Tiago Tresoldi; translations of the tutorials are in preparation http://hermes.sourceforge.net/nltk-br/

1.4.3 Release [February 2005] NLTK 1.4.3 has been released; this is the first version which is compatible with Python 2.4.

3.2 Installing NLTK

NLTK requires Python versions 2.6-2.7 or 3.2+

3.2.1 Mac/Unix

1. Install NLTK: run `sudo pip install -U nltk`
2. Install Numpy (optional): run `sudo pip install -U numpy`
3. Test installation: run `python` then type `import nltk`

For older versions of Python it might be necessary to install setuptools (see http://pypi.python.org/pypi/setuptools) and to install pip (`sudo easy_install pip`).

3.2.2 Windows

These instructions assume that you do not already have Python installed on your machine.

32-bit binary installation

1. Install Python 3.4: http://www.python.org/downloads/ (avoid the 64-bit versions)
2. Install Numpy (optional): http://sourceforge.net/projects/numpy/files/NumPy/ (the version that specifies python3.4)
3. Install NLTK: http://pypi.python.org/pypi/nltk
4. Test installation: `Start>Python34`, then type `import nltk`

3.2.3 Installing Third-Party Software

Please see: https://github.com/nltk/nltk/wiki/Installing-Third-Party-Software

3.3 Installing NLTK Data

NLTK comes with many corpora, toy grammars, trained models, etc. A complete list is posted at: http://nltk.org/nltk_data/

To install the data, first install NLTK (see http://nltk.org/install.html), then use NLTK's data downloader as described below.

Apart from individual data packages, you can download the entire collection (using "all"), or just the data required for the examples and exercises in the book (using "book"), or just the corpora and no grammars or trained models (using "all-corpora").

3.3.1 Interactive installer

For central installation on a multi-user machine, do the following from an administrator account.

Run the Python interpreter and type the commands:

```
>>> import nltk
>>> nltk.download()
```

A new window should open, showing the NLTK Downloader. Click on the File menu and select Change Download Directory. For central installation, set this to `C:\nltk_data` (Windows), or `/usr/share/nltk_data` (Mac, Unix). Next, select the packages or collections you want to download.

If you did not install the data to one of the above central locations, you will need to set the `NLTK_DATA` environment variable to specify the location of the data. (On a Windows machine, right click on "My Computer" then select `Properties > Advanced > Environment Variables > User Variables > New...`)

Test that the data has been installed as follows. (This assumes you downloaded the Brown Corpus):

```
>>> from nltk.corpus import brown
>>> brown.words()
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
```

Installing via a proxy web server

If your web connection uses a proxy server, you should specify the proxy address as follows. In the case of an authenticating proxy, specify a username and password. If the proxy is set to None then this function will attempt to detect the system proxy.

```
>>> nltk.set_proxy('http://proxy.example.com:3128', ('USERNAME', 'PASSWORD'))
>>> nltk.download()
```

3.3.2 Command line installation

The downloader will search for an existing `nltk_data` directory to install NLTK data. If one does not exist it will attempt to create one in a central location (when using an administrator account) or otherwise in the user's filespace. If necessary, run the download command from an administrator account, or using sudo. The default system location on Windows is `C:\nltk_data`; and on Mac and Unix is `/usr/share/nltk_data`. You can use the `-d` flag to specify a different location (but if you do this, be sure to set the `NLTK_DATA` environment variable accordingly).

Python 2.5-2.7: Run the command `python -m nltk.downloader all`. To ensure central installation, run the command `sudo python -m nltk.downloader -d /usr/share/nltk_data all`.

Windows: Use the "Run..." option on the Start menu. Windows Vista users need to first turn on this option, using `Start -> Properties -> Customize` to check the box to activate the "Run..." option.

Test the installation: Check that the user environment and privileges are set correctly by logging in to a user account, starting the Python interpreter, and accessing the Brown Corpus (see the previous section).

3.4 Contribute to NLTK

The Natural Language Toolkit exists thanks to the efforts of dozens of voluntary developers who have contributed functionality and bugfixes since the project began in 2000 (contributors).

In 2015 we are especially keen to improve NLTK coverage for: dependency parsing, machine translation, sentiment analysis, twitter processing.

New material in these areas will be covered in the second edition of the NLTK book, anticipated in early 2016.

- desired enhancements
- contribute a corpus
- nltk-dev mailing list
- GitHub Project

3.4.1 NLTK Core Developers

The NLTK project is led by Steven Bird, Ewan Klein, and Edward Loper. Individual packages are maintained by the following people:

Semantics Dan Garrette, Austin, USA (`nltk.sem, nltk.inference`)

Parsing Peter Ljunglöf, Gothenburg, Sweden (`nltk.parse, nltk.featstruct`)

Metrics Joel Nothman, Sydney, Australia (`nltk.metrics, nltk.tokenize.punkt`)

Python 3 Mikhail Korobov, Ekaterinburg, Russia

Releases Steven Bird, Melbourne, Australia

3.5 nltk Package

3.5.1 `nltk` Package

The Natural Language Toolkit (NLTK) is an open source Python library for Natural Language Processing. A free online book is available. (If you use the library for academic research, please cite the book.)

Steven Bird, Ewan Klein, and Edward Loper (2009). Natural Language Processing with Python. O'Reilly Media Inc. http://nltk.org/book

@version: 3.0.5

`nltk.__init__.demo()`

3.5.2 `align` Module

Experimental functionality for bitext alignment. These interfaces are prone to change.

3.5.3 `collocations` Module

Tools to identify collocations — words that often appear consecutively — within corpora. They may also be used to find other associations between word occurrences. See Manning and Schutze ch. 5 at http://nlp.stanford.edu/fsnlp/promo/colloc.pdf and the Text::NSP Perl package at http://ngram.sourceforge.net

Finding collocations requires first calculating the frequencies of words and their appearance in the context of other words. Often the collection of words will then requiring filtering to only retain useful content terms. Each ngram of words may then be scored according to some association measure, in order to determine the relative likelihood of each ngram being a collocation.

The BigramCollocationFinder and TrigramCollocationFinder classes provide these functionalities, dependent on being provided a function which scores a ngram given appropriate frequency counts. A number of standard association measures are provided in bigram_measures and trigram_measures.

class nltk.collocations.**BigramCollocationFinder**(*word_fd*, *bigram_fd*, *window_size=2*)
Bases: nltk.collocations.AbstractCollocationFinder

A tool for the finding and ranking of bigram collocations or other association measures. It is often useful to use from_words() rather than constructing an instance directly.

default_ws = 2

classmethod from_words(*words*, *window_size=2*)
Construct a BigramCollocationFinder for all bigrams in the given sequence. When window_size > 2, count non-contiguous bigrams, in the style of Church and Hanks's (1990) association ratio.

score_ngram(*score_fn*, *w1*, *w2*)
Returns the score for a given bigram using the given scoring function. Following Church and Hanks (1990), counts are scaled by a factor of 1/(window_size - 1).

class nltk.collocations.**TrigramCollocationFinder**(*word_fd*, *bigram_fd*, *wildcard_fd*, *trigram_fd*)
Bases: nltk.collocations.AbstractCollocationFinder

A tool for the finding and ranking of trigram collocations or other association measures. It is often useful to use from_words() rather than constructing an instance directly.

bigram_finder()
Constructs a bigram collocation finder with the bigram and unigram data from this finder. Note that this does not include any filtering applied to this finder.

default_ws = 3

classmethod from_words(*words*, *window_size=3*)
Construct a TrigramCollocationFinder for all trigrams in the given sequence.

score_ngram(*score_fn*, *w1*, *w2*, *w3*)
Returns the score for a given trigram using the given scoring function.

class nltk.collocations.**QuadgramCollocationFinder**(*word_fd*, *quadgram_fd*, *ii*, *iii*, *ixi*, *ixxi*, *iixi*, *ixii*)
Bases: nltk.collocations.AbstractCollocationFinder

A tool for the finding and ranking of quadgram collocations or other association measures. It is often useful to use from_words() rather than constructing an instance directly.

default_ws = 4

classmethod from_words(*words*, *window_size=4*)

score_ngram(*score_fn*, *w1*, *w2*, *w3*, *w4*)

3.5.4 data Module

Functions to find and load NLTK resource files, such as corpora, grammars, and saved processing objects. Resource files are identified using URLs, such as nltk:corpora/abc/rural.txt or http://nltk.org/sample/toy.cfg. The following URL protocols are supported:

- file:path: Specifies the file whose path is *path*. Both relative and absolute paths may be used.
- http://host/path: Specifies the file stored on the web server *host* at path *path*.

- `nltk:path`: Specifies the file stored in the NLTK data package at *path*. NLTK will search for these files in the directories specified by `nltk.data.path`.

If no protocol is specified, then the default protocol `nltk:` will be used.

This module provides to functions that can be used to access a resource file, given its URL: `load()` loads a given resource, and adds it to a resource cache; and `retrieve()` copies a given resource to a local file.

`nltk.data.`**`path = ['/home/docs/nltk_data', '/usr/share/nltk_data', '/usr/local/share/nltk_data', '/usr/lib/nltk_data', '/usr/loc`**
A list of directories where the NLTK data package might reside. These directories will be checked in order when looking for a resource in the data package. Note that this allows users to substitute in their own versions of resources, if they have them (e.g., in their home directory under ~/nltk_data).

class `nltk.data.`**`PathPointer`**
Bases: `object`

An abstract base class for 'path pointers,' used by NLTK's data package to identify specific paths. Two subclasses exist: `FileSystemPathPointer` identifies a file that can be accessed directly via a given absolute path. `ZipFilePathPointer` identifies a file contained within a zipfile, that can be accessed by reading that zipfile.

`file_size``()`
Return the size of the file pointed to by this path pointer, in bytes.

Raises IOError If the path specified by this pointer does not contain a readable file.

`join``(`*fileid*`)`
Return a new path pointer formed by starting at the path identified by this pointer, and then following the relative path given by `fileid`. The path components of `fileid` should be separated by forward slashes, regardless of the underlying file system's path seperator character.

`open``(`*encoding=None*`)`
Return a seekable read-only stream that can be used to read the contents of the file identified by this path pointer.

Raises IOError If the path specified by this pointer does not contain a readable file.

class `nltk.data.`**`FileSystemPathPointer`**`(`**args*, ***kwargs*`)`
Bases: *`nltk.data.PathPointer`*, `unicode`

A path pointer that identifies a file which can be accessed directly via a given absolute path.

`file_size``()`

`join``(`*fileid*`)`

`open``(`*encoding=None*`)`

`path`
The absolute path identified by this path pointer.

class `nltk.data.`**`BufferedGzipFile`**`(`**args*, ***kwargs*`)`
Bases: `gzip.GzipFile`

A `GzipFile` subclass that buffers calls to `read()` and `write()`. This allows faster reads and writes of data to and from gzip-compressed files at the cost of using more memory.

The default buffer size is 2MB.

`BufferedGzipFile` is useful for loading large gzipped pickle objects as well as writing large encoded feature files for classifier training.

`SIZE = 2097152`

`close``()`

flush(*lib_mode=2*)

read(*size=None*)

write(*data*, *size=-1*)

Parameters

- **data** (*bytes*) – bytes to write to file or buffer
- **size** (*int*) – buffer at least size bytes before writing to file

class nltk.data.**GzipFileSystemPathPointer**(**args*, ***kwargs*)
Bases: *nltk.data.FileSystemPathPointer*

A subclass of FileSystemPathPointer that identifies a gzip-compressed file located at a given absolute path. GzipFileSystemPathPointer is appropriate for loading large gzip-compressed pickle objects efficiently.

open(*encoding=None*)

class nltk.data.**GzipFileSystemPathPointer**(**args*, ***kwargs*)
Bases: *nltk.data.FileSystemPathPointer*

A subclass of FileSystemPathPointer that identifies a gzip-compressed file located at a given absolute path. GzipFileSystemPathPointer is appropriate for loading large gzip-compressed pickle objects efficiently.

open(*encoding=None*)

nltk.data.**find**(*resource_name*, *paths=None*)
Find the given resource by searching through the directories and zip files in paths, where a None or empty string specifies an absolute path. Returns a corresponding path name. If the given resource is not found, raise a LookupError, whose message gives a pointer to the installation instructions for the NLTK downloader.

Zip File Handling:

- If resource_name contains a component with a .zip extension, then it is assumed to be a zipfile; and the remaining path components are used to look inside the zipfile.
- If any element of nltk.data.path has a .zip extension, then it is assumed to be a zipfile.
- If a given resource name that does not contain any zipfile component is not found initially, then find() will make a second attempt to find that resource, by replacing each component *p* in the path with *p.zip/p*. For example, this allows find() to map the resource name corpora/chat80/cities.pl to a zip file path pointer to corpora/chat80.zip/chat80/cities.pl.
- When using find() to locate a directory contained in a zipfile, the resource name must end with the forward slash character. Otherwise, find() will not locate the directory.

Parameters **resource_name** (*str or unicode*) – The name of the resource to search for. Resource names are posix-style relative path names, such as corpora/brown. Directory names will be automatically converted to a platform-appropriate path separator.

Return type str

nltk.data.**retrieve**(*resource_url*, *filename=None*, *verbose=True*)
Copy the given resource to a local file. If no filename is specified, then use the URL's filename. If there is already a file named filename, then raise a ValueError.

Parameters **resource_url** (*str*) – A URL specifying where the resource should be loaded from. The default protocol is "nltk:", which searches for the file in the the NLTK data package.

nltk.data.**FORMATS = {u'cfg': u'A context free grammar.', u'raw': u'The raw (byte string) contents of a file.', u'fcfg': u'A f**
A dictionary describing the formats that are supported by NLTK's load() method. Keys are format names, and values are format descriptions.

nltk.data.**AUTO_FORMATS = {u'cfg': u'cfg', u'txt': u'text', u'fcfg': u'fcfg', u'pcfg': u'pcfg', u'val': u'val', u'yaml': u'yaml**
A dictionary mapping from file extensions to format names, used by load() when format="auto" to decide the format for a given resource url.

nltk.data.**load**(*resource_url*, *format=u'auto'*, *cache=True*, *verbose=False*, *logic_parser=None*, *fstruct_reader=None*, *encoding=None*)
Load a given resource from the NLTK data package. The following resource formats are currently supported:

- `pickle`
- `json`
- `yaml`
- `cfg` (context free grammars)
- `pcfg` (probabilistic CFGs)
- `fcfg` (feature-based CFGs)
- `fol` (formulas of First Order Logic)
- `logic` (Logical formulas to be parsed by the given logic_parser)
- `val` (valuation of First Order Logic model)
- `text` (the file contents as a unicode string)
- `raw` (the raw file contents as a byte string)

If no format is specified, `load()` will attempt to determine a format based on the resource name's file extension. If that fails, `load()` will raise a `ValueError` exception.

For all text formats (everything except `pickle`, `json`, `yaml` and `raw`), it tries to decode the raw contents using UTF-8, and if that doesn't work, it tries with ISO-8859-1 (Latin-1), unless the `encoding` is specified.

Parameters

- **resource_url** (*str*) – A URL specifying where the resource should be loaded from. The default protocol is "nltk:", which searches for the file in the the NLTK data package.
- **cache** (*bool*) – If true, add this resource to a cache. If load() finds a resource in its cache, then it will return it from the cache rather than loading it. The cache uses weak references, so a resource wil automatically be expunged from the cache when no more objects are using it.
- **verbose** (*bool*) – If true, print a message when loading a resource. Messages are not displayed when a resource is retrieved from the cache.
- **logic_parser** (*LogicParser*) – The parser that will be used to parse logical expressions.
- **fstruct_reader** (FeatStructReader) – The parser that will be used to parse the feature structure of an fcfg.
- **encoding** (*str*) – the encoding of the input; only used for text formats.

nltk.data.**show_cfg**(*resource_url*, *escape=u'##'*)
Write out a grammar file, ignoring escaped and empty lines.

Parameters

- **resource_url** (*str*) – A URL specifying where the resource should be loaded from. The default protocol is "nltk:", which searches for the file in the the NLTK data package.

- **escape** (*str*) – Prepended string that signals lines to be ignored

nltk.data.**clear_cache**()
Remove all objects from the resource cache. :see: load()

class nltk.data.**LazyLoader**(**args*, ***kwargs*)
Bases: object

class nltk.data.**OpenOnDemandZipFile**(**args*, ***kwargs*)
Bases: zipfile.ZipFile

A subclass of zipfile.ZipFile that closes its file pointer whenever it is not using it; and re-opens it when it needs to read data from the zipfile. This is useful for reducing the number of open file handles when many zip files are being accessed at once. OpenOnDemandZipFile must be constructed from a filename, not a file-like object (to allow re-opening). OpenOnDemandZipFile is read-only (i.e. write() and writestr() are disabled.

read(*name*)

write(**args*, ***kwargs*)

Raises NotImplementedError OpenOnDemandZipfile is read-only

writestr(**args*, ***kwargs*)

Raises NotImplementedError OpenOnDemandZipfile is read-only

class nltk.data.**GzipFileSystemPathPointer**(**args*, ***kwargs*)
Bases: *nltk.data.FileSystemPathPointer*

A subclass of FileSystemPathPointer that identifies a gzip-compressed file located at a given absolute path. GzipFileSystemPathPointer is appropriate for loading large gzip-compressed pickle objects efficiently.

open(*encoding=None*)

class nltk.data.**SeekableUnicodeStreamReader**(**args*, ***kwargs*)
Bases: object

A stream reader that automatically encodes the source byte stream into unicode (like codecs.StreamReader); but still supports the seek() and tell() operations correctly. This is in contrast to codecs.StreamReader, which provide *broken* seek() and tell() methods.

This class was motivated by StreamBackedCorpusView, which makes extensive use of seek() and tell(), and needs to be able to handle unicode-encoded files.

Note: this class requires stateless decoders. To my knowledge, this shouldn't cause a problem with any of python's builtin unicode encodings.

DEBUG = True
If true, then perform extra sanity checks.

bytebuffer = None
A buffer to use bytes that have been read but have not yet been decoded. This is only used when the final bytes from a read do not form a complete encoding for a character.

char_seek_forward(*offset*)
Move the read pointer forward by offset characters.

close()
Close the underlying stream.

closed
True if the underlying stream is closed.

decode = None
The function that is used to decode byte strings into unicode strings.

encoding = None
The name of the encoding that should be used to encode the underlying stream.

errors = None
The error mode that should be used when decoding data from the underlying stream. Can be 'strict', 'ignore', or 'replace'.

linebuffer = None
A buffer used by `readline()` to hold characters that have been read, but have not yet been returned by `read()` or `readline()`. This buffer consists of a list of unicode strings, where each string corresponds to a single line. The final element of the list may or may not be a complete line. Note that the existence of a linebuffer makes the `tell()` operation more complex, because it must backtrack to the beginning of the buffer to determine the correct file position in the underlying byte stream.

mode
The mode of the underlying stream.

name
The name of the underlying stream.

next()
Return the next decoded line from the underlying stream.

read(*size=None*)
Read up to `size` bytes, decode them using this reader's encoding, and return the resulting unicode string.

Parameters **size** (*int*) – The maximum number of bytes to read. If not specified, then read as many bytes as possible.

Return type unicode

readline(*size=None*)
Read a line of text, decode it using this reader's encoding, and return the resulting unicode string.

Parameters **size** (*int*) – The maximum number of bytes to read. If no newline is encountered before `size` bytes have been read, then the returned value may not be a complete line of text.

readlines(*sizehint=None*, *keepends=True*)
Read this file's contents, decode them using this reader's encoding, and return it as a list of unicode lines.

Return type list(unicode)

Parameters

- **sizehint** – Ignored.
- **keepends** – If false, then strip newlines.

seek(*offset*, *whence=0*)
Move the stream to a new file position. If the reader is maintaining any buffers, then they will be cleared.

Parameters

- **offset** – A byte count offset.
- **whence** – If 0, then the offset is from the start of the file (offset should be positive), if 1, then the offset is from the current position (offset may be positive or negative); and if 2, then the offset is from the end of the file (offset should typically be negative).

stream = None
The underlying stream.

tell()
Return the current file position on the underlying byte stream. If this reader is maintaining any buffers, then the returned file position will be the position of the beginning of those buffers.

xreadlines()
Return self

3.5.5 downloader Module

The NLTK corpus and module downloader. This module defines several interfaces which can be used to download corpora, models, and other data packages that can be used with NLTK.

Downloading Packages

If called with no arguments, `download()` will display an interactive interface which can be used to download and install new packages. If Tkinter is available, then a graphical interface will be shown, otherwise a simple text interface will be provided.

Individual packages can be downloaded by calling the `download()` function with a single argument, giving the package identifier for the package that should be downloaded:

```
>>> download('treebank')
[nltk_data] Downloading package 'treebank'...
[nltk_data]   Unzipping corpora/treebank.zip.
```

NLTK also provides a number of "package collections", consisting of a group of related packages. To download all packages in a colleciton, simply call `download()` with the collection's identifier:

```
>>> download('all-corpora')
[nltk_data] Downloading package 'abc'...
[nltk_data]   Unzipping corpora/abc.zip.
[nltk_data] Downloading package 'alpino'...
[nltk_data]   Unzipping corpora/alpino.zip.
  ...
[nltk_data] Downloading package 'words'...
[nltk_data]   Unzipping corpora/words.zip.
```

Download Directory

By default, packages are installed in either a system-wide directory (if Python has sufficient access to write to it); or in the current user's home directory. However, the `download_dir` argument may be used to specify a different installation target, if desired.

See `Downloader.default_download_dir()` for more a detailed description of how the default download directory is chosen.

NLTK Download Server

Before downloading any packages, the corpus and module downloader contacts the NLTK download server, to retrieve an index file describing the available packages. By default, this index file is loaded from `http://www.nltk.org/nltk_data/`. If necessary, it is possible to create a new `Downloader` object, specifying a different URL for the package index file.

Usage:

```
python nltk/downloader.py [-d DATADIR] [-q] [-f] [-k] PACKAGE_IDS
```

or:

```
python -m nltk.downloader [-d DATADIR] [-q] [-f] [-k] PACKAGE_IDS
```

class nltk.downloader.**Collection**(*id*, *children*, *name=None*, ***kw*)
Bases: object

A directory entry for a collection of downloadable packages. These entries are extracted from the XML index file that is downloaded by Downloader.

children = None
A list of the Collections or Packages directly contained by this collection.

static fromxml(*xml*)

id = None
A unique identifier for this collection.

name = None
A string name for this collection.

packages = None
A list of Packages contained by this collection or any collections it recursively contains.

unicode_repr()

class nltk.downloader.**Downloader**(*server_index_url=None*, *download_dir=None*)
Bases: object

A class used to access the NLTK data server, which can be used to download corpora and other data packages.

DEFAULT_URL = u'http://www.nltk.org/nltk_data/'
The default URL for the NLTK data server's index. An alternative URL can be specified when creating a new Downloader object.

INDEX_TIMEOUT = 3600
The amount of time after which the cached copy of the data server index will be considered 'stale,' and will be re-downloaded.

INSTALLED = u'installed'
A status string indicating that a package or collection is installed and up-to-date.

NOT_INSTALLED = u'not installed'
A status string indicating that a package or collection is not installed.

PARTIAL = u'partial'
A status string indicating that a collection is partially installed (i.e., only some of its packages are installed.)

STALE = u'out of date'
A status string indicating that a package or collection is corrupt or out-of-date.

clear_status_cache(*id=None*)

collections()

corpora()

default_download_dir()
Return the directory to which packages will be downloaded by default. This value can be overridden using the constructor, or on a case-by-case basis using the download_dir argument when calling download().

On Windows, the default download directory is `PYTHONHOME/lib/nltk`, where *PYTHONHOME* is the directory containing Python, e.g. `C:\Python25`.

On all other platforms, the default directory is the first of the following which exists or which can be created with write permission: `/usr/share/nltk_data`, `/usr/local/share/nltk_data`, `/usr/lib/nltk_data`, `/usr/local/lib/nltk_data`, `~/nltk_data`.

download(*info_or_id=None*, *download_dir=None*, *quiet=False*, *force=False*, *prefix=u'[nltk_data] '*, *halt_on_error=True*, *raise_on_error=False*)

download_dir
The default directory to which packages will be downloaded. This defaults to the value returned by `default_download_dir()`. To override this default on a case-by-case basis, use the `download_dir` argument when calling `download()`.

incr_download(*info_or_id*, *download_dir=None*, *force=False*)

index()
Return the XML index describing the packages available from the data server. If necessary, this index will be downloaded from the data server.

info(*id*)
Return the `Package` or `Collection` record for the given item.

is_installed(*info_or_id*, *download_dir=None*)

is_stale(*info_or_id*, *download_dir=None*)

list(*download_dir=None*, *show_packages=True*, *show_collections=True*, *header=True*, *more_prompt=False*, *skip_installed=False*)

models()

packages()

status(*info_or_id*, *download_dir=None*)
Return a constant describing the status of the given package or collection. Status can be one of `INSTALLED`, `NOT_INSTALLED`, `STALE`, or `PARTIAL`.

update(*quiet=False*, *prefix=u'[nltk_data] '*)
Re-download any packages whose status is STALE.

url
The URL for the data server's index file.

xmlinfo(*id*)
Return the XML info record for the given item

class `nltk.downloader.`**DownloaderGUI**(*dataserver*, *use_threads=True*)
Bases: `object`

Graphical interface for downloading packages from the NLTK data server.

COLUMNS = [u'', u'Identifier', u'Name', u'Size', u'Status', u'Unzipped Size', u'Copyright', u'Contact', u'License', u'Autl
A list of the names of columns. This controls the order in which the columns will appear. If this is edited, then `_package_to_columns()` may need to be edited to match.

COLUMN_WEIGHTS = {u'': 0, u'Status': 0, u'Name': 5, u'Size': 0}
A dictionary specifying how columns should be resized when the table is resized. Columns with weight 0 will not be resized at all; and columns with high weight will be resized more. Default weight (for columns not explicitly listed) is 1.

COLUMN_WIDTHS = {u'': 1, u'Status': 12, u'Name': 45, u'Unzipped Size': 10, u'Identifier': 20, u'Size': 10}
A dictionary specifying how wide each column should be, in characters. The default width (for columns not explicitly listed) is specified by DEFAULT_COLUMN_WIDTH.

DEFAULT_COLUMN_WIDTH = 30
The default width for columns that are not explicitly listed in COLUMN_WIDTHS.

HELP = u'This tool can be used to download a variety of corpora and models\nthat can be used with NLTK. Each corpus c

INITIAL_COLUMNS = [u'', u'Identifier', u'Name', u'Size', u'Status']
The set of columns that should be displayed by default.

about(**e*)

c = u'Status'

destroy(**e*)

help(**e*)

mainloop(**args*, ***kwargs*)

class nltk.downloader.**DownloaderMessage**
Bases: object

A status message object, used by incr_download to communicate its progress.

class nltk.downloader.**DownloaderShell**(*dataserver*)
Bases: object

run()

class nltk.downloader.**ErrorMessage**(*package*, *message*)
Bases: *nltk.downloader.DownloaderMessage*

Data server encountered an error

class nltk.downloader.**FinishCollectionMessage**(*collection*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has finished working on a collection of packages.

class nltk.downloader.**FinishDownloadMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has finished downloading a package.

class nltk.downloader.**FinishPackageMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has finished working on a package.

class nltk.downloader.**FinishUnzipMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has finished unzipping a package.

class nltk.downloader.**Package**(*id*, *url*, *name=None*, *subdir=u''*, *size=None*, *unzipped_size=None*, *checksum=None*, *svn_revision=None*, *copyright=u'Unknown'*, *contact=u'Unknown'*, *license=u'Unknown'*, *author=u'Unknown'*, *unzip=True*, ***kw*)
Bases: object

A directory entry for a downloadable package. These entries are extracted from the XML index file that is downloaded by Downloader. Each package consists of a single file; but if that file is a zip file, then it can be automatically decompressed when the package is installed.

author = None
Author of this package.

checksum = None
The MD-5 checksum of the package file.

contact = None
Name & email of the person who should be contacted with questions about this package.

copyright = None
Copyright holder for this package.

filename = None
The filename that should be used for this package's file. It is formed by joining `self.subdir` with `self.id`, and using the same extension as `url`.

static fromxml(*xml*)

id = None
A unique identifier for this package.

license = None
License information for this package.

name = None
A string name for this package.

size = None
The filesize (in bytes) of the package file.

subdir = None
The subdirectory where this package should be installed. E.g., `'corpora'` or `'taggers'`.

svn_revision = None
A subversion revision number for this package.

unicode_repr()

unzip = None
A flag indicating whether this corpus should be unzipped by default.

unzipped_size = None
The total filesize of the files contained in the package's zipfile.

url = None
A URL that can be used to download this package's file.

class nltk.downloader.**ProgressMessage**(*progress*)
Bases: *nltk.downloader.DownloaderMessage*

Indicates how much progress the data server has made

class nltk.downloader.**SelectDownloadDirMessage**(*download_dir*)
Bases: *nltk.downloader.DownloaderMessage*

Indicates what download directory the data server is using

class nltk.downloader.**StaleMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

The package download file is out-of-date or corrupt

class nltk.downloader.**StartCollectionMessage**(*collection*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has started working on a collection of packages.

class nltk.downloader.**StartDownloadMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has started downloading a package.

class nltk.downloader.**StartPackageMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has started working on a package.

class nltk.downloader.**StartUnzipMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

Data server has started unzipping a package.

class nltk.downloader.**UpToDateMessage**(*package*)
Bases: *nltk.downloader.DownloaderMessage*

The package download file is already up-to-date

nltk.downloader.**build_index**(*root*, *base_url*)
Create a new data.xml index file, by combining the xml description files for various packages and collections. root should be the path to a directory containing the package xml and zip files; and the collection xml files. The root directory is expected to have the following subdirectories:

```
root/
  packages/ .................. subdirectory for packages
    corpora/ ................. zip & xml files for corpora
    grammars/ ................ zip & xml files for grammars
    taggers/ ................. zip & xml files for taggers
    tokenizers/ .............. zip & xml files for tokenizers
    etc.
  collections/ ............... xml files for collections
```

For each package, there should be two files: package.zip (where *package* is the package name) which contains the package itself as a compressed zip file; and package.xml, which is an xml description of the package. The zipfile package.zip should expand to a single subdirectory named package/. The base filename package must match the identifier given in the package's xml file.

For each collection, there should be a single file collection.zip describing the collection, where *collection* is the name of the collection.

All identifiers (for both packages and collections) must be unique.

nltk.downloader.**download_gui**()

nltk.downloader.**download_shell**()

nltk.downloader.**md5_hexdigest**(*file*)
Calculate and return the MD5 checksum for a given file. file may either be a filename or an open stream.

nltk.downloader.**unzip**(*filename*, *root*, *verbose=True*)
Extract the contents of the zip file filename into the directory root.

nltk.downloader.**update**()

3.5.6 featstruct Module

Basic data classes for representing feature structures, and for performing basic operations on those feature structures. A feature structure is a mapping from feature identifiers to feature values, where each feature value is either a basic value

(such as a string or an integer), or a nested feature structure. There are two types of feature structure, implemented by two subclasses of `FeatStruct`:

- feature dictionaries, implemented by `FeatDict`, act like Python dictionaries. Feature identifiers may be strings or instances of the `Feature` class.
- feature lists, implemented by `FeatList`, act like Python lists. Feature identifiers are integers.

Feature structures are typically used to represent partial information about objects. A feature identifier that is not mapped to a value stands for a feature whose value is unknown (*not* a feature without a value). Two feature structures that represent (potentially overlapping) information about the same object can be combined by unification. When two inconsistent feature structures are unified, the unification fails and returns None.

Features can be specified using "feature paths", or tuples of feature identifiers that specify path through the nested feature structures to a value. Feature structures may contain reentrant feature values. A "reentrant feature value" is a single feature value that can be accessed via multiple feature paths. Unification preserves the reentrance relations imposed by both of the unified feature structures. In the feature structure resulting from unification, any modifications to a reentrant feature value will be visible using any of its feature paths.

Feature structure variables are encoded using the `nltk.sem.Variable` class. The variables' values are tracked using a bindings dictionary, which maps variables to their values. When two feature structures are unified, a fresh bindings dictionary is created to track their values; and before unification completes, all bound variables are replaced by their values. Thus, the bindings dictionaries are usually strictly internal to the unification process. However, it is possible to track the bindings of variables if you choose to, by supplying your own initial bindings dictionary to the `unify()` function.

When unbound variables are unified with one another, they become aliased. This is encoded by binding one variable to the other.

Lightweight Feature Structures

Many of the functions defined by `nltk.featstruct` can be applied directly to simple Python dictionaries and lists, rather than to full-fledged `FeatDict` and `FeatList` objects. In other words, Python `dicts` and `lists` can be used as "light-weight" feature structures.

```
>>> from nltk.featstruct import unify
>>> unify(dict(x=1, y=dict()), dict(a='a', y=dict(b='b')))
{'y': {'b': 'b'}, 'x': 1, 'a': 'a'}
```

However, you should keep in mind the following caveats:

- Python dictionaries & lists ignore reentrance when checking for equality between values. But two FeatStructs with different reentrances are considered nonequal, even if all their base values are equal.
- FeatStructs can be easily frozen, allowing them to be used as keys in hash tables. Python dictionaries and lists can not.
- FeatStructs display reentrance in their string representations; Python dictionaries and lists do not.
- FeatStructs may *not* be mixed with Python dictionaries and lists (e.g., when performing unification).
- FeatStructs provide a number of useful methods, such as `walk()` and `cyclic()`, which are not available for Python dicts and lists.

In general, if your feature structures will contain any reentrances, or if you plan to use them as dictionary keys, it is strongly recommended that you use full-fledged `FeatStruct` objects.

class `nltk.featstruct.`**`FeatStruct`**
Bases: `nltk.sem.logic.SubstituteBindingsI`

A mapping from feature identifiers to feature values, where each feature value is either a basic value (such as a string or an integer), or a nested feature structure. There are two types of feature structure:

- feature dictionaries, implemented by `FeatDict`, act like Python dictionaries. Feature identifiers may be strings or instances of the `Feature` class.
- feature lists, implemented by `FeatList`, act like Python lists. Feature identifiers are integers.

Feature structures may be indexed using either simple feature identifiers or 'feature paths.' A feature path is a sequence of feature identifiers that stand for a corresponding sequence of indexing operations. In particular, `fstruct[(f1,f2,...,fn)]` is equivalent to `fstruct[f1][f2]...[fn]`.

Feature structures may contain reentrant feature structures. A "reentrant feature structure" is a single feature structure object that can be accessed via multiple feature paths. Feature structures may also be cyclic. A feature structure is "cyclic" if there is any feature path from the feature structure to itself.

Two feature structures are considered equal if they assign the same values to all features, and have the same reentrancies.

By default, feature structures are mutable. They may be made immutable with the `freeze()` method. Once they have been frozen, they may be hashed, and thus used as dictionary keys.

copy(*deep=True*)
Return a new copy of `self`. The new copy will not be frozen.

Parameters **deep** – If true, create a deep copy; if false, create a shallow copy.

cyclic()
Return True if this feature structure contains itself.

equal_values(*other*, *check_reentrance=False*)
Return True if `self` and `other` assign the same value to to every feature. In particular, return true if `self[p]==other[p]` for every feature path *p* such that `self[p]` or `other[p]` is a base value (i.e., not a nested feature structure).

Parameters **check_reentrance** – If True, then also return False if there is any difference between the reentrances of `self` and `other`.

Note the == is equivalent to `equal_values()` with `check_reentrance=True`.

freeze()
Make this feature structure, and any feature structures it contains, immutable. Note: this method does not attempt to 'freeze' any feature value that is not a `FeatStruct`; it is recommended that you use only immutable feature values.

frozen()
Return True if this feature structure is immutable. Feature structures can be made immutable with the `freeze()` method. Immutable feature structures may not be made mutable again, but new mutable copies can be produced with the `copy()` method.

remove_variables()
Return the feature structure that is obtained by deleting any feature whose value is a `Variable`.

Return type *FeatStruct*

rename_variables(*vars=None*, *used_vars=()*, *new_vars=None*)

See `nltk.featstruct.rename_variables()`

retract_bindings(*bindings*)

See `nltk.featstruct.retract_bindings()`

substitute_bindings(*bindings*)

See `nltk.featstruct.substitute_bindings()`

subsumes(*other*)
Return True if `self` subsumes `other`. I.e., return true If unifying `self` with `other` would result in a feature structure equal to `other`.

unify(*other*, *bindings=None*, *trace=False*, *fail=None*, *rename_vars=True*)

variables()

See `nltk.featstruct.find_variables()`

walk()
Return an iterator that generates this feature structure, and each feature structure it contains. Each feature structure will be generated exactly once.

class `nltk.featstruct.`**FeatDict**(*features=None*, ***morefeatures*)
Bases: *nltk.featstruct.FeatStruct*, `dict`

A feature structure that acts like a Python dictionary. I.e., a mapping from feature identifiers to feature values, where a feature identifier can be a string or a `Feature`; and where a feature value can be either a basic value (such as a string or an integer), or a nested feature structure. A feature identifiers for a `FeatDict` is sometimes called a "feature name".

Two feature dicts are considered equal if they assign the same values to all features, and have the same reentrances.

See `FeatStruct` for information about feature paths, reentrance, cyclic feature structures, mutability, freezing, and hashing.

clear() → None. Remove all items from D.
If self is frozen, raise ValueError.

get(*name_or_path*, *default=None*)
If the feature with the given name or path exists, return its value; otherwise, return `default`.

has_key(*name_or_path*)
Return true if a feature with the given name or path exists.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised If self is frozen, raise ValueError.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty. If self is frozen, raise ValueError.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D
If self is frozen, raise ValueError.

unicode_repr()
Display a single-line representation of this feature structure, suitable for embedding in other representations.

update(*features=None*, ***morefeatures*)

class `nltk.featstruct.`**FeatList**(*features=()*)
Bases: *nltk.featstruct.FeatStruct*, `list`

A list of feature values, where each feature value is either a basic value (such as a string or an integer), or a nested feature structure.

Feature lists may contain reentrant feature values. A "reentrant feature value" is a single feature value that can be accessed via multiple feature paths. Feature lists may also be cyclic.

Two feature lists are considered equal if they assign the same values to all features, and have the same reentrances.

See `FeatStruct` for information about feature paths, reentrance, cyclic feature structures, mutability, freezing, and hashing.

append(**args*, ***kwargs*)
L.append(object) – append object to end If self is frozen, raise ValueError.

extend(**args*, ***kwargs*)
L.extend(iterable) – extend list by appending elements from the iterable If self is frozen, raise ValueError.

insert(**args*, ***kwargs*)
L.insert(index, object) – insert object before index If self is frozen, raise ValueError.

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range. If self is frozen, raise ValueError.

remove(**args*, ***kwargs*)
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present. If self is frozen, raise ValueError.

reverse(**args*, ***kwargs*)
L.reverse() – reverse *IN PLACE* If self is frozen, raise ValueError.

sort(**args*, ***kwargs*)
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1 If self is frozen, raise ValueError.

nltk.featstruct.**unify**(*fstruct1*, *fstruct2*, *bindings=None*, *trace=False*, *fail=None*, *rename_vars=True*, *fs_class=u'default'*)

Unify `fstruct1` with `fstruct2`, and return the resulting feature structure. This unified feature structure is the minimal feature structure that contains all feature value assignments from both `fstruct1` and `fstruct2`, and that preserves all reentrancies.

If no such feature structure exists (because `fstruct1` and `fstruct2` specify incompatible values for some feature), then unification fails, and `unify` returns None.

Bound variables are replaced by their values. Aliased variables are replaced by their representative variable (if unbound) or the value of their representative variable (if bound). I.e., if variable *v* is in `bindings`, then *v* is replaced by `bindings[v]`. This will be repeated until the variable is replaced by an unbound variable or a non-variable value.

Unbound variables are bound when they are unified with values; and aliased when they are unified with variables. I.e., if variable *v* is not in `bindings`, and is unified with a variable or value *x*, then `bindings[v]` is set to *x*.

If `bindings` is unspecified, then all variables are assumed to be unbound. I.e., `bindings` defaults to an empty dict.

```
>>> from nltk.featstruct import FeatStruct
>>> FeatStruct('[a=?x]').unify(FeatStruct('[b=?x]'))
[a=?x, b=?x2]
```

Parameters

- **bindings** (*dict(Variable -> any)*) – A set of variable bindings to be used and updated during unification.
- **trace** (*bool*) – If true, generate trace output.
- **rename_vars** (*bool*) – If True, then rename any variables in `fstruct2` that are also used in `fstruct1`, in order to avoid collisions on variable names.

nltk.featstruct.**subsumes**(*fstruct1*, *fstruct2*)
Return True if fstruct1 subsumes fstruct2. I.e., return true if unifying fstruct1 with fstruct2 would result in a feature structure equal to fstruct2.

Return type bool

nltk.featstruct.**conflicts**(*fstruct1*, *fstruct2*, *trace=0*)
Return a list of the feature paths of all features which are assigned incompatible values by fstruct1 and fstruct2.

Return type list(tuple)

class nltk.featstruct.**Feature**(*name*, *default=None*, *display=None*)
Bases: object

A feature identifier that's specialized to put additional constraints, default values, etc.

default
Default value for this feature.

display
Custom display location: can be prefix, or slash.

name
The name of this feature.

read_value(*s*, *position*, *reentrances*, *parser*)

unicode_repr()

unify_base_values(*fval1*, *fval2*, *bindings*)
If possible, return a single value.. If not, return the value UnificationFailure.

class nltk.featstruct.**SlashFeature**(*name*, *default=None*, *display=None*)
Bases: *nltk.featstruct.Feature*

read_value(*s*, *position*, *reentrances*, *parser*)

class nltk.featstruct.**RangeFeature**(*name*, *default=None*, *display=None*)
Bases: *nltk.featstruct.Feature*

RANGE_RE = <_sre.SRE_Pattern object>

read_value(*s*, *position*, *reentrances*, *parser*)

unify_base_values(*fval1*, *fval2*, *bindings*)

class nltk.featstruct.**FeatStructReader**(*features=(*slash*, *type*)*, *fdict_class=<class 'nltk.featstruct.FeatStruct'>*, *flist_class=<class 'nltk.featstruct.FeatList'>*, *logic_parser=None*)
Bases: object

VALUE_HANDLERS = [(u'read_fstruct_value', <_sre.SRE_Pattern object at 0x7f7480a9ed70>), (u'read_var_value', <_sre.

fromstring(*s*, *fstruct=None*)
Convert a string representation of a feature structure (as displayed by repr) into a FeatStruct. This process imposes the following restrictions on the string representation:

- Feature names cannot contain any of the following: whitespace, parentheses, quote marks, equals signs, dashes, commas, and square brackets. Feature names may not begin with plus signs or minus signs.
- Only the following basic feature value are supported: strings, integers, variables, None, and unquoted alphanumeric strings.

- For reentrant values, the first mention must specify a reentrance identifier and a value; and any subsequent mentions must use arrows (`'->'`) to reference the reentrance identifier.

read_app_value(*s*, *position*, *reentrances*, *match*)
Mainly included for backwards compat.

read_fstruct_value(*s*, *position*, *reentrances*, *match*)

read_int_value(*s*, *position*, *reentrances*, *match*)

read_logic_value(*s*, *position*, *reentrances*, *match*)

read_partial(*s*, *position=0*, *reentrances=None*, *fstruct=None*)
Helper function that reads in a feature structure.

Parameters

- **s** – The string to read.
- **position** – The position in the string to start parsing.
- **reentrances** – A dictionary from reentrance ids to values. Defaults to an empty dictionary.

Returns A tuple (val, pos) of the feature structure created by parsing and the position where the parsed feature structure ends.

Return type bool

read_set_value(*s*, *position*, *reentrances*, *match*)

read_str_value(*s*, *position*, *reentrances*, *match*)

read_sym_value(*s*, *position*, *reentrances*, *match*)

read_tuple_value(*s*, *position*, *reentrances*, *match*)

read_value(*s*, *position*, *reentrances*)

read_var_value(*s*, *position*, *reentrances*, *match*)

3.5.7 grammar Module

Basic data classes for representing context free grammars. A "grammar" specifies which trees can represent the structure of a given text. Each of these trees is called a "parse tree" for the text (or simply a "parse"). In a "context free" grammar, the set of parse trees for any piece of a text can depend only on that piece, and not on the rest of the text (i.e., the piece's context). Context free grammars are often used to find possible syntactic structures for sentences. In this context, the leaves of a parse tree are word tokens; and the node values are phrasal categories, such as `NP` and `VP`.

The `CFG` class is used to encode context free grammars. Each `CFG` consists of a start symbol and a set of productions. The "start symbol" specifies the root node value for parse trees. For example, the start symbol for syntactic parsing is usually `S`. Start symbols are encoded using the `Nonterminal` class, which is discussed below.

A Grammar's "productions" specify what parent-child relationships a parse tree can contain. Each production specifies that a particular node can be the parent of a particular set of children. For example, the production `<S> -> <NP> <VP>` specifies that an `S` node can be the parent of an `NP` node and a `VP` node.

Grammar productions are implemented by the `Production` class. Each `Production` consists of a left hand side and a right hand side. The "left hand side" is a `Nonterminal` that specifies the node type for a potential parent; and the "right hand side" is a list that specifies allowable children for that parent. This lists consists of `Nonterminals` and text types: each `Nonterminal` indicates that the corresponding child may be a `TreeToken` with the specified node type; and each text type indicates that the corresponding child may be a `Token` with the with that type.

The Nonterminal class is used to distinguish node values from leaf values. This prevents the grammar from accidentally using a leaf value (such as the English word "A") as the node of a subtree. Within a CFG, all node values are wrapped in the Nonterminal class. Note, however, that the trees that are specified by the grammar do *not* include these Nonterminal wrappers.

Grammars can also be given a more procedural interpretation. According to this interpretation, a Grammar specifies any tree structure *tree* that can be produced by the following procedure:

Set tree to the start symbol
Repeat until tree contains no more nonterminal leaves:
    Choose a production prod with whose left hand side
        lhs is a nonterminal leaf of tree.
    Replace the nonterminal leaf with a subtree, whose node
        value is the value wrapped by the nonterminal lhs, and
        whose children are the right hand side of prod.

The operation of replacing the left hand side (*lhs*) of a production with the right hand side (*rhs*) in a tree (*tree*) is known as "expanding" *lhs* to *rhs* in *tree*.

class nltk.grammar.**Nonterminal**(*symbol*)
Bases: object

A non-terminal symbol for a context free grammar. Nonterminal is a wrapper class for node values; it is used by Production objects to distinguish node values from leaf values. The node value that is wrapped by a Nonterminal is known as its "symbol". Symbols are typically strings representing phrasal categories (such as "NP" or "VP"). However, more complex symbol types are sometimes used (e.g., for lexicalized grammars). Since symbols are node values, they must be immutable and hashable. Two Nonterminals are considered equal if their symbols are equal.

See CFG, Production

Variables **_symbol** – The node value corresponding to this Nonterminal. This value must be immutable and hashable.

symbol()
Return the node value corresponding to this Nonterminal.

Return type (any)

unicode_repr()
Return a string representation for this Nonterminal.

Return type str

nltk.grammar.**nonterminals**(*symbols*)
Given a string containing a list of symbol names, return a list of Nonterminals constructed from those symbols.

Parameters **symbols** (*str*) – The symbol name string. This string can be delimited by either spaces or commas.

Returns A list of Nonterminals constructed from the symbol names given in symbols. The Nonterminals are sorted in the same order as the symbols names.

Return type list(Nonterminal)

class nltk.grammar.**CFG**(*start*, *productions*, *calculate_leftcorners=True*)
Bases: object

A context-free grammar. A grammar consists of a start state and a set of productions. The set of terminals and nonterminals is implicitly specified by the productions.

If you need efficient key-based access to productions, you can use a subclass to implement it.

check_coverage(*tokens*)
Check whether the grammar rules cover the given list of tokens. If not, then raise an exception.

classmethod fromstring(*input*, *encoding=None*)
Return the CFG corresponding to the input string(s).

Parameters **input** – a grammar, either in the form of a string or as a list of strings.

is_binarised()
Return True if all productions are at most binary. Note that there can still be empty and unary productions.

is_chomsky_normal_form()
Return True if the grammar is of Chomsky Normal Form, i.e. all productions are of the form A -> B C, or A -> "s".

is_flexible_chomsky_normal_form()
Return True if all productions are of the forms A -> B C, A -> B, or A -> "s".

is_leftcorner(*cat*, *left*)
True if left is a leftcorner of cat, where left can be a terminal or a nonterminal.

Parameters

- **cat** (Nonterminal) – the parent of the leftcorner
- **left** (*Terminal or Nonterminal*) – the suggested leftcorner

Return type bool

is_lexical()
Return True if all productions are lexicalised.

is_nonempty()
Return True if there are no empty productions.

is_nonlexical()
Return True if all lexical rules are "preterminals", that is, unary rules which can be separated in a preprocessing step.

This means that all productions are of the forms A -> B1 ... Bn (n>=0), or A -> "s".

Note: is_lexical() and is_nonlexical() are not opposites. There are grammars which are neither, and grammars which are both.

leftcorner_parents(*cat*)
Return the set of all nonterminals for which the given category is a left corner. This is the inverse of the leftcorner relation.

Parameters **cat** (Nonterminal) – the suggested leftcorner

Returns the set of all parents to the leftcorner

Return type set(Nonterminal)

leftcorners(*cat*)
Return the set of all nonterminals that the given nonterminal can start with, including itself.

This is the reflexive, transitive closure of the immediate leftcorner relation: (A > B) iff (A -> B beta)

Parameters **cat** (Nonterminal) – the parent of the leftcorners

Returns the set of all leftcorners

Return type set(Nonterminal)

max_len()
Return the right-hand side length of the longest grammar production.

min_len()
Return the right-hand side length of the shortest grammar production.

productions(*lhs=None*, *rhs=None*, *empty=False*)
Return the grammar productions, filtered by the left-hand side or the first item in the right-hand side.

Parameters

- **lhs** – Only return productions with the given left-hand side.
- **rhs** – Only return productions with the given first item in the right-hand side.
- **empty** – Only return productions with an empty right-hand side.

Returns A list of productions matching the given constraints.

Return type list(Production)

start()
Return the start symbol of the grammar

Return type *Nonterminal*

unicode_repr()

class nltk.grammar.**Production**(*lhs*, *rhs*)
Bases: object

A grammar production. Each production maps a single symbol on the "left-hand side" to a sequence of symbols on the "right-hand side". (In the case of context-free productions, the left-hand side must be a Nonterminal, and the right-hand side is a sequence of terminals and Nonterminals.) "terminals" can be any immutable hashable object that is not a Nonterminal. Typically, terminals are strings representing words, such as "dog" or "under".

See CFG

See DependencyGrammar

See Nonterminal

Variables

- **_lhs** – The left-hand side of the production.
- **_rhs** – The right-hand side of the production.

is_lexical()
Return True if the right-hand contain at least one terminal token.

Return type bool

is_nonlexical()
Return True if the right-hand side only contains Nonterminals

Return type bool

lhs()
Return the left-hand side of this Production.

Return type *Nonterminal*

rhs()
Return the right-hand side of this Production.

Return type sequence(Nonterminal and terminal)

unicode_repr()
Return a concise string representation of the Production.

Return type str

class nltk.grammar.**PCFG**(*start*, *productions*, *calculate_leftcorners=True*)
Bases: *nltk.grammar.CFG*

A probabilistic context-free grammar. A PCFG consists of a start state and a set of productions with probabilities. The set of terminals and nonterminals is implicitly specified by the productions.

PCFG productions use the ProbabilisticProduction class. PCFGs impose the constraint that the set of productions with any given left-hand-side must have probabilities that sum to 1 (allowing for a small margin of error).

If you need efficient key-based access to productions, you can use a subclass to implement it.

Variables ***EPSILON*** – The acceptable margin of error for checking that productions with a given left-hand side have probabilities that sum to 1.

EPSILON = 0.01

classmethod fromstring(*input*, *encoding=None*)
Return a probabilistic PCFG corresponding to the input string(s).

Parameters **input** – a grammar, either in the form of a string or else as a list of strings.

class nltk.grammar.**ProbabilisticProduction**(*lhs*, *rhs*, ***prob*)
Bases: *nltk.grammar.Production*, *nltk.probability.ImmutableProbabilisticMixIn*

A probabilistic context free grammar production. A PCFG ProbabilisticProduction is essentially just a Production that has an associated probability, which represents how likely it is that this production will be used. In particular, the probability of a ProbabilisticProduction records the likelihood that its right-hand side is the correct instantiation for any given occurrence of its left-hand side.

See Production

class nltk.grammar.**DependencyGrammar**(*productions*)
Bases: object

A dependency grammar. A DependencyGrammar consists of a set of productions. Each production specifies a head/modifier relationship between a pair of words.

contains(*head*, *mod*)

Parameters

- **head** (*str*) – A head word.
- **mod** (*str*) – A mod word, to test as a modifier of 'head'.

Returns true if this DependencyGrammar contains a DependencyProduction mapping 'head' to 'mod'.

Return type bool

classmethod fromstring(*input*)

unicode_repr()
Return a concise string representation of the DependencyGrammar

class nltk.grammar.**DependencyProduction**(*lhs*, *rhs*)
Bases: *nltk.grammar.Production*

A dependency grammar production. Each production maps a single head word to an unordered list of one or more modifier words.

class nltk.grammar.**ProbabilisticDependencyGrammar**(*productions*, *events*, *tags*)
Bases: object

contains(*head*, *mod*)
Return True if this DependencyGrammar contains a DependencyProduction mapping 'head' to 'mod'.

Parameters

- **head** (*str*) – A head word.
- **mod** (*str*) – A mod word, to test as a modifier of 'head'.

Return type bool

unicode_repr()
Return a concise string representation of the ProbabilisticDependencyGrammar

nltk.grammar.**induce_pcfg**(*start*, *productions*)
Induce a PCFG grammar from a list of productions.

The probability of a production A -> B C in a PCFG is:

count(A -> B C)
P(B, C | A) = ———————— where * is any right hand side
count(A -> *)

Parameters

- **start** (Nonterminal) – The start symbol
- **productions** (*list(Production)*) – The list of productions that defines the grammar

nltk.grammar.**read_grammar**(*input*, *nonterm_parser*, *probabilistic=False*, *encoding=None*)
Return a pair consisting of a starting category and a list of Productions.

Parameters

- **input** – a grammar, either in the form of a string or else as a list of strings.
- **nonterm_parser** – a function for parsing nonterminals. It should take a (string, position) as argument and return a (nonterminal, position) as result.
- **probabilistic** (*bool*) – are the grammar rules probabilistic?
- **encoding** (*str*) – the encoding of the grammar, if it is a binary string

3.5.8 help Module

Provide structured access to documentation.

nltk.help.**brown_tagset**(*tagpattern=None*)

nltk.help.**claws5_tagset**(*tagpattern=None*)

nltk.help.**upenn_tagset**(*tagpattern=None*)

3.5.9 probability Module

Classes for representing and processing probabilistic information.

The FreqDist class is used to encode "frequency distributions", which count the number of times that each outcome of an experiment occurs.

The ProbDistI class defines a standard interface for "probability distributions", which encode the probability of each outcome for an experiment. There are two types of probability distribution:

- "derived probability distributions" are created from frequency distributions. They attempt to model the probability distribution that generated the frequency distribution.
- "analytic probability distributions" are created directly from parameters (such as variance).

The ConditionalFreqDist class and ConditionalProbDistI interface are used to encode conditional distributions. Conditional probability distributions can be derived or analytic; but currently the only implementation of the ConditionalProbDistI interface is ConditionalProbDist, a derived distribution.

class nltk.probability.**ConditionalFreqDist**(*cond_samples=None*)

Bases: collections.defaultdict

A collection of frequency distributions for a single experiment run under different conditions. Conditional frequency distributions are used to record the number of times each sample occurred, given the condition under which the experiment was run. For example, a conditional frequency distribution could be used to record the frequency of each word (type) in a document, given its length. Formally, a conditional frequency distribution can be defined as a function that maps from each condition to the FreqDist for the experiment under that condition.

Conditional frequency distributions are typically constructed by repeatedly running an experiment under a variety of conditions, and incrementing the sample outcome counts for the appropriate conditions. For example, the following code will produce a conditional frequency distribution that encodes how often each word type occurs, given the length of that word type:

```
>>> from nltk.probability import ConditionalFreqDist
>>> from nltk.tokenize import word_tokenize
>>> sent = "the the the dog dog some other words that we do not care about"
>>> cfdist = ConditionalFreqDist()
>>> for word in word_tokenize(sent):
...     condition = len(word)
...     cfdist[condition][word] += 1
```

An equivalent way to do this is with the initializer:

```
>>> cfdist = ConditionalFreqDist((len(word), word) for word in word_tokenize(sent))
```

The frequency distribution for each condition is accessed using the indexing operator:

```
>>> cfdist[3]
FreqDist({'the': 3, 'dog': 2, 'not': 1})
>>> cfdist[3].freq('the')
0.5
>>> cfdist[3]['dog']
2
```

When the indexing operator is used to access the frequency distribution for a condition that has not been accessed before, ConditionalFreqDist creates a new empty FreqDist for that condition.

N()
Return the total number of sample outcomes that have been recorded by this ConditionalFreqDist.

Return type int

conditions()
Return a list of the conditions that have been accessed for this ConditionalFreqDist. Use the indexing operator to access the frequency distribution for a given condition. Note that the frequency distributions for some conditions may contain zero sample outcomes.

Return type *list*

plot(**args*, ***kwargs*)
Plot the given samples from the conditional frequency distribution. For a cumulative plot, specify cumulative=True. (Requires Matplotlib to be installed.)

Parameters

- **samples** (list) – The samples to plot
- **title** (*str*) – The title for the graph
- **conditions** (list) – The conditions to plot (default is all)

tabulate(**args*, ***kwargs*)
Tabulate the given samples from the conditional frequency distribution.

Parameters

- **samples** (list) – The samples to plot
- **title** (*str*) – The title for the graph
- **conditions** (list) – The conditions to plot (default is all)

unicode_repr()
Return a string representation of this ConditionalFreqDist.

Return type str

class nltk.probability.**ConditionalProbDist**(*cfdist*, *probdist_factory*, **factory_args*, ***factory_kw_args*)
Bases: *nltk.probability.ConditionalProbDistI*

A conditional probability distribution modeling the experiments that were used to generate a conditional frequency distribution. A ConditionalProbDist is constructed from a ConditionalFreqDist and a ProbDist factory:

- The ConditionalFreqDist specifies the frequency distribution for each condition.
- The ProbDist factory is a function that takes a condition's frequency distribution, and returns its probability distribution. A ProbDist class's name (such as MLEProbDist or HeldoutProbDist) can be used to specify that class's constructor.

The first argument to the ProbDist factory is the frequency distribution that it should model; and the remaining arguments are specified by the factory_args parameter to the ConditionalProbDist constructor. For example, the following code constructs a ConditionalProbDist, where the probability distribution for each condition is an ELEProbDist with 10 bins:

```
>>> from nltk.corpus import brown
>>> from nltk.probability import ConditionalFreqDist
>>> from nltk.probability import ConditionalProbDist, ELEProbDist
>>> cfdist = ConditionalFreqDist(brown.tagged_words()[:5000])
>>> cpdist = ConditionalProbDist(cfdist, ELEProbDist, 10)
>>> cpdist['passed'].max()
```

```
    'VBD'
    >>> cpdist['passed'].prob('VBD')
    0.423...
```

class nltk.probability.**ConditionalProbDistI**
Bases: dict

A collection of probability distributions for a single experiment run under different conditions. Conditional probability distributions are used to estimate the likelihood of each sample, given the condition under which the experiment was run. For example, a conditional probability distribution could be used to estimate the probability of each word type in a document, given the length of the word type. Formally, a conditional probability distribution can be defined as a function that maps from each condition to the ProbDist for the experiment under that condition.

conditions()
Return a list of the conditions that are represented by this ConditionalProbDist. Use the indexing operator to access the probability distribution for a given condition.

Return type *list*

unicode_repr()
Return a string representation of this ConditionalProbDist.

Return type str

class nltk.probability.**CrossValidationProbDist**(*freqdists*, *bins*)
Bases: *nltk.probability.ProbDistI*

The cross-validation estimate for the probability distribution of the experiment used to generate a set of frequency distribution. The "cross-validation estimate" for the probability of a sample is found by averaging the held-out estimates for the sample in each pair of frequency distributions.

SUM_TO_ONE = False

discount()

freqdists()
Return the list of frequency distributions that this ProbDist is based on.

Return type list(FreqDist)

prob(*sample*)

samples()

unicode_repr()
Return a string representation of this ProbDist.

Return type str

class nltk.probability.**DictionaryConditionalProbDist**(*probdist_dict*)
Bases: *nltk.probability.ConditionalProbDistI*

An alternative ConditionalProbDist that simply wraps a dictionary of ProbDists rather than creating these from FreqDists.

class nltk.probability.**DictionaryProbDist**(*prob_dict=None*, *log=False*, *normalize=False*)
Bases: *nltk.probability.ProbDistI*

A probability distribution whose probabilities are directly specified by a given dictionary. The given dictionary maps samples to probabilities.

logprob(*sample*)

max()

prob(*sample*)

samples()

unicode_repr()

class nltk.probability.**ELEProbDist**(*freqdist*, *bins=None*)
Bases: *nltk.probability.LidstoneProbDist*

The expected likelihood estimate for the probability distribution of the experiment used to generate a frequency distribution. The "expected likelihood estimate" approximates the probability of a sample with count *c* from an experiment with *N* outcomes and *B* bins as *(c+0.5)/(N+B/2)*. This is equivalent to adding 0.5 to the count for each bin, and taking the maximum likelihood estimate of the resulting frequency distribution.

unicode_repr()
Return a string representation of this ProbDist.

Return type str

class nltk.probability.**FreqDist**(*samples=None*)
Bases: collections.Counter

A frequency distribution for the outcomes of an experiment. A frequency distribution records the number of times each outcome of an experiment has occurred. For example, a frequency distribution could be used to record the frequency of each word type in a document. Formally, a frequency distribution can be defined as a function mapping from each sample to the number of times that sample occurred as an outcome.

Frequency distributions are generally constructed by running a number of experiments, and incrementing the count for a sample every time it is an outcome of an experiment. For example, the following code will produce a frequency distribution that encodes how often each word occurs in a text:

```
>>> from nltk.tokenize import word_tokenize
>>> from nltk.probability import FreqDist
>>> sent = 'This is an example sentence'
>>> fdist = FreqDist()
>>> for word in word_tokenize(sent):
...     fdist[word.lower()] += 1
```

An equivalent way to do this is with the initializer:

```
>>> fdist = FreqDist(word.lower() for word in word_tokenize(sent))
```

B()
Return the total number of sample values (or "bins") that have counts greater than zero. For the total number of sample outcomes recorded, use FreqDist.N(). (FreqDist.B() is the same as len(FreqDist).)

Return type int

N()
Return the total number of sample outcomes that have been recorded by this FreqDist. For the number of unique sample values (or bins) with counts greater than zero, use FreqDist.B().

Return type int

Nr(*r*, *bins=None*)

copy()
Create a copy of this frequency distribution.

Return type *FreqDist*

freq(*sample*)
Return the frequency of a given sample. The frequency of a sample is defined as the count of that sample divided by the total number of sample outcomes that have been recorded by this FreqDist. The count of a

sample is defined as the number of times that sample outcome was recorded by this FreqDist. Frequencies are always real numbers in the range [0, 1].

Parameters **sample** (*any*) – the sample whose frequency should be returned.

Return type float

hapaxes()
Return a list of all samples that occur once (hapax legomena)

Return type *list*

max()
Return the sample with the greatest number of outcomes in this frequency distribution. If two or more samples have the same number of outcomes, return one of them; which sample is returned is undefined. If no outcomes have occurred in this frequency distribution, return None.

Returns The sample with the maximum number of outcomes in this frequency distribution.

Return type any or None

pformat(*maxlen=10*)
Return a string representation of this FreqDist.

Parameters **maxlen** (*int*) – The maximum number of items to display

Return type string

plot(**args*, ***kwargs*)
Plot samples from the frequency distribution displaying the most frequent sample first. If an integer parameter is supplied, stop after this many samples have been plotted. For a cumulative plot, specify cumulative=True. (Requires Matplotlib to be installed.)

Parameters

- **title** (*bool*) – The title for the graph
- **cumulative** – A flag to specify whether the plot is cumulative (default = False)

pprint(*maxlen=10*, *stream=None*)
Print a string representation of this FreqDist to 'stream'

Parameters

- **maxlen** (*int*) – The maximum number of items to print
- **stream** – The stream to print to. stdout by default

r_Nr(*bins=None*)
Return the dictionary mapping r to Nr, the number of samples with frequency r, where Nr > 0.

Parameters **bins** (*int*) – The number of possible sample outcomes. `bins` is used to calculate Nr(0). In particular, Nr(0) is `bins-self.B()`. If `bins` is not specified, it defaults to `self.B()` (so Nr(0) will be 0).

Return type int

tabulate(**args*, ***kwargs*)
Tabulate the given samples from the frequency distribution (cumulative), displaying the most frequent sample first. If an integer parameter is supplied, stop after this many samples have been plotted.

Parameters **samples** (list) – The samples to plot (default is all samples)

unicode_repr()
Return a string representation of this FreqDist.

Return type string

class nltk.probability.**SimpleGoodTuringProbDist**(*freqdist*, *bins=None*)
Bases: *nltk.probability.ProbDistI*

SimpleGoodTuring ProbDist approximates from frequency to frequency of frequency into a linear line under log space by linear regression. Details of Simple Good-Turing algorithm can be found in:

- Good Turing smoothing without tears" (Gale & Sampson 1995), Journal of Quantitative Linguistics, vol. 2 pp. 217-237.
- "Speech and Language Processing (Jurafsky & Martin), 2nd Edition, Chapter 4.5 p103 (log(Nc) = a + b*log(c))
- http://www.grsampson.net/RGoodTur.html

Given a set of pair (xi, yi), where the xi denotes the frequency and yi denotes the frequency of frequency, we want to minimize their square variation. E(x) and E(y) represent the mean of xi and yi.

- slope: b = sigma ((xi-E(x)(yi-E(y))) / sigma ((xi-E(x))(xi-E(x)))
- intercept: a = E(y) - b.E(x)

SUM_TO_ONE = False

check()

discount()
This function returns the total mass of probability transfers from the seen samples to the unseen samples.

find_best_fit(*r*, *nr*)
Use simple linear regression to tune parameters self._slope and self._intercept in the log-log space based on count and Nr(count) (Work in log space to avoid floating point underflow.)

freqdist()

max()

prob(*sample*)
Return the sample's probability.

Parameters **sample** (*str*) – sample of the event

Return type float

samples()

smoothedNr(*r*)
Return the number of samples with count r.

Parameters **r** (*int*) – The amount of frequency.

Return type float

unicode_repr()
Return a string representation of this ProbDist.

Return type str

class nltk.probability.**HeldoutProbDist**(*base_fdist*, *heldout_fdist*, *bins=None*)
Bases: *nltk.probability.ProbDistI*

The heldout estimate for the probability distribution of the experiment used to generate two frequency distributions. These two frequency distributions are called the "heldout frequency distribution" and the "base frequency distribution." The "heldout estimate" uses uses the "heldout frequency distribution" to predict the probability of each sample, given its frequency in the "base frequency distribution".

In particular, the heldout estimate approximates the probability for a sample that occurs *r* times in the base distribution as the average frequency in the heldout distribution of all samples that occur *r* times in the base distribution.

This average frequency is *Tr[r]/(Nr[r].N)*, where:

- *Tr[r]* is the total count in the heldout distribution for all samples that occur *r* times in the base distribution.
- *Nr[r]* is the number of samples that occur *r* times in the base distribution.
- *N* is the number of outcomes recorded by the heldout frequency distribution.

In order to increase the efficiency of the `prob` member function, *Tr[r]/(Nr[r].N)* is precomputed for each value of *r* when the `HeldoutProbDist` is created.

Variables

- **`_estimate`** – A list mapping from *r*, the number of times that a sample occurs in the base distribution, to the probability estimate for that sample. `_estimate[r]` is calculated by finding the average frequency in the heldout distribution of all samples that occur *r* times in the base distribution. In particular, `_estimate[r]` = *Tr[r]/(Nr[r].N)*.
- **`_max_r`** – The maximum number of times that any sample occurs in the base distribution. `_max_r` is used to decide how large `_estimate` must be.

`SUM_TO_ONE` = False

`base_fdist`()
Return the base frequency distribution that this probability distribution is based on.

Return type *FreqDist*

`discount`()

`heldout_fdist`()
Return the heldout frequency distribution that this probability distribution is based on.

Return type *FreqDist*

`max`()

`prob`(*sample*)

`samples`()

`unicode_repr`()

Return type str

Returns A string representation of this `ProbDist`.

class `nltk.probability.`**`ImmutableProbabilisticMixIn`**(***kwargs*)
Bases: *nltk.probability.ProbabilisticMixIn*

`set_logprob`(*prob*)

`set_prob`(*prob*)

class `nltk.probability.`**`LaplaceProbDist`**(*freqdist*, *bins=None*)
Bases: *nltk.probability.LidstoneProbDist*

The Laplace estimate for the probability distribution of the experiment used to generate a frequency distribution. The "Laplace estimate" approximates the probability of a sample with count *c* from an experiment with *N* outcomes and *B* bins as *(c+1)/(N+B)*. This is equivalent to adding one to the count for each bin, and taking the maximum likelihood estimate of the resulting frequency distribution.

`unicode_repr`()

Return type str

Returns A string representation of this `ProbDist`.

class `nltk.probability.`**LidstoneProbDist**(*freqdist*, *gamma*, *bins=None*)
Bases: *nltk.probability.ProbDistI*

The Lidstone estimate for the probability distribution of the experiment used to generate a frequency distribution. The "Lidstone estimate" is parameterized by a real number *gamma*, which typically ranges from 0 to 1. The Lidstone estimate approximates the probability of a sample with count *c* from an experiment with *N* outcomes and *B* bins as `c+gamma)/(N+B*gamma)`. This is equivalent to adding *gamma* to the count for each bin, and taking the maximum likelihood estimate of the resulting frequency distribution.

SUM_TO_ONE = False

discount()

freqdist()
Return the frequency distribution that this probability distribution is based on.

Return type *FreqDist*

max()

prob(*sample*)

samples()

unicode_repr()
Return a string representation of this `ProbDist`.

Return type str

class `nltk.probability.`**MLEProbDist**(*freqdist*, *bins=None*)
Bases: *nltk.probability.ProbDistI*

The maximum likelihood estimate for the probability distribution of the experiment used to generate a frequency distribution. The "maximum likelihood estimate" approximates the probability of each sample as the frequency of that sample in the frequency distribution.

freqdist()
Return the frequency distribution that this probability distribution is based on.

Return type *FreqDist*

max()

prob(*sample*)

samples()

unicode_repr()

Return type str

Returns A string representation of this `ProbDist`.

class `nltk.probability.`**MutableProbDist**(*prob_dist*, *samples*, *store_logs=True*)
Bases: *nltk.probability.ProbDistI*

An mutable probdist where the probabilities may be easily modified. This simply copies an existing probdist, storing the probability values in a mutable dictionary and providing an update method.

logprob(*sample*)

prob(*sample*)

samples()

update(*sample*, *prob*, *log=True*)
Update the probability for the given sample. This may cause the object to stop being the valid probability distribution - the user must ensure that they update the sample probabilities such that all samples have probabilities between 0 and 1 and that all probabilities sum to one.

Parameters

- **sample** (*any*) – the sample for which to update the probability
- **prob** (*float*) – the new probability
- **log** (*bool*) – is the probability already logged

class nltk.probability.**KneserNeyProbDist**(*freqdist*, *bins=None*, *discount=0.75*)
Bases: *nltk.probability.ProbDistI*

Kneser-Ney estimate of a probability distribution. This is a version of back-off that counts how likely an n-gram is provided the n-1-gram had been seen in training. Extends the ProbDistI interface, requires a trigram FreqDist instance to train on. Optionally, a different from default discount value can be specified. The default discount is set to 0.75.

discount()
Return the value by which counts are discounted. By default set to 0.75.

Return type float

max()

prob(*trigram*)

samples()

set_discount(*discount*)
Set the value by which counts are discounted to the value of discount.

Parameters **discount** (*float (preferred, but int possible)*) – the new value to discount counts by

Return type None

unicode_repr()
Return a string representation of this ProbDist

Return type str

class nltk.probability.**ProbDistI**
Bases: object

A probability distribution for the outcomes of an experiment. A probability distribution specifies how likely it is that an experiment will have any given outcome. For example, a probability distribution could be used to predict the probability that a token in a document will have a given type. Formally, a probability distribution can be defined as a function mapping from samples to nonnegative real numbers, such that the sum of every number in the function's range is 1.0. A ProbDist is often used to model the probability distribution of the experiment used to generate a frequency distribution.

SUM_TO_ONE = True

discount()
Return the ratio by which counts are discounted on average: c*/c

Return type float

generate()
Return a randomly selected sample from this probability distribution. The probability of returning each sample `samp` is equal to `self.prob(samp)`.

logprob(*sample*)
Return the base 2 logarithm of the probability for a given sample.

Parameters **sample** (*any*) – The sample whose probability should be returned.

Return type float

max()
Return the sample with the greatest probability. If two or more samples have the same probability, return one of them; which sample is returned is undefined.

Return type any

prob(*sample*)
Return the probability for a given sample. Probabilities are always real numbers in the range [0, 1].

Parameters **sample** (*any*) – The sample whose probability should be returned.

Return type float

samples()
Return a list of all samples that have nonzero probabilities. Use `prob` to find the probability of each sample.

Return type *list*

class nltk.probability.**ProbabilisticMixIn**(***kwargs*)
Bases: `object`

A mix-in class to associate probabilities with other classes (trees, rules, etc.). To use the `ProbabilisticMixIn` class, define a new class that derives from an existing class and from ProbabilisticMixIn. You will need to define a new constructor for the new class, which explicitly calls the constructors of both its parent classes. For example:

```
>>> from nltk.probability import ProbabilisticMixIn
>>> class A:
...     def __init__(self, x, y): self.data = (x,y)
...
>>> class ProbabilisticA(A, ProbabilisticMixIn):
...     def __init__(self, x, y, **prob_kwarg):
...         A.__init__(self, x, y)
...         ProbabilisticMixIn.__init__(self, **prob_kwarg)
```

See the documentation for the ProbabilisticMixIn `constructor<__init__>` for information about the arguments it expects.

You should generally also redefine the string representation methods, the comparison methods, and the hashing method.

logprob()
Return `log(p)`, where `p` is the probability associated with this object.

Return type float

prob()
Return the probability associated with this object.

Return type float

set_logprob(*logprob*)
Set the log probability associated with this object to `logprob`. I.e., set the probability associated with this object to `2**(logprob)`.

Parameters **logprob** (*float*) – The new log probability

set_prob(*prob*)
Set the probability associated with this object to `prob`.

Parameters **prob** (*float*) – The new probability

class nltk.probability.**UniformProbDist**(*samples*)
Bases: *nltk.probability.ProbDistI*

A probability distribution that assigns equal probability to each sample in a given set; and a zero probability to all other samples.

max()

prob(*sample*)

samples()

unicode_repr()

class nltk.probability.**WittenBellProbDist**(*freqdist*, *bins=None*)
Bases: *nltk.probability.ProbDistI*

The Witten-Bell estimate of a probability distribution. This distribution allocates uniform probability mass to as yet unseen events by using the number of events that have only been seen once. The probability mass reserved for unseen events is equal to *T / (N + T)* where *T* is the number of observed event types and *N* is the total number of observed events. This equates to the maximum likelihood estimate of a new type event occurring. The remaining probability mass is discounted such that all probability estimates sum to one, yielding:

- *p = T / Z (N + T)*, if count = 0
- *p = c / (N + T)*, otherwise

discount()

freqdist()

max()

prob(*sample*)

samples()

unicode_repr()
Return a string representation of this `ProbDist`.

Return type str

nltk.probability.**add_logs**(*logx*, *logy*)
Given two numbers `logx` = *log(x)* and `logy` = *log(y)*, return *log(x+y)*. Conceptually, this is the same as returning `log(2**(logx)+2**(logy))`, but the actual implementation avoids overflow errors that could result from direct computation.

nltk.probability.**log_likelihood**(*test_pdist*, *actual_pdist*)

nltk.probability.**sum_logs**(*logs*)

nltk.probability.**entropy**(*pdist*)

3.5.10 `text` Module

This module brings together a variety of NLTK functionality for text analysis, and provides simple, interactive interfaces. Functionality includes: concordancing, collocation discovery, regular expression search over tokenized strings, and distributional similarity.

class `nltk.text.`**`ContextIndex`**(*tokens*, *context_func=None*, *filter=None*, *key=<function <lambda>>*)
Bases: `object`

A bidirectional index between words and their 'contexts' in a text. The context of a word is usually defined to be the words that occur in a fixed window around the word; but other definitions may also be used by providing a custom context function.

`common_contexts`(*words*, *fail_on_unknown=False*)
Find contexts where the specified words can all appear; and return a frequency distribution mapping each context to the number of times that context was used.

Parameters

- **`words`** (*str*) – The words used to seed the similarity search
- **`fail_on_unknown`** – If true, then raise a value error if any of the given words do not occur at all in the index.

`similar_words`(*word*, *n=20*)

`tokens`()

Return type list(str)

Returns The document that this context index was created from.

`word_similarity_dict`(*word*)
Return a dictionary mapping from words to 'similarity scores,' indicating how often these two words occur in the same context.

class `nltk.text.`**`ConcordanceIndex`**(*tokens*, *key=<function <lambda>>*)
Bases: `object`

An index that can be used to look up the offset locations at which a given word occurs in a document.

`offsets`(*word*)

Return type list(int)

Returns A list of the offset positions at which the given word occurs. If a key function was specified for the index, then given word's key will be looked up.

`print_concordance`(*word*, *width=75*, *lines=25*)
Print a concordance for `word` with the specified context window.

Parameters

- **`word`** (*str*) – The target word
- **`width`** (*int*) – The width of each line, in characters (default=80)
- **`lines`** (*int*) – The number of lines to display (default=25)

`tokens`()

Return type list(str)

Returns The document that this concordance index was created from.

`unicode_repr`()

class nltk.text.**TokenSearcher**(*tokens*)
Bases: object

A class that makes it easier to use regular expressions to search over tokenized strings. The tokenized string is converted to a string where tokens are marked with angle brackets – e.g., '<the><window><is><still><open>'. The regular expression passed to the findall() method is modified to treat angle brackets as non-capturing parentheses, in addition to matching the token boundaries; and to have '.' not match the angle brackets.

findall(*regexp*)
Find instances of the regular expression in the text. The text is a list of tokens, and a regexp pattern to match a single token must be surrounded by angle brackets. E.g.

```
>>> from nltk.text import TokenSearcher
>>> print('hack'); from nltk.book import text1, text5, text9
hack...
>>> text5.findall("<.*><.*><bro>")
you rule bro; telling you bro; u twizted bro
>>> text1.findall("<a>(<.*>)<man>")
monied; nervous; dangerous; white; white; white; pious; queer; good;
mature; white; Cape; great; wise; wise; butterless; white; fiendish;
pale; furious; better; certain; complete; dismasted; younger; brave;
brave; brave; brave
>>> text9.findall("<th.*>{3,}")
thread through those; the thought that; that the thing; the thing
that; that that thing; through these than through; them that the;
through the thick; them that they; thought that the
```

Parameters **regexp** (*str*) – A regular expression

class nltk.text.**Text**(*tokens*, *name=None*)
Bases: object

A wrapper around a sequence of simple (string) tokens, which is intended to support initial exploration of texts (via the interactive console). Its methods perform a variety of analyses on the text's contexts (e.g., counting, concordancing, collocation discovery), and display the results. If you wish to write a program which makes use of these analyses, then you should bypass the Text class, and use the appropriate analysis function or class directly instead.

A Text is typically initialized from a given document or corpus. E.g.:

```
>>> import nltk.corpus
>>> from nltk.text import Text
>>> moby = Text(nltk.corpus.gutenberg.words('melville-moby_dick.txt'))
```

collocations(*num=20*, *window_size=2*)
Print collocations derived from the text, ignoring stopwords.

Seealso find_collocations

Parameters

- **num** (*int*) – The maximum number of collocations to print.
- **window_size** (*int*) – The number of tokens spanned by a collocation (default=2)

common_contexts(*words*, *num=20*)
Find contexts where the specified words appear; list most frequent common contexts first.

Parameters

- **word** (*str*) – The word used to seed the similarity search
- **num** (*int*) – The number of words to generate (default=20)

Seealso ContextIndex.common_contexts()

concordance(*word*, *width=79*, *lines=25*)
Print a concordance for `word` with the specified context window. Word matching is not case-sensitive. :seealso: `ConcordanceIndex`

count(*word*)
Count the number of times this word appears in the text.

dispersion_plot(*words*)
Produce a plot showing the distribution of the words through the text. Requires pylab to be installed.

Parameters **words** (*list(str)*) – The words to be plotted

Seealso nltk.draw.dispersion_plot()

findall(*regexp*)
Find instances of the regular expression in the text. The text is a list of tokens, and a regexp pattern to match a single token must be surrounded by angle brackets. E.g.

```
>>> print('hack'); from nltk.book import text1, text5, text9
hack...
>>> text5.findall("<.*><.*><bro>")
you rule bro; telling you bro; u twizted bro
>>> text1.findall("<a>(<.*>)<man>")
monied; nervous; dangerous; white; white; white; pious; queer; good;
mature; white; Cape; great; wise; wise; butterless; white; fiendish;
pale; furious; better; certain; complete; dismasted; younger; brave;
brave; brave; brave
>>> text9.findall("<th.*>{3,}")
thread through those; the thought that; that the thing; the thing
that; that that thing; through these than through; them that the;
through the thick; them that they; thought that the
```

Parameters **regexp** (*str*) – A regular expression

index(*word*)
Find the index of the first occurrence of the word in the text.

plot(**args*)
See documentation for FreqDist.plot() :seealso: nltk.prob.FreqDist.plot()

readability(*method*)

similar(*word*, *num=20*)
Distributional similarity: find other words which appear in the same contexts as the specified word; list most similar words first.

Parameters

- **word** (*str*) – The word used to seed the similarity search
- **num** (*int*) – The number of words to generate (default=20)

Seealso ContextIndex.similar_words()

unicode_repr()

vocab()

Seealso nltk.prob.FreqDist

class nltk.text.**TextCollection**(*source*, *name=None*)
Bases: *nltk.text.Text*

A collection of texts, which can be loaded with list of texts, or with a corpus consisting of one or more texts, and which supports counting, concordancing, collocation discovery, etc. Initialize a TextCollection as follows:

```
>>> import nltk.corpus
>>> from nltk.text import TextCollection
>>> print('hack'); from nltk.book import text1, text2, text3
hack...
>>> gutenberg = TextCollection(nltk.corpus.gutenberg)
>>> mytexts = TextCollection([text1, text2, text3])
```

Iterating over a TextCollection produces all the tokens of all the texts in order.

idf(*term*, *method=None*)
The number of texts in the corpus divided by the number of texts that the term appears in. If a term does not appear in the corpus, 0.0 is returned.

tf(*term*, *text*, *method=None*)
The frequency of the term in text.

tf_idf(*term*, *text*)

3.5.11 toolbox Module

Module for reading, writing and manipulating Toolbox databases and settings files.

class nltk.toolbox.**StandardFormat**(*filename=None*, *encoding=None*)
Bases: object

Class for reading and processing standard format marker files and strings.

close()
Close a previously opened standard format marker file or string.

fields(*strip=True*, *unwrap=True*, *encoding=None*, *errors='strict'*, *unicode_fields=None*)
Return an iterator that returns the next field in a (marker, value) tuple, where marker and value are unicode strings if an encoding was specified in the fields() method. Otherwise they are non-unicode strings.

Parameters

- **strip** (*bool*) – strip trailing whitespace from the last line of each field
- **unwrap** (*bool*) – Convert newlines in a field to spaces.
- **encoding** (*str or None*) – Name of an encoding to use. If it is specified then the fields() method returns unicode strings rather than non unicode strings.
- **errors** (*str*) – Error handling scheme for codec. Same as the decode() builtin string method.
- **unicode_fields** (*sequence*) – Set of marker names whose values are UTF-8 encoded. Ignored if encoding is None. If the whole file is UTF-8 encoded set encoding='utf8' and leave unicode_fields with its default value of None.

Return type iter(tuple(str, str))

open(*sfm_file*)
Open a standard format marker file for sequential reading.

Parameters **sfm_file** (*str*) – name of the standard format marker input file

open_string(*s*)
Open a standard format marker string for sequential reading.

Parameters **s** (*str*) – string to parse as a standard format marker input file

raw_fields()
Return an iterator that returns the next field in a (marker, value) tuple. Linebreaks and trailing white space are preserved except for the final newline in each field.

Return type iter(tuple(str, str))

class nltk.toolbox.**ToolboxData**(*filename=None*, *encoding=None*)
Bases: *nltk.toolbox.StandardFormat*

parse(*grammar=None*, ***kwargs*)

class nltk.toolbox.**ToolboxSettings**
Bases: *nltk.toolbox.StandardFormat*

This class is the base class for settings files.

parse(*encoding=None*, *errors='strict'*, ***kwargs*)
Return the contents of toolbox settings file with a nested structure.

Parameters

- **encoding** (*str*) – encoding used by settings file
- **errors** (*str*) – Error handling scheme for codec. Same as `decode()` builtin method.
- **kwargs** (*dict*) – Keyword arguments passed to `StandardFormat.fields()`

Return type ElementTree._ElementInterface

nltk.toolbox.**add_blank_lines**(*tree*, *blanks_before*, *blanks_between*)
Add blank lines before all elements and subelements specified in blank_before.

Parameters

- **elem** (*ElementTree._ElementInterface*) – toolbox data in an elementtree structure
- **blank_before** (*dict(tuple)*) – elements and subelements to add blank lines before

nltk.toolbox.**add_default_fields**(*elem*, *default_fields*)
Add blank elements and subelements specified in default_fields.

Parameters

- **elem** (*ElementTree._ElementInterface*) – toolbox data in an elementtree structure
- **default_fields** (*dict(tuple)*) – fields to add to each type of element and subelement

nltk.toolbox.**demo**()

nltk.toolbox.**remove_blanks**(*elem*)
Remove all elements and subelements with no text and no child elements.

Parameters **elem** (*ElementTree._ElementInterface*) – toolbox data in an elementtree structure

nltk.toolbox.**sort_fields**(*elem*, *field_orders*)
Sort the elements and subelements in order specified in field_orders.

Parameters

- **elem** (*ElementTree._ElementInterface*) – toolbox data in an elementtree structure

- **field_orders** (*dict(tuple)*) – order of fields for each type of element and subelement

nltk.toolbox.**to_settings_string**(*tree*, *encoding=None*, *errors='strict'*, *unicode_fields=None*)

nltk.toolbox.**to_sfm_string**(*tree*, *encoding=None*, *errors='strict'*, *unicode_fields=None*)
Return a string with a standard format representation of the toolbox data in tree (tree can be a toolbox database or a single record).

Parameters

- **tree** (*ElementTree._ElementInterface*) – flat representation of toolbox data (whole database or single record)
- **encoding** (*str*) – Name of an encoding to use.
- **errors** (*str*) – Error handling scheme for codec. Same as the `encode()` builtin string method.
- **unicode_fields** (*dict(str) or set(str)*) –

Return type str

3.5.12 tree Module

Class for representing hierarchical language structures, such as syntax trees and morphological trees.

class nltk.tree.**ImmutableProbabilisticTree**(*node*, *children=None*, ***prob_kwargs*)
Bases: *nltk.tree.ImmutableTree*, *nltk.probability.ProbabilisticMixIn*

classmethod convert(*val*)

copy(*deep=False*)

unicode_repr()

class nltk.tree.**ImmutableTree**(*node*, *children=None*)
Bases: *nltk.tree.Tree*

append(*v*)

extend(*v*)

pop(*v=None*)

remove(*v*)

reverse()

set_label(*value*)
Set the node label. This will only succeed the first time the node label is set, which should occur in ImmutableTree.__init__().

sort()

class nltk.tree.**ProbabilisticMixIn**(***kwargs*)
Bases: object

A mix-in class to associate probabilities with other classes (trees, rules, etc.). To use the ProbabilisticMixIn class, define a new class that derives from an existing class and from ProbabilisticMixIn. You will need to define a new constructor for the new class, which explicitly calls the constructors of both its parent classes. For example:

```
>>> from nltk.probability import ProbabilisticMixIn
>>> class A:
...     def __init__(self, x, y): self.data = (x,y)
...
>>> class ProbabilisticA(A, ProbabilisticMixIn):
...     def __init__(self, x, y, **prob_kwarg):
...         A.__init__(self, x, y)
...         ProbabilisticMixIn.__init__(self, **prob_kwarg)
```

See the documentation for the ProbabilisticMixIn `constructor<__init__>` for information about the arguments it expects.

You should generally also redefine the string representation methods, the comparison methods, and the hashing method.

logprob()
Return `log(p)`, where `p` is the probability associated with this object.

Return type float

prob()
Return the probability associated with this object.

Return type float

set_logprob(*logprob*)
Set the log probability associated with this object to `logprob`. I.e., set the probability associated with this object to `2**(logprob)`.

Parameters **logprob** (*float*) – The new log probability

set_prob(*prob*)
Set the probability associated with this object to `prob`.

Parameters **prob** (*float*) – The new probability

class nltk.tree.**ProbabilisticTree**(*node*, *children=None*, ***prob_kwargs*)
Bases: *nltk.tree.Tree*, *nltk.probability.ProbabilisticMixIn*

classmethod convert(*val*)

copy(*deep=False*)

unicode_repr()

class nltk.tree.**Tree**(*node*, *children=None*)
Bases: list

A Tree represents a hierarchical grouping of leaves and subtrees. For example, each constituent in a syntax tree is represented by a single Tree.

A tree's children are encoded as a list of leaves and subtrees, where a leaf is a basic (non-tree) value; and a subtree is a nested Tree.

```
>>> from nltk.tree import Tree
>>> print(Tree(1, [2, Tree(3, [4]), 5]))
(1 2 (3 4) 5)
>>> vp = Tree('VP', [Tree('V', ['saw']),
...                  Tree('NP', ['him'])])
>>> s = Tree('S', [Tree('NP', ['I']), vp])
>>> print(s)
(S (NP I) (VP (V saw) (NP him)))
>>> print(s[1])
```

```
    (VP (V saw) (NP him))
    >>> print(s[1,1])
    (NP him)
    >>> t = Tree.fromstring("(S (NP I) (VP (V saw) (NP him)))")
    >>> s == t
    True
    >>> t[1][1].set_label('X')
    >>> t[1][1].label()
    'X'
    >>> print(t)
    (S (NP I) (VP (V saw) (X him)))
    >>> t[0], t[1,1] = t[1,1], t[0]
    >>> print(t)
    (S (X him) (VP (V saw) (NP I)))
```

The length of a tree is the number of children it has.

```
    >>> len(t)
    2
```

The set_label() and label() methods allow individual constituents to be labeled. For example, syntax trees use this label to specify phrase tags, such as "NP" and "VP".

Several Tree methods use "tree positions" to specify children or descendants of a tree. Tree positions are defined as follows:

- The tree position *i* specifies a Tree's *i*th child.
- The tree position `()` specifies the Tree itself.
- If *p* is the tree position of descendant *d*, then *p+i* specifies the *i*th child of *d*.

I.e., every tree position is either a single index *i*, specifying `tree[i]`; or a sequence *i1, i2, ..., iN*, specifying `tree[i1][i2]...[iN]`.

Construct a new tree. This constructor can be called in one of two ways:

- **`Tree(label, children)` constructs a new tree with the** specified label and list of children.
- `Tree.fromstring(s)` constructs a new tree by parsing the string `s`.

chomsky_normal_form(*factor=u'right'*, *horzMarkov=None*, *vertMarkov=0*, *childChar=u'|'*, *parentChar=u'^'*)

This method can modify a tree in three ways:

1. Convert a tree into its Chomsky Normal Form (CNF) equivalent – Every subtree has either two non-terminals or one terminal as its children. This process requires the creation of more"artificial" non-terminal nodes.
2. Markov (vertical) smoothing of children in new artificial nodes
3. Horizontal (parent) annotation of nodes

Parameters

- **factor** (*str = [left|right]*) – Right or left factoring method (default = "right")
- **horzMarkov** (*int | None*) – Markov order for sibling smoothing in artificial nodes (None (default) = include all siblings)
- **vertMarkov** (*int | None*) – Markov order for parent smoothing (0 (default) = no vertical annotation)

- **childChar** (*str*) – A string used in construction of the artificial nodes, separating the head of the original subtree from the child nodes that have yet to be expanded (default = "|")
- **parentChar** (*str*) – A string used to separate the node representation from its vertical annotation

collapse_unary(*collapsePOS=False*, *collapseRoot=False*, *joinChar=u'+'*)
Collapse subtrees with a single child (ie. unary productions) into a new non-terminal (Tree node) joined by 'joinChar'. This is useful when working with algorithms that do not allow unary productions, and completely removing the unary productions would require loss of useful information. The Tree is modified directly (since it is passed by reference) and no value is returned.

Parameters

- **collapsePOS** (*bool*) – 'False' (default) will not collapse the parent of leaf nodes (ie. Part-of-Speech tags) since they are always unary productions
- **collapseRoot** (*bool*) – 'False' (default) will not modify the root production if it is unary. For the Penn WSJ treebank corpus, this corresponds to the TOP -> productions.
- **joinChar** (*str*) – A string used to connect collapsed node values (default = "+")

classmethod convert(*tree*)
Convert a tree between different subtypes of Tree. `cls` determines which class will be used to encode the new tree.

Parameters **tree** (Tree) – The tree that should be converted.

Returns The new Tree.

copy(*deep=False*)

draw()
Open a new window containing a graphical diagram of this tree.

flatten()
Return a flat version of the tree, with all non-root non-terminals removed.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> print(t.flatten())
(S the dog chased the cat)
```

Returns a tree consisting of this tree's root connected directly to its leaves, omitting all intervening non-terminal nodes.

Return type *Tree*

freeze(*leaf_freezer=None*)

classmethod fromstring(*s*, *brackets=u'()'*, *read_node=None*, *read_leaf=None*, *node_pattern=None*, *leaf_pattern=None*, *remove_empty_top_bracketing=False*)
Read a bracketed tree string and return the resulting tree. Trees are represented as nested brackettings, such as:

```
(S (NP (NNP John)) (VP (V runs)))
```

Parameters

- **s** (*str*) – The string to read

- **brackets** (*str (length=2)*) – The bracket characters used to mark the beginning and end of trees and subtrees.
- **read_leaf** (*read_node,*) – If specified, these functions are applied to the substrings of `s` corresponding to nodes and leaves (respectively) to obtain the values for those nodes and leaves. They should have the following signature:

  read_node(str) -> value

  For example, these functions could be used to process nodes and leaves whose values should be some type other than string (such as `FeatStruct`). Note that by default, node strings and leaf strings are delimited by whitespace and brackets; to override this default, use the `node_pattern` and `leaf_pattern` arguments.
- **leaf_pattern** (*node_pattern,*) – Regular expression patterns used to find node and leaf substrings in `s`. By default, both nodes patterns are defined to match any sequence of non-whitespace non-bracket characters.
- **remove_empty_top_bracketing** (*bool*) – If the resulting tree has an empty node label, and is length one, then return its single child instead. This is useful for treebank trees, which sometimes contain an extra level of bracketing.

Returns A tree corresponding to the string representation `s`. If this class method is called using a subclass of Tree, then it will return a tree of that type.

Return type *Tree*

height()

Return the height of the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.height()
5
>>> print(t[0,0])
(D the)
>>> t[0,0].height()
2
```

Returns The height of this tree. The height of a tree containing no children is 1; the height of a tree containing only leaves is 2; and the height of any other tree is one plus the maximum of its children's heights.

Return type int

label()

Return the node label of the tree.

```
>>> t = Tree.fromstring('(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))')
>>> t.label()
'S'
```

Returns the node label (typically a string)

Return type any

leaf_treeposition(*index*)

Returns The tree position of the `index`-th leaf in this tree. I.e., if `tp=self.leaf_treeposition(i)`, then `self[tp]==self.leaves()[i]`.

Raises IndexError If this tree contains fewer than `index+1` leaves, or if `index<0`.

leaves()
Return the leaves of the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.leaves()
['the', 'dog', 'chased', 'the', 'cat']
```

Returns a list containing this tree's leaves. The order reflects the order of the leaves in the tree's hierarchical structure.

Return type *list*

node
Outdated method to access the node value; use the label() method instead.

pformat(*margin=70*, *indent=0*, *nodesep=u''*, *parens=u'()'*, *quotes=False*)

Returns A pretty-printed string representation of this tree.

Return type str

Parameters

- **margin** (*int*) – The right margin at which to do line-wrapping.
- **indent** (*int*) – The indentation level at which printing begins. This number is used to decide how far to indent subsequent lines.
- **nodesep** – A string that is used to separate the node from the children. E.g., the default value `':'` gives trees like `(S: (NP: I) (VP: (V: saw) (NP: it)))`.

pformat_latex_qtree()
Returns a representation of the tree compatible with the LaTeX qtree package. This consists of the string `\Tree` followed by the tree represented in bracketed notation.

For example, the following result was generated from a parse tree of the sentence `The announcement astounded us`:

```
\Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
```

See http://www.ling.upenn.edu/advice/latex.html for the LaTeX style file for the qtree package.

Returns A latex qtree representation of this tree.

Return type str

pos()
Return a sequence of pos-tagged words extracted from the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.pos()
[('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
```

Returns a list of tuples containing leaves and pre-terminals (part-of-speech tags). The order reflects the order of the leaves in the tree's hierarchical structure.

Return type list(tuple)

pprint(***kwargs*)
Print a string representation of this Tree to 'stream'

pretty_print (*sentence=None*, *highlight=()*, *stream=None*, ***kwargs*)
Pretty-print this tree as ASCII or Unicode art. For explanation of the arguments, see the documentation for *nltk.treeprettyprinter.TreePrettyPrinter*.

productions()
Generate the productions that correspond to the non-terminal nodes of the tree. For each subtree of the form (P: C1 C2 ... Cn) this produces a production of the form P -> C1 C2 ... Cn.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.productions()
[S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP, V -> 'chased',
NP -> D N, D -> 'the', N -> 'cat']
```

Return type list(Production)

set_label (*label*)
Set the node label of the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.set_label("T")
>>> print(t)
(T (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))
```

Parameters **label** (*any*) – the node label (typically a string)

subtrees (*filter=None*)
Generate all the subtrees of this tree, optionally restricted to trees matching the filter function.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> for s in t.subtrees(lambda t: t.height() == 2):
...     print(s)
(D the)
(N dog)
(V chased)
(D the)
(N cat)
```

Parameters **filter** (*function*) – the function to filter all local trees

treeposition_spanning_leaves (*start*, *end*)

Returns The tree position of the lowest descendant of this tree that dominates `self.leaves()[start:end]`.

Raises ValueError if `end <= start`

treepositions (*order='preorder'*)

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))")
>>> t.treepositions()
[(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0, 0), ...]
>>> for pos in t.treepositions('leaves'):
...     t[pos] = t[pos][::-1].upper()
>>> print(t)
(S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT) (N TAC))))
```

Parameters **order** – One of: `preorder`, `postorder`, `bothorder`, `leaves`.

un_chomsky_normal_form(*expandUnary=True*, *childChar=u'|'*, *parentChar=u'^'*, *unaryChar=u'+'*)

This method modifies the tree in three ways:

1.Transforms a tree in Chomsky Normal Form back to its original structure (branching greater than two)

2.Removes any parent annotation (if it exists)

3.(optional) expands unary subtrees (if previously collapsed with collapseUnary(...))

Parameters

- **expandUnary** (*bool*) – Flag to expand unary or not (default = True)
- **childChar** (*str*) – A string separating the head node from its children in an artificial node (default = "|")
- **parentChar** (*str*) – A sting separating the node label from its parent annotation (default = "^")
- **unaryChar** (*str*) – A string joining two non-terminals in a unary production (default = "+")

unicode_repr()

nltk.tree.**bracket_parse**(*s*)

Use Tree.read(s, remove_empty_top_bracketing=True) instead.

nltk.tree.**sinica_parse**(*s*)

Parse a Sinica Treebank string and return a tree. Trees are represented as nested brackettings, as shown in the following example (X represents a Chinese character): S(goal:NP(Head:Nep:XX)|theme:NP(Head:Nhaa:X)|quantity:Dab:X|Head:VL2:X)#0(PERIODCATEGORY)

Returns A tree corresponding to the string representation.

Return type *Tree*

Parameters **s** (*str*) – The string to be converted

class nltk.tree.**ParentedTree**(*node*, *children=None*)

Bases: nltk.tree.AbstractParentedTree

A Tree that automatically maintains parent pointers for single-parented trees. The following are methods for querying the structure of a parented tree: parent, parent_index, left_sibling, right_sibling, root, treeposition.

Each ParentedTree may have at most one parent. In particular, subtrees may not be shared. Any attempt to reuse a single ParentedTree as a child of more than one parent (or as multiple children of the same parent) will cause a ValueError exception to be raised.

ParentedTrees should never be used in the same tree as Trees or MultiParentedTrees. Mixing tree implementations may result in incorrect parent pointers and in TypeError exceptions.

left_sibling()

The left sibling of this tree, or None if it has none.

parent()

The parent of this tree, or None if it has no parent.

parent_index()

The index of this tree in its parent. I.e., ptree.parent()[ptree.parent_index()] is ptree. Note that ptree.parent_index() is not necessarily equal to ptree.parent.index(ptree), since the index() method returns the first child that is equal to its argument.

right_sibling()
The right sibling of this tree, or None if it has none.

root()
The root of this tree. I.e., the unique ancestor of this tree whose parent is None. If `ptree.parent()` is None, then `ptree` is its own root.

treeposition()
The tree position of this tree, relative to the root of the tree. I.e., `ptree.root[ptree.treeposition] is ptree`.

class nltk.tree.**MultiParentedTree**(*node*, *children=None*)
Bases: nltk.tree.AbstractParentedTree

A `Tree` that automatically maintains parent pointers for multi-parented trees. The following are methods for querying the structure of a multi-parented tree: `parents()`, `parent_indices()`, `left_siblings()`, `right_siblings()`, `roots`, `treepositions`.

Each `MultiParentedTree` may have zero or more parents. In particular, subtrees may be shared. If a single `MultiParentedTree` is used as multiple children of the same parent, then that parent will appear multiple times in its `parents()` method.

`MultiParentedTrees` should never be used in the same tree as `Trees` or `ParentedTrees`. Mixing tree implementations may result in incorrect parent pointers and in `TypeError` exceptions.

left_siblings()
A list of all left siblings of this tree, in any of its parent trees. A tree may be its own left sibling if it is used as multiple contiguous children of the same parent. A tree may appear multiple times in this list if it is the left sibling of this tree with respect to multiple parents.

Type list(MultiParentedTree)

parent_indices(*parent*)
Return a list of the indices where this tree occurs as a child of `parent`. If this child does not occur as a child of `parent`, then the empty list is returned. The following is always true:

```
for parent_index in ptree.parent_indices(parent):
    parent[parent_index] is ptree
```

parents()
The set of parents of this tree. If this tree has no parents, then `parents` is the empty set. To check if a tree is used as multiple children of the same parent, use the `parent_indices()` method.

Type list(MultiParentedTree)

right_siblings()
A list of all right siblings of this tree, in any of its parent trees. A tree may be its own right sibling if it is used as multiple contiguous children of the same parent. A tree may appear multiple times in this list if it is the right sibling of this tree with respect to multiple parents.

Type list(MultiParentedTree)

roots()
The set of all roots of this tree. This set is formed by tracing all possible parent paths until trees with no parents are found.

Type list(MultiParentedTree)

treepositions(*root*)
Return a list of all tree positions that can be used to reach this multi-parented tree starting from `root`. I.e., the following is always true:

```
        for treepos in ptree.treepositions(root):
            root[treepos] is ptree
```

class nltk.tree.**ImmutableParentedTree**(*node*, *children=None*)
Bases: *nltk.tree.ImmutableTree*, *nltk.tree.ParentedTree*

class nltk.tree.**ImmutableMultiParentedTree**(*node*, *children=None*)
Bases: *nltk.tree.ImmutableTree*, *nltk.tree.MultiParentedTree*

3.5.13 treetransforms Module

A collection of methods for tree (grammar) transformations used in parsing natural language.

Although many of these methods are technically grammar transformations (ie. Chomsky Norm Form), when working with treebanks it is much more natural to visualize these modifications in a tree structure. Hence, we will do all transformation directly to the tree itself. Transforming the tree directly also allows us to do parent annotation. A grammar can then be simply induced from the modified tree.

The following is a short tutorial on the available transformations.

1. Chomsky Normal Form (binarization)

   It is well known that any grammar has a Chomsky Normal Form (CNF) equivalent grammar where CNF is defined by every production having either two non-terminals or one terminal on its right hand side. When we have hierarchically structured data (ie. a treebank), it is natural to view this in terms of productions where the root of every subtree is the head (left hand side) of the production and all of its children are the right hand side constituents. In order to convert a tree into CNF, we simply need to ensure that every subtree has either two subtrees as children (binarization), or one leaf node (non-terminal). In order to binarize a subtree with more than two children, we must introduce artificial nodes.

   There are two popular methods to convert a tree into CNF: left factoring and right factoring. The following example demonstrates the difference between them. Example:

```
    Original       Right-Factored     Left-Factored

         A              A                      A
       / | \          /   \                  /          B  C  D   ==>  B    A|<C-D>   OR   A|<B-C>
                        /  \              /                           C    D        B    C
```

2. Parent Annotation

   In addition to binarizing the tree, there are two standard modifications to node labels we can do in the same traversal: parent annotation and Markov order-N smoothing (or sibling smoothing).

   The purpose of parent annotation is to refine the probabilities of productions by adding a small amount of context. With this simple addition, a CYK (inside-outside, dynamic programming chart parse) can improve from 74% to 79% accuracy. A natural generalization from parent annotation is to grandparent annotation and beyond. The tradeoff becomes accuracy gain vs. computational complexity. We must also keep in mind data sparcity issues. Example:

```
    Original       Parent Annotation

         A               A^<?>
       / | \            /          B  C  D   ==>  B^<A>    A|<C-D>^<?>     where ? is the
                             /  \                parent of A
                          C^<A>   D^<A>
```

3. Markov order-N smoothing

Markov smoothing combats data sparcity issues as well as decreasing computational requirements by limiting the number of children included in artificial nodes. In practice, most people use an order 2 grammar. Example:

```
    Original       No Smoothing       Markov order 1   Markov order 2   etc.

     __A__            A                    A               A
    / /|\ \          /   \                /   \           /        B C D E F  ==>  B    A|<C-D-E-F
                        /   \                /   \           /                          C    .
```

Annotation decisions can be thought about in the vertical direction (parent, grandparent, etc) and the horizontal direction (number of siblings to keep). Parameters to the following functions specify these values. For more information see:

Dan Klein and Chris Manning (2003) "Accurate Unlexicalized Parsing", ACL-03. http://www.aclweb.org/anthology/P03-1054

4. Unary Collapsing

Collapse unary productions (ie. subtrees with a single child) into a new non-terminal (Tree node). This is useful when working with algorithms that do not allow unary productions, yet you do not wish to lose the parent information. Example:

```
     A
     |
     B    ==>    A+B
    / \          /      C    D       C    D
```

nltk.treetransforms.**chomsky_normal_form**(*tree*, *factor='right'*, *horzMarkov=None*, *vertMarkov=0*, *childChar='|'*, *parentChar='^'*)

nltk.treetransforms.**un_chomsky_normal_form**(*tree*, *expandUnary=True*, *childChar='|'*, *parentChar='^'*, *unaryChar='+'*)

nltk.treetransforms.**collapse_unary**(*tree*, *collapsePOS=False*, *collapseRoot=False*, *joinChar='+'*)

Collapse subtrees with a single child (ie. unary productions) into a new non-terminal (Tree node) joined by 'joinChar'. This is useful when working with algorithms that do not allow unary productions, and completely removing the unary productions would require loss of useful information. The Tree is modified directly (since it is passed by reference) and no value is returned.

Parameters

- **tree** (Tree) – The Tree to be collapsed
- **collapsePOS** (*bool*) – 'False' (default) will not collapse the parent of leaf nodes (ie. Part-of-Speech tags) since they are always unary productions
- **collapseRoot** (*bool*) – 'False' (default) will not modify the root production if it is unary. For the Penn WSJ treebank corpus, this corresponds to the TOP -> productions.
- **joinChar** (*str*) – A string used to connect collapsed node values (default = "+")

3.5.14 util Module

class nltk.util.**AbstractLazySequence**

Bases: object

An abstract base class for read-only sequences whose values are computed as needed. Lazy sequences act like tuples – they can be indexed, sliced, and iterated over; but they may not be modified.

The most common application of lazy sequences in NLTK is for corpus view objects, which provide access to the contents of a corpus without loading the entire corpus into memory, by loading pieces of the corpus from disk as needed.

The result of modifying a mutable element of a lazy sequence is undefined. In particular, the modifications made to the element may or may not persist, depending on whether and when the lazy sequence caches that element's value or reconstructs it from scratch.

Subclasses are required to define two methods: `__len__()` and `iterate_from()`.

count(*value*)
Return the number of times this list contains `value`.

index(*value*, *start=None*, *stop=None*)
Return the index of the first occurrence of `value` in this list that is greater than or equal to `start` and less than `stop`. Negative start and stop values are treated like negative slice bounds – i.e., they count from the end of the list.

iterate_from(*start*)
Return an iterator that generates the tokens in the corpus file underlying this corpus view, starting at the token number `start`. If `start>=len(self)`, then this iterator will generate no tokens.

unicode_repr()
Return a string representation for this corpus view that is similar to a list's representation; but if it would be more than 60 characters long, it is truncated.

class nltk.util.**Index**(*pairs*)
Bases: collections.defaultdict

class nltk.util.**LazyConcatenation**(*list_of_lists*)
Bases: *nltk.util.AbstractLazySequence*

A lazy sequence formed by concatenating a list of lists. This underlying list of lists may itself be lazy. `LazyConcatenation` maintains an index that it uses to keep track of the relationship between offsets in the concatenated lists and offsets in the sublists.

iterate_from(*start_index*)

class nltk.util.**LazyEnumerate**(*lst*)
Bases: *nltk.util.LazyZip*

A lazy sequence whose elements are tuples, each ontaining a count (from zero) and a value yielded by underlying sequence. `LazyEnumerate` is useful for obtaining an indexed list. The tuples are constructed lazily – i.e., when you read a value from the list, `LazyEnumerate` will calculate that value by forming a tuple from the count of the i-th element and the i-th element of the underlying sequence.

`LazyEnumerate` is essentially a lazy version of the Python primitive function `enumerate`. In particular, the following two expressions are equivalent:

```
>>> from nltk.util import LazyEnumerate
>>> sequence = ['first', 'second', 'third']
>>> list(enumerate(sequence))
[(0, 'first'), (1, 'second'), (2, 'third')]
>>> list(LazyEnumerate(sequence))
[(0, 'first'), (1, 'second'), (2, 'third')]
```

Lazy enumerations can be useful for conserving memory in cases where the argument sequences are particularly long.

A typical example of a use case for this class is obtaining an indexed list for a long sequence of values. By constructing tuples lazily and avoiding the creation of an additional long sequence, memory usage can be significantly reduced.

class nltk.util.**LazyMap**(*function*, **lists*, ***config*)
Bases: *nltk.util.AbstractLazySequence*

A lazy sequence whose elements are formed by applying a given function to each element in one or more underlying lists. The function is applied lazily – i.e., when you read a value from the list, LazyMap will calculate that value by applying its function to the underlying lists' value(s). LazyMap is essentially a lazy version of the Python primitive function map. In particular, the following two expressions are equivalent:

```
>>> from nltk.util import LazyMap
>>> function = str
>>> sequence = [1,2,3]
>>> map(function, sequence)
['1', '2', '3']
>>> list(LazyMap(function, sequence))
['1', '2', '3']
```

Like the Python map primitive, if the source lists do not have equal size, then the value None will be supplied for the 'missing' elements.

Lazy maps can be useful for conserving memory, in cases where individual values take up a lot of space. This is especially true if the underlying list's values are constructed lazily, as is the case with many corpus readers.

A typical example of a use case for this class is performing feature detection on the tokens in a corpus. Since featuresets are encoded as dictionaries, which can take up a lot of memory, using a LazyMap can significantly reduce memory usage when training and running classifiers.

iterate_from(*index*)

class nltk.util.**LazySubsequence**(*source*, *start*, *stop*)
Bases: *nltk.util.AbstractLazySequence*

A subsequence produced by slicing a lazy sequence. This slice keeps a reference to its source sequence, and generates its values by looking them up in the source sequence.

MIN_SIZE = 100
The minimum size for which lazy slices should be created. If LazySubsequence() is called with a subsequence that is shorter than MIN_SIZE, then a tuple will be returned instead.

iterate_from(*start*)

class nltk.util.**LazyZip**(**lists*)
Bases: *nltk.util.LazyMap*

A lazy sequence whose elements are tuples, each containing the i-th element from each of the argument sequences. The returned list is truncated in length to the length of the shortest argument sequence. The tuples are constructed lazily – i.e., when you read a value from the list, LazyZip will calculate that value by forming a tuple from the i-th element of each of the argument sequences.

LazyZip is essentially a lazy version of the Python primitive function zip. In particular, an evaluated LazyZip is equivalent to a zip:

```
>>> from nltk.util import LazyZip
>>> sequence1, sequence2 = [1, 2, 3], ['a', 'b', 'c']
>>> zip(sequence1, sequence2)
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(LazyZip(sequence1, sequence2))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> sequences = [sequence1, sequence2, [6,7,8,9]]
>>> list(zip(*sequences)) == list(LazyZip(*sequences))
True
```

Lazy zips can be useful for conserving memory in cases where the argument sequences are particularly long.

A typical example of a use case for this class is combining long sequences of gold standard and predicted values in a classification or tagging task in order to calculate accuracy. By constructing tuples lazily and avoiding the creation of an additional long sequence, memory usage can be significantly reduced.

iterate_from(*index*)

class nltk.util.**OrderedDict**(*data=None*, ***kwargs*)
Bases: dict

clear()

copy()

items()

keys(*data=None*, *keys=None*)

popitem()

setdefault(*key*, *failobj=None*)

update(*data*)

values()

nltk.util.**bigrams**(*sequence*, ***kwargs*)
Return the bigrams generated from a sequence of items, as an iterator. For example:

```
>>> from nltk.util import bigrams
>>> list(bigrams([1,2,3,4,5]))
[(1, 2), (2, 3), (3, 4), (4, 5)]
```

Use bigrams for a list version of this function.

Parameters **sequence** (*sequence or iter*) – the source data to be converted into bigrams

Return type iter(tuple)

nltk.util.**binary_search_file**(*file*, *key*, *cache={}*, *cacheDepth=-1*)
Return the line from the file with first word key. Searches through a sorted file using the binary search algorithm.

Parameters

- **file** (*file*) – the file to be searched through.
- **key** (*str*) – the identifier we are searching for.

nltk.util.**breadth_first**(*tree*, *children=<built-in function iter>*, *maxdepth=-1*)
Traverse the nodes of a tree in breadth-first order. (No need to check for cycles.) The first argument should be the tree root; children should be a function taking as argument a tree node and returning an iterator of the node's children.

nltk.util.**clean_html**(*html*)

nltk.util.**clean_url**(*url*)

nltk.util.**elementtree_indent**(*elem*, *level=0*)
Recursive function to indent an ElementTree._ElementInterface used for pretty printing. Run indent on elem and then output in the normal way.

Parameters

- **elem** (*ElementTree._ElementInterface*) – element to be indented. will be modified.
- **level** (*nonnegative integer*) – level of indentation for this element

Return type ElementTree._ElementInterface

Returns Contents of elem indented to reflect its structure

nltk.util.**filestring**(*f*)

nltk.util.**flatten**(**args*)
Flatten a list.

```
>>> from nltk.util import flatten
>>> flatten(1, 2, ['b', 'a' , ['c', 'd']], 3)
[1, 2, 'b', 'a', 'c', 'd', 3]
```

Parameters **args** – items and lists to be combined into a single list

Return type *list*

nltk.util.**guess_encoding**(*data*)
Given a byte string, attempt to decode it. Tries the standard 'UTF8' and 'latin-1' encodings, Plus several gathered from locale information.

The calling program *must* first call:

```
locale.setlocale(locale.LC_ALL, '')
```

If successful it returns `(decoded_unicode, successful_encoding)`. If unsuccessful it raises a `UnicodeError`.

nltk.util.**in_idle**()
Return True if this function is run within idle. Tkinter programs that are run in idle should never call `Tk.mainloop`; so this function should be used to gate all calls to `Tk.mainloop`.

Warning This function works by checking `sys.stdin`. If the user has modified `sys.stdin`, then it may return incorrect results.

Return type bool

nltk.util.**invert_dict**(*d*)

nltk.util.**invert_graph**(*graph*)
Inverts a directed graph.

Parameters **graph** (*dict(set)*) – the graph, represented as a dictionary of sets

Returns the inverted graph

Return type dict(set)

nltk.util.**ngrams**(*sequence*, *n*, *pad_left=False*, *pad_right=False*, *pad_symbol=None*)
Return the ngrams generated from a sequence of items, as an iterator. For example:

```
>>> from nltk.util import ngrams
>>> list(ngrams([1,2,3,4,5], 3))
[(1, 2, 3), (2, 3, 4), (3, 4, 5)]
```

Use ngrams for a list version of this function. Set pad_left or pad_right to true in order to get additional ngrams:

```
>>> list(ngrams([1,2,3,4,5], 2, pad_right=True))
[(1, 2), (2, 3), (3, 4), (4, 5), (5, None)]
```

Parameters

- **sequence** (*sequence or iter*) – the source data to be converted into ngrams
- **n** (*int*) – the degree of the ngrams

- **pad_left** (*bool*) – whether the ngrams should be left-padded
- **pad_right** (*bool*) – whether the ngrams should be right-padded
- **pad_symbol** (*any*) – the symbol to use for padding (default is None)

Return type iter(tuple)

nltk.util.**pr**(*data*, *start=0*, *end=None*)
Pretty print a sequence of data items

Parameters

- **data** (*sequence or iter*) – the data stream to print
- **start** (*int*) – the start position
- **end** (*int*) – the end position

nltk.util.**print_string**(*s*, *width=70*)
Pretty print a string, breaking lines on whitespace

Parameters

- **s** (*str*) – the string to print, consisting of words and spaces
- **width** (*int*) – the display width

nltk.util.**py25**()

nltk.util.**py26**()

nltk.util.**py27**()

nltk.util.**re_show**(*regexp*, *string*, *left='{'*, *right='}'*)
Return a string with markers surrounding the matched substrings. Search str for substrings matching `regexp` and wrap the matches with braces. This is convenient for learning about regular expressions.

Parameters

- **regexp** (*str*) – The regular expression.
- **string** (*str*) – The string being matched.
- **left** (*str*) – The left delimiter (printed before the matched substring)
- **right** (*str*) – The right delimiter (printed after the matched substring)

Return type str

nltk.util.**set_proxy**(*proxy*, *user=None*, *password=''*)
Set the HTTP proxy for Python to download through.

If `proxy` is None then tries to set proxy from environment or system settings.

Parameters

- **proxy** – The HTTP proxy server to use. For example: 'http://proxy.example.com:3128/'
- **user** – The username to authenticate with. Use None to disable authentication.
- **password** – The password to authenticate with.

nltk.util.**tokenwrap**(*tokens*, *separator=' '*, *width=70*)
Pretty print a list of text tokens, breaking lines on whitespace

Parameters

- **tokens** (list) – the tokens to print

- **separator** (*str*) – the string to use to separate tokens
- **width** (*int*) – the display width (default=70)

nltk.util.**transitive_closure**(*graph*, *reflexive=False*)

Calculate the transitive closure of a directed graph, optionally the reflexive transitive closure.

The algorithm is a slight modification of the "Marking Algorithm" of Ioannidis & Ramakrishnan (1998) "Efficient Transitive Closure Algorithms".

Parameters

- **graph** (*dict(set)*) – the initial graph, represented as a dictionary of sets
- **reflexive** (*bool*) – if set, also make the closure reflexive

Return type dict(set)

nltk.util.**trigrams**(*sequence*, ***kwargs*)

Return the trigrams generated from a sequence of items, as an iterator. For example:

```
>>> from nltk.util import trigrams
>>> list(trigrams([1,2,3,4,5]))
[(1, 2, 3), (2, 3, 4), (3, 4, 5)]
```

Use trigrams for a list version of this function.

Parameters **sequence** (*sequence or iter*) – the source data to be converted into trigrams

Return type iter(tuple)

nltk.util.**unique_list**(*xs*)

nltk.util.**usage**(*obj*, *selfname='self'*)

3.5.15 wsd Module

nltk.wsd.**lesk**(*context_sentence*, *ambiguous_word*, *pos=None*, *synsets=None*)

Return a synset for an ambiguous word in a context.

Parameters **context_sentence** (*iter*) – The context sentence where the ambiguous word

occurs, passed as an iterable of words. :param str ambiguous_word: The ambiguous word that requires WSD. :param str pos: A specified Part-of-Speech (POS). :param iter synsets: Possible synsets of the ambiguous word. :return: `lesk_sense` The Synset() object with the highest signature overlaps.

This function is an implementation of the original Lesk algorithm (1986) [1].

Usage example:

```
>>> lesk(['I', 'went', 'to', 'the', 'bank', 'to', 'deposit', 'money', '.'], 'bank', 'n')
Synset('savings_bank.n.02')
```

[1] Lesk, Michael. "Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone." Proceedings of the 5th Annual International Conference on Systems Documentation. ACM, 1986. http://dl.acm.org/citation.cfm?id=318728

3.5.16 Subpackages

- genindex
- modindex

- search

_

a

c

d

f

g

h

p

t

u

w

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X