Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Manipulating Lexical Forms with the PyLexibank FormSpec

Johann-Mattis List
Chair for Multilingual Computational Linguistics
University of Passau

Multilingual lexical data is typically stored in a wide variety of forms, based on many idiosyncratic decisions that vary from dataset to dataset. Here, a simple but efficient solution for the manipulation of lexical data in multilingual wordlists will be introduced. This solution, the PyLexibank FormSpec, was originall developed for the conversion of various kinds of lexical data to Cross-Linguistic Data Formats, but it can also be used as a standalone. This study offers a basic tutorial that illustrates how the FormSpec can be put to concrete use.

1 Introduction

When working with lexical data in multilingual settings, one encounters a huge variety of ways in which similar kinds of information about lexical forms are encoded. While handling such cases may often require targeted solutions that may in the worst case consist in manual corrections of individual data points, our work with the Lexibank repository (List et al. 2022) has also allowed us to detect certain recurring idiosyncracies in lexical data that can be handled with unified approaches. These approaches have been integrated into the PyLexibank software package (Forkel et al. 2021) and provide important help in converting various kinds of cross-linguistic data with lexical forms to Cross-Linguistic Data Formats (Forkel et al. 2018).

In this small study, I will introduce one particular solution that deals with lexical forms before their conversion to phonetic transcriptions. This solutions, as simple as it may seem, has helped us a lot in constructing the Lexibank repository that by now aggregates standardized data from more than 120 different datasets (Blum et al. 2025).

Computer-Assisted Language Comparison in Practice
Volume 8, Number 2
URL: https://calc.hypotheses.org/8877 DOI: 10.15475/calcip.2025.2.4

Published on 28/10/2025

2 Background

While the basic information that scholars want to provide in a dictionary or a wordlist can be described in pretty simple and straightforward terms, the techniques that scholars use in order to mark this kind of information in concrete datasets vary greatly. While the major information that we need to provide when listing word forms in a given language consists in a triple of language, form, and meaning (List 2014; Gévaudan 2007), linguistic practice adds various forms of complexity and inconsistency to this triple structure. Language names are rarely used in a standardized form, making it at times difficult to identify the varieties in question. Short glosses used to represent meanings are often highly idiosyncratic and can at times only be understood from the larger context of the concept list in which they are assembled (List, Cysouw, and Forkel 2016). Forms are given in a mix of transcriptions, orthographic information, often expanded by additional information that can often only be understood when taking detailed contextual information into account.

As an example, consider cases where data are given in tabular form, where columns represent languages and rows represent meanings, and word forms are placed into the respective cells. This format can be found in numerous publications and is considered some kind of a standard among many linguists. The problem of the format is, that it invites inconsistencies regarding the representation of the lexical forms. These inconsistencies surface in numerous occasions. Thus, if more than one word is found to express a given meaning in a given language, scholars use various ways to code for this, using characters like comman, semicolon, or slash as a delimiter when listing multipel word forms, with many datasets using different delimiters without any clear semantics attached to them. Another problem consists in the use of brackets, which are also used in multiple variants, ranging from square brackets over normal brackets to curly braces. Here again, semantics of bracket use are rarely consistent, ranging from reading variants over pronunciation differences to metainformation that relates to the language or the concept in question rather than to the word form itself. An additional problem consists in the explicit marking of missing data, which varies also greatly, ranging from empty cells over dashes invarious forms to explicit entries, such as “no data” or “missing entry”.

While inconsistencies may seem to be unproblematic when inspecting data directly by eyeballing them, they may cause huge problems when trying to digest data with the computer. If missing data is marked by an entry “no entry”, for example, it can easily occur that this entry will make its way into the final database, leading to the false impression that the word for “apple” in some language variety is “no entry” instead of being simply missing from the record.

The problems arising from variation in lexical entries in cross-linguistic datasets may not only sound funny but also evitable. One would expect computationally versed people to be able to spot or predict such problems when trying to convert a dataset to some standardized format. When dealing with idiosyncrasies of individual data, however, it is helpful to make use of some standardized routines that help to solve problems that often recur across different datasets in a unified way.

3 Getting Started with the PyLexibank FormSpec

When developing the framework that would later be used to feed the Lexibank repository with data (Blum et al. 2025; List et al. 2022), we started out with individual solutions to deal with inconsistencies in lexical entries. Lexical entries were thus dealt with on a case-to-case basis, using standard routines for text manipulation offered by Python. When adding more data, however, we began to realize that certain problems with lexical forms would recur with a certain regularity. Entries for missing data would be marked idiosyncratically, multiple forms within the same cell would be separated with different separation symbols, and brackets would force us to apply at times quite complex regular expressions.

In order to address these problems, a new functionality to handle lexical forms flexibly in a unified way was added to PyLexibank (Forkel et al. 2021), the library that we used to convert data that we would obtain in raw form from published resources into Cross-Linguistic Data Formats (Forkel et al. 2018). This FormSpec, as it is called in PyLexibank, addresses the three major problems summarized above. It deals with brackets (preferably removing everything that is inside a bracket, given that both additional morphemes and meta-information can both not be reliably interpreted when standardizing a form entry). It deals with separators used to describe several variants within the same cell of a data entry. Finally, it also deals with missing data, allowing users to provide a list of the symbol combinations used to indicate that a cell contains no data. Additionally, the FormSpec provides some basic cleaning operations of lexical forms, stripping certain characters from the form and applying standard Unicode normalization (Moran and Cysouw 2018: 17).

While the *FormSpec* is automatically applied whenever you use CLDFBench (Forkel and List 2020) and PyLexibank to create a CLDF dataset, you can also test its functionality directly in an interactive Python session. In order to get started, all you need is a fresh installation of the PyLexibank package, which you can easily obtain with the help of the Python package index `pip`.

```
$ pip install pylexibank
```

Equipped in this form, all you need to load the *FormSpec* is to import it from your interactive Python session or from within a Python script.

```
from pylexibank import FormSpec
```

In order to use the *FormSpec*, you must initialize it first. This means, you predefine its behavior in cleaning a given lexical form. The call signature of the class is as shown below.

```
class FormSpec(builtins.object)
 |  FormSpec(
 |      brackets={'(': ')'},
 |      separators=(';', '/', ','),
 |      missing_data=('?', '-'),
 |      strip_inside_brackets=True,
 |      replacements=NOTHING,
 |      first_form_only=False,
 |      normalize_whitespace=True,
 |      normalize_unicode=None
 |  ) -> None
```

We define pairs of brackets by means of a dictionary in which the key is the opening bracket and the value is the closing bracket. This would not work with cases where a bracket is defined by the same start and end symbol, but our experience shows that most datasets would use traditional brackets for which start and end symbols are defined. The separators handle multiple forms for the same concept. Missing data are passed as a list (or more strictly speaking, a tuple, according to the call, but a list will also be accepted). If the option `strip_inside_brackets` is set to `True`, this means that the algorithm deletes content inside brackets. With respect to the order of execution, note that in cases where a separator, used as a separator of multiple word forms, is also passed inside a bracket, the algorithm would not split the text at this point, but first identify the brackets in the text and then apply the segmentation operation. The option `first_form_only` will yield only the first form of multiple potential forms, when set to `True`. Normalization can be done with respect to whitespace (deleting and unifying whitespace) and Unicode (where one would have to choose between NFD and NFC). The option replacements allows to define a list consisting of tuples of source-target strings, where the source string is what will be replaced and the target string is the replacement.

Having initialized the *FormSpec* by calling the class with particular parameters, one can use it by calling its split-method with two arguments, the first argument being always None when using it outside the context of CLDFBench, while the second argument is the string one wants to manipulate. This is illustrated in the following example.

```
>>> fs = FormSpec()
>>> for form in fs.split(None, "this, is; a (form)"):
...     print(form)
this
is
a
```

4 Usage Examples

When dealing with the *FormSpec*, it is important to be aware about the order by which actions are carried out when using the functionality. In the following, we will go through some examples that illustrate basic use-cases. We start with the handling of brackets, which are – as I mentioned before – defined as a dictionary (opening bracket as a key, closing bracket as the value). This allows us to define all kinds of potentially strange brackets that could occur in one's data.

```
>>> fs = FormSpec(brackets={"<": ">", "{": "}"})
>>> fs.split(None, "this <really?>, is, an {example}")
['this', 'is', 'an']
```

The `missing_data` argument allows you to specify any string that could occur as missing data. The FormSpec generally assumes that whitespace to the left or the right of the string will be stripped.

```
>>> fs = FormSpec(missing_data=("???", "?"))
>>> fs.split(None, "???, really,?, ")
['really']
```

For separators, there is a particular restriction that only single-character strings can be used as a separator. Thus, passing a string of more than one character will throw an error. Depending on the data, however, one can find workarounds that would nevertheless allow us to separate even strings where multiple characters have been used as a separator. As an example, consider the following output, where three slashes have been used as a separator.

```
>>> fs = FormSpec(separators=(',', ';', '/'))

>>> fs.split(None, "hallo /// welt / hier / bin ////
ich")
['hallo', 'welt', 'hier', 'bin', 'ich']
```

According to the way in which FormSpec works, the internal splitting process will only return those forms that consist of at least one character that is not a whitespace character. The *FormSpec* splits the string in the example into 10 different forms, but only five are returned, since they are note empty.

If you want to use the option to replace strings by other strings during the form conversion with the *FormSpec*, it is important to keep in mind that the replacement is carried out after all splitting operations have been carried out. This limits the possibilities of application, on the one hand, but it also reduces complexity, since the replacements are quite restricted and they do not interfere with the process of splitting a string into several forms. As an example, consider the following lines, where the replacement of the string /x/ to the string / is not carried out, given that / is also defined as a character that triggers the string to be split into parts.

```
>>> fs = FormSpec(
...     separators=(',', ';', '/'),
...     replacements=[('/x/', '/')]
...     )
>>> fs.split(None, "hallo /// welt / hier / bin /x/ ich")
['hallo', 'welt', 'hier', 'bin', 'x', 'ich']
```

5 Outlook

Although the *FormSpec* is based on a limited number of options, the functionality has proven very useful in practice, especially when populating the Lexibank repository (Blum et al. 2025). It seems that the decision to limit the scope of the method to a very dedicated range of options, deciding, among others, against the possibility to apply regular expressions, was helpful, given that the results triggered by the current *FormSpec* can still be easily understood when considering input and output strings. With more complex operations, we would quickly loose the possibility to trace individual decisions made in the code we used to convert raw data into standardized CLDF data points.

References

Blum, Frederic, Carlos Barrientos, Johannes Englisch, Robert Forkel, Simon J. Greenhill, Christoph Rzymski, and Johann-Mattis List. 2025. “Lexibank 2: Pre-Computed Features for Large-Scale Lexical Data [version 2; peer review: 3 approved].” Open Research Europe 5 (126): 1–24. https://doi.org/https://doi.org/10.12688/openreseurope.20216.2.

Forkel, Robert, Simon J Greenhill, Hans-Jörg Bibiko, Christoph Rzymski, Tiago Tresoldi, and Johann-Mattis List. 2021. PyLexibank. The Python Curation Library for Lexibank [Software Library, Version 2.8.2]. Geneva: Zenodo. https://doi.org/10.5281/zenodo.2630582.

Forkel, Robert, and Johann-Mattis List. 2020. “CLDFBench. Give Your Cross-Linguistic Data a Lift.” In Proceedings of the Twelfth International Conference on Language Resources and Evaluation, 6997–7004. Luxembourg: European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2020/pdf/2020.lrec-1.864.pdf.

Forkel, Robert, Johann-Mattis List, Simon J. Greenhill, Christoph Rzymski, Sebastian Bank, Michael Cysouw, Harald Hammarström, Martin Haspelmath, Gereon A. Kaiping, and Russell D. Gray. 2018. “Cross-Linguistic Data Formats, Advancing Data Sharing and Re-Use in Comparative Linguistics.” Scientific Data 5 (180205): 1–10. https://doi.org/10.1038/sdata.2018.205.

Gévaudan, Paul. 2007. Typologie Des Lexikalischen Wandels: Bedeutungswandel, Wortbildung Und Entlehnung Am Beispiel Der Romanischen Sprachen. Tübingen: Stauffenburg.

List, Johann-Mattis. 2014. Sequence Comparison in Historical Linguistics. Düsseldorf: Düsseldorf University Press. https://doi.org/10.1515/9783110720082.

List, Johann-Mattis, Michael Cysouw, and Robert Forkel. 2016. “Concepticon. A Resource for the Linking of Concept Lists.” In Proceedings of the Tenth International Conference on Language Resources and Evaluation, edited by Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, 2393–2400. Luxembourg: European Language Resources Association (ELRA). https://aclanthology.org/L16-1379/.

List, Johann-Mattis, Robert Forkel, Simon J. Greenhill, Christoph Rzymski, Johannes Englisch, and Russell D. Gray. 2022. “Lexibank, a Public Repository of Standardized Wordlists with Computed Phonological and Lexical Features.” Scientific Data 9 (316): 1–31. https://doi.org/10.1038/s41597-022-01432-0.

Moran, Steven, and Michael Cysouw. 2018. The Unicode Cookbook for Linguists: Managing Writing Systems Using Orthography Profiles. Berlin: Language Science Press. https://langsci-press.org/catalog/book/176.

Funding Information

This project has received funding from the European Research Council (ERC) under the European Union's Horizon Europe research and innovation programme (Grant agreement No. 101044282). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.