# lexedata

*Release 1.0.1*

Melvin Steiger, Gereon A. Kaiping

Apr 27, 2022

# LEXEDATA DOCUMENTATION

CHAPTER

ONE

# INTRODUCTION

Lexedata is a set of tools for managing, editing, and annotating large lexical datasets in *cross-linguistic data format (CLDF)*. You can find how commands are organized in lexedata, and a description of all available commands and operations in the manual. For an example using a simple dataset, see the lexedata tour below.

In order to use lexedata you need to be somewhat familiar with the command line and the CLDF format. We also strongly recommend using git for version control, so you can easily access and restore previous versions of your dataset and collaborate with others. If you are not familiar with CLDF, git and the command line, you can find a brief introduction and further references under "Introductions to the Ecosystem". You can also find a glossary with the main terms that are used in the manual and lexedata help.

Lexedata is open access software in development. Please report any problems and suggest any improvements you would like to see by opening an issue on the Lexedata GitHub repository.

## 1.1 Installation

1. In order to install and use Lexedata you need to have Python 3.8 (or newer) installed.

If you are unsure if this is the case, open a terminal window and type `python --version`. If Python 3.7 (or newer) is installed you will see the version. If you don't have any version of Python or it is a version of Python 2, then you need to download and install Python 3. There are different distributions of Python and most of them should work. A popular one that we have tested is Anaconda. Once you have downloaded and installed Anaconda, close and open the terminal again, and type `python --version`. You should see the current version of Python 3 you just downloaded.

If you are ever stuck with the python prompt, which starts with >>>, type `quit()` in order to exit Python.

2. Install the lexedata package.

In your terminal window type `pip install lexedata`. This will install lexedata and all its dependencies on your computer. Now you should be ready to use lexedata!

3. Install CLDF catalogs

Lexedata uses CLDF catalogs, Glottolog for languages, CLTS for phonetic transcription symbols, and Concepticon for concepts, in some of its scripts. You can install them using `cldfbench catconfig`, it will prompt you about the installation process and download and install those catalogs. Make sure to *agree* to cloning the three repositories (`**[y/N]**`_ defaults to "No" for each catalog) and you will end up with local copies of them.

### 1.1.1 Updating lexedata and Catalogs

You can update lexedata when there is a new release by typing `pip install --upgrade lexedata`. This will also update all the packages that lexedata is dependent on. However, the CLDF catalogs, (Glottolog, CLTS, and Concepticon) are not automatically updated. It is good to update those once in a while to get the most up-to-date information, by typing `cldfbench catupdate`.

## 1.2 A tour of lexedata

This tutorial will take you on a tour through the command line functionality of the lexedata package. We will start with a small lexical dataset in an interleaved tabular format, where each column corresponds to a language and each pair of rows to a concept, with preliminary cognate codes. We will take you through turning the dataset into CLDF, editing cognate judgements, and exporting the dataset as phylogenetic alignment.

(To prevent this tutorial from becoming obsolete, our continuous integration testing system 'follows' this tutorial when we update the software. So if it appears overly verbose or rigid to you at times, it's because it has a secondary function as test case. This is also the reason we use the command line where we can, even in places where a GUI tool would be handy: Our continuous integration tester cannot use the GUI.)

```
$ python -m lexedata.importer.excel_interleaved --help
[...]
$ export LANG=C
```

Lexedata is a collection of command line tools. If you have never worked on the command line before, check out *our quick primer on the command line*. This tutorial further assumes you have a working *Installation* of lexedata and *Working with git*. The tutorial will manipulate the Git repository using Git's command line interface, but you can use a Git GUI instead. While it is possible to use lexedata without Git, we do not recommend this, as lexedata does not have an 'undo' function. Git is a version control system and allows you to access and restore previous versions, while it also makes collaboration with others easier.

### 1.2.1 Importing a dataset into CLDF

First, create a new empty directory. We will collect the data and run the analyses inside that folder. Open a command line interface, and make sure its working directory is that new folder. For example, start terminal and execute

```
$ mkdir bantu
$ cd bantu
```

For this tutorial, we will be using lexical data from the Bantu family, collected by Hilde Gunnink. The dataset is a subset of an earlier version (deliberately, so this tour can show some steps in the cleaning process) of her lexical dataset. The data is stored in an Excel file which you can download from https://github.com/Anaphory/lexedata/blob/master/src/lexeadata/data/example-bantu.xlsx in the lexedata repository. (I will use the most recent version here, which comes shipped with lexedata. Sorry this looks a bit cryptic, but as I said, this way the testing system also knows where to find the file.)

```
$ python -c 'import pkg_resources; open("bantu.xlsx", "wb").write(pkg_resources.resource_
↪stream("lexedata", "data/example-bantu.xlsx").read())'
```

(curl is a command line tool to download files from URLs, available under Linux and Windows. You can, of course, download the file yourself using whatever method you are most comfortable with, and save it as `Bantu.xlsx` in this folder.)

If you look at this data (I will do it in Python, but feel free to open it in Excel), you will see that

```
$ python -c 'from openpyxl import load_workbook
> for row in load_workbook("bantu.xlsx").active.iter_rows():
>   row_text = [(c.value or "").strip() for c in row]
>   if any(row_text):
>     print(*row_text, sep="\t")' # doctest: +NORMALIZE_WHITESPACE
        Duala      Ntomba      Ngombe                        Bushoong                [.
↪..]
all     s       (nk)umá     ńsò (Bastin et al 1999)      kim (Bastin et al 1999)    [...]
        1          9           10                            11                      [.
↪..]
arm     dia        lobk       lò-bókò (PL: màbókò) [...]   l (Bastin et al 1999)    [...]
        7          1           1                             1                       [.
↪..]
ashes   mabúdú     metókó      búdùlù ~ pùdùlù ([...])      bu-tók (Bastin et al 1999) [.
↪..]
        17         16          17                            16                      [.
↪..]
[...]
```

it is table with one column for each language, and every pair of rows contains the data for one concept. The first row of each pair contains the forms for the concept (a cell can have multiple forms separated by comma), and the second row contains cognacy judgements for those forms.

This is one of several formats supported by lexedata for import. The corresponding importer is called `excel_interleaved` and it works like this:

```
$ python -m lexedata.importer.excel_interleaved --help
usage: python -m lexedata.importer.excel_interleaved [-h]
                                                     [--sheets SHEET [SHEET ...]]
                                                     [--directory DIRECTORY]
                                                     [--loglevel LOGLEVEL]
                                                     [-q] [-v]
                                                     EXCEL

Import data in the "interleaved" format from an Excel spreadsheet. [...]
[...]

positional arguments:
  EXCEL                 The Excel file to parse

option[...]:
  -h, --help            show this help message and exit
  --sheets SHEET [SHEET ...]
                        Excel sheet name(s) to import (default: all sheets)
  --directory DIRECTORY
                        Path to directory where forms.csv is to be created
                        (default: current working directory)

Logging:
  --loglevel LOGLEVEL
  -q
  -v
```

So this importer needs to be told about the Excel file to import, and it can be told about the destination directory of the import and about sheet names to import, eg. if your Excel file contains additional non-wordlist data in separate worksheets.

Like nearly every lexedata scripts, this one has logging controls to change the verbosity. There are 5 levels of logging: CRITICAL, ERROR, WARNING, INFO, and DEBUG. Normally, scripts operate on the INFO level: They are tell us about anything that might be relevant about the progress and successes. If that's too much output, you can make it -*q*-uieter to only display warnings, which tell us about anthing where the script found data not up to standard and had to fall back to some workaround to proceed. Even less output happens on the ERROR level ("Your data had issues that made me unable to complete the current step, but I can still recover to do *something* more") and the CRITICAL level ("I found something that makes me unable to proceed at all."). We run many of the examples here in quiet mode, you probably don't want to do that.

With that in mind, we can run the interleaved importer simply with the Excel file as argument:

```
$ python -m lexedata.importer.excel_interleaved -q bantu.xlsx
WARNING:lexedata:F48: Multiple forms (ly-aki, ma-ki) did not match single cognateset (1),
↪ using that cognateset for each form.
WARNING:lexedata:H30: Multiple forms (képié, mpfô) did not match single cognateset (9),␣
↪using that cognateset for each form.
WARNING:lexedata:H90: Multiple forms (o-zyâ, o-jib) did not match single cognateset (1),␣
↪using that cognateset for each form.
WARNING:lexedata:H200: Multiple forms (okáàr, mukal) did not match single cognateset (2),
↪ using that cognateset for each form.
WARNING:lexedata:I160: Multiple forms (kk, k) did not match single cognateset (3), using␣
↪that cognateset for each form.
WARNING:lexedata:J144: Multiple forms (mũ-thanga, gĩ-thangathĩ) did not match single␣
↪cognateset (4), using that cognateset for each form.
WARNING:lexedata:Cell N16 was empty, but cognatesets ? were given in N17.
WARNING:lexedata:N28: Multiple forms (igi-ho (cloud, sky), ibi-chu (clouds)) did not␣
↪match single cognateset (2), using that cognateset for each form.
```

This shows a few minor issues in the data, but the import has succeeded, giving us a FormTable in the file `forms.csv`:

```
$ head forms.csv
ID,Language_ID,Parameter_ID,Form,Comment,Cognateset_ID
duala_all,Duala,all,s,,1
duala_arm,Duala,arm,dia,,7
duala_ashes,Duala,ashes,mabúdú,,17
duala_bark,Duala,bark,bwelé,,23
duala_belly,Duala,belly,dibum,,1
duala_big,Duala,big,éndn,,1
duala_bird,Duala,bird,inn,,1
duala_bite,Duala,bite,kukwa,,6
duala_black,Duala,black,wínda,,21
```

A well-structured `forms.csv` is a valid, "metadata-free" CLDF wordlist. In this case, the data contains a column that CLDF does not know out-of-the-box, but otherwise the dataset is fine.

```
$ cldf validate forms.csv
[...] UserWarning: Unspecified column "Cognateset_ID" in table forms.csv
  warnings.warn(
```

### Working with git

This is the point where it really makes sense to start working with `git`.

```
$ git init
[...]
Initialized empty Git repository in [...]bantu/.git/
$ git config user.name 'Lexedata'
$ git config user.email 'lexedata@example.com'
$ git add forms.csv
$ git commit -m "Initial import"
[master (root-commit) [...]] Initial import
 1 file changed, 1593 insertions(+)
 create mode 100644 forms.csv
```

### Adding metadata and explicit tables

A better structure for a lexical dataset – or any dataset, really – is to provide metadata. A CLDF dataset is described by a metadata file in JSON format. You can write such a file by hand in any text editor, but lexedata comes with a script that is able to guess some properties of the dataset and give you a metadata file template.

```
$ python -m lexedata.edit.add_metadata
INFO:lexedata:CLDF freely understood the columns ['Comment', 'Form', 'ID', 'Language_ID',
↪ 'Parameter_ID'] in your forms.csv.
INFO:lexedata:Column Cognateset_ID seems to be a http://cldf.clld.org/v1.0/terms.rdf
↪#cognatesetReference column.
INFO:lexedata:Also added column Segments, as expected for a FormTable.
INFO:lexedata:Also added column Source, as expected for a FormTable.
INFO:lexedata:FormTable re-written.
```

Lexedata has recognized the cognate judgement column correctly as what it is and also added two new columns to the dataset for sources (so we can track the origin of the data in a well-structured way) and for phonemic segmentation, which is useful in particular when working with sound correspondences on a segment-by-segment level. We will add segments in *a future section*.

With the new metadata file and the new columns, the dataset now looks like this:

```
$ ls
Wordlist-metadata.json
bantu.xlsx
forms.csv
$ cldf validate Wordlist-metadata.json
$ head Wordlist-metadata.json
{
    "@context": [
        "http://www.w3.org/ns/csvw",
        {
            "@language": "en"
        }
    ],
    "dc:conformsTo": "http://cldf.clld.org/v1.0/terms.rdf#Wordlist",
    "dc:contributor": [
        "https://github.com/Anaphory/lexedata/blob/master/src/lexedata/edit/add_metadata.
↪py"
```

(continues on next page)

(continued from previous page)

```
$ head forms.csv
ID,Language_ID,Parameter_ID,Form,Comment,Cognateset_ID,Segments,Source
duala_all,Duala,all,s,,1,,
duala_arm,Duala,arm,dia,,7,,
duala_ashes,Duala,ashes,mabúdú,,17,,
duala_bark,Duala,bark,bwelé,,23,,
duala_belly,Duala,belly,dibum,,1,,
duala_big,Duala,big,éndn,,1,,
duala_bird,Duala,bird,inn,,1,,
duala_bite,Duala,bite,kukwa,,6,,
duala_black,Duala,black,wínda,,21,,
```

The `cldf validate` script only outputs problems, so if it prints out nothing, it means that the dataset conforms to the CLDF standard! That's a good starting point to create a new commit.

```
$ git add Wordlist-metadata.json
$ git commit -m "Add metadata file"
[master [...]] Add metadata file
 1 file changed, 87 insertions(+)
 create mode 100644 Wordlist-metadata.json
```

Now that we have a good starting point, we can start working with the data and improving it. First, we change the template metadata file to include an actual description of what most people might understand when we say "metadata": Authors, provenience, etc.

```
{
    "@context": [
        "http://www.w3.org/ns/csvw",
        {
            "@language": "en"
        }
    ],
    "dc:conformsTo": "http://cldf.clld.org/v1.0/terms.rdf#Wordlist",
    "dc:contributor": [
        "https://github.com/Anaphory/lexedata/blob/master/src/lexedata/edit/
↪add_metadata.py"
    ],
    "dialect": {
        "commentPrefix": null
    },
    "tables": [
        {
            "dc:conformsTo": "http://cldf.clld.org/v1.0/terms.rdf#FormTable",
            "dc:extent": 1592,
            "tableSchema": {
                "columns": [
                    {
                        "datatype": {
                            "base": "string",
                            "format": "[a-zA-Z0-9_-]+"
                        },
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#id
↪",
```

(continues on next page)

(continued from previous page)

```
                        "required": true,
                        "name": "ID"
                    },
                    {
                        "dc:description": "A reference to a language (or␣
↪variety) the form belongs to",
                        "dc:extent": "singlevalued",
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#languageReference",
                        "required": true,
                        "name": "Language_ID"
                    },
                    {
                        "dc:description": "A reference to the meaning denoted␣
↪by the form",
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#parameterReference",
                        "required": true,
                        "name": "Parameter_ID"
                    },
                    {
                        "dc:description": "The written expression of the form.␣
↪If possible the transcription system used for the written form should be␣
↪described in CLDF metadata (e.g. via adding a common property␣
↪`dc:conformsTo` to the column description using concept URLs of the GOLD␣
↪Ontology (such as [phonemicRep](http://linguistics-ontology.org/gold/2010/
↪phonemicRep) or [phoneticRep](http://linguistics-ontology.org/gold/2010/
↪phoneticRep)) as values).",
                        "dc:extent": "singlevalued",
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#form",
                        "required": true,
                        "name": "Form"
                    },
                    {
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#comment",
                        "required": false,
                        "name": "Comment"
                    },
                    {
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#cognatesetReference",
                        "name": "Cognateset_ID"
                    },
                    {
                        "dc:extent": "multivalued",
```

(continues on next page)

(continued from previous page)

```
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#segments",
                        "required": false,
                        "separator": " ",
                        "name": "Segments"
                    },
                    {
                        "datatype": "string",
                        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf
↪#source",
                        "required": false,
                        "separator": ";",
                        "name": "Source"
                    }
                ],
                "primaryKey": [
                    "ID"
                ]
            },
            "url": "forms.csv"
        }
    ]
}
```

—Wordlist-metadata.json

And commit.

```
$ git commit -am "Add metadata"
[...]
```

## Adding satellite tables

Another useful step is to make languages, concepts, and cognate codes explicit. Currently, all the dataset knows about these their names. We can generate a scaffold for metadata about languages etc. with another tool.

```
$ python -m lexedata.edit.add_table LanguageTable
INFO:lexedata:Found 14 different entries for your new LanguageTable.
$ python -m lexedata.edit.add_table ParameterTable
INFO:lexedata:Found 100 different entries for your new ParameterTable.
WARNING:lexedata:Some of your reference values are not valid as IDs: ['go to', 'rain (v)
↪', 'sick, be', 'sleep (v)']. You can transform them into valid ids by running lexedata.
↪edit.simplify_ids
```

"Parameter" is CLDF speak for the things sampled per-language. In a StructureDataset this might be typological features, in a Wordlist the ParameterTable contains the concepts. The warning we will ignore for now.

Every form belongs to one language, and every language has multiple forms. This is a simple 1:n relationship. Every form has and one or more concepts associated to it (in this way, CLDF supports annotating polysemies) and every concept has several forms, in different languages but also synonyms within a single language. This can easily be reflected by entries in the FormTable. So far, so good.

```
$ git add languages.csv parameters.csv
$ git commit -am "Add language and concept tables"
[master [...]] Add language and concept tables
 3 files changed, 246 insertions(+), 1 deletion(-)
 create mode 100644 languages.csv
 create mode 100644 parameters.csv
```

The logic behind cognate judgements is slightly different. A form belongs to one or more cognate sets, but in addition to the cognate class, there may be additional properties of a cognate judgement, such as alignments, segments the judgement is about (if it is a partial cognate judgement), comments ("dubious: m~t is unexplained") or the source claiming the etymological relationship. Because of this, there is a separate table for cognate judgements, the CognateTable, and *that* table then refers to a CognatesetTable we can make explicit.

```
$ python -m lexedata.edit.add_cognate_table
CRITICAL:lexedata:You must specify whether cognatesets have dataset-wide unique ids or␣
↪not (--unique-id)
```

In our example dataset, cognate class "1" for all is not cognate with class "1" for arm, so we need to tell `add_cognate_table` that these IDs are only unique within a concept:

```
$ python -m lexedata.edit.add_cognate_table -q --unique-id concept
WARNING:lexedata:No segments found for form duala_all (s).
WARNING:lexedata:No segments found for form duala_arm (dia).
WARNING:lexedata:No segments found for form duala_ashes (mabúdú).
WARNING:lexedata:No segments found for form duala_bark (bwelé).
WARNING:lexedata:No segments found for 1592 forms. You can generate segments using␣
↪`lexedata.edit.segment_using_clts`.
```

### Clean the data

The cognate table needs to represent whether some or all of a form is judged to be cognate, and for that it needs the segments to be present. So before we continue, we use git to undo the creation of the cognate table.

```
$ git checkout .
Updated 2 paths from the index
```

Adding segments at this stage is dangerous: Some of our forms still contain comments etc., and as first step we should move those out of the actual form column.

```
$ python -m lexedata.edit.clean_forms
ERROR:lexedata:Line 962: Form 'raiha (be long' has unbalanced brackets. I did not modify␣
↪the row.
INFO:lexedata:Line 106: Split form 'lopoho ~ mpoho ~ lòpòhó' into 3 elements.
INFO:lexedata:Line 113: Split form 'lokúa ~ nkúa' into 2 elements.
INFO:lexedata:Line 116: Split form 'yǒmbi ~ bíómbi' into 2 elements.
INFO:lexedata:Line 154: Split form 'lopíko ~ mpíko' into 2 elements.
INFO:lexedata:Line 162: Split form 'ngómbá ~ ngòmbá' into 2 elements.
INFO:lexedata:Line 165: Split form 'lokála ~ nkála' into 2 elements.
INFO:lexedata:Line 169: Split form 'moólo ~ miólo' into 2 elements.
INFO:lexedata:Line 171: Split form 'mbókà ~ mambóka' into 2 elements.
INFO:lexedata:Line 194: Split form 'yěmi ~ elemi' into 2 elements.
INFO:lexedata:Line 211: Split form 'búdùlù ~ pùdùlù' into 2 elements.
```

(continues on next page)

(continued from previous page)

```
INFO:lexedata:Line 212: Split form 'émpósù ~ ímpósù' into 2 elements.
INFO:lexedata:Line 214: Split form 'nn ~ nnn' into 2 elements.
[...]
```

Good job! Sometimes the form that is more interesting for historical linguistics may have ended up in the ‘variants’ column, but overall, this is a big improvement.

## Add phonemic segments

Then we add the segments using the dedicated script.

```
$ python -m lexedata.edit.add_segments -q # doctest: +NORMALIZE_WHITESPACE
WARNING:lexedata:In form duala_one (line 67): Impossible sound '/' encountered in p / ew␣
↪- You cannot use CLTS extended normalization with this script. The slash was skipped␣
↪and not included in the segments.
WARNING:lexedata:In form duala_snake (line 84): Unknown sound ' encountered in nam'a␣
↪bwaba
WARNING:lexedata:In form ngombe_all (line 210): Unknown sound ń encountered in ńsò
WARNING:lexedata:In form ngombe_cold (line 227): Unknown sound  encountered in pyo
WARNING:lexedata:In form bushoong_dog_s2 (line 363): Unknown sound m encountered in mmbwá
WARNING:lexedata:In form bushoong_neck_s2 (line 411): Unknown sound  encountered in ikl’l
WARNING:lexedata:In form bushoong_sleep_v (line 430): Unknown sound ' encountered in abem
↪'t
WARNING:lexedata:In form nzebi_bone (line 564): Unknown sound š encountered in l-šií
WARNING:lexedata:In form nzebi_give (line 587): Unknown sound š encountered in š
WARNING:lexedata:In form nzebi_hair (line 589): Unknown sound * encountered in l-náàŋgá␣
↪* náàŋgá
WARNING:lexedata:In form nzebi_nail (line 612): Unknown sound * encountered in l-âdà *␣
↪âdà
WARNING:lexedata:In form nzebi_path (line 618): Unknown sound * encountered in ndzilá *␣
↪mà-ndzilá
WARNING:lexedata:In form nzebi_person (line 619): Unknown sound * encountered in mùù-tù␣
↪* bàà-tà
WARNING:lexedata:In form nzebi_seed (line 627): Unknown sound š encountered in ì-šdí
WARNING:lexedata:In form nzadi_arm (line 655): Unknown sound ` encountered in lwŏ`
WARNING:lexedata:In form nzadi_new_s2 (line 740): Unknown sound * encountered in odzá:ng␣
↪* nzáng
WARNING:lexedata:In form nzadi_rain_s2 (line 750): Unknown sound  encountered in mbvl
WARNING:lexedata:In form nzadi_tongue (line 779): Unknown sound  encountered in llm
WARNING:lexedata:In form nzadi_tongue (line 779): Unknown sound  encountered in llm
WARNING:lexedata:In form lega_woman_s2 (line 903): Unknown sound o encountered in mo-kazi
WARNING:lexedata:In form kikuyu_long_s2 (line 963): Unknown sound ( encountered in raiha␣
↪(be long
WARNING:lexedata:In form kikuyu_tail_s2 (line 1009): Unknown sound ' encountered in gĩ-
↪tong'oe
WARNING:lexedata:In form swahili_bite (line 1141): Unknown sound ' encountered in ng'ata
| LanguageID   | Sound   |   Occurrences | Comment                                       ␣
↪                                                 |
|--------------+---------+---------------+-----------------------------------------------
↪-------------------------------------------------|
| Duala        |         |             1 | illegal symbol                                ␣
↪                                                 |
```

(continues on next page)

(continued from previous page)

```
| Duala         | '       |             1 | unknown sound                                  ␣
↪                                       |
| Ngombe        | ń      |             1 | unknown sound                                  ␣
↪                                      |
| Ngombe        |       |             1 | unknown sound                                  ␣
↪                                     |
| Bushoong      | m      |             1 | unknown sound                                  ␣
↪                                      |
| Bushoong      |        |             1 | unknown sound                                  ␣
↪                                      |
| Bushoong      | '       |             1 | unknown sound                                  ␣
↪                                       |
| Nzebi         | š      |             3 | unknown sound                                  ␣
↪                                      |
| Nzebi         | *       |             4 | unknown sound                                  ␣
↪                                       |
| Nzadi         |        |             8 | '' replaced by '' in segments. Run with `--
↪replace-form` to apply this also to the forms. |
| Nzadi         | `       |             1 | unknown sound                                  ␣
↪                                       |
| Nzadi         | *       |             1 | unknown sound                                  ␣
↪                                       |
| Nzadi         |       |             2 | unknown sound                                  ␣
↪                                     |
| Nzadi         |        |             1 | unknown sound                                  ␣
↪                                      |
| Lega          | o      |             1 | unknown sound                                  ␣
↪                                      |
| Kikuyu        | (       |             1 | unknown sound                                  ␣
↪                                       |
| Kikuyu        | '       |             1 | unknown sound                                  ␣
↪                                       |
| Swahili       | '       |             1 | unknown sound                                  ␣
↪                                       |
```

Some of those warnings relate to unsplit forms. We should clean up a bit, and tell `clean_forms` about new separators and re-run:

```
$ git checkout .
Updated 2 paths from the index
$ sed -i.bak -e '/kikuyu_long_s2/s/(be long/(be long)/' forms.csv
$ python -m lexedata.edit.clean_forms -k '~' '*' -s ',' ';' '/'
INFO:lexedata:Line 66: Split form 'p / ew' into 2 elements.
[...]
INFO:lexedata:Line 588: Split form 'l-náàŋgá * náàŋgá' into 2 elements.
INFO:lexedata:Line 611: Split form 'l-âdà * âdà' into 2 elements.
INFO:lexedata:Line 617: Split form 'ndzilá * mà-ndzilá' into 2 elements.
INFO:lexedata:Line 618: Split form 'mùù-tù * bàà-tà' into 2 elements.
INFO:lexedata:Line 625: Split form 'm ~ mn' into 2 elements.
INFO:lexedata:Line 725: Split form 'i-baa ~ i-bál' into 2 elements.
INFO:lexedata:Line 739: Split form 'odzá:ng * nzáng' into 2 elements.
[...]
```

(continues on next page)

(continued from previous page)

```
$ python -m lexedata.edit.add_segments -q --replace-form # doctest: +NORMALIZE_WHITESPACE
WARNING:lexedata:In form duala_snake (line 84): Unknown sound ' encountered in nam'a␣
↪bwaba
WARNING:lexedata:In form ngombe_all (line 210): Unknown sound ń encountered in ńsò
WARNING:lexedata:In form ngombe_cold (line 227): Unknown sound  encountered in pyo
WARNING:lexedata:In form bushoong_dog_s2 (line 363): Unknown sound m encountered in mmbwá
WARNING:lexedata:In form bushoong_neck_s2 (line 411): Unknown sound  encountered in ikl’l
WARNING:lexedata:In form bushoong_sleep_v (line 430): Unknown sound ' encountered in abem
↪'t
WARNING:lexedata:In form nzebi_bone (line 564): Unknown sound š encountered in l-šií
WARNING:lexedata:In form nzebi_give (line 587): Unknown sound š encountered in š
WARNING:lexedata:In form nzebi_seed (line 627): Unknown sound š encountered in ì-šdí
WARNING:lexedata:In form nzadi_arm (line 655): Unknown sound ` encountered in lwǒ`
WARNING:lexedata:In form nzadi_rain_s2 (line 750): Unknown sound  encountered in mbvl
WARNING:lexedata:In form nzadi_tongue (line 779): Unknown sound  encountered in llm
WARNING:lexedata:In form nzadi_tongue (line 779): Unknown sound  encountered in llm
WARNING:lexedata:In form lega_woman_s2 (line 903): Unknown sound o encountered in mo-kazi
WARNING:lexedata:In form kikuyu_tail_s2 (line 1009): Unknown sound ' encountered in gĩ-
↪tong'oe
WARNING:lexedata:In form swahili_bite (line 1141): Unknown sound ' encountered in ng'ata
| LanguageID   | Sound   |   Occurrences | Comment                                      |
|--------------+---------+---------------+----------------------------------------------|
| Duala        | '       |             1 | unknown sound                                |
| Ngombe       | ń      |             1 | unknown sound                                |
| Ngombe       |        |             1 | unknown sound                                |
| Bushoong     | m      |             1 | unknown sound                                |
| Bushoong     |        |             1 | unknown sound                                |
| Bushoong     | '       |             1 | unknown sound                                |
| Nzebi        | š      |             3 | unknown sound                                |
| Nzadi        |        |             8 | '' replaced by '' in segments and forms. |
| Nzadi        | `       |             1 | unknown sound                                |
| Nzadi        |        |             2 | unknown sound                                |
| Nzadi        |        |             1 | unknown sound                                |
| Lega         | o      |             1 | unknown sound                                |
| Kikuyu       | '       |             1 | unknown sound                                |
| Swahili      | '       |             1 | unknown sound                                |
```

There are a few unknown symbols left in the data, but most of it is clean IPA now.

```
$ git commit -am "Clean up forms"
[...]
```

### Add more tables

With the segments in place, we can go back to adding the cognate table back in and proceed to add the cognateset table.

```
$ python -m lexedata.edit.add_cognate_table -q --unique-id concept
$ python -m lexedata.edit.add_table CognatesetTable
INFO:lexedata:Found 651 different entries for your new CognatesetTable.
$ git add cognates.csv cognatesets.csv
$ git commit -am "Add cognate and cognateset tables"
[...]
```

### Create a consistent dataset

Now all the external properties of a form can be annotated with explicit metadata in their own table files, for example for the languages:

```
ID,Name,Macroarea,Latitude,Longitude,Glottocode,ISO639P3code
Bushoong,Bushoong,,,,,
Duala,Duala,,,,,
Fwe,Fwe,,,,,
Ha,Ha,,,,,
Kikuyu,Kikuyu,,,,,
Kiyombi,Kiyombi,,,,,
Lega,Lega,,,,,
Luganda,Luganda,,,,,
Ngombe,Ngombe,,,,,
Ntomba,Ntomba,,,,,
Nyamwezi,Nyamwezi,,,,,
Nzadi,Nzadi,,,,,
Nzebi,Nzebi,,,,,
Swahili,Swahili,,,,,
```

—languages.csv

If you edit files by hand, it's always good to check CLDF compliance afterwards – small typos are just too easy to make, and they don't catch the eye.

```
$ git commit -am "Update language metadata"
[...]
$ cldf validate Wordlist-metadata.json
WARNING parameters.csv:37:1 ID: invalid lexical value for string: go to
WARNING parameters.csv:70:1 ID: invalid lexical value for string: rain (v)
WARNING parameters.csv:77:1 ID: invalid lexical value for string: sick, be
WARNING parameters.csv:80:1 ID: invalid lexical value for string: sleep (v)
WARNING parameters.csv:37:1 ID: invalid lexical value for string: go to
WARNING parameters.csv:70:1 ID: invalid lexical value for string: rain (v)
WARNING parameters.csv:77:1 ID: invalid lexical value for string: sick, be
WARNING parameters.csv:80:1 ID: invalid lexical value for string: sleep (v)
WARNING forms.csv:39 Key `go to` not found in table parameters.csv
WARNING forms.csv:72 Key `rain (v)` not found in table parameters.csv
WARNING forms.csv:79 Key `sick, be` not found in table parameters.csv
WARNING forms.csv:82 Key `sleep (v)` not found in table parameters.csv
[...]
```

Ah, we had been warned about something like this above. We can easily fix this by removing the 'format' restriction from ParameterTable's ID column:

```
$ patch -u --verbose > /dev/null << EOF
> --- Wordlist-metadata.json        2021-12-12 02:04:28.519080902 +0100
> +++ Wordlist-metadata.json~       2021-12-12 02:05:36.161817085 +0100
> @@ -181,8 +181,7 @@
>                  "columns": [
>                      {
>                          "datatype": {
> -                            "base": "string",
> -                            "format": "[a-zA-Z0-9_\\\-]+"
> +                            "base": "string"
>                          },
>                          "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#id",
>                          "required": true,
> @@ -329,4 +328,4 @@
>              "url": "cognatesets.csv"
>          }
>      ]
> -}
> \ No newline at end of file
> +}
> EOF
```

Now the dataset conforms to cldf:

```
$ cldf validate Wordlist-metadata.json
$ git commit -am "Make dataset valid!"
[...]
```

## Extended extended CLDF compatibility

We have taken this dataset from a somewhat ideosyncratic format to metadata-free CLDF and to a dataset with extended CLDF compliance. The `cldf validate` script checks for strict conformance with the CLDF standard. However, there are some assumptions which lexedata and also some other CLDF-aware tools tend to make which are not strictly mandated by the CLDF specifications. One such assumption is the one that led to the issue above:

> Each CLDF data table SHOULD contain a column which uniquely identifies a row in the table. This column SHOULD be marked using:
>
> - a propertyUrl of http://cldf.cld.org/v1.0/terms.rdf#id
> - the column name ID in the case of metadata-free conformance.
>
> To allow usage of identifiers as path components of URIs and ensure they are portable across systems, identifiers SHOULD be composed of alphanumeric characters, underscore _ and hyphen - only, i.e. match the regular expression `[a-zA-Z0-9\-_]+` (see RFC 3986).
>
> —https://github.com/cldf/cldf#identifier

Because of the potential use in URLs, our table adder adds tables with the ID format that we encountered above. This specification uses the word 'SHOULD', not 'MUST', which allows to ignore the requirement in certain circumstances and thus `cldf validate` does not enforce it. We do however provide a separate report script that points out this and other deviations from sensible assumptions.

```
$ python -m lexedata.report.extended_cldf_validate 2>&1 | head -n 2
WARNING:lexedata:Table parameters.csv has an unconstrained ID column ID. Consider␣
↪setting its format to [a-zA-Z0-9_-]+ and/or running `lexedata.edit.simplify_ids`.
INFO:lexedata:Caching table forms.csv
```

As that message tells us (I have cut off all the later messages, showing only the first two lines of output), we can fix this using another tool from the lexedata toolbox:

```
$ python -m lexedata.edit.simplify_ids --table parameters.csv
INFO:lexedata:Handling table parameters.csv...
[...]
$ git commit -am "Regenerate concept IDs"
[...]
```

This was however not the only issue with the data.

```
$ python -m lexedata.report.extended_cldf_validate -q
WARNING:lexedata:In cognates.csv, row 2: Referenced segments in form resolve to   s ,␣
↪while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 3: Referenced segments in form resolve to d i a,␣
↪while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 4: Referenced segments in form resolve to m a b ú␣
↪d ú, while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 5: Referenced segments in form resolve to b w e l␣
↪é, while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 6: Referenced segments in form resolve to d i b u␣
↪m, while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 7: Referenced segments in form resolve to é n d  n␣
↪, while alignment contains segments .
WARNING:lexedata:In cognates.csv, row 8: Referenced segments in form resolve to i n  n,␣
↪while alignment contains segments .
[...]
```

The alignment column of the cognate table is empty, so for no form is there a match between the segments assigned to a cognate set (the segment slice, applied to the segments in the FormTable) and the segments occuring in the alignment. The easy way out here is the alignment script – which is not very clever, but working on the cognate data in detail is a later step.

```
$ python -m lexedata.edit.align
INFO:lexedata:Caching table FormTable
100%|| 1592/1592 [...]
INFO:lexedata:Aligning the cognate segments
100%|| 1592/1592 [...]
$ git commit -am "Align"
[...]
```

Lastly, with accented unicode characters, there are (simlified) two different conventions: Storing the characters as composed as possible (so è would be a single character) or as decomposed as possible (storing è as a combining ` character and e). We generally use the composed "NFC" convention, so if you are in doubt, you can always normalize them to that convention.

```
$ python -m lexedata.edit.normalize_unicode
INFO:lexedata:Normalizing [...]forms.csv...
```

(continues on next page)

(continued from previous page)

```
INFO:lexedata:Normalizing [...]languages.csv...
INFO:lexedata:Normalizing [...]parameters.csv...
INFO:lexedata:Normalizing [...]cognates.csv...
INFO:lexedata:Normalizing [...]cognatesets.csv...
$ python -m lexedata.report.extended_cldf_validate -q
$ git commit -am "Get data ready to start editing"
[...]
```

We have told the extended validator to be quiet, so no output means it has nothing to complain about: Our dataset is not only valid CLDF, but also compatible with the general assumptions of lexedata.

### 1.2.2 Editing the dataset

We are about to start editing. In the process, we may introduce new issues into the dataset. Therefore it makes sense to mark this current version with a git tag. If we ever need to return to this version, the tag serves as a memorable anchor.

```
$ git tag import_complete
```

#### Adding status columns

While editing datasets, it is often useful to track the status of different objects. This holds in particular when some non-obvious editing steps are done automatically. Due to this, lexedata supports status columns. Many scripts fill the status column of a table they manipulate with a short message. The `align` script has already done that for us:

```
$ head -n3 cognates.csv
ID,Form_ID,Cognateset_ID,Segment_Slice,Alignment,Source,Status_Column
duala_all,duala_all,all_1,1:4,  s  - -,,automatically aligned
duala_arm,duala_arm,arm_7,1:3,d i a,,automatically aligned
```

Most scripts do not add a status column if there is none. To make use of this functionality, we therefore add status columns to all tables.

```
$ python -m lexedata.edit.add_status_column
INFO:lexedata:Tables to have a status column: ['forms.csv', 'cognatesets.csv', 'cognates.
↪csv', 'parameters.csv']
INFO:lexedata:Table cognates.csv already contains a Status_Column.
$ git commit -am "Add status columns"
[...]
```

#### Improve Concepts

The first items we want to edit are the concepts, and the links between the forms and the concepts. Currently, our parameter table lists for every concept only a name and an ID derived from the name. There is also space for a description, which we have left unfilled.

For many subsequent tasks, it is useful to know whether concepts are related or not. The CLICS$^3$ database contains a network of colexifications: Concepts that are expressed by the same form in vastly different languages can be assumed to be related. Lexedata comes with a copy of the CLICS$^3$ network, but in order to use it, we need to map concepts to Concepticon, a catalog of concepts found in different word lists.

### Guess Concepticon links

Concepticon comes with some functionality to guess concepticon IDs based on concept glosses. The concepticon script only takes one gloss language into account. Lexedata provides a script that can take multiple gloss languages – we don't have those here, but the lexedata script can also add Concepticon's normalized glosses and definitions to our parameter table, so we use that script here. Our "Name" column in the ParameterTable contains English ("en") glosses, so pass that information to the script:

```
$ python -m lexedata.edit.add_concepticon -q -l Name=en --add-concept-set-names --add-
↪definitions
OrderedDict([('ID', 'bark'), ('Name', 'bark'), ('Description', None), ('Status_Column',␣
↪None), ('Concepticon_ID', None)]) 2 [('1204', 3), ('1206', 1)]
OrderedDict([('ID', 'breast'), ('Name', 'breast'), ('Description', None), ('Status_Column
↪', None), ('Concepticon_ID', None)]) 2 [('1402', 3), ('1592', 1)]
[...]
```

The output shows the concepts in our dataset with some ambiguous mappings to concepticon. Now is the time to check andif necessary fix the mappings.

```
$ cat parameters.csv
ID,Name,Description,Status_Column,Concepticon_ID,Concepticon_Gloss,Concepticon_Definition
all,all,,automatic Concepticon link,98,ALL,The totality of.
arm,arm,,automatic Concepticon link,1673,ARM,"The upper limb, extending from the␣
↪shoulder to the wrist and sometimes including the hand."
[...]
$ sed -i.bak -s 's/^go_to.*/go_to,go to,,Concepticon link checked,695,GO,To get from one␣
↪place to another by any means./' parameters.csv
$ sed -i.bak -s 's/automatic Concepticon link/Concepticon link checked/' parameters.csv
```

### Merging polysemous forms

There are a few identical forms in different concepts. Because we have connected our concepts to Concepticon, and therefore we have access to their CLICS$^3$ network, the homophones report can tell us whether two concepts are connected and thus likely polysemies of a single word:

```
$ python -m lexedata.report.homophones -o homophones.txt
$ cat homophones.txt
Ntomba, 'lopoho': Connected:
     ntomba_bark (bark)
     ntomba_skin (skin)
Ngombe, 'nn': Connected:
     ngombe_big (big)
     ngombe_many (many)
Bushoong, 'yn': Connected:
     bushoong_go_to (go_to)
     bushoong_walk (walk)
Bushoong, 'dǐin': Connected:
     bushoong_name (name)
     bushoong_tooth (tooth)
Nzadi, 'o-tûm': Unconnected:
     nzadi_dig (dig)
     nzadi_heart_s2 (heart)
```

(continues on next page)

(continued from previous page)

```
Lega, 'nda': Connected:
     lega_go_to (go_to)
     lega_walk (walk)
Kikuyu, 'rĩa': Connected:
     kikuyu_eat (eat)
     kikuyu_what (what)
Kikuyu, 'erũ': Unconnected:
     kikuyu_new (new)
     kikuyu_white (white)
Swahili, 'jua': Unconnected:
     swahili_know (know)
     swahili_sun (sun)
Ha, 'inda': Unconnected:
     ha_belly (belly)
     ha_louse (louse)
Ha, 'gwa': Unconnected:
     ha_fall (fall)
     ha_rain_v (rain_v)
Fwe, 'wa': Unconnected:
     fwe_fall (fall)
     fwe_give_s2 (give)
Fwe, 'ya': Unconnected:
     fwe_go_to (go_to)
     fwe_new (new)
```

The output is not as helpful as we might have hoped (that 'bark' and 'skin' are connected makes sense, but 'eat' and 'what' are connected and 'new' and 'white' disconnected?). We can edit this[1] to keep the polysemies

```
$ cat > polysemies.txt << EOF
> Ntomba, 'lopoho': Connected:
>    ntomba_skin (skin)
>    ntomba_bark (bark)
> Ngombe, 'nn': Connected:
>    ngombe_big (big)
>    ngombe_many (many)
> Kikuyu, 'erũ': Unconnected:
>    kikuyu_new (new)
>    kikuyu_white (white)
> Bushoong, 'yn': Connected:
>    bushoong_go_to (go_to)
>    bushoong_walk (walk)
> Lega, 'nda': Connected:
>    lega_go_to (go_to)
>    lega_walk (walk)
> EOF
```

and feed this file into the 'homophones merger', which turns separate forms into polysemous forms connected to multiple concepts.

```
$ grep 'kikuyu_\(white\|new\)' forms.csv cognates.csv
forms.csv:kikuyu_new,Kikuyu,new,erũ,,e r ũ,,
```

(continues on next page)

[1] The syntax I used to describe files before does not like indented lines in the file, but they are integral to the structure of the polysemies list.

(continued from previous page)

```
forms.csv:kikuyu_white,Kikuyu,white,erũ,,e r ũ,,
cognates.csv:kikuyu_new,kikuyu_new,new_3,1:3,e r ũ,,automatically aligned
cognates.csv:kikuyu_white,kikuyu_white,white_2,1:3,e r ũ,,automatically aligned
$ python -m lexedata.edit.merge_homophones polysemies.txt
WARNING:lexedata:I had to set a separator for your forms' concepts. I set it to ';'.
INFO:lexedata:Going through forms and merging
100%|| 1592/1592 [...]
$ grep 'kikuyu_\(white\|new\)' forms.csv cognates.csv
forms.csv:kikuyu_new,Kikuyu,new;white,erũ,,e r ũ,,
cognates.csv:kikuyu_new,kikuyu_new,new_3,1:3,e r ũ,,automatically aligned
cognates.csv:kikuyu_white,kikuyu_new,white_2,1:3,e r ũ,,automatically aligned
$ git commit -am "Annotate polysemies"
[master [...]] Annotate polysemies
 4 files changed, 3302 insertions(+), 3288 deletions(-)
 rewrite parameters.csv (100%)
```

## Improve Cognatesets

Now the dataset is in a very good shape. We can now start with the historical linguistics, editing cognatesets and alignments.

## Merge cognatesets

From combining polysemous forms, we now have forms which are in two cognate sets. Apart from this artefact of how we handle the data, cognate sets which do not represent disjoint, consecutive groups of segments also occur when morpheme boundaries have been eroded or when a language has non-concatenative morphemes, which is the case that gives the name to our script reporting these.

```
$ python -m lexedata.report.nonconcatenative_morphemes > overlapping_cogsets
[...]
WARNING:lexedata:In form ntomba_skin, segments are associated with multiple cognate sets.
INFO:lexedata:In form ntomba_skin, segments 1:6 (l o p o h o) are in both cognate sets␣
↪bark_22 and skin_27.
WARNING:lexedata:In form ngombe_big, segments are associated with multiple cognate sets.
INFO:lexedata:In form ngombe_big, segments 1:4 (n  n ) are in both cognate sets big_1␣
↪and many_12.
WARNING:lexedata:In form bushoong_go_to, segments are associated with multiple cognate␣
↪sets.
INFO:lexedata:In form bushoong_go_to, segments 1:4 (y   n) are in both cognate sets go_
↪to_1 and walk_1.
WARNING:lexedata:In form lega_go_to, segments are associated with multiple cognate sets.
INFO:lexedata:In form lega_go_to, segments 1:4 ( n d a) are in both cognate sets go_to_2␣
↪and walk_1.
WARNING:lexedata:In form kikuyu_new, segments are associated with multiple cognate sets.
INFO:lexedata:In form kikuyu_new, segments 1:3 (e r ũ) are in both cognate sets new_3␣
↪and white_2.
$ cat overlapping_cogsets # doctest: +NORMALIZE_WHITESPACE
Cluster of overlapping cognate sets:
    bark_22
    skin_27
```

(continues on next page)

(continued from previous page)

```
Cluster of overlapping cognate sets:
    big_1
    many_12
Cluster of overlapping cognate sets:
    go_to_1
    go_to_2
    walk_1
Cluster of overlapping cognate sets:
    new_3
    white_2
```

There are other ways to merge cognate sets, which we will see in a moment, but this kind of structured report is suitable for automatic merging, in the same manner as the homophones:

```
$ python -m lexedata.edit.merge_cognate_sets overlapping_cogsets
[...]
INFO:lexedata:Writing cognates.csv back to file...
```

(TODO: This script does not yet merge the two different judgements that associate one form with the now one cognate set.)

### Central Concepts

Our cognate sets can now contain forms associated with multiple concepts. For further work it is often useful to track 'central' concepts, or tentative semantic reconstructions, together with the cognate sets. Lexedata can generall help bootstrap this, using again the link to Concepticon and CLICS[3].

```
$ python -m lexedata.edit.add_central_concepts
[... progress output]
$ git commit -am "Add central concepts"
[...]
```

### Informative reports

If we want to check the phoneme inventories implied by the segmentation generated initially, we can use one of the reports:

```
$ python -m lexedata.report.segment_inventories -q --language Nzebi # doctest:␣
↪+NORMALIZE_WHITESPACE
Nzebi
| Sound   |   Occurrences | Comment      |
|---------+---------------+--------------|
| m       |            37 |              |
| l       |            36 |              |
| ù       |            31 |              |
| à       |            31 |              |
| a       |            26 |              |
| b       |            25 |              |
| n       |            23 |              |
| t       |            17 |              |
```

(continues on next page)

(continued from previous page)

```
|       |            17 |              |
| k       |            17 |              |
| g       |            15 |              |
| u       |            15 |              |
| i       |            14 |              |
| s       |            14 |              |
| d       |            14 |              |
| á       |            13 |              |
| x       |            12 |              |
| ì       |            12 |              |
| í       |            12 |              |
| ŋ       |            11 |              |
| y       |            11 |              |
|       |            11 |              |
|        |            11 |              |
|       |            10 |              |
|        |            10 |              |
| ú       |             8 |              |
| o       |             7 |              |
| â       |             7 |              |
|        |             6 |              |
|       |             6 |              |
| z       |             6 |              |
|        |             5 |              |
| e       |             5 |              |
|       |             4 |              |
|       |             4 |              |
| v       |             4 |              |
| û       |             4 |              |
| ò       |             3 |              |
| p       |             3 |              |
| š       |             3 | Invalid BIPA |
| f       |             2 |              |
| ô       |             2 |              |
| ê       |             2 |              |
| é       |             2 |              |
| è       |             2 |              |
| ă       |             2 |              |
|        |             2 |              |
| r       |             1 |              |
| _       |             1 | Marker       |
| î       |             1 |              |
```

The reports fulfill different functions. Some, as you have seen, focus on issues with the internal correctness of the dataset. Others, like the `segment_inventories` report above or *lexedata.report.coverage*, are useful for statistical summaries of the dataset. And a third group, such as the homophones report, generate output that can be used as input to other `lexedata` scripts. Another example for such a script is *lexedata.report.filter*, which filters a table from the dataset. This is useful for any of the scripts that can take a list from a file as input, such as the concept list for *lexedata.exporter.matrix*. For example, the concepts that are 'number's according to their concepticon definition can be found using

```
$ python -m lexedata.report.filter Concepticon_Definition number ParameterTable -q
```

(continues on next page)

(continued from previous page)

```
ID,Name,Description,Status_Column,Concepticon_ID,Concepticon_Gloss,Concepticon_Definition
many,many,,Concepticon link checked,1198,MANY,An indefinite large number of.
one,one,,Concepticon link checked,1493,ONE,The natural number one (1).
two,two,,Concepticon link checked,1498,TWO,The natural number two (2).
```

## 1.2.3 Computer-assisted historical linguistics

We can now modify the cognate judgements. Lexedata currently supports two ways to do this, both work by exporting the lexical dataset to an external format more handy for editing, and then importing it back in.

### Cognate Excel

The first export-import loop works to provide us with a large table showing the cognate sets per language, using *lexedata.exporter.cognates* and *lexedata.importer.cognates*. Showing that does not very well fit the format of this tutorial, so we will skip it for now. But feel free to try it out: If you commit your status before your try out this loop, you always have a safe state to come back to. If you also re-import and re-export frequently, you decrease the chance of accidentally introducing errors to the format which Lexedata cannot parse, or at least the time it takes you to find and correct such errors.

### Edictor

The second export-import loop lexedata implements exports to the TSV format used by edictor, a JavaScript-based in-browser editor for lexical datasets. (Edictor runs purely inside your browser on your computer, there is no data transmission involved.)

For this example, we will look more closely at the concepts of locomotion. We have already seen some overlap between the forms for 'go to' and 'walk', so we will check those in more detail. First, we select the subset of the data that is interesting here. Let us consider the concepts

```
ID
go_to
walk
path
come
stand
```

—concepts_filter

and all languages:

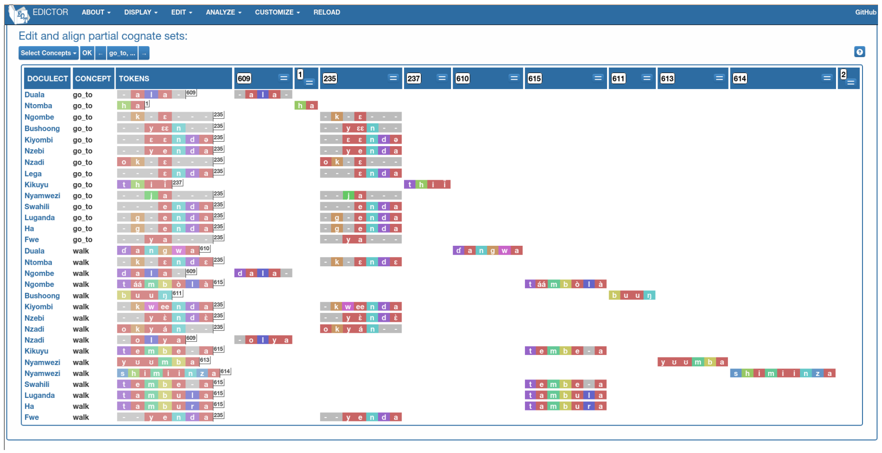
```
$ python -m lexedata.exporter.edictor --concepts concepts_filter -q
INFO:lexedata:Reading concept IDs from column with header ID
$ head cognate.tsv # doctest: +NORMALIZE_WHITESPACE
ID  CLDF_id DOCULECT        CONCEPT IPA     comment TOKENS  source  variants      ␣
↪COGID   ALIGNMENT       _parameterReference
1   duala_come      Duala   come    p               p                       127     p  -  ␣
↪come
2   duala_go_to     Duala   go_to   ala             a l a                     236     a l␣
↪a   go_to
3   duala_path      Duala   path    ngea            n g e a                   424     n g␣
↪e a - -     path
```

(continues on next page)

(continued from previous page)

```
4   duala_stand     Duala   stand   tm mny              t  m  _ m  n y                    ␣
↪564     t  m  _ m  n y    stand
5   duala_walk      Duala   walk    angwa           a n g w a                    610   ␣
↪  a n g w a     walk
6   ntomba_come     Ntomba  come    yá              y á                    125     y á -
↪ - - - -     come
7   ntomba_go_to    Ntomba  go_to   ha              h a                    235     h a -
↪ - -         go_to
8   ntomba_path     Ntomba  path    mbókà           m b ó k à              ~mambóka    ␣
↪   426     m b ó k à      path
9   ntomba_stand    Ntomba  stand   tlm          t  l  m                     565     t ␣
↪l  m - - -         stand
```

This gives us a tab-separated value file, by default named *cognate.tsv*, which we can load in Edictor and edit there.



| DOCULECT | CONCEPT | TOKENS |
|---|---|---|
| Duala | go_to | - a l a - 609 |
| Ntomba | go_to | h a 1 |
| Ngombe | go_to | - k - ɛ - - - 235 |
| Bushoong | go_to | - - y ɛɛ n - - 235 |
| Kiyombi | go_to | - - ɛ ɛ n d ə 235 |
| Nzebi | go_to | - - y e n d a 235 |
| Nzadi | go_to | o k - ɛ - - - 235 |
| Lega | go_to | - - - ɛ n d a 235 |
| Kikuyu | go_to | t h i ĩ 237 |
| Nyamwezi | go_to | - - j a - - - 235 |
| Swahili | go_to | - - - e n d a 235 |
| Luganda | go_to | - g - e n d a 235 |
| Ha | go_to | - g - e n d a 235 |
| Fwe | go_to | - - y a - - - 235 |
| Duala | walk | ɗ a n g w a 610 |
| Ntomba | walk | - k - ɛ n d ɛ 235 |
| Ngombe | walk | d a l a - 609 |
| Ngombe | walk | t áá m b ò l à 615 |
| Bushoong | walk | b u u ŋ 611 |
| Kiyombi | walk | - k w ee n d a 235 |
| Nzebi | walk | - - y ɛ̀ n d ɛ̀ 235 |
| Nzadi | walk | o k y á n - - 235 |
| Nzadi | walk | - o l y a 609 |
| Kikuyu | walk | t e m b e - a 615 |
| Nyamwezi | walk | y ʊ ʊ m b a 613 |
| Nyamwezi | walk | s h i m i i n z a 614 |
| Swahili | walk | t e m b e - a 615 |
| Luganda | walk | t a m b u l a 615 |
| Ha | walk | t a m b u r a 615 |
| Fwe | walk | - - y e n d a 235 |

This is not the point to show you the workings of Edictor. I have edited things a bit, the result is in the documentation.

```
$ curl -L https://github.com/Anaphory/lexedata/blob/master/docs/examples/cognate.tsv?
↪raw=true -o new_cognate.tsv
[...]
$ python -m lexedata.importer.edictor -i new_cognate.tsv
[...]
INFO:lexedata:The header of your edictor file will be interpreted as ['', 'id',
↪'languageReference', '', 'form', 'comment', 'segments', 'source', 'variants',
↪'cognatesetReference', 'alignment', 'parameterReference'].
[...]
```

## 1.2.4 Further steps

### Phylogenetics

There is of course still much room for improvement, but just for demonstration purposes:

```
$ python -m lexedata.exporter.phylogenetics --coding multistate
[...]
Bushoong  2,0,3,4,2,0,2,0,(7,8),1,2,1,(3,11),(7,8),13,0,0,12,(0,3),0,0,0,0,4,0,0,0,0,2,3,
↪1,0,4,(0,1),7,0,(2,5),0,0,3,(2,3),(2,3),2,2,0,0,8,1,5,1,2,(0,6),5,1,7,2,5,3,0,0,(5,8),
↪(1,6),5,4,0,2,0,0,5,4,4,6,0,5,0,7,4,11,6,5,3,0,(8,9),0,1,4,8,0,?,0,0,3,1,(0,3),0,0,3,0,
↪(2,8),3
```

(continues on next page)

(continued from previous page)

```
Duala      0,2,4,5,0,0,0,4,6,1,1,1,4,2,4,2,0,(4,6,7),0,0,3,0,0,7,0,0,0,1,2,0,4,0,0,5,1,3,
 ↪7,3,1,3,1,7,1,2,2,3,10,2,5,3,0,0,4,0,5,0,8,2,3,0,0,5,3,3,1,0,0,0,3,7,3,3,2,0,1,3,7,11,
 ↪9,2,1,6,(4,5),5,0,6,5,3,2,0,2,2,2,2,1,1,2,0,2,2
Fwe        0,0,0,0,0,0,0,0,0,0,0,0,(0,11),0,(2,13),0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 ↪0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,(0,3),0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
 ↪4,0,1,0,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,6,3
Ha         0,0,2,?,7,0,(0,1,4),?,10,3,0,1,(0,10),3,(1,12,13),0,0,(0,1,5),0,0,(0,5),0,0,0,
 ↪0,0,(0,2),0,0,8,0,(0,1),0,0,0,0,(1,7),0,0,3,(4,5),(5,6),7,0,(0,1),0,(3,7),1,4,0,4,3,1,
 ↪0,6,0,7,3,0,0,(3,10),0,1,0,0,0,0,0,6,(0,8),8,6,0,(0,8),0,4,0,2,(0,12),0,0,0,0,3,0,0,0,
 ↪2,0,0,0,3,0,6,0,0,0,0,(1,7,9),1
Kikuyu     0,0,7,(1,8),7,0,0,2,4,(2,8),3,3,(0,1),3,(3,7),3,0,2,1,(0,1),0,0,0,(1,3),0,0,0,
 ↪0,0,6,6,1,0,0,0,1,7,(0,1),0,4,4,(3,4),6,0,1,0,4,1,(1,9),0,4,0,1,0,1,0,(1,2),3,0,1,0,3,
 ↪0,0,0,0,0,0,(7,8,9),0,8,6,0,8,0,4,0,(3,4),1,6,2,(1,2),1,(1,2),0,1,0,(1,4),0,0,1,3,0,6,
 ↪0,1,0,0,6,3
Kiyombi    1,0,4,2,0,0,0,6,5,4,1,1,0,9,6,0,0,9,0,0,0,0,0,0,0,0,0,0,4,6,7,5,0,4,6,0,3,0,0,
 ↪3,2,6,3,4,0,1,9,1,8,0,0,4,2,2,3,2,5,3,1,0,6,7,3,0,0,0,0,0,5,4,5,6,0,6,0,7,0,13,5,5,0,0,
 ↪9,6,0,5,1,0,?,0,0,3,1,0,0,?,3,0,3,3
Lega       0,0,0,8,7,0,0,2,9,1,0,1,0,?,0,2,0,13,0,0,0,0,0,2,0,0,0,2,?,0,1,2,2,4,4,0,7,0,1,
 ↪3,4,3,7,0,1,0,11,1,2,0,6,0,1,0,7,0,8,3,0,0,10,4,2,1,0,0,0,0,5,1,8,6,0,2,0,0,0,11,(2,10,
 ↪11,13),5,0,0,2,0,0,3,7,0,1,0,0,3,0,0,0,0,1,0,0,(0,3)
Luganda    3,1,(1,7),7,1,(0,4),0,2,3,9,4,1,(5,8,9),1,(7,10,11),0,0,(3,12),0,0,4,0,0,0,(0,
 ↪2),(0,1),0,0,(0,1),7,0,3,3,0,(0,2),0,6,0,3,3,5,3,7,0,1,0,(2,6),1,3,2,7,2,1,0,6,0,7,3,0,
 ↪0,(0,1,2,10,11),0,1,2,0,1,0,2,1,3,(1,2,8),2,1,(1,9),0,4,(1,9),(6,7,8),3,(0,1),3,(3,4),
 ↪(0,3),0,0,2,0,0,0,0,(0,1),3,0,6,0,0,0,0,8,3
Ngombe     1,0,4,4,0,0,2,5,6,1,0,1,(4,11),(5,6),(5,13),(0,1),0,8,0,(2,3),2,0,(0,1),(0,6),
 ↪3,0,0,0,(2,3),2,2,0,0,(2,3,5),0,0,5,0,0,3,(2,4),1,(1,4),3,0,0,8,1,5,(0,1),(2,3),(0,5),
 ↪(0,5),(0,1),7,(0,1),4,1,2,2,(4,8),(2,5),(3,5),4,(0,2),(0,3),0,0,?,5,4,5,2,4,0,?,3,0,7,
 ↪5,4,0,7,0,0,6,2,0,?,0,0,(1,3),1,(1,6),(0,1),1,6,1,4,3
Ntomba     4,0,3,4,2,0,2,2,4,1,0,1,0,4,13,0,0,12,0,4,0,0,0,4,0,0,3,0,2,2,1,0,4,5,0,2,2,0,
 ↪0,(1,2,3),2,3,(2,4),2,0,0,8,1,6,1,2,5,5,0,7,1,4,3,0,0,0,6,4,4,0,2,0,0,2,(2,6),8,6,2,3,
 ↪0,2,2,0,8,5,3,0,6,6,0,(4,6),(3,4),0,3,0,0,3,1,0,0,2,0,0,3,3
Nyamwezi   0,1,7,2,7,1,0,2,7,6,0,1,5,2,8,0,0,5,0,0,0,0,0,0,(0,2),0,0,0,0,6,5,(1,4),(0,1),
 ↪4,(0,3),0,7,0,0,0,5,3,6,(0,6),1,0,5,1,10,0,4,(1,7),1,0,7,0,3,0,0,0,(0,10),0,0,2,0,0,0,
 ↪0,?,2,8,(1,6),0,8,0,(4,5),0,(5,12),0,7,0,0,10,0,0,0,0,0,0,0,0,3,0,(4,5),0,0,6,0,7,(4,5)
Nzadi      1,0,6,4,(2,4,5,6),3,(2,3),1,5,1,0,(1,4,5),0,(11,12),13,1,0,11,0,0,0,0,0,(0,4),
 ↪0,0,0,3,2,5,1,0,4,0,0,0,5,0,0,(3,5),2,6,(2,5),(2,5),0,2,9,1,(6,7),1,(0,1),4,?,(3,4),4,
 ↪2,(3,4),3,0,0,(0,7,9),(0,9),0,5,(0,3),2,0,1,5,4,(6,7),(4,6),0,(0,4),0,1,(6,8),(0,10),4,
 ↪(3,4,5),3,5,?,4,0,6,(6,8),0,4,(0,1,2),0,3,1,(0,1),0,0,5,0,1,3
Nzebi      0,0,5,6,3,0,0,7,6,5,1,1,2,10,6,0,0,10,2,0,1,0,0,5,0,0,4,3,2,4,3,0,?,6,5,0,4,0,
 ↪0,3,3,6,4,2,0,1,9,1,5,0,0,4,3,0,7,1,6,3,0,3,0,8,0,0,0,0,0,0,5,4,3,(0,6),0,7,0,7,5,13,0,
 ↪5,0,5,0,6,0,5,8,0,4,0,0,3,3,0,2,0,4,0,5,3
Swahili    0,1,7,3,8,2,1,(0,3),(1,2),7,0,2,(6,7),13,9,0,0,5,0,0,4,1,0,8,(0,1),0,1,0,0,1,5,
 ↪1,0,4,0,0,7,2,2,0,5,3,6,1,0,0,1,1,9,0,5,0,1,0,2,0,2,(0,3),4,0,0,0,0,6,0,0,0,0,4,9,8,6,
 ↪0,8,0,6,0,9,0,7,0,0,3,0,0,0,0,0,5,0,0,3,0,6,0,1,6,0,8,3
```

## 1.3 `lexedata` Manual

### 1.3.1 lexedata commands

You can access the Lexedata tools through commands in your terminal. Every command has the following general form:

> python -m lexedata.*package*.*command_name* [--*optional-argument* VALUE] [--*switch*] POSITIONAL ARGUMENTS

Elements in italics above need to be replaced depending on the exact command you are using. Elements in capital letters need to be replaced depending on the exact operation you want to perform on your dataset. Optional elements are enclosed in brackets. There could be multiple positional arguments, optional arguments and switches (with only a space as a separator). Positional arguments are not preceded by a hyphen and need to occur in strict order (if there are more than two of them). Optional arguments and switches are always preceded by two hyphens and they can occur in any order after the command. Optional arguments require a value (often there is a default value), while switches do not. Some optional arguments or switches have a short name of one letter in addition to their regular name and in this case they are preceded by one hyphen (e.g. to access the help of any command you can use the switch `--help` or `-h`). Many positional arguments and most optional arguments have default settings.

In order to maintain the integrity of the CLDF format, which is a relational database, lexedata scripts often have to operate on multiple files. It is therefore common that a lexedata command has as an optional argument the metadata file (defaulting to the Wordlist-metadata.json file of the current directory). Thus, the script operates on as many files of the directory containing the CLDF dataset as necessary for the operation at hand. For more information on the CLDF format and its files, see here.

Commands in Lexedata are organized in four packages: lexedata.importer[1], lexedata.edit, lexedata.report, and lexedata.exporter[1]. If a command name consists of more than one word, the words are separated by an underscore[1]. However, if optional arguments or switches consist of more than one word, the words are separated by a hyphen. If you need to replace an element in capital letters with a phrase including a space, enclose the whole phrase in quotes (`"MORE THAN ONE WORD"`).

Probably the most important thing to know before you get started with Lexedata is how to get help. This manual contains all available commands and describes their most common uses but it is not exhaustive as far as optional arguments and switches are concerned. It is highly recommended to first read the help of any new command you are thinking of using. You can access the help of any command by typing

> python -m lexedata.*package*.*command_name* --help

The help explains how the command is used, what it does and lists all the positional and optional arguments, along with their default values, if any. If you find the help confusing, or something is missing, do not hesitate to let us know by opening an issue on GitHub.

### 1.3.2 Importing data (lexedata.importer)

#### Importing a lexical dataset to CLDF from excel

Lexedata has different ways of creating a lexical dataset from raw data depending on the format the raw data are in. Currently, three different dataset formats are supported: "long format", "matrix", and "interleaved". The file format in all cases is assumed to be .xlsx. The "long format" assumes that every field in CLDF is a different column in your spreadsheet, and that each sheet of your spreadsheet is a different language. In other words, it assumes that you have a wordlist per language. The "matrix" format assumes that your spreadsheet is a comparative wordlist with different languages as different columns. You may have different kinds of information in each cell (e.g. forms and translations),

[1] In case you are wondering why the packages are called **importer** and **exporter**, not import and export: The word `import` has special status in Python, so it was not available as name of a package! The hyphen `-` is used as subtraction symbol in Python, so it also may not be part of a name.

as long as they are machine readable. The matrix importer can also import a separate xls spreadsheet with cognate sets *at the same time* that the forms are imported to a CLDF dataset. The "interleaved" format assumes that you have alternating rows of data and cognate coding. It is similar to the "matrix" format, with the addition that under every row there is an extra row with numerical codes representing cognate classes.

There are multiple types of information that lexedata can extract from excel files. A first distinction is between the content of the cells and the contents of a comment or note associated with the cell. Lexedata reads excel comments and stores them in a dedicated comment field when importing from a matrix format. However, it does not import comments from an interleaved or long-format dataset. Within the cell contents, there could be different types of information as well: e.g. more precise translations, sources, variant forms etc. Automatic separation of these different types of information depends on them being machine readable, i.e. they should be able to be automatically detected, because they are enclosed in different kinds of brackets. The "matrix" format importer script is the most sophisticated and customizable in terms of automatic separation of cell contents.

### Importing a lexical dataset using the "long" format

In order to import a lexical dataset in a "long" excel format you need to provide lexedata with a json file describing the CLDF dataset you want to build (see how to add a metadata file). The relevant command is:

```
python -m lexedata.importer.excel_long_format EXCEL
```

You can exclude individual sheets from being imported by using the option `--exclude-sheet SHEET`. The script can ignore missing or superfluous columns in case your excel file does not match exactly the description in the metadata file (see command help for more information).

The long format assumes that each type of information is in a separate column and the content of a cell cannot be separated further in different cells (however, fields including separators are properly recognized provided that you have described them in the json file). Excel comments (or notes) are not supported with this importer script and they will be ignored.

### Importing a lexical dataset using the "matrix" format

The matrix format importer is highly customizable, since many different types of information can be included in each cell, apart from the form itself. As long as the cells are machine readable, lexedata can extract most of the information and correctly assign it in different fields of a CLDF dataset. The matrix importer is customized through a special key in the metadata (json) file. At the same time as the importation of forms, cognate information can be imported to the CLDF dataset from a separate spreadsheet, where every row is a cognate set. The contents of the cells of this second cognatesets spreadsheet are assumed to be easily relatable to the forms (e.g. matching the forms and source).

For an example of the special key in the metadata file, see the smallmawetiguarani dataset. You can customize the two cell parsers (CellParser and CognateCellParser), as well as the conditions for a matching form between the two spreadsheets (check_for_match).

In order to import only a comparative wordlist file (without cognatesets), type

```
python -m lexedata.importer.excel_matrix EXCEL
```

If you want to import cognate sets from a separate spreadsheet, you can use the optional argument `--cogsets COGSET_EXCEL`.

#### Importing a lexical dataset using the "interleaved" format

In the interleaved format, even rows contain forms, while odd rows contain the associated cognate codes. The cognate codes are assumed to be in the same order as the forms and a cognate code is expected for every form (even if this results in a cognate code being present in a cell multiple times). In case your excel file contains question marks to indicate missing data, the importing script will ignore them. The interleaved importer is the only importing script in lexedata that does not require a metadata (.json) file. However, it also cannot perform any automatic treatment of information within the cells, apart from basic separation (using commas and semi-colons as separators). For example, if forms in the xlsx file are separated by ; and then some of the forms are followed by parentheses containing translations, each cell is going to be parsed according to the ; and everything in between will be parsed as the form (including the parentheses). You can edit further this dataset to separate the additional kinds of information by editing the forms.csv file that will be created. Note that any excel comments present in your file will be ignored. The resulting forms.csv file contains an empty comment field by default. In order to import a dataset of the "interleaved" format you should use the command

```
python -m lexedata.importer.excel_interleaved FILENAME.xlxs
```

where `FILENAME.xlsx` should be replaced with the name of the excel file containing the dataset. Only a `forms.csv` will be created, which contains a `Cognateset_ID` column with the cognate codes. This format is similar to the LingPy format. Note that for any further use of this CLDF dataset with lexedata, you need to add a metadata file.

#### Adding a new language/new data to an existing lexical dataset

The importation script using the long format can be used to add new data to an existing dataset, as in the case of adding an new variety or further lexical items for an existing variety (see importing a lexical dataset using the "long" format).

### 1.3.3 Editing a CLDF dataset (lexedata.edit)

The "edit" package includes a series of scripts to automate common targeted or batch edits in a lexical dataset. It also includes scripts that create links to [Concepticon(http://concepticon.clld.org) and integrate with LingPy.

If you need to do editing of raw data in your dataset (such as a transcription, translation, form comment etc), you need to do this manually. For `parameters.csv`, `forms.csv`, and `languages.csv`, you need to open the file in question in a spreadsheet editor, edit it, and save it again in the .csv format. For `cognates.csv` and `cognatesets.csv`, we recommend that you use the Cognate Table export (see export a Cognate Table). Whenever editing a cldf dataset, it is always a good idea to validate the dataset before and after (see CLDF validate) to make sure that everything went smoothly.

#### CLDF dataset structure and curation

#### How to add a metadata file (add_metadata)

If your CLDF dataset contains only a FormTable (the bare minimum for a CLDF dataset), you can automatically add a metadata (json) file using the command

```
python -m lexedata.edit.add_metadata
```

Lexedata will try to automatically detect CLDF roles for your columns (such as #form, #languageReference, #parameterReference, #comment, etc) and create a corresponding json file. We recommend that you inspect and manually adjust this json file before you proceed (see the metadata file). You can also use this command to obtain a starting metadata file for a new dataset. In this case, you can start from an empty FormTable that contains the columns you would like to include. The add_metadata command can be used also for LingPy output files, in order to obtain a CLDF dataset with metadata.

### How to add tables to your dataset (add_table)

Once you have a metadata file, you can add tables to your dataset (such as LanguageTable, ParameterTable) automatically. Such tables are required for most lexedata commands. The relevant command is

```
python -m lexedata.edit.add_table TABLE
```

The only table that cannot be added with this script is the CognateTable, which of course requires cognate judgements (see how to add a CognateTable.

### How to add a CognateTable (add_cognate_table)

The output of LingPy has cognate judgements as a column within the FormTable, rather than in a separate CognateTable as is the standard in CLDF. This is also the format of a FormTable generated when importing an "interleaved" dataset with Lexedata (see importing a lexical dataset using the "interleaved" format). In these cases, Lexedata can subsequently create an explicit CognateTable using the command

```
python -m lexedata.edit.add_cognate_table
```

Note that you have to indicate if your cognate judgement IDs are unique within the same concept or are valid across the dataset. In other words, if for every concept you have a cognateset with the code 1, then you need to indicate the option `--unique-id concept`, while if you have cross-concept cognate sets you need to indicate the option `--unique-id dataset`.

### How to normalize unicode (normalize_unicode)

Linguistic data come with a lot of special characters especially for the forms, but also for language names etc. Many of the input methods used for such datasets result in seemingly identical glyphs, which are not necessarily the same unicode character (and are therefore not considered identical by a computer) . Lexedata can normalize unicode across your dataset with the command

```
python -m lexedata.edit.normalize_unicode
```

You can find more info about what unicode normalization here.

### Workflow and tracking aid (add_status_column)

When developing and editing a comparative dataset for historical linguistics, you may need to keep track of operations, such as manual checks, input by collaborators etc. You may also wish to inspect manually some of the automatic operations of lexedata. To facilitate such tasks, you can add a "status column" in any of the CLDF tables using the command

```
python -m lexedata.edit.add_status_column
```

How you use status columns is largely up to you. You may manually keep track of your workflow in this column using specific codewords of your choice. Lexedata scripts that perform automatic operations that are not trivial (such as alignments, automatic cognate set detection) usually leave a message in the status column (which is customizable).

### Valid and transparent IDs (simplify_ids)

In a CLDF dataset, all IDs need to be unique and be either numeric or restricted alphanumeric (i.e. containing only leters, numbers and underscores). Lexedata command

```
python -m lexedata.edit.simplify_ids
```

verifies that all IDs in the dataset are valid and changes them appropriately if necessary. You can also choose to make your IDs transparent (option `--transparent`) so that you can easily tell what they correspond to. With transparent IDs, instead of a numeric ID, a form will have an ID consisting of the languageID and the parameterID: e.g. the word "main" in French would have the ID stan1290_hand.

### How to replace or merge IDs (replace_id and replace_id_column)

Sometimes you may need to replace an object's ID (e.g. language ID, concept ID, etc), e.g. if accidentally you have used the same ID twice. Lexedata can replace the id of an object and propagate this change in all tables where it is used as a foreign key (i.e. to link back to that object). The relevant command is

```
python -m lexedata.edit.replace_id TABLE ORIGINAL_ID REPLACEMENT_ID
```

If you intend to merge two IDs, e.g. if you decide to conflate two concepts because they are not distinct in the languages under study, or two doculects that you want to consider as one. you need to use the optional argument `--merge`. Keep in mind that lexedata cannot automatically merge two or more rows in the table in question, so if for example you merged two Language IDs, then you will have two rows in languages.csv with identical IDs. This will cause a warning if you try to validate your dataset (see CLDF validate). If you want to completely merge the rows, you need to do this by opening the corresponding csv in a spreadsheet or text editor.

In case you want to replace an entire ID column of a table, then you need to add the new intended ID column to the table and use the command

```
python -m lexedata.edit.replace_id_column TABLE REPLACEMENT
```

### Operations on FormTable

### Merge polysemous forms (merge_homophones)

In large datasets, there may be identical forms within the same language, corresponding to homophonous or polysemous words. You can use

```
python -m lexedata.report.homophones
```

to detect identical forms present in the dataset (see detect potential homophonous or polysemous forms). Once you decide which forms are in fact polysemous, you can use

```
python -m lexedata.edit.merge_homophones MERGE_FILE
```

in order to merge them into one form with multiple meanings. The MERGE_FILE contains the forms to be merged, in the same format as the output report from the `lexedata.report.homophones` command. There are multiple merge functions available for the different metadata associated with forms (e.g. for comments the default merge function is concatenate, while for sources it is union). If you need to modify the default behavior of the command you can use the optional argument `--merge COLUMN:MERGER`, where COLUMN is the name of the column in your dataset and MERGER is the merge function you want to use (from a list of functions that can be found in the help).

### Segment forms (add_segments)

In order to align forms to find correspondence sets and for automatic cognate detection, you need to segment the forms included in your dataset. Lexedata can do this automatically using CLTS. To use this functionality type

```
python -m lexedata.edit.add_segments TRANCRIPTION_COLUMN
```

where transcription column refers to the column that contains the forms to be segmented (the #form column by default). A column "Segments" will be added to your FormTable. The segmenter makes some educated guesses and automatic corrections regarding segments (e.g. obviously wrong placed tiebars for affricates, non-IPA stress symbols, etc). All these corrections are listed in the segmenter's report for you to review. You may choose to apply these corrections to the form column as well, using the switch `--replace_form`.

### Operations on ParameterTable (concepts)

### Linking concepts to Concepticon (add_concepticon)

Lexedata can automatically link the concepts present in a dataset with concept sets in the Concepticon. The relevant command is

```
python -m lexedata.edit.add_concepticon
```

Your ParameterTable will now have a new column: Concepticon ID, with the corresponding ID of a concept set in Concepticon. We recommend that you manually inspect these links for errors. In order to facilitate this task, you can also add columns for the concept set name (`--add_concept_set_name`) and the concepticon definition (`--add_definitions`). Finally, if your ParameterTable contains a Status Column (see workflow and tracking aid), any links to the Concepticon will be tagged as automatic, or you can define a custom message using `--status_update "STATUS UPDATE"`.

### Operations on CognateTable (judgements) and CognatesetTable

### Adding central concepts to cognate sets (add_central_concepts)

If you are using cross-concept cognate sets, and you want to assign each cognate set to a certain concept (e.g. for the purposes of assigning absences in root presence coding), you can use lexedata to automatically add central concepts to your cognate sets. The central concept of each cognateset will be used as a proxy for assigning absences: If the central concept is attested in a language with a different root, then the root in question will be coded as absent for this language (see the glossary for more explanations for central concepts, coding methods and absence heuristics for root presence coding).

The central concept of each cognate set is determined by the CLICS graph of all the concepts associated with this cognate set (this requires having your concepts linked to the Concepticon). The concept with the highest centrality in the CLICS graph will be retained as the central concept. In the absence of Concepticon links, the central concept is the most common concept. In order to add central concepts to your dataset, type

```
python -m lexedata.edit.add_central_concepts
```

. This will add a #parameterReference column to your CognatesetTable containing the central concept. You can of course manually review and edit the central concepts. If you rely on central concepts for coding and you heavily edit your cognate judgements, consider re-assigning central concepts with the switch `--overwrite`.

#### Adding trivial cognate sets (add_singleton_cognatesets)

Depending on your workflow, you may not have assigned forms to trivial (singleton) cognate sets, where they would be the only members. This could also be true for loanwords that are products of separate borrowing events, even if they have the same source. You can automatically assign any form that is not already in a cognate set to a trivial cognate set of one member (a singleton cognate set) using the command

```
python -m lexedata.edit.add_singleton_cognate_sets
```

If you want to be careful about morphology, you can tell this script to create singletons for every uncoded set of segments by using

```
python -m lexedata.edit.add_singleton_cognate_sets --by-segment
```

This will create a separate cognate set for every contiguous slice of segments that are not in any cognate set yet, eg. one for the prefix and a separate one for the suffix of a word with a stem that is already coded.

#### Merging cognate sets

You can write a file listing cognate sets to be merged into one and feed it to

```
python -m lexedata.edit.merge_cognate_sets MERGEFILE
```

which bulk-merges these cognatesets. The main use of this command is to merge cognatesets found to be strongly overlapping by the nonconcatenative morphemes report.

### 1.3.4 Reporting and checking data integrity (lexedata.report)

The report package contains scripts that check for data integrity and generate reports for your dataset. You can use these reports to identify potential problems in the dataset, to track your progress, or to report statistics in publications (e.g. coverage for each language in a dataset).

#### Language coverage

You can obtain various statistics related to coverage (how many concepts have corresponding forms in each language) with the command

```
python -m lexedata.report.coverage
```

Among others, you can find which languages have corresponding forms for specific concepts, which languages have at least a given coverage percentage etc. NA forms (that correspond to concepts that do not exist in a particular language) by default count towards coverage, while missing forms don't. You can customize the treatment of missing and NA forms with the optional argument `--missing`.

### Segment inventories

You can get a report on all segments used for each language and their frequency in the dataset by typing

```
python -m lexedata.report.segment_inventories
```

This can be useful to locate rare or even erroneous transcriptions, non-standard IPA symbols etc. You can subset the report to one or a smaller number of languages for clarity using `--languages`.

### Detect potential homophonous or polysemous forms

In large datasets, you may have identical forms associated with different concepts. This could be the case because there are homophonous, unrelated forms, or because there is in fact one underlying polysemous form. Lexedata can help you detect potential homophones or polysemies by using the command

```
python -m lexedata.report.homophones
```

The output of this command is a list of all groups of identical forms in the data and their associated concepts, along with the information if the associated concepts are connected in CLICS or not (if your concepts are linked to Concepticon, see linking concepts to Concepticon). You can choose to merge the polysemous forms, so you have one form associated to multiple concepts. In order to perform this operation, edit the output file of `lexedata.report.homophones`, so that only the groups of forms that are to be merged remain and then use `lexedata.edit.merge_homophones` (see merge polysemous forms).

### Non-concatenative morphology

You can get a detailed report on potentially non-concatenative morphemes (segments that belong to more than one cognate sets, or cognate sets that refer to segments in a form which don't simply follow each other, due to infixes/circumfixes or metathesis) by running

```
python -m lexedata.report.nonconcatenative_morphemes
```

For a report on cognate judgements more focused on structural integrity, see cognate judgements.

The `nonconcatenative_morphemes` command outputs a report of clusters of cognate sets that have an overlap of 50% in at least one form. As such, it indicates that these cognate sets may be candidates that could be merged. Like the segment inventories report, you can output this report to a file and edit it before feeding it into the cognateset merger command. This command has a very similar interface to the homophones merger, and in fact a main use case is

1. Detecting homophones
2. Merging homophones into polysemous forms)
3. Detecting the pairs of cognate sets that now both point to the same newly-merged polysemous form, using this script
4. Merging those cognatesets

#### Cognate judgements

The cognate judgement report checks for issues involving cognate judgements, the segment slice column, the referenced segments and the alignment. For example, it checks if the referenced segments match what is present in the alignment and are contiguous (including identically looking but underlyingly different unicode characters), if the segment slice is valid based on the length of the form, and if the length of all alignments in a cognateset match. It also checks that missing (“”) and NA (“-”) forms are not assigned to any cognate set. In order to obtain the judgements report, you can use the command

```
python -m lexedata.report.judgements
```

This report is part of the checks that the extended CLDF validate executes, because it focuses on potential issues in the data model (eg. ‘alignments’ of different length, which makes them invalid as alignments). If you want additionally to check for cognatesets that contain non-contiguous segments (eg. because they skip an infix), you can use the switch `--strict`; this switch is not available through `extended_cldf_validate`. A more extensive report about the structure of segments inside cognate sets can be obtained for the non-concatenative morphology report.

#### CLDF validate

Before and after a variety of operations with lexedata, and especially after manual editing of raw data, it is highly recommended to validate your dataset so you can catch and fix any inconsistencies. You can do a basic validation using the relevant command in the `pycldf` package, or an extended one with lexedata. For the basic validation, type: `cldf validate METADATA`, where `METADATA` stands for the metadata (json) file of your dataset. For the extended cldf validation, type

```
python -m lexedata.report.extended_cldf_validate
```

The extended validation includes all operations of the basic pycldf command, but also checks further potential issues related to cognate judgements, unicode normalization, internal references, etc.

#### Filter dataset

The `filter` command gives you the possibility to filter any table of your dataset according to a particular column using regular expressions. You can use this command to output a subset of the dataset to a file and use it as input for further commands in lexedata (in particular for the subsetting operations supported by various commands) or other downstream analyses. The command is

```
python -m lexedata.report.filter COLUMN FILTER [TABLE]
```

, where `COLUMN` is the column to be filtered, the `FILTER` the expression to filter by and `TABLE` the table of the dataset that you want to filter. For more details, refer to the command’s help.

### 1.3.5 Exporting data (lexedata.exporter)

The `exporter` package contains two types of scripts.

1. Scripts which export the data to make it available for editing, through an xlsx cognate matrix or through Edictor and LingPy. These export scripts come with a corresponding importer script in `lexedata.importer`, so that data can be exported, edited externally, and re-imported.
2. Scripts which export the data for other use. The matrix exporter generates a comparative word list that can be edited for inclusion in a publication, while the phylogenetics exporter generates character sequences that can be used in phylogenetics software.

### The xlsx cognate matrix loop

Lexedata offers the possibility to edit and annotate within- or across-concept cognate sets in a spreadsheet format using the spreadsheet editor of your choice (we have successfully used Google sheets, LibreOffice Calc and Microsoft Excel, but it should work in principle on any spreadsheet editor). In order to use this functionality, you export your cognate judgements to a cognate matrix (in xlsx format) which you can edit, and then re-import this modified cognate matrix back into your cldf dataset. This process will overwrite previously existing cognate sets and cognate judgements, as well as any associated comments (CognateTable comments and CognatesetTable comments).

**Important:** You cannot edit the raw data (forms, translation, form comments, etc.) through this process. To edit raw data, see editing a CLDF dataset.

It is always a good idea to validate your dataset before and after any edits to make sure that everything is linked as it should in the cldf format. You can use `cldf validate METADATA_FILE` or

```
python -m lexedata.report.extended_cldf_validate
```

for a more thorough check (see also CLDF validate). The latter is particularly useful, because it checks a lot of additional assumptions about cognate judgements.

In order to export an xlsx cognate matrix, you should type

```
python -m lexedata.exporter.cognates FILENAME.xlsx
```

The cognate matrix will be written to an excel file with the specified name. There are optional arguments to sort the languages and the cognate sets in this table, as well as to assign any forms not currently in a cognate set to automatic singleton cognate sets (see command help for more information; see also: lexedata.edit.add_singleton_cognatesets).

You can open and edit the cognate matrix in the spreadsheet editor of your choice. You can update cognate sets, cognate judgements and associated metadata (in particular segment slices, alignments, CognateTable comments, and CognatesetTable comments). This workflow can allow specialists to work on the cognacy judgements in a familiar format (such as excel), or allow a team to work collaboratively on Google sheets, while at the same time keeping the dataset in the standard cldf format. You just need to remember that you need to preserve the general format in order to reimport your modified cognate sets into your cldf dataset.

### Re-importing cognate sets from a cognate matrix

You can reimport the cognate matrix xlsx spreadsheet by typing:

```
python -m lexedata.importer.cognates FILENAME.xlsx
```

This will re-generate the CognateTable with the cognate judgements, as well as the CognatesetTable, in your CLDF dataset according to the cognate matrix.

### The Edictor editing loop

The 'etymological dictionary editor' EDICTOR provides access to another way of modifying cognate sets and alignments, with interfaces for partial cognate annotation, a graphical alignment editor and summaries of sound correspondences. Like the LingPy Python library for historical linguistics, Edictor uses a format that is similar to CLDF's form table, but it also includes cognate judgements and alignments. In addition, it differs in several format choices that look minor to the human eye, but matter greatly for automatic processing by a computer.

To work with CLDF data in Edictor or LingPy, Lexedata provides a pair of an exporter and importer script

```
python -m lexedata.exporter.edictor FILENAME.tsv
python -m lexedata.importer.edictor FILENAME.tsv
```

which allow the exporting of a CLDF dataset to Edictor's TSV format and importing the data back after editing.

**Important:** This loop is brittle.

- Edictor can sometimes halt processing without warning, in particular on large datasets or datasets with special assumptions, such as non-concatenative morphemes, polysemous forms, or multi-line comments.
- Such datasets also quickly become unwieldy in Edictor.
- In order to support edits using Edictor better, the exporter allows the restriction of the dataset to a subset of languages and concepts. The importer does its best to try integrating the changes made in Edictor back to the dataset. However, there are a lot of special cases which are insufficiently tested.

Be careful with the Edictor loop. Re-import often, commit often to be able to undo changes, and don't hesitate to raise an issue concerning undesired behavior.

### Export a comparative wordlist

In particular for the supplementary material of a publication, it is often useful to provide a comparative word list in a human-readable layout in addition to a deposited archive of the CLDF dataset (eg. on Zenodo or figshare). The command

```
python -m lexedata.exporter.matrix FILENAME.xlsx
```

helps preparing such a layout. It generates an Excel sheet which contains one column per language and one row per concept. Optionally, forms can be hyperlinked to their corresponding page or anchor in a web-browsable database version of the dataset, while concepts can be output with their Concepticon links. Also, you can specify a subset of languages and concepts to include, eg. only the *primary concept*s.

### Export coded data for phylogenetic analyses

Lexedata is a powerful tool to prepare linguistic data for phylogenetic analyses. The command `python -m lexedata.exporter.phylogenetics` can be used to export a cldf dataset containing cognate judgements as a coded matrix for phylogenetic analyses to be used by standard phylogenetic software, (such as BEAST2, MrBayes or revBayes). Different formats are supported, such as nexus, csv, a beast-friendly xml format, as well as a raw alignment format (similar to the FASTA format used in bioinformatics). Lexedata also supports different coding methods for phylogenetic analyses: *root-meaning coding*, cross-concept cognate sets AKA *root presence coding*, and *multistate coding*.

Finally, you can use Lexedata to filter and export a portion of your dataset for phylogenetic analyses, e.g. if some languages or concepts are not fully coded yet, or if you want to exclude specific cognate sets that are not reviewed yet.

## 1.4 Internal API reference

*Lexedata* is a set of tools for managing, editing, and annotating large lexical datasets in CLDF.

Lexedata is open access software in development. Please report any problems and suggest any improvements you would like to see by opening an issue on the Lexedata GitHub repository

### 1.4.1 lexedata.edit package

#### Submodules

#### lexedata.edit.add_central_concepts module

`lexedata.edit.add_central_concepts.`**`add_central_concepts_to_cognateset_table`**(*dataset: pycldf.dataset.Dataset, add_column: bool = True, overwrite_existing: bool = True, logger: logging.Logger = <Logger lexedata (INFO)>, status_update: typing.Optional = None*) → pycldf.dataset.Dataset

`lexedata.edit.add_central_concepts.`**`central_concept`**(*concepts: Counter[str], concepts_to_concepticon: Mapping[str, int], clics: Optional[networkx.classes.graph.Graph]*)

Find the most central concept among a weighted set.

If the concepts are not linked to CLICS through Concepticon references, we can only take the simple majority among the given concepts.

```
>>> concepts = {"woman": 3, "mother": 4, "aunt": 3}
>>> central_concept(concepts, {}, None)
'mother'
```

However, if the concepts can be linked to the CLICS graph, centrality actually can be defined using that graph.

```
>>> concepticon_mapping = {"arm": 1673, "five": 493, "hand": 1277, "leaf": 628}
>>> central_concept(
...   collections.Counter(["arm", "hand", "five", "leaf"]),
...   concepticon_mapping,
...   load_clics()
... )
'hand'
```

When counts and concepticon references are both given, the value with the maximum product of CLICS centrality and count is returned. If the concepts do not form a connected subgraph in CLICS (eg. 'five', 'hand', 'lower arm', 'palm of hand' – with no attested form meaning 'arm' to link them), only centrality within the disjoint subgraphs is considered, so in this example, 'hand' would be considered the most central concept.

`lexedata.edit.add_central_concepts.`**`concepts_to_concepticon`**(*dataset: pycldf.dataset.Wordlist*) → Mapping[str, int]

`lexedata.edit.add_central_concepts.`**`connected_concepts`**(*dataset: pycldf.dataset.Wordlist*) → Mapping[str, Counter[str]]

For each cognate set it the dataset, check which concepts it is connected to.

```
>>>
```

`lexedata.edit.add_central_concepts.`**`load_concepts_by_form`**(*dataset: pycldf.dataset.Dataset*) → Dict[str, Sequence[str]]

Look up all concepts for each form, and return them as dictionary.

`lexedata.edit.add_central_concepts.`**`reshape_dataset`**(*dataset: pycldf.dataset.Wordlist*, *add_column: bool = True*) → pycldf.dataset.Dataset

## lexedata.edit.add_cognate_table module

Add a CognateTable to the dataset.

If the dataset has a CognateTable, do nothing. If the dataset has no cognatesetReference column anywhere, add an empty CognateTable. If the dataset has a cognatesetReference in the FormTable, extract that to a separate cognateTable, also transferring alignments if they exist. If the dataset has a cognatesetReference anywhere else, admit you don't know what is going on and die.

`lexedata.edit.add_cognate_table.`**`add_cognate_table`**(*dataset: pycldf.dataset.Wordlist*, *split: bool = True*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → None

## lexedata.edit.add_concepticon module

Guess which Concepticon concepts the entries in the ParameterTable refer to.

The full list of available gloss languages uses the ISO 693-1 two-letter codes and can be found on https://github.com/concepticon/concepticon-data/tree/master/mappings (or in the mappings/ folder of your local Concepticon catalog installation).

Fill the Concepticon_ID (or generally, #concepticonReference) column of the dateset's ParameterTable with best guesses for Concepticon IDs, based on gloss columns in potentially different languages.

`lexedata.edit.add_concepticon.`**`add_concepticon_definitions`**(*dataset: pycldf.dataset.Dataset*, *column_name: str = 'Concepticon_Definition'*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → None

`lexedata.edit.add_concepticon.`**`add_concepticon_names`**(*dataset: pycldf.dataset.Wordlist*, *column_name: str = 'Concepticon_Gloss'*)

`lexedata.edit.add_concepticon.`**`add_concepticon_references`**(*dataset: pycldf.dataset.Wordlist*, *gloss_languages: Mapping[str, str]*, *status_update: Optional[str]*, *overwrite: bool = False*) → None

Guess Concepticon links for a multilingual Concept table.

Fill the concepticonReference column of the dateset's ParameterTable with best guesses for Concepticon IDs, based on gloss columns in different languages.

**Parameters**

- **dataset** (*A pycldf.Wordlist with a concepticonReference column in its*) – ParameterTable
- **gloss_languages** (*A mapping from ParameterTable column names to ISO-639-1*) – language codes that Concepticon has concept lists for (eg. en, fr, de, es, zh, pt)
- **status_update** (*String written to Status_Column of #parameterTable if provided*) –
- **overwrite** (*Overwrite existing Concepticon references*) –

lexedata.edit.add_concepticon.**create_concepticon_for_concepts**(*dataset: pycldf.dataset.Dataset*, *language: Sequence[Tuple[str, str]]*, *concepticon_glosses: bool*, *concepticon_definition: bool*, *overwrite: bool*, *status_update: Optional[str]*)

lexedata.edit.add_concepticon.**equal_separated**(*option: str*) → Tuple[str, str]

## lexedata.edit.add_metadata module

Adds a metadata.json file automatically starting from a forms.csv file. Lexedata can guess metadata for a number of columns, including, but not limited to, all default CLDF properties (e.g. Language, Form) and CLDF reference properties (e.g. parameterReference). We recommend that you check the metadata file created and adjust if necessary.

## lexedata.edit.add_segments module

Segment the form.

Take a form, in a phonemic transcription compatible with IPA, and split it into phonemic segments, which are written back to the Segments column of the FormTable. Segmentation essentially uses CLTS, including diphthongs and affricates.

For details on the segmentation procedure, see the manual.

class lexedata.edit.add_segments.**ReportEntry**(*count: int = 0*, *comment: str = ''*)

Bases: object

**comment: str**

**count: int**

class lexedata.edit.add_segments.**SegmentReport**(*sounds: MutableMapping[str, lexedata.edit.add_segments.ReportEntry] = NOTHING*)

Bases: object

**sounds: MutableMapping[str,** *lexedata.edit.add_segments.ReportEntry***]**

`lexedata.edit.add_segments.`**`add_segments_to_dataset`**(*dataset: pycldf.dataset.Dataset*, *transcription: str*, *overwrite_existing: bool*, *replace_form: bool*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

`lexedata.edit.add_segments.`**`cleanup`**(*form: str*) → str

```
>>> cleanup("dummy;form")
'dummy'
>>> cleanup("dummy,form")
'dummy'
>>> cleanup("(dummy)")
'dummy'
>>> cleanup("dummy-form")
'dummy+form'
```

`lexedata.edit.add_segments.`**`segment_form`**(*formstring: str*, *report: lexedata.edit.add_segments.SegmentReport*, *system=<pyclts.transcriptionsystem.TranscriptionSystem object>*, *split_diphthongs: bool = True*, *context_for_warnings: str = ''*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[pyclts.models.Symbol]

Segment the form.

First, apply some pre-processing replacements. Forms supplied contain all sorts of noise and lookalike symbols. This function comes with reasonable defaults, but if you encounter other problems, or you actually want to be strict about IPA transcriptions, pass a dictionary of your choice as *pre_replace*.

Then, naïvely segment the form using the IPA tokenizer from the *segments* package. Check each returned segment to see whether it is valid according to CLTS's BIPA, and if not, try to fix some issues (in particular pre-aspirated or pre-nasalized consonants showing up as post-aspirated resp. post-nasalized vowels, which BIPA does not accept).

```
>>> [str(x) for x in segment_form("iund", report=SegmentReport())]
['i', '', 'u', 'n', 'd', '']
>>> [str(x) for x in segment_form("mokoi", report=SegmentReport())]
['m', 'o', 'k', 'o', 'i']
>>> segment_form("panonootsíkoú", report=SegmentReport())
[<pyclts.models.Consonant: voiceless bilabial stop consonant>, <pyclts.models.
↪Vowel: unrounded open front vowel>, <pyclts.models.Consonant: devoiced voiced␣
↪alveolar nasal consonant>, <pyclts.models.Vowel: rounded close-mid back vowel>,
↪<pyclts.models.Consonant: voiced alveolar nasal consonant>, <pyclts.models.Vowel:␣
↪rounded close-mid back vowel>, <pyclts.models.Vowel: rounded close-mid back ...␣
↪vowel>, <pyclts.models.Consonant: voiceless alveolar sibilant affricate consonant>
↪, <pyclts.models.Vowel: unrounded close front ... vowel>, <pyclts.models.
↪Consonant: voiceless velar stop consonant>, <pyclts.models.Vowel: long rounded␣
↪close-mid back vowel>, <pyclts.models.Consonant: voiceless glottal stop consonant>
↪, <pyclts.models.Vowel: rounded close back ... vowel>]
```

### lexedata.edit.add_singleton_cognatesets module

Add trivial cognatesets

Make sure that every segment of every form is in at least one cognateset (there can be more than one, eg. for nasalization), by creating singleton cognatesets for streaks of segments not in cognatesets.

`lexedata.edit.add_singleton_cognatesets.`**`create_singletons`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], status: typing.Optional[str] = None, by_segment: bool = False, logger: logging.Logger = <Logger lexedata (INFO)>*) → Tuple[Sequence[*lexedata.types.CogSet*], Sequence[*lexedata.types.Judgement*]]

Create singleton cognate judgements for forms that don't have cognate judgements.

Depending on by_segment, singletons are created for every range of segments that is not in any cognate set yet (True) or just for every form where no segment is in any cognate sets (False).

`lexedata.edit.add_singleton_cognatesets.`**`uncoded_forms`**(*forms: Iterable[*lexedata.types.Form*], judged: Container[lexedata.types.Form_ID]*) → Iterator[Tuple[lexedata.types.Form_ID, range]]

Find the uncoded forms, and represent them as segment slices.

```
>>> list(uncoded_forms([
...   {"id": "f1", "form": "ex", "segments": list("ex")},
...   {"id": "f2", "form": "test", "segments": list("test")},
... ], {"f1"}))
[('f2', range(0, 4))]
```

`lexedata.edit.add_singleton_cognatesets.`**`uncoded_segments`**(*segment_to_cognateset: typing.Mapping[lexedata.types.Form_ID, typing.List[typing.Set[lexedata.types.Cognateset_ID]]], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterator[Tuple[lexedata.types.Form_ID, range]]

Find the slices of uncoded segments.

```
>>> list(uncoded_segments({"f1": [{}, {}, {"s1"}, {}]}))
[('f1', range(0, 2)), ('f1', range(3, 4))]
```

### lexedata.edit.add_status_column module

`lexedata.edit.add_status_column.`**`add_status_column_to_table`**(*dataset: pycldf.dataset.Dataset*, *table_name: str*) → None

`lexedata.edit.add_status_column.`**`status_column_to_table_list`**(*dataset: pycldf.dataset.Dataset*, *tables: List[str]*) → pycldf.dataset.Dataset

### lexedata.edit.add_table module

Creates an empty table with all the references to that table found in the dataset.

This script can be used to add LanguageTable, CognatesetTable, and ParameterTable (i.e. the table with the concepts). For a CognateTable, see the help of lexedata.edit.add_cognate_table.

### lexedata.edit.align module

Automatically align morphemes within each cognateset.

If possible, align using existing lexstat scorer.

`lexedata.edit.align.`**`align`**(*forms*)

‘Align’ forms by adding gap characters to the end.

TODO: This is DUMB. Write a function that does this more sensibly, using LexStat scorers where available.

`lexedata.edit.align.`**`aligne_cognate_table`**(*dataset: pycldf.dataset.Dataset*, *status_update: Optional[str] = None*)

### lexedata.edit.change_id_column module

`lexedata.edit.change_id_column.`**`rename`**(*ds*, *old_values_to_new_values*, *logger: logging.Logger*, *status_update: Optional[str]*)

`lexedata.edit.change_id_column.`**`replace_column`**(*dataset: pycldf.dataset.Dataset, original: str, replacement: str, column_replace: bool, smush: bool, status_update: typing.Optional[str], logger: logging.Logger = <Logger lexedata (INFO)>*) → None

`lexedata.edit.change_id_column.`**`substitute_many`**(*row*, *columns*, *old_values_to_new_values*, *status_update: Optional[str]*)

### lexedata.edit.clean_forms module

Clean forms.

Move comma-separated alternative forms to the variants column. Move elements in brackets to the comments if they are separated from the forms by whitespace; strip brackets and move the form with brackets to the variants if there is no whitespace separating it.

This is a rough heuristic, but hopefully it helps with the majority of cases.

**exception** `lexedata.edit.clean_forms.`**`Skip`**(*message*)

Bases: `Exception`

Mark this form to be skipped.

`lexedata.edit.clean_forms.`**`clean_forms`**(*table: typing.Iterable[lexedata.edit.clean_forms.R], form_column_name='form', variants_column_name='variants', split_at=[',', ';'], split_at_and_keep=['~'], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterator[lexedata.edit.clean_forms.R]

Split all forms that contain separators into form+variants.

```
>>> for row in clean_forms([
...   {'F': 'a ~ æ', 'V': []},
...   {'F': 'b-, be-', 'V': ['b-']}],
...   "F", "V"):
...   print(row)
{'F': 'a', 'V': ['~æ']}
{'F': 'b-', 'V': ['b-', 'be-']}
```

`lexedata.edit.clean_forms.`**`treat_brackets`**(*table: typing.Iterable[lexedata.edit.clean_forms.R], form_column_name='form', variants_column_name='variants', comment_column_name='comment', bracket_pairs=[('(', ')')], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterator[lexedata.edit.clean_forms.R]

Make sure forms contain no brackets.

```
>>> for row in treat_brackets([
...   {'F': 'a(m)ba', 'V': [], 'C': ''},
...   {'F': 'da (dialectal)', 'V': [], 'C': ''},
...   {'F': 'tu(m) (informal)', 'V': [], 'C': '2p'}],
...   "F", "V", "C"):
...   print(row)
{'F': 'amba', 'V': ['aba'], 'C': ''}
{'F': 'da', 'V': [], 'C': '(dialectal)'}
{'F': 'tum', 'V': ['tu'], 'C': '2p; (informal)'}
```

Skipping works even when it is noticed only late in the process.

```
>>> for row in treat_brackets([
...   {'F': 'a[m]ba (unbalanced', 'V': [], 'C': ''},
...   {'F': 'tu(m) (informal', 'V': [], 'C': ''}],
...   "F", "V", "C", [("[", "]"), ("(", ")")]):
...   print(row)
{'F': 'a[m]ba (unbalanced', 'V': [], 'C': ''}
{'F': 'tu(m) (informal', 'V': [], 'C': ''}
```

`lexedata.edit.clean_forms.`**`unbracket_single_form`**(*form*, *opening_bracket*, *closing_bracket*)

Remove a type of brackets from a form.

Return the modified form, the variants (i.e. a list containing the form with brackets), and all comments (bracket parts that were separated from the form by whitespace)

```
>>> unbracket_single_form("not in here anyway", "(", ")")
('not in here anyway', [], [])
```

```
>>> unbracket_single_form("da (dialectal)", "(", ")")
('da', [], ['(dialectal)'])
```

```
>>> unbracket_single_form("da(n)", "(", ")")
('dan', ['da'], [])
```

```
>>> unbracket_single_form("(n)da(s) (dialectal)", "(", ")")
('ndas', ['da', 'das', 'nda'], ['(dialectal)'])
```

### lexedata.edit.detect_cognates module

Similarity code tentative cognates in a word list and align them

`lexedata.edit.detect_cognates.`**`clean_segments`**(*segment_string: List[str]*) → Iterable[pyclts.models.Symbol]

Reduce the row's segments to not contain empty morphemes.

This function removes all unknown sound segments (/0/) from the segments string it is passed, and removes empty morphemes by collapsing subsequent morpheme boundary markers (_#◦+→←) into one.

```
>>> segments = "+ _ t a + 0 + a t"
>>> c = clean_segments(segments)
>>> [str(s) for s in c]
['t', 'a', '+', 'a', 't']
```

`lexedata.edit.detect_cognates.`**`cognate_code_to_file`**(*metadata: pathlib.Path*, *ratio: float*, *soundclass: str*, *cluster_method: str*, *threshold: float*, *initial_threshold: float*, *gop: float*, *mode: str*, *output_file: pathlib.Path*) → None

`lexedata.edit.detect_cognates.`**`filter_function_factory`**(*dataset:* lexedata.types.Wordlist) → Callable[[Dict[str, Any]], bool]

`lexedata.edit.detect_cognates.`**`sha1`**(*path*)

### lexedata.edit.merge_cognate_sets module

Read a homophones report (an edited one, most likely) and merge all pairs of form in there.

Different treatment for separated fields, and un-separated fields Form variants into variants? Make sure concepts have a separator What other columns give warnings, what other columns give errors?

*Optionally*, merge cognate sets that get merged by this procedure.

`lexedata.edit.merge_cognate_sets.`**`merge_cogsets`**(*data: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], mergers: typing.Mapping[str, typing.Callable[[typing.Sequence[lexedata.edit.merge_homophones.C], typing.Optional[lexedata.types.Form]], typing.Optional[lexedata.edit.merge_homophones.C]]], cogset_groups: typing.MutableMapping[lexedata.types.Cognateset_ID, typing.Sequence[lexedata.types.Cognateset_ID]], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[*lexedata.types.CogSet*]

Merge cognate sets in a dataset.

TODO: Construct an example that shows that the order given in *cogset_groups* is maintained.

**Changes cogset_groups:** Groups that are skipped are removed

`lexedata.edit.merge_cognate_sets.`**`merge_group`**(*cogsets: typing.Sequence[lexedata.types.CogSet], target: lexedata.types.CogSet, mergers: typing.Mapping[str, typing.Callable[[typing.Sequence[lexedata.edit.merge_homophones.C], typing.Optional[lexedata.types.Form]], typing.Optional[lexedata.edit.merge_homophones.C]]], dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], logger: logging.Logger = <Logger lexedata (INFO)>*) → *lexedata.types.CogSet*

Merge one group of cognate sets.

The target is assumed to be already included in the forms.

## lexedata.edit.merge_homophones module

Read a homophones report (an edited one, most likely) and merge all pairs of form in there.

Different treatment for separated fields, and un-separated fields Form variants into variants? Make sure concepts have a separator What other columns give warnings, what other columns give errors?

*Optionally*, merge cognate sets that get merged by this procedure.

**`exception`** `lexedata.edit.merge_homophones.`**`Skip`**

Bases: `Exception`

Skip this merge, leave all forms as expected.

This is not an Error! It is more akin to StopIteration.

`lexedata.edit.merge_homophones.`**`cancel_and_skip`**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

If entries differ, do not merge this set of forms.

```
>>> cancel_and_skip([])
```

```
>>> cancel_and_skip([1, 2])
Traceback (most recent call last):
...
    raise Skip
lexedata.edit.merge_homophones.Skip
```

```
>>> cancel_and_skip([1, 1])
1
```

`lexedata.edit.merge_homophones.`**`concatenate`**(*sequence: Sequence[lexedata.edit.merge_homophones.C]*, *target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

Concatenate all entries, even if they are identical, in the given order.

Strings are concatenated using '; ' as a separator. Other iterables are flattened.

```
>>> concatenate([[1, 2], [2, 4]])
[1, 2, 2, 4]
>>> concatenate([["a", "b"], ["c", "a"]])
['a', 'b', 'c', 'a']
>>> concatenate(["a", "b"])
'a; b'
```

```
>>> concatenate([]) is None
True
>>> concatenate([[1, 2], [2, 4]])
[1, 2, 2, 4]
>>> concatenate([None, [1], [3]])
[1, 3]
>>> concatenate([[1, 1], [2]])
[1, 1, 2]
>>> concatenate([["a", "b"], ["c", "a"]])
['a', 'b', 'c', 'a']
>>> concatenate([None, "a", "b"])
'; a; b'
>>> concatenate(["a", "b", "a", ""])
'a; b; a; '
>>> concatenate(["a", "b", None, "a"])
'a; b; ; a'
>>> concatenate(["a", "b", "a; c", None])
'a; b; a; c; '
```

`lexedata.edit.merge_homophones.`**`constant_factory`**(*c: lexedata.edit.merge_homophones.C*) → Callable[[Sequence[lexedata.edit.merge_homophones.C], Optional[*lexedata.types.Form*]], Optional[lexedata.edit.merge_homophones.C]]

Create a merger that always returns c.

This is useful eg. for the status column, which needs to be updated when forms are merged, to a value that does not depend on the earlier status.

```
>>> constant = constant_factory("a")
>>> constant([None, 'b'])
'a'
>>> constant([])
'a'
```

lexedata.edit.merge_homophones.**default**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[lexedata.types.Form] = None*) → Optional[lexedata.edit.merge_homophones.C]

Merge with senbible defaults.

Union for sequence-shaped entries (strings, and lists with a separator in the metadata), must_be_equal otherwise

```
>>> default([1, 2])
Traceback (most recent call last):
AssertionError: ...
>>> default([[1, 2], [3, 4]])
[1, 2, 3, 4]
>>> default(["a; b", "a", "c; b"])
'a; b; c'
```

lexedata.edit.merge_homophones.**first**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[lexedata.types.Form] = None*) → Optional[lexedata.edit.merge_homophones.C]

Take the first nonzero entry, no matter whether the others match or not.

```
>>> first([1, 2])
1
>>> first([])
>>> first([None, 1, 2])
1
```

lexedata.edit.merge_homophones.**format_mergers**(*mergers: Mapping[str, Callable[[Sequence[lexedata.edit.merge_homophones.C], Optional[lexedata.types.Form]], Optional[lexedata.edit.merge_homophones.C]]]*) → str

lexedata.edit.merge_homophones.**isiterable**(*obj: object*) → bool

Test whether object is iterable, BUT NOT A STRING.

For merging purposes, we consider strings ATOMIC and thus NOT iterable.

lexedata.edit.merge_homophones.**merge_forms**(*data: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], mergers: typing.Mapping[str, typing.Callable[[typing.Sequence[lexedata.edit.merge_homophones.C], typing.Optional[lexedata.types.Form]], typing.Optional[lexedata.edit.merge_homophones.C]]], homophone_groups: typing.MutableMapping[lexedata.types.Form_ID, typing.Sequence[lexedata.types.Form_ID]], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[*lexedata.types.Form*]

Merge forms from a dataset.

TODO: Construct an example that shows that the order given in *homophone_groups* is maintained.

**Changes homophone_groups:** Groups that are skipped are removed

`lexedata.edit.merge_homophones.`**`merge_group`**(*forms: typing.Sequence[lexedata.types.Form], target: lexedata.types.Form, mergers: typing.Mapping[str, typing.Callable[[typing.Sequence[lexedata.edit.merge_homophones.C], typing.Optional[lexedata.types.Form]], typing.Optional[lexedata.edit.merge_homophones.C]]], dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], logger: logging.Logger = <Logger lexedata (INFO)>*) → *lexedata.types.Form*

Merge one group of homophones.

```
>>> merge_group(
...   [{"Parameter_ID": [1, 1]}, {"Parameter_ID": [2]}],
...   {"Parameter_ID": [1, 1]}, {"Parameter_ID": union}, util.fs.new_wordlist())
{'Parameter_ID': [1, 2]}
```

The target is assumed to be already included in the forms.

```
>>> merge_group(
...   [{"Parameter_ID": [1, 1]}, {"Parameter_ID": [2]}],
...   {"Parameter_ID": [1, 1]}, {"Parameter_ID": concatenate}, util.fs.new_
↪wordlist())
{'Parameter_ID': [1, 1, 2]}
```

`lexedata.edit.merge_homophones.`**`must_be_equal`**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

End with an error if entries are not equal.

```
>>> must_be_equal([1, 2])
Traceback (most recent call last):
AssertionError: assert 2 <= 1
>>> must_be_equal([1, 1])
1
>>> must_be_equal([])
```

`lexedata.edit.merge_homophones.`**`must_be_equal_or_null`**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

End with an error if those entries which are present are not equal.

```
>>> must_be_equal_or_null([1, 2])
Traceback (most recent call last):
AssertionError: assert 2 <= 1
...
```

```
>>> must_be_equal_or_null([1, 1])
1
>>> must_be_equal_or_null([1, 1, None])
1
```

`lexedata.edit.merge_homophones.`**`parse_homophones_old_format`**(*report: TextIO*) → Mapping[lexedata.types.Form_ID, Sequence[lexedata.types.Form_ID]]

Parse legacy homophones merge instructions >>> from io import StringIO >>> file = StringIO("Unconnected: Matsigenka kis {('ANGRY', '19148'), ('FIGHT (v. Or n.)', '19499'), ('CRITICIZE, SCOLD', '19819')}") >>> parse_homophones_old_format(file) defaultdict(<class 'list'>, {'19148': ['19499', '19819']})

`lexedata.edit.merge_homophones.`**`parse_homophones_report`**(*report: TextIO*) → Mapping[lexedata.types.Form_ID, Sequence[lexedata.types.Form_ID]]

Parse legacy homophones merge instructions

The format of the input file is the same as the output of the homophones report >>> from io import StringIO >>> file = StringIO("ache, e.ta.'kã: Unknown (but at least one concept not found)n" … " ache_one (one)n" … " ache_single_3 (single)n") >>> parse_homophones_report(file) defaultdict(<class 'list'>, {'ache_one': ['ache_one', 'ache_single_3']})

`lexedata.edit.merge_homophones.`**`parse_merge_override`**(*string: str*) → Tuple[str, Callable[[Sequence[lexedata.edit.merge_homophones.C], Optional[*lexedata.types.Form*]], Optional[lexedata.edit.merge_homophones.C]]]

`lexedata.edit.merge_homophones.`**`transcription`**(*wrapper: str = '{}'*)

Make a closure that adds variants to a variants column.

```
>>> row = {"variants": None}
>>> orthographic = transcription("<{}>")
>>> orthographic(["a", "a", "an"], row)
'a'
>>> row
{'variants': ['<an>']}
```

`lexedata.edit.merge_homophones.`**`union`**(*sequence: Sequence[lexedata.edit.merge_homophones.C], target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

Concatenate all entries, without duplicates.

Iterables are flattened. Strings are considered sequences of '; '-separated strings and flattened accordingly. Empty values are ignored.

```
>>> union([]) is None
True
>>> union([[1, 2], [2, 4]])
[1, 2, 4]
>>> union([None, [1], [3]])
[1, 3]
>>> union([[1, 1], [2]])
[1, 2]
>>> union([["a", "b"], ["c", "a"]])
```

(continues on next page)

(continued from previous page)

```
['a', 'b', 'c']
>>> union([None, "a", "b"])
'a; b'
>>> union(["a", "b", "a", ""])
'a; b'
>>> union(["a", "b", "a", None])
'a; b'
>>> union(["a", "b", "a; c", None])
'a; b; c'
>>> union([['one', 'one'], ['one1', 'one1'], ['two1', None], ['one'], ['one']])
['one', 'one1', 'two1']
```

`lexedata.edit.merge_homophones.`**`warn`**(*sequence: Sequence[lexedata.edit.merge_homophones.C]*, *target: Optional[*lexedata.types.Form*] = None*) → Optional[lexedata.edit.merge_homophones.C]

Print a warning if entries are not equal, but proceed taking the first one.

```
>>> warn([1, 2])
1
>>> warn([1, 1])
1
```

## lexedata.edit.normalize_unicode module

Normalize a dataset or file to NFC unicode normalization.

Make sure every string entry in every table of the dataset uses NFC unicode normalization, or take a list of files that each gets normalized.

`lexedata.edit.normalize_unicode.`**`n`**(*s: str*) → str

`lexedata.edit.normalize_unicode.`**`normalize`**(*file*, *original_encoding='utf-8'*)

## lexedata.edit.replace_id module

## lexedata.edit.replace_id_column module

## lexedata.edit.simplify_ids module

Clean up all ID columns in the dataset.

Take every ID column and convert it to either an integer-valued or a restricted-string-valued (only containing a-z, 0-9, or _) column, maintaining uniqueness of IDs, and keeping IDs as they are where they fit the format.

Optionally, create 'transparent' IDs, that is alphanumerical IDs which are derived from the characteristic columns of the corresponding table. For example, the ID of a FormTable would be derived from language and concept; for a CognatesetTable from the central concept if there is one.

### Module contents

## 1.4.2 lexedata.exporter package

### Submodules

### lexedata.exporter.cognates module

**class** lexedata.exporter.cognates.**BaseExcelWriter**(*dataset: pycldf.dataset.Dataset*, *database_url: typing.Optional[str] = None*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

Bases: `object`

Class logic for matrix-shaped Excel export.

**collect_forms_by_row**(*judgements: Iterable[*lexedata.types.Judgement*]*, *rows: Iterable[Union[lexedata.types.Cognateset_ID, lexedata.types.Parameter_ID]]*) → Mapping[lexedata.types.Cognateset_ID, Mapping[lexedata.types.Form_ID, Sequence[*lexedata.types.Judgement*]]]

Collect forms by row object (ie. concept or cognate set)

**create_excel**(*rows: Iterable[*lexedata.types.RowObject*]*, *languages*, *judgements: Iterable[*lexedata.types.Judgement*]*, *forms*, *size_sort: bool = False*) → None

Convert the initial CLDF into an Excel cognate view

The Excel file has columns "CogSet", one column each mirroring the other cognateset metadata, and then one column per language.

The rows contain cognate data. If a language has multiple reflexes in the same cognateset, these appear in different cells, one below the other.

**create_formcell**(*form:* lexedata.types.Form, *column: int*, *row: int*) → None

Fill the given cell with the form's data.

In the cell described by ws, column, row, dump the data for the form: Write into the the form data, and supply a comment from the judgement if there is one.

**create_formcells**(*row_forms: Iterable[*lexedata.types.Form*]*, *row_index: int*) → int

Writes all forms for given cognate set to Excel.

Take all forms for a given cognate set as given by the database, create a hyperlink cell for each form, and write those into rows starting at row_index.

Return the row number of the first empty row after this cognate set, which can then be filled by the following cognate set.

**abstract form_to_cell_value**(*form:* lexedata.types.Form)

Format a form into text for an Excel cell value

**header: List[Tuple[str, str]]**

**row_table: str**

**abstract set_header**(*dataset:* lexedata.types.Wordlist*[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID]*)

Define the header for the first few columns

**abstract write_row_header**(*row_object: lexedata.types.RowObject*, *row_index: int*)

Write a row header

Write into the first few columns of the row *row_index* of self.ws the metadata of a row, eg. concept ID and gloss or cognateset ID, cognateset name and status.

**class** lexedata.exporter.cognates.**ExcelWriter**(*dataset: pycldf.dataset.Dataset*, *database_url: typing.Optional[str] = None*, *singleton_cognate: bool = False*, *singleton_status: typing.Optional[str] = None*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

Bases: *lexedata.exporter.cognates.BaseExcelWriter*

Class logic for cognateset Excel export.

**form_to_cell_value**(*form: lexedata.types.Form*) → str

Build a string describing the form itself

Provide the best transcription and all translations of the form strung together.

```
>>> ds = util.fs.new_wordlist(FormTable=[], CognatesetTable=[], CognateTable=[])
>>> E = ExcelWriter(dataset=ds)
>>> E.form_to_cell_value({"form": "f", "parameterReference": "c"})
'f ‘c’'
>>> E.form_to_cell_value(
...   {"form": "f", "parameterReference": "c", "formComment": "Not empty"})
'f ‘c’ '
>>> E.form_to_cell_value(
...   {"form": "fo", "parameterReference": "c", "segments": ["f", "o"]})
'{ f o } ‘c’'
>>> E.form_to_cell_value(
...   {"form": "fo",
...    "parameterReference": "c",
...    "segments": ["f", "o"],
...    "segmentSlice": ["1:1"]})
'{ f }o ‘c’'
```

TODO: This function should at some point support alignments, so that the following call will return ‘{ - f - }o ‘c’” instead.

```
>>> E.form_to_cell_value(
...   {"form": "fo",
...    "parameterReference": "c",
...    "segments": ["f", "o"],
...    "segmentSlice": ["1:1"],
...    "alignment": ["", "f", ""]})
'{ f }o ‘c’'
```

**header: List[Tuple[str, str]]**

**row_table: str = 'CognatesetTable'**

**set_header**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID]*)

Define the header for the first few columns

**write_row_header**(*cogset*, *row_number: int*)

Write a row header

Write into the first few columns of the row *row_index* of self.ws the metadata of a row, eg. concept ID and gloss or cognateset ID, cognateset name and status.

**ws: openpyxl.worksheet.worksheet.Worksheet**

lexedata.exporter.cognates.**cogsets_and_judgements**(*dataset, status: typing.Optional[str], by_segment=True, logger: logging.Logger = <Logger lexedata (INFO)>*)

lexedata.exporter.cognates.**parser**()

lexedata.exporter.cognates.**properties_as_key**(*data*, *columns*)

lexedata.exporter.cognates.**sort_cognatesets**(*cogsets: List[*lexedata.types.CogSet*]*, *judgements: Sequence[*lexedata.types.Judgement*] = []*, *sort_column: Optional[str] = None*, *size: bool = True*) → None

Sort cognatesets by a given column, and optionally by size.

## lexedata.exporter.edictor module

Export a dataset to Edictor/Lingpy.

Input for edictor is a .tsv file containing the forms. The first column needs to be 'ID', containing 1-based integers. Cognatesets IDs need to be 1-based integers.

lexedata.exporter.edictor.**add_edictor_settings**(*file*, *dataset*)

Write a block of Edictor setting comments to a file.

Edictor takes some comments in its TSV files as directives for how it should behave. The important settings here are to set the morphology mode to 'partial' and pass the order of languages and concepts through. Everything else is pretty much edictor standard.

lexedata.exporter.edictor.**forms_to_tsv**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], languages: typing.Iterable[str], concepts: typing.Set[str], cognatesets: typing.Iterable[str], logger: logging.Logger = <Logger lexedata (INFO)>*)

lexedata.exporter.edictor.**glue_in_alignment**(*global_alignment*, *cogsets*, *new_alignment*, *new_cogset*, *segments: slice*)

Add a partial alignment to a global alignment with gaps

NOTE: This function does not check for overlapping alignments, it just assumes alignments do not overlap!

```
>>> alm = "(t) (e) (s) (t)".split()
>>> cogsets = [None]
>>> glue_in_alignment(alm, cogsets, list("es-"), 1, slice(1, 3))
>>> alm
['(t)', '+', 'e', 's', '-', '+', '(t)']
>>> cogsets
[None, 1, None]
>>> glue_in_alignment(alm, cogsets, list("-t-"), 2, slice(3, 4))
```

(continues on next page)

(continued from previous page)

```
>>> alm
['(t)', '+', 'e', 's', '-', '+', '-', 't', '-']
>>> cogsets
[None, 1, 2]
```

This is independent of the order in which alignments are glued in.

```
>>> alm = "(t) (e) (s) (t)".split()
>>> cogsets = [None]
>>> glue_in_alignment(alm, cogsets, list("-t-"), 2, slice(3, 4))
>>> alm
['(t)', '(e)', '(s)', '+', '-', 't', '-']
>>> cogsets
[None, 2]
>>> glue_in_alignment(alm, cogsets, list("es-"), 1, slice(1, 3))
>>> alm
['(t)', '+', 'e', 's', '-', '+', '-', 't', '-']
>>> cogsets
[None, 1, 2]
```

Of course, it also works for coplete forms, not just for partial cognate judgements.

```
>>> alm = "(t) (e) (s) (t)".split()
>>> cogsets = [None]
>>> glue_in_alignment(alm, cogsets, list("t-es-t-"), 3, slice(0, 4))
>>> alm
['t', '-', 'e', 's', '-', 't', '-']
>>> cogsets
[3]
```

`lexedata.exporter.edictor.`**`rename`**(*form_column*)

`lexedata.exporter.edictor.`**`write_edictor_file`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], file: TextIO, forms: Mapping[lexedata.types.Form_ID, Mapping[str, Any]], judgements_about_form, cognateset_numbers*)

Write the judgements of a dataset to file, in edictor format.

## lexedata.exporter.matrix module

**class** `lexedata.exporter.matrix.`**`MatrixExcelWriter`**(*dataset: pycldf.dataset.Dataset, database_url: typing.Optional[str] = None, logger: logging.Logger = <Logger lexedata (INFO)>*)

Bases: *`lexedata.exporter.cognates.BaseExcelWriter`*

Class logic for Excel matrix export.

**`create_formcell`**(*form: lexedata.types.Form, column: int, row: int*) → None

Fill the given cell with the form's data.

In the cell described by ws, column, row, dump the data for the form: Write into the the form data, and supply a comment from the judgement if there is one.

**`form_to_cell_value`**(*form: lexedata.types.Form*) → str

Format a form into text for an Excel cell value

**`header: List[Tuple[str, str]]`**

**`row_table: str = 'ParameterTable'`**

**`set_header`**(*dataset*)

Define the header for the first few columns

**`write_row_header`**(*cogset, row*)

Write a row header

Write into the first few columns of the row *row_index* of self.ws the metadata of a row, eg. concept ID and gloss or cognateset ID, cognateset name and status.

**`ws: openpyxl.worksheet.worksheet.Worksheet`**

## lexedata.exporter.phylogenetics module

`class lexedata.exporter.phylogenetics.`**`AbsenceHeuristic`**(*value*)

Bases: `enum.Enum`

An enumeration.

**`CENTRALCONCEPT = 0`**

**`HALFPRIMARYCONCEPTS = 1`**

`class lexedata.exporter.phylogenetics.`**`CodingProcedure`**(*value*)

Bases: `enum.Enum`

An enumeration.

**`MULTISTATE = 2`**

**`ROOTMEANING = 1`**

**`ROOTPRESENCE = 0`**

`class lexedata.exporter.phylogenetics.`**`Cognateset_ID`**

Bases: `str`

`class lexedata.exporter.phylogenetics.`**`Language_ID`**

Bases: `str`

`class lexedata.exporter.phylogenetics.`**`Parameter_ID`**

Bases: `str`

`lexedata.exporter.phylogenetics.`**`add_partitions`**(*data_object: lxml.etree.Element, partitions*) lexedata.exporter.phylogenetics.**apply_heuristics**(*dataset: lexedata.types.Wordlist, heuristic: typing.Optional[lexedata.exporter.phylogenetics.AbsenceHeuristic] = None, primary_concepts: typing.Union[lexedata.types.WorldSet[lexedata.types.Parameter_ID], typing.AbstractSet[lexedata.types.Parameter_ID]] = <lexedata.types.WorldSet object>, logger: logging.Logger = <Logger lexedata (INFO)>*) → Mapping[lexedata.types.Cognateset_ID, Set[lexedata.types.Parameter_ID]]

Compute the relevant concepts for cognatesets, depending on the heuristic.

These concepts will be considered when deciding whether a root is deemed absent in a language.

For the CentralConcept heuristic, the relevant concepts are the central concept of a cognateset, as given by the #parameterReference column of the CognatesetTable. A central concept not included in the primary_concepts is ignored with a warning.

```
>>> ds = util.fs.new_wordlist()
>>> cst = ds.add_component("CognatesetTable")
>>> ds["CognatesetTable"].tableSchema.columns.append(
...     pycldf.dataset.Column(
...         name="Central_Concept",
...         propertyUrl="http://cldf.clld.org/v1.0/terms.rdf#parameterReference"))
>>> ds.auto_constraints(cst)
>>> ds.write(CognatesetTable=[
...     {"ID": "cognateset1", "Central_Concept": "concept1"}
... ])
>>> apply_heuristics(ds, heuristic=AbsenceHeuristic.CENTRALCONCEPT) == {'cognateset1
↪': {'concept1'}}
True
```

This extends to the case where a cognateset may have more than one central concept.

```
>>> ds = util.fs.new_wordlist()
>>> cst = ds.add_component("CognatesetTable")
>>> ds["CognatesetTable"].tableSchema.columns.append(
...     pycldf.dataset.Column(
...         name="Central_Concepts",
...         propertyUrl="http://cldf.clld.org/v1.0/terms.rdf#parameterReference",
...         separator=","))
>>> ds.auto_constraints(cst)
>>> ds.write(CognatesetTable=[
...     {"ID": "cognateset1", "Central_Concepts": ["concept1", "concept2"]}
... ])
>>> apply_heuristics(ds, heuristic=AbsenceHeuristic.CENTRALCONCEPT) == {
...     'cognateset1': {'concept1', 'concept2'}}
True
```

For the HalfPrimaryConcepts heurisitc, the relevant concepts are all primary concepts connected to a cognateset.

```
>>> ds = util.fs.new_wordlist(
...     FormTable=[
...         {"ID": "f1", "Parameter_ID": "c1", "Language_ID": "l1", "Form": "x"},
...         {"ID": "f2", "Parameter_ID": "c2", "Language_ID": "l1", "Form": "x"}],
...     CognateTable=[
```

(continues on next page)

(continued from previous page)

```
...         {"ID": "1", "Form_ID": "f1", "Cognateset_ID": "s1"},
...         {"ID": "2", "Form_ID": "f2", "Cognateset_ID": "s1"}])
>>> apply_heuristics(ds, heuristic=AbsenceHeuristic.HALFPRIMARYCONCEPTS) == {
...     's1': {'c1', 'c2'}}
True
```

NOTE: This function cannot guarantee that every concept has at least one relevant concept, there may be cognatesets without! A cognateset with 0 relevant concepts will always be included, because 0 is at least half of 0.

`lexedata.exporter.phylogenetics.`**`compress_indices`**(*indices: Set[int]*) → Iterator[slice]

Turn groups of largely contiguous indices into slices.

```
>>> list(compress_indices(set(range(10))))
[slice(0, 10, None)]
```

```
>>> list(compress_indices([1, 2, 5, 6, 7]))
[slice(1, 3, None), slice(5, 8, None)]
```

`lexedata.exporter.phylogenetics.`**`fill_beast`**(*data_object: lxml.etree.Element*, *languages*, *sequences*) → None

Add sequences to BEAST as Alignment object.

```
>>> xml = ET.fromstring("<beast><data /></beast>")
>>> fill_beast(xml.find(".//data"), ["L1", "L2"], ["0110", "0011"])
>>> print(ET.tostring(xml).decode("utf-8"))
<beast><data id="vocabulary" dataType="integer" spec="Alignment">
<sequence id="language_data_vocabulary:L1" taxon="L1" value="0110"/>
<sequence id="language_data_vocabulary:L2" taxon="L2" value="0011"/>
<taxonset id="taxa" spec="TaxonSet"><plate var="language" range="{languages}">
↪<taxon id="$(language)" spec="Taxon"/></plate></taxonset></data></beast>
```

`lexedata.exporter.phylogenetics.`**`format_nexus`**(*languages: Iterable[str]*, *sequences: Iterable[str]*, *n_symbols: int*, *n_characters: int*, *datatype: str*, *partitions: Optional[Mapping[str, Iterable[int]]] = None*)

Format a Nexus output with the sequences.

This function only formats and performs no further validity checks!

```
>>> print(format_nexus(
...   ["l1", "l2"],
...   ["0010", "0111"],
...   2, 3,
...   "binary",
...   {"one": [1], "two": [2,3]}
... ))
#NEXUS
Begin Taxa;
  Dimensions ntax=2;
  TaxLabels l1 l2;
End;
```

(continues on next page)

(continued from previous page)

```
Begin Characters;
  Dimensions NChar=3;
  Format Datatype=Restriction Missing=? Gap=- Symbols="0 1" ;
  Matrix
    [The first column is constant zero, for programs with ascertainment correction]
    l1  0010
    l2  0111
  ;
End;
Begin Sets;
  CharSet one=1;
  CharSet two=2 3;
End;
```

`lexedata.exporter.phylogenetics.`**`multistate_code`**(*dataset: Mapping[lexedata.types.Language_ID, Mapping[lexedata.types.Parameter_ID, Set[lexedata.types.Cognateset_ID]]]*) → Tuple[Mapping[lexedata.types.Language_ID, Sequence[Set[int]], Sequence[int]]

Create a multistate root-meaning coding from cognate codes in a dataset

Take the cognate code information from a wordlist, i.e. a mapping of the form {Language ID: {Concept ID: {Cognateset ID}}}, and generate a multistate alignment from it that lists for every meaning which roots are used to represent that meaning in each language.

Also return the number of roots for each concept.

## Examples

```
>>> alignment, lengths = multistate_code({"Language": {"Meaning": {"Cognateset 1"}}}
↪)
>>> alignment =={'Language': [{0}]}
True
>>> lengths == [1]
True
```

```
>>> alignment, statecounts = multistate_code(
...     {"l1": {"m1": {"c1"}},
...      "l2": {"m1": {"c2"}, "m2": {"c1", "c3"}}})
>>> alignment["l1"][1]
set()
>>> alignment["l2"][1] == {0, 1}
True
>>> statecounts
[2, 2]
```

`lexedata.exporter.phylogenetics.`**`parser`**()

Construct the CLI argument parser for this script.

`lexedata.exporter.phylogenetics.`**`raw_binary_alignment`**(*alignment*)

`lexedata.exporter.phylogenetics.`**`raw_multistate_alignment`**(*alignment, long_sep: str = ','*) `lexedata.exporter.phylogenetics.`**`read_cldf_dataset`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], code_column: typing.Optional[str] = None, logger: logging.Logger = <Logger lexedata (INFO)>*) → Mapping[lexedata.types.Language_ID, Mapping[lexedata.types.Parameter_ID, Set[lexedata.types.Cognateset_ID]]]

Load a CLDF dataset.

Load the file as *json* CLDF metadata description file, or as metadata-free dataset contained in a single csv file.

The distinction is made depending on the file extension: *.json* files are loaded as metadata descriptions, all other files are matched against the CLDF module specifications. Directories are checked for the presence of any CLDF datasets in undefined order of the dataset types.

If use_ids == False, the reader is free to choose language names or language glottocodes for the output if they are unique.

## Examples

```
>>> import tempfile
>>> dirname = Path(tempfile.mkdtemp(prefix="lexedata-test"))
>>> target = dirname / "forms.csv"
>>> _size = open(target, "w").write('''
... ID,Language_ID,Parameter_ID,Form,Cognateset_ID
... '''.strip())
>>> ds = pycldf.Wordlist.from_data(target)
```

{‘autaa’: defaultdict(<class ‘set’>, {‘Woman’: {‘WOMAN1’}, ‘Person’: {‘PERSON1’}})} TODO: FIXME THIS EXAMPLE IS INCOMPLETE

**Parameters** **fname** (*str or Path*) – Path to a CLDF dataset

**Return type** Data

`lexedata.exporter.phylogenetics.`**`read_structure_dataset`**(*dataset: pycldf.dataset.StructureDataset, logger: logging.Logger = <Logger lexedata (INFO)>*) → MutableMapping[lexedata.types.Language_ID, MutableMapping[lexedata.types.Parameter_ID, Set]]

`lexedata.exporter.phylogenetics.`**`read_wordlist`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], code_column: typing.Optional[str], logger: logging.Logger = <Logger lexedata (INFO)>*) → MutableMapping[lexedata.types.Language_ID, MutableMapping[lexedata.types.Parameter_ID, Set]] `lexedata.exporter.phylogenetics.`**`root_meaning_code`**(*dataset: typing.Mapping[lexedata.types.Language_ID, typing.Mapping[lexedata.types.Parameter_ID, typing.Set[lexedata.types.Cognateset_ID]]], core_concepts: typing.Set[lexedata.types.Parameter_ID] = <lexedata.types.WorldSet object>, ascertainment: typing.Sequence[typing.Literal['0', '1', '?']] = ['0']*) → Tuple[Mapping[lexedata.types.Language_ID, List[Literal['0', '1', '?']]], Mapping[lexedata.types.Parameter_ID, Mapping[lexedata.types.Cognateset_ID, int]]]

Create a root-meaning coding from cognate codes in a dataset

Take the cognate code information from a wordlist, i.e. a mapping of the form {Language ID: {Concept ID: {Cognateset ID}}}, and generate a binary alignment from it that lists for every meaning which roots are used to represent that meaning in each language.

Return the aligment, and the list of slices belonging to each meaning.

The default ascertainment is the a single absence ('0'): The configuration where a form is absent from all languages is never observed, but always possible, so we add this entry for the purposes of ascertainment correction.

## Examples

```
>>> alignment, concepts = root_meaning_code({"Language": {"Meaning": {"Cognateset 1
↪"}}})
>>> alignment
{'Language': ['0', '1']}
```

```
>>> alignment, concepts = root_meaning_code(
...   {"l1": {"m1": {"c1"}},
...    "l2": {"m1": {"c2"}, "m2": {"c1", "c3"}}})
>>> sorted(concepts)
['m1', 'm2']
>>> sorted(concepts["m1"])
['c1', 'c2']
>>> {language: sequence[concepts["m1"]["c1"]] for language, sequence in alignment.
↪items()}
{'l1': '1', 'l2': '0'}
>>> {language: sequence[concepts["m2"]["c3"]] for language, sequence in alignment.
↪items()}
{'l1': '?', 'l2': '1'}
>>> list(zip(*sorted(zip(*alignment.values()))))
[('0', '0', '1', '?', '?'), ('0', '1', '0', '1', '1')]
```

`lexedata.exporter.phylogenetics.`**`root_presence_code`**(*dataset: typing.Mapping[lexedata.types.Language_ID, typing.Mapping[lexedata.types.Parameter_ID, typing.Set[lexedata.types.Cognateset_ID]]], relevant_concepts: typing.Mapping[lexedata.types.Cognateset_ID, typing.Iterable[lexedata.types.Parameter_ID]], ascertainment: typing.Sequence[typing.Literal['0', '1', '?']] = ['0'], logger: logging.Logger = <Logger lexedata (INFO)>*) → Tuple[Mapping[lexedata.types.Language_ID, List[Literal['0', '1', '?']]], Mapping[lexedata.types.Cognateset_ID, int]]

Create a root-presence/absence coding from cognate codes in a dataset

Take the cognate code information from a wordlist, i.e. a mapping of the form {Language ID: {Concept ID: {Cognateset ID}}}, and generate a binary alignment from it that lists for every root whether it is present in that language or not. Return that, and the association between cognatesets and characters.

```
>>> alignment, roots = root_presence_code(
...     {"Language": {"Meaning": {"Cognateset 1"}}},
...     relevant_concepts={"Cognateset 1": ["Meaning"]})
>>> alignment
{'Language': ['0', '1']}
>>> roots
{'Cognateset 1': 1}
```

The first entry in each sequence is always '0': The configuration where a form is absent from all languages is never observed, but always possible, so we add this entry for the purposes of ascertainment correction.

If a root is attested at all, in any concept, it is considered present. Because the word list is never a complete description of the language's lexicon, the function employs a heuristic to generate 'absent' states.

If a root is unattested, and at least half of the relevant concepts associated with this root are attested, but each expressed by another root, the root is assumed to be absent in the target language. (If there is exactly one central concept, then that central concept being attested or unknown is a special case of this general rule.) Otherwise the presence/absence of the root is considered unknown.

```
>>> alignment, roots = root_presence_code(
...     {"l1": {"m1": {"c1"}},
...      "l2": {"m1": {"c2"}, "m2": {"c1", "c3"}}},
...     relevant_concepts={"c1": ["m1"], "c2": ["m1"], "c3": ["m2"]})
>>> sorted(roots)
['c1', 'c2', 'c3']
>>> sorted_roots = sorted(roots.items())
>>> {language: [sequence[k[1]] for k in sorted_roots] for language, sequence in␣
↪alignment.items()}
{'l1': ['1', '0', '?'], 'l2': ['1', '1', '1']}
>>> list(zip(*sorted(zip(*alignment.values()))))
[('0', '0', '1', '?'), ('0', '1', '1', '1')]
```

### Module contents

## 1.4.3 lexedata.importer package

### Submodules

### lexedata.importer.cognates module

Load #cognate and #cognatesets from excel file into CLDF

`class lexedata.importer.cognates.`**`CognateEditParser`**(*output_dataset: pycldf.dataset.Dataset, row_type=<class 'lexedata.types.CogSet'>, top: int = 2, cellparser: lexedata.util.excel.NaiveCellParser = <class 'lexedata.util.excel.CellParser'>, row_header=['set', 'Name', None], check_for_match: typing.List[str] = ['Form'], check_for_row_match: typing.List[str] = ['Name'], check_for_language_match: typing.List[str] = ['Name']*)

Bases: `lexedata.importer.excel_matrix.ExcelCognateParser`

**`language_from_column`**(*column: List[openpyxl.cell.cell.Cell]*) → *lexedata.types.Language*

**`properties_from_row`**(*row: List[openpyxl.cell.cell.Cell]*) → Optional[*lexedata.types.RowObject*]

`lexedata.importer.cognates.`**`header_from_cognate_excel`**(*ws: openpyxl.worksheet.worksheet.Worksheet, dataset: pycldf.dataset.Dataset, logger: logging.Logger = <Logger lexedata (INFO)>*)

`lexedata.importer.cognates.`**`import_cognates_from_excel`**(*ws: openpyxl.worksheet.worksheet.Worksheet, dataset: pycldf.dataset.Dataset, extractor: re.Pattern = re.compile('/(?P<ID>[^/]*)/?$'), logger: logging.Logger = <Logger lexedata (INFO)>*) → None

### lexedata.importer.edictor module

`lexedata.importer.edictor.`**`edictor_to_cldf`**(*dataset:* lexedata.types.Wordlist*[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], new_cogsets: Mapping[lexedata.types.Cognateset_ID, List[Tuple[lexedata.types.Form_ID, range, Sequence[str]]]], affected_forms: Set[lexedata.types.Form_ID], source: List[str] = []*)

`lexedata.importer.edictor.`**`extract_partial_judgements`**(*segments: typing.Sequence[str], cognatesets: typing.Sequence[int], global_alignment: typing.Sequence[str], logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterator[Tuple[range, int, Sequence[str]]]

Extract the different partial cognate judgements.

Segments has no morpheme boundary markers, they are inferred from global_alignment. The number of cognatesets and marked segments in global_alignment must match.

```
>>> next(extract_partial_judgements("t e s t".split(), [3], "t e s t".split()))
(range(0, 4), 3, ['t', 'e', 's', 't'])
```

```
>>> partial = extract_partial_judgements("t e s t".split(), [0, 1, 2], "( t ) + e -␣
↪s + - t -".split())
>>> next(partial)
(range(1, 3), 1, ['e', '-', 's'])
>>> next(partial)
(range(3, 4), 2, ['-', 't', '-'])
```

`lexedata.importer.edictor.`**`load_forms_from_tsv`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], input_file: pathlib.Path, logger: logging.Logger = <Logger lexedata (INFO)>*) → Mapping[int, Sequence[Tuple[lexedata.types.Form_ID, range, Sequence[str]]]]

This function overwrites dataset's FormTable

`lexedata.importer.edictor.`**`match_cognatesets`**(*new_cognatesets: Mapping[int, Sequence[Tuple[lexedata.types.Form_ID, range, Sequence[str]]]], reference_cognatesets: Mapping[lexedata.types.Cognateset_ID, Sequence[Tuple[lexedata.types.Form_ID, range, Sequence[str]]]]*) → Mapping[int, Optional[lexedata.types.Cognateset_ID]]

Match two different cognateset assignments with each other.

Map the new_cognatesets to the reference_cognatesets by trying to maximize the overlap between each new cognateset and the reference cognateset it is mapped to.

So, if two cognatesets got merged, they get mapped to the bigger one:

```
>>> match_cognatesets(
...   {0: ["a", "b", "c"]},
...   {">": ["a", "b"], "<": ["c"]}
... )
{0: '>'}
```

If a cognateset got split, it gets mapped to the bigger part and the other part becomes a new, unmapped set:

```
>>> match_cognatesets(
...   {0: ["a", "b"], 1: ["c"]},
...   {"": ["a", "b", "c"]}
... )
{0: '', 1: None}
```

If a single form (or a relatively small number) gets moved between cognatesets, the mapping is maintained:

```
>>> match_cognatesets(
...   {0: ["a", "b"], 1: ["c", "d", "e"]},
...   {0: ["a", "b", "c"], 1: ["d", "e"]}
```

(continues on next page)

(continued from previous page)

```
... ) == {1: 1, 0: 0}
True
```

(As you see, the function is a bit more general than the type signature implies.)

### lexedata.importer.excel_interleaved module

Import data in the "interleaved" format from an Excel spreadsheet.

Here, every even row contains cells with forms (cells may contain multiple forms), while every odd row contains the associated cognate codes (a one-to-one relationship between forms and codes is expected). Forms and cognate codes are separated by commas (",") and semi-colons (";"). Any other information existing in the cell will be parsed as part of the form or the cognate code.

`lexedata.importer.excel_interleaved.`**`import_interleaved`**(*ws: openpyxl.worksheet.worksheet.Worksheet*, *logger: logging.Logger = <Logger lexedata (INFO)>*, *ids: typing.Optional[typing.Set[lexedata.types.Cognateset_ID]] = None*) → Iterable[Tuple[lexedata.types.Form_ID, lexedata.types.Language_ID, lexedata.types.Parameter_ID, Optional[str], Optional[str], lexedata.types.Cognateset_ID]]

### lexedata.importer.excel_long_format module

**`class`** `lexedata.importer.excel_long_format.`**`ImportLanguageReport`**(*is_new_language: bool = False*, *new: int = 0*, *existing: int = 0*, *skipped: int = 0*, *concepts: int = 0*)

Bases: `object`

**`concepts: int`**

**`existing: int`**

**`is_new_language: bool`**

**`new: int`**

**`skipped: int`**

`lexedata.importer.excel_long_format.`**`add_single_languages`**(*metadata: pathlib.Path*, *sheets: Iterable[openpyxl.worksheet.worksheet.Worksheet]*, *match_form: Optional[List[str]]*, *concept_name: Optional[str]*, *language_name: Optional[str]*, *ignore_missing: bool*, *ignore_superfluous: bool*, *status_update: Optional[str]*, *logger: logging.Logger*) → Mapping[str, *lexedata.importer.excel_long_format.ImportLanguageReport*] `lexedata.importer.excel_long_format.`**`get_headers_from_excel`**(*sheet: openpyxl.worksheet.worksheet.Worksheet*) → Iterable[str]

`lexedata.importer.excel_long_format.`**`import_data_from_sheet`**(*sheet*, *sheet_header*, *language_id: str*, *implicit: Mapping[Literal['languageReference', 'id', 'value'], str] = {}*, *concept_column: Tuple[str, str] = ('Concept_ID', 'Concept_ID')*) → Iterable[*lexedata.types.Form*]

`lexedata.importer.excel_long_format.`**`parser`**()

`lexedata.importer.excel_long_format.`**`read_single_excel_sheet`**(*dataset: pycldf.dataset.Dataset*, *sheet: openpyxl.worksheet.worksheet.Worksheet*, *logger: logging.Logger = <Logger lexedata (INFO)>*, *match_form: typing.Optional[typing.List[str]] = None*, *entries_to_concepts: typing.Mapping[str, str] = <lexedata.types.KeyKeyDict object>*, *concept_column: typing.Optional[str] = None*, *language_name_column: typing.Optional[str] = None*, *ignore_missing: bool = False*, *ignore_superfluous: bool = False*, *status_update: typing.Optional[str] = None*) → Mapping[str, *lexedata.importer.excel_long_format.ImportLanguageRepor*

## lexedata.importer.excel_matrix module

**class** `lexedata.importer.excel_matrix.`**`DB`**(*output_dataset: pycldf.dataset.Wordlist*)

Bases: `object`

An in-memobry cache of a dataset.

The cache_dataset method is only called in the load_dataset method, but also used for finer control by the cognates importer. This means that if you would load the CognateParser directly, it doesn't work as the cache is empty and the code will throw errors (e.g. when trying to look for candidates we get a key error). If you use the CognateParser elsewhere, make sure to cache the dataset explicitly, eg. by using DB.from_dataset!

**`add_source`**(*source_id*)

**`associate`**(*form_id: str*, *row:* lexedata.types.RowObject, *comment: Optional[str] = None*) → bool

**`cache: Dict[str, Dict[Hashable, Dict[str, Any]]]`**

**`cache_dataset`**(*logger: logging.Logger = <Logger lexedata (INFO)>*)

**`commit`**()

**drop_from_cache**(*table: str*)

**empty_cache**()

**find_db_candidates**(*object: lexedata.importer.excel_matrix.Ob*, *properties_for_match: Iterable[str]*, *edit_dist_threshold: Optional[int] = None*) → Iterable[str]

**classmethod from_dataset**(*dataset*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

Create a (filled) cache from a dataset.

**insert_into_db**(*object:* lexedata.types.Object) → None

**make_id_unique**(*object:* lexedata.types.Object) → str

**retrieve**(*table_type: str*)

**source_ids: Set[str]**

**write_dataset_from_cache**(*tables: Optional[Iterable[str]] = None*)

**class** lexedata.importer.excel_matrix.**ExcelCognateParser**(*output_dataset: pycldf.dataset.Dataset*, *row_type=<class 'lexedata.types.CogSet'>*, *top: int = 2*, *cellparser: lexedata.util.excel.NaiveCellParser = <class 'lexedata.util.excel.CellParser'>*, *row_header=['set', 'Name', None]*, *check_for_match: typing.List[str] = ['Form']*, *check_for_row_match: typing.List[str] = ['Name']*, *check_for_language_match: typing.List[str] = ['Name']*)

Bases: *lexedata.importer.excel_matrix.ExcelParser*[*lexedata.types.CogSet*]

**associate**(*form_id: str*, *row:* lexedata.types.RowObject, *comment: Optional[str] = None*) → bool

**handle_form**(*params*, *row_object:* lexedata.types.CogSet, *cell_with_forms*, *this_lan*, *status_update: Optional[str]*)

**on_form_not_found**(*form: typing.Dict[str, typing.Any]*, *cell_identifier: typing.Optional[str] = None*, *language_id: typing.Optional[str] = None*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → bool

Should I add a missing object? No, but inform the user.

Send a warning (ObjectNotFoundWarning) reporting the missing object and cell.

**Returns** False

**Return type** The object should not be added.

**on_language_not_found**(*language: Dict[str, Any]*, *cell_identifier: Optional[str] = None*) → bool

Should I add a missing object? No, the object missing is an error.

Raise an exception (ObjectNotFoundWarning) reporting the missing object and cell.

**Raises** *ObjectNotFoundWarning* –

**on_row_not_found**(*row_object:* lexedata.types.CogSet, *cell_identifier: Optional[str] = None*) → bool

Create row object

**properties_from_row**(*row: List[openpyxl.cell.cell.Cell]*) → Optional[*lexedata.types.CogSet*]

class lexedata.importer.excel_matrix.**ExcelParser**(*output_dataset: pycldf.dataset.Dataset, row_type: typing.Type[lexedata.types.R], top: int = 2, cellparser: lexedata.util.excel.NaiveCellParser = <class 'lexedata.util.excel.CellParser'>, row_header: typing.List[str] = ['set', 'Name', None], check_for_match: typing.List[str] = ['ID'], check_for_row_match: typing.List[str] = ['Name'], check_for_language_match: typing.List[str] = ['Name'], fuzzy=0*)

Bases: `Generic[lexedata.types.R]`

**handle_form**(*params*, *row_object: lexedata.types.R*, *cell_with_forms*, *this_lan: str*, *status_update: Optional[str]*)

**language_from_column**(*column: List[openpyxl.cell.cell.Cell]*) → *lexedata.types.Language*

**on_form_not_found**(*form: Dict[str, Any]*, *cell_identifier: Optional[str] = None*, *language_id: Optional[str] = None*) → bool

Create form

**on_language_not_found**(*language: Dict[str, Any]*, *cell_identifier: Optional[str] = None*) → bool

Create language

**on_row_not_found**(*row_object: lexedata.types.R*, *cell_identifier: Optional[str] = None*) → bool

Create row object

**parse_all_languages**(*sheet: openpyxl.worksheet.worksheet.Worksheet*) → Dict[str, str]

Parse all language descriptions in the focal sheet.

**Returns** languages

**Return type** A dictionary mapping columns ("B", "C", "D", …) to language IDs

**parse_cells**(*sheet: openpyxl.worksheet.worksheet.Worksheet*, *status_update: Optional[str] = None*) → None

**properties_from_row**(*row: List[openpyxl.cell.cell.Cell]*) → Optional[lexedata.types.R]

lexedata.importer.excel_matrix.**cells_are_empty**(*cells: Iterable[openpyxl.cell.cell.Cell]*) → bool

lexedata.importer.excel_matrix.**excel_parser_from_dialect**(*output_dataset: pycldf.dataset.Wordlist*, *dialect: NamedTuple*, *cognate: bool*) → Type[*lexedata.importer.excel_matrix.ExcelParser*]

lexedata.importer.excel_matrix.**load_dataset**(*metadata: pathlib.Path, lexicon: typing.Optional[str], cognate_lexicon: typing.Optional[str] = None, status_update: typing.Optional[str] = None, logger: logging.Logger = <Logger lexedata (INFO)>*)

### Module contents

## 1.4.4 lexedata.report package

### Submodules

### lexedata.report.coverage module

`class lexedata.report.coverage.`**`Missing`**`(`*value*`)`

Bases: `enum.Enum`

An enumeration.

**`COUNT_NORMALLY = 1`**

**`IGNORE = 0`**

**`KNOWN = 2`**

`lexedata.report.coverage.`**`coverage_report`**`(`*dataset:* lexedata.types.Wordlist*[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID]*, *min_percentage: float = 0.0*, *with_concept: Iterable[lexedata.types.Parameter_ID] = {}*, *missing:* lexedata.report.coverage.Missing *= Missing.KNOWN*, *only_coded: bool = True*`)` → List[List[str]]

`lexedata.report.coverage.`**`coverage_report_concepts`**`(`*dataset: pycldf.dataset.Dataset*`)`

### lexedata.report.extended_cldf_validate module

Validate a CLDF wordlist.

This script runs some more validators specific to CLDF Wordlist datasets in addition to the validation implemented in the *pycldf* core. Some of those tests are not yet mandated by the CLDF standard, but are assumptions which some tools (including lexedata) tacitly make, so this validator makes them explicit.

TODO: There may be programmatic ways to fix the issues that this script reports. Those automatic fixes should be made more obvious.

`lexedata.report.extended_cldf_validate.`**`check_foreign_keys`**`(`*dataset: pycldf.dataset.Dataset*, *logger: logging.Logger = <Logger lexedata (INFO)>*`)`

`lexedata.report.extended_cldf_validate.`**`check_id_format`**`(`*dataset: pycldf.dataset.Dataset*, *logger: logging.Logger = <Logger lexedata (INFO)>*`)`

`lexedata.report.extended_cldf_validate.`**`check_na_form_has_no_alternative`**(*dataset: lexedata.types.Wordlist[lexedata.types.Langu lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], logger: logging.Logger = <Logger lexedata (INFO)>*)

`lexedata.report.extended_cldf_validate.`**`check_no_separator_in_ids`**(*dataset: pycldf.dataset.Dataset, logger: <Logger lexedata (INFO)> = <Logger lexedata (INFO)>*) → bool

`lexedata.report.extended_cldf_validate.`**`check_segmentslice_separator`**(*dataset*, *logger=None*) → bool

`lexedata.report.extended_cldf_validate.`**`check_unicode_data`**(*dataset: pycldf.dataset.Dataset, unicode_form: str = 'NFC', logger: logging.Logger = <Logger lexedata (INFO)>*) → bool

`lexedata.report.extended_cldf_validate.`**`log_or_raise`**(*message*, *log: logging.Logger = <Logger lexedata (INFO)>*)

## lexedata.report.filter module

Filter some table by some column.

Print the partial table to STDOUT or a file, so it can be used as subset-filter for some other script, and output statistics (how many included, how many excluded, what proportion, maybe sub-statistics for xxxReference columns, i.e. by language or by conceptr) to STDERR.

For example, assume you want to filter your FormTable down to those forms that start with a 'b', except for those forms from Fantastean varieties, which all have a name containing 'Fantastean'. You can do this using two calls to this program like this:

**python -m lexedata.report.filter Form '^b' FormTable -c ID -c Language_ID |** python -m lexedata.report.filter -V Language_ID 'Fantastean' -c ID

If you are aware of standard Unix tools, this script is a column-aware, but otherwise vastly reduced implementation of *grep*.

`lexedata.report.filter.`**`filter`**(*table: typing.Iterable[lexedata.report.filter.R], column: str, filter: re.Pattern, invert: bool = False, logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterator[lexedata.report.filter.R]

Return all rows matching a filter

Match the filter regular expression and return all rows in the table where the filter matches the column. (Or all where it does not, if invert==True.)

```
>>> list(filter([
...   {"C": "A"},
...   {"C": "An"},
...   {"C": "T"},
...   {"C": "E"},
... ], "C", re.compile("A"), invert=True))
[{'C': 'T'}, {'C': 'E'}]
```

`lexedata.report.filter.`**`parser`**`()`

## lexedata.report.homophones module

Generate a report of homophones.

List all groups of homophones in the dataset, together with (if available) the minimal spanning tree according to clics, in order to identify polysemies vs. accidental homophones

`lexedata.report.homophones.`**`list_homophones`**`(`*dataset: pycldf.dataset.Dataset*, *out: io.TextIOBase*, *logger: logging.Logger = <Logger lexedata (INFO)>*`)` → None

## lexedata.report.judgements module

Check that the judgements make sense.

`lexedata.report.judgements.`**`check_cognate_table`**`(`*dataset: pycldf.dataset.Wordlist*, *logger=<Logger lexedata (INFO)>*, *strict_concatenative=False*`)` → bool

Check that the CognateTable makes sense.

The cognate table MUST have an indication of forms, in a #formReference column, and cognate sets, in a #cognatesetReference column. It SHOULD have segment slices (#segmentSlice) and alignments (#alignment).

- The segment slice must be a valid (1-based, inclusive) slice into the segments of the form
- The alignment must match the segment slice applied to the segments of the form
- The length of the alignment must match the lengths of other alignments of that cognate set
- NA forms (Including “” for “source reports form as unknown” must not be in cognatesets)

If checking for strictly concatenative morphology, also check that the segment slice is a contiguous, non-overlapping section of the form.

Having no cognates is a valid choice for a dataset, so this function returns True if no CognateTable was found.

`lexedata.report.judgements.`**`log_or_raise`**`(`*message*, *logger=<Logger lexedata (INFO)>*`)`

### lexedata.report.nonconcatenative_morphemes module

`lexedata.report.nonconcatenative_morphemes.`**`cluster_overlaps`**(*overlapping_cognatesets: typing.Iterable[typing.Tuple[lexedata.types.Cognateset_ID, lexedata.types.Cognateset_ID]], out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*) → None

`lexedata.report.nonconcatenative_morphemes.`**`network_of_overlaps`**(*which_segment_belongs_to_which_cognateset: Mapping[lexedata.types.Form_ID, List[Set[lexedata.types.Cognateset_ID]]], forms_cache: Optional[Mapping[lexedata.types.Form_ID, lexedata.types.Form]] = None*) → Set[Tuple[lexedata.types.Cognateset_ID, lexedata.types.Cognateset_ID]]

`lexedata.report.nonconcatenative_morphemes.`**`segment_to_cognateset`**(*dataset: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], cognatesets: typing.Container[lexedata.types.Cognateset_ID], logger: logging.Logger = <Logger lexedata (INFO)>*) → Mapping[lexedata.types.Form_ID, List[Set[lexedata.types.Cognateset_ID]]]

### lexedata.report.segment_inventories module

Report the segment inventory of each language.

Report the phonemes (or whatever is represented by the #segments column) for each language, each with frequencies and whether the segments are valid CLTS.

`lexedata.report.segment_inventories.`**`comment_on_sound`**(*sound: str*) → str

Return a comment on the sound, if necessary.

```
>>> comment_on_sound("a")
''
>>> comment_on_sound("_")
'Marker'
>>> comment_on_sound("(")
'Invalid BIPA'
```

`lexedata.report.segment_inventories.`**`count_segments`**(*dataset:* lexedata.types.Wordlist*[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID]*, *languages: Container[lexedata.types.Language_ID]*)

### Module contents

## 1.4.5 lexedata.util package

### Submodules

### lexedata.util.add_metadata module

Starting with a forms.csv, add metadata for all columns we know about.

`lexedata.util.add_metadata.`**`add_metadata`**(*fname: pathlib.Path*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

### lexedata.util.excel module

Various helper functions for Excel file parsing

**`class`** `lexedata.util.excel.`**`CellParser`**(*dataset: pycldf.dataset.Dataset*, *element_semantics: typing.Iterable[typing.Tuple[str, str, str, bool]] = [('<', '>', 'form', True), ('(', ')', 'comment', False), ('{', '}', 'source', False)]*, *separation_pattern: str = '([;,])'*, *variant_separator: typing.Optional[typing.List[str]] = ['~', '%']*, *add_default_source: typing.Optional[str] = '{1}'*, *logger: logging.Logger = <Logger lexedata (INFO)>*)

Bases: *`lexedata.util.excel.NaiveCellParser`*

**`c: Dict[str, str]`**

**`create_cldf_form`**(*properties: Dict[str, Any]*) → Optional[str]

Return first transcription out of properties as a candidate for cldf_form. Order of transcriptions corresponds to order of cell_parser_semantics as provided in the metadata.

**`parse_form`**(*form_string: str*, *language_id: str*, *cell_identifier: str = ''*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → Optional[*lexedata.types.Form*]

Create a dictionary of columns from a form description.

Extract each value (transcriptions, comments, sources etc.) from a string describing a single form.

**`postprocess_form`**(*properties: Dict[str, Any]*, *language_id: str*) → None

Modify the form in-place

Fix some properties of the form. This is the place to add default sources, cut of delimiters, split unmarked variants, etc.

**separate**(*values: str*, *context: str = ''*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[str]

Separate different form descriptions in one string.

Separate forms separated by comma or semicolon, unless the comma or semicolon occurs within a set of matching component delimiters (eg. brackets)

If the brackets don't match, the whole remainder string is passed on, so that the form parser can try to recover as much as possible or throw an exception.

**source_from_source_string**(*source_string: str*, *language_id: typing.Optional[str]*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → str

Parse a string referencing a language-specific source

**property transcriptions**

class lexedata.util.excel.**CellParserHyperlink**(*dataset: pycldf.dataset.Dataset*, *extractor: re.Pattern*)

Bases: `lexedata.util.excel.NaiveCellParser`

**c: Dict[str, str]**

**parse**(*cell: openpyxl.cell.cell.Cell*, *language_id: str*, *cell_identifier: str = ''*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[*lexedata.types.Judgement*]

Return form properties for every form in the cell

class lexedata.util.excel.**MawetiCellParser**(*dataset: pycldf.dataset.Dataset*, *element_semantics: Iterable[Tuple[str, str, str, bool]]*, *separation_pattern: str*, *variant_separator: list*, *add_default_source: Optional[str]*)

Bases: `lexedata.util.excel.CellParser`

**c: Dict[str, str]**

**postprocess_form**(*properties: Dict[str, Any]*, *language_id: str*) → None

Post processing specific to the Maweti dataset

class lexedata.util.excel.**MawetiCognateCellParser**(*dataset: pycldf.dataset.Dataset*, *element_semantics: Iterable[Tuple[str, str, str, bool]]*, *separation_pattern: str*, *variant_separator: list*, *add_default_source: Optional[str]*)

Bases: `lexedata.util.excel.MawetiCellParser`

**c: Dict[str, str]**

**parse_form**(*values*, *language*, *cell_identifier: str = ''*)

Create a dictionary of columns from a form description.

Extract each value (transcriptions, comments, sources etc.) from a string describing a single form.

class lexedata.util.excel.**NaiveCellParser**(*dataset: pycldf.dataset.Dataset*)

Bases: `object`

**c: Dict[str, str]**

**cc**(*short*, *long*, *dataset*)

Cache the name of a column, or complain if it doesn't exist

**parse**(*cell: openpyxl.cell.cell.Cell*, *language_id: str*, *cell_identifier: str = ''*, *logger: logging.Logger = <Logger lexedata (INFO)>*) → Iterable[*lexedata.types.Form*]

Return form properties for every form in the cell

**parse_form**(*form_string: str*, *language_id: str*, *cell_identifier: str = ''*) → Optional[*lexedata.types.Form*]

**separate**(*values: str*, *context: str = ''*) → Iterable[str]

Separate different form descriptions in one string.

Separate forms separated by comma.

lexedata.util.excel.**alignment_from_braces**(*text*, *start=0*)

Convert a brace-delimited morpheme description into slices and alignments.

The "-" character is used as the alignment gap character, so it does not count towards the segment slices.

If opening or closing brackets are missing, the slice goes until the end of the form.

```
>>> alignment_from_braces("t{e x t")
([(2, 4)], ['e', 'x', 't'])
>>> alignment_from_braces("t e x}t")
([(1, 3)], ['t', 'e', 'x'])
>>> alignment_from_braces("t e x t")
([(1, 4)], ['t', 'e', 'x', 't'])
```

lexedata.util.excel.**check_brackets**(*string*, *bracket_pairs*)

Check whether all brackets match.

This function can check the matching of simple bracket pairs, like this:

```
>>> b = {"(": ")", "[": "]", "{": "}"}
>>> check_brackets("([])", b)
True
>>> check_brackets("([]])", b)
False
>>> check_brackets("([[])", b)
False
>>> check_brackets("This (but [not] this)", b)
True
```

But it can also deal with multi-character matches

```
>>> b = {"(": ")", "begin": "end"}
>>> check_brackets("begin (__ (!) xxx) end", b)
True
>>> check_brackets("begin (__ (!) end) xxx", b)
False
```

This includes multi-character matches where some pair is a subset of another pair. Here the order of the pairs in the dictionary is important – longer pairs must be defined first.

```
>>> b = {":::": ":::", ":": ":"}
>>> check_brackets("::: :::", b)
True
>>> check_brackets("::::::", b)
True
>>> check_brackets("::::", b)
False
>>> check_brackets(":: ::", b)
True
```

In combination, these features allow for natural escape sequences:

```
>>> b = {"!(": "", "!)": "", "(": ")", "[": "]"}
>>> check_brackets("(text)", b)
True
>>> check_brackets("(text", b)
False
>>> check_brackets("text)", b)
False
>>> check_brackets("(te[xt)]", b)
False
>>> check_brackets("!(text", b)
True
>>> check_brackets("text!)", b)
True
>>> check_brackets("!(te[xt!)]", b)
True
```

`lexedata.util.excel.`**`clean_cell_value`**`(`*cell: openpyxl.cell.cell.Cell*, *logger=<Logger lexedata (INFO)>*`)`

Return the value of an Excel cell in a useful format and normalized.

`lexedata.util.excel.`**`components_in_brackets`**`(`*form_string*, *bracket_pairs*`)`

Find all elements delimited by complete pairs of matching brackets.

```
>>> b = {"!/": "", "(": ")", "[": "]", "{": "}", "/": "/"}
>>> components_in_brackets("/aha/ (exclam. !/ int., also /ah/)",b)
['', '/aha/', ' ', '(exclam. !/ int., also /ah/)', '']
```

Recovery from mismatched delimiters early in the string is difficult. The following example is still waiting for the first '/' to be closed by the end of the string.

```
>>> components_in_brackets("/aha (exclam. !/ int., also /ah/)",b)
['', '/aha (exclam. !/ int., also /ah/)']
```

`lexedata.util.excel.`**`get_cell_comment`**`(`*cell: openpyxl.cell.cell.Cell*`)` → str

Get the comment of a cell.

Get the normalized comment of a cell: Guaranteed to be a string (empty if no comment), with lines joined by spaces instead and all 'lexedata' author annotations stripped.

```
>>> from openpyxl.comments import Comment
>>> wb = op.Workbook()
>>> ws = wb.active
>>> ws["A1"].comment = Comment('''This comment
... contains a linebreak and a signature.
...   -lexedata.exporter''',
... 'lexedata')
>>> get_cell_comment(ws["A1"])
'This comment contains a linebreak and a signature.'
>>> get_cell_comment(ws["A2"])
''
```

`lexedata.util.excel.`**`normalize_header`**`(`*row: Iterable[openpyxl.cell.cell.Cell]*`)` → Iterable[str]

### lexedata.util.fs module

`lexedata.util.fs.`**`copy_dataset`**(*original: pathlib.Path*, *target: pathlib.Path*) → pycldf.dataset.Dataset

Return a copy of the dataset at original.

Copy the dataset (metadata and relative table URLs) from *original* to *target*, and return the new dataset at *target*.

`lexedata.util.fs.`**`get_dataset`**(*fname: pathlib.Path*) → pycldf.dataset.Dataset

Load a CLDF dataset.

Load the file as *json* CLDF metadata description file, or as metadata-free dataset contained in a single csv file.

The distinction is made depending on the file extension: *.json* files are loaded as metadata descriptions, all other files are matched against the CLDF module specifications. Directories are checked for the presence of any CLDF datasets in undefined order of the dataset types.

**Parameters** **fname** (`str or Path`) – Path to a CLDF dataset

**Return type** Dataset

`lexedata.util.fs.`**`new_wordlist`**(*path: Optional[pathlib.Path] = None*, ***data*) → *lexedata.types.Wordlist*[str, str, str, str, str]

Create a new CLDF wordlist.

By default, the wordlist is created in a new temporary directory, but you can specify a path to create it in.

To immediately fill some tables, provide keyword arguments. The necessary components will be created in default shape, so

```
>>> ds = new_wordlist()
```

will only have a FormTable

```
>>> [table.url.string for table in ds.tables]
['forms.csv']
```

but it is possible to generate a dataset with more tables from scratch

```
>>> ds = new_wordlist(
...     FormTable=[],
...     LanguageTable=[],
...     ParameterTable=[],
...     CognatesetTable=[],
...     CognateTable=[])
>>> [table.url.string for table in ds.tables]
['forms.csv', 'languages.csv', 'parameters.csv', 'cognatesets.csv', 'cognates.csv']
>>> sorted(f.name for f in ds.directory.iterdir())
['Wordlist-metadata.json', 'cognates.csv', 'cognatesets.csv', 'forms.csv',
↪'languages.csv', 'parameters.csv']
```

### lexedata.util.simplify_ids module

`lexedata.util.simplify_ids.`**`clean_mapping`**`(`*rows: Mapping[str, Mapping[str, str]]*`)` → Mapping[str, str]

Create unique normalized IDs.

```
>>> clean_mapping({"A": {}, "B": {}})
{'A': 'a', 'B': 'b'}
```

```
>>> clean_mapping({"A": {}, "a": {}})
{'A': 'a', 'a': 'a_x2'}
```

`lexedata.util.simplify_ids.`**`simplify_table_ids_and_references`**`(`*ds: lexedata.types.Wordlist[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID], table: csvw.metadata.Table, transparent: bool = True, logger: logging.Logger = <Logger lexedata (INFO)>*`)` → bool

Simplify the IDs of the given table.

`lexedata.util.simplify_ids.`**`update_ids`**`(`*ds: pycldf.dataset.Dataset, table: csvw.metadata.Table, mapping: typing.Mapping[str, str], logger: logging.Logger = <Logger lexedata (INFO)>*`)`

Update all IDs of the table in the database, also in foreign keys, according to mapping.

`lexedata.util.simplify_ids.`**`update_integer_ids`**`(`*ds: pycldf.dataset.Dataset*, *table: csvw.metadata.Table*, *logger: logging.Logger = <Logger lexedata (INFO)>*`)`

Update all IDs of the table in the database, also in foreign keys.

### Module contents

**`class`** `lexedata.util.`**`KeyKeyDict`**

Bases: `Mapping[str, str]`

## 1.4.6 lexedata.cli module

**`class`** `lexedata.cli.`**`ChangeLoglevel`**`(`*option_strings*, *dest*, *const*, *nargs=None*, ***kwargs*`)`

Bases: `argparse.Action`

**`class`** `lexedata.cli.`**`Exit`**`(`*value*`)`

Bases: `enum.IntEnum`

An enumeration.

**`CLI_ARGUMENT_ERROR = 2`**

**`FILE_NOT_FOUND = 10`**

**`INVALID_COLUMN_NAME = 6`**

`INVALID_DATASET = 8`

`INVALID_ID = 5`

`INVALID_INPUT = 9`

`INVALID_TABLE_NAME = 7`

`NO_COGNATETABLE = 3`

`NO_SEGMENTS = 4`

**class** `lexedata.cli.`**`SetOrFromFile`**(*option_strings*, *dest*, *nargs='+'*, *default=<lexedata.types.WorldSet object>*, *help=None*, *autohelp=True*, *metavar=None*, ***kwargs*)

Bases: `argparse.Action`

`lexedata.cli.`**`add_log_controls`**(*parser: argparse.ArgumentParser*)

`lexedata.cli.`**`enum_from_lower`**(*enum: Type[enum.Enum]*)

`lexedata.cli.`**`parser`**(*name: str*, *description: str*, ***kwargs*) → argparse.ArgumentParser

`lexedata.cli.`**`setup_logging`**(*args: argparse.Namespace*)

`lexedata.cli.`**`tq`**(*iter*, *task*, *logger=<Logger lexedata (INFO)>*, *total: typing.Optional[typing.Union[int, float]] = None*)

## 1.4.7 lexedata.error_handling module

**exception** `lexedata.error_handling.`**`MultipleCandidatesWarning`**

Bases: `UserWarning`

**exception** `lexedata.error_handling.`**`ObjectNotFoundWarning`**

Bases: `UserWarning`

`lexedata.error_handling.`**`create`**(*db_object: Dict[str, Any]*, *cell: Optional[str] = None*) → bool

Should I add a missing object? Yes, quietly.

Give permission to add the object.

**Returns** **True**

**Return type** The object should be added.

`lexedata.error_handling.`**`error`**(*db_object: Dict[str, Any]*, *cell: Optional[str] = None*) → bool

Should I add a missing object? No, the object missing is an error.

Raise an exception (ObjectNotFoundWarning) reporting the missing object and cell.

**Raises** *`ObjectNotFoundWarning`* –

`lexedata.error_handling.`**`ignore`**(*db_object: Dict[str, Any]*, *cell: Optional[str] = None*) → bool

Should I add a missing object? No, drop it quietly.

**Returns** **False**

**Return type** The object should not be added.

`lexedata.error_handling.`**`warn`**`(`*db_object: Dict[str, Any]*, *cell: Optional[str] = None*`)` → bool

Should I add a missing object? No, but inform the user.

Send a warning (ObjectNotFoundWarning) reporting the missing object and cell.

**Returns** **False**

**Return type** The object should not be added.

`lexedata.error_handling.`**`warn_and_create`**`(`*db_object: Dict[str, Any]*, *cell: Optional[str] = None*`)` → bool

Should I add a missing object? Yes, but inform the user.

Send a warning (ObjectNotFoundWarning) reporting the missing object and cell, and give permission to add the object.

**Returns** **True**

**Return type** The object should be added.

## 1.4.8 lexedata.types module

**class** `lexedata.types.`**`CogSet`**

Bases: *`lexedata.types.RowObject`*

cognate set

**class** `lexedata.types.`**`Concept`**

Bases: *`lexedata.types.RowObject`*

concept

**class** `lexedata.types.`**`Form`**

Bases: *`lexedata.types.Object`*

form

**class** `lexedata.types.`**`Judgement`**

Bases: *`lexedata.types.RowObject`*

cognate judgement

**class** `lexedata.types.`**`KeyKeyDict`**

Bases: `Mapping[str, str]`

**class** `lexedata.types.`**`Language`**

Bases: *`lexedata.types.Object`*

language

**class** `lexedata.types.`**`Object`**

Bases: `Dict[str, Any]`

**class** `lexedata.types.`**`Reference`**

Bases: *`lexedata.types.Object`*

reference

**class** `lexedata.types.`**`RowObject`**

Bases: *`lexedata.types.Object`*

A row in a lexical dataset, i.e. a concept or cognateset `class lexedata.types.Source`

Bases: `lexedata.types.Object`

source

`class lexedata.types.Wordlist`(*tablegroup: csvw.metadata.TableGroup*)

Bases: `pycldf.dataset.Wordlist`, `Generic[lexedata.types.Language_ID, lexedata.types.Form_ID, lexedata.types.Parameter_ID, lexedata.types.Cognate_ID, lexedata.types.Cognateset_ID]`

`class lexedata.types.WorldSet`

Bases: `Generic[lexedata.types.H]`

`intersection`(*other: Union[lexedata.types.WorldSet[lexedata.types.H], Set[lexedata.types.H]]*) → Union[*lexedata.types.WorldSet*[lexedata.types.H], Set[lexedata.types.H]]

## 1.5 Glossary

**Central concept** The 'central' meaning among all forms that derive from the same etymological root. This can be a the reconstructed meaning of the root in a proto-language, but it can e.g. also be a less rigid shorthand for the central meaning of polysemous forms. Central concepts can be assigned manually, or automatically using `lexedata.edit.add_central_concepts`. For the automatic assignment, lexedata uses the colexification patterns present in the CLICS database.

**See also:**

*central concept AH*

**Central concept AH** An absence heuristic. One of two ways to assign absences in the case of *root presence coding*, based on assigning each cross-concept cognate set to a concept (the central concept). Then, a `0` would be coded if the *central concept* of a *cross-concept cognate set* is expressed by a different root. This is the same as the way absences are assigned with the *root-meaning coding* method. The results are however different, since in root presence coding items that have undergone semantic shift are included thus forming a less "sparse" matrix. Central concepts are in this operation treated as the most likely concept where a reflex of a form would be found. If they are expressed with a different root, then we conclude that the root in question must be absent.

**See also:**

*half primary concepts AH*

**Cross-concept cognate set** A cognate set that includes all descendants or reflexes of a protoform irrespective of their meaning (i.e. including items that have undergone semantic shift). In traditional historical linguistics words are termed cognate if they share a common protoform and they have been passed down to daughter languages from a common ancestor through vertical transmission (i.e. no borrowing has occured). According to this definition, while it is expected that the meaning of cognate words is related, it doesn't have to be identical. In many phylogenetic studies the term "cognate set" has been used for sets of words that derive from a common protoform and additionally have the same meaning. In this manual we are explicit by distinguishing between cross-concept cognate sets and within-concept cognate sets. Lexedata can work with both, but there are some functionalities that only make sense with a particular kind of cognate sets. Also, keep in mind that once cross-concept cognate sets are constructed, then the derivation of within-concept cognate sets is trivial (and lexedata can do it automatically).

**See also:**

*within-concept cognate set*

**Export** Lexedata is CLDF-centric, so 'export' is always 'away from CLDF'.

**Half primary concepts AH** An absence heuristic. One of two ways to assign absences for root presence coding, based each *primary concept* associated with the root in question (for all languages in a dataset), instead of privileging one of them (the *central concept*).

More precisely, a root is deemed absent when at least half of the primary concepts associated with this root are expressed by other roots for a given language. For example, a cross-concept cognate set may include items that mean (in different languages) HEAD, HAIR, and TOP OF THE HEAD. Let us assume that HEAD and HAIR were among the primary concepts, while TOP OF THE HEAD was not. Then for a given language the root in question would be coded as absent if at least one (half of the two) primary concepts HEAD and HAIR is expressed by a *different* root. Only if we don't know terms for both HEAD and HAIR in this language – or generally, if more than half of the primary concepts associated to the root are missing –, then the root in question would be assigned a `?`.

**Import** Lexedata is CLDF-centric, so 'import' is always 'towards CLDF'.

**Missing form** A missing form is a special type of null (non-existent) form representing explicit missing data in the dataset and it has an empty form field (`""`). There are in total three types of null forms that can be represented a dataset:

- missing forms: These forms have been searched for and are unknown according to a sources. They are represented as form row with an empty #form column in the dataset.
- not-entered forms: These forms do not have any representation in the dataset and correspond to data not yet retrieved or searched for in sources.
- *NA form*-s, with form `-`

**Multistate coding** One of three *phylogenetic coding method*'s implemented in lexedata. In this coding method, each *primary concept* corresponds to a multistate character, with each within-concept cognate set corresponding to a different state. It is available for datasets with either within- or cross-concept cognate sets.

**NA form** An NA form is a special type of null (non-existent) form corresponding to a concept not present in the language in question and it is represented with a dash `-`. For example, it is possible that terms for particular species of flora and fauna, or even for natural phenomena, such as snow, do not exist in a language. Another case could be color terms. In a dataset, it is possible that a concept is present in some languages, but not in others. An NA form conveys that the concept is not applicable to this language. It is in this way distinct from missing data, i.e. that we do not know the corresponding form for this concept in this language (but we assume there is one). NA forms are treated the same as missing data in many cases, but not all. In root-meaning coding, an NA form leads to absences `0` to all associated cognate sets, while a missing form leads to `?`.

**Phylogenetic coding method** The method used to derive a character matrix that can be used as input for phylogenetic analyses. There are three main coding methods for lexical data that have been used for phylogenetic analyses. We will briefly list them here.

1. *root-meaning coding*
2. *root presence coding*
3. *multistate coding*

**Primary Concept** A concept that has been systematically searched for in the languages of the dataset. When building a lexical dataset, it is typical procedure to start with a comparative wordlist including a number of basic concepts (e.g. a Swadesh list). Within lexedata, we call such concepts primary. Any other concepts present in parameters.csv are secondary. A dataset with within-concept cognate sets, often contains only primary concepts (however, it is possible that one has been keeping track of additional meanings for each word, thus leading to the inclusion of a number of secondary concepts as well.). A dataset with cross-concept cognate sets is very likely to include secondary concepts, especially if one has searched for cognate forms extensively among synonyms or closely related concepts to the primary concepts. Primary concepts matter for specific operations in lexedata. You can either provide a list of primary concepts or generate it through lexedata.report.filter if you have primary concepts annotated in your ParameterTable.

**See also:**

*half primary concepts AH*, *Secondary Concept*

**See also:**

*secondary concept*

**Root presence coding** One of three *phylogenetic coding method*'s implemented in lexedata. This coding method converts every cross-concept cognate set in the dataset into a binary character (with 1 denoting presence of a reflex of this root in the language and 0 absence). It can be used only when the dataset contains cross-concept cognatesets. Strictly speaking, any non-attestation of a reflex of a particular root in a language should lead to a ?, since we can almost never be sure that a root is indeed absent and it doesn't survive in some marginal meaning. This is even more true in cases of language families that have not been intensively studied. However, a character matrix consisting of 1s and ?s is not informative for phylogenetic analyses, so we need a heuristic to convert in a principled way some of these question marks to absencies. Lexedata provides two absence heuristics:

1. *central concept ah*
2. *half primary concepts ah*

**Root-meaning coding** One of three coding methods implemented in lexedata. This coding method converts every within-concept cognate set in the dataset into a binary character (with 1 representing presence of this root-meaning association in a particular language and 0 absence). When a root-meaning association is not attested in a language, the character is coded as 0 if the meaning in question is expressed with a different root, and as ? if the meaning is not attested at all. The root-meaning coding method can be used for datasets with either cross-concept or within-concept cognate sets.

**See also:**

*phylogenetic coding method*

**Secondary Concept** Any concept that has not been systematically searched for in the languages of the dataset. When building a lexical dataset, it is typical procedure to start with a comparative wordlist including a number of basic concepts (e.g. a Swadesh list). Within lexedata, we call such concepts, that have been systematically searched for, primary. Additionaly secondary concepts may be present in a dataset for various reasons: they may be secondary meanings of basic forms or correspond to forms that are cognate to other basic forms. A dataset with within-concept cognate sets, often contains only primary concepts (however, it is possible that one has been keeping track of additional meanings for each word, thus leading to the inclusion of a number of secondary concepts as well.). A dataset with cross-concept cognate sets is very likely to include secondary concepts, especially if one has searched for cognate forms extensively among synonyms or closely related concepts to the primary concepts.

**See also:**

*primary concept*

**Segment_Slice column** Segment_Slice is a column of the CognateTable that can be used to identify a particular section of the form, so that different parts of the form can be assigned to different cognate sets. This is part of the CLDF standard.

**Status column** A tracking column present in any of the cldf tables in order to facilitate workflow. Lexedata scripts can also update such columns with customizable messages to facilitate manual checking and tracking of automatic operations. This column is not part of the current v1.1 of the CLDF standard, which will treat it just as any other text column.

**Within-concept cognate set** A cognate set that includes descendants or reflexes of a protoform that additionally have the same meaning. While in traditional historical linguistics words are termed cognate if they share a common protoform irrespective of their meaning, in many phylogenetic studies the term "cognate set" has been used for sets of words that not only share an ancestral protoform but all express the same concept. In this manual we are explicit by distinguishing between cross-concept cognate sets and within-concept cognate sets. Lexedata can work with both, but there are some functionalities that only make sense with a particular kind of cognate sets.

Also, keep in mind that cross-concept cognate sets cannot be automatically derived from within-concept cognate sets (since this requires linguistic expertise), while the reverse is possible.

**See also:**

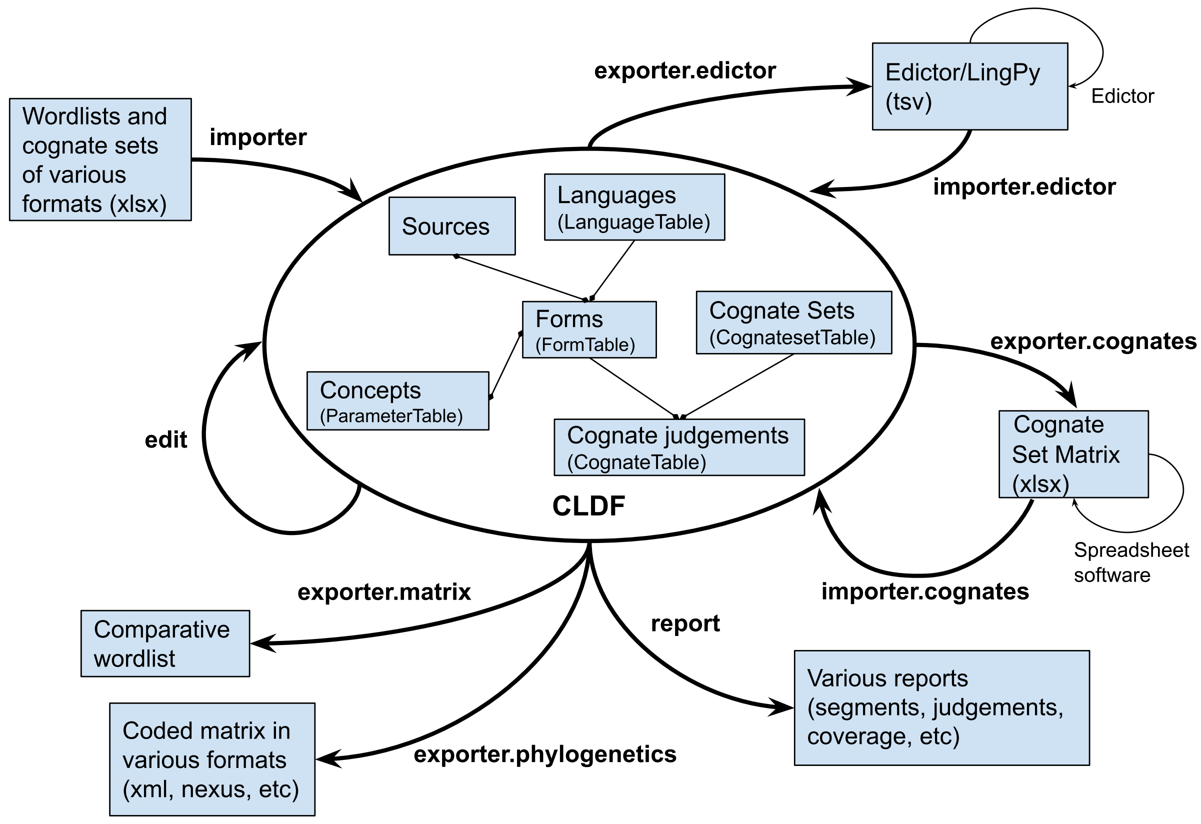
*cross-concept cognate set*

## 1.6 Summary

Lexedata is a collection of tools to support the editing process of comparative lexical data. Wordlists are a comparatively easily collected type of language documentation that is nonetheless quite data-rich and useful for the systematic comparison of languages [@list2021lexibank]. They are an important resource in comparative and historical linguistics, including their use as raw data for language phylogenetics [@gray2009language;@grollemund2015bantu].

The `lexedata` package uses the "Cross-Linguistic Data Format" (CLDF, @cldf11, @cldf-paper) as the main data format for a relational database containing forms, languages, concepts, and etymological relationships. The CLDF specification builds on top of the CSV for the Web (CSVW, @pollock2015metadata) specs by the W3C, and as such consists of one or more comma-separated value (CSV) files that get their semantics from a metadata file in JSON format.

Implemented in Python as a set of command line tools, Lexedata provides various helper functions to address issues that frequently arise when working with comparative wordlists for multiple languages, as shown in \autoref{fig:structure}. These include importing from and exporting to formats more familiar to linguists, as well as bulk edit functions and associated integrity checks. For example, there are scripts for importing data from MS Excel sheets of various common formats into CLDF, checking for homophones, manipulating etymological judgements, and exporting coded datasets for use in phylogenetic software.


exporter.edictor
Edictor/LingPy (tsv)
Edictor
Wordlists and cognate sets of various formats (xlsx)
importer
importer.edictor
Sources
Languages (LanguageTable)
Forms (FormTable)
Cognate Sets (CognatesetTable)
Concepts (ParameterTable)
Cognate judgements (CognateTable)
CLDF
edit
exporter.cognates
Cognate Set Matrix (xlsx)
Spreadsheet software
importer.cognates
exporter.matrix
Comparative wordlist
report
Various reports (segments, judgements, coverage, etc)
Coded matrix in various formats (xml, nexus, etc)
exporter.phylogenetics

## 1.7 Statement of Need

Maintaining the integrity of CLDF as a relational database is difficult using general CSV editing tools. This holds in particular for the usual dataset size of hundreds of languages and concepts, and formats unfamiliar to most linguists. Dedicated relational database software, which simplifies the maintenance of the data constraints, would set an even bigger hurdle to researchers, even to those who are reasonably computer-savvy.

The major existing tool for curating lexical datasets in other formats and providing them as CLDF for interoperability is cldfbench [@cldfbench]. However, cldfbench assumes that the data curator is not necessarily in a position to edit the dataset. As such, it provides a very flexible interface to transform and curate CLDF datasets, at the cost of making this accessible through an API which requires writing Python code.

Given that the majority of comparative linguists are unfamiliar with programming, Lexedata is designed to not need any programming skills. In contrast with cldfbench, Lexedata is written for the purpose of not only curating, but also collecting and editing the dataset. It therefore imposes additional constraints on the dataset which are very useful in editing tasks, but not strictly required by CLDF.

There are two major existing tools for editing lexical datasets, LingPy [@lingpy] and Edictor [@edictor]. Edictor is a browser-based graphical user interface tool to edit cognate annotations, while LingPy is a Python library focused on automating manipulations of lexical datasets, such as automatic cognate detection. Both of these pre-date the CLDF format, and while their common data format inspired some features of CLDF, it has some differences. Lexedata provides export and import functionality for this TSV-based format to and from CLDF. In addition, Lexedata exposes a major LingPy functionality, the Automatic Cognate Detection (ACD, @list2017potential) using Lexstat [@list2012lexstat], to work directly on CLDF datasets. This avoids both memory issues arising from LingPy's approach to load the entire dataset into memory and the need to convert between CLDF and LingPy.

Lexedata is designed to facilitate adding comments to cognate sets and cognate judgements, through the annotation tools in the Excel format (which naturally extend to comment threads in Google Sheets for collaborative editing), as well as tracking the editing workflow through status columns with customizable messages. Last but not least, to ensure that the user retains a good sense of control and overview, Lexedata includes helpful warning messages that suggest potential solutions and next steps to the user, while it keeps the user informed about batch operations with intermediate info messages and final reports.

In summary, Lexedata addresses the need to curate and edit a lexical dataset in CLDF format without the ability to program, which is still a rare skill among comparative linguists. It allows this without sacrificing the power and familiarity of existing software, such as GUI spreadsheed apps or Edictor, and by providing user-friendly access to format conversions and bulk editing functionality through simple terminal commands.

## 1.8 Research use

The extensive lexical dataset editing functionality is currently used by projects at UC Berkeley and Universität Zürich for Arawakan and Mawetí-Guaraní languages and at Universiteit Gent for Bantu. Precursor scripts have also been used for Timor-Alor-Pantar and Austronesian languages [@lexirumah-paper]. The export to phylogenetic alignments, derived from BEASTling [@maurits2017beastling;@beastling14], has been used in different language phylogenetics projects that are already under review [@kaiping2019subgrouping;@gunnink2022bantu].

## 1.9 Acknowledgement

Development of Lexedata was funded by the Swiss National Science Foundation (SNSF) Sinergia Project "Out of Asia" CRSII5_183578.

## 1.10 References

## 1.11 The CLDF format

The CLDF format is designed for sharing and reusing comparative linguistic data. A CLDF lexical dataset consists of a series of tables (.csv files), a metadata (.json) file describing the structure of each table and their inter-relationships, and a bibliography (.bib) file containing the relevant sources. A typical CLDF lexical dataset consists of the following tables: LanguageTable, ParameterTable (listing concepts), FormTable, CognatesetTable, and CognateTable. Not all files are necessary: the bare minimum for a valid CLDF dataset is a FormTable. However, lexedata requires a metadata file for almost every operation.

Each table (.csv file) has to have an ID column. Below, we briefly describe the typical files of a lexical CLDF dataset and how they interact with Lexedata when necessary. For each file, you can find below the necessary columns and typical columns. You can always add any custom columns as needed for your dataset. There is also the possibility to add further tables depending on your needs. For more information on the CLDF format, you can refer to https://cldf.clld.org/.

We recommend that you keep all these files in one folder which is versioned with git. You can use Github or Gitlab for this purpose.

### 1.11.1 The LanguageTable

The `languages.csv` file contains the different varieties included in your lexical dataset and their metadata. Every row is a variety.

- Necessary columns: Language_ID
- Typical columns: Name

### 1.11.2 The ParameterTable (concepts)

The `parameters.csv` file (sometimes also named `concepts.csv` in datasets with manually created metadata) contains the different concepts (meanings) included in your lexical dataset. Every row is a concept.

- Necessary columns: Parameter_ID
- Typical columns: Name, Definition, Concepticon_ID

### 1.11.3 The FormTable

The `forms.csv` file is the core of a lexical dataset. A well-structured `forms.csv` on its own, even without accompanying metadata, can already be understood as a lexical dataset by many CLDF-aware applications. The table contains all the different forms included in your dataset. Every row is a form with its associated metadata. To add page numbers for the sources in the cldf format, you should use the format source[page]. You can have different page numbers and page ranges within the square brackets (e.g. `smith2003[45, 48, 52-56]`).

- Necessary columns: Form_ID, Form, Concept_ID, Language_ID
- Typical columns: Comment, Segments, Source

### 1.11.4 The CognatesetTable

The `cognatesets.csv` file contains the cognate sets included in your dataset and their metadata. Every row is a cognate set. Note that, depending on the dataset, cognate set here can either mean cross-concept cognate set (all forms descending from the same protoform), or within-concept cognate set (all forms descending from the same protoform that have the same meaning).

- Necessary columns: Cognateset_ID
- Typical columns: Comment, Source

### 1.11.5 The CognateTable (judgements)

The `cognates.csv` file contains all the individual cognate judgements included in the dataset, i.e. it links every form to the cognateset it belongs to. Every row is a cognate judgement. Note that it is possible to have forms belonging to multiple cognate sets (e.g. to account for partial cognacy). Another way of accommodating partial cognacy is to assign parts of forms to different cognate sets, by using the Segment_Slice column.

- Necessary columns: Cognate_ID, Form_ID, Cognateset_ID
- Typical columns: Comment, Segment_Slice

### 1.11.6 The bibliography

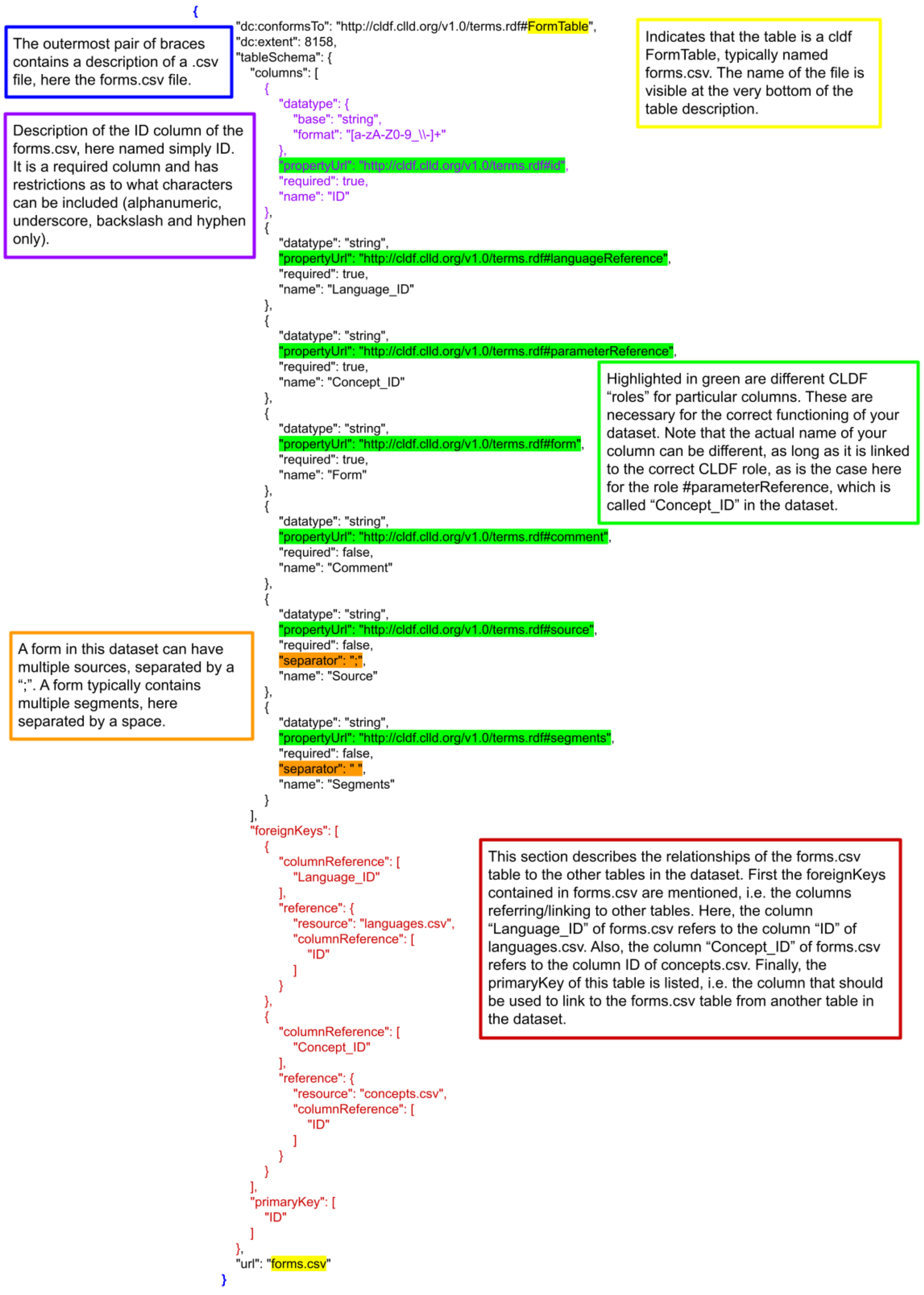
The `sources.bib` file contains references to all sources used in the dataset. The entries in the Source column of the forms.csv (or any other table) must be identical to a handle (unique code) of a reference in the `sources.bib`.

### 1.11.7 The metadata file

The `Wordlist-metadata.json` file contains a detailed description of all the CSV files, their columns and their interrelationships. It is not required for a CLDF dataset, but it is necessary for the vast majority of operations using lexedata. For simple datasets, lexedata can create automatically a metadata file. However, for more complex datasets, you would need to provide one yourself.

For an example of a simple metadata file, see XXX. Every change to the structure of the dataset (e.g. insertion or deletion of a column in any table) needs to be reflected in the netadata file for the dataset to be a valid cldf dataset.

Below you can see a typical description of a table in the metadata file.



```
{
  "dc:conformsTo": "http://cldf.clld.org/v1.0/terms.rdf#FormTable",
  "dc:extent": 8158,
  "tableSchema": {
    "columns": [
      {
        "datatype": {
          "base": "string",
          "format": "[a-zA-Z0-9_\\-]+"
        },
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#id",
        "required": true,
        "name": "ID"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#languageReference",
        "required": true,
        "name": "Language_ID"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#parameterReference",
        "required": true,
        "name": "Concept_ID"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#form",
        "required": true,
        "name": "Form"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#comment",
        "required": false,
        "name": "Comment"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#source",
        "required": false,
        "separator": ";",
        "name": "Source"
      },
      {
        "datatype": "string",
        "propertyUrl": "http://cldf.clld.org/v1.0/terms.rdf#segments",
        "required": false,
        "separator": " ",
        "name": "Segments"
      }
    ],
    "foreignKeys": [
      {
        "columnReference": [
          "Language_ID"
        ],
        "reference": {
          "resource": "languages.csv",
          "columnReference": [
            "ID"
          ]
        }
      },
      {
        "columnReference": [
          "Concept_ID"
        ],
        "reference": {
          "resource": "concepts.csv",
          "columnReference": [
            "ID"
          ]
        }
      }
    ],
    "primaryKey": [
      "ID"
    ]
  },
  "url": "forms.csv"
}
```

The outermost pair of braces contains a description of a .csv file, here the forms.csv file.

Indicates that the table is a cldf FormTable, typically named forms.csv. The name of the file is visible at the very bottom of the table description.

Description of the ID column of the forms.csv, here named simply ID. It is a required column and has restrictions as to what characters can be included (alphanumeric, underscore, backslash and hyphen only).

Highlighted in green are different CLDF “roles” for particular columns. These are necessary for the correct functioning of your dataset. Note that the actual name of your column can be different, as long as it is linked to the correct CLDF role, as is the case here for the role #parameterReference, which is called “Concept_ID” in the dataset.

A form in this dataset can have multiple sources, separated by a “;”. A form typically contains multiple segments, here separated by a space.

This section describes the relationships of the forms.csv table to the other tables in the dataset. First the foreignKeys contained in forms.csv are mentioned, i.e. the columns referring/linking to other tables. Here, the column “Language_ID” of forms.csv refers to the column “ID” of languages.csv. Also, the column “Concept_ID” of forms.csv refers to the column ID of concepts.csv. Finally, the primaryKey of this table is listed, i.e. the column that should be used to link to the forms.csv table from another table in the dataset.

## 1.12 A short introduction to the command line

While you may be used to driving applications by pointing and clicking with the mouse and very occasionally typing text, command-line interfaces (CLI) use text commands to drive computer programs. In some sense similar to human language, these text commands must obey a specific syntax to be understood. However, unlike human language, if you don't follow the syntax exactly, nothing will happen. This syntax powers compositionality, which makes automating complex or repetitive tasks easier – so it is appropriate for lexedata.

All lexedata tools are run from the command line, according to principles explained in the *lexedata Manual*. There are other excellent introductions to the power of the command line, but if you are completely new to it, the following section may help you get started.

### 1.12.1 Navigation using the command line

You can navigate to a specific folder using the command line on your terminal (also 'console' or 'command prompt' or 'CMD') as follows: You can see the directory (folder) you are in at the moment (current directory) within the prompt. In order to go to a directory below (contained in the current directory), type `cd [relative directory path]` (`cd` stands for *change directory*), e.g. `cd Documents/arawak/data`.

If you do not know the path you want to get to, you can open it in your file browser (eg. Finder or Explorer) and usually find it there, either in an address bar or in the folder properties. Sometimes, it also works to drag-and-drop the folder into your terminal.

Note that directory names are case sensitive and that they can be automatically filled in (if they are unique) by pressing the tab key. In order to go to a directory above (the directory containing the current directory), type `cd ..`. Note that you can type any path combining up and down steps. So, if I am in the data directory given as an example above, in order to go to the directory maweti-guarani which is within Documents, I can type `cd ../../maweti-guarani`.

At any point you can see the contents of your current directory by typing `ls` or `dir`, depending on your operating system. (Don't worry about typing the wrong one: the worst to happen is your system telling you "I don't know that command.")

## 1.13 Working with git

Git is a version control system. It keeps track of changes so you can easily revert to an earlier version or store a snapshot of your project (e.g. the state of the dataset for a particular article you published). While you could use Lexedata without git, we highly recommend storing your data in a repository (folder) versioned with git. In this section we are going to cover some basic commands to get you started. You can find more detailed instructions and more information in begin with git and also in the tutorials by Github. You can also download and use the GitHub Desktop application if you prefer to not use the *command line* to interact with GitHub. However, you do need to use the command line to interact with lexedata.

### 1.13.1 Setting up git

If you start with your own blank slate for a dataset, a `git init` in your dataset folder will set it up as a git repository. If you have a template or a dataset already on Github, `git clone https://github.com/USERNAME/REPOSITORY` will create a local copy for you. (You can manually move that local copy on your computer in case it ended up in the wrong place.)

Git may expect some more setup, e.g. to know your name and an email address to be credited as author of changes you commit, but it is generally good at telling you these things.

### 1.13.2 Basic git commands

Below we are going to describe the use of the most basic git commands. We assume a setup with a local git repository (on your computer) and a remote repository (e.g. on GitHub).

`git fetch`: This command "informs" the git on your computer about the status of the remote repository. It does *not* update or change any files in your local repository.

`git status`: This commands gives you a detailed report of the status of your local repository, also in relation to your remote repository. In the report you can see any changes that you have done to your local repository, and if they are commited or not, as well as any committed changes that have happened in the remote repository in the meantime.

`git pull`: With this command you can update your local repository to be identical to the remote one. This will not work if you have uncommitted changes in your local repository to protect your work.

`git add FILENAME`: This command adds new or modified files to git, or it "stages" the changes to be committed.

`git commit -m "COMMIT MESSAGE"`: After adding (or staging) all the changes that you want to commit, you can use this command to commit the changes to your local repository with an associated commit message. Typically the commit message contains a summary of the changes. This command will *not* update the remote repository.

`git push`: This command will push (or publish) your local commits to the remote repository, which will be updated to reflect these new changes.

To ensure dataset integrity, we recommend running `cldf validate Wordlist-metadata.json` before committing and pushing, so that any cldf errors are caught and corrected (see cldf-format-validation)).

CHAPTER

TWO

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX