wn

Release 1.1.0

Michael Wayne Goodman

Mar 21, 2026

CONTENTS

CHAPTER

ONE

OVERVIEW

This package provides an interface to wordnet data, from simple lookup queries, to graph traversals, to more sophisticated algorithms and metrics. Features include:

- Support for wordnets in the WN-LMF format
- A SQLite database backend for data consistency and efficient queries
- Accurate modeling of Words, Senses, and Synsets

CHAPTER

TWO

QUICK START

```
$ pip install wn
```

```
>>> import wn
>>> wn.download('ewn:2020')
>>> wn.synsets('coffee')
[Synset('ewn-04979718-n'), Synset('ewn-07945591-n'), Synset('ewn-07945759-n'), Synset(
↪'ewn-12683533-n')]
```

CHAPTER

THREE

CONTENTS

3.1 Installation and Configuration

See also

This guide is for installing and configuring the Wn software. For adding lexicons to the database, see *Working with Lexicons*.

3.1.1 Installing from PyPI

Install the latest release from PyPI:

```
pip install wn
```

3.1.2 The Data Directory

By default, Wn stores its data (such as downloaded LMF files and the database file) in a `.wn_data/` directory under the user's home directory. This directory can be changed (see *Configuration* below). Whenever Wn attempts to download a resource or access its database, it will check for the existence of, and create if necessary, this directory, the `.wn_data/downloads/` subdirectory, and the `.wn_data/wn.db` database file. The file system will look like this:

```
.wn_data/
├── downloads
│   ├── ...
│   └── ...
└── wn.db
```

The `...` entries in the `downloads/` subdirectory represent the files of resources downloaded from the web. Their filename is a hash of the URL so that Wn can avoid downloading the same file twice.

3.1.3 Configuration

The *wn.config* object contains the paths Wn uses for local storage and information about resources available on the web. To change the directory Wn uses for storing data locally, modify the `wn.config.data_directory` member:

```
import wn
wn.config.data_directory = '~/Projects/wn_data'
```

You can alternatively set the `WN_DATA_DIR` environment variable prior to importing Wn. On Unix-like systems, this would look like:

```
$ export WN_DATA_DIR=~/path/to/wn_data
$ python3 ...
```

If you are using Wn from the command line, a third option is to use the `--dir` or `-d` option:

```
$ python3 -m wn --dir ~/path/to/wn_data download ...
```

There are some things to note:

- The downloads directory and database path are always relative to the data directory and cannot be changed directly.
- This change only affects subsequent operations, so any data in the previous location will not be moved or deleted.
- This change only affects the current session. If you want a script or application to always use the new location, it must reset the data directory each time it is initialized.

You can also add project information for remote resources. First you add a project, with a project ID, full name, and language code. Then you create one or more versions for that project with a version ID, resource URL, and license information. This may be done either through the *wn.config* object's *add_project()* and *add_project_version()* methods, or loaded from a TOML file via the *wn.config* object's *load_index()* method.

```
wn.config.add_project('ewn', 'English WordNet', 'en')
wn.config.add_project_version(
    'ewn', '2020',
    'https://en-word.net/static/english-wordnet-2020.xml.gz',
    'https://creativecommons.org/licenses/by/4.0/',
)
```

3.1.4 Rebuilding the Database

New versions of Wn may occasionally alter the database schema in a way that makes an existing database incompatible with the code. You will see an error like this (abbreviated):

```
>>> import wn
>>> wn.Wordnet("oewn:2024")
Traceback (most recent call last):
  [...]
wn.DatabaseError: Wn's schema has changed and is no longer compatible with the database.
Lexicons currently installed:
  odenet:1.4
  oewn:2023
  oewn:2024
  omw-arb:1.4
  [...]]
Run wn.reset_database(rebuild=True) to rebuild the database.
```

You can then run, as directed, *wn.reset_database()* with `rebuild=True`, which will delete the database, initialize a new one, and attempt to add all the lexicons that were previously added. You can also run with `rebuild=False` to reinitialize the database without re-adding lexicons, or alternatively simply delete the database file from your filesystem. See the documentation for *wn.reset_database()* for more information.

3.1.5 Installing From Source

If you wish to install the code from the source repository (e.g., to get an unreleased feature or to contribute toward Wn's development), clone the repository and use Hatch to start a virtual environment with Wn installed:

```
$ git clone https://github.com/goodmami/wn.git
$ cd wn
$ hatch shell
```

3.2 Command Line Interface

Some of Wn's functionality is exposed via the command line.

3.2.1 Global Options

-d DIR, **--dir** DIR

Change to use DIR as the data directory prior to invoking any commands.

3.2.2 Subcommands

3.2.3 download

Download and add projects to the database given one or more project specifiers or URLs.

```
$ python -m wn download oewn:2021 omw:1.4 cili
$ python -m wn download https://en-word.net/static/english-wordnet-2021.xml.gz
```

--index FILE

Use the index at FILE to resolve project specifiers.

```
$ python -m wn download --index my-index.toml mywn
```

--no-add

Download and cache the remote file, but don't add it to the database.

3.2.4 cache

View the files in the download cache. The **download** command caches the (often compressed) files to the filesystem prior to adding to Wn's database. The files are renamed with a hash of the URL to avoid name clashes, but this also makes it hard to determine what a particular file is. This command cross-references the downloaded files with what is in the index. An optional project specifier argument can help narrow down the results.

```
$ python -m wn cache  # many results; abbreviated here
af909070c29845b952d1799551bffc302e28d2c5        own-en  1.0.0   https://github.com/own-
↪pt/openWordnet-PT/releases/download/v1.0.0/own-en.tar.gz
e25af66e46775b00d689619787013e6a35e5cbf7        oewn    2025    https://en-word.net/
↪static/english-wordnet-2025.xml.gz
5a26d97a0081996db4cd621638a8a9b0da09aa25        odenet  1.4     https://github.com/
↪hdaSprachtechnologie/odenet/releases/download/v1.4/odenet-1.4.tar.xz
[...]
$ python -m wn cache "oewn:2025*" # narrowed results
e25af66e46775b00d689619787013e6a35e5cbf7        oewn    2025    https://en-word.net/
↪static/english-wordnet-2025.xml.gz
```

(continues on next page)

(continued from previous page)

```
0f5371187dcfe7e05f2a93ab85b4e1168859a5c2        oewn    2025+   https://en-word.net/
↪static/english-wordnet-2025-plus.xml.gz
```

--full-paths-only

Only print the full path of each cache file. This can be useful when one wants to pipe the results to other commands. For example, on Unix-like systems, the following will delete matching cache entries:

```
$ python -m wn cache --full-paths-only "omw*:1.4" | xargs rm
```

3.2.5 lexicons

The `lexicons` subcommand lets you quickly see what is installed:

```
$ python -m wn lexicons
omw-en       1.4     [en]    OMW English Wordnet based on WordNet 3.0
omw-sk       1.4     [sk]    Slovak WordNet
omw-pl       1.4     [pl]    plWordNet
omw-is       1.4     [is]    IceWordNet
omw-zsm      1.4     [zsm]   Wordnet Bahasa (Malaysian)
omw-sl       1.4     [sl]    sloWNet
omw-ja       1.4     [ja]    Japanese Wordnet
...
```

-l LG, **--lang** LG

--lexicon SPEC

The `--lang` or `--lexicon` option can help you narrow down the results:

```
$ python -m wn lexicons --lang en
oewn      2021    [en]    Open English WordNet
omw-en    1.4     [en]    OMW English Wordnet based on WordNet 3.0
$ python -m wn lexicons --lexicon "omw-*"
omw-en    1.4     [en]    OMW English Wordnet based on WordNet 3.0
omw-sk    1.4     [sk]    Slovak WordNet
omw-pl    1.4     [pl]    plWordNet
omw-is    1.4     [is]    IceWordNet
omw-zsm   1.4     [zsm]   Wordnet Bahasa (Malaysian)
```

3.2.6 projects

The `projects` subcommand lists all known projects in Wn's index. This is helpful to see what is available for downloading.

```
$ python -m wn projects
ic      cili    1.0     [---]   Collaborative Interlingual Index
ic      oewn    2025+   [en]    Open English WordNet
ic      oewn    2025    [en]    Open English WordNet
ic      oewn    2024    [en]    Open English WordNet
ic      oewn    2023    [en]    Open English WordNet
ic      oewn    2022    [en]    Open English WordNet
ic      oewn    2021    [en]    Open English WordNet
ic      ewn     2020    [en]    Open English WordNet
```

(continues on next page)

(continued from previous page)

```
ic      ewn     2019    [en]    Open English WordNet
ic      odenet  1.4     [de]    Open German WordNet
i-      odenet  1.3     [de]    Open German WordNet
ic      omw     2.0     [mul]   Open Multilingual Wordnet
ic      omw     1.4     [mul]   Open Multilingual Wordnet
...
```

3.2.7 validate

Given a path to a WN-LMF XML file, check the file for structural problems and print a report.

```
$ python -m wn validate english-wordnet-2021.xml
```

--select CHECKS

Run the checks with the given comma-separated list of check codes or categories.

```
$ python -m wn validate --select E,W201,W204 deWordNet.xml
```

--output-file FILE

Write the report to FILE as a JSON object instead of printing the report to stdout.

3.3 FAQ

3.3.1 Is Wn related to the NLTK's *nltk.corpus.wordnet* module?

Only in spirit. There was an effort to develop the NLTK's module as a standalone package (see https://github.com/nltk/wordnet/), but development had slowed. Wn has the same broad goals and a similar API as that standalone package, but fundamental architectural differences demanded a complete rewrite, so Wn was created as a separate project. With approval from the other package's maintainer, Wn acquired the wn project on PyPI and can be seen as its successor.

3.3.2 Is Wn compatible with the NLTK's module?

The API is intentionally similar, but not exactly the same (for instance see the next question), and there are differences in the ways that results are retrieved, particularly for non-English wordnets. See *Migrating from the NLTK* for more information. Also see *Where is the Princeton WordNet data?*.

3.3.3 Where are the `Lemma` objects? What are `Word` and `Sense` objects?

Unlike the original WNDB data format of the original WordNet, the WN-LMF XML format grants words (called *lexical entries* in WN-LMF and a *`Word`* object in Wn) and word senses (*`Sense`* in Wn) explicit, first-class status alongside synsets. While senses are essentially links between words and synsets, they may contain metadata and be the source or target of sense relations, so in some ways they are more like nodes than edges when the wordnet is viewed as a graph. The NLTK's module, using the WNDB format, combines the information of a word and a sense into a single object called a `Lemmas`. Wn also has an unrelated concept called a *`lemma()`*, but it is merely the canonical form of a word.

3.3.4 Where is the Princeton WordNet data?

The original English wordnet, named simply *WordNet* but often referred to as the *Princeton WordNet* to better distinguish it from other projects, is specifically the data distributed by Princeton in the WNDB format. The Open Multilingual Wordnet (OMW) packages an export of the WordNet data as the *OMW English Wordnet based on WordNet 3.0* which is used by Wn (with the lexicon ID `omw-en`). It also has a similar export for WordNets 1.5, 1.6, 1.7, 1.7.1, 2.0,

2.1, and 3.1 data (`omw-en15`, `omw-en16`, `omw-en17`, `omw-en171`, `omw-en20`, `omw-en21`, and `omw-en31`, respectively). All of these are highly compatible with the original data and can be used as drop-in replacements.

Prior to Wn version 0.9 (and, correspondingly, prior to the OMW data version 1.4), the `pwn:3.0` and `pwn:3.1` English wordnets distributed by OMW were incorrectly called the *Princeton WordNet* (for WordNet 3.0 and 3.1, respectively). From Wn version 0.9 (and from version 1.4 of the OMW data), these are called the *OMW English Wordnet based on WordNet 3.0/3.1* (`omw-en:1.4` and `omw-en31:1.4`, respectively). These lexicons are intentionally compatible with the original WordNet data, and the 1.4 versions are even more compatible than the previous `pwn:3.0` and `pwn:3.1` lexicons, so it is strongly recommended to use them over the previous versions. Similarly, the 2.0 version of OMW is more compatible yet. The data corresponding to WordNet versions 1.5 through 2.1 are only available from OMW 2.0.

3.3.5 Why don't all wordnets share the same synsets?

The Open Multilingual Wordnet (OMW) contains wordnets for many languages created using the *expand* methodology [VOSSEN1998], where non-English wordnets provide words on top of the English wordnet's synset structure. This allows new wordnets to be built in much less time than starting from scratch, but with a few drawbacks, such as that words cannot be added if they do not have a synset in the English wordnet, and that it is difficult to version the wordnets independently (e.g., for reproducibility of experiments involving wordnet data) as all are interconnected. Wn, therefore, creates new synsets for each wordnet added to its database, and synsets then specify which resource they belong to. Queries can specify which resources may be examined. Also see *Interlingual Queries*.

3.3.6 Why does Wn's database get so big?

The *OMW English Wordnet based on WordNet 3.0* takes about 114 MiB of disk space in Wn's database, which is only about 8 MiB more than it takes as a WN-LMF XML file. The NLTK, however, uses the obsolete WNDB format which is more compact, requiring only 35 MiB of disk space. The difference with the Open Multilingual Wordnet 1.4 is more striking: it takes about 659 MiB of disk space in the database, but only 49 MiB in the NLTK. Part of the difference here is that the OMW files in the NLTK are simple tab-separated-value files listing only the words added to each synset for each language. In addition, Wn creates new synsets for each wordnet added (see the previous question). One more reason is that Wn creates various indexes in the database for efficient lookup.

3.4 Working with Lexicons

3.4.1 Terminology

In Wn, the following terminology is used:

lexicon
: An inventory of words, senses, synsets, relations, etc. that share a namespace (i.e., that can refer to each other).

wordnet
: A group of lexicons (but usually just one).

resource
: A file containing lexicons.

package
: A directory containing a resource and optionally some metadata files.

collection
: A directory containing packages and optionally some metadata files.

project
: A general term for a resource, package, or collection, particularly pertaining to its creation, maintenance, and distribution.

In general, each resource contains one lexicon. For large projects like the Open English WordNet, that lexicon is also a wordnet on its own. For a collection like the Open Multilingual Wordnet, most lexicons do not include relations as they are instead expected to use those from the OMW's included English wordnet, which is derived from the Princeton WordNet. As such, a wordnet for these sub-projects is best thought of as the grouping of the lexicon with the lexicon providing the relations.

3.4.2 Lexicon and Project Specifiers

Wn uses *lexicon specifiers* to deal with the possibility of having multiple lexicons and multiple versions of lexicons loaded in the same database. The specifiers are the joining of a lexicon's name (ID) and version, delimited by `:`. Here are the possible forms:

```
*           -- any/all lexicons
id          -- the most recently added lexicon with the given id
id:*        -- all lexicons with the given id
id:version  -- the lexicon with the given id and version
*:version   -- all lexicons with the given version
```

For example, if `ewn:2020` was installed followed by `ewn:2019`, then `ewn` would specify the `2019` version, `ewn:*` would specify both versions, and `ewn:2020` would specify the `2020` version.

The same format is used for *project specifiers*, which refer to projects as defined in Wn's index. In most cases the project specifier is the same as the lexicon specifier (e.g., `ewn:2020` refers both to the project to be downloaded and the lexicon that is installed), but sometimes it is not. The 1.4 release of the Open Multilingual Wordnet, for instance, has the project specifier `omw:1.4` but it installs a number of lexicons with their own lexicon specifiers (`omw-zsm:1.4`, `omw-cmn:1.4`, etc.). When only an id is given (e.g., `ewn`), a project specifier gets the *first* version listed in the index (in the default index, conventionally, the first version is the latest release).

3.4.3 Filtering Queries with Lexicons

Queries against the database will search all installed lexicons unless they are filtered by `lang` or `lexicon` arguments:

```
>>> import wn
>>> len(wn.words())
1538449
>>> len(wn.words(lang="en"))
318289
>>> len(wn.words(lexicon="oewn:2024"))
161705
```

The `lexicon` parameter can also take multiple specifiers so you can include things like lexicon extensions or to explicitly include multiple lexicons:

```
>>> len(wn.words(lexicon="oewn:2024 omw-en:1.4"))
318289
```

If a lexicon selected by the `lexicon` or `lang` arguments specifies a dependency, the dependency is automatically added as an *expand* lexicon. Explicitly set `expand=''` to disable this behavior:

```
>>> wn.lexicons(lexicon="omw-es:1.4")[0].requires()  # omw-es requires omw-en
{'omw-en:1.4': <Lexicon omw-en:1.4 [en]>}
>>> es = wn.Wordnet("omw-es:1.4")
>>> es.lexicons()
[<Lexicon omw-es:1.4 [es]>]
>>> es.expanded_lexicons()  # omw-en automatically added
```

(continues on next page)

(continued from previous page)

```
[<Lexicon omw-en:1.4 [en]>]
>>> es_no_en = wn.Wordnet("omw-es:1.4", expand='')
>>> es_no_en.lexicons()
[<Lexicon omw-es:1.4 [es]>]
>>> es_no_en.expanded_lexicons()  # no expand lexicons
[]
```

Also see *Cross-lingual Relation Traversal* for selecting expand lexicons for relations.

The objects returned by queries retain the "lexicon configuration" used, which includes the lexicons and expand lexicons. This configuration determines which lexicons are searched during secondary queries. The lexicon configuration also stores a flag indicating whether no lexicon filters were used at all, which triggers *default mode* secondary queries.

3.4.4 Default Mode Queries

A special "default mode" is activated when making a module-function query (*wn.words()*, *wn.synsets()*, etc.) or instantiating a *wn.Wordnet* object with no `lexicon` or `lang` argument (so-named because the mode is triggered by using the default values of `lexicon` and `lang`):

```
>>> w = wn.Wordnet()
>>> wn.words("pineapple")  # for example
```

Default-mode causes the following behavior:

1. Primary queries search any installed lexicon
2. Secondary queries only search the lexicon of the primary entity (e.g., `Synset.words()` only finds words from the same lexicon as the synset). If the lexicon has any extensions or is itself an extension, any extension/base lexicons are also included.
3. If the `expand` argument is **None** (always true for module functions like *wn.synsets()*), all installed lexicons are used as expand lexicons for relations queries.

Warning

Default-mode queries are not reproducible as the results can change as lexicons are added or removed from the database. For anything more than a casual query, it is highly suggested to instead create a *wn.Wordnet* object with fully-specified `lexicon` and `expand` arguments.

3.4.5 Downloading Lexicons

Use *wn.download()* to download lexicons from the web given either an indexed project specifier or the URL of a resource, package, or collection.

```
>>> import wn
>>> wn.download('odenet')  # get the latest Open German WordNet
>>> wn.download('odenet:1.3')  # get the 1.3 version
>>> # download from a URL
>>> wn.download('https://github.com/omwn/omw-data/releases/download/v1.4/omw-1.4.tar.xz')
```

The project specifier is only used to retrieve information from Wn's index. The lexicon IDs of the corresponding resource files are what is stored in the database.

3.4.6 Adding Local Lexicons

Lexicons can be added from local files with *wn.add()*:

```
>>> wn.add('~/data/omw-1.4/omw-nb/omw-nb.xml')
```

Or with the parent directory as a package:

```
>>> wn.add('~/data/omw-1.4/omw-nb/')
```

Or with the grandparent directory as a collection (installing all packages contained by the collection):

```
>>> wn.add('~/data/omw-1.4/')
```

Or from a compressed archive of one of the above:

```
>>> wn.add('~/data/omw-1.4/omw-nb/omw-nb.xml.xz')
>>> wn.add('~/data/omw-1.4/omw-nb.tar.xz')
>>> wn.add('~/data/omw-1.4.tar.xz')
```

3.4.7 Listing Installed Lexicons

If you wish to see which lexicons have been added to the database, *wn.lexicons()* returns the list of *wn.Lexicon* objects that describe each one.

```
>>> for lex in wn.lexicons():
...     print(f'{lex.id}:{lex.version}\t{lex.label}')
...
omw-en:1.4      OMW English Wordnet based on WordNet 3.0
omw-nb:1.4      Norwegian Wordnet (Bokmål)
odenet:1.3      Offenes Deutsches WordNet
ewn:2020        English WordNet
ewn:2019        English WordNet
```

3.4.8 Removing Lexicons

Lexicons can be removed from the database with *wn.remove()*:

```
>>> wn.remove('omw-nb:1.4')
```

Note that this removes a single lexicon and not a project, so if, for instance, you've installed a multi-lexicon project like `omw`, you will need to remove each lexicon individually or use a star specifier:

```
>>> wn.remove('omw-*:1.4')
```

3.4.9 WN-LMF Files, Packages, and Collections

Wn can handle projects with 3 levels of structure:

- WN-LMF XML files
- WN-LMF packages
- WN-LMF collections

WN-LMF XML Files

A WN-LMF XML file is a file with a `.xml` extension that is valid according to the WN-LMF specification.

WN-LMF Packages

If one needs to distribute metadata or additional files along with WN-LMF XML file, a WN-LMF package allows them to include the files in a directory. The directory should contain exactly one `.xml` file, which is the WN-LMF XML file. In addition, it may contain additional files and Wn will recognize three of them:

LICENSE (.txt | .md | .rst)
: the full text of the license

README (.txt | .md | .rst)
: the project README

citation.bib
: a BibTeX file containing academic citations for the project

```
omw-sq/
├── omw-sq.xml
├── LICENSE.txt
└── README.md
```

WN-LMF Collections

In some cases a project may manage multiple resources and distribute them as a collection. A collection is a directory containing subdirectories which are WN-LMF packages. The collection may contain its own README, LICENSE, and citation files which describe the project as a whole.

```
omw-1.4/
├── omw-sq
│   ├── oms-sq.xml
│   ├── LICENSE.txt
│   └── README.md
├── omw-lt
│   ├── citation.bib
│   ├── LICENSE
│   └── omw-lt.xml
├── ...
├── citation.bib
├── LICENSE
└── README.md
```

3.5 Basic Usage

See also

This document covers the basics of querying wordnets, filtering results, and performing secondary queries on the results. For adding, removing, or inspecting lexicons, see *Working with Lexicons*. For more information about interlingual queries, see *Interlingual Queries*.

For the most basic queries, Wn provides several module functions for retrieving words, senses, and synsets:

```
>>> import wn
>>> wn.words('pike')
[Word('ewn-pike-n')]
>>> wn.senses('pike')
[Sense('ewn-pike-n-03311555-04'), Sense('ewn-pike-n-07795351-01'), Sense('ewn-pike-n-
↪03941974-01'), Sense('ewn-pike-n-03941726-01'), Sense('ewn-pike-n-02563739-01')]
>>> wn.synsets('pike')
[Synset('ewn-03311555-n'), Synset('ewn-07795351-n'), Synset('ewn-03941974-n'), Synset(
↪'ewn-03941726-n'), Synset('ewn-02563739-n')]
```

Once you start working with multiple wordnets, these simple queries may return more than desired:

```
>>> wn.words('pike')
[Word('ewn-pike-n'), Word('wnja-n-66614')]
>>> wn.words('chat')
[Word('ewn-chat-n'), Word('ewn-chat-v'), Word('frawn-lex14803'), Word('frawn-lex21897')]
```

You can specify which language or lexicon you wish to query:

```
>>> wn.words('pike', lang='ja')
[Word('wnja-n-66614')]
>>> wn.words('chat', lexicon='frawn')
[Word('frawn-lex14803'), Word('frawn-lex21897')]
```

But it might be easier to create a *Wordnet* object and use it for queries:

```
>>> wnja = wn.Wordnet(lang='ja')
>>> wnja.words('pike')
[Word('wnja-n-66614')]
>>> frawn = wn.Wordnet(lexicon='frawn')
>>> frawn.words('chat')
[Word('frawn-lex14803'), Word('frawn-lex21897')]
```

In fact, the simple queries above implicitly create such a *Wordnet* object, but one that includes all installed lexicons.

3.5.1 Primary Queries

The queries shown above are "primary" queries, meaning they are the first step in a user's interaction with a wordnet. Operations performed on the resulting objects are then *secondary queries*. Primary queries optionally take several fields for filtering the results, namely the word form and part of speech. Synsets may also be filtered by an interlingual index (ILI).

Searching for Words

The *wn.words()* function returns a list of *Word* objects that match the given word form or part of speech:

```
>>> wn.words('pencil')
[Word('ewn-pencil-n'), Word('ewn-pencil-v')]
>>> wn.words('pencil', pos='v')
[Word('ewn-pencil-v')]
```

Calling the function without a word form will return all words in the database:

```
>>> len(wn.words())
311711
>>> len(wn.words(pos='v'))
29419
>>> len(wn.words(pos='v', lexicon='ewn'))
11595
```

If you know the word identifier used by a lexicon, you can retrieve the word directly with the *wn.word()* function. Identifiers are guaranteed to be unique within a single lexicon, but not across lexicons, so it's best to call this function from an instantiated *Wordnet* object or with the `lexicon` parameter specified. If multiple words are found when querying multiple lexicons, only the first is returned.

```
>>> wn.word('ewn-pencil-n', lexicon='ewn')
Word('ewn-pencil-n')
```

Searching for Senses

The *wn.senses()* and *wn.sense()* functions behave similarly to *wn.words()* and *wn.word()*, except that they return matching *Sense* objects.

```
>>> wn.senses('plow', pos='n')
[Sense('ewn-plow-n-03973894-01')]
>>> wn.sense('ewn-plow-v-01745745-01')
Sense('ewn-plow-v-01745745-01')
```

Senses represent a relationship between a *Word* and a *Synset*. Seen as an edge between nodes, senses are often given less prominence than words or synsets, but they are the natural locus of several interesting features such as sense relations (e.g., for derived words) and the natural level of representation for translations to other languages.

Searching for Synsets

The *wn.synsets()* and *wn.synset()* functions are like those above but allow the `ili` parameter for filtering by interlingual index, which is useful in interlingual queries:

```
>>> wn.synsets('scepter')
[Synset('ewn-14467142-n'), Synset('ewn-07282278-n')]
>>> wn.synset('ewn-07282278-n').ili
'i74874'
>>> wn.synsets(ili='i74874')
[Synset('ewn-07282278-n'), Synset('wnja-07267573-n'), Synset('frawn-07267573-n')]
```

3.5.2 Secondary Queries

Once you have gotten some results from a primary query, you can perform operations on the *Word*, *Sense*, or *Synset* objects to get at further information in the wordnet.

Exploring Words

Here are some of the things you can do with *Word* objects:

```
>>> w = wn.words('goose')[0]
>>> w.pos  # part of speech
'n'
```

(continues on next page)

(continued from previous page)

```
>>> w.forms()  # other word forms (e.g., irregular inflections)
['goose', 'geese']
>>> w.lemma()  # canonical form
'goose'
>>> w.derived_words()
[Word('ewn-gosling-n'), Word('ewn-goosy-s'), Word('ewn-goosey-s')]
>>> w.senses()
[Sense('ewn-goose-n-01858313-01'), Sense('ewn-goose-n-10177319-06'), Sense('ewn-goose-n-
↪07662430-01')]
>>> w.synsets()
[Synset('ewn-01858313-n'), Synset('ewn-10177319-n'), Synset('ewn-07662430-n')]
```

Since translations of a word into another language depend on the sense used, *Word.translate* returns a dictionary mapping each sense to words in the target language:

```
>>> for sense, ja_words in w.translate(lang='ja').items():
...     print(sense, ja_words)
...
Sense('ewn-goose-n-01858313-01') [Word('wnja-n-1254'), Word('wnja-n-33090'), Word('wnja-
↪n-38995')]
Sense('ewn-goose-n-10177319-06') []
Sense('ewn-goose-n-07662430-01') [Word('wnja-n-1254')]
```

Exploring Senses

Compared to *Word* and *Synset* objects, there are relatively few operations available on *Sense* objects. Sense relations and translations, however, are important operations on senses.

```
>>> s = wn.senses('dark', pos='n')[0]
>>> s.word()    # each sense links to a single word
Word('ewn-dark-n')
>>> s.synset()  # each sense links to a single synset
Synset('ewn-14007000-n')
>>> s.get_related('antonym')
[Sense('ewn-light-n-14006789-01')]
>>> s.get_related('derivation')
[Sense('ewn-dark-a-00273948-01')]
>>> s.translate(lang='fr')  # translation returns a list of senses
[Sense('frawn-lex52992--13983515-n')]
>>> s.translate(lang='fr')[0].word().lemma()
'obscurité'
```

Exploring Synsets

Many of the operations people care about happen on synsets, such as hierarchical relations and metrics.

```
>>> ss = wn.synsets('hound', pos='n')[0]
>>> ss.senses()
[Sense('ewn-hound-n-02090203-01'), Sense('ewn-hound_dog-n-02090203-02')]
>>> ss.words()
[Word('ewn-hound-n'), Word('ewn-hound_dog-n')]
>>> ss.lemmas()
```

(continues on next page)

(continued from previous page)

```
['hound', 'hound dog']
>>> ss.definition()
'any of several breeds of dog used for hunting typically having large drooping ears'
>>> ss.hypernyms()
[Synset('ewn-02089774-n')]
>>> ss.hypernyms()[0].lemmas()
['hunting dog']
>>> len(ss.hyponyms())
20
>>> ss.hyponyms()[0].lemmas()
['Afghan', 'Afghan hound']
>>> ss.max_depth()
15
>>> ss.shortest_path(wn.synsets('dog', pos='n')[0])
[Synset('ewn-02090203-n'), Synset('ewn-02089774-n'), Synset('ewn-02086723-n')]
>>> ss.translate(lang='fr')  # translation returns a list of synsets
[Synset('frawn-02087551-n')]
>>> ss.translate(lang='fr')[0].lemmas()
['chien', 'chien de chasse']
```

3.5.3 Filtering by Language

The `lang` parameter of *wn.words()*, *wn.senses()*, *wn.synsets()*, and *Wordnet* allows a single BCP 47 language code. When this parameter is used, only entries in the specified language will be returned.

```
>>> import wn
>>> wn.words('chat')
[Word('ewn-chat-n'), Word('ewn-chat-v'), Word('frawn-lex14803'), Word('frawn-lex21897')]
>>> wn.words('chat', lang='fr')
[Word('frawn-lex14803'), Word('frawn-lex21897')]
```

If a language code not used by any lexicon is specified, a *wn.Error* is raised.

3.5.4 Filtering by Lexicon

The `lexicon` parameter of *wn.words()*, *wn.senses()*, *wn.synsets()*, and *Wordnet* take a string of space-delimited *lexicon specifiers*. Entries in a lexicon whose ID matches one of the lexicon specifiers will be returned. For these, the following rules are used:

- A full `id:version` string (e.g., `ewn:2020`) selects a specific lexicon
- Only a lexicon `id` (e.g., `ewn`) selects the most recently added lexicon with that ID
- A star * may be used to match any lexicon; a star may not include a version

```
>>> wn.words('chat', lexicon='ewn:2020')
[Word('ewn-chat-n'), Word('ewn-chat-v')]
>>> wn.words('chat', lexicon='wnja')
[]
>>> wn.words('chat', lexicon='wnja frawn')
[Word('frawn-lex14803'), Word('frawn-lex21897')]
```

3.6 Interlingual Queries

This guide explains how interlingual queries work within Wn. To get started, you'll need at least two lexicons that use interlingual indices (ILIs). For this guide, we'll use the Open English WordNet (`oewn:2024`), the Open German WordNet (`odenet:1.4`), also known as OdeNet, and the Japanese wordnet (`omw-ja:1.4`).

```
>>> import wn
>>> wn.download('oewn:2024')
>>> wn.download('odenet:1.4')
>>> wn.download('omw-ja:1.4')
```

We will query these wordnets with the following *Wordnet* objects:

```
>>> en = wn.Wordnet('oewn:2024')
>>> de = wn.Wordnet('odenet:1.4')
```

The object for the Japanese wordnet will be discussed and created below, in *Cross-lingual Relation Traversal*.

3.6.1 What are Interlingual Indices?

It is common for users of the Princeton WordNet to refer to synsets by their WNDB offset and type, but this is problematic because the offset is a byte-offset in the wordnet data files and it will differ for wordnets in other languages and even between versions of the same wordnet. Interlingual indices (ILIs) address this issue by providing stable identifiers for concepts, whether for a synset across versions of a wordnet or across languages.

The idea of ILIs was proposed by [Vossen99] and it came to fruition with the release of the Collaborative Interlingual Index (CILI; [Bond16]). CILI therefore represents an instance of, and a namespace for, ILIs. There could, in theory, be alternative indexes for particular domains (e.g., names of people or places), but currently there is only the one.

As an example, the synset for *apricot* (fruit) in WordNet 3.0 is `07750872-n`, but it is `07766848-n` in WordNet 3.1. In OdeNet 1.4, which is not released in the WNDB format and therefore doesn't use offsets at all, it is `13235-n` for the equivalent word (*Aprikose*). However, all three use the same ILI: `i77784`.

Generally, only one synset within a wordnet will be mapped to a particular ILI, but this may not always be true, nor does every synset necessarily map to an ILI. Some concepts that are lexicalized in one language may not be in another language. For example, *rice* in English may refer to the rice plant, rice grain, or cooked rice, but in languages like Japanese they are distinct things (*ine*, *kome*, and *meshi* / *gohan*, respectively).

The `ili` property of Synsets serves two purposes in Wn. Mainly it is for encoding the ILI identifier associated with the synset, but it is also used to indicate when a lexicon is proposing a new concept that is not yet part of CILI. In the latter case, a WN-LMF lexicon file will have the special value of `in` for a synset's ILI and it will provide an `<ILIDefinition>` element. In Wn, this translates to *wn.Synset.ili* returning **None**, the same as if no ILI were mapped at all. Both synsets with proposed ILIs and those with no ILI cannot be used in interlingual queries. Proposed ILIs can be inspected using the *wn.ili.get_proposed* function, if you know have the synset, or *wn.ili.get_all_proposed* to get all of them.

3.6.2 Using Interlingual Indices

For synsets that have an associated ILI, you can retrieve it via the *wn.Synset.ili* property:

```
>>> apricot = en.synsets('apricot')[1]
>>> apricot.ili
'i77784'
```

The value is a `str` ILI identifier. These may be used directly for things like interlingual synset lookups:

```
>>> de.synsets(ili=apricot.ili)[0].lemmas()
['Marille', 'Aprikose']
```

There may be more information about the ILI itself which you can get from the *wn.ili* module:

```
>>> from wn import ili
>>> apricot_ili = ili.get(apricot.ili)
>>> apricot_ili
ILI(id='i77784')
```

From this object you can get various properties of the ILI, such as the ID string, its status, and its definition, but if you have not added CILI to Wn's database, it will not be very informative:

```
>>> apricot_ili.id
'i77784'
>>> apricot_ili.status
'presupposed'
>>> apricot_ili.definition() is None
True
```

The `presupposed` status means that the ILI ID is in use by a lexicon, but there is no other source of truth for the index. CILI can be downloaded just like a lexicon:

```
>>> wn.download('cili:1.0')
```

Now the status and definition should be more useful:

```
>>> apricot_ili.status
'active'
>>> apricot_ili.definition()
'downy yellow to rosy-colored fruit resembling a small peach'
```

3.6.3 Translating Words, Senses, and Synsets

Rather than manually inserting the ILI IDs into Wn's lookup functions as shown above, Wn provides the *wn.Synset.translate()* method to make it easier:

```
>>> apricot.translate(lexicon='odenet:1.4')
[Synset('odenet-13235-n')]
```

The method returns a list for two reasons: first, it's not guaranteed that the target lexicon has only one synset with the ILI and, second, you can translate to more than one lexicon at a time.

Sense objects also have a *translate()* method, returning a list of senses instead of synsets:

```
>>> de_senses = apricot.senses()[0].translate(lexicon='odenet:1.4')
>>> [s.word().lemma() for s in de_senses]
['Marille', 'Aprikose']
```

Word have a *translate()* method, too, but it works a bit differently. Since each word may be part of multiple synsets, the method returns a mapping of each word sense to the list of translated words:

```
>>> result = en.words('apricot')[0].translate(lexicon='odenet:1.4')
>>> for sense, de_words in result.items():
```

(continues on next page)

(continued from previous page)

```
...     print(sense, [w.lemma() for w in de_words])
...
Sense('oewn-apricot__1.20.00..') []
Sense('oewn-apricot__1.13.00..') ['Marille', 'Aprikose']
Sense('oewn-apricot__1.07.00..') ['lachsrosa', 'lachsfarbig', 'in Lachs', 'lachsfarben',
↪'lachsrot', 'lachs']
```

The three senses above are for *apricot* as a tree, a fruit, and a color. OdeNet does not have a synset for apricot trees, or it has one not associated with the appropriate ILI, and therefore it could not translate any words for that sense.

3.6.4 Cross-lingual Relation Traversal

ILIs have a second use in Wn, which is relation traversal for wordnets that depend on other lexicons, i.e., those created with the *expand* methodology. These wordnets, such as many of those in the Open Multilingual Wordnet, do not include synset relations on their own as they were built using the English WordNet as their taxonomic scaffolding. Trying to load such a lexicon when the lexicon it requires is not added to the database presents a warning to the user:

```
>>> ja = wn.Wordnet('omw-ja:1.4')
[...] WnWarning: lexicon dependencies not available: omw-en:1.4
>>> ja.expanded_lexicons()
[]
```

> **Warning**
>
> Do not rely on the presence of a warning to determine if the lexicon has its expand lexicon loaded. Python's default warning filter may only show the warning the first time it is encountered. Instead, inspect *wn.Wordnet.expanded_lexicons()* to see if it is non-empty.

When a dependency is unmet, Wn only issues a warning, not an error, and you can continue to use the lexicon as it is, but it won't be useful for exploring relations such as hypernyms and hyponyms:

```
>>> anzu = ja.synsets(ili='i77784')[0]
>>> anzu.lemmas()
['', '', '']
>>> anzu.hypernyms()
[]
```

One way to resolve this issue is to install the lexicon it requires:

```
>>> wn.download('omw-en:1.4')
>>> ja = wn.Wordnet('omw-ja:1.4')  # no warning
>>> ja.expanded_lexicons()
[<Lexicon omw-en:1.4 [en]>]
```

Wn will detect the dependency and load `omw-en:1.4` as the *expand* lexicon for `omw-ja:1.4` when the former is in the database. You may also specify an expand lexicon manually, even one that isn't the specified dependency:

```
>>> ja = wn.Wordnet('omw-ja:1.4', expand='oewn:2024')  # no warning
>>> ja.expanded_lexicons()
[<Lexicon oewn:2024 [en]>]
```

In this case, the Open English WordNet is an actively-developed fork of the lexicon that `omw-ja:1.4` depends on, and it should contain all the relations, so you'll see little difference between using it and `omw-en:1.4`. This works because the relations are found using ILIs and not synset offsets. You may still prefer to use the specified dependency if you have strict compatibility needs, such as for experiment reproducibility and/or compatibility with the NLTK. Using some other lexicon as the expand lexicon may yield very different results. For instance, `odenet:1.4` is much smaller than the English wordnets and has fewer relations, so it would not be a good substitute for `omw-ja:1.4`'s expand lexicon.

When an appropriate expand lexicon is loaded, relations between synsets, such as hypernyms, are more likely to be present:

```
>>> anzu = ja.synsets(ili='i77784')[0]  # recreate the synset object
>>> anzu.hypernyms()
[Synset('omw-ja-07705931-n')]
>>> anzu.hypernyms()[0].lemmas()
['']
>>> anzu.hypernyms()[0].translate(lexicon='oewn:2024')[0].lemmas()
['edible fruit']
```

3.7 The Structure of a Wordnet

A **wordnet** is an online lexicon which is organized by concepts.

The basic unit of a wordnet is the synonym set (**synset**), a group of words that all refer to the same concept. Words and synsets are linked by means of conceptual-semantic relations to form the structure of wordnet.

3.7.1 Words, Senses, and Synsets

We all know that **words** are the basic building blocks of languages, a word is built up with two parts, its form and its meaning, but in natural languages, the word form and word meaning are not in an elegant one-to-one match, one word form may connect to many different meanings, so hereforth, we need **senses**, to work as the unit of word meanings, for example, the word *bank* has at least two senses:

1. bank[1]: financial institution, like *City Bank*;
2. bank[2]: sloping land, like *river bank*;

Since **synsets** are group of words sharing the same concept, bank[1]and bank[2]are members of two different synsets, although they have the same word form.

On the other hand, different word forms may also convey the same concept, such as *cab* and *taxi*, these word forms with the same concept are grouped together into one synset.

Figure: relations between words, senses and synsets

3.7.2 Synset Relations

In wordnet, synsets are linked with each other to form various kinds of relations. For example, if the concept expressed by a synset is more general than a given synset, then it is in a *hypernym* relation with the given synset. As shown in the figure below, the synset with *car*, *auto* and *automobile* as its member is the *hypernym* of the other synset with *cab*, *taxi* and *hack*. Such relation which is built on the synset level is categorized as synset relations.

Figure: example of synset relations

3.7.3 Sense Relations

Some relations in wordnet are also built on sense level, which can be further divided into two types, relations that link sense with another sense, and relations that link sense with another synset.

> **Note**
>
> In wordnet, synset relation and sense relation can both employ a particular relation type, such as domain topic.

Sense-Sense

Sense to sense relations emphasize the connections between different senses, especially when dealing with morphologically related words. For example, *behavioral* is the adjective to the noun *behavior*, which is known as in the *pertainym* relation with *behavior*, however, such relation doesn't exist between *behavioral* and *conduct*, which is a synonym of *behavior* and is in the same synset. Here *pertainym* is a sense-sense relation.

Figure: example of sense-sense relations

Sense-Synset

Sense-synset relations connect a particular sense with a synset. For example, *cursor* is a term in the *computer science* discipline, in wordnet, it is in the *has domain topic* relation with the *computer science* synset, but *pointer*, which is in the same synset with *cursor*, is not a term, thus has no such relation with *computer science* synset.

Figure: example of sense-synset relations

3.7.4 Other Information

A wordnet should be built in an appropriate form, two schemas are accepted:

- XML schema based on the Lexical Markup Framework (LMF)
- JSON-LD using the Lexicon Model for Ontologies

The structure of a wordnet should contain below info:

Definition

Definition is used to define senses and synsets in a wordnet, it is given in the language of the wordnet it came from.

Example

Example is used to clarify the senses and synsets in a wordnet, users can understand the definition more clearly with a given example.

Metadata

A wordnet has its own metadata, based on the Dublin Core, to state the basic info of it, below table lists all the items in the metadata of a wordnet:

contributor	Optional	str
coverage	Optional	str
creator	Optional	str
date	Optional	str
description	Optional	str
format	Optional	str
identifier	Optional	str
publisher	Optional	str
relation	Optional	str
rights	Optional	str
source	Optional	str
subject	Optional	str
title	Optional	str
type	Optional	str
status	Optional	str
note	Optional	str
confidence	Optional	float

3.8 Lemmatization and Normalization

Wn provides two methods for expanding queries: *lemmatization* and *normalization*. Wn also has a setting that allows *alternative forms* stored in the database to be included in queries.

See also

The *wn.morphy* module is a basic English lemmatizer included with Wn.

3.8.1 Lemmatization

When querying a wordnet with wordforms from natural language text, it is important to be able to find entries for inflected forms as the database generally contains only lemmatic forms, or *lemmas* (or *lemmata*, if you prefer irregular plurals).

```
>>> import wn
>>> en = wn.Wordnet('oewn:2021')
>>> en.words('plurals')
[]
>>> en.words('plural')
[Word('oewn-plural-a'), Word('oewn-plural-n')]
```

Lemmas are sometimes called *citation forms* or *dictionary forms* as they are often used as the head words in dictionary entries. In Natural Language Processing (NLP), *lemmatization* is a technique where a possibly inflected word form is transformed to yield a lemma. In Wn, this concept is generalized somewhat to mean a transformation that yields a form matching wordforms stored in the database. For example, the English word *sparrows* is the plural inflection of *sparrow*, while the word *leaves* is ambiguous between the plural inflection of the nouns *leaf* and *leave* and the 3rd-person singular inflection of the verb *leave*.

For tasks where high-accuracy is needed, wrapping the wordnet queries with external tools that handle tokenization, lemmatization, and part-of-speech tagging will likely yield the best results as this method can make use of word context. That is, something like this:

```
for lemma, pos in fancy_shmancy_analysis(corpus):
    synsets = w.synsets(lemma, pos=pos)
```

For modest needs, however, Wn provides a way to integrate basic lemmatization directly into the queries.

Lemmatization in Wn works as follows: if a *wn.Wordnet* object is instantiated with a *lemmatizer* argument, then queries involving wordforms (e.g., *wn.Wordnet.words()*, *wn.Wordnet.senses()*, *wn.Wordnet.synsets()*) will first lemmatize the wordform and then check all resulting wordforms and parts of speech against the database as successive queries.

Lemmatization Functions

The *lemmatizer* argument of *wn.Wordnet* is a callable that takes two string arguments: (1) the original wordform, and (2) a part-of-speech or **None**. It returns a dictionary mapping parts-of-speech to sets of lemmatized wordforms. The signature is as follows:

```
lemmatizer(s: str, pos: str | None) -> Dict[str | None, Set[str]]
```

The part-of-speech may be used by the function to determine which morphological rules to apply. If the given part-of-speech is **None**, then it is not specified and any rule may apply. A lemmatizer that only deinflects should not change any specified part-of-speech, but this is not a requirement, and a function could be provided that undoes derivational morphology (e.g., *democratic* → *democracy*).

Querying With Lemmatization

As the needs of lemmatization differs from one language to another, Wn does not provide a lemmatizer by default, and therefore it is unavailable to the convenience functions *wn.words()*, *wn.senses()*, and *wn.synsets()*. A lemmatizer can be added to a *wn.Wordnet* object. For example, using *wn.morphy*:

```
>>> import wn
>>> from wn.morphy import Morphy
>>> en = wn.Wordnet('oewn:2021', lemmatizer=Morphy())
>>> en.words('sparrows')
[Word('oewn-sparrow-n')]
>>> en.words('leaves')
[Word('oewn-leave-v'), Word('oewn-leaf-n'), Word('oewn-leave-n')]
```

Querying Without Lemmatization

When lemmatization is not used, inflected terms may not return any results:

```
>>> en = wn.Wordnet('oewn:2021')
>>> en.words('sparrows')
[]
```

Depending on the lexicon, there may be situations where results are returned for inflected lemmas, such as when the inflected form is lexicalized as its own entry:

```
>>> en.words('glasses')
[Word('oewn-glasses-n')]
```

Or if the lexicon lists the inflected form as an alternative form. For example, the English Wordnet lists irregular inflections as alternative forms:

```
>>> en.words('lemmata')
[Word('oewn-lemma-n')]
```

See below for excluding alternative forms from such queries.

3.8.2 Alternative Forms in the Database

A lexicon may include alternative forms in addition to lemmas for each word, and by default these are included in queries. What exactly is included as an alternative form depends on the lexicon. The English Wordnet, for example, adds irregular inflections (or "exceptional forms"), while the Japanese Wordnet includes the same word in multiple orthographies (original, hiragana, katakana, and two romanizations). For the English Wordnet, this means that you might get basic lemmatization for irregular forms only:

```
>>> en = wn.Wordnet('oewn:2021')
>>> en.words('learnt', pos='v')
[Word('oewn-learn-v')]
>>> en.words('learned', pos='v')
[]
```

If this is undesirable, the alternative forms can be excluded from queries with the *search_all_forms* parameter:

```
>>> en = wn.Wordnet('oewn:2021', search_all_forms=False)
>>> en.words('learnt', pos='v')
[]
>>> en.words('learned', pos='v')
[]
```

3.8.3 Normalization

While lemmatization deals with morphological variants of words, normalization handles minor orthographic variants. Normalized forms, however, may be invalid as wordforms in the target language, and as such they are only used behind the scenes for query expansion and not presented to users. For instance, a user might attempt to look up *résumé* in the English wordnet, but the wordnet only contains the form without diacritics: *resume*. With strict string matching, the entry would not be found using the wordform in the query. By normalizing the query word, the entry can be found. Similarly in the Spanish wordnet, *soñar* (to dream) and *sonar* (to ring) are two different words. A user who types *soñar* likely does not want to get results for *sonar*, but one who types *sonar* may be a non-Spanish speaker who is unaware of the missing diacritic or does not have an input method that allows them to type the diacritic, so this query would return both entries by matching against the normalized forms in the database. Wn handles all of these use cases.

When a lexicon is added to the database, potentially two wordforms are inserted for every one in the lexicon: the original wordform and a normalized form. When querying against the database, the original query string is first compared with the original wordforms and, if normalization is enabled, with the normalized forms in the database as well. If this first attempt yields no results and if normalization is enabled, the query string is normalized and tried again.

Normalization Functions

The normalized form is obtained from a *normalizer* function, passed as an argument to *wn.Wordnet*, that takes a single string argument and returns a string. That is, a function with the following signature:

```
normalizer(s: str) -> str
```

While custom *normalizer* functions could be used, in practice the choice is either the default normalizer or **None**. The default normalizer works by downcasing the string and performing NFKD normalization to remove diacritics. If the normalized form is the same as the original, only the original is inserted into the database.

Table 1: Examples of normalization

Original Form	Normalized Form
résumé	resume
soñar	sonar
San José	san jose

Querying With Normalization

By default, normalization is enabled when a *wn.Wordnet* is created. Enabling normalization does two things: it allows queries to check the original wordform in the query against the normalized forms in the database and, if no results are returned in the first step, it allows the queried wordform to be normalized as a back-off technique.

```
>>> en = wn.Wordnet('oewn:2021')
>>> en.words('résumé')
[Word('oewn-resume-n'), Word('oewn-resume-v')]
>>> es = wn.Wordnet('omw-es:1.4')
>>> es.words('soñar')
[Word('omw-es-soñar-v')]
>>> es.words('sonar')
[Word('omw-es-sonar-v'), Word('omw-es-soñar-v')]
```

Note

Users may supply a custom *normalizer* function to the *wn.Wordnet* object, but currently this is discouraged as the result is unlikely to match normalized forms in the database and there is not yet a way to customize the normalization of forms added to the database.

Querying Without Normalization

Normalization can be disabled by passing **None** as the argument of the *normalizer* parameter of *wn.Wordnet*. The queried wordform will not be checked against normalized forms in the database and neither will it be normalized as a back-off technique.

```
>>> en = wn.Wordnet('oewn:2021', normalizer=None)
>>> en.words('résumé')
[]
>>> es = wn.Wordnet('omw-es:1.4', normalizer=None)
>>> es.words('soñar')
[Word('omw-es-soñar-v')]
>>> es.words('sonar')
[Word('omw-es-sonar-v')]
```

Note

It is not possible to disable normalization for the convenience functions *wn.words()*, *wn.senses()*, and *wn.synsets()*.

3.9 Migrating from the NLTK

This guide is for users of the NLTK's `nltk.corpus.wordnet` module who are migrating to Wn. It is not guaranteed that Wn will produce the same results as the NLTK's module, but with some care its behavior can be very similar.

3.9.1 Overview

One important thing to note is that Wn will search all wordnets in the database by default where the NLTK would only search the English.

```
>>> from nltk.corpus import wordnet as nltk_wn
>>> nltk_wn.synsets('chat')                  # only English
>>> nltk_wn.synsets('chat', lang='fra')      # only French
>>> import wn
>>> wn.synsets('chat')                       # all wordnets
>>> wn.synsets('chat', lang='fr')            # only French
```

With Wn it helps to create a *wn.Wordnet* object to pre-filter the results by language or lexicon.

```
>>> en = wn.Wordnet('omw-en:1.4')
>>> en.synsets('chat')                      # only the OMW English Wordnet
```

3.9.2 Equivalent Operations

The following table lists equivalent API calls for the NLTK's wordnet module and Wn assuming the respective modules have been instantiated (in separate Python sessions) as follows:

NLTK:

```
>>> from nltk.corpus import wordnet as wn
>>> ss = wn.synsets("chat", pos="v")[0]
```

Wn:

```
>>> import wn
>>> en = wn.Wordnet('omw-en:1.4')
>>> ss = en.synsets("chat", pos="v")[0]
```

Primary Queries

NLTK	Wn
`wn.langs()`	`[lex.language for lex in wn.lexicons()]`
`wn.lemmas("chat")`	–
–	`en.words("chat")`
–	`en.senses("chat")`
`wn.synsets("chat")`	`en.synsets("chat")`
`wn.synsets("chat", pos="v")`	`en.synsets("chat", pos="v")`
`wn.all_synsets()`	`en.synsets()`
`wn.all_synsets(pos="v")`	`en.synsets(pos="v")`

Synsets – Basic

NLTK	Wn
`ss.lemmas()`	–
–	`ss.senses()`
–	`ss.words()`
`ss.lemmas_names()`	`ss.lemmas()`
`ss.definition()`	`ss.definition()`
`ss.examples()`	`ss.examples()`
`ss.pos()`	`ss.pos`

Synsets – Relations

NLTK	Wn
`ss.hypernyms()`	`ss.get_related("hypernym")`
`ss.instance_hypernyms()`	`ss.get_related("instance_hypernym")`
`ss.hypernyms() + ss.instance_hypernyms()`	`ss.hypernyms()`
`ss.hyponyms()`	`ss.get_related("hyponym")`
`ss.member_holonyms()`	`ss.get_related("holo_member")`
`ss.member_meronyms()`	`ss.get_related("mero_member")`
`ss.closure(lambda x:  x.hypernyms())`	`ss.closure("hypernym")`

Synsets – Taxonomic Structure

NLTK	Wn
`ss.min_depth()`	`ss.min_depth()`
`ss.max_depth()`	`ss.max_depth()`
`ss.hypernym_paths()`	`[list(reversed([ss] + p)) for p in ss.hypernym_paths()]`
`ss.common_hypernyms(ss)`	`ss.common_hypernyms(ss)`
`ss.lowest_common_hypernyms(ss)`	`ss.lowest_common_hypernyms(ss)`
`ss.shortest_path_distance(ss)`	`len(ss.shortest_path(ss))`

(these tables are incomplete)

3.10 wn

Wordnet Interface.

3.10.1 Project Management Functions

wn.**download**(*project_or_url: str*, *add: bool = True*, *progress_handler: type[~wn.util.ProgressHandler] | None = <class 'wn.util.ProgressBar'>*) → Path

Download the resource specified by *project_or_url*.

First the URL of the resource is determined and then, depending on the parameters, the resource is downloaded and added to the database. The function then returns the path of the cached file.

If *project_or_url* starts with *'http://'* or *'https://'*, then it is taken to be the URL for the resource. Otherwise, *project_or_url* is taken as a *project specifier* and the URL is taken from a matching entry in Wn's project index. If no project matches the specifier, *wn.Error* is raised.

If the URL has been downloaded and cached before, the cached file is used. Otherwise the URL is retrieved and stored in the cache.

If the *add* paramter is `True` (default), the downloaded resource is added to the database.

```
>>> wn.download("ewn:2020")
Added ewn:2020 (English WordNet)
```

The *progress_handler* parameter takes a subclass of *wn.util.ProgressHandler*. An instance of the class will be created, used, and closed by this function.

wn.add(*source: str | ~pathlib.Path*, *progress_handler: type[~wn.util.ProgressHandler] | None = <class 'wn.util.ProgressBar'>*) → None

Add the LMF or ILI file at *source* to the database.

The file at *source* may be gzip-compressed or plain text file.

```
>>> wn.add("english-wordnet-2020.xml")
Added ewn:2020 (English WordNet)
```

The *progress_handler* parameter takes a subclass of *wn.util.ProgressHandler*. An instance of the class will be created, used, and closed by this function.

wn.add_lexical_resource(*resource: ~wn.lmf.LexicalResource*, *progress_handler: type[~wn.util.ProgressHandler] | None = <class 'wn.util.ProgressBar'>*) → None

Add the lexical resource *resource* to the database.

The *resource* argument is an in-memory lexical resource as from *wn.lmf.load()* and not a file on disk.

```
>>> resource = wn.lmf.load("english-wordnet-2024.xml")
>>> wn.add_lexical_resource(resource)
Added ewn:2020 (English WordNet)
```

The *progress_handler* parameter takes a subclass of *wn.util.ProgressHandler*. An instance of the class will be created, used, and closed by this function.

wn.remove(*lexicon: str*, *progress_handler: type[~wn.util.ProgressHandler] = <class 'wn.util.ProgressBar'>*) → None

Remove lexicon(s) from the database.

The *lexicon* argument is a *lexicon specifier*. Note that this removes a lexicon and not a project, so the lexicons of projects containing multiple lexicons will need to be removed individually or, if applicable, a star specifier.

The *progress_handler* parameter takes a subclass of *wn.util.ProgressHandler*. An instance of the class will be created, used, and closed by this function.

```
>>> wn.remove("ewn:2019")  # removes a single lexicon
>>> wn.remove("*:1.3+omw")  # removes all lexicons with version 1.3+omw
```

wn.export(*lexicons: Sequence[Lexicon]*, *destination: str | Path*, *version: str = '1.4'*) → None

Export lexicons from the database to a WN-LMF file.

More than one lexicon may be exported in the same file, subject to these conditions:

- identifiers on wordnet entities must be unique in all lexicons

- lexicons extensions may not be exported with their dependents

```
>>> w = wn.Wordnet(lexicon="omw-cmn:1.4 omw-zsm:1.4")
>>> wn.export(w.lexicons(), "cmn-zsm.xml")
```

Parameters

- **lexicons** – sequence of *wn.Lexicon* objects
- **destination** – path to the destination file
- **version** – LMF version string

wn.**projects**() → list[*ResolvedProjectInfo*]

Return the list of indexed projects.

This returns the same dictionaries of information as *wn.config.get_project_info*, but for all indexed projects.

Example

```
>>> infos = wn.projects()
>>> len(infos)
36
>>> infos[0]["label"]
'Open English WordNet'
```

wn.**reset_database**(*rebuild: bool = False*) → None

Delete and recreate the database file.

If *rebuild* is `True`, Wn will attempt to add all lexicons that are added in the existing database. Note that this will only attempt to add indexed projects via their lexicon specifiers, (using `wn.download(specifier)`) regardless of how they were originally added, and will not attempt to add resources from unindexed URLs or local files (unless those local files are cached versions of indexed resources).

This function is useful when database schema changes necessitate a rebuild or when testing requires a clean database.

Warning

This will completely delete the database and all added resources. It does not delete the download cache. Using `rebuild=True` does not re-add non-lexicon resources like CILI files or unindexed resources, so you will need to add those manually.

3.10.2 Wordnet Query Functions

While it is best to first instantiate a *Wordnet* object with a specific lexicon and use that for querying (see *Default Mode Queries*), the following functions are also available for quick and simple queries.

wn.**word**(*id: str*, ** (Keyword-only parameters separator (PEP 3102))*, *lexicon: str | None = None*, *lang: str | None = None*) → *Word*

Return the word with *id* in *lexicon*.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The *id* argument is then passed to the *Wordnet.word()* method.

```
>>> wn.word("ewn-cell-n")
Word('ewn-cell-n')
```

wn.**words**(*form: str | None = None*, *pos: str | None = None*, ***, *lexicon: str | None = None*, *lang: str | None = None*) → list[*Word*]

Return the list of matching words.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The remaining arguments are passed to the *Wordnet.words()* method.

```
>>> len(wn.words())
282902
>>> len(wn.words(pos="v"))
34592
>>> wn.words(form="scurry")
[Word('ewn-scurry-n'), Word('ewn-scurry-v')]
```

wn.**lemmas**(*form: str | None = None*, *pos: str | None = None*, ***, *data: Literal[False] = False*, *lexicon: str | None = None*, *lang: str | None = None*) → list[str]

wn.**lemmas**(*form: str | None = None*, *pos: str | None = None*, ***, *data: Literal[True] = True*, *lexicon: str | None = None*, *lang: str | None = None*) → list[*Form*]

wn.**lemmas**(*form: str | None = None*, *pos: str | None = None*, ***, *data: bool*, *lexicon: str | None = None*, *lang: str | None = None*) → list[str] | list[*Form*]

Return the list of lemmas for matching words.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The remaining arguments are passed to the *Wordnet.lemmas()* method.

If the *data* argument is **False** (the default), the lemmas are returned as `str` types. If it is **True**, *wn.Form* objects are used instead.

```
>>> wn.lemmas("wolves")
['wolf']
>>> wn.lemmas("wolves", data=True)
[Form(value='wolf')]
>>> len(wn.lemmas(pos="v"))
11617
```

wn.**sense**(*id: str*, ***, *lexicon: str | None = None*, *lang: str | None = None*) → *Sense*

Return the sense with *id* in *lexicon*.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The *id* argument is then passed to the *Wordnet.sense()* method.

```
>>> wn.sense("ewn-flutter-v-01903884-02")
Sense('ewn-flutter-v-01903884-02')
```

wn.**senses**(*form: str | None = None*, *pos: str | None = None*, ***, *lexicon: str | None = None*, *lang: str | None = None*) → list[*Sense*]

Return the list of matching senses.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The remaining arguments are passed to the *Wordnet.senses()* method.

```
>>> len(wn.senses("twig"))
3
>>> wn.senses("twig", pos="n")
[Sense('ewn-twig-n-13184889-02')]
```

wn.**synset**(*id: str*, *, *lexicon: str | None = None*, *lang: str | None = None*) → *Synset*

Return the synset with *id* in *lexicon*.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The *id* argument is then passed to the *Wordnet.synset()* method.

```
>>> wn.synset("ewn-03311152-n")
Synset('ewn-03311152-n')
```

wn.**synsets**(*form: str | None = None*, *pos: str | None = None*, *ili: str | None = None*, *, *lexicon: str | None = None*, *lang: str | None = None*) → list[*Synset*]

Return the list of matching synsets.

This will create a *Wordnet* object using the *lang* and *lexicon* arguments. The remaining arguments are passed to the *Wordnet.synsets()* method.

```
>>> len(wn.synsets("couch"))
4
>>> wn.synsets("couch", pos="v")
[Synset('ewn-00983308-v')]
```

wn.**lexicons**(*, *lexicon: str | None = '*'*, *lang: str | None = None*) → list[*Lexicon*]

Return the lexicons matching a language or lexicon specifier.

Example

```
>>> wn.lexicons(lang="en")
[<Lexicon ewn:2020 [en]>, <Lexicon omw-en:1.4 [en]>]
```

3.10.3 The Wordnet Class

class wn.**Wordnet**(*lexicon: str | None = None, *, lang: str | None = None, expand: str | None = None, normalizer: ~collections.abc.Callable[[str], str] | None = <function normalize_form>, lemmatizer: ~collections.abc.Callable[[str, str | None], dict[str | None, set[str]]] | None = None, search_all_forms: bool = True*)

Class for interacting with wordnet data.

A wordnet object acts essentially as a filter by first selecting matching lexicons and then searching only within those lexicons for later queries. Lexicons can be selected on instantiation with the *lexicon* or *lang* parameters. The *lexicon* parameter is a string with a space-separated list of *lexicon specifiers*. The *lang* argument is a BCP 47 language code that selects any lexicon matching the given language code. As the *lexicon* argument more precisely selects lexicons, it is the recommended method of instantiation. Omitting both *lexicon* and *lang* arguments triggers *default-mode* queries.

Some wordnets were created by translating the words from a larger wordnet, namely the Princeton WordNet, and then relying on the larger wordnet for structural relations. An *expand* argument is a second space-separated list of lexicon specifiers which are used for traversing relations, but not as the results of queries. Setting *expand* to an empty string (`expand=''`) disables expand lexicons. For more information, see *Cross-lingual Relation Traversal*.

The *normalizer* argument takes a callable that normalizes word forms in order to expand the search. The default function downcases the word and removes diacritics via NFKD normalization so that, for example, searching for *san josé* in the English WordNet will find the entry for *San Jose*. Setting *normalizer* to **None** disables normalization and forces exact-match searching. For more information, see *Normalization*.

The *lemmatizer* argument may be **None**, which is the default and disables lemmatizer-based query expansion, or a callable that takes a word form and optional part of speech and returns base forms of the original word. To support lemmatizers that use the wordnet for instantiation, such as *wn.morphy*, the lemmatizer may be assigned to the *lemmatizer* attribute after creation. For more information, see *Lemmatization*.

If the *search_all_forms* argument is **True** (the default), searches of word forms consider all forms in the lexicon; if **False**, only lemmas are searched. Non-lemma forms may include, depending on the lexicon, morphological exceptions, alternate scripts or spellings, etc.

lemmatizer

A lemmatization function or **None**.

word(*id: str*) → *Word*

Return the first word in this wordnet with identifier *id*.

words(*form: str | None = None*, *pos: str | None = None*) → list[*Word*]

Return the list of matching words in this wordnet.

Without any arguments, this function returns all words in the wordnet's selected lexicons. A *form* argument restricts the words to those matching the given word form, and *pos* restricts words by their part of speech.

lemmas(*form: str | None = None*, *pos: str | None = None*, *, *data: Literal[False] = False*) → list[str]

lemmas(*form: str | None = None*, *pos: str | None = None*, *, *data: Literal[True] = True*) → list[*Form*]

lemmas(*form: str | None = None*, *pos: str | None = None*, *, *data: bool*) → list[str] | list[*Form*]

Return the list of lemmas for matching words in this wordnet.

Without any arguments, this function returns all distinct lemma forms in the wordnet's selected lexicons. A *form* argument restricts the words to those matching the given word form, and *pos* restricts words by their part of speech.

If the *data* argument is **False** (the default), only distinct lemma forms are returned as `str` types. If it is **True**, *wn.Form* objects are returned for all matching entries, which may include multiple Form objects with the same lemma string.

Example

```
>>> wn.Wordnet().lemmas("wolves")
['wolf']
>>> wn.Wordnet().lemmas("wolves", data=True)
[Form(value='wolf')]
```

sense(*id: str*) → *Sense*

Return the first sense in this wordnet with identifier *id*.

senses(*form: str | None = None*, *pos: str | None = None*) → list[*Sense*]

Return the list of matching senses in this wordnet.

Without any arguments, this function returns all senses in the wordnet's selected lexicons. A *form* argument restricts the senses to those whose word matches the given word form, and *pos* restricts senses by their word's part of speech.

synset(*id: str*) → *Synset*

Return the first synset in this wordnet with identifier *id*.

synsets(*form: str | None = None*, *pos: str | None = None*, *ili: str | None = None*) → list[*Synset*]

Return the list of matching synsets in this wordnet.

Without any arguments, this function returns all synsets in the wordnet's selected lexicons. A *form* argument restricts synsets to those whose member words match the given word form. A *pos* argument restricts synsets to those with the given part of speech. An *ili* argument restricts synsets to those with the given interlingual index; generally this should select a unique synset within a single lexicon.

lexicons() → list[*Lexicon*]

Return the list of lexicons covered by this wordnet.

expanded_lexicons() → list[*Lexicon*]

Return the list of expand lexicons for this wordnet.

describe() → str

Return a formatted string describing the lexicons in this wordnet.

Example

```
>>> oewn = wn.Wordnet("oewn:2021")
>>> print(oewn.describe())
Primary lexicons:
  oewn:2021
    Label  : Open English WordNet
    URL    : https://github.com/globalwordnet/english-wordnet
    License: https://creativecommons.org/licenses/by/4.0/
    Words  : 163161 (a: 8386, n: 123456, r: 4481, s: 15231, v: 11607)
    Senses : 211865
    Synsets: 120039 (a: 7494, n: 84349, r: 3623, s: 10727, v: 13846)
    ILIs   : 120039
```

3.10.4 Words, Senses, and Synsets

The results of primary queries against a lexicon are *Word*, *Sense*, or *Synset* objects. See *The Structure of a Wordnet* for more information about the concepts these object represent.

Word Objects

class wn.**Word**

Word (or "lexical entry") objects encode information about word forms independent from their meaning.

id: str

The identifier used within a lexicon.

pos: str

The part of speech of the Word.

lemma(*, *data: Literal[False] = False*) → str

lemma(*, *data: Literal[True] = True*) → *Form*

lemma(*, *data: bool*) → str | *Form*

Return the canonical form of the word.

If the *data* argument is **False** (the default), the lemma is returned as a `str` type. If it is **True**, a *wn.Form* object is used instead.

Example

```
>>> wn.words("wolves")[0].lemma()
'wolf'
>>> wn.words("wolves")[0].lemma(data=True)
Form(value='wolf')
```

forms(*, *data: Literal[False] = False*) → list[str]

forms(*, *data: Literal[True] = True*) → list[*Form*]

forms(*, *data: bool*) → list[str] | list[*Form*]

Return the list of all encoded forms of the word.

If the *data* argument is **False** (the default), the forms are returned as `str` types. If it is **True**, *wn.Form* objects are used instead.

Example

```
>>> wn.words("wolf")[0].forms()
['wolf', 'wolves']
>>> wn.words("wolf")[0].forms(data=True)
[Form(value='wolf'), Form(value='wolves')]
```

senses() → list[*Sense*]

Return the list of senses of the word.

Example

```
>>> wn.words("zygoma")[0].senses()
[Sense('ewn-zygoma-n-05292350-01')]
```

synsets() → list[*Synset*]

Return the list of synsets of the word.

Example

```
>>> wn.words("addendum")[0].synsets()
[Synset('ewn-06411274-n')]
```

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the word's metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

derived_words() → list[*Word*]

Return the list of words linked through derivations on the senses.

Example

```
>>> wn.words("magical")[0].derived_words()
[Word('ewn-magic-n'), Word('ewn-magic-n')]
```

translate(*lexicon: str | None = None*, *, *lang: str | None = None*) → dict[*Sense*, list[*Word*]]

Return a mapping of word senses to lists of translated words.

Parameters

- **lexicon** – lexicon specifier of translated words
- **lang** – BCP-47 language code of translated words

Example

```
>>> w = wn.words("water bottle", pos="n")[0]
>>> for sense, words in w.translate(lang="ja").items():
...     print(sense, [jw.lemma() for jw in words])
Sense('ewn-water_bottle-n-04564934-01') ['']
```

Sense Objects

class wn.**Sense**

Sense objects represent a pairing of a *Word* and a *Synset*.

id: str

The identifier used within a lexicon.

word() → *Word*

Return the word of the sense.

Example

```
>>> wn.senses("spigot")[0].word()
Word('pwn-spigot-n')
```

synset() → *Synset*

Return the synset of the sense.

Example

```
>>> wn.senses("spigot")[0].synset()
Synset('pwn-03325088-n')
```

examples(*, *data: Literal[False] = False*) → list[str]

examples(*, *data: Literal[True] = True*) → list[*Example*]

examples(*, *data: bool*) → list[str] | list[*Example*]

Return the list of examples for the sense.

If the *data* argument is **False** (the default), the examples are returned as str types. If it is **True**, *wn.Example* objects are used instead.

lexicalized() → bool

Return True if the sense is lexicalized.

adjposition() → str | None

Return the adjective position of the sense.

Values include `"a"` (attributive), `"p"` (predicative), and `"ip"` (immediate postnominal). Note that this is only relevant for adjectival senses. Senses for other parts of speech, or for adjectives that are not annotated with this feature, will return `None`.

frames() → list[str]

Return the list of subcategorization frames for the sense.

counts(*, *data: Literal[False] = False*) → list[int]

counts(*, *data: Literal[True] = True*) → list[*Count*]

counts(*, *data: bool*) → list[int] | list[*Count*]

Return the corpus counts stored for this sense.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the sense's metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

relations(**args: str*, *data: Literal[False] = False*) → dict[str, list[*Sense*]]

relations(**args: str*, *data: Literal[True] = True*) → dict[*Relation*, *Sense*]

relations(**args: str*, *data: bool = False*) → dict[str, list[*Sense*]] | dict[*Relation*, *Sense*]

Return a mapping of relation names to lists of senses.

One or more relation names may be given as positional arguments to restrict the relations returned. If no such arguments are given, all relations starting from the sense are returned.

If the *data* argument is **`False`** (default), the returned object maps from the relation name (a `str`) to a list of *`Sense`* objects. If *data* is **`True`**, it instead maps from a *`Relation`* to a single *`Sense`*.

See *`get_related()`* for getting a flat list of related senses.

synset_relations(**args: str*, *data: Literal[False] = False*) → dict[str, list[*Synset*]]

synset_relations(**args: str*, *data: Literal[True] = True*) → dict[*Relation*, *Synset*]

synset_relations(**args: str*, *data: bool = False*) → dict[str, list[*Synset*]] | dict[*Relation*, *Synset*]

Return a mapping of relation names to lists of synsets.

One or more relation names may be given as positional arguments to restrict the relations returned. If no such arguments are given, all relations starting from the sense are returned.

If the *data* argument is **`False`** (default), the returned object maps from the relation name (a `str`) to a list of *`Synset`* objects. If *data* is **`True`**, it instead maps from a *`Relation`* to a single *`Synset`*.

See *`get_related_synsets()`* for getting a flat list of related synsets.

get_related(**args: str*) → list[*Sense*]

Return a list of related senses.

One or more relation types should be passed as arguments which determine the kind of relations returned.

Example

```
>>> physics = wn.senses("physics", lexicon="ewn")[0]
>>> for sense in physics.get_related("has_domain_topic"):
...     print(sense.word().lemma())
coherent
chaotic
incoherent
```

get_related_synsets(**args: str*) → list[*Synset*]

Return a list of related synsets.

closure(**args: str*) → Iterator[T]

relation_paths(**args: str*, *end: T | None = None*) → Iterator[list[T]]

translate(*lexicon: str | None = None*, ***, *lang: str | None = None*) → list[*Sense*]

Return a list of translated senses.

Parameters

- **lexicon** – lexicon specifier of translated senses
- **lang** – BCP-47 language code of translated senses

Example

```
>>> en = wn.senses("petiole", lang="en")[0]
>>> pt = en.translate(lang="pt")[0]
>>> pt.word().lemma()
'pecíolo'
```

Synset Objects

class wn.**Synset**

Synset objects represent a set of words that share a meaning.

id: str

The identifier used within a lexicon.

pos: str

The part of speech of the Synset.

property ili: str | None

The interlingual index of the Synset.

definition(***, *data: Literal[False] = False*) → str | None

definition(***, *data: Literal[True] = True*) → *Definition* | None

definition(***, *data: bool*) → str | *Definition* | None

Return the first definition found for the synset.

If the *data* argument is **False** (the default), the definition is returned as a `str` type. If it is **True**, a *wn.Definition* object is used instead.

Example

```
>>> wn.synsets("cartwheel", pos="n")[0].definition()
'a wheel that has wooden spokes and a metal rim'
>>> wn.synsets("cartwheel", pos="n")[0].definition(data=True)
[Definition(text='a wheel that has wooden spokes and a metal rim',
  language=None, source_sense_id=None)]
```

definitions(*, *data: Literal[False] = False*) → list[str]

definitions(*, *data: Literal[True] = True*) → list[*Definition*]

definitions(*, *data: bool*) → list[str] | list[*Definition*]

Return the list of definitions for the synset.

If the *data* argument is **False** (the default), the definitions are returned as `str` objects. If it is **True**, *wn.Definition* objects are used instead.

Example

```
>>> wn.synsets("tea", pos="n")[0].definitions()
['a beverage made by steeping tea leaves in water']
>>> wn.synsets("tea", pos="n")[0].definitions(data=True)
[Definition(text='a beverage made by steeping tea leaves in water',
  language=None, source_sense_id=None)]
```

examples(*, *data: Literal[False] = False*) → list[str]

examples(*, *data: Literal[True] = True*) → list[*Example*]

examples(*, *data: bool*) → list[str] | list[*Example*]

Return the list of examples for the synset.

If the *data* argument is **False** (the default), the examples are returned as `str` types. If it is **True**, *wn.Example* objects are used instead.

Example

```
>>> wn.synsets("orbital", pos="a")[0].examples()
['"orbital revolution"', '"orbital velocity"']
```

senses() → list[*Sense*]

Return the list of sense members of the synset.

Example

```
>>> wn.synsets("umbrella", pos="n")[0].senses()
[Sense('ewn-umbrella-n-04514450-01')]
```

lexicalized() → bool

Return True if the synset is lexicalized.

lexfile() → str | None

Return the lexicographer file name for this synset, if any.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the synset's metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

words() → list[*Word*]

Return the list of words linked by the synset's senses.

Example

```
>>> wn.synsets("exclusive", pos="n")[0].words()
[Word('ewn-scoop-n'), Word('ewn-exclusive-n')]
```

lemmas(*, *data: Literal[False] = False*) → list[str]

lemmas(*, *data: Literal[True] = True*) → list[*Form*]

lemmas(*, *data: bool*) → list[str] | list[*Form*]

Return the list of lemmas of words for the synset.

If the *data* argument is **False** (the default), the lemmas are returned as `str` types. If it is **True**, *wn.Form* objects are used instead.

Example

```
>>> wn.synsets("exclusive", pos="n")[0].lemmas()
['scoop', 'exclusive']
>>> wn.synsets("exclusive", pos="n")[0].lemmas(data=True)
[Form(value='scoop'), Form(value='exclusive')]
```

hypernyms() → list[*Synset*]

Return the list of synsets related by any hypernym relation.

Both the `hypernym` and `instance_hypernym` relations are traversed.

hyponyms() → list[*Synset*]

Return the list of synsets related by any hyponym relation.

Both the `hyponym` and `instance_hyponym` relations are traversed.

holonyms() → list[*Synset*]

Return the list of synsets related by any holonym relation.

Any of the following relations are traversed: `holonym`, `holo_location`, `holo_member`, `holo_part`, `holo_portion`, `holo_substance`.

meronyms() → list[*Synset*]

Return the list of synsets related by any meronym relation.

Any of the following relations are traversed: `meronym`, `mero_location`, `mero_member`, `mero_part`, `mero_portion`, `mero_substance`.

relations(**args: str*, *data: Literal[False] = False*) → dict[str, list[*Synset*]]

relations(**args: str*, *data: Literal[True] = True*) → dict[*Relation*, *Synset*]

relations(**args: str*, *data: bool = False*) → dict[str, list[*Synset*]] | dict[*Relation*, *Synset*]

Return a mapping of synset relations.

One or more relation names may be given as positional arguments to restrict the relations returned. If no such arguments are given, all relations starting from the synset are returned.

If the *data* argument is **False** (default), the returned object maps from the relation name (a `str`) to a list of *Synset* objects. If *data* is **True**, it instead maps from a *Relation* to a single *Synset*.

See *get_related()* for getting a flat list of related synsets.

Example

```
>>> button_rels = wn.synsets("button")[0].relations()
>>> for relname, sslist in button_rels.items():
...     print(relname, [ss.lemmas() for ss in sslist])
hypernym [['fixing', 'holdfast', 'fastener', 'fastening']]
hyponym [['coat button'], ['shirt button']]
```

get_related(**args: str*) → list[*Synset*]

Return the list of related synsets.

One or more relation names may be given as positional arguments to restrict the relations returned. If no such arguments are given, all relations starting from the synset are returned.

This method does not preserve the relation names that lead to the related synsets. For a mapping of relation names to related synsets, see *relations()*.

Example

```
>>> fulcrum = wn.synsets("fulcrum")[0]
>>> [ss.lemmas() for ss in fulcrum.get_related()]
[['pin', 'pivot'], ['lever']]
```

closure(**args: str*) → Iterator[T]

relation_paths(**args: str*, *end: T | None = None*) → Iterator[list[T]]

translate(*lexicon: str | None = None*, ***, *lang: str | None = None*) → list[*Synset*]

Return a list of translated synsets.

Parameters

- **lexicon** – lexicon specifier of translated synsets
- **lang** – BCP-47 language code of translated synsets

Example

```
>>> es = wn.synsets("araña", lang="es")[0]
>>> en = es.translate(lexicon="ewn")[0]
>>> en.lemmas()
['spider']
```

hypernym_paths(*simulate_root=False*)

Shortcut for *wn.taxonomy.hypernym_paths()*.

min_depth(*simulate_root=False*)

Shortcut for *wn.taxonomy.min_depth()*.

max_depth(*simulate_root=False*)

Shortcut for *wn.taxonomy.max_depth()*.

shortest_path(*other*, *simulate_root=False*)

Shortcut for *wn.taxonomy.shortest_path()*.

common_hypernyms(*other*, *simulate_root=False*)

Shortcut for *wn.taxonomy.common_hypernyms()*.

lowest_common_hypernyms(*other*, *simulate_root=False*)

Shortcut for *wn.taxonomy.lowest_common_hypernyms()*.

3.10.5 Relations

The `Sense.relation_map()` and `Synset.relation_map()` methods return a dictionary mapping *Relation* objects to resolved target senses or synsets. They differ from *Sense.relations()* and *Synset.relations()* in two main ways:

1. Relation objects map 1-to-1 to their targets instead of to a list of targets sharing the same relation name.
2. Relation objects encode not just relation names, but also the identifiers of sources and targets, the lexicons they came from, and any metadata they have.

One reason why *Relation* objects are useful is for inspecting relation metadata, particularly in order to distinguish `other` relations that differ only by the value of their `dc:type` metadata:

```
>>> oewn = wn.Wordnet('oewn:2024')
>>> alloy = oewn.senses("alloy", pos="v")[0]
>>> alloy.relations()  # appears to only have one 'other' relation
{'derivation': [Sense('oewn-alloy__1.27.00..')], 'other': [Sense('oewn-alloy__1.27.00..
↪')]}
>>> for rel in alloy.relation_map():  # but in fact there are two
...     print(rel, rel.subtype)
...
Relation('derivation', 'oewn-alloy__2.30.00..', 'oewn-alloy__1.27.00..') None
Relation('other', 'oewn-alloy__2.30.00..', 'oewn-alloy__1.27.00..') material
Relation('other', 'oewn-alloy__2.30.00..', 'oewn-alloy__1.27.00..') result
```

Another reason why they are useful is to determine the source of a relation used in *interlingual queries*.

```
>>> es = wn.Wordnet("omw-es", expand="omw-en")
>>> mapa = es.synsets("mapa", pos="n")[0]
>>> rel, tgt = next(iter(mapa.relation_map().items()))
>>> rel, rel.lexicon()  # relation comes from omw-en
(Relation('hypernym', 'omw-en-03720163-n', 'omw-en-04076846-n'), <Lexicon omw-en:1.4␣
↪[en]>)
>>> tgt, tgt.words(), tgt.lexicon()  # target is in omw-es
(Synset('omw-es-04076846-n'), [Word('omw-es-representación-n')], <Lexicon omw-es:1.4␣
↪[es]>)
```

class wn.**Relation**

Relation objects model relations between senses or synsets.

name

The name of the relation. Also called the relation "type".

source_id

The identifier of the source entity of the relation.

target_id

The identifier of the target entity of the relation.

subtype

The value of the `dc:type` metadata.

If `dc:type` is not specified in the metadata, `None` is returned instead.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the associated metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

3.10.6 Additional Classes

class wn.**Form**

Form objects are returned by *Word.lemma()* and *Word.forms()* when the `data=True` argument is used, and they make accessible several optional properties of word forms. The word form itself is available via the *value* attribute.

```
>>> inu = wn.words('', lexicon='wnja')[0]
>>> inu.forms(data=True)[3]
Form(value='')
>>> inu.forms(data=True)[3].script
'hira'
```

The *script* is often unspecified (i.e., **None**) and this carries the implicit meaning that the form uses the canonical script for the word's language or wordnet, whatever it may be.

value

The word form string.

id

An optional form identifier used within a lexicon. These identifiers are often **None**.

script

The script of the word form. This should be an ISO 15924 code, or **None**.

pronunciations()

Return the list of *Pronunciation* objects.

tags()

Return the list of *Tag* objects.

lexicon() → *Lexicon*

Return the lexicon containing the element.

class wn.**Pronunciation**

Pronunciation objects encode a text or audio representation of how a word is pronounced. They are returned by *Form.pronunciations()*.

value: str

The encoded pronunciation.

variety: str | None

The language variety this pronunciation belongs to.

notation: str | None

The notation used to encode the pronunciation. For example: the International Phonetic Alphabet (IPA).

phonemic: bool

True when the encoded pronunciation is a generalized phonemic description, or **False** for more precise phonetic transcriptions.

audio: str | None

A URI to an associated audio file.

lexicon() → *Lexicon*

Return the lexicon containing the element.

class wn.**Tag**(*tag: str*, *category: str*, *_lexicon: str = ''*)

A general-purpose tag class for word forms.

Tag objects encode categorical information about word forms. They are returned by *Form.tags()*.

tag: str

The text value of the tag.

category: str

The category, or kind, of the tag.

lexicon() → *Lexicon*

Return the lexicon containing the element.

class wn.**Count**(*value: int*, *_lexicon: str = ''*, *_metadata: Metadata | None = None*)

A count of sense occurrences in some corpus.

Count objects model sense counts previously computed over some corpus. They are returned by *Sense.counts()*.

value: int

The count of sense occurrences.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the associated metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

class wn.**Example**

Example objects model example phrases for senses and synsets. They are returned by *Sense.examples()* and *Synset.examples()* when the `data`=**True** argument is given.

text: **str**

The example text.

language: **str** | **None**

The language of the example.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the example's metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

class wn.**Definition**

Definition objects model synset definitions. They are returned by *Synset.definition()* when the `data`=**True** argument is given.

text: **str**

The example text.

language: **str** | **None**

The language of the example.

source_sense_id: **str** | **None**

The id of the particular sense the definition is for.

lexicon() → *Lexicon*

Return the lexicon containing the element.

metadata() → Metadata

Return the example's metadata.

confidence() → float

Return the confidence score of the element.

If the element does not have an explicit confidence score, the value defaults to that of the lexicon containing the element.

3.10.7 Interlingual Indices

As of Wn v1.0.0, see *wn.ili* classes and functions for ILIs

3.10.8 Lexicon Objects

class wn.**Lexicon**

Lexicon objects contain attributes and metadata about a single *lexicon*.

id: **str**

The lexicon's identifier.

label: str

The full name of lexicon.

language: str

The BCP 47 language code of lexicon.

email: str

The email address of the wordnet maintainer.

license: str

The URL or name of the wordnet's license.

version: str

The version string of the resource.

url: str | None

The project URL of the wordnet.

citation: str | None

The canonical citation for the project.

logo: str | None

A URL or path to a project logo.

metadata() → Metadata

Return the associated metadata.

confidence() → float

Return the confidence score of the lexicon.

If the lexicon does not specify a confidence score, it defaults to 1.0.

specifier() → str

Return the *id:version* lexicon specifier.

modified() → bool

Return True if the lexicon has local modifications.

requires() → dict[str, *Lexicon* | None]

Return the lexicon dependencies.

extends() → *Lexicon* | None

Return the lexicon this lexicon extends, if any.

If this lexicon is not an extension, return None.

extensions(*depth: int = 1*) → list[*Lexicon*]

Return the list of lexicons extending this one.

By default, only direct extensions are included. This is controlled by the *depth* parameter, which if you view extensions as children in a tree where the current lexicon is the root, *depth=1* are the immediate extensions. Increasing this number gets extensions of extensions, or setting it to a negative number gets all "descendant" extensions.

describe(*full: bool = True*) → str

Return a formatted string describing the lexicon.

The *full* argument (default: **True**) may be set to **False** to omit word and sense counts.

Also see: *Wordnet.describe()*

3.10.9 The wn.config Object

Wn's data storage and retrieval can be configured through the *wn.config* object.

See also

Installation and Configuration describes how to configure Wn using the *wn.config* instance.

wn.config = <wn._config.WNConfig object>

It is an instance of the *WNConfig* class, which is defined in a non-public module and is not meant to be instantiated directly. Configuration should occur through the single *wn.config* instance.

class wn._config.**WNConfig**

data_directory

The file system directory where Wn's data is stored.

Assign a new path to change where the database and downloads are stored.

```
>>> wn.config.data_directory = "~/.cache/wn"
>>> wn.config.database_path
PosixPath('/home/username/.cache/wn/wn.db')
>>> wn.config.downloads_directory
PosixPath('/home/username/.cache/wn/downloads')
```

database_path

The path to the database file.

The database path is derived from *data_directory* and cannot be changed directly.

allow_multithreading

If set to **True**, the database connection may be shared across threads. In this case, it is the user's responsibility to ensure that multiple threads don't try to write to the database at the same time. The default is **False**.

downloads_directory

The file system directory where downloads are cached.

The downloads directory is derived from *data_directory* and cannot be changed directly.

add_project(*id: str*, *type:* ResourceType = *ResourceType.WORDNET*, *label: str | None = None*, *language: str | None = None*, *license: str | None = None*, *error: str | None = None*) → None

Add a new wordnet project to the index.

Parameters

- **id** – short identifier of the project
- **type** – project type (default 'wordnet')
- **label** – full name of the project
- **language** – BCP 47 language code of the resource
- **license** – link or name of the project's default license
- **error** – if set, the error message to use when the project is accessed

add_project_version(*id: str*, *version: str*, *url: str | None = None*, *error: str | None = None*, *license: str | None = None*) → None

Add a new resource version for a project.

Exactly one of *url* or *error* must be specified.

Parameters

- **id** – short identifier of the project
- **version** – version string of the resource
- **url** – space-separated list of web addresses for the resource
- **license** – link or name of the resource's license; if not given, the project's default license will be used.
- **error** – if set, the error message to use when the project is accessed

get_project_info(*arg: str*) → *ResolvedProjectInfo*

Return information about an indexed project version.

If the project has been downloaded and cached, the `"cache"` key will point to the path of the cached file, otherwise its value is `None`.

Parameters

arg – a project specifier

Example

```
>>> info = wn.config.get_project_info("oewn:2021")
>>> info["label"]
'Open English WordNet'
```

get_cache_path(*url: str*) → Path

Return the path for caching *url*.

Note that this is just a path operation and does not signify that the file exists in the file system.

list_cache_entries(*arg: str = '*'*) → list[*CacheEntry*]

Return a list of cached resources.

Use *arg* as a pattern to match project specifiers. It defaults to *"*"* to select all cached entries.

Each entry on the list is a dictionary with the keys: * *"path"* – the path of the cached file * *"id"* – the ID of the cached resource * *"version"* – the version of the cached resource * *"url"* – the URL of the cached resource

Note that cached files are stored with a hash of their URL as the filename and that it is not feasible to recover the URL from the hash alone. Therefore, for lexicons downloaded with a URL that does not appear in the index, the ID, version, and URL and will be **None** instead.

update(*data: dict[str, Any]*) → None

Update the configuration with items in *data*.

Items are only inserted or replaced, not deleted. If a project index is provided in the `"index"` key, then either the project must not already be indexed or any project fields (label, language, or license) that are specified must be equal to the indexed project.

load_index(*path: str | Path*) → None

Load and update with the project index at *path*.

The project index is a TOML file containing project and version information. For example:

```
[ewn]
  label = "Open English WordNet"
  language = "en"
  license = "https://creativecommons.org/licenses/by/4.0/"
  [ewn.versions.2019]
    url = "https://en-word.net/static/english-wordnet-2019.xml.gz"
  [ewn.versions.2020]
    url = "https://en-word.net/static/english-wordnet-2020.xml.gz"
```

Auxiliary WNConfig Types

The following classes are argument or return types of *WNConfig* objects. They are documented here for reference, but are not meant to be created directly.

class wn._config.**ResourceType**(**values*)

Enumeration of resource types.

WORDNET = 'wordnet'

ILI = 'ili'

class wn._config.**ProjectInfo**

Dictionary of information about a project.

error: str | None

label: str | None

language: str | None

license: str | None

type: *ResourceType*

versions: dict[str, *VersionInfo*]

class wn._config.**VersionInfo**

Dictionary of information about a resource version.

error: str | None

license: str | None

resource_urls: list[str]

class wn._config.**ResolvedProjectInfo**

Dictionary of information about a specific project resource.

cache: Path | None

id: str

label: str | None

language: str | None

license: str | None

resource_urls: list[str]

type: *ResourceType*

version: str

class wn._config.**CacheEntry**

Dictionary of information about files in the download cache.

id: str | None

path: Path

url: str | None

version: str | None

3.10.10 Exceptions

exception wn.**Error**

Generic error class for invalid wordnet operations.

exception wn.**DatabaseError**

Error class for issues with the database.

exception wn.**WnWarning**

Generic warning class for dubious wordnet operations.

3.11 wn.compat

Compatibility modules for Wn.

This subpackage is a namespace for compatibility modules when working with particular lexicons. Wn is designed to be agnostic to the language or lexicon and not favor one over the other (with the exception of *wn.morphy*, which is English-specific). However, there are some kinds of functionality that would be useful to include in Wn, even if they don't generalize to all lexicons.

3.11.1 Included modules

wn.compat.sensekey

Functions Related to Sense Keys

Sense keys are identifiers of senses that (mostly) persist across wordnet versions. They are only used by the English wordnets. For the OMW lexicons derived from the Princeton WordNet and the EWN 2019/2020 lexicons, the sense key is encoded in the `identifier` metadata of a Sense:

```
>>> import wn
>>> en = wn.Wordnet("omw-en:1.4")
>>> sense = en.sense("omw-en-carrousel-02966372-n")
>>> sense.metadata()
{'identifier': 'carrousel%1:06:01::'}
```

For OEWN 2021+ lexicons, the sense key is encoded in the sense ID, but some characters are escaped or replaced to ensure it is a valid XML ID.

```
>>> oewn = wn.Wordnet("oewn:2024")
>>> sense = oewn.sense("oewn-carousel__1.06.01..")
>>> sense.id
'oewn-carousel__1.06.01..'
```

This module has four functions:

1. *escape()* transforms a sense key into a form that is valid for XML IDs. The *flavor* keyword argument specifies the escaping mechanism and it defaults to "oewn-v2".
2. *unescape()* transforms an escaped sense key back into the original form. The *flavor* keyword is the same as with *escape()*.
3. *sense_key_getter()* creates a function for retrieving the sense key for a given *wn.Sense* object. Depending on the lexicon, it will retrieve the sense key from metadata or it will unescape the sense ID.
4. *sense_getter()* creates a function for retrieving a *wn.Sense* object given a sense key. Depending on the lexicon, it will build and use a mapping of sense key metadata to *wn.Sense* objects, or it will escape the sense key and use the escaped form as the `id` argument for *wn.Wordnet.sense()*.

See also

The documentation from the Princeton WordNet: https://wordnet.princeton.edu/documentation/senseidx5wn

wn.compat.sensekey.**escape**(*sense_key: str*, */ (Positional-only parameter separator (PEP 570))*, *flavor: str = 'oewn-v2'*) → str

Return an escaped sense key that is valid for XML IDs.

The *flavor* argument specifies how the escaping will be done. Its default value is "oewn-v2", which escapes like the Open English Wordnet 2025 editions, including separate rules for the left and right side of the % delimiter. The other possible value is "oewn", which escapes like the Open English Wordnet 2024 and prior editions.

```
>>> from wn.compat import sensekey
>>> sensekey.escape("ceramic%3:01:00::")
'ceramic__3.01.00..'
```

wn.compat.sensekey.**unescape**(*s: str*, */*, *flavor: str = 'oewn-v2'*) → str

Return the original form of an escaped sense key.

The *flavor* argument specifies how the unescaping will be done. Its default value is "oewn-v2", which unescapes like the Open English Wordnet 2025 editions, including separate rules for the left and right side of the __ delimiter. The other possible value is "oewn", which unescapes like the Open English Wordnet 2024 and prior editions.

```
>>> from wn.compat import sensekey
>>> sensekey.unescape("ceramic__3.01.00..")
'ceramic%3:01:00::'
```

Note that this function does not remove any lexicon ID prefixes on sense IDs, so that may need to be done manually:

```
>>> sensekey.unescape("oewn-ceramic__3.01.00..")
'oewn-ceramic%3:01:00::'
>>> sensekey.unescape("oewn-ceramic__3.01.00..".removeprefix("oewn-"))
'ceramic%3:01:00::'
```

wn.compat.sensekey.**sense_key_getter**(*lexicon: str*) → Callable[[*Sense*], str | None]

Return a function that gets sense keys from senses.

The *lexicon* argument determines how the function will retrieve the sense key; i.e., whether it is from the `identifier` metadata or unescaping the sense ID. For any unsupported lexicon, an error is raised.

The function that is returned accepts one argument, a *wn.Sense* (ideally from the same lexicon specified in the *lexicon* argument), and returns a `str` if the sense key exists in the lexicon or `None` otherwise.

```
>>> import wn
>>> from wn.compat import sensekey
>>> oewn = wn.Wordnet("oewn:2024")
>>> get_sense_key = sensekey.sense_key_getter("oewn:2024")
>>> get_sense_key(oewn.senses("alabaster")[0])
'alabaster%3:01:00::'
```

When unescaping a sense ID, if the ID starts with its lexicon's ID and a hyphen (e.g., *"oewn-"*), it is assumed to be a conventional ID prefix and is removed prior to unescaping.

wn.compat.sensekey.**sense_getter**(*lexicon: str*, *wordnet:* Wordnet | *None* = *None*) → Callable[[str], *Sense* | None]

Return a function that gets the sense for a sense key.

The *lexicon* argument determines how the function will retrieve the sense; i.e., whether a mapping between a sense's `identifier` metadata and the sense will be created and used or the escaped sense key is used as the sense ID. For any unsupported lexicon, an error is raised.

The optional *wordnet* object is used as the source of the returned *wn.Sense* objects. If none is provided, a new *wn.Wordnet* object is created using the *lexicon* argument.

The function that is returned accepts one argument, a `str` of the sense key, and returns a *wn.Sense* if the sense key exists in the lexicon or `None` otherwise.

```
>>> import wn
>>> from wn.compat import sensekey
>>> get_sense = sensekey.sense_getter("oewn:2024")
>>> get_sense("alabaster%3:01:00::")
Sense('oewn-alabaster__3.01.00..')
```

Warning

The mapping built for the `omw-en*` or `ewn` lexicons requires significant memory—around 100MiB—to use. The `oewn` lexicons do not require such a mapping and the memory usage is negligible.

3.12 wn.constants

Constants and literals used in wordnets.

3.12.1 Synset Relations

wn.constants.**SYNSET_RELATIONS**

- agent
- also
- attribute
- be_in_state
- causes
- classified_by
- classifies
- co_agent_instrument
- co_agent_patient
- co_agent_result
- co_instrument_agent
- co_instrument_patient
- co_instrument_result
- co_patient_agent
- co_patient_instrument
- co_result_agent
- co_result_instrument
- co_role
- direction
- domain_region
- domain_topic
- exemplifies
- entails
- eq_synonym
- has_domain_region
- has_domain_topic
- is_exemplified_by
- holo_location
- holo_member
- holo_part
- holo_portion
- holo_substance
- holonym
- hypernym

- hyponym
- in_manner
- instance_hypernym
- instance_hyponym
- instrument
- involved
- involved_agent
- involved_direction
- involved_instrument
- involved_location
- involved_patient
- involved_result
- involved_source_direction
- involved_target_direction
- is_caused_by
- is_entailed_by
- location
- manner_of
- mero_location
- mero_member
- mero_part
- mero_portion
- mero_substance
- meronym
- similar
- other
- patient
- restricted_by
- restricts
- result
- role
- source_direction
- state_of
- target_direction
- subevent
- is_subevent_of

- antonym
- feminine
- has_feminine
- masculine
- has_masculine
- young
- has_young
- diminutive
- has_diminutive
- augmentative
- has_augmentative
- anto_gradable
- anto_simple
- anto_converse
- ir_synonym

3.12.2 Sense Relations

wn.constants.**SENSE_RELATIONS**

- antonym
- also
- participle
- pertainym
- derivation
- domain_topic
- has_domain_topic
- domain_region
- has_domain_region
- exemplifies
- is_exemplified_by
- similar
- other
- feminine
- has_feminine
- masculine
- has_masculine
- young

- has_young
- diminutive
- has_diminutive
- augmentative
- has_augmentative
- anto_gradable
- anto_simple
- anto_converse
- simple_aspect_ip
- secondary_aspect_ip
- simple_aspect_pi
- secondary_aspect_pi

wn.constants.**SENSE_SYNSET_RELATIONS**

- domain_topic
- domain_region
- exemplifies
- other

wn.constants.**REVERSE_RELATIONS**

```
{
    'hypernym': 'hyponym',
    'hyponym': 'hypernym',
    'instance_hypernym': 'instance_hyponym',
    'instance_hyponym': 'instance_hypernym',
    'antonym': 'antonym',
    'eq_synonym': 'eq_synonym',
    'similar': 'similar',
    'meronym': 'holonym',
    'holonym': 'meronym',
    'mero_location': 'holo_location',
    'holo_location': 'mero_location',
    'mero_member': 'holo_member',
    'holo_member': 'mero_member',
    'mero_part': 'holo_part',
    'holo_part': 'mero_part',
    'mero_portion': 'holo_portion',
    'holo_portion': 'mero_portion',
    'mero_substance': 'holo_substance',
    'holo_substance': 'mero_substance',
    'also': 'also',
    'state_of': 'be_in_state',
    'be_in_state': 'state_of',
    'causes': 'is_caused_by',
    'is_caused_by': 'causes',
```

(continues on next page)

(continued from previous page)

```
    'subevent': 'is_subevent_of',
    'is_subevent_of': 'subevent',
    'manner_of': 'in_manner',
    'in_manner': 'manner_of',
    'attribute': 'attribute',
    'restricts': 'restricted_by',
    'restricted_by': 'restricts',
    'classifies': 'classified_by',
    'classified_by': 'classifies',
    'entails': 'is_entailed_by',
    'is_entailed_by': 'entails',
    'domain_topic': 'has_domain_topic',
    'has_domain_topic': 'domain_topic',
    'domain_region': 'has_domain_region',
    'has_domain_region': 'domain_region',
    'exemplifies': 'is_exemplified_by',
    'is_exemplified_by': 'exemplifies',
    'role': 'involved',
    'involved': 'role',
    'agent': 'involved_agent',
    'involved_agent': 'agent',
    'patient': 'involved_patient',
    'involved_patient': 'patient',
    'result': 'involved_result',
    'involved_result': 'result',
    'instrument': 'involved_instrument',
    'involved_instrument': 'instrument',
    'location': 'involved_location',
    'involved_location': 'location',
    'direction': 'involved_direction',
    'involved_direction': 'direction',
    'target_direction': 'involved_target_direction',
    'involved_target_direction': 'target_direction',
    'source_direction': 'involved_source_direction',
    'involved_source_direction': 'source_direction',
    'co_role': 'co_role',
    'co_agent_patient': 'co_patient_agent',
    'co_patient_agent': 'co_agent_patient',
    'co_agent_instrument': 'co_instrument_agent',
    'co_instrument_agent': 'co_agent_instrument',
    'co_agent_result': 'co_result_agent',
    'co_result_agent': 'co_agent_result',
    'co_patient_instrument': 'co_instrument_patient',
    'co_instrument_patient': 'co_patient_instrument',
    'co_result_instrument': 'co_instrument_result',
    'co_instrument_result': 'co_result_instrument',
    'pertainym': 'pertainym',
    'derivation': 'derivation',
    'simple_aspect_ip': 'simple_aspect_pi',
    'simple_aspect_pi': 'simple_aspect_ip',
    'secondary_aspect_ip': 'secondary_aspect_pi',
    'secondary_aspect_pi': 'secondary_aspect_ip',
```

(continues on next page)

(continued from previous page)

```
    'feminine': 'has_feminine',
    'has_feminine': 'feminine',
    'masculine': 'has_masculine',
    'has_masculine': 'masculine',
    'young': 'has_young',
    'has_young': 'young',
    'diminutive': 'has_diminutive',
    'has_diminutive': 'diminutive',
    'augmentative': 'has_augmentative',
    'has_augmentative': 'augmentative',
    'anto_gradable': 'anto_gradable',
    'anto_simple': 'anto_simple',
    'anto_converse': 'anto_converse',
    'ir_synonym': 'ir_synonym',
}
```

3.12.3 Parts of Speech

wn.constants.**PARTS_OF_SPEECH**

- `n` – Noun
- `v` – Verb
- `a` – Adjective
- `r` – Adverb
- `s` – Adjective Satellite
- `t` – Phrase
- `c` – Conjunction
- `p` – Adposition
- `x` – Other
- `u` – Unknown

wn.constants.**NOUN = 'n'**

wn.constants.**VERB = 'v'**

wn.constants.**ADJECTIVE = 'a'**

wn.constants.**ADJ**

Alias of *ADJECTIVE*

wn.constants.**ADJECTIVE_SATELLITE = 's'**

wn.constants.**ADJ_SAT**

Alias of *ADJECTIVE_SATELLITE*

wn.constants.**PHRASE = 't'**

wn.constants.**CONJUNCTION = 'c'**

wn.constants.**CONJ**

Alias of *CONJUNCTION*

wn.constants.**ADPOSITION = 'p'**

wn.constants.**ADP = 'p'**

Alias of *ADPOSITION*

wn.constants.**OTHER = 'x'**

wn.constants.**UNKNOWN = 'u'**

3.12.4 Adjective Positions

wn.constants.**ADJPOSITIONS**

- a – Attributive
- ip – Immediate Postnominal
- p – Predicative

3.12.5 Lexicographer Files

wn.constants.**LEXICOGRAPHER_FILES**

```
{
    'adj.all': 0,
    'adj.pert': 1,
    'adv.all': 2,
    'noun.Tops': 3,
    'noun.act': 4,
    'noun.animal': 5,
    'noun.artifact': 6,
    'noun.attribute': 7,
    'noun.body': 8,
    'noun.cognition': 9,
    'noun.communication': 10,
    'noun.event': 11,
    'noun.feeling': 12,
    'noun.food': 13,
    'noun.group': 14,
    'noun.location': 15,
    'noun.motive': 16,
    'noun.object': 17,
    'noun.person': 18,
    'noun.phenomenon': 19,
    'noun.plant': 20,
    'noun.possession': 21,
    'noun.process': 22,
    'noun.quantity': 23,
    'noun.relation': 24,
    'noun.shape': 25,
    'noun.state': 26,
    'noun.substance': 27,
```

(continues on next page)

(continued from previous page)

```
    'noun.time': 28,
    'verb.body': 29,
    'verb.change': 30,
    'verb.cognition': 31,
    'verb.communication': 32,
    'verb.competition': 33,
    'verb.consumption': 34,
    'verb.contact': 35,
    'verb.creation': 36,
    'verb.emotion': 37,
    'verb.motion': 38,
    'verb.perception': 39,
    'verb.possession': 40,
    'verb.social': 41,
    'verb.stative': 42,
    'verb.weather': 43,
    'adj.ppl': 44,
}
```

3.13 wn.ic

Information Content is a corpus-based metrics of synset or sense specificity.

The mathematical formulae for information content are defined in *Formal Description*, and the corresponding Python API function are described in *Calculating Information Content*. These functions require information content weights obtained either by *computing them from a corpus*, or by *loading pre-computed weights from a file*.

Note

The term *information content* can be ambiguous. It often, and most accurately, refers to the result of the `information_content()` function ($\mathrm{IC}(c)$ in the mathematical notation), but is also sometimes used to refer to the corpus frequencies/weights ($\mathrm{freq}(c)$ in the mathematical notation) returned by `load()` or `compute()`, as these weights are the basis of the value computed by `information_content()`. The Wn documentation tries to consistently refer to former as the *information content value*, or just *information content*, and the latter as *information content weights*, or *weights*.

3.13.1 Formal Description

The Information Content (IC) of a concept (synset) is a measure of its specificity computed from the wordnet's taxonomy structure and corpus frequencies. It is defined by Resnik 1995 ([RES95]), following information theory, as the negative log-probability of a concept:

$$\mathrm{IC}(c) = -\log p(c)$$

A concept's probability is the empirical probability over a corpus:

$$p(c) = \frac{\mathrm{freq}(c)}{N}$$

Here, N is the total count of words of the same category as concept c ([RES95] only considered nouns) where each word has some representation in the wordnet, and freq is defined as the sum of corpus counts of words in $\mathrm{words}(c)$,

which is the set of words subsumed by concept c:

$$\text{freq}(c) = \sum_{w \in \text{words}(c)} \text{count}(w)$$

It is common for freq to not contain actual frequencies but instead weights distributed evenly among the synsets for a word. These weights are calculated as the word frequency divided by the number of synsets for the word:

$$\text{freq}_{\text{distributed}}(c) = \sum_{w \in \text{words}(c)} \frac{\text{count}(w)}{|\text{synsets}(w)|}$$

3.13.2 Example

In the Princeton WordNet 3.0 (hereafter *WordNet*, but note that the equivalent lexicon in Wn is the *OMW English Wordnet based on WordNet 3.0* with specifier `omw-en:1.4`), the frequency of a concept like **stone fruit** is not just the number of occurrences of *stone fruit*, but also includes the counts of the words for its hyponyms (*almond*, *olive*, etc.) and other taxonomic descendants (*Jordan almond*, *green olive*, etc.). The word *almond* has two synsets: one for the fruit or nut, another for the plant. Thus, if the word *almond* is encountered n times in a corpus, then the weight (either the frequency n or distributed weight $\frac{n}{2}$) is added to the total weights for both synsets and to those of their ancestors, but not for descendant synsets, such as for **Jordan almond**. The fruit/nut synset of almond has two hypernym paths which converge on **fruit**:

1. **almond stone fruit fruit**
2. **almond nut seed fruit**

The weight is added to each ancestor (**stone fruit**, **nut**, **seed**, **fruit**, ...) once. That is, the weight is not added to the convergent ancestor for **fruit** twice, but only once.

3.13.3 Calculating Information Content

`wn.ic.information_content`(*synset:* Synset, *freq:* *dict[str, dict[str | None, float]]*) → float

Calculate the Information Content value for a synset.

The information content of a synset is the negative log of the synset probability (see *synset_probability()*).

`wn.ic.synset_probability`(*synset:* Synset, *freq:* *dict[str, dict[str | None, float]]*) → float

Calculate the synset probability.

The synset probability is defined as freq(ss)/N where freq(ss) is the IC weight for the synset and N is the total IC weight for all synsets with the same part of speech.

Note: this function is not generally used directly, but indirectly through *information_content()*.

3.13.4 Computing Corpus Weights

If pre-computed weights are not available for a wordnet or for some domain, they can be computed given a corpus and a wordnet.

The corpus is an iterable of words. For large corpora it may help to use a generator for this iterable, but the entire vocabulary (i.e., unique words and counts) will be held at once in memory. Multi-word expressions are also possible if they exist in the wordnet. For instance, WordNet has *stone fruit*, with a single space delimiting the words, as an entry.

The *wn.Wordnet* object must be instantiated with a single lexicon, although it may have expand-lexicons for relation traversal. For best results, the wordnet should use a lemmatizer to help it deal with inflected wordforms from running text.

wn.ic.**compute**(*corpus: Iterable[str]*, *wordnet:* Wordnet, *distribute_weight: bool = True*, *smoothing: float = 1.0*) → dict[str, dict[str | None, float]]

Compute Information Content weights from a corpus.

Parameters

- **corpus** – An iterable of string tokens. This is a flat list of words and the order does not matter. Tokens may be single words or multiple words separated by a space.
- **wordnet** – An instantiated *wn.Wordnet* object, used to look up synsets from words.
- **distribute_weight** – If **True**, the counts for a word are divided evenly among all synsets for the word.
- **smoothing** – The initial value given to each synset.

Example

```
>>> import wn, wn.ic, wn.morphy
>>> ewn = wn.Wordnet("ewn:2020", lemmatizer=wn.morphy.morphy)
>>> freq = wn.ic.compute(["Dogs", "run", ".", "Cats", "sleep", "."], ewn)
>>> dog = ewn.synsets("dog", pos="n")[0]
>>> cat = ewn.synsets("cat", pos="n")[0]
>>> frog = ewn.synsets("frog", pos="n")[0]
>>> freq["n"][dog.id]
1.125
>>> freq["n"][cat.id]
1.1
>>> freq["n"][frog.id]  # no occurrence; smoothing value only
1.0
>>> carnivore = dog.lowest_common_hypernyms(cat)[0]
>>> freq["n"][carnivore.id]
1.3250000000000002
```

3.13.5 Reading Pre-computed Information Content Files

The *load()* function reads pre-computed information content weights files as used by the WordNet::Similarity Perl module or the NLTK Python package. These files are computed for a specific version of a wordnet using the synset offsets from the WNDB format, which Wn does not use. These offsets therefore must be converted into an identifier that matches those used by the wordnet. By default, *load()* uses the lexicon identifier from its *wordnet* argument with synset offsets (padded with 0s to make 8 digits) and parts-of-speech from the weights file to format an identifier, such as `omw-en-00001174-n`. For wordnets that use a different identifier scheme, the *get_synset_id* parameter of *load()* can be given a callable created with *wn.util.synset_id_formatter()*. It can also be given another callable with the same signature as shown below:

```
get_synset_id(*, offset: int, pos: str) -> str
```

When loading pre-computed information content files, it is recommended to use the ones with smoothing (i.e., `*-add1.dat` or `*-resnik-add1.dat`) to avoid math domain errors when computing the information content value.

Warning

The weights files are only valid for the version of wordnet for which they were created. Files created for WordNet 3.0 do not work for WordNet 3.1 because the offsets used in its identifiers are different, although the *get_synset_id* parameter of *load()* could be given a function that performs a suitable mapping. Some Open Multilingual Wordnet

wordnets use the WordNet 3.0 offsets in their identifiers and can therefore technically use the weights, but this usage is discouraged because the distributional properties of text in another language and the structure of the other wordnet will not be compatible with that of the English WordNet. For these cases, it is recommended to compute new weights using *compute()*.

wn.ic.**load**(*source: str | Path*, *wordnet:* Wordnet, *get_synset_id: Callable | None = None*) → dict[str, dict[str | None, float]]

Load an Information Content mapping from a file.

Parameters

- **source** – A path to an information content weights file.
- **wordnet** – A *wn.Wordnet* instance with synset identifiers matching the offsets in the weights file.
- **get_synset_id** – A callable that takes a synset offset and part of speech and returns a synset ID valid in *wordnet*.

Raises

wn.Error – If *wordnet* does not have exactly one lexicon.

Example

```
>>> import wn, wn.ic
>>> pwn = wn.Wordnet("pwn:3.0")
>>> path = "~/nltk_data/corpora/wordnet_ic/ic-brown-resnik-add1.dat"
>>> freq = wn.ic.load(path, pwn)
```

3.14 wn.ili

Interlingual Indices

This module provides classes and functions for inspecting Interlingual Index (ILI) objects, both existing and proposed and including their definitions and any metadata, for synsets and lexicons.

Note

See *Interlingual Queries* for background and usage information about ILIs.

3.14.1 Functions for Getting ILI Objects

The following functions are for getting individual *ILI* and *ProposedILI* objects from ILI identifiers or synsets, respectively, or to list all such known objects.

wn.ili.**get**(*id: str*) → *ILI* | None

Get the ILI object with the given id.

The *id* argument is a string ILI identifier. If *id* does not match a known ILI, **None** is returned. Note that a **None** value does not necessarily mean that there is no such ILI, but rather that no resource declaring that ILI has been loaded into Wn's database.

Example:

```
>>> from wn import ili
>>> ili.get("i12345")
ILI('i12345')
>>> ili.get("i0") is None
True
```

wn.ili.**get_all**(*, *status:* ILIStatus | *str* | *None* = *None*, *lexicon:* *str* | *None* = *None*) → list[*ILI*]

Get the list of all matching ILI objects.

The *status* argument may be a string matching a single *ILIStatus*, or a union of one or more *ILIStatus* values. The *lexicon* argument is a space-separated string of lexicon specifiers. All ILIs with a matching status and lexicon will be returned.

Example:

```
>>> from wn import ili
>>> len(ili.get_all())
117442
```

wn.ili.**get_proposed**(*synset:* Synset) → *ProposedILI* | None

Get a proposed ILI for *synset* if it exists.

The synset itself does not give a good indication if it has an associated proposed ILI. The *wn.Synset.ili* value will be **None**, but this is also true if there is no ILI at all. In most cases it is easier to list the proposed ILIs for a lexicon using *get_all_proposed()*, then to retrieve their associated synsets.

Example:

```
>>> import wn
>>> from wn import ili
>>> en = wn.Wordnet("oewn:2024")
>>> en.synset("oewn-00002935-r").ili is None
True
>>> ili.get_proposed(en.synset("oewn-00002935-r"))
ProposedILI(_synset='oewn-00002935-r', _lexicon='oewn:2024')
```

wn.ili.**get_all_proposed**(*lexicon:* *str* | *None* = *None*) → list[*ProposedILI*]

Get the list of all proposed ILI objects.

The *lexicon* argument is a space-separated string of lexicon specifiers. Proposed ILIs matching the lexicon will be returned.

Example:

```
>>> from wn import ili
>>> proposed = ili.get_all_proposed("oewn:2024")
>>> proposed[0]
ProposedILI(_synset='oewn-00002935-r', _lexicon='oewn:2024')
>>> proposed[0].synset()
Synset('oewn-00002935-r')
```

3.14.2 ILI Status

The status of an ILI object (*ILI.status* or *ProposedILI.status*) indicates what is known about its validity. Explicit information about ILIs can be added to Wn with *wn.add()* (e.g., `wn.add("cili")`), but without it Wn can only make a guess.

If a lexicon has synsets referencing some ILI identifier and no ILI file has been loaded, that ILI would have a status of *ILIStatus.PRESUPPOSED*. If an ILI file has been loaded that lists the identifier, it would have a status of *ILIStatus.ACTIVE*, whether or not a lexicon has been added that uses the ILI. Both of these cases use *ILI* objects.

A synset in the WN-LMF format may also propose a new ILI. It won't have an identifier, but it should have a definition. These have the status of *ILIStatus.PROPOSED*. The *ProposedILI* is used for these objects, and that is the only status they have.

The *ILIStatus.UNKNOWN* status is just a default (e.g., when manually creating an *ILI* object) and won't be encountered in normal scenarios.

class wn.ili.**ILIStatus**(**values*)

UNKNOWN = 'unknown'

ACTIVE = 'active'

PRESUPPOSED = 'presupposed'

PROPOSED = 'proposed'

3.14.3 ILI Classes

class wn.ili.**ILI**(*id: str = <property object>*, *status: ILIStatus = ILIStatus.UNKNOWN*, *_definition_text: str | None = None*, *_definition_metadata: Metadata | None = None*)

A class for interlingual indices.

id: str

The ILI identifier.

status: *ILIStatus*

The status of the ILI.

definition(*, *data: bool = False*) → str | *ILIDefinition* | None

Return the ILI's definition.

If the *data* argument is **False** (the default), the definition is returned as a `str` type. If it is **True**, a *wn.ILIDefinition* object is used instead.

Note that *ILI* objects will not have definitions unless an ILI resource has been added, but *ProposedILI* objects will have definitions if one is provided by the proposing lexicon.

class wn.ili.**ProposedILI**(*_synset: 'str'*, *_lexicon: 'str'*, *_definition_text: 'str | None' = None*, *_definition_metadata: 'Metadata | None' = None*)

property id: Literal[None]

Always return **None**.

Proposed ILIs do not have identifiers. This method is kept for interface consistency.

property status: Literal[ILIStatus.PROPOSED]

Always return *ILIStatus.PROPOSED*.

Proposed ILI objects are only used for ILIs that are proposed.

definition(*, *data: bool = False*) → str | *ILIDefinition* | None

Return the ILI's definition.

If the *data* argument is **False** (the default), the definition is returned as a `str` type. If it is **True**, a *wn.ILIDefinition* object is used instead.

Note that *ILI* objects will not have definitions unless an ILI resource has been added, but *ProposedILI* objects will have definitions if one is provided by the proposing lexicon.

synset() → *Synset*

Return the synset object associated with the proposed ILI.

lexicon() → *Lexicon*

Return the lexicon containing the element.

3.14.4 ILI Definitions

Most likely someone inspecting the definition of an *ILI* or *ProposedILI* only cares about the definition text, but for completeness' sake the *ILIDefinition* object models the text along with any metadata that may have appeared in the WN-LMF lexicon file. ILI files do not currently model metadata.

class wn.ili.**ILIDefinition**(*text: str*, *_metadata: Metadata | None = None*, *_lexicon: str | None = None*)

Class for modeling ILI definitions.

text: **str**

metadata() → Metadata

Return the ILI's metadata.

3.15 wn.lmf

Reader for the Lexical Markup Framework (LMF) format.

wn.lmf.**load**(*source: str | ~pathlib.Path*, *progress_handler: type[~wn.util.ProgressHandler] | None = <class 'wn.util.ProgressBar'>*) → LexicalResource

Load wordnets encoded in the WN-LMF format.

Parameters
source – path to a WN-LMF file

wn.lmf.**scan_lexicons**(*source: str | Path*) → list[ScanInfo]

Scan *source* and return only the top-level lexicon info.

The returned info is a dictionary containing the *id*, *version*, and *label* attributes from a lexicon. If the Lexicon is an extension, an *extends* key maps to a dictionary with the *id* and *version* of the base lexicon, otherwise it maps to **None**.

wn.lmf.**is_lmf**(*source: str | Path*) → bool

Return True if *source* is a WN-LMF file.

3.16 wn.morphy

A simple English lemmatizer that finds and removes known suffixes.

See also

The Princeton WordNet documentation describes the original implementation of Morphy.

The *Lemmatization and Normalization* guide describes how Wn handles lemmatization in general.

3.16.1 Initialized and Uninitialized Morphy

There are two ways of using Morphy in Wn: initialized and uninitialized.

Unintialized Morphy is a simple callable that returns lemma *candidates* for some given wordform. That is, the results might not be valid lemmas, but this is not a problem in practice because subsequent queries against the database will filter out the invalid ones. This callable is obtained by creating a *Morphy* object with no arguments:

```
>>> from wn import morphy
>>> m = morphy.Morphy()
```

As an uninitialized Morphy cannot predict which lemmas in the result are valid, it always returns the original form and any transformations it can find for each part of speech:

```
>>> m('lemmata', pos='n')  # exceptional form
{'n': {'lemmata'}}
>>> m('lemmas', pos='n')   # regular morphology with part-of-speech
{'n': {'lemma', 'lemmas'}}
>>> m('lemmas')            # regular morphology for any part-of-speech
{None: {'lemmas'}, 'n': {'lemma'}, 'v': {'lemma'}}
>>> m('wolves')            # invalid forms may be returned
{None: {'wolves'}, 'n': {'wolf', 'wolve'}, 'v': {'wolve', 'wolv'}}
```

This lemmatizer can also be used with a *wn.Wordnet* object to expand queries:

```
>>> import wn
>>> ewn = wn.Wordnet('ewn:2020')
>>> ewn.words('lemmas')
[]
>>> ewn = wn.Wordnet('ewn:2020', lemmatizer=morphy.Morphy())
>>> ewn.words('lemmas')
[Word('ewn-lemma-n')]
```

An initialized Morphy is created with a *wn.Wordnet* object as its argument. It then uses the wordnet to build lists of valid lemmas and exceptional forms (this takes a few seconds). Once this is done, it will only return lemmas it knows about:

```
>>> ewn = wn.Wordnet('ewn:2020')
>>> m = morphy.Morphy(ewn)
>>> m('lemmata', pos='n')  # exceptional form
{'n': {'lemma'}}
>>> m('lemmas', pos='n')   # regular morphology with part-of-speech
{'n': {'lemma'}}
>>> m('lemmas')            # regular morphology for any part-of-speech
{'n': {'lemma'}}
>>> m('wolves')            # invalid forms are pre-filtered
{'n': {'wolf'}}
```

In order to use an initialized Morphy lemmatizer with a *wn.Wordnet* object, it must be assigned to the object after creation:

```
>>> ewn = wn.Wordnet('ewn:2020')  # default: lemmatizer=None
>>> ewn.words('lemmas')
[]
>>> ewn.lemmatizer = morphy.Morphy(ewn)
```

(continues on next page)

(continued from previous page)

```
>>> ewn.words('lemmas')
[Word('ewn-lemma-n')]
```

There is little to no difference in the results obtained from a *wn.Wordnet* object using an initialized or uninitialized *Morphy* object, but there may be slightly different performance profiles for future queries.

3.16.2 Default Morphy Lemmatizer

As a convenience, an uninitialized Morphy lemmatizer is provided in this module via the *morphy* member.

wn.morphy.**morphy**

A *Morphy* object created without a *wn.Wordnet* object.

3.16.3 The Morphy Class

class wn.morphy.**Morphy**(*wordnet:* Wordnet | *None* = *None*)

The Morphy lemmatizer class.

Objects of this class are callables that take a wordform and an optional part of speech and return a dictionary mapping parts of speech to lemmas. If objects of this class are not created with a *wn.Wordnet* object, the returned lemmas may be invalid.

Parameters
wordnet – optional *wn.Wordnet* instance

Example

```
>>> import wn
>>> from wn.morphy import Morphy
>>> ewn = wn.Wordnet("ewn:2020")
>>> m = Morphy(ewn)
>>> m("axes", pos="n")
{'n': {'axe', 'ax', 'axis'}}
>>> m("geese", pos="n")
{'n': {'goose'}}
>>> m("gooses")
{'n': {'goose'}, 'v': {'goose'}}
>>> m("goosing")
{'v': {'goose'}}
```

3.17 wn.project

Wordnet and ILI Packages and Collections

wn.project.**get_project**(*, *project:* *str* | *None* = *None*, *path:* *str* | *Path* | *None* = *None*) → *Project*

Return the *Project* object for *project* or *path*.

The *project* argument is a project specifier and will look in the download cache for the project data. If the project has not been downloaded and cached, an error will be raised.

The *path* argument looks for project data at the given path. It can point to a resource file, a package directory, or a collection directory. Unlike *iterpackages()*, this function does not iterate over packages within a collection, and instead the *Collection* object is returned.

> **Note**
>
> If the target is compressed or archived, the data will be extracted to a temporary directory. It is the user's responsibility to delete this temporary directory, which is indicated by *Project.path*.

wn.project.**iterpackages**(*path: str | Path*, *delete: bool = True*) → Iterator[*Package*]

Yield any wordnet or ILI packages found at *path*.

The *path* argument can point to one of the following:

- a lexical resource file or ILI file
- a wordnet package directory
- a wordnet collection directory
- a tar archive containing one of the above
- a compressed (gzip or lzma) resource file or tar archive

The *delete* argument determines whether any created temporary directories will be deleted after iteration is complete. When it is `True`, the package objects can only be inspected during iteration. If one needs persistent objects (e.g., `pkgs = list(iterpackages(...))`), then set *delete* to `False`.

> **Warning**
>
> When *delete* is set to `False`, the user is responsible for cleaning up any temporary directories. The *Project.path* attribute indicates the path of the temporary directory.

wn.project.**is_package_directory**(*path: str | Path*) → bool

Return `True` if *path* appears to be a wordnet or ILI package.

wn.project.**is_collection_directory**(*path: str | Path*) → bool

Return `True` if *path* appears to be a wordnet collection.

3.17.1 Project Classes

Projects can be simple resource files, *Package* directories, or *Collection* directories. For API consistency, resource files are modeled as a virtual package (*ResourceOnlyPackage*).

class wn.project.**Project**

The base class for packages and collections.

This class is not used directly, but all subclasses will implement the methods listed here.

property path: `Path`

The path of the project directory or resource file.

For *Package* and *Collection* objects, the path is its directory. For *ResourceOnlyPackage* objects, the path is the same as from *resource_file()*

readme() → Path | None

Return the path of the README file, or `None` if none exists.

license() → Path | None

Return the path of the license, or `None` if none exists.

citation() → Path | None

Return the path of the citation, or None if none exists.

class wn.project.**Package**(*path: str | Path*)

Bases: *Project*

A wordnet or ILI package.

A package is a directory with a resource file and optional metadata files.

property type: str | None

Return the name of the type of resource contained by the package.

Valid return values are: - "wordnet" – the resource is a WN-LMF lexicon file - "ili" – the resource is an interlingual index file - None – the resource type is undetermined

resource_file() → Path

Return the path of the package's resource file.

class wn.project.**ResourceOnlyPackage**(*path: str | Path*)

Bases: *Package*

A virtual package for a single-file resource.

This class is for resource files that are not distributed in a package directory. The *readme()*, *license()*, and *citation()* methods all return None.

class wn.project.**Collection**(*path: str | Path*)

Bases: *Project*

A wordnet or ILI collection

Collections are directories that contain package directories and optional metadata files.

packages() → list[*Package*]

Return the list of packages in the collection.

3.18 wn.similarity

Synset similarity metrics.

3.18.1 Taxonomy-based Metrics

The *Path*, *Leacock-Chodorow*, and *Wu-Palmer* similarity metrics work by finding path distances in the hypernym/hyponym taxonomy. As such, they are most useful when the synsets are, in fact, arranged in a taxonomy. For the Princeton WordNet and derivative wordnets, such as the Open English Wordnet and OMW English Wordnet based on WordNet 3.0 available to Wn, synsets for nouns and verbs are arranged taxonomically: the nouns mostly form a single structure with a single root while verbs form many smaller structures with many roots. Synsets for the other parts of speech do not use hypernym/hyponym relations at all. This situation may be different for other wordnet projects or future versions of the English wordnets.

The similarity metrics tend to fail when the synsets are not connected by some path. When the synsets are in different parts of speech, or even in separate lexicons, this failure is acceptable and expected. But for cases like the verbs in the Princeton WordNet, it might be more useful to pretend that there is some unique root for all verbs so as to create a path connecting any two of them. For this purpose, the *simulate_root* parameter is available on the *path()*, *lch()*, and *wup()* functions, where it is passed on to calls to *wn.Synset.shortest_path()* and *wn.Synset.lowest_common_hypernyms()*. Setting *simulate_root* to **True** can, however, give surprising results if the words are from a different lexicon. Currently, computing similarity for synsets from a different part of speech raises an error.

Path Similarity

When p is the length of the shortest path between two synsets, the path similarity is:

$$\frac{1}{p+1}$$

The similarity score ranges between 0.0 and 1.0, where the higher the score is, the more similar the synsets are. The score is 1.0 when a synset is compared to itself, and 0.0 when there is no path between the two synsets (i.e., the path distance is infinite).

wn.similarity.**path**(*synset1:* Synset, *synset2:* Synset, *simulate_root:* *bool* = *False*) → float

Return the Path similarity of *synset1* and *synset2*.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **simulate_root** – When `True`, a fake root node connects all other roots; default: `False`.

Example

```
>>> import wn
>>> from wn.similarity import path
>>> ewn = wn.Wordnet("ewn:2020")
>>> spatula = ewn.synsets("spatula")[0]
>>> path(spatula, ewn.synsets("pancake")[0])
0.058823529411764705
>>> path(spatula, ewn.synsets("utensil")[0])
0.2
>>> path(spatula, spatula)
1.0
>>> flip = ewn.synsets("flip", pos="v")[0]
>>> turn_over = ewn.synsets("turn over", pos="v")[0]
>>> path(flip, turn_over)
0.0
>>> path(flip, turn_over, simulate_root=True)
0.16666666666666666
```

Leacock-Chodorow Similarity

When p is the length of the shortest path between two synsets and d is the maximum taxonomy depth, the Leacock-Chodorow similarity is:

$$-\log\left(\frac{p+1}{2d}\right)$$

wn.similarity.**lch**(*synset1:* Synset, *synset2:* Synset, *max_depth:* *int*, *simulate_root:* *bool* = *False*) → float

Return the Leacock-Chodorow similarity between *synset1* and *synset2*.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **max_depth** – The taxonomy depth (see *wn.taxonomy.taxonomy_depth()*)

- **simulate_root** – When **True**, a fake root node connects all other roots; default: **False**.

Example

```
>>> import wn, wn.taxonomy
>>> from wn.similarity import lch
>>> ewn = wn.Wordnet("ewn:2020")
>>> n_depth = wn.taxonomy.taxonomy_depth(ewn, "n")
>>> spatula = ewn.synsets("spatula")[0]
>>> lch(spatula, ewn.synsets("pancake")[0], n_depth)
0.8043728156701697
>>> lch(spatula, ewn.synsets("utensil")[0], n_depth)
2.0281482472922856
>>> lch(spatula, spatula, n_depth)
3.6375861597263857
>>> v_depth = taxonomy.taxonomy_depth(ewn, "v")
>>> flip = ewn.synsets("flip", pos="v")[0]
>>> turn_over = ewn.synsets("turn over", pos="v")[0]
>>> lch(flip, turn_over, v_depth, simulate_root=True)
1.3862943611198906
```

Wu-Palmer Similarity

When *LCS* is the lowest common hypernym (also called "least common subsumer") between two synsets, i is the shortest path distance from the first synset to *LCS*, j is the shortest path distance from the second synset to *LCS*, and k is the number of nodes (distance + 1) from *LCS* to the root node, then the Wu-Palmer similarity is:

$$\frac{2k}{i + j + 2k}$$

wn.similarity.**wup**(*synset1: Synset*, *synset2: Synset*, *simulate_root=False*) → float

Return the Wu-Palmer similarity of *synset1* and *synset2*.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **simulate_root** – When **True**, a fake root node connects all other roots; default: **False**.

Raises

wn.Error – When no path connects the *synset1* and *synset2*.

Example

```
>>> import wn
>>> from wn.similarity import wup
>>> ewn = wn.Wordnet("ewn:2020")
>>> spatula = ewn.synsets("spatula")[0]
>>> wup(spatula, ewn.synsets("pancake")[0])
0.2
>>> wup(spatula, ewn.synsets("utensil")[0])
0.8
>>> wup(spatula, spatula)
```

(continues on next page)

(continued from previous page)

```
1.0
>>> flip = ewn.synsets("flip", pos="v")[0]
>>> turn_over = ewn.synsets("turn over", pos="v")[0]
>>> wup(flip, turn_over, simulate_root=True)
0.2857142857142857
```

3.18.2 Information Content-based Metrics

The *Resnik*, *Jiang-Conrath*, and *Lin* similarity metrics work by computing the information content of the synsets and/or that of their lowest common hypernyms. They therefore require information content weights (see `wn.ic`), and the values returned necessarily depend on the weights used.

Resnik Similarity

The Resnik similarity (Resnik 1995) is the maximum information content value of the common subsumers (hypernym ancestors) of the two synsets. Formally it is defined as follows, where c_1 and c_2 are the two synsets being compared.

$$\max_{c \in S(c_1,c_2)} \mathrm{IC}(c)$$

Since a synset's information content is always equal or greater than the information content of its hypernyms, $S(c_1, c_2)$ above is more efficiently computed using the lowest common hypernyms instead of all common hypernyms.

`wn.similarity.res`(*synset1:* Synset, *synset2:* Synset, *ic:* *dict[str, dict[str | None, float]]*) → float

Return the Resnik similarity between *synset1* and *synset2*.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **ic** – Information Content weights.

Example

```
>>> import wn, wn.ic, wn.taxonomy
>>> from wn.similarity import res
>>> pwn = wn.Wordnet("pwn:3.0")
>>> ic = wn.ic.load("~/nltk_data/corpora/wordnet_ic/ic-brown.dat", pwn)
>>> spatula = pwn.synsets("spatula")[0]
>>> res(spatula, pwn.synsets("pancake")[0], ic)
0.8017591149538994
>>> res(spatula, pwn.synsets("utensil")[0], ic)
5.87738923441087
```

Jiang-Conrath Similarity

The Jiang-Conrath similarity metric (Jiang and Conrath, 1997) combines the ideas of the taxonomy-based and information content-based metrics. It is defined as follows, where c_1 and c_2 are the two synsets being compared and c_0 is the lowest common hypernym of the two with the highest information content weight:

$$\frac{1}{\mathrm{IC}(c_1) + \mathrm{IC}(c_2) - 2(\mathrm{IC}(c_0))}$$

This equation is the simplified form given in the paper were several parameterized terms are cancelled out because the full form is not often used in practice.

There are two special cases:

1. If the information content of c_0, c_1, and c_2 are all zero, the metric returns zero. This occurs when both c_1 and c_2 are the root node, but it can also occur if the synsets did not occur in the corpus and the smoothing value was set to zero.
2. Otherwise if $c_1 + c_2 = 2c_0$, the metric returns infinity. This occurs when the two synsets are the same, one is a descendant of the other, etc., such that they have the same frequency as each other and as their lowest common hypernym.

`wn.similarity.jcn`(*synset1:* Synset, *synset2:* Synset, *ic:* *dict[str, dict[str | None, float]]*) → float

Return the Jiang-Conrath similarity of two synsets.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **ic** – Information Content weights.

Example

```
>>> import wn, wn.ic, wn.taxonomy
>>> from wn.similarity import jcn
>>> pwn = wn.Wordnet("pwn:3.0")
>>> ic = wn.ic.load("~/nltk_data/corpora/wordnet_ic/ic-brown.dat", pwn)
>>> spatula = pwn.synsets("spatula")[0]
>>> jcn(spatula, pwn.synsets("pancake")[0], ic)
0.04061799236354239
>>> jcn(spatula, pwn.synsets("utensil")[0], ic)
0.10794048564613007
```

Lin Similarity

Another formulation of information content-based similarity is the Lin metric (Lin 1997), which is defined as follows, where c_1 and c_2 are the two synsets being compared and c_0 is the lowest common hypernym with the highest information content weight:

$$\frac{2(\mathrm{IC}(c_0))}{\mathrm{IC}(c_1) + \mathrm{IC}(c_0)}$$

One special case is if either synset has an information content value of zero, in which case the metric returns zero.

`wn.similarity.lin`(*synset1:* Synset, *synset2:* Synset, *ic:* *dict[str, dict[str | None, float]]*) → float

Return the Lin similarity of two synsets.

Parameters

- **synset1** – The first synset to compare.
- **synset2** – The second synset to compare.
- **ic** – Information Content weights.

Example

```
>>> import wn, wn.ic, wn.taxonomy
>>> from wn.similarity import lin
>>> pwn = wn.Wordnet("pwn:3.0")
>>> ic = wn.ic.load("~/nltk_data/corpora/wordnet_ic/ic-brown.dat", pwn)
>>> spatula = pwn.synsets("spatula")[0]
>>> lin(spatula, pwn.synsets("pancake")[0], ic)
0.061148956278604116
>>> lin(spatula, pwn.synsets("utensil")[0], ic)
0.5592415686750427
```

3.19 wn.taxonomy

Functions for working with hypernym/hyponym taxonomies.

3.19.1 Overview

Among the valid synset relations for wordnets (see *wn.constants.SYNSET_RELATIONS*), those used for describing *is-a* taxonomies are given special treatment and they are generally the most well-developed relations in any wordnet. Typically these are the `hypernym` and `hyponym` relations, which encode *is-a-type-of* relationships (e.g., a *hermit crab* is a type of *decapod*, which is a type of *crustacean*, etc.). They also include `instance_hypernym` and `instance_hyponym`, which encode *is-an-instance-of* relationships (e.g., *Oregon* is an instance of *American state*).

The taxonomy forms a multiply-inheriting hierarchy with the synsets as nodes. In the English wordnets, such as the Princeton WordNet and its derivatives, nearly all nominal synsets form such a hierarchy with single root node, while verbal synsets form many smaller hierarchies without a common root. Other wordnets may have different properties, but as many are based off of the Princeton WordNet, they tend to follow this structure.

Functions to find paths within the taxonomies form the basis of all *wordnet similarity measures*. For instance, the *Leacock-Chodorow Similarity* measure uses both *shortest_path()* and (indirectly) *taxonomy_depth()*.

3.19.2 Wordnet-level Functions

Root and leaf synsets in the taxonomy are those with no ancestors (`hypernym`, `instance_hypernym`, etc.) or hyponyms (`hyponym`, `instance_hyponym`, etc.), respectively.

Finding root and leaf synsets

`wn.taxonomy.`**`roots`**(*wordnet:* Wordnet, *pos: str | None = None*) → list[*Synset*]

Return the list of root synsets in *wordnet*.

Parameters

- **`wordnet`** – The wordnet from which root synsets are found.
- **`pos`** – If given, only return synsets with the specified part of speech.

Example

```
>>> import wn, wn.taxonomy
>>> ewn = wn.Wordnet("ewn:2020")
>>> len(wn.taxonomy.roots(ewn, pos="v"))
573
```

wn.taxonomy.**leaves**(*wordnet:* Wordnet, *pos:* *str* | *None* = *None*) → list[*Synset*]

Return the list of leaf synsets in *wordnet*.

Parameters

- **wordnet** – The wordnet from which leaf synsets are found.
- **pos** – If given, only return synsets with the specified part of speech.

Example

```
>>> import wn, wn.taxonomy
>>> ewn = wn.Wordnet("ewn:2020")
>>> len(wn.taxonomy.leaves(ewn, pos="v"))
10525
```

Computing the taxonomy depth

The taxonomy depth is the maximum depth from a root node to a leaf node within synsets for a particular part of speech.

wn.taxonomy.**taxonomy_depth**(*wordnet:* Wordnet, *pos:* *str*) → int

Return the maximum depth of the taxonomy for the given part of speech.

Parameters

- **wordnet** – The wordnet for which the taxonomy depth will be calculated.
- **pos** – The part of speech for which the taxonomy depth will be calculated.

Example

```
>>> import wn, wn.taxonomy
>>> ewn = wn.Wordnet("ewn:2020")
>>> wn.taxonomy.taxonomy_depth(ewn, "n")
19
```

3.19.3 Synset-level Functions

wn.taxonomy.**hypernym_paths**(*synset:* Synset, *simulate_root:* *bool* = *False*) → list[list[*Synset*]]

Return the list of hypernym paths to a root synset.

Parameters

- **synset** – The starting synset for paths to a root.
- **simulate_root** – If `True`, find the path to a simulated root node.

Example

```
>>> import wn, wn.taxonomy
>>> dog = wn.synsets("dog", pos="n")[0]
>>> for path in wn.taxonomy.hypernym_paths(dog):
...     for i, ss in enumerate(path):
...         print(" " * i, ss, ss.lemmas()[0])
 Synset('pwn-02083346-n') canine
  Synset('pwn-02075296-n') carnivore
```

(continues on next page)

(continued from previous page)

```
  Synset('pwn-01886756-n') eutherian mammal
   Synset('pwn-01861778-n') mammalian
    Synset('pwn-01471682-n') craniate
     Synset('pwn-01466257-n') chordate
      Synset('pwn-00015388-n') animal
       Synset('pwn-00004475-n') organism
        Synset('pwn-00004258-n') animate thing
         Synset('pwn-00003553-n') unit
          Synset('pwn-00002684-n') object
           Synset('pwn-00001930-n') physical entity
            Synset('pwn-00001740-n') entity
Synset('pwn-01317541-n') domesticated animal
 Synset('pwn-00015388-n') animal
  Synset('pwn-00004475-n') organism
   Synset('pwn-00004258-n') animate thing
    Synset('pwn-00003553-n') unit
     Synset('pwn-00002684-n') object
      Synset('pwn-00001930-n') physical entity
       Synset('pwn-00001740-n') entity
```

wn.taxonomy.**min_depth**(*synset:* Synset, *simulate_root:* *bool* = *False*) → int

Return the minimum taxonomy depth of the synset.

Parameters

- **synset** – The starting synset for paths to a root.
- **simulate_root** – If **True**, find the depth to a simulated root node.

Example

```
>>> import wn, wn.taxonomy
>>> dog = wn.synsets("dog", pos="n")[0]
>>> wn.taxonomy.min_depth(dog)
8
```

wn.taxonomy.**max_depth**(*synset:* Synset, *simulate_root:* *bool* = *False*) → int

Return the maximum taxonomy depth of the synset.

Parameters

- **synset** – The starting synset for paths to a root.
- **simulate_root** – If **True**, find the depth to a simulated root node.

Example

```
>>> import wn, wn.taxonomy
>>> dog = wn.synsets("dog", pos="n")[0]
>>> wn.taxonomy.max_depth(dog)
13
```

wn.taxonomy.**shortest_path**(*synset:* Synset, *other:* Synset, *simulate_root:* *bool* = *False*) → list[*Synset*]

Return the shortest path from *synset* to the *other* synset.

Parameters

- **other** – endpoint synset of the path
- **simulate_root** – if **True**, ensure any two synsets are always connected by positing a fake root node

Example

```
>>> import wn, wn.taxonomy
>>> dog = ewn.synsets("dog", pos="n")[0]
>>> squirrel = ewn.synsets("squirrel", pos="n")[0]
>>> for ss in wn.taxonomy.shortest_path(dog, squirrel):
...     print(ss.lemmas())
['canine', 'canid']
['carnivore']
['eutherian mammal', 'placental', 'placental mammal', 'eutherian']
['rodent', 'gnawer']
['squirrel']
```

wn.taxonomy.**common_hypernyms**(*synset:* Synset, *other:* Synset, *simulate_root:* *bool* = *False*) → list[*Synset*]

Return the common hypernyms for the current and *other* synsets.

Parameters

- **other** – synset that is a hyponym of any shared hypernyms
- **simulate_root** – if **True**, ensure any two synsets always share a hypernym by positing a fake root node

Example

```
>>> import wn, wn.taxonomy
>>> dog = ewn.synsets("dog", pos="n")[0]
>>> squirrel = ewn.synsets("squirrel", pos="n")[0]
>>> for ss in wn.taxonomy.common_hypernyms(dog, squirrel):
...     print(ss.lemmas())
['entity']
['physical entity']
['object', 'physical object']
['unit', 'whole']
['animate thing', 'living thing']
['organism', 'being']
['fauna', 'beast', 'animate being', 'brute', 'creature', 'animal']
['chordate']
['craniate', 'vertebrate']
['mammalian', 'mammal']
['eutherian mammal', 'placental', 'placental mammal', 'eutherian']
```

wn.taxonomy.**lowest_common_hypernyms**(*synset:* Synset, *other:* Synset, *simulate_root:* *bool* = *False*) → list[*Synset*]

Return the common hypernyms furthest from the root.

Parameters

- **other** – synset that is a hyponym of any shared hypernyms
- **simulate_root** – if **True**, ensure any two synsets always share a hypernym by positing a fake root node

Example

```
>>> import wn, wn.taxonomy
>>> dog = ewn.synsets("dog", pos="n")[0]
>>> squirrel = ewn.synsets("squirrel", pos="n")[0]
>>> len(wn.taxonomy.lowest_common_hypernyms(dog, squirrel))
1
>>> wn.taxonomy.lowest_common_hypernyms(dog, squirrel)[0].lemmas()
['eutherian mammal', 'placental', 'placental mammal', 'eutherian']
```

3.20 wn.util

Wn utility classes.

wn.util.**synset_id_formatter**(*fmt: str = '{prefix}-{offset:08}-{pos}'*, ***kwargs*) → Callable

Return a function for formatting synset ids.

The *fmt* argument can be customized. It will be formatted using any other keyword arguments given to this function and any given to the resulting function. By default, the format string expects a `prefix` string argument for the namespace (such as a lexicon id), an `offset` integer argument (such as a WNDB offset), and a `pos` string argument.

Parameters

- **fmt** – A Python format string
- ****kwargs** – Keyword arguments for the format string.

Example

```
>>> pwn_synset_id = synset_id_formatter(prefix="pwn")
>>> pwn_synset_id(offset=1174, pos="n")
'pwn-00001174-n'
```

class wn.util.**ProgressHandler**(***, *message: str = ''*, *count: int = 0*, *total: int = 0*, *refresh_interval: int = 0*, *unit: str = ''*, *status: str = ''*, *file: ~typing.TextIO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*)

An interface for updating progress in long-running processes.

Long-running processes in Wn, such as *wn.download()* and *wn.add()*, call to a progress handler object as they go. The default progress handler used by Wn is *ProgressBar*, which updates progress by formatting and printing a textual bar to stderr. The *ProgressHandler* class may be used directly, which does nothing, or users may create their own subclasses for, e.g., updating a GUI or some other handler.

The initialization parameters, except for `file`, are stored in a *kwargs* member and may be updated after the handler is created through the *set()* method. The *update()* method is the primary way a counter is updated. The *flash()* method is sometimes called for simple messages. When the process is complete, the *close()* method is called, optionally with a message.

kwargs

A dictionary storing the updateable parameters for the progress handler. The keys are:

- `message` (`str`) – a generic message or name
- `count` (`int`) – the current progress counter
- `total` (`int`) – the expected final value of the counter

- `unit` (`str`) – the unit of measurement
- `status` (`str`) – the current status of the process

close() → None

Close the progress handler.

This might be useful for closing file handles or cleaning up resources.

flash(*message: str*) → None

Issue a message unrelated to the current counter.

This may be useful for multi-stage processes to indicate the move to a new stage, or to log unexpected situations.

set(**kwargs*) → None

Update progress handler parameters.

Calling this method also runs *update()* with an increment of 0, which causes a refresh of any indicator without changing the counter.

update(*n: int = 1*, *force: bool = False*) → None

Update the counter with the increment value *n*.

This method should update the `count` key of *kwargs* with the increment value *n*. After this, it is expected to update some user-facing progress indicator.

If *force* is **True**, any indicator will be refreshed regardless of the value of the refresh interval.

class wn.util.**ProgressBar**(*, *message: str = ''*, *count: int = 0*, *total: int = 0*, *refresh_interval: int = 0*, *unit: str = ''*, *status: str = ''*, *file: ~typing.TextIO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>*)

A *ProgressHandler* subclass for printing a progress bar.

Example

```
>>> p = ProgressBar(message="Progress: ", total=10, unit=" units")
>>> p.update(3)
Progress: [#########                    ] (3/10 units)
```

See *format()* for a description of how the progress bar is formatted.

FMT = '\r{message}{bar}{counter}{status}'

The default formatting template.

close() → None

Print a newline so the last printed bar remains on screen.

flash(*message: str*) → None

Overwrite the progress bar with *message*.

format() → str

Format and return the progress bar.

The bar is is formatted according to *FMT*, using variables from `kwargs` and two computed variables:

- `bar`: visualization of the progress bar, empty when `total` is 0
- `counter`: display of `count`, `total`, and `units`

```
>>> p = ProgressBar(message="Progress", count=2, total=10, unit="K")
>>> p.format()
'\rProgress [######                          ] (2/10K) '
>>> p = ProgressBar(count=2, status="Counting...")
>>> p.format()
'\r (2) Counting...'
```

update(*n: int = 1*, *force: bool = False*) → None

Increment the count by *n* and print the reformatted bar.

3.21 wn.validate

Wordnet lexicon validation.

This module is for checking whether the the contents of a lexicon are valid according to a series of checks. Those checks are:

Code	Message
E101	ID is not unique within the lexicon.
W201	Lexical entry has no senses.
W202	Redundant sense between lexical entry and synset.
W203	Redundant lexical entry with the same lemma and synset.
E204	Synset of sense is missing.
W301	Synset is empty (not associated with any lexical entries).
W302	ILI is repeated across synsets.
W303	Proposed ILI is missing a definition.
W304	Existing ILI has a spurious definition.
W305	Synset has a blank definition.
W306	Synset has a blank example.
W307	Synset repeats an existing definition.
E401	Relation target is missing or invalid.
W402	Relation type is invalid for the source and target.
W403	Redundant relation between source and target.
W404	Reverse relation is missing.
W501	Synset's part-of-speech is different from its hypernym's.
W502	Relation is a self-loop.

wn.validate.**validate**(*lex: ~wn.lmf.Lexicon | ~wn.lmf.LexiconExtension*, *select: ~collections.abc.Sequence[str] = ('E', 'W')*, *progress_handler: type[~wn.util.ProgressHandler] | None = <class 'wn.util.ProgressBar'>*) → dict[str, _Check]

Check *lex* for validity and return a report of the results.

The *select* argument is a sequence of check codes (e.g., `E101`) or categories (`E` or `W`).

The *progress_handler* parameter takes a subclass of *wn.util.ProgressHandler*. An instance of the class will be created, used, and closed by this function.

BIBLIOGRAPHY

[VOSSEN1998] Piek Vossen. 1998. *Introduction to EuroWordNet.* Computers and the Humanities, 32(2): 73–89.

[Vossen99] Vossen, Piek, Wim Peters, and Julio Gonzalo. "Towards a universal index of meaning." In Proceedings of ACL-99 workshop, Siglex-99, standardizing lexical resources, pp. 81-90. University of Maryland, 1999.

[Bond16] Bond, Francis, Piek Vossen, John Philip McCrae, and Christiane Fellbaum. "CILI: the Collaborative Interlingual Index." In Proceedings of the 8th Global WordNet Conference (GWC), pp. 50-57. 2016.

[RES95] Resnik, Philip. "Using information content to evaluate semantic similarity." In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Canada, pp. 448-453. 1995.

PYTHON MODULE INDEX

W

INDEX

Symbols

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W