Cylleneus Documentation

Release 0.8.1

William Michael Short

Jul 13, 2020

Contents:

CHAPTER 1

Cylleneus

- Free software: Apache Software License 2.0
- Documentation: https://cylleneus.readthedocs.io.

1.1 Overview

Cylleneus is a next-generation search engine for electronic corpora of Greek and Latin, which enables texts to be searched on the basis of their semantic and morpho-syntactic properties. This means that, for the first time, texts can be searched by the *meanings* of words as well as by the kinds of grammatical constructions they occur in. Semantic search takes advantage of the Ancient Greek WordNet, Latin WordNet and Sanskrit WordNet and is fully implemented, and thus is available for any annotated or plain-text corpus. However, semantic queries may still be imprecise due to the on-going nature of these two projects. Syntactic search functionality is still under development and is available for only certain structured corpora. Morphological searching and query filtering will work with any Latin corpus, and any Greek corpus with sufficient morhological annotation.

1.2 Features

- Advanced search capabilities for Greek, Latin, Sanskrit (and soon ancient Egyptian)
- Semantic search: find words based on their meanings in English, Italian, Spanish, or French
- Syntactic search: finds words based on the kinds of grammatical constructions they appear in

- Morphological search: find words, or filter the results of other queries, based on morphological properties
- Fast: once a corpus is indexed, most query types produce results nearly instantaneously
- Sophisticated: query types can be combined into complex search patterns
- Extensible: indexing pipelines can be created for any corpus type
- Currently supports: Atlas, AGLDT, LASLA, Perseus (XML or JSON format), CAMENA, PROIEL and any plaintext source (e.g. Digital Latin Library); Digital Corpus of Sanskrit; Ramses. Diorisis support in development.
- Free: completely open-source and redistributable

1.3 Installation

```
pip install cylleneus
```

1.4 Setup

The Cylleneus engine requires texts to be indexed before they can be searched. For convenience and testing, several pre-indexed mini-corpora are available. These need to be placed in a proper folder hierarchy within the user's data directory.

MacOS `~/Library/Application Support/Cylleneus/corpus`

Windows `C:\Documents and Settings\<User>\Application Data\Local Settings\Cylleneus\corpus` or `C:\Documents and Settings\<User>\AppData\Local\Cylleneus\corpus`

Linux: `~/.local/share/Cylleneus/corpus`

To enable gloss-based searches, Cylleneus relies on the MultiWordNet. The setup process should install the latest version of the `multiwordnet` library, and also compile the necessary databases, but in case this step has been omitted you can do it manually. To do so, launch the Python REPL and enter the following commands.

```
>>> from multiwordnet.db import compile
>>> for language in ['common', 'english', 'latin', 'french', 'spanish', 'italian',
↪'portuguese', 'hebrew']:
...     compile(language)
```

To test that everything is working properly, run the battery of query tests in `tests/test_query_types.py` over the packaged subcorpora.

1.5 Working with corpora

New documentation is available (in the form of an interactive Jupyter Notebook) for integrating new corpus types into Cylleneus.

1.6 Indexing

Ready-made tools are provided for indexing texts from the Perseus Digital Library (in JSON or TEI XML format), the LASLA corpus, the PROIEL corpus, the AGLDT corpus, the PHI5 corpus, and from plain-text sources (for instance, the Latin Library). To index a corpus (or part of one), the raw source should be placed in an appropriately

named directory within `/corpus/<name>/text/`. Then you can use any of the scripts in the `scripts` directory, modifying it for your own needs. The script for indexing texts from the DLL can be adapted to any plain-text source document. If you want to use texts from another corpus entirely, you will need to create an indexing pipeline tailored to the structure of that corpus. See the documentation for instructions.

Basic indexing functionality is also provided through a command-line interface. `$ cylleneus --help` displays the complete list of available indexing commands.

To create the index for a corpus, you will need to have a folder `text` in an appropriately named directory. (For examples of the correct directory structure, see the sample corpora).

```
$ cylleneus create --corpus latin_library # create the 'latin_library' corpus
from scratch using the available texts
```

To add a document or documents to a corpus, you must provide the original source files and indicate the correct path.

```
$ cylleneus index --corpus perseus # display the current index of corpus
'perseus'
```

```
$ cylleneus add --corpus lasla --path "/corpus/lasla/texts/
Catullus_Catullus_Catul.BPN" # for plaintext corpora you will also need to
specify --author and --title, as this cannot be inferred from filenames or
other metadata
```

Indexes should probably always be optimized, though this process can be slow if the corpus is large.

```
$ cylleneus optimize --corpus latin_library
```

1.7 Searching

The best way to search the available corpora (or to import new files individually) is to use the web app or shell script, available separately. Searches can of course also be conducted programmatically via the library's API.

```
>>> from cylleneus.corpus import Corpus
```

```
>>> from cylleneus.search import Searcher, Collection
```

```
>>> corpus = Corpus("perseus")
```

```
>>> clct = Collection(corpus.works)
```

```
>>> searcher = Searcher(clct)
```

```
>>> results = searcher.search("<habeo>")
```

1.8 Query Types

Currently, Cylleneus enables the following types of queries:

1.8.1 Word-form queries

Form '...'

Example 'virtutem'

Description matches a literal string

1.8.2 Lemma-based queries

Form <...>

Example <virtus>

Description matches any form of the specified lemma

More precision can be introduced by using LEMLAT URIs, along with morphological tagging. For example, in the Cylleneus shell `search <dico>` will match occurrences both of *dico*, *dicere* and of *dico*, *dicare*. To distinguish between them, you can use the relevant URIs: `<dico:d1349>` (*dicare*) or `<dico:d1350>`. Alternatively, you can specify an appropriate morphological tag: `<dico=v1spia--3->` or <dico=v1spia–1->‘‘.

1.8.3 Gloss-based queries

Form [...]

Example [en?courage]

Description matches any word with the same meaning as the specified gloss. Can be ‘en’, ‘it’, ‘es’, or ‘fr’.

Example [n#05595229]

Description matches any word with the meaning defined by the specified synset offset ID

1.8.4 Domain-based queries

Form {...}

Example {611}, {Anatomy}

Description matches any word of any part of speech whose meaning falls within the specified domain. Cylleneus uses the Dewey Decimal Classification System as a general topic index.

1.8.5 Morphology-based queries

Form :...

Example :ACC.SG.

Description matches any word with the specified morphological properties, given in Leipzig notation. Annotations can be given as distinct query terms, or can be used as filters for lemma- or gloss-based queries. (For example, `<virtus>:PL.` will match only plural forms of this word).

1.8.6 Morphology-based filtering

Form <...>|...

Example <virtus>|GEN.SG.

Description filters results for only genitive singular forms

Form [...]:...

Example [en?attack]¦VB.PL.

Description filters results for only plural verb forms

Form {...}:...

Example {Anatomy}|ACC.

Description filters results for only accusative forms

1.8.7 Lexical-relation queries

Form <?::...>

Example </::virtus>

Description matches any word with the specified lexical relation to the given lemma

1.8.8 Semantic-relation queries

Form [?::...]

Example [@::en?courage]

Description matches any word with the specified semantic relation to the given gloss

Example [@::n#05595229]

Description matches any word with the specified semantic relation to the given synset

1.8.9 Syntax-based queries

Form /.../

Example /ablative absolute/

Description syntactical constructions (currently, only the LASLA corpus supports this)

Gloss-based searches enable searching by the meanings of words, and queries can be specified in English (en?), Italian (it?), Spanish (es?), or French (fr?). (NB. The vocabulary for Italian, Spanish, and French is significantly smaller than English). It is also possible to search by synset ID number: this capability is exposed for future development of an interface where users can search for a specific sense. Normally, queries will be specified as English terms, which resolve to sets of synsets. Queries involving lexical and semantic relations depend on information available from the Latin Wordnet 2.0. As this project is on-going, rich relational information may be available only for a subset of vocabulary. However, as new information becomes available, search results should become more comprehensive and more accurate.

1.8.10 Types of lexical relations

Code	Description
\	derives from (e.g., <\::femina> would match any lemma derived from *femina*, namely, *femineus*)
/	relates to (the converse of *derives from*)
+c	composed of (e.g., <+c::cum> would match any lemma composed by *cum*)
-c	composes (e.g., <-c::compono> would match lexical elements that compose *compono*, namely, *cum* and *pono*).
<	participle (verbs only)

1.8.11 Types of semantic relations

Code	Description
`!`	antonym of
`@`	hypernym of
`~`	hyponym of
`|`	nearest to
`*`	entails
`#m`	member of
`#p`	part of
`#s`	substance of
`+r`	has role
`%m`	has member
`%p`	has part
`%s`	has substance
`-r`	is role of
`>`	causes
`^`	see also
`$`	verb group
`=`	attribute

Query types can be combined into complex adjacency or proximity searches. An adjacency search specifies a particular ordering of the query terms (typically, but not necessarily, sequential); a proximity search simply finds contexts where all the query terms occur, regardless of order. Adjacency searches must be enclosed with double quotes ("…"), optionally specifying a degree of 'slop', that is, the number of words that may intervene between matched terms, using '~' followed by the number of permissible intervening words.

1.8.12 Examples

`"cui dono"` matches the literal string 'cui dono'

`"si quid <habeo>"` matches 'si' followed by 'quid' followed by any form of *habeo*

`"cum :ABL."` matches 'cum' followed by any word in the ablative causes

`"in <ager>|PL."` matches 'in' followed by any plural form of *ager*

`"<magnus> <animus>"~2` matches any form of *magnus* followed by any form of *animus*, including if separated by a single word

`<honos> <virtus>` matches any context including both any form of *honos* and any form of *virtus*

1.9 To Do

In no particular order…

- Perseus CTS alignment for corpora with non-standard text annotations
- implement high-order syntactic search for different annotation schemes
- manually-curated WordNet-based semantic mark-up ('sembanks') for texts

1.10 Credits

CHAPTER 2

Installation

2.1 Stable release

To install Cylleneus, run this command in your terminal:

```
$ pip install cylleneus
```

This is the preferred method to install Cylleneus, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Cylleneus can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/cylleneus/cylleneus
```

Or download the tarball:

```
$ curl  -OL https://github.com/cylleneus/cylleneus/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Cylleneus in a project:

```
import cylleneus
```

See the included interactive notebooks for detailed examples of usage.

CHAPTER 4

About

4.1 Towards a semantic and syntactic search engine for electronic corpora of Greek and Latin

4.1.1 William Michael Short

The University of Exeter, Department of Classics & Ancient History

Existing applications for browsing and searching electronic corpora of Greek and Latin fall into two broad categories. The first comprises basic textual search tools using 'regular expression' pattern matching. These tools allow users to perform word-form and 'wildcard' searches, including lexical co-occurrence queries. For example, a user can search for the Latin word *lacrimas* near words containing the stem *-fund-* to discover where authors write about 'shedding tears', whether with the verb *profundo*, *effundo*, or simply *fundo*. The second category comprises tools that can query texts tagged with various sorts of metadata, including information about metre, genre and time period, the parts of speech of words, intertextual references, or parallel translations. Both types have made corpus search indispensable for studying Greek and Latin, which are morphologically complex languages with relatively free word order. However, while improving the efficiency, precision, and scope of corpus-based research, they have left a window for new applications that redefine the sorts of questions users are able to ask of ancient texts. The search engine software we are designing will enable its users to explore Greek and Latin texts in new ways by opening these texts to queries based – for the first time – on their semantic as well as syntactic properties. Using our tool, users will be able to search Latin (and, at a later stage of the project, Greek) texts by inputting meanings in English (or another language) without concern for how these meanings are captured lexically in the target corpus. This means that in the above example the user could replace the wildcard *-fund-* with the meaning 'shed' or 'pour', and the search engine would return instances with *profundo*, *effundo*, and *fundo*, but also passages where the semantically similar verbs *mitto* and *mano* appear, which otherwise will be missed. Likewise, by replacing the fixed word-form *lacrimas* with the meaning 'tears' or 'crying', our engine would automatically include additional instances where this notion is represented by *fletus*, *ploratus*, or *delacrimatio*. In principle, searches could be conducted across both corpora simultaneously, without the user needing to constrain her query to specific words in Greek or Latin. Meaning-based search improves upon pattern matching by abstracting away from the lexicon. It enables entirely novel kinds of queries, as well. Consider that the Latin word locus, literally, 'a place', can be used metaphorically in the sense of 'an idea'. The same metaphor can be detected in many other expressions where different kinds of mental operations ('planning', 'agreeing', 'conceiving') are conveyed figuratively by verbs of spatial motion ('entering', 'coming to', 'approaching'). With existing tools,

identifying such patterns of metaphorical structure requires painstakingly searching the corpus for forms of words that signify spatial motion, then culling these results for only those occurrences where the sense must be figurative. By contrast, a tool that permits searching by word sense (i.e., 'an idea') and by semantic field ('mental phenomena') would make identifying linguistic details like this effortless. Even for users with limited knowledge of the ancient languages, this functionality can provide insight into how concepts were understood in historical societies. The kinds of queries enabled by our semantic search engine can also inform literary interpretation. Take Vergil's famous line, *mihi frigidus horror / membra quatit*, 'Cold fear shakes my limbs' (*Aen*. 3.29–30). A simple collocation search suggests this expression is a product of Vergil's poetic imagination: the pairing *frigidus horror* occurs only twice elsewhere in Latin literature, once in a passage of Lucretius that probably served as a model for the Vergilian phrase, and again in a verse of Ovid that was in turn likely influenced by it. Apparent fodder for intertextual studies of Latin epic poetry, or for tracing out relationships of imitation and allusion between Latin authors. Yet a search for words denoting 'fear' in conjunction with words signifying 'cold' or 'shivering' reveals that representations of fear in these terms are in fact quite regular across different authors, genres, and time periods. In other words, what appears to be an instance of creative literary expression may instead be a manifestation of Latin speakers' entirely regular conceptualisation of fear. Whereas traditional methods of corpus search tend to obscure this kind of fact, our software could help distinguish what is creative about texts from what is conventional, and thus highlight where authors are actually engaged in novel meaning-making. Along with meanings, users will also be able to specify bare morphological and syntactic features as part of search parameters, enabling complex queries like 'any adjective in the semantic field of "racial, ethnic, and national groups" when it precedes its noun in the dative case'. We believe this ability to search texts for grammatical configurations independent of any particular lexical instantiation could have a significant impact on the study of ancient languages, which abound in structurally intricate – and very often seemingly functionally equivalent – syntactic constructions. In particular, it could propel the growing field of cognitive classical linguistics, which treats constructions as having meanings in and of themselves, distinct from the meanings of the words of which they are composed, as well as classical linguistics more generally, where focus has recently shifted to the interface of semantics and morpho-syntactic structure. The search engine could also aid teachers and their students by facilitating exemplification of syntactical rules based on the actual usage of ancient authors. We have identified two main and largely separate challenges to implementing semantic and syntactic search. The first will be to implement meaning-based queries. Today's electronic databanks of Latin texts – including those parsed corpora which include morpho-syntactic annotations – contain basically no semantic information. Our solution will be to integrate the data of the Latin WordNet, a lexical knowledge-base for this language created as part of the Fondazione Bruno Kessler's MultiWordNet Project, into existing treebanks. In a WordNet, words or phrases are assigned to one or more 'synsets', corresponding to the different senses they possess. A synset thus represents a grouping of semantically related expressions, or, from a different angle, an atomized meaning-component in the language. The words of languages within the MultiWordNet are keyed to over 100,000 distinct synsets from English and can be assigned language-specific synsets glossed in English. As the Latin WordNet presently contains only a portion of the Latin vocabulary, however, a crucial step in our effort will be to expand it by about 30,000 lemmas, adding morphological information as well as coding a dense network of lexical and semantic relations into the database. In a future stage, the WordNet will be rearchitected to distinguish metonymic and metaphorical from literal senses of words, as well as to be aware of large-scale mappings that structure figurative usage supralexically. Basing our engine's semantic model on the MultiWordNet offers key advantages. By preprocessing and tagging texts within our corpora with each lemma's synsets, searching for meanings becomes highly efficient. As the Latin WordNet will include information about lexical and semantic relations, users will also be able to conduct searches based on, for instance, antonymy or etymological derivation ('the word mitto not in the sense "pour"', 'any word derived from the verb *fari*'). Furthermore, because word senses are defined according to a common pool of synsets, theoretically users will be able to enter queries in any language for which a WordNet is available. Users can also specify semantic fields as search parameters, enabling queries over sets of terms grouped by conceptual domain (e.g., 'economy', 'military', 'agriculture'). Finally, the inclusion of multiple synset assignments within the annotation structure, together with a mechanism of moderated crowd-sourced 'up-voting' whereby users can indicate informed judgments about senses in context, will help avoid predetermining the meaning of ancient texts, and builds into the engine itself an awareness that literary texts are generally open to having various interpretations. The second challenge is to determine the correct syntactic structure of texts and to devise an annotation schema at an appropriate level of linguistic description. Our engine, built on the open-source ANNIS platform, will aggregate curated and verified morpho-syntactic data in the form of treebanks, drawing primarily on the database of rich text annotations in the Greek and Latin Dependency Treebank created by the Perseus Project at Tufts University, with additions from the Index Thomisticus and the PROIEL project providing coverage of different authors, genres, and

time periods. The treebanks in these repositories have undergone manual review and correction of the tagged syntactic information to ensure a high degree of accuracy. However, because they encode linguistic information in different ways and with differing degrees of granularity, our task will be to devise a high-level model that can integrate treebanks based on potentially very different grammars, along with the semantic data of the WordNet.

CHAPTER 5

Cylleneus

5.1 Corpus management

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/cylleneus/cylleneus/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Cylleneus could always use more documentation, whether as part of the official Cylleneus docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/cylleneus/cylleneus/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *cylleneus* for local development.

1. Fork the *cylleneus* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cylleneus.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cylleneus
$ cd cylleneus/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cylleneus tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/cylleneus/cylleneus/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_cylleneus
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 7

Credits

7.1 Development Lead

- William Michael Short <w.short@exeter.ac.uk>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

8.1 0.0.3 (2019-07-30)

- General improvements to corpus management, indexing, searching and CL tools

8.2 0.0.2 (2019-07-15)

- Improvements to indexing and searching

8.3 0.0.1 (2019-06-15)

- Public release on GitHub.

CHAPTER 9

Examples

A simple demonstration of how Cylleneus can power intertextual searches.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search