## CONTENTS

1 Introduction 3  
1.1 Purpose .................................................................................................................. 3  
1.2 Platforms ................................................................................................................ 3  

2 Installation 5  
2.1 Dependencies .......................................................................................................... 5  
2.2 Installing OntoNotes DB Tool .................................................................................. 5  

3 Getting Started 7  
3.1 Loading Data to Database ....................................................................................... 7  
3.2 Loading and Dumping Data from Database .............................................................. 11  

4 Database and API Design 13  
4.1 Overview ................................................................................................................ 13  
4.2 Corpus .................................................................................................................... 13  
4.3 Trees ....................................................................................................................... 13  
4.4 Propositions ............................................................................................................ 16  
4.5 Word Senses ........................................................................................................... 16  
4.6 Name ...................................................................................................................... 18  
4.7 Coreference ............................................................................................................ 18  
4.8 Banks ..................................................................................................................... 18  

5 API Reference 21  
5.1 `on.common` – Utility and Logging Functions ....................................................... 21  
5.2 `on.corpora` – classes for interpreting annotation ................................................ 27  
5.3 `on.tools` – scripts for manipulating the ontonotes data ........................................ 54  

6 Appendix A: Reporting Bugs 55  

Python Module Index 57  

Index 59
This document describes the installation and basic uses of the OntoNotes Database Tool (DB Tool). It shows how to build the OntoNotes database from the individual source files and describes the design of the database tables, as well as parts of the API that can be used to manipulate the data. It also provides some documentation on the core API.
INTRODUCTION

1.1 Purpose

The OntoNotes DB Tool provides code for working with the OntoNotes data, including building and accessing the relational database. The linked tables of the database store a text corpus along with the layers of annotation specifying syntactic structure, propositional structure, word senses for nouns and verbs, names, and the coreference between entities mentioned in the text. It also contains metadata such as proposition frames, sense inventories, and where available, speaker information, pointers to the original or translation and lemma details.

Also included are functions that extract particular views of the merged data. Each of the individual input annotation formats, which was used in building the database, can also be extracted as a view. There is also a combined “OntoNotes Normal Form” view that includes all of the layers in a version intended for human review.

In addition to the top-level routines used to build the database and to extract views, the DB Tool provides an API with access to each of the individual tables, allowing users to construct many kinds of database queries more conveniently than would be possible in raw SQL, especially for defining new views.

1.2 Platforms

The OntoNotes DB Tool was built with cross-platform compatibility in mind, and therefore all the core components of the system as well as the external libraries that were used are themselves available for multiple platforms – at least for Linux/Unix and Windows. Given the limited resource availability during development, however, all development was done in the Linux environment and we could not thoroughly test the distribution across other platforms. It would be quite surprising if it does not work out of the box, or with a few minor tweaks in a Windows environment. The majority of the code is written in Python. The version used for development was Python 2.5.1.

Since this is a software in its Beta release and has not been thoroughly tested, it is quite likely that it has bugs in it. Please refer to Appendix A: Reporting Bugs for more details in how to go about reporting bugs that you might find.

See Also:

The OntoNotes Project The official OntoNotes project webpage, with details about the individual components, latest developments, etc.
2.1 Dependencies

The OntoNotes DB Tool is implemented in Python, using a MySQL database and Python access routines from the MySQLdb package. Users will need to have Python installed, and if they wish to use any of the code that works with the database they will need MySQL as well.

2.1.1 Python

The tool was designed under Python 2.5, but should work on any later Python 2.x release. Python 3 is not supported, though users working with Python 3 might have success with the official Python 2.x to 3.0 ‘2to3’ converter.

2.1.2 MySQL and MySQLdb

These two dependencies are optional, but recommended. The tool can function in two modes, either reading information from the filesystem or the database. To have the latter mode available, you must have a MySQL database to store the data as well as the MySQLdb Python module. You can download MySQL from http://dev.mysql.com/downloads/ and MySQLdb from http://sourceforge.net/projects/mysql-python.

The version of MySQL that we have tested the OntoNotes DB Tool is version 5.0, but we do not use much of the newly added functionality as of now, and so an earlier version of MySQL will do as well – in case you already have it installed. We would highly recommend using version 4.1 or later as they have better Unicode support. This is quite important, as all the data uses the UTF-8 character encoding.

See Also:

Official MySQL Documentation  Extensive documentation on MySQL can be found at site.
Writing MySQL Scripts with Python DB-API  This is a good starting point for writing scripts using MySQLdb API.

2.2 Installing OntoNotes DB Tool

Once you have the dependencies installed, we can install the OntoNotes DB Tool. It follows the standard Python module distribution guidelines and uses the Python distutils package. After you have installed the above mentioned dependencies, you can install the OntoNotes DB Tool by running:

$ python setup.py install

See Also:
Installing Python Modules  Documentation on how to install Python modules
Now we will look at four – probably the four most frequent – tasks that an end-user would be likely to use the tool for.

### 3.1 Loading Data to Database

First we have to initialize the database. This is done with the script `init_db.py`.

There are a pair of scripts, `init_db.py` and `load_to_db.py`, for loading the data to the database. First, make sure the data is in the format we distributed it in. For reference, this means:

```
.../data/<lang>/annotations/<genre>/<source>/<section>/<file>
```

Now we initialize the database. This creates the relevant tables:

```
$ python on/tools/init_db.py --init my_db my_server my_name ontonotes-release-4.0/
```

Note that `d_loc` doesn’t end in “/data”.

Once the tables are created, we can load frames or sense inventories to the data for languages we desire. Load the frames before the sense inventories:

```
$ python on/tools/init_db.py --frames=english my_db my_server my_name ontonotes-release-4.0/
```
Next, create a configuration file patterned after the example below: (shipped as tools/config.example)

###
### This is a configuration file in the format expected by
### ConfigParser. Any line beginning with a hash mark (#) is a
### comment. Lines beginning with three hash marks are used for
### general commentary, lines with only one are example configuration
### settings
###
### You must set corpus.data_in or fill out the section db
### and will likely want to change other settings
###
### Note that fields that take lists (for example banks) want space
### separated elements. For example:
###
###   banks: parse prop coref sense
###
### Also note that all settings in the configuration file may be
### overridden on the command line if the tool is using
### on.common.util.load_options as all the tools here do. The syntax
### for that is:
###
###   python some_tool -c some_config Section.Option=Value
###
### For example:
###
###   python load_to_db.py -c config.example corpus.data_in=/corpus/data
###
### Because we use spaces to separate elements of lists, you may need
### to put argument values in quotes:
###
###   python load_to_db.py -c config.example corpus.banks="name prop"
###
### In general, each section here is used by one api function.
###

[corpus]
###### This section is used by on.ontonotes.__init__ ######
###
### data_in: Where to look for the data
###
### this should be a path ending in 'data/'; the root of the ontonotes
### file system heirarchy.

data_in: /path/to/the/data/

### If you set data_in to a specific parse file, as so:
#
# data_in: /path/to/the/data/english/annotations/nw/wsj/00/wsj_0020.parse
#
### Then only that document will be loaded and you can ignore the next paragraph on the load variable and leave it unset

### What to load:

### There is a hierarchy: lang / genre / source. We specify what to load in the form lang-genre-source, with the final term optional.

### The example configuration below loads only the wsj (which is in english.nw):

load: english-nw-wsj

### Note that specific sources are optional. To load all the chinese broadcast news data, you would set load to:

# load: chinese-bn

### One can also load multiple sections, for example if one wants to work with english and chinese broadcast communications data (half of which is parallel):

# load: english-bc chinese-bc

### This a flexible configuration system, but if it is not sufficiently flexible, one option would be to look at the source code for on.ontonotes.from_files() to see how to manually and precisely select what data is loaded.

### One can also load only some individual documents. The configuration values that control this are:

### suffix, and

### prefix

### The default, if none of these are set, is to load all files for each loaded source regardless of their four digit id. This is equivalent to the setting either of them to nothing:

# prefix:
# suffix:

### Set them to space separated lists of prefixes and suffixes:

# suffix: 6 7 8
# prefix: 00 01 02

### The above settings would load all files whose id ends in 6, 7, or 8 and whose id starts with 00, 01, or 02.

3.1. Loading Data to Database
### We can also set the granularity of loading. A subcorpus is an arbitrary collection of documents, that by default will be all the documents for one source. If we want to have finer grained divisions, we can. This means that the code:

```python
    # config has corpus.load=english-nw
    a_ontonotes = on.ontonotes.from_files(config)
    print len(a_ontonotes)
```

Would produce different output for each granularity setting. If granularity=source, english-nw has both wsj and xinhua, so the ontonotes object would contain two subcorpus instances and it would print "1". For granularity=section, there are 25 sections in the wsj and 4 in xinhua, so we would have each of these as a subcorpus instance and print "29". For granularity=file, there are a total of 922 documents in all those sections, so it would create 922 subcorpus objects each containing only a single document.

###

granularity: source
# granularity: section
# granularity: file

### Once we’ve selected which documents to put in our subcorpus with on.ontonotes.from_files, we need to decide which banks to load. That is, do we want to load the data in the .prop files? What about the .coref files?

###

banks: parse coref sense name parallel prop speaker

### You need not load all the banks:

```python
    # banks: parse coref sense
    # banks: parse
```

### The name and sense data included in this distribution is all word indexed. If you want to process different data with different indexing (indexing that counts traces) then set these variables appropriately.

```python
wsd-indexing: word # token | nword_vtoken | notoken_vword
name-indexing: word # token
```

### Normally, when you load senses or propositions, the metadata that
### tells you how to interpret them is also loaded. If you don’t wish
### to load these, generally because they are slow to load, you can
### say so here. A value of "senses" means "don’t load the sense
### inventories" and a value of "frames" means "don’t load the frame
### files". One can specify both.
### The default value is equivalent to "corpus.ignore_inventories="
### and results in loading both of them as needed.
###
### # ignore-inventories: senses frames

# [db]
# This section is used by on.ontonotes.db_cursor as well as other functions that call db_cursor() such as load_banks
### Configuration information for the db
###
# db: ontonotes_v3
# host: your-mysql-host
# db-user: your-mysql-user
#

Be sure to include a **db** section so the script has information about what database to load to and where to find it.

Now we can run **tools.load_to_db** to actually complete the loading (note that the exact output will depend on

the options you chose in the configuration file. Here we’re loading parses and word sense data for the Wall Street
Journal sections 02 and 03):

```
$ python on/tools/load_to_db.py --config=load_to_db.conf
```

Loading english

```bash
nw
wsj
```

....[...].... found 200 files starting with any of ['02', '03'] in the \subcorpus all@wsj@nw@en@on Initializing DB... writing the type tables to db.............done. Loading all@wsj@nw@en@on reading the treebank ....[...]........... 4646 trees in the treebank reading the sense bank ....[...]........... reading the sense inventory files .[...]... 2164 aligning and enriching treebank with senses ....[...]...... writing document bank to db.[...]......done. writing treebank to db.[...]......[...]......done. writing sense bank to db.[...]......[...]......done.

At this point the database is populated and usable.

The source code of **tools.load_to_db** is a good example of how to use several aspects of the API.

### 3.2 Loading and Dumping Data from Database

Similarly, you can dump data from the database with the script **tools/files_from_db.py**. Again create a
configuration file based on config.example, but this time add to the configuration:

```
[FilesFromDb]
out_dir: /path/to/my/output/directory
```

### 3.2. Loading and Dumping Data from Database
Then run the code as:

```bash
$ python on/tools/files_from_db.py --config files_from_db.conf
```

If you for some reason wanted to send files to a different output directory than specified in the config file, you could override it on the command line:

```bash
$ python on/tools/files_from_db.py --config files_from_db.conf \
    FilesFromDb.out_dir=/path/to/another/output/directory
```

This form of command line overriding of configuration variables can be quite useful and works for all variables. All scripts use `on.common.util.load_options()` to parse command line arguments and load their config file. The `load_options` function calls `on.common.util.parse_cfg_args()` to deal with command line config overrides.
4.1 Overview

The entire database ER diagram is as shown below. The space is divided logically into “Corpus”, “Trees”, “Propositions”, “Senses”, “Names”, “Coreference” and “Ontology.”

We will look at the individual group of tables in the following subsections. The design of the database was chosen to be as close as possible to the object-oriented structure of the overall OntoNotes design. It should also be noted that we have traded-off database design principles for ease of querying in the computational semantics space. That is, we have not tried to reach the third or higher database normal form. The advantage of this is that there is a seamless connection between the database world and the object-oriented world, and therefore complex queries can be answered either using SQL or using the methods of the objects themselves. It should also be noted that the primary keys of almost all the tables are composite formed using the concatenation of individual foreign keys separated by the “@” symbol. This facilitates understanding the object in most cases by just looking at its primary key. The seamlessness is further enhanced by the use of the same primary keys for object ids. We decided to use the MyISAM database engine, and enforce only the primary key constraints, as adding foreign key constraints add a significant overhead on the queries, and since the data are not likely to be updated, it is only at load time that the constraint checking is important. Constraint checking is hence done in the DB Tool logic. Even though MyISAM ignores foreign key constraints, we do include them in table create statements to document them.

4.2 Corpus

The collection of tables that manage the corpora are shown in Figure 1.1. The “ontonotes” table stores the main ontonotes id. There can be more than one “subcorpus” (subcorpora) associated with the OntoNotes table. Each subcorpus represents an arbitrary set of documents, by default all the documents for a given source. The “subcorpus” table has associated with it a “language_id” and can contain many “file”s. Each file can contain one or more “documents”. In the current situation, there is only one document per file in all subcorpora, but this might change in the future. Then, each document has associated with it a list of “sentences”.

4.3 Trees

The tables that are used to represent the Treebank are shown in Figure 1.2. The central table in this region is the “tree” table which contains all the nodes in all the trees in the corpora. Since we use the Treebank tokenization as the lowest granularity for identifying elements in the corpus, we show the “token” table in this region. It contains back-pointer to the sentence and the tree that it belongs to. Associated with the tree nodes are various meta tags such as the part-of-speech tag, stored in the “pos_type” table, the phrase types, stored in the “phrase_type” table, the function tags stored in the “function_tag” table, and the syntactic links stored in the “syntactic_link” table.
Figure 4.1: The OntoNotes database ER diagram
4.3. Trees
4.4 Propositions

The tables that depict the PropBank information are shown in Figure 1.3. At the core is the “proposition” table which stores all the propositions in the corpora. The “predicate” and the “predicate_node” tables store the many-to-many relationship that can be associated with the predicates and the nodes in the tree. The same is the case with the “argument” and the “argument_node” table. The “argument_type” and the “predicate_type” tables store the predicate and argument types as defined in PropBank.

![Proposition Tables Diagram](image)

Figure 4.4: Proposition Tables

4.5 Word Senses

The OntoNotes word sense is stored in the “on_sense” table shown in Figure 1.4 There are connections between the OntoNotes sense and the Proposition Bank’s frames which are captured in the “on_sense_type_pb_sense_type” table. The WordNet senses which are grouped to form the OntoNotes sense and occasionally vice-versa are captured by the “on_sense_type_wn_sense_type” table. The frames files in PropBank restrict the types of core arguments that a predicate can take, and this information is stored in the “pb_sense_type_argument_type_table” which then has connections to the “argument_type” table in the PropBank table space. It should be noted that the WordNet mapping are only available for the English part of the corpus. There is no equivalent in the Chinese part.
Figure 4.5: Word Sense Tables
4.6 Name

The Name Entity relations are stored in the “name_entity” and “name_entity_type” tables shown in Figure 1.5. These are then connected to the respective tokens in the sentence, and are also connected to the appropriated nodes in the tree whenever there is a possible alignment.

4.7 Coreference

The “coreference_link” and “coreference_chain” tables, shown in Figure 1.6, in the Coreference table space store the information required to capture equivalent entities in the corpus. They also have information on the string span in the corpus that they are associated with, and in case of alignments (which in most cases is true since the coreference annotation was done on top of the treebank, withholding some exceptional entities) the node information is stored in the “coreference_link” table.

4.8 Banks

Each annotation project created a set of related data which is called a “bank” of that annotation – in accordance with Treebank, PropBank, etc. Each bank has three representations: as a table in the database, as a python object, and as a file. The correspondences are:

<table>
<thead>
<tr>
<th>Bank Name</th>
<th>Database Table</th>
<th>Python Module</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>tree</td>
<td>on.corpora.tree</td>
<td>.parse</td>
</tr>
<tr>
<td>sense</td>
<td>on_sense</td>
<td>on.corpora.sense</td>
<td>.sense</td>
</tr>
<tr>
<td>proposition</td>
<td>argument, predicate</td>
<td>on.corpora.proposition</td>
<td>prop</td>
</tr>
<tr>
<td>coreference</td>
<td>coreference_link</td>
<td>on.corpora.coreference</td>
<td>coref</td>
</tr>
<tr>
<td>name</td>
<td>name_entity</td>
<td>on.corpora.name</td>
<td>.name</td>
</tr>
<tr>
<td>speaker</td>
<td>speaker_sentence</td>
<td>on.corpora.speaker</td>
<td>.speaker</td>
</tr>
<tr>
<td>parallel</td>
<td>parallel_sentence, parallel_document</td>
<td>on.corpora.parallel</td>
<td>.parallel</td>
</tr>
</tbody>
</table>
This elucidates the connection between the three levels of representation – text files, python objects and database tables. There are actually multiple database tables used to represent the data. In the correspondence table above, we’ve listed the database table which contains the actual annotation alongside the python module which contains that bank. For example, for word sense annotation, we list the on_sense table along side the on.corpora.sense module. The on_sense table has the actual annotation, including fields for the lemma, sense, and a pointer to the annotated word in the tree. The on.corpora.sense module describes a on.corpora.sense.sense_bank containing on.corpora.sense.sense_tagged_document instances which contain on.corpora.sense.on_sense instances. This also elides the supporting annotation, such as the sense inventories needed to interpret and ground the sense annotations. For the details on each bank, look up the documentation for the referenced python module.
API REFERENCE

There are three top level divisions of the API. The first, tools contains simple tools for anticipated tasks such as loading data to the database. These tools are also designed to function as sample code for using the API. The second is a common module that holds utilities used by all the code. The final is the largest and most complex, the corpora module. It holds all the routines for dealing with individual banks: building them from files or the database, aligning them with each other, writing to files or the database, and querying the python objects about their contents.

5.1 on.common – Utility and Logging Functions

5.1.1 util – Utility functions

See:

- Dealing with config file, command line options, etc:
  - load_options()
  - load_config()
  - parse_cfg_args()
  - FancyConfigParser
  - register_config()
- Buckwalter Arabic encoding:
  - buckwalter2unicode()
  - unicode2buckwalter()
  - devocalize_buckwalter()
- DB:
  - esc()
  - insert_ignoring_dups()
  - is_db_ref()
  - make_db_ref()
  - is_not_loaded()
  - make_not_loaded()
- SGML (.name and .coref files):
• File System:
  - matches_an_affix()
  - mkdirs()
  - output_file_name()
  - sopen()
  -.listdir()
  - listdir_full()
  - listdir_both()

• Other:
  - bunch()
  - get_lemma()

Functions:

on.common.util.buckwalter2unicode(b_word, sgml_safety=True)
  Given a string in Buckwalter ASCII encoded Arabic, return the Unicode version.

on.common.util.unicode2buckwalter(u_word, sgml_safe=False, devocalize=False)
  Given a Unicode word, return the Buckwalter ASCII encoded version.
  If sgml_safe is set, run the output through make_sgml_safe() before returning.
  If devocalize is set delete a,u,i,o before returning.

on.common.util.register_config(section, value, allowed_values=[], doc=None, required=False, section_required=False, allow_multiple=False)
  make decorator so funcs can specify which config options they take.
  usage is:

  @register_config('corpus', 'load', 'specify which data to load to the db
  def load_banks(config):
    ...

  The special value '__dynamic' means that some config values are created dynamically and we can’t
  verify if a config argument is correct simply by seeing if it’s on the list. Documentation is also
  generated to this effect.

  If allowed_values is non-empty, then check to see that the setting the user chose is on the list.

  If allow_multiple is True, then when checking whether only allowed values are being given
  the key is first split on whitespace and then each component is tested.

  If required is True, then if the section exists it must specify this value. If the section does not
  exist, it is free to ignore this value. See section_required.

  If section_required is True, then issue an error if section is not defined by the user. Often
  wanted in combination with required.
on.common.util.insert_ignoring_dups(inserter, a_cursor, *values)

insert values to db ignoring duplicates

The caller can be a string, another class instance or a class:

- string : take to be an sql insert statement class : use it’s sql_insert_statement field, then proceed as with string instance: get it’s __class__ and proceed as with class

So any of the following are good:

- insert_ignoring_dups(self, a_cursor, id, tag)
- insert_ignoring_dups(cls, a_cursor, id, tag)
- insert_ignoring_dups(self.__class__.weirdly_named_sql_insert_statement, a_cursor, id, tag)

on.common.util.matches_an_affix(s, affixes)

Does the given id match the affixes?

Affixes = prefixes, suffixes

Given either a four digit string or a document id, return whether at least one of the prefixes and at least one of the suffixes matches it

on.common.util.output_file_name(doc_id, doc_type, out_dir='')

Determine what file to write an X_document to

doc_id: a document id doc_type: the type of the document, like a suffix (parse, prop, name, ...)
out_dir: if set, make the output as a child of out_dir

on.common.util.get_lemma(a_leaf, verb2morph, noun2morph, fail_on_not_found=False)

return the lemma for a_leaf’s word

if we have appropriate word2morph hashes, look the work up there. Otherwise just return the word. Functionally, for chinese we use the word itself and for english we have the hashes. When we get to doing arabic we’ll need to add a case.

if fail_on_not_found is set, return "" instead of a_leaf.word if we don’t have a mapping for this lemma.

on.common.util.load_config(cfg_name=None, config_append=[])

Load a configuration file to memory.

The given configuration file name can be a full path, in which case we simply read that configuration file. Otherwise, if you give ‘myconfig’ or something similar, we look in the current directory and the home directory. We also look to see if files with this name and extension ‘.conf’ exist. So for ‘myconfig’ we would look in the following places:

- */myconfig
- */myconfig.conf
- *[home]/.myconfig
- *[home]/.myconfig.conf

Once we find the configuration, we load it. We also extend ConfigParser to support [ ] notation. So you could look up key k in section s with config[s,k]. See FancyConfigParser().

If config_append is set we use parse_cfg_args() and add any values it creates to the config object. These values override any previous ones.

on.common.util.mkdir(long_path)

Make the given path exist. If the path already exists, raise an exception.
on.common.util.load_options(parser=None, argv=[], positional_args=True)
parses sys.argv, possibly exiting if there are mistakes

If you set parser to a ConfigParser object, then you have control over the usage string and you can
prepopulate it with options you intend to use. But don’t set a --config/-c option; load_options
uses that to find a configuration file to load

If a parser was passed in, we return (config, parser, [args]). Otherwise we return
(config, [args]). Args is only included if positional_args is True and there are posi-
tional arguments

See load_config() for details on the --config option.

on.common.util.parse_cfg_args(arg_list)
Parse command-line style config settings to a dictionary.

If you want to override configuration file values on the command line or set ones that were not set,
this should make it simpler. Given a list in format [section.key=value, ...] return a dictionary in form
{ (section, key): value, ... }.

So we might have:

['corpus.load=english-mz',
 'corpus.data_in=/home/user/corpora/ontonotes/data/']

we would then return the dictionary:

{ ('corpus', 'load') : 'english-mz',
  ('corpus', 'data_in') : '/home/user/corpora/ontonotes/data/'}

See also load_config() and load_options()

on.common.util.listdir(dirname)
List a dir’s child dirs, sorted and without hidden files.

Basically os.listdir(), sorted and without hidden (in the Unix sense: starting with a '.') files.

on.common.util.listdir_full(dirname)
A full path to file version of on.common.util.listdir().

on.common.util.listdir_both(dirname)
return a list of short_path, full_path tuples
identical to zip(listdir(dirname), listdir_full(dirname))

exception on.common.util.NotInConfigError
Because people might want to use a dictionary in place of a ConfigParser object, use a
NotInConfigError as the error to catch for config[section, value] call. For example:

try:
    load_data(config['Data', 'data_location'])
except on.common.util.NotInConfigError:
    print 'Loading data failed. Sorry.'

class on.common.util.bunch(**kwargs)
a simple class for short term holding related variables
change code like:
```python
def foo_some(a_onetonotes, b_onetonotes):
a_sense_bank = ...
a_onetonotes.foo(a_sense_bank)
a_...
a_

b_sense_bank = ...
b_onetonotes.foo(b_sense_bank)
b_...
b_

big_func(a_bar, b_bar)
```

To:

```python
def foo_some():
a = bunch(ontonotes=a_onetonotes)
b = bunch(ontonotes=b_onetonotes)

for v in [a, b]:
    v.sense_bank = ...
    v.ontonotes.foo(v.sense_bank)
    v. ...
    v. ...

big_func(a.bar, b.bar)
```

Or:

```python
def foo_some():
    def foo_one(v):
        v.sense_bank = ...
        v.ontonotes.foo(v.sense_bank)
        v. ...
        v. ...
        return v

    big_func(foo_one(bunch(ontonotes=a_onetonotes)).bar,
             foo_one(bunch(ontonotes=b_onetonotes)).bar)
```

Basically it lets you group similar things. It’s adding hierarchy to the local variables. It’s a hash table with more convenient syntax.

```python
on.common.util.is_db_ref(a_hash)
```

Is this hash a reference to the database?

If a hash (sense inventories, frames, etc) is equal to {'DB': a_cursor} that means instead of using the hash as information we should go look for our information in the database instead.

```python
on.common.util.make_db_ref(a_cursor)
```

Create a hash substitute that means ‘go look in the db instead’.

```python
See is_db_ref()
```

```python
on.common.util.is_not_loaded(a_hash)
```

Do we have no intention of loading the data a_hash is supposed to contain?
If a hash has a single key ‘NotLoaded’ that means we don’t intend to load that hash and we shouldn’t complain about data inconsistency involving the hash. So if we’re loading senses and the sense_inventory_hash is_not_loaded() then we shouldn’t drop senses for being references againstlemmas that don’t exist.

```
on.common.util.make_not_loaded()
    Create a hash substitute that means ‘act as if you had this information’

See is_not_loaded()
```

```
on.common.util.esc(*varargs)
given a number of arguments, return escaped (for mysql) versions of each of them
```

```
on.common.util.make_sgml_safe(s, reverse=False, keep_turn=True)
return a version of the string that can be put in an sgml document

This means changing angle brackets and ampersands to ‘-LAB-‘, ‘-RAB-‘, and ‘-AMP-‘. Needed for creating .name and .coref files.

If keep_turn is set, <TURN> in the input is turned into [TURN], not turned into -LAB-TURN-RAB-
```

```
on.common.util.make_sgml_unsafe(s)
return a version of the string that has real <, >, and &.

Convert the ‘escaped’ versions of dangerous characters back to their normal ascii form. Needed for reading .name and .coref files, as well as any other sgml files like the frames and the sense inventories and pools.

See make_sgml_safe()
```

```
class on.common.util.FancyConfigParser(defaults=None)
make a config parser with support for config[section, value]

raises FancyConfigParserError on improper usage.
```

### 5.1.2 log – Logging and reporting functions

See:

- error()
- warning()

Functions:

```
on.common.log.error(error_string, terminate_program=True, current_frame=False)
    Print error messages to stderr, optionally sys.exit.

on.common.log.warning(warning_string, verbosity=0)
    print warning string depending on the value of on.common.log.VERBOSITY

on.common.log.info(text, newline=True)
    write the text to standard error followed by a newline

on.common.log.debug(debug_object, debug_flag, verbosity=0)
    write each argument to stderr, space separated, with a trailing newline
```

Chapter 5. API Reference
5.2 on.corpora – classes for interpreting annotation

class on.corpora.subcorpus(a_ontonotes, physical_root_dir, cursor=None, prefix=[], suffix=[], lang=None, source=None, genre=None, strict_directory_structure=False, extensions=['parse', 'prop', 'sense', 'parallel', 'coref', 'name', 'speaker'], max_files='', old_id='')

A subcorpus represents an arbitrary collection of documents.

Initializing

The best way to deal with subcorpora is not to initialize them yourself at all. Create an ontonotes object with the config file, then ask it about its subcorpora. See on.ontonotes.

The following may be too much detail for your purposes.

When you __init__ a subcorpus, that’s only telling it which documents to include. It doesn’t actually load any of them, just makes a list. The load_banks() method does the actual file reading.

Which collection of documents a subcorpus represents depends on how you load it. The main way to do this is to use the constructor of on.ontonotes.

Loading the subcorpus directly through its constructor is complex, but provides slightly more flexibility. You need to first determine how much you current directory structure matches the one that ontonotes ships with. If you left it in the format:

`.../data/<lang>/annotations/<genre>/<source>/<section>/<files>`

Then all you need to do is initialize on.corpora.subcorpus with:

```python
a_subcorpus = on.corpora.subcorpus(a_ontonotes, data_location)
```

where data_location is as much of the data as you want to load, perhaps `.../data/english/annotations/nw/wsj/03`.

If you’re not using the original directory structure, you need to specify lang, genre, and source (ex: `’english’, ‘nw’, and ‘wsj’`) so that ids can be correctly determined.

If you want to load some of the data under a directory node but not all, prefix and suffix let you choose to load only some files. All documents have a four digit numeric ID that identifies them given their language, genre, and source. As in, the document `.../data/english/annotations/nw/wsj/00/wsj_0012` (which has multiple files (`parse`, `name`, `sense`, ...)) has id 0012. Prefix and suffix are lists of strings that have to match these IDs. If you set prefix to `[’0’, ’11’, ’313’] then the only documents considered will be those with ids starting with ’0’, ’11’ or ’313’. Similarly with suffix. So:

```python
prefix = [’00’, ’01’], suffix = [’1’, ’2’, ’3’, ’4’]
```

means we’ll load (for cnn):

```python
cnn_0001
cnn_0002
... 
cnn_0004
```
but no files whose ids end not in 1, 2, 3, or 4 or whose ids start with anything except ‘00’ and ‘01’.

Using

A subcorpus that’s been fully initialized always contains a treebank, and generally contains other banks. To access a bank you can use [] syntax. For example, to access the sense bank, you could do:

```python
a_sense_bank = a_subcorpus['sense']
```

If you iterate over a subcorpus you get the names of all the loaded banks in turn. So you could do something like:

```python
for a_bank_name, a_bank in a_subcorpus.iteritems():
    print 'I found a %s bank and it had %d %s_documents' % (a_bank_name, len(a_bank))
```

`load_banks (config)`
Load the individual bank data for the subcorpus to memory

Once a subcorpus is initialized we know what documents it represents (as in cnn_0013) but we’ve not loaded the actual files (as in cnn_0013.parse, cnn_0013.sense, ...). We often only want to load some of these, so specify which extensions (prop, parse, coref) you want with the corpus.banks config variable

This code will, for each bank, load the files and then enrich the treebank with appropriate links. For example, enriching the treebank with sense data sets the `on.corpora.tree.tree.on_sense` attribute of every tree leaf that’s been sense tagged. Once all enrichment has happened, one can go through the trees and be able to access all the annotation.

(Minor exception: some name and coreference data is not currently fully aligned with the tree and is inaccessible in this manner)

`write_to_db (a_cursor, only_these_banks=[])`  
Write the subcorpus and all files and banks within to the database.

Generally it’s better to use `on.ontonotes.write_to_db()` as that will write the type tables as well. If you don’t, perhaps for reasons of memory usage, write individual subcorpora to the database, you need to call `on.ontonotes.write_type_tables_to_db()` after the last time you call `write_to_db()`.

**Parameters:**

- `a_cursor` – The output of `on.ontonotes.get_db_cursor()`
- `only_these_banks` – if set, load only these extensions to the db

`copy ()`
make a duplicate of this subcorpus that represents the same documents

Note: if you had already loaded some banks these are absent in the copy.

`backed_by ()`
Returns either ‘db’ or ‘fs’

We can be pulling our data from the database or the filesystem depending on how we were created. Note that even if we’re reading from the file system, if the db is available we use it for sense inventory and frame lookups.
__getitem__(key)
The standard way to access individual banks is with [] notation.

The keys are extensions. To iterate over multiple banks in parallel, do something like:

```python
for a_tree_doc, a_sense_doc in zip(a_subcorpus['parse'], a_subcorpus['sense']):
    pass
```

Note that this will not work if some parses do not have sense documents.

**all_banks** *(standard_extension)*
The way to get a list of all banks of a type.

For example, if you have:

```
cnn_0000.no_traces_parse
cnn_0000.auto_traces_parse
cnn_0000.parse
```

If you want to iterate over all trees in all treebanks, you could do:

```python
for a_treebank in a_subcorpus.all_banks('parse'):
    for a_tree_document in a_treebank:
        for a_tree in a_tree_document:
            pass
```

class on.corpora.abstract_bank(a_subcorpus, tag, extension)
A superclass for all bank classes

All banks support the following psuedocode usage:

```python
if load_from_files:
    a_some_bank = some_bank(a_subcorpus, tag[, optional arguments])
else: # load from db
    a_some_bank = some_bank.from_db(a_subcorpus, tag, a_cursor[, opt_args])

# only if a_some_bank is not a treebank
a_some_bank.enrich_treebank(a_treebank[, opt_args])

for a_some_document in a_some_bank:
    # get the corresponding another document
    another_document = another_bank.get_document(a_some_document)

if write_to_files:
    a_some_document.dump_view(a_cursor=None, out_dir)
else: # write to db
    a_some_document.write_to_db(a_cursor)
```

See:

- `on.corpora.tree.treebank`
- `on.corpora.sense.sense_bank`
- `on.corpora.proposition.proposition_bank`
- `on.corpora.coreference.coreference_bank`
### 5.2.1 tree – Syntactic Parse Annotation

See:
- `on.corpora.tree.tree`
- `on.corpora.tree.treebank`

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>treebank</td>
<td>treebank</td>
<td>All .parse files for a <code>on.corpora.subcorpus</code></td>
</tr>
<tr>
<td>None</td>
<td>tree_document</td>
<td>A .parse file</td>
</tr>
<tr>
<td>tree</td>
<td>tree</td>
<td>An S-expression in a .parse file</td>
</tr>
<tr>
<td>syntactic_link</td>
<td>syntactic_link</td>
<td>The numbers after ‘~’ and ‘=’ in trees</td>
</tr>
<tr>
<td>lemma</td>
<td>lemma</td>
<td>.lemma files (arabic only)</td>
</tr>
</tbody>
</table>

The `treebank` class represents a collection of `tree_document` classes and provides methods for manipulating trees. Further, because annotation in other banks was generally done relative to these parse trees, much of the code works relative to the trees. For example, the `on.corpora.document` data, their `on.corpora.sentence` data, and their `on.corpora.token` data are all derived from the trees.

Attributes:

- `banks`
  - A hash from standard extensions (coref, name, ...) to bank instances

The following two attributes are set during enrichment with parallel banks. For sentence level annotation, see `tree.translations` and `tree.originals`. 
translations
A list of tree_document instances in other subcorpora that represent translations of this document

original
A single tree_document instance representing the original document that this one was translated from. It doesn’t make sense to have more than one of these.

Methods:

sentence_tokens_as_lists (make_sgml_safe=False, strip_traces=False)
all the words in this document broken into lists by sentence

So ‘Good morning. My name is John.’ becomes:

[['Good', 'morning', '.'],  
['My', 'name', 'is', 'john', '.']]

This doesn’t actually return a list, but instead a generator. To get this as a (very large) list of lists, just call it as list(a_tree_document.sentence_tokens_as_lists())

If `make_sgml_safe` is True, on.common.util.make_sgml_safe() is called for each word.

If `strip_traces` is True, trace leaves are not included in the output.

class on.corpora.tree.tree (tag, word=None, document_tag='gold')
root trees, internal nodes, and leaves are all trees.

Contained by: tree_document if a root tree, tree otherwise Contains: None if a leaf, tree otherwise

Attributes:

Always available:

parent
The parent of this tree. If None then we are the root

lemma
Applicable only to Arabic leaves. The morphological lemma of the word as a string. In Chinese the word is the lemma, so use word. In English the best you can do is use either the lemma attribute of on_sense or the lemma attribute of proposition.

See Also: lemma_object

lemma_object
Applicable only to Arabic leaves. There is a lot more information in the .lemma file for each leaf than just the lemma string, so if available a lemma instance is here.

word
Applicable only to leaves. The word of text corresponding to the leaf. To extract all the words for a tree, see get_word_string().

For arabic, the word of a tree is always the vocalized unicode representation. For other representations, see the get_word() method.

tag
Every node in the tree has a tag, which represents part of speech, phrase type, or function type information. For example, the leaf (NNS cabbages) has a tag of NNS while the subtree (NP (NNS cabbages)) has a tag of NP.

children
A list of the child nodes of this tree. For leaves this will be the empty list.
reference_leaves
   A list of trace leaves in this tree that point to this subtree

identity_subtree
   The subtree in this tree that this trace leaf points to

Available only after enrichment:

The following attributes represent the annotation. They are set during the enrichment process, which happens automatically unless you are invoking things manually at a low level. You must, of course, specify that a bank is to be loaded for its annotations to be available. For example, if the configuration variable \texttt{corpora.banks} is set to "parse sense", then leaves will have \texttt{on\_sense} attributes but not \texttt{proposition} attributes.

Each annotation variable specifies its level, the bank that sets it, and the class whose instance it is set to. Leaf level annotation applies only to leaves, tree level annotation only to sentences, and subtree annotation to any subtree in between, including leaves.

Order is not significant in any of the lists.

on\_sense
   Leaf level, sense bank, \texttt{on\_sense}

proposition
   Subtree level, proposition bank, \texttt{proposition}
   This is attached to the same subtree as the primary predicate node of the proposition.

predicate_node_list
   Subtree level, proposition bank, list of \texttt{predicate\_node}

argument_node_list
   Subtree level, proposition bank, list of \texttt{argument\_node}

link_node_list
   Subtree level, proposition bank, list of \texttt{link\_node}

named_entity
   Subtree level, name bank, \texttt{name\_entity}

start\_named\_entity\_list
   Leaf level, name bank, list of \texttt{name\_entity}
   Name entities whose initial word is this leaf.

end\_named\_entity\_list
   Leaf level, name bank, list of \texttt{name\_entity}
   Name entities whose final word is this leaf.

coreference_link
   Subtree level, coreference bank, \texttt{coreference\_link}

coreference_chain
   Subtree level, coreference bank, \texttt{coreference\_chain}
   The coreference chain that \texttt{coreference\_link} belongs to.

start\_coreference\_link\_list
   Leaf level, coreference bank, list of \texttt{coreference\_link}
   Coreference links whose initial word is this leaf.

end\_coreference\_link\_list
   Leaf level, coreference bank, list of \texttt{coreference\_link}
Coreference links whose final word is this leaf.

**coref_section**
- Tree level, coreference bank, string
  
  The Broadcast Conversation documents, because they are very long, were divided into sections for coreference annotation. We tried to break them up at natural places, those where the show changed topic, to minimize the chance of cross-section coreference. The annotators then did standard coreference annotation on each section separately as if it were its own document. Post annotation, we merged all sections into one .coref file, with each section as a TEXT span. So you can have a pair of references to John Smith in section 1 and another pair of references in section 2, but they form two separate chains. That is, every coreference chain is within only one coreference section.

**translations**
- Tree level, parallel_bank, list of tree
  
  Trees (in other subcorpora) that are translations of this tree

**originals**
- Tree level, parallel_bank, list of tree
  
  Trees (in other subcorpora) that are originals of this tree

**speaker_sentence**
- Tree level, speaker bank, speaker_sentence

Methods:

- **is_noun**()
  
  Is the part of speech of this leaf NN or NNS assuming the Penn Treebank’s tagset?

- **is_verb**()
  
  Is the part of speech of this leaf VN or VBX for some X assuming the Penn Treebank’s tagset?

- **is_aux**(prop=False)
  
  Does this leaf represent an auxilliary verb?
  
  Note: only makes sense if english
  
  All we do is say that a leaf is auxilliary if:
  
  • it is a verb
  
  • the next leaf (skipping adverbs) is also a verb
  
  This does not deal with all cases. For example, in the sentence ‘Have you eaten breakfast?’, the initial ‘have’ is an auxilliary verb, but we report it as not so. There should be no false positives, but we don’t get all the cases.

  If the argument ‘prop’ is true, then we use a less restrictive definition of auxilliary that represents closer what is legal for proptaggers to tag. That is, if we have a verb following a verb, and the second verb is under an NP, don’t count the first verb as aux.

- **is_leaf**()
  
  does this tree node represent a leaf?

- **is_trace**()
  
  does this tree node represent a trace?

- **is_root**()
  
  check whether the tree is a root

- **is_punct**()
  
  is this leaf punctuation?
is_conj()
   is this leaf a conjunction?

is_trace_indexed()
   if we’re a trace, do we have a defined reference index?

is_trace_origin()
   if we’re a trace, do we have a defined identity index?

get_root()
   Return the root of this tree, which may be ourself.

get_subtree(a_id)
   return the subtree with the specified id

get_leaf_by_word_index(a_word_index)
   given a word index, return the leaf at that index

get_leaf_by_token_index(a_token_index)
   given a token index, return the leaf at that index

get_subtree_by_span(start, end)
   given start and end of a span, return the highest subtree that represents it
   The arguments start and end may either be leaves or token indecies.
   Returns None if there is no matching subtree.

pretty_print(offset='', buckwalter=False, vocalized=True)
   return a string representing this tree in a human readable format

get_word(buckwalter=False, vocalized=True, clean_speaker_names=True, interpret_html_escapes=True, indexed_traces=True)

get_word_string(buckwalter=False, vocalized=True)
   return the words for this tree, separated by spaces.

get_trace_adjusted_word_string(buckwalter=False, vocalized=True)
   The same as get_word_string() but without including traces

get_plain_sentence()
   display this sentence with as close to normal typographical conventions as we can.
   Note that the return value of this function does not follow ontonotes tokenization.

pointer(indexing='token')
   (document_id, tree_index, sentence_index)
   Return a triple in the format of the pointers in a sense or prop file.

fix_trace_index_locations()
   Reconcile the two forms of trace index notation; set up syntactic link pointers
   All trees distributed with ontonotes have been through this process.
   There are two forms used in the annotation, one by ann taylor, the other by the ldc.
   terminology note: we’re using ‘word’ and ‘tag’ as in (tag word) and (tag (tag word))
   In the original Penn Treebank system, a trace is a link between exactly two tree nodes, with the
target index on the tag of the parent and the reference index on the word of the trace. If there are
more than one node in a trace, they’re chained, with something like:

   (NP-1 (NP target)) ...
   (NP-2 (-NONE- +1)) ...
   (NP (-NONE- +2)) ...

   \begin{verbatim}
   (NP-1 (NP target)) ...
   (NP-2 (-NONE- +1)) ...
   (NP (-NONE- +2)) ...
   \end{verbatim}
In the LDC system a trace index can apply to arbitrarily many nodes and all indecies are on the tag of the parent. So this same situation would be notated as:

\[(NP-1 \ (NP \ target)) \ldots \]
\[(NP-1 \ (-NONE- \ *)) \ldots \]
\[(NP-1 \ (-NONE- \ *)) \ldots \]

We’re leaving everything in the original Penn Treebank format alone, but changing the ldc format to a hybrid mode where there can be multiple nodes in a trace chain, but the reference indecies are on the words:

\[(NP-1 \ (NP \ target)) \ldots \]
\[(NP \ (-NONE- \ -*1)) \ldots \]
\[(NP \ (-NONE- \ -*1)) \ldots \]

There are a few tricky details:

- We have to be able to tell by looking at a tree which format it’s in
  - if there are ever words with trace indicies, this mean’s we’re using the original Penn Treebank format
- We might not be able to tell what the target is
  - The ideal case has one or more -NONE- tags on traces and exactly one without -NONE-
  - If there are more than one without -NONE- then we need to pick one to be the target.
    Choose the leftmost
  - If there is none without -NONE- then we also need to pick one to be the target. Again choose the leftmost.

We also need to deal with gapping. This is for sentences like:

‘Mary likes Bach and Susan, Bethoven’

These are notated as:

\[(S \ (S \ (NP-SBJ=1 \ Mary) \]
\[(VP \ likes \]
\[(NP=2 \ Bach))\)
\] and
\[(S \ (NP-SBJ=1 \ Susan) \]
\[(NP=2 \ Beethoven))\)

in the LDC version, and as:

\[(S \ (S \ (NP-SBJ=1 \ Mary) \]
\[(VP \ likes \]
\[(NP-2 \ Bach))\)
\] and
\[(S \ (NP-SBJ=1 \ Susan) \]
\[(NP=2 \ Beethoven))\)

in the original Penn Treebank version.

We convert them all to the original Penn Treebank version, with the target having a single hyphen and references having the double hyphens.

There can also be trees with both gapping and normal traces, as in:

\[(NP \ fears \]
\[(SBAR \ that \]
\[(S \ (S \ (NP-SBJ=2 \ the \ Thatcher \ government) \]
\[(VP \ may \ (VP \ be \ (PP-PRD=3 \ in \ (NP \ turmoil))))\)])\)
\] and
OntoNotes DB Tool Documentation, Release 0.999b

(S (NP-SBJ-1=2 (NP Britain ’s) Labor Party)
 (VP=3 positioned
  (S (NP-SBJ-1 *)
   (VP to (VP regain (NP (NP control)
    (PP of (NP the government)))))))))))

So we need to deal with things like ‘NP-SBJ-1=2’ properly.

Also set up leaf.reference_leaves and leaf.identity_subtree

get_sentence_index()
the index of the sentence (zero-indexed) that this tree represents

get_word_index(sloppy=False)
the index of the word in the sentence not counting traces
sloppy is one of: False: be strict ‘next’ : if self is a trace, take the next non-trace leaf ‘prev’ : if self is a trace, take the prev non-trace leaf

get_token_index()
the index of the word in the sentence including traces

get_height()
how many times removed this node is from its initial leaf.

Examples:
• height of (NNS cabbages) is 0
• height of (NP (NNS cabbages)) is 1
• height of (PP (IN of) (NP (NNS cabbages))) is also 1 because (IN of) is its initial leaf

Used by propositions

leaves (regen_cache=False)
generate the leaves under this subtree

subtrees (regen_cache=False)
generate the subtrees under this subtree, including this one

order is always top to bottom; if A contains B then index(A) < index(B)

__getitem__(x)
get a leaf, list of leaves, or subtree of this tree

The semantics of this when used with a slice are tricky. For many purposes you would do better to use one of the following instead:
• tree.leaves() – if you want a list of leaves
• tree.subtrees() – if you want a list of subtrees
• tree.get_subtree_by_span() – if you want a subtree matching start and end leaves or indexes.

This function is nice, though, especially for interactive use. The logic is:
• if the argument is a single index, return that leaf, or index error if it does not exist
• otherwise think of the tree as a list of leaves. The argument is then interpreted just as list.__getitem__ does, with full slice support.
• if such interpretation leads to a list of leaves that is a proper subtree of this one, return that subtree
  – note that if a subtree has a single child, two such subtrees can match. If more than one matches, we take the highest one.
• if all the slice can be interpreted to represent is an arbitrary list of leaves, return that list.

For example, consider the following tree:
The simplest thing we can do is look at individual leaves, such as tree[2]:

(NNP Heartland)

Note that leaves act as subtrees, so even if we index a leaf, tree[2][0] we get it back again:

(NNP Heartland)

If we look at a valid subtree, like with tree[0:4], we see:

(PP-MNR (IN Like)
   (NP (JJ many)
      (NNP Heartland)
      (NNS states)))

If our indexes do not fall exactly on subtree bounds, we instead get a list of leaves:

[(IN Like),
   (JJ many),
   (NNP Heartland),
   (NNS states),
   (, ,)]

Extended slices are supported, though they’re probably not very useful. For example, we can make a list of the even leaves of the tree in reverse order with tree[::-2]:

[(. .),
   (NN state),
   (RB anywhere),
   (NN farm),
   (IN on),
   (NNS people),
   (VBG keeping),
   (NN trouble),
   (VBZ has),]
get_other_leaf(index)

Get leaves relative to this one. An index of zero is this leaf, negative one would be the previous leaf, etc. If the leaf does not exist, we return None

class on.corpora.tree.lemma(input_string, b_transliteration, comment, index, offset, unvocalized_string, vocalized_string, vocalized_input, pos, gloss, lemma, coarse_sense, leaf_id)

arabic trees have extra lemma information

class on.corpora.tree.syntactic_link(type, word, reference_subtree_id, identity_subtree_id)

Links between tree nodes

Example:

(TOP (SBARQ (WHNP-1 (WHADJP (WRB How) (JJ many)))
  (NNS ups))
  (CC and)
  (NNS downs))
(SQ (MD can)
  (NP-SBJ (CD one)
    (NN woman)))
  (VP (VB have)
    (NP (-NONE- *T*-1))))
  (. /??))

The node (-NONE- *T*-1) is a syntactic link back to (WHNP-1 (WHADJP (WRB How) (JJ many)) (NNS ups) (CC and) (NNS downs)).

Links have an identity subtree (How many ups and downs) and a reference subtree (-NONE- *T*-1) and are generally thought of as a link from the reference back to the identity.

class on.corpora.tree.compound_function_tag(a_function_tag_string, subtree)

class on.corpora.tree.tree_exception

5.2.2 proposition – Proposition Annotation

See:

• on.corpora.proposition.proposition
• on.corpora.proposition.proposition_bank
• on.corpora.proposition.frame_set
• on.corpora.sense.pb_sense_type

Correspondences:
### Database Tables

<table>
<thead>
<tr>
<th>Database Table</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition_bank</td>
<td>proposition_bank</td>
<td>All .prop files in an on.corpora.subcorpus</td>
</tr>
<tr>
<td>None</td>
<td>proposition</td>
<td>A single .prop file</td>
</tr>
<tr>
<td>proposition</td>
<td>proposition</td>
<td>A line in a .prop file, with everything after the ----- an “argument field”</td>
</tr>
<tr>
<td>predicate_analogue</td>
<td>predicate_analogue</td>
<td>REL argument fields (should only be one)</td>
</tr>
<tr>
<td>None</td>
<td>argument_analogue</td>
<td>ARG argument fields</td>
</tr>
<tr>
<td>argument_analogue</td>
<td>argument_analogue</td>
<td>LINK argument fields</td>
</tr>
<tr>
<td>link_analogue</td>
<td>link_analogue</td>
<td>Asterisk-separated components of a predicate_analogue. Each part is coreferential.</td>
</tr>
<tr>
<td>predicate_analogue</td>
<td>predicate_analogue</td>
<td>Asterisk-separated components of an argument_analogue. Each part is coreferential.</td>
</tr>
<tr>
<td>predicate_analogue</td>
<td>predicate_analogue</td>
<td>Asterisk-separated components of a link_analogue. Each part is coreferential.</td>
</tr>
<tr>
<td>predicate_analogue</td>
<td>predicate_analogue</td>
<td>Comma-separated components of predicates. The parts together make up the predicate.</td>
</tr>
<tr>
<td>argument_analogue</td>
<td>argument_analogue</td>
<td>Comma-separated components of arguments. The parts together make up the argument.</td>
</tr>
<tr>
<td>link_analogue</td>
<td>link_analogue</td>
<td>Comma-separated components of links. The parts together make up the link.</td>
</tr>
<tr>
<td>None</td>
<td>frame_set</td>
<td>An xml frame file (FF)</td>
</tr>
<tr>
<td>pb_sense_type</td>
<td>on.corpora.sense</td>
<td>Field six of a prop line and a FF’s frameset/predicate/roleset element’s id attribute</td>
</tr>
<tr>
<td>pb_sense_type</td>
<td>pb_sense_type</td>
<td>pb_sense_type/argument_types mapping between roleset.id and roleset/role.n</td>
</tr>
<tr>
<td>tree</td>
<td>on.corpora.tree</td>
<td>The first three fields of a prop line</td>
</tr>
</tbody>
</table>

This may be better seen with an example. The prop line:

```plaintext
bc/cnn/00/cnn_0000@all@cnn@bc@en@on 191 3 gold say-v say.01 ----- 1:1-ARGM-DIS 2:1-ARG0 3:0-rel 4:1*6:1,8:1-ARG1
```

breaks up as:

<table>
<thead>
<tr>
<th>Python Object</th>
<th>File Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition</td>
<td>bc/cnn/00/cnn_0000@all@cnn@bc@en@on 191 3 gold say-v say.01 ----- 1:1-ARGM-DIS 2:1-ARG0 3:0-rel 4:1*6:1,8:1-ARG1</td>
</tr>
<tr>
<td>on.corpora.sense</td>
<td>bc/cnn/00/cnn_0000@all@cnn@bc@en@on 191 3</td>
</tr>
<tr>
<td>predicate_analogue</td>
<td>3:0<em>1:1-ARGM-DIS,2:1-ARG0, and 4:1</em>6:1,8:1-ARG1</td>
</tr>
<tr>
<td>predicate_node</td>
<td>each of 1:1,2:1,4:1, and 6:1,8:1</td>
</tr>
<tr>
<td>predicate</td>
<td>each of 1:1,2:1,4:1, and 6:1,8:1</td>
</tr>
<tr>
<td>argument_analogue</td>
<td>each of 1:1,2:1,4:1, and 6:1,8:1</td>
</tr>
<tr>
<td>argument</td>
<td>each of 1:1,2:1,4:1, and 6:1,8:1</td>
</tr>
<tr>
<td>argument_node</td>
<td>each of 1:1,2:1,4:1, and 6:1,8:1</td>
</tr>
</tbody>
</table>

Similarly, the prop line:

```plaintext
bc/cnn/00/cnn_0000@all@cnn@bc@en@on 309 5 gold go-v go.15 ----- 2:0*1:1-ARG1 4:1-ARGM-ADV 5:0,6:1-rel
```

breaks up as:
### Python Object | File Text
---|---
proposition | bc/cnn/00/cnn_0000@all@cnn@bc@en@on 309 5 gold go-v go.15 ------ 2:0*1:1-ARG1 4:1-ARGM-ADV 5:0,6:1-rel
on.corpora.tree | bc/cnn/00/cnn_0000@all@cnn@bc@en@on 309 5
predicate_analogue | 5:0,6:1-rel
predicate | each of 5:0 and 6:1
argument_analogue | each of 2:0*1:1-ARG1 and 4:1-ARGM-ADV
each of 2:0,1:1 and 4:1
argument_node | each of 2:0,1:1, and 4:1

Classes:

```python
class on.corpora.proposition.proposition_bank(a_subcorpus, tag, a_cursor=None, extension='prop', a_frame_set_hash=None)
```

Extends: on.corpora.abstract_bank
Contains: proposition_document

```python
class on.corpora.proposition.proposition_document(document_id, extension='prop')
```

Contained by: proposition_bank
Contains: proposition

```python
class on.corpora.proposition.proposition(encoded_prop, subcorpus_id, document_id, a_proposition_bank=None, tag=None)
```

a proposition annotation; a line in a .prop file
Contained by: proposition_document
Contains: predicate_analogue, argument_analogue, and link_analogue (in that order)
Attributes:

- **lemma**
  Which frame_set this leaf was annotated against
- **pb_sense_num**
  Which sense in the frame_set the arguments are relative to
- **predicate**
  A predicate_analogue instance
- **quality**
  gold double annotated, adjudicated, release format
- **type**
  v standard proposition
  n nominalization proposition
- **argument_analogues**
  A list of argument_analogue
- **link_analogue**
  A list of link_analogue
- **document_id**
- **enc_prop**
  This proposition and all it contains, encoded as a string. Lines in the .prop files are in this format.

Methods:
write_to_db(cursor)
write this proposition and all its components to the database

__getitem__(idx)
return first the predicate_analogue, then the argument analogues, then the link analogues

class on.corpora.proposition.abstract_proposition_bit(a_parent)
any subcomponent of a proposition after the '-----'.

Attributes:

id
index_in_parent
lemma
pb_sense_num
proposition
document_id
enc_self
Encode whatever we represent as a string, generally by combining the encoded representations
of sub-components

Class Hierarchy:

- abstract_proposition_bit – things with parents
  - abstract_node
  - abstract_holder
- abstract_node – things that align with subtrees
  - argument_node
  - predicate_node
  - link_node
- abstract_holder – things with children
  - abstract_analogue
  - abstract_node_holder
- abstract_analogue – children are coreferential (split on '*')
  - argument_analogue
  - predicate_analogue
  - link_analogue
- abstract_node_holder – children together make up an entity (split on ',')
  - argument
  - predicate
  - link

class on.corpora.proposition.abstract_holder(sep, a_parent)
represents any proposition bit that holds other proposition bits

Extends abstract_proposition_bit

5.2. on.corpora – classes for interpreting annotation
See:

- `abstract_analogue`
- `abstract_node_holder`

**class** `on.corpora.proposition.abstract_analogue(a_parent, a_analogue_type)`

represents argument_analogue, predicate_analogue, link_analogue

Example: `0:1,3:2*2:0-ARGM`

Extends: `abstract_holder`

Represents:

- argument_analogue
- predicate_analogue
- link_analogue

This class is used for the space separated portions of a proposition after the ‘-----’

All children are coreferential, and usually all but one are traces.

**class** `on.corpora.proposition.abstract_node_holder(a_parent)`

represents argument, predicate, link

Example: `0:1,3:2`

Extends: `abstract_holder`

Represents:

- argument
- predicate
- link

This class is used for any bit of a proposition which has representation A,B where A and B are nodes

**class** `on.corpora.proposition.abstract_node(sentence_index, token_index, height, parent)`

represents argument_node, predicate_node, and link_node

Example: `0:1`

Extends: `abstract_proposition_bit`

Represents:

- argument_node
- predicate_node
- link_node

This class is used for any bit of a proposition which has representation A:B

Attributes:

- `sentence_index` which tree we’re in
- `token_index` which leaf in the tree we are
- `height` how far up from the leaves we are (a leaf is height 0)
parent
   an abstract_node_holder to add yourself to

subtree
   which on.corpora.tree.tree we’re aligned to. None until enrichment.

is_ich_node
   True only for argument nodes. True when proposition taggers would separate this node from
   others with a ‘;’ in the encoded form. That is, True if the subtree we are attached to is indexed
   to an *ICH* leaf or we have an *ICH* leaf among our leaves, False otherwise.

errcoms
   This is a list, by default the empty list. If errors are found in loading this proposition, strings
   that can be passed to on.common.log.reject() or on.common.log.adjust() are
   appended to it along with comments, like:

   errcomms.append([‘reason’, [‘explanation’, ‘details’, ...]])

Initially, a node is created with sentence and token indecies. During enrichment we gain a reference to a
on.corpora.tree.tree instance. After enrichment, requests for sentence and token indecies are forwarded to the subtree.

class on.corpora.proposition.predicate_analogue (enc_predicates, a_type, sentence_index,
   token_index, a_proposition)

   The REL-tagged field of a proposition.
   Extends: abstract_analogue
   Contained by: proposition
   Contains: predicate

class on.corpora.proposition.predicate (enc_predicate, sentence_index, token_index,
   a_predicate_analogue)

   Extends: abstract_node_holder
   Contained by: predicate_analogue
   Contains: predicate_node

class on.corpora.proposition.predicate_node (sentence_index, token_index, height,
   a_predicate, primary=False)

   represents the different nodes of a multi-word predicate
   Extends: abstract_node
   Contained by: predicate

   Attributes:

   a_predicate
      on.corpora.proposition.predicate

   sentence_index
      which tree in the document do we belong to

   token_index
      token index of this node within the predicate’s tree

   height
      how far up in the tree from the leaf at token_index we need to go to get the subtree this node
      represents

   primary
      are we the primary predicate?
class on.corpora.proposition.argument_analogue(eng_argument_analogue, a_proposition)
    Extends: abstract_analogue
    Contained by: proposition
    Contains: argument

class on.corpora.proposition.argument(eng_argument, a_argument_analogue)
    Extends: abstract_node_holder
    Contained by: argument_analogue
    Contains: argument_node

class on.corpora.proposition.argument_node(sentence_index, token_index, height, a_argument)
    Extends: abstract_node
    Contained by: argument

class on.corpora.proposition.link_analogue(eng_links, a_type, a_proposition, a_associated_argument)
    Extends: abstract_analogue
    Contained by: proposition
    Contains: link

class on.corpora.proposition.link(eng_link, a_link_analogue)
    Extends: abstract_node_holder
    Contained by: link_analogue
    Contains: link_node

Attributes:

    associated_argument
    the argument_analogue this link is providing additional detail for

class on.corpora.proposition.link_node(sentence_index, token_index, height, parent)
    Extends: abstract_node
    Contained by: link

class on.corpora.proposition.frame_set(a_xml_string, a_subcorpus=None, lang_id=None)
    information for interpreting a proposition annotation

5.2.3 sense – Word Sense Annotation

See:

• on.corpora.sense.on_sense
• on.corpora.sense.sense_bank
• on.corpora.sense.on_sense_type

Word sense annotation consists of specifying which sense a word is being used in. In the .sense file format, a word sense would be annotated as:

This tells us that word 9 of sentence 6 in broadcast news document cnn_0001 has the lemma “fire”, is a noun, and has sense 4. The sense numbers, such as 4, are defined in the sense inventory files. Looking up sense 4 of fire-n in data/english/metadata/sense-inventories/fire-n.xml, we see:
Just knowing that word 9 of sentence 6 in some document has some sense is not very useful on its own. We need to match this data with the document it was annotated against. The python code can do this for you. First, load the data you’re interested in, to memory with `on.corpora.tools.load_to_memory`. Then we can iterate over all the leaves to look for cases where a_leaf was tagged with a noun sense “fire”:

```python
fire_n_leaves = []
for a_subcorpus in a_onetonotes:
    for a_tree_document in a_subcorpus["tree"]:
        for a_tree in a_tree_document:
            for a_leaf in a_tree.leaves():
                if a_leaf.on_sense:
                    if a_leaf.on_sense.lemma == "fire" and a_leaf.on_sense.pos == "n":
                        fire_n_leaves.append(a_leaf)
```

Now say we want to print the sentences for each tagged example of “fire-n”:

```python
# first we collect all the sentences for each sense of fire
sense_to_sentences = defaultdict(list)
for a_leaf in fire_n_leaves:
    a_sense = a_leaf.on_sense.sense
    a_sentence = a_leaf.get_root().get_word_string()
    sense_to_sentences[a_sense].append(a_sentence)

# then we print them
for a_sense, sentences in sense_to_sentences.iteritems():
    a_sense_name = on_sense_type.get_name("fire", "n", a_sense)
    print "Sense %s: %s" % (a_sense, a_sense_name)
    for a_sentence in sentences:
        print "  ", a_sentence

print ""
```

Correspondences:
<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>sense_bank</td>
<td>sense_bank</td>
<td>All .sense files in a on.corpora.subcorpus</td>
</tr>
<tr>
<td>None</td>
<td>sense_tagged_doc</td>
<td>A single .sense file</td>
</tr>
<tr>
<td>on_sense</td>
<td>on_sense</td>
<td>A line in a .sense file</td>
</tr>
<tr>
<td>None</td>
<td>sense_inventory</td>
<td>A sense inventory xml file (SI)</td>
</tr>
<tr>
<td>on_sense_type</td>
<td>on_sense_type</td>
<td>Fields four and six of a sense line and the inventory/sense element of a SI</td>
</tr>
<tr>
<td>on_sense_lemma</td>
<td>type_sense_lemma</td>
<td>The inventory/ita element of a SI</td>
</tr>
<tr>
<td>wn_sense_type</td>
<td>wn_sense_type</td>
<td>The inventory/sense/mappings/wn element of a SI</td>
</tr>
<tr>
<td>pb_sense_type</td>
<td>pb_sense_type</td>
<td>The inventory/sense/mappings/pb element of a SI</td>
</tr>
<tr>
<td>tree</td>
<td>on.corpora.tree</td>
<td>The first three fields of a sense line</td>
</tr>
</tbody>
</table>

Classes:

```python
class on.corpora.sense.sense_bank(a_subcorpus, tag, a_cursor=None, extension='sense', a_sense_inv_hash=None, a_frame_set_hash=None, indexing='word')

Extends: on.corpora.abstract_bank
Contains: sense_tagged_document
```

```python
class on.corpora.sense.sense_tagged_document(sense_tagged_document_string, document_id, a_sense_bank, a_cursor=None, preserve_ita=False, indexing='word')

Contained by: sense_bank
Contains: on_sense
```

```python
class on.corpora.sense.on_sense(document_id, tree_index, word_index, lemma, pos, ann_1_sense, ann_2_sense, adj_sense, sense, adjudicated_flag, a_cursor=None, indexing='word')

A sense annotation; a line in a .sense file.

Contained by: sense_tagged_document
```

Attributes:

- **lemma**: Together with the pos, a reference to a sense_inventory.
- **pos**: Either n or v. Indicates whether this leaf was annotated by people who primarily tagged nouns or verbs. This should agree with on.corpora.tree.tree.is_noun() and is_verb() methods for English and Arabic, but not Chinese.
- **sense**: Which sense in the sense_inventory the annotators gave this leaf.

```python
class on.corpora.sense.on_sense_type(lemma, pos, group, sense_num, name, sense_type)

Information to interpret on_sense annotations

Contained by: sense_inventory
```

Attributes:

- **lemma**
- **sense_num**
- **pos**: Either ‘n’ or ‘v’, depending on whether this is a noun sense or a verb sense.
**wn_sense_types**
list of *wn_sense_type* instances

**pb_sense_types**
list of *pb_sense_type* instances (frame senses)

**sense_type**
the type of the sense, such as ‘Event’

Methods:

```python
classmethod get_name(a_lemma, a_pos, a_sense)
given a lemma, pos, and sense number, return the name from the sense inventory
```

```python
class on.corpora.sense.on_sense_lemma_type(a_on_sense)
computes and holds ita statistics for a lemma/pos combination
```

```python
class on.corpora.sense.sense_inventory(a_fname, a_xml_string, a_lang_id, a_frame_set_hash={})
Contains: on_sense_type
```

```python
class on.corpora.sense.pb_sense_type(lemma, num)
A frame sense

Contained by: on.corpora.proposition.frame_set.on_sense_type
```

```python
class on.corpora.sense.wn_sense_type(lemma, wn_sense_num, pos, wn_version)
a wordnet sense, for mapping ontonotes senses to wordnet senses

Contained by: on_sense_type
```

---

### 5.2.4 coreference – Coreferential Entity Annotation

See:

- on.corpora.coreference.coreference_link
- on.corpora.coreference.coreference_bank

Coreference annotation consists of indicating which mentions in a text refer to the same entity. The `.coref` file format looks like this:

Correspondences:

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>coreference_bank</td>
<td>coreference_bank</td>
<td>All .coref files in an on.corpora.subcorpus</td>
</tr>
<tr>
<td>None</td>
<td>coreference_document</td>
<td>A .coref file (a DOC span)</td>
</tr>
<tr>
<td>tree.coreference_section</td>
<td>on.corpora.tree.tree.coref_section</td>
<td>An annotation section of a .coref file (a TEXT span)</td>
</tr>
<tr>
<td>tree</td>
<td>on.corpora.tree.tree</td>
<td>A line in a .coref file</td>
</tr>
<tr>
<td>coreference_chain</td>
<td>coreference_chain</td>
<td>All COREF spans with a given ID</td>
</tr>
<tr>
<td>coreference_chain.type</td>
<td>coreference_chain.type</td>
<td>The TYPE field of a coreference link (the same for all links in a chain)</td>
</tr>
<tr>
<td>coreference_chain.speaker</td>
<td>coreference_chain.speaker</td>
<td>The TYPE field of a coreference chain (the same for all links in a chain)</td>
</tr>
<tr>
<td>coreference_link</td>
<td>coreference_link</td>
<td>A single COREF span</td>
</tr>
<tr>
<td>coreference_link.type</td>
<td>coreference_link.type</td>
<td>The SUBTYPE field of a coreference link</td>
</tr>
</tbody>
</table>

Note that coreference section information is stored very differently the files than in the database and python objects. For more details see the on.corpora.tree.tree.coref_section documentation

---

### 5.2. on.corpora – classes for interpreting annotation

47
Classes:

```python
class on.corpora.coreference.coreference_bank(a_subcorpus, tag, a_cursor=None, extension='coref', indexing='token', messy_muc_input='false')
```

Contains: coreference_document

```python
class on.corpora.coreference.coreference_document(enc_doc_string, document_id, extension='coref', indexing='token', a_cursor=None, adjudicated=True, messy_muc_input=False)
```

Contained by: coreference_bank

Contains: coreference_chain

```python
class on.corpora.coreference.coreference_chain(type, identifier, section, document_id, a_cursor=None, speaker='')
```

Contained by: coreference_document

Contains: coreference_link

Attributes:

- **identifier**
  Which coref chain this is. This value is unique to this document, though not across documents.

- **type**
  Whether we represent an APPOS reference or an IDENT one.

- **section**
  Which section of the coreference document we belong in. See on.corpora.tree.tree.coref_section for more details.

- **document_id**
  The id of the document that we belong to

- **coreference_links**
  A list of coreference_link instances. Better to use [] or iteration on the chain than to use this list directly, though.

- **speaker**
  A string or the empty string. For coref chains that are coreferent with one of the speakers in the document, this will be set to the speaker’s name. To see which speakers are responsible for which sentences, either use the .speaker file or look at the on.corpora.tree.speaker_sentence attribute of trees. During the coreference annotation process the human annotators had access to the name of the speaker for each line.

  Note that the speaker attribute does not represent the person who spoke this sentence.

```python
class on.corpora.coreference.coreference_link(type, coreference_chain, a_cursor=None, start_char_offset=0, end_char_offset=0, precise=False)
```

A coreference annotation

Contained by: coreference_chain

Attributes:

- **string**

- **start_token_index**

- **end_token_index**

- **start_word_index**
end_word_index

sentence_index

start_leaf
   An on.corpora.tree.tree instance. None until enrichment

end_leaf
   An on.corpora.tree.tree instance. None until enrichment

subtree
   An on.corpora.tree.tree instance. None until enrichment. After enrichment, if we could not align this span with any node in the tree, it remains None.

subtree_id
   After enrichment, evaluates to subtree.id. This value is written to the database, and so is available before enrichment when one is loading from the database.

type
   All coreference chains with type IDENT have coreference links with type IDENT. If the coreference chain has type APPOS (appositive) then one coreference link will be the HEAD while the other links will be ATTRIB.

coreference_chain
   What on.corpora.coreference.coreference_chain contains this link.

start_char_offset
   In the case of a token like ‘Japan-China’ we want to be able to tag ‘Japan’ and ‘China’ separately. We do this by specifying a character offset from the beginning and end to describe how much of the token span we care about. So in this case to tag only ‘China’ we would set start_char_offset to 6. To tag only ‘Japan’ we would set end_char_offset to ‘6’. If these offsets are 0, then, we use whole tokens.

   These correspond to the ‘S_OFF’ and ‘E_OFF’ attributes in the coref files.

   For the most complex cases, something like ‘Hong Kong-Zhuhai-Macau’, we specify both the start and the end offsets. The coref structure looks like:

   <COREF>Hong <COREF><COREF>Kong-Zhuhai-Macau</COREF></COREF></COREF>

   And the offsets are E_OFF=13 for Hong Kong, S_OFF=5 and E_OFF=6 for ‘Zhuhai’, and S_OFF=12 for ‘Macau’

end_char_offset
   See coreference_link.start_char_offset

   Before enrichment, generally either the token indices or the word indices will be set but not both. After enrichment, both sets of indices will work and will delegate their responses to start_leaf or end_leaf as appropriate.

5.2.5 name – Name-Entity Annotation

See:
   • on.corpora.name.name_entity
   • on.corpora.name.name_bank

Correspondences:
<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>name_bank</td>
<td>name_bank</td>
<td>All .name files in an on.corpora.subcorpus</td>
</tr>
<tr>
<td>None</td>
<td>name_tagged_document</td>
<td>A .name file</td>
</tr>
<tr>
<td>tree</td>
<td>on.corpora.tree.tree</td>
<td>A line in a .name file</td>
</tr>
<tr>
<td>name_entity</td>
<td>name_entity</td>
<td>A single ENAMEX, TIMEX, or NUMEX span</td>
</tr>
<tr>
<td>None</td>
<td>name_entity_set</td>
<td>All name_entity instances for one on.corpora.tree.tree</td>
</tr>
</tbody>
</table>

### class on.corpora.name.name_bank(a_subcorpus, tag, a_cursor=None, extension='name', indexing='word')

Contains name_tagged_document

### class on.corpora.name.name_tagged_document(document_string, document_id, extension='name', indexing='word', a_cursor=None)

Contained by: name_bank

Contains: name_entity_set

### class on.corpora.name.name_entity(sentence_index, document_id, type, start_index, end_index, string, indexing='word', start_char_offset=0, end_char_offset=0)

A name annotation

Contained by: name_entity_set

Attributes:

- **string**
- **start_token_index**
- **end_token_index**
- **start_word_index**
- **end_word_index**
- **sentence_index**
- **start_leaf**
  - An on.corpora.tree.tree instance. None until enrichment
- **end_leaf**
  - An on.corpora.tree.tree instance. None until enrichment
- **subtree**
  - An on.corpora.tree.tree instance. None until unrichment. After enrichment, if we could not align this span with any node in the tree, it remains None.
- **subtree_id**
  - After enrichment, evaluates to subtree .id. This value is written to the database, and so is available before enrichment when one is loading from the database.
- **type**
  - The type of this named entity, such as PERSON or NORP.

Before enrichment, generally either the token indecies or the word indecies will be set but not both. After enrichment, both sets of indecies will work and will delegate their responses to start_leaf or end_leaf as appropriate.

### class on.corpora.name.name_entity_set(a_document_id)

all the name entities for a single sentence

Contained by: name_tagged_document
5.2.6 ontology – Ontology Annotation

class on.corpora.ontology.ontology(a_id, a_upper_model, a_sense_pool_collection, a_cursor=None)
class on.corpora.ontology.upper_model(a_id, a_um_string, a_cursor=None)
class on.corpora.ontology.sense_pool(a_sense_pool_id, a_sense_pool_string, a_cursor=None)
class on.corpora.ontology.sense_pool_collection(a_id, root_dir, a_cursor=None)
class on.corpora.ontology.concept(a_concept_string, a_cursor=None)
class on.corpora.ontology.feature(a_feature)

exception on.corpora.ontology.no_such_parent_concept_error
exception on.corpora.ontology.no_such_parent_sense_pool_error

5.2.7 speaker – Speaker Metadata for Broadcast Conversation Documents

See:

- speaker_sentence
- speaker_bank

Speaker metadata is additional information collected at the sentence level about speakers before annotation. The data is stored in .speaker files:

```bash
$ head data/english/annotations/bc/cnn/00/cnn_0000.speaker
0.0584225900682 12.399083739 speaker1 male native
0.0584225900682 12.399083739 speaker1 male native
0.0584225900682 12.399083739 speaker1 male native
0.0584225900682 12.399083739 speaker1 male native
0.0584225900682 12.399083739 speaker1 male native
12.3271665044 21.6321665044 paula_zahn female native
12.3271665044 21.6321665044 paula_zahn female native
12.3271665044 21.6321665044 paula_zahn female native
12.3271665044 21.6321665044 paula_zahn female native
12.3271665044 27.7053583252 paula_zahn female native
```

There is one .speaker line for each tree in the document, so above is the speaker metadata for the first 10 trees in cnn_0000. The columns are start_time, stop_time, name, gender, and competency. These values are available in attributes of speaker_sentence with those names.

You might notice that the start and stop times don’t make sense. How can speaker1 say five things where each begins at time 0.05 and ends at time 12.4? When speakers said a group of parsable statements in quick succession, start and stop times were usually only recorded for the group. I’m going to refer to these groups as ‘annotation groups’. An annotation group is roughly analogous to a sentence (by which I mean a single tree); it represents a sequence of words that the annotator doing the transcription grouped together.

Another place this is confusing is with paula_zahn’s final sentence. It has the same start time as her previous four sentences, but a different end time. This is because that tree contains words from two different annotation groups. When this happens, the .speaker line will use the start_time of the initial group and the end_time of the final group. When this happens with the other columns (speakers completing each other’s sentences) we list all values separated by commas, but this is rare. One example would be tree 41 in english bc msnbc document 0006 where George Bush
OntoNotes DB Tool Documentation, Release 0.999b

completes one of Andrea Mitchel’s sentences. With ‘CODE’ statements added to make it clear where the breaks between speakers go, the tree looks like:

\[
(\begin{array}{c}
(NP \ CODE <176.038501264:182.072501264:Andrea_Mitchel:42>)
(NP (NNS Insights) (CC and) (NN analysis))
(PP (IN from)
(NP (NP (NN Bill) (NNP Bennett))
(NP (NP (NN radio) (NN host))
(CC and)
(NP (NP (NN author))
(PP (IN of)
(NP-TTL (NP (NNP America))
(CODE <182.072501264:185.713501264:Andrea_Mitchel:43>)
(NP (NP (NN John) (NNP Harwood))
(PP (IN of)
(NP (NP (DT The)
(NML (NNP Wall) (NNP Street))
(NNP Journal))
(CODE <185.713501264:188.098501264:Andrea_Mitchel:44>)
(NP (NP (NN Dana) (NNP Priest))
(PP (IN of)
(NP (NP (DT The) (NNP Washington) (NNP Post))))
(CODE <188.098501264:190.355501264:George_W_Bush:45>)
(CC And)
(NP (NP (NN William) (NNP Safire))
(PP (IN of)
(NP (DT The)
(NML (NNP New) (NNP York))
(NNP Times))))
\end{array})
\]

This gives a speaker file that looks like:

\$ cat data/english/annotations/bc/msnbc/00/msnbc_0006.speaker
...
160.816917439 163.569917439 Andrea_Mitchel female native
163.569917439 173.243917439 George_W_Bush male native
173.243917439 176.038501264 Andrea_Mitchel female native
176.038501264 190.355501264 Andrea_Mitchel,Andrea_Mitchel,Andrea_Mitchel,George_W_Bush
194.102780118 204.535780118 George_W_Bush male native
204.535780118 212.240780118 George_W_Bush male native
...

Note that the information about when in the statement George Bush took over for Andrea Mitchel is not retained.

This happens 14 times in the english bc data and not at all in the chinese.

Correspondences:

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>speaker_bank</td>
<td>All .speaker files in an on.corpora.subcorpus</td>
</tr>
<tr>
<td>None</td>
<td>speaker_document</td>
<td>A .speaker file</td>
</tr>
<tr>
<td>speaker_sentence</td>
<td>speaker_sentence</td>
<td>A line in a .speaker file</td>
</tr>
</tbody>
</table>
class on.corpora.speaker.speaker_bank(a_subcorpus, tag, a_cursor=None, extension='speaker')
Contains: speaker_document

class on.corpora.speaker.speaker_document(document_id, extension='speaker')
Contained by: speaker_bank
Contains: speaker_document

class on.corpora.speaker.speaker_sentence(line_number, document_id, start_time, stop_time, name, gender, competence)
Contained by: speaker_document

Attributes:

start_time
What time this utterance or series of utterances began. If some speaker says three things in quick
succession, we may have parsed these as three separate trees but timing information could have
only been recorded for the three as a block.

stop_time
What time this utterance or series of utterances ended. The same caveat as with start_time
applies.

name
The name of the speaker. This might be something like ‘speaker_1’ if the data was not entered.

gender
The gender of the speaker. Generally ‘male’ or ‘female’.

competence
The competency of the speaker in the language. Generally ‘native’.

5.2.8 parallel – Alignment Metadata for Parallel Texts

See:
• parallel_sentence
• parallel_document
• parallel_bank

Correspondences:

<table>
<thead>
<tr>
<th>Database Tables</th>
<th>Python Objects</th>
<th>File Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>parallel_bank</td>
<td>All .parallel files in an on.corpora.subcorpus</td>
</tr>
<tr>
<td>parallel_document</td>
<td>parallel_document</td>
<td>The second line (original/translation line) in a .parallel file</td>
</tr>
<tr>
<td>parallel_sentence</td>
<td>parallel_sentence</td>
<td>All lines in a .parallel file after the first two (map lines)</td>
</tr>
</tbody>
</table>

class on.corpora.parallel.parallel_bank(a_subcorpus, tag, a_cursor=None, extension='parallel')
class on.corpora.parallel.parallel_document(id_original, id_translation, extension='parallel')
class on.corpora.parallel.parallel_sentence(id_original, id_translation)
5.3 on.tools – scripts for manipulating the ontonotes data

See:

- on/tools/config.example
- on/tools/create_onfs.py
- on/tools/files_from_db.py
- on/tools/init_db.py
- on/tools/iterate-over-stuff.py

5.3.1 load_to_db – load data from files to database

This code shows how to load ontonotes data to the database. The database needs to have already been initialized and the sense inventories and frames need to have already been loaded. See on.tools.init_db to see how to do that.

5.3.2 init_db – initialize the db and load inventories

This code does two things: initializes the database (creating the tables) and optionally loads the sense inventories and frame files. This needs to happen before data can be loaded to the database with the on.tools.load_to_db command.

For usage information, run this command with no arguments:

$ python init_db.py
APPENDIX A: REPORTING BUGS

The OntoNotes DB Tool remains in beta and has not yet been thoroughly tested for various possible use. We have tried to make sure that for the most standard tasks that it is supposed to be used it works without any hitches – provided that the data is in the right location, dependencies are properly met, etc. With future releases of the GALE OntoNotes corpus, we will be releasing updated versions of this tool. In the meanwhile, any major bug-fix revision would be made available at the OntoNotes webpage hosted at http://www.bbn.com/NLP/OntoNotes.

If you would like to report a bug, submit suggestions, or be notified when an update is available, you should send an email to pradhan@bbn.com

See Also:

How to Report Bugs Effectively  Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines  Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
on.common, 21
on.common.log, 26
on.common.util, 21
on.corpora, 26
on.corpora.coreference, 47
on.corpora.ontology, 51
on.corpora.parallel, 53
on.corpora.proposition, 38
on.corpora.sense, 44
on.corpora.speaker, 51
on.corpora.tree, 30
on.tools, 53
on.tools.init_db, 54
on.tools.load_to_db, 54
Symbols

__getitem__() (on.corpora.proposition.proposition method), 41
__getitem__() (on.corpora.subcorpus method), 28
__getitem__() (on.corpora.tree.tree method), 36

A

a_predicate (on.corpora.proposition.predicate_node attribute), 43
abstract_analogue (class in on.corpora.proposition), 42
abstract_bank (class in on.corpora), 29
abstract_holder (class in on.corpora.proposition), 41
abstract_node (class in on.corpora.proposition), 42
abstract_node_holder (class in on.corpora.proposition), 42
abstract_proposition_bit (class in on.corpora.proposition), 41
all_banks() (on.corpora.subcorpus method), 29
argument (class in on.corpora.proposition), 44
argument_analogue (class in on.corpora.proposition), 43
argument_analogues (on.corpora.proposition.proposition attribute), 40
argument_node (class in on.corpora.proposition), 44
argument_node_list (on.corpora.tree.tree attribute), 32
associated_argument (on.corpora.proposition.proposition attribute), 44

B

backed_by() (on.corpora.subcorpus method), 28
banks (on.corpora.tree.treebank attribute), 30
buckwalter2unicode() (in module on.common.util), 22
bunch (class in on.common.util), 24

C

children (on.corpora.tree.tree attribute), 31
competence (on.corpora.speaker.speaker_sentence attribute), 53
compound_function_tag (class in on.corpora.tree), 38
concept (class in on.corpora.ontology), 51
copy() (on.corpora.subcorpus method), 28
coref_section (on.corpora.tree.tree attribute), 33
coreference_bank (class in on.corpora.coreference), 48
coreference_chain (class in on.corpora.coreference), 48
coreference_chain (on.corpora.coreference.coreference_link attribute), 49
coreference_chain (on.corpora.tree.tree attribute), 32
coreference_document (class in on.corpora.coreference), 48
coreference_link (class in on.corpora.coreference), 48
coreference_link (on.corpora.tree.tree attribute), 32
coreference_links (on.corpora.coreference.coreference_chain attribute), 48

coref_section (on.corpora.tree.tree attribute), 33
copy() (on.corpora.subcorpus method), 28
coreference_bank (class in on.corpora.coreference), 48
coreference_chain (class in on.corpora.coreference), 48
coreference_chain (on.corpora.coreference.coreference_link attribute), 49
coreference_chain (on.corpora.tree.tree attribute), 32
coreference_document (class in on.corpora.coreference), 48
coreference_link (class in on.corpora.coreference), 48
coreference_link (on.corpora.tree.tree attribute), 32
coreference_links (on.corpora.coreference.coreference_chain attribute), 48

debug() (in module on.common.log), 26
document (class in on.corpora), 30
document_bank (class in on.corpora), 30
document_id (on.corpora.coreference.coreference_chain attribute), 48
document_id (on.corpora.proposition.abstract_proposition_bit attribute), 41
document_id (on.corpora.proposition.proposition attribute), 40

document_id (on.corpora.coreference.coreference_chain attribute), 48
document_id (on.corpora.proposition.abstract_proposition_bit attribute), 41
document_id (on.corpora.proposition.proposition attribute), 40

E

e_enc_prop (on.corpora.proposition.proposition attribute), 40
e_enc_self (on.corpora.proposition.abstract_proposition_bit attribute), 41
end_char_offset (on.corpora.coreference.coreference_link attribute), 49
end_coreference_link_list (on.corpora.tree.tree attribute), 32
end_leaf (on.corpora.coreference.coreference_link attribute), 49
end_leaf (on.corpora.name.name_entity attribute), 50
end_named_entity_list (on.corpora.tree.tree attribute), 32
end_token_index (on.corpora.coreference.coreference_link attribute), 48
end_token_index (on.corpora.name.name_entity attribute), 50
end_word_index (on.corpora.coreference.coreference_link attribute), 48
end_word_index (on.corpora.name.name_entity attribute), 50
errcomms (on.corpora.proposition.abstract_node attribute), 43
error() (in module on.common.log), 26
esc() (in module on.common.util), 26

F
FancyConfigParser (class in on.common.util), 26
feature (class in on.corpora.ontology), 51
file (class in on.corpora), 30
fix_trace_index_locations() (on.corpora.tree.tree method), 34
frame_set (class in on.corpora.proposition), 44

G
gender (on.corpora.speaker.speaker_sentence attribute), 53
get_height() (on.corpora.tree.tree method), 36
get_leaf_by_token_index() (on.corpora.tree.tree method), 34
get_leaf_by_word_index() (on.corpora.tree.tree method), 34
get_lemma() (in module on.common.util), 23
get_name() (on.corpora.sense.on_sense_type class method), 47
get_other_leaf() (on.corpora.tree.tree method), 38
get_plain_sentence() (on.corpora.tree.tree method), 34
get_root() (on.corpora.tree.tree method), 34
get_subtree() (on.corpora.tree.tree method), 34
get_subtree_by_span() (on.corpora.tree.tree method), 34
get_trace_adjusted_word_string() (on.corpora.tree.tree method), 34
get_word() (on.corpora.tree.tree method), 34
get_word_index() (on.corpora.tree.tree method), 34
get_word_string() (on.corpora.tree.tree method), 34

H
height (on.corpora.proposition.abstract_node attribute), 42
height (on.corpora.proposition.predicate_node attribute), 43

I
id (on.corpora.proposition.abstract_proposition_bit attribute), 41
identifier (on.corpora.coreference.coreference_chain attribute), 48
identity_subtree (on.corpora.tree.tree attribute), 32
index_in_parent (on.corpora.proposition.abstract_proposition_bit attribute), 41
info() (in module on.common.log), 26
insert_ignoring_dups() (in module on.common.util), 22
is_aux() (on.corpora.tree.tree method), 33
is_conj() (on.corpora.tree.tree method), 33
is_db_ref() (in module on.common.util), 25
is_ich_node (on.corpora.proposition.abstract_node attribute), 43
is_leaf() (on.corpora.tree.tree method), 33
is_not_loaded() (in module on.common.util), 25
is_noun() (on.corpora.tree.tree method), 33
is_root() (on.corpora.tree.tree method), 33
is_trace() (on.corpora.tree.tree method), 33
is_trace_indexed() (on.corpora.tree.tree method), 34
is_trace_origin() (on.corpora.tree.tree method), 34
is_verb() (on.corpora.tree.tree method), 33

L
leaves() (on.corpora.tree.tree method), 36
lemma (class in on.corpora.tree), 38
lemma (on.corpora.proposition.abstract_proposition_bit attribute), 41
lemma (on.corpora.proposition.proposition attribute), 40
lemma (on.corpora.sense.on_sense attribute), 46
lemma (on.corpora.sense.on_sense_type attribute), 46
lemma (on.corpora.tree.tree attribute), 31
lemma_object (on.corpora.tree.tree attribute), 31
link (class in on.corpora.proposition), 44
link_analogue (class in on.corpora.proposition), 44
link_analogue (on.corpora.proposition.proposition attribute), 40
link_node (class in on.corpora.proposition), 44
listdir() (in module on.common.util), 24
listdir_both() (in module on.common.util), 24
listdir_full() (in module on.common.util), 24
load_banks() (on.corpora.subcorpus method), 28
load_options() (in module on.common.util), 23

M
make_db_ref() (in module on.common.util), 25
make_not_loaded() (in module on.common.util), 26
make_sgml_safe() (in module on.common.util), 26
make_sgml_unsafe() (in module on.common.util), 26
matches_an_affix() (in module on.common.util), 23
mdirs() (in module on.common.util), 23

N
name (on.corpora.speaker.speaker_sentence attribute), 53
name_bank (class in on.corpora.name), 50
name_entity (class in on.corpora.name), 50
name_entity_set (class in on.corpora.name), 50
name_tagged_document (class in on.corpora.name), 50
named_entity (on.corpora.tree.tree attribute), 32
no_such_parent_concept_error, 51
no_such_parent_sense_pool_error, 51
NotInConfigError, 24

O
on.common (module), 21
on.common.log (module), 26
on.common.util (module), 21
on.corpora (module), 26
on.corpora.coreference (module), 47
on.corpora.name (module), 49
on.corpora.ontology (module), 51
on.corpora.parallel (module), 53
on.corpora.proposition (module), 38
on.corpora.sense (module), 44
on.corpora.speaker (module), 51
on.corpora.tree (module), 30
on.tools (module), 53
on.tools.init_db (module), 54
on.tools.load_to_db (module), 54
on_sense (class in on.corpora.sense), 46
on_sense (on.corpora.tree.tree attribute), 32
on_sense_lemma_type (class in on.corpora.sense), 47
on_sense_type (class in on.corpora.sense), 46
ontology (class in on.corpora.ontology), 51
original (on.corpora.tree.tree_document attribute), 31
originals (on.corpora.tree.tree attribute), 33
output_file_name() (in module on.common.util), 23

P
parallel_bank (class in on.corpora.parallel), 53
parallel_document (class in on.corpora.parallel), 53
parallel_sentence (class in on.corpora.parallel), 53
parent (on.corpora.proposition.abstract_node attribute), 42
parent (on.corpora.tree.tree attribute), 31
parse_cfg_args() (in module on.common.util), 24
pb_sense_num (on.corpora.proposition.abstract_proposition bit attribute), 41
pb_sense_num (on.corpora.proposition.proposition attribute), 40
pb_sense_type (class in on.corpora.sense), 47
pb_sense_types (on.corpora.sense.on_sense_type attribute), 47
pointer() (on.corpora.tree.tree method), 34
pos (on.corpora.sense.on_sense attribute), 46
pos (on.corpora.sense.on_sense_type attribute), 46
predicate (class in on.corpora.proposition), 43
predicate (on.corpora.proposition.proposition attribute), 40
predicate_analogue (class in on.corpora.proposition), 43
predicate_node (class in on.corpora.proposition), 43
predicate_node_list (on.corpora.tree.tree attribute), 32
pretty_print() (on.corpora.tree.tree method), 34
primary (on.corpora.proposition.predicate_node attribute), 43
propo

Index 61
start_leaf (on.corpora.name.name_entity attribute), 50
start_named_entity_list (on.corpora.tree.tree attribute), 32
start_time (on.corpora.speaker.speaker_sentence attribute), 53
start_token_index (on.corpora.coreference.coreference_link attribute), 48
start_token_index (on.corpora.name.name_entity attribute), 50
start_word_index (on.corpora.coreference.coreference_link attribute), 48
start_word_index (on.corpora.name.name_entity attribute), 50
status() (in module on.common.log), 26
stop_time (on.corpora.speaker.speaker_sentence attribute), 53
string (on.corpora.coreference.coreference_link attribute), 48
string (on.corpora.name.name_entity attribute), 50
subcorpus (class in on.corpora), 27
subtree (on.corpora.coreference.coreference_link attribute), 49
subtree (on.corpora.name.name_entity attribute), 50
subtree (on.corpora.proposition.abstract_node attribute), 43
subtree_id (on.corpora.coreference.coreference_link attribute), 49
subtree_id (on.corpora.name.name_entity attribute), 50
subtrees() (on.corpora.tree.tree method), 36
syntactic_link (class in on.corpora.tree), 38

T
tag (on.corpora.tree.tree attribute), 31
token (class in on.corpora), 30
token_index (on.corpora.proposition.abstract_node attribute), 42
token_index (on.corpora.proposition.predicate_node attribute), 43
translations (on.corpora.tree.tree attribute), 33
translations (on.corpora.tree.tree_document attribute), 30
tree (class in on.corpora.tree), 31
tree_document (class in on.corpora.tree), 30
tree_exception, 38
treebank (class in on.corpora.tree), 30
type (on.corpora.coreference.coreference_chain attribute), 48
type (on.corpora.coreference.coreference_link attribute), 49
type (on.corpora.name.name_entity attribute), 50
type (on.corpora.proposition.proposition attribute), 40

U
unicode2buckwalter() (in module on.common.util), 22
upper_model (class in on.corpora.ontology), 51