LingPy Datastructures

Preliminaries

In this notebook, some basic concepts behind LingPy datastructures shall be illustrated. All examples are based on the development version of LingPy (2.2).

```
In [1]: from lingpy import *
```

Basic Input Format

Let us first have a look at the file we want to work with. This file is a simple sample of four Slavic languages, Russian, Polish, Czech and Bulgarian. Let's look at its basic format.

```
In [2]: print(''.join(open("slavic.qlc").readlines()[:9]))
```

```
# Wordlist

# DATA
ID      CONCEPT LANGUAGE        IPA
#
1       all     Russian fsʲe
2       all     Polish  fʃɨsʦɨ
3       all     Bulgarian       vsiʧki
4       all     Czech   fʃɪxnʲɪ
```

This input format is pretty straightforward: We have a basic csv-structure with tabstops as separators, and hash-symbols as comment symbols (they are ignored and only help to account for pretty formatting). The first "real" line indicates the content of the columns, and the following lines contain the data.

Note that the order of the columns is strictly arbitrary in this format: Whether we put "CONCEPT" first, or "LANGUAGE", does not make a difference. Similarly, case is ignored (upper case is just for convenience in order to distinguish the header from the rest of the data for the user looking at the raw text file). Internally, once read into LingPy, all headers are represented in lower case.

The Wordlist Class

We open this file in LingPy by using the **Wordlist** class. This is the class that provides the basic datastructure used in most LingPy calculations. It defines generic methods for input, output, modification of data, and calculation of new data, that are regularly used in all daugther classes (**LexStat**, **Alignments**, **PhyBo**) that are used for specific calculations (cognate judgments, alignment analyses, borrowing detection). We load the wordlist by simply calling the class with the file containing the data as first argument

```
In [3]: wl = Wordlist('slavic.qlc')
```

Once we have loaded the data, we can illustrate some specific things that a are handled by a Wordlist. For example, we can check the length of the Wordlist, which is defined as the *number of words* in the dataset.

```
In [4]: len(wl)

Out[4]: 454
```

We can check width (*number of languages*) and height (*number of concepts*):

```
In [5]: wl.width,wl.height

Out[5]: (4, 110)
```

In principal, a wordlist is a very flexible format that retrieves all its "hard" information from a rc-file provided as a default in LingPy. This file provides aliases and data types for the data that is loaded into a wordlist. The idea is a three-dimensional represent of (linguistic) data. The first dimension is called **col** (*column*, usually "language"), the second one is called **row** (*row*, usually "concept"), the third is called **entry**, and in contrast to the first two dimensions, which have to consist of unique items, it contains flexible values, such as "ipa" (phonetic sequence), "cogid" (identifier for cognate sets), "tokens" (tokenized representation of phonetic sequences), etc.

Accessing Data

The key idea of the `Wordlist` class is that each the multi-dimensional data used in linguistic calculations can be displayed as a table in which columns represent languages and rows represent concepts. Slicing according to a given entry is easily done by accessing the respective entry as a class-attribute. In the resulting two-dimensional list, missing values are given the value `0`.

```
In [6]: ipa_strings = wl.ipa
        print(ipa_strings[0])
        print(ipa_strings[1])

        ['kratək', 'kraːtkiː', 'krutki', 'kɐrotkʲɪj']
        ['kɤs', 0, 0, 0]
```

Note that for `Wordlist` every value in the original data is an entry. However, the entries which are identified with `col` and `row` are protected and yield unnested output.

```
In [7]: print(wl.concept[0:10])
        print(wl.taxa)

        ['all', 'ashes', 'bark', 'belly', 'big', 'bird', 'bite (V)', 'black',
        'blood', 'bone']
        ['Bulgarian', 'Czech', 'Polish', 'Russian']
```

Apart from the class-attributes which are built on the fly, two important methods for specific slicing are provided by `Wordlist`, the `get_list()` method and the `get_dict()` method. The former returns the entries (as specified) for a given language or concept.

```
In [8]: russian_words = wl.get_list(col="Russian",entry = "ipa")
        words_for_hand = wl.get_list(row="hand",entry="ipa")
        print(russian_words[0:10])
        print(words_for_hand)

        ['kɐrotkʲɪj', 0, 'ɪtʲi', 'jɪjʦo', 'glas', 'rot', 'prʲɪjtʲi', 'prʲɪxɐdʲitʲ',
        'ɐgonʲ', 'znatʲ']
        [['rəka', 'rʊka', 'rɛ̃ka', 'ruka']]
```

Note the difference between the output here: The first list is not nested, while the second is. The reason is, that languages can have multiple words for the same meaning. If multiple words occur, they will be represented in additional lists.

The latter method returns a dictionary representation of the same data.

```
In [9]: russian = wl.get_dict(language="Russian",entry="ipa")
        print(russian['hand'])
```

```
['ruka']
```

Note that in this case we used `language` as a keyword instead of `col`. This is possible, since in the `rc-file` for our `Wordlist` in LingPy, an alias was defined for `col`. There are more aliases available, and they can easily be changed by modifying the `rc-file` which is always loaded before accessing a `Wordlist`.

```
In [10]: russian = wl.get_dict(doculect="Russian",entry="ipa")
         print(russian['foot'])
```

```
['nega']
```

Note that in contrast to the class attribute that allows us to access `Wordlist` entries in a two-dimensional list, the `col` and `row` attributes (`language`, `concept`) are not protected by this method. Calling either `get_list()` or `get_dict()` with `entry="language"` will return the (rather strange) output of all languages of a given word of a given language.

```
In [11]: russian_language = wl.get_dict(doculect="Russian",entry="language")
         print(russian_language['foot'])
```

```
['Russian']
```

As a last important feature of the two methods, consider the output that you get if *no* argument is passed for the `entry` keyword.

```
In [12]: russian_ids = wl.get_list(doculect="Russian")
         print(russian_ids[0:10])
```

```
[430, 0, 378, 100, 104, 233, 65, 69, 117, 185]
```

If `entry` is not specified, the `Wordlist` returns a list if IDs. These IDs are the same IDs that are defined in the input file under the (protected) ID column. As in the other methods for data accessing the `0` ID indicates a missing entry. In order to return only existing entries in the `get_list()` method, set the `flat` keyword to `True`.

```
In [13]: russian_ids = wl.get_list(doculect="Russian",flat=True)
         print(russian_ids[0:10])
```

```
[430, 378, 100, 104, 233, 65, 69, 117, 185, 18]
```

Using either the `get_dict()` or the `get_list()` method to retrieve the IDs of the words in the dataset may turn out to be useful if one wants slice the data in specific ways. Here, another important way to access data in a `Wordlist` becomes important. According to procedure, a `Wordlist` can be treated as a simple dictionary, with the key being the unique ID.

```
In [14]: wl[1]
```

```
Out[14]: ['all', 'Russian', 'fsʲe']
```

In this way, we can likewise *iterate* over all keys in a wordlist.

```
In [15]: for key in wl:
             if key < 10:
                 print(wl[key])
```

```
['all', 'Russian', 'fsʲe']
['all', 'Polish', 'fʃɨstɕɨ']
['all', 'Bulgarian', 'vsit͡ʃki']
['all', 'Czech', 'fʃɪxnʲɪ']
['ashes', 'Russian', 'zɐla']
['ashes', 'Polish', 'pɔpʲuw']
['ashes', 'Bulgarian', 'pɛpɛl']
['ashes', 'Czech', 'popɛl']
['ashes', 'Russian', 'pʲepɪlʲ']
```

But this is not all! We can use the same names we indicated for the entries in the input file (and all their aliases defined in the `rc-file`) to access only specific entries for a corresponding key.

```
In [16]: for key in wl:
             if key < 3:
                 print(wl[key,"ipa"])
             elif key < 6:
                 print(wl[key,"language"])
             elif key < 10:
                 print(wl[key,"concept"])
```

```
fsʲe
fʃɨstɕɨ
Bulgarian
Czech
Russian
ashes
ashes
ashes
ashes
```

The above demonstrated methods and techniques for slicing provide a high degree of flexibility for handling wordlists in your code. Keeping in mind the basic idea of aliasing entries, accessing datapoints from different perspectives, virtually all transformations of complex data can be done within just a few lines of code.

Manipulating Data

For the manipulation of data, the `Wordlist` class offers two important methods. The first and most important one (the second is treated in detail below), the `add_entries()` method, adds or modifies a new entry by passing the name of the (new) entry, the source entry for modification, and the function to modify. Using this method, we can add `tokens` (tokenized representations of phonetic sequences) to our current wordlist, using the `ipa2tokens()` function.

```
In [17]: wl.add_entries('tokens','ipa',lambda x:ipa2tokens(x))
         print(wl.tokens[0])
```

```
[['k', 'r', 'a', 't', 'ə', 'k'], ['k', 'r', 'aː', 't', 'k', 'iː'], ['k',
'r', 'u', 't', 'k', 'i'], ['k', 'ɐ', 'r', 'o', 't', 'kʲ', 'ɪ', 'j']]
```

What happens here is that the basic internal representation of `Wordlist` data as a dictionary with integer keys for each word and a lists as values is extended by one `entry`. The basic values of a `Wordlist` along with their position in this internal representation are stored in the `header` attribute.

```
In [18]: wl.header
```

```
Out[18]: {'concept': 0, 'doculect': 1, 'ipa': 2, 'tokens': 3}
```

The `ipa2tokens()` function is a rough, but flexible way to tokenize IPA-encoded sequences. The full specification of the method reads as:

```
ipa2tokens(
    istring,
    diacritics=None,
    vowels=None,
    tones=None,
    combiners='͜͡',
    breaks='.-',
    stress="ˈˌ'",
    merge_vowels=True
    )
```

Note that all `None` values in this specification indicate that default values are used instead of user-defined values. The basic idea of the function is that the user passes the basic classes of characters (diacritics, vowels, tones, combining characters, break characters, and stress characters), with consonants being defined negatively as all the remaining characters. The function then parses each string accordingly, combining diacritics with preceeding characters, vowels with vowels, if `merge_vowels` is set to `True`, and returns a tokenized representation (a `list`) of the original string.

```
In [19]: ipa2tokens("mybloodyhalloween",vowels="aeiouy")
```

```
Out[19]: ['m', 'y', 'b', 'l', 'oo', 'd', 'y', 'h', 'a', 'l', 'l', 'o', 'w', 'ee',
'n']
```

We can now do the same with the tokens what we have done with the other data before. One difference, however, is that the `tokens` are stored as a list and not as a string. This is also defined in the `rc-file`. If tokens are passed in the input file, they are stored as space-separated strings, and the `rc-file` determines that `tokens` are represented as a list if they are loaded into a `Wordlist`.

```
In [20]: wl[1,'ipa'],wl[1,'tokens']
```

```
Out[20]: ('fsʲe', ['f', 'sʲ', 'e'])
```

Just in order to illustrate how the internal representation of data is handled by `Wordlist`, let's add another entry. We convert the tokens into sound-class strings using the `tokens2class` method and see what happens to the header.

```
In [21]: wl.add_entries('classes','tokens',lambda x:''.join(tokens2class(x,model='sca'))
         wl.header
```

```
Out[21]: {'classes': 4, 'concept': 0, 'doculect': 1, 'ipa': 2, 'tokens': 3}
```

Output

In order to write wordlist data back to file, there are two important methods, one for internal, csv-output, one for external, formatted output (`txt`, `html`, `tex`). The first method is the `output()` method. Its basic use is straightforward: By passing the desired format as a first argument, this format is automatically written to the file whose name can be specified. So far, we haven't done any calculations, so all we can do is to write the data to a csv-file.

```
In [22]: wl.output('qlc',filename='slavic2')
```

Or we can only write out all the taxa in the data.

```
In [23]: wl.output('taxa',filename='slavic2')
         print(open('slavic2.taxa').read())

          Bulgarian
          Czech
          Polish
          Russian
```

The most important output format here is the `csv`-format. When writing wordlist data to `csv`, there are more possibilities. For example, we can specify that only specific columns should be written to file:

```
In [24]: wl.output('qlc',filename='slavic_columns',subset=True,cols=['language','concept
```

Note that in order to write subsets to `csv`, we first need to set the `subset` keyword to `True`. Using the `cols` keyword, we can specify which columns and in which order should be written to file. The `formatter` keyword can be used to guarantee "pretty" formatting of the `csv`-file by grouping the words according to the specified entry with help of (meaningless) hash (#) symbols.

A further way to export data is by defining specific conditions for the rows. Here, the code is a bit more complex, since, internally, the `eval()` function is used to evaluate whether a specific condition for the given row holds. The conditions are given in the `rows` keyword, which is a dictionary with the column whose condition should be checked as key and a Python statement for evaluation passed as a string. The following line thus writes only Russian and Polish language entries to file.

```
In [25]: wl.output(
             'qlc',
             filename='slavic_rows',
             subset=True,
             rows=dict(
                 language="in ['Russian','Polish']"
                 ),
             formatter="concept"
             )
```

The second method, the `export()` method structures the data according to specific `sections` and `entries`. Its use is a bit complicated, but it offers straightforward output to html, tex, or simple structured text.

```
In [26]: wl.export(
             'html',
             sections = dict(
                 h1 = ('concept','<h1>Concept: "{0}"</h1>\n'),
                 h2 = ('language','{0} '),
                 ),
             entries = [
                         ('ipa','<span class="ipa">[{0}]</span>\n'),
                         ('classes','<code>"{0}"</code>\n<br>'),
                         ],
             filename = "slavic_export"
             )
```

```
In [27]: print(''.join(open('slavic_export.html').readlines()[0:10]))
```

```
<h1>Concept: "all"</h1>
Russian <span class="ipa">[fsʲe]</span>
<code>"BSE"</code>
<br>Polish <span class="ipa">[fʃɨsʨɨ]</span>
<code>"BSISCI"</code>
<br>Czech <span class="ipa">[fʃɪxnʲɪ]</span>
<code>"BSIGNI"</code>
<br>Bulgarian <span class="ipa">[vsiʧki]</span>
<code>"BSICKI"</code>
<br><h1>Concept: "ashes"</h1>
```

Although it looks rather simple in the output, the `export()` method is very powerful and especially useful for pretty formatted output in publications. Often, normal readers of publications don't want to read txt-files in the supplementaries. Using `export()` to create high-level PDF-files with LaTeX or putting output in `html` on a website may come in very handy...

LexStat

Preliminaries

So far, we were only handling the data in some ways and did not come farther than tokenizing our IPA-strings. In order to calculate distances, trees, etc. we need to carry out cognate judgments. In order to do this, we need the `LexStat` class which is a descendant of the `Wordlist` class. Loading data into `LexStat` is no different than loading it into `Wordlist`. `LexStat` automatically handles IPA-tokenization, if no `tokens` are passed from the input file. But once we already calculated the tokens for our wordlist, we can likewise take the file `slavic2.csv` for our `LexStat` calculation.

```
In [28]: lex = LexStat('slavic2.qlc')
```

Cognate Judgments

Carrying out cognate judgments can be done in a simple way, using the `cluster()` method. We use this method out of the box for three out of four possible methods (`"ned"`,`"sca"`,`"turchin"`), but for the original `"lexstat"`-method, we need to calculate a scoring function first, using the `get_scorer()` method.

```
In [29]: lex.get_scorer(verbose=True,force=True)
```

Since the calculation of the scoring function is time-consuming, we might want to save the data. This is easy with help of the generic `pickle()` method provided by `Wordlist`. This function creates a cache directory (called `__lingpy__`), that stores our current wordlist in as `slavic2.bin`.

```
In [30]: lex.pickle()
```

Now we can start the calculation of cognate judgments using the `cluster()` method. We set the threshold to 0.6

which usually works quite well on simple datasets.

```
In [31]: lex.cluster(method='lexstat',threshold=0.5)
```

Output

Having calculated the cognate judgments, we should output the data to another wordlist file.

```
In [32]: lex.output('qlc',filename='slavic_lexstat',ignore=['scorer'])
```

If we have a look at the resulting format, we can see that `LexStat` added a lot of new values, including numerical references, sonority profiles, sound class strings, etc.

```
In [33]: print(''.join(open('slavic_lexstat.qlc').readlines()[0:40]))
```

```
# Wordlist

# JSON
<json>
{
    "params": {
        "cscorer": {
            "modestring": "global-2-0.50:local-1-0.50",
            "restricted_chars": "_T",
            "method": "shuffle",
            "threshold": 0.7,
            "preprocessing": "True:upgma:-2",
            "ratio": [
                2,
                1
            ],
            "runs": 1000,
            "factor": 0.3,
            "vscale": 0.5
        },
        "cluster": "lexstat_single_0.50"
    },
    "doculect": [
        "Bulgarian",
        "Czech",
        "Polish",
        "Russian"
    ]
}
</json>

# DATA
ID      CONCEPT DOCULECT        IPA     TOKENS  CLASSES SONARS
PROSTRINGS      LANGID  NUMBERS WEIGHTS DUPLICATES      SCAID   LEXSTATID
#
1       all     Russian fsʲe    f sʲ e  BSE     3 3 7   MBX     4
4.B.c 4.S.C 4.E.V       1.1 1.75 1.5    0       1       1
2       all     Polish  fʃɨstɕɨ  f ʃ ɨ s tɕ ɨ     BSISCI  3 3 7 3 2 7
MBXMBZ  3       3.B.c 3.S.C 3.I.V 3.S.c 3.C.C 3.I.V     1.1 1.75 1.5 1.1
1.75 0.8        0       1       1
3       all     Bulgarian       vsitʃki  v s i tʃ k i     BSICKI  3 3 7 2 1
7       MBXMBZ  1       1.B.c 1.S.C 1.I.V 1.C.c 1.K.C 1.I.V     1.1 1.75
1.5 1.1 1.75 0.8        0       1       1
4       all     Czech   fʃɪxnʲɪ f ʃ ɪ x nʲ ɪ    BSIGNI  3 3 7 3 4 7
MBXBCZ  2       2.B.c 2.S.C 2.I.V 2.G.C 2.N.C 2.I.V     1.1 1.75 1.5 1.75
1.5 0.8 0       1       1
#
5       ashes   Russian zɐla    z ɐ l a SELA    3 7 5 7 AXBZ    4
4.S.C 4.E.V 4.L.C 4.A.V 2.0 1.5 1.75 0.8        0       2       5
```

The most interesting columns we find in this output are the `SCAID` and `LEXSTATID`. These contain the cognate judgments for both the simple language-independent SCA-method and the language-specific LexStat-method. Since the LexStat-method builds on the SCA method, SCA cognate IDs are automatically calculated.

Etymological Dictionaries

Now that we have calculated cognate sets, we can use these to illustrate another important generic method for data-manipulation provided by `Wordlist`: the `get_etymdict()` method.

```
In [34]: etd = lex.get_etymdict(ref='lexstatid',entry='ipa')
         print(etd[1])
```

```
[['vsiʧki'], ['fʃɪxnʲɪ'], ['fʃɨstɕɨ'], ['fsʲe']]
```

This method calculates an *etymological dictionary* of our data. An etymological dictionary is a different representation of `Wordlist` data: While in traditional wordlists, rows represent concepts and columns languages, an etymological dictionary consists of cognate sets which are linked to the words. In our case, an etymological dictionary is a dictionary with cognate sets representing keys and nested lists representing languages in columns and concepts in rows, but basically, one can also think of an etymological dictionary as being a two-dimensional list in which the rows correspond to cognate sets, the cells contain words (or links to their IDs) and the columns represent languages.

```
In [35]: # define a format function for pretty output
         f = lambda x: '{0:10}'.format(x[0]) if x != 0 else '{0:10}'.format('-')

         # print the taxa in the first row
         print('\t'.join([f([l]) for l in lex.taxa]))

         # print the reflexes
         print('\t'.join([f(x) for x in etd[1]]))

         # print the concepts
         print('\t'.join([f(x) for x in lex.get_etymdict(ref='lexstatid',entry='concept'
```

```
Bulgarian       Czech           Polish          Russian
vsiʧki          fʃɪxnʲɪ         fʃɨstɕɨ         fsʲe
all             all             all             all
```

Note that the values in etymological dictionaries are nested lists, since it is not guaranteed that each cognate has only one reflex in one language. Furthermore, note that missing reflexes for a given cognate set receive a `0` as value in the etymological dictionary created by `Wordlist`, and that when not specifying the `entry` keyword, the etymological dictionary will use the IDs of the words to reflect their reflexes.

```
In [36]: etd = lex.get_etymdict(ref='scaid')
         print(etd[5])
         print([lex[x[0],'ipa'] for x in etd[5] if x != 0])
         print([lex[x[0],'language'] for x in etd[5] if x != 0])
         print([lex[x[0],'concept'] for x in etd[5] if x != 0])
```

```
[0, [17], [15], 0]
['brɪxo', 'bʒux']
['Czech', 'Polish']
['belly', 'belly']
```

Phylogenetic Trees

`Wordlist` comes along with another method for data manipulation: `calculate()` can be used to calculate all different kinds of things, such as distance matrices (counting the number of shared cognates), phylogenetic trees (using Neighbor-Joining or UPGMA), or major flat groupings of the taxa (using flat cluster algorithms). What these methods require is that cognate sets are defined in the data. Given that we just calculated them using `LexStat`, we can now use the `calculate()` method to create a phylogenetic tree of the data.

```
In [37]: lex.calculate('tree',tree_calc='neighbor',ref='lexstatid')
```

When calculating the tree, `Wordlist` (or `LexStat` in this case) first calculates distances using the amount of shared cognate sets among all language pairs and then calculates a phylogenetic tree using the specified method. The tree itself can be accessed as a newly created attribute of the class. Furthermore, trees are represented with as PhyloNode objects (borrowed from PyCogent library), and offer a few more nice methods for data-manipulation.

In [38]:
```
print(lex.tree)
print(lex.tree.asciiArt())
```

```
(((Bulgarian,Russian),Czech),Polish);
                              /-Bulgarian
                    /edge.0--|
          /edge.1--|          \-Russian
         |         |
-root----|          \-Czech
         |
          \-Polish
```

Alignments

Preliminaries

Cognate judgments are surely a good start, even if there is the possibility that the algorithm fails to detect or wrongly detects cognates. Nevertheless, when dealing only with the identifiers for the cognate sets (`"lexstatid"`, `"scaid"`), it is not easy to evaluate what the method actually does, since the `csv` format for output is simply not very pleasing to look at.

A simple solution that is alread provided by the `Wordlist` class is to use the `export()` method, but this time with an additional header for cognate IDs.

In [39]:
```
lex.export(
           "html",
           sections = dict(
                           h1 = ("concepts","<h1>Concept: {0}</h1>\n"),
                           h2 = ("lexstatid","<h2>Cognate Set: {0}</h2>\n"),
                           ),
           entries = [
                      ("language","{0}: "),
                      ("ipa","[{0}] <br>")
                      ],
           filename = "slavic_lexstat"
           )
```

```
In [40]: print(''.join(open("slavic_lexstat.html").readlines()[0:10]))
```

```
<h1>Concept: all</h1>
<h2>Cognate Set: 1</h2>
Bulgarian: [vsiʧki] <br>Czech: [fʃɪxnʲɪ] <br>Polish: [fʃɨsʦɨ] <br>Russian:
[fsʲe] <br><h1>Concept: ashes</h1>
<h2>Cognate Set: 5</h2>
Russian: [zɐla] <br><h2>Cognate Set: 2</h2>
Bulgarian: [pɛpɛl] <br>Czech: [popɛl] <br>Polish: [pɔpʲuw] <br>Russian:
[pʲepɪlʲ] <br><h1>Concept: bark</h1>
<h2>Cognate Set: 6</h2>
Bulgarian: [kora] <br>Czech: [kuːra] <br>Polish: [kɔra] <br>Russian: [kɐra]
<br><h1>Concept: belly</h1>
<h2>Cognate Set: 10</h2>
Russian: [ʒɨvot] <br><h2>Cognate Set: 9</h2>
```

Carrying out Alignment Analyses

An even better solution, however, is to *align* the reflexes of a given cognate set. This can be done with help of the `Alignments` class which is again a direct descendant of `Wordlist`. When loading data into `Alignments`, we need to specify which columns contains the cognate sets, using the `ref` keyword.

```
In [55]: alm = Alignments('slavic_lexstat.qlc', ref='lexstatid')
```

In contrast to simple `Wordlist` objects, `Alignments` has an additional dictionary which stores all cognate sets specified when loading it in groups that can then be easily aligned, using LingPy's specific functions for phonetic alignment. Using the defaults, this can be done very easily.

```
In [56]: alm.align(verbose=False,ref='lexstatid')
```

Alignments are stored in a specific dictionary attribute called `msa`. We can access them by specifying which cognate sets we used (`"lexstatid"` in our case), and the ID of a given cognate set. Using LingPy's convenient `SCA()` method, that creates various alignment objects on the fly (MSA, `Alignments`, PSA), depending on the input data, we can easily check out some of the results.

```
In [57]: msa = SCA(alm.msa['lexstatid'][1])
         print(msa)
```

```
v       s       i       ʧ       k       i
f       ʃ       ɪ       x       nʲ      ɪ
f       ʃ       ɨ       s       ʦ       ɨ
f       sʲ      e       -       -       -
```

Output

Controlling and inspecting alignment analyses on the terminal is not recommended for large files. Hence, it is helpful to output the data to text files or even other formats. Let us first output the data to `csv`-format.

```
In [58]: alm.output('qlc',filename='slavic_alignments')
```

When opening this file, we see that all alignments are added to the header of the file, using HTML-like tags.

In [59]:
```python
print(''.join(open('slavic_alignments.qlc').readlines()[0:12]))
print('...\n')
print(''.join(open('slavic_alignments.qlc').readlines()[40:60]))
print('...\n')
print(''.join(open('slavic_alignments.qlc').readlines()[1000:1010]))
```

```
# Wordlist

# JSON
<json>
{
    "params": {
        "cscorer": {
            "ratio": [
                2,
                1
            ],
            "vscale": 0.5,

...

#
<msa id="2" ref="lexstatid">
# Cognate Set: 2 ("ashes")
7       Bulgarian       p       ɛ       p       ɛ       l
8       Czech           p       o       p       ɛ       l
6       Polish          p       ɔ       pʲ      u       w
9       Russian         pʲ      e       p       ɪ       lʲ
</msa>
#
<msa id="6" ref="lexstatid">
# Cognate Set: 6 ("bark")
12      Bulgarian       k       o       r       a
13      Czech           k       uː      r       a
11      Polish          k       ɔ       r       a
10      Russian         k       ɐ       r       a
</msa>
#
<msa id="12" ref="lexstatid">
# Cognate Set: 12 ("big")
21      Czech   v       ɛ       l       k       iː

...

130     foot    Polish  nɔga    n ɔ g a NUKA    4 7 1 7 AXBZ    3
3.N.C 3.U.V 3.K.C 3.A.V 2.0 1.5 1.75 0.8        0       46      73
131     foot    Bulgarian       krak    k r a k KRAK    1 5 7 1 ACXN
1       1.K.C 1.R.C 1.A.V 1.K.c 2.0 1.5 1.5 0.8 0       45      72
132     foot    Czech   noɦa    n o ɦ a NUHA    4 7 3 7 AXBZ    2
2.N.C 2.U.V 2.H.C 2.A.V 2.0 1.5 1.75 0.8        0       46      73
#
133     full    Russian polnɨj  p o l n ɨ j     PULNIJ  1 7 5 4 7 6
AXMBYN  4       4.P.C 4.U.V 4.L.c 4.N.C 4.I.V 4.J.c     2.0 1.5 1.1 1.75
1.3 0.8 0       47      74
134     full    Polish  pɛwnɨ   p ɛ w n ɨ       PEWNI   1 7 6 4 7
AXMBZ   3       3.P.C 3.E.V 3.W.c 3.N.C 3.I.V   2.0 1.5 1.1 1.75 0.8
0       47      74
135     full    Bulgarian       pɤlɛn   p ɤ l ɛ n       PELEN   1 7 5 7
4       AXBYN   1       1.P.C 1.E.V 1.L.C 1.E.V 1.N.c   2.0 1.5 1.75 1.3
0.8     0       47      74
136     full    Czech   pl̩niː   p l̩ n iː         PLNI    1 7 4 7 AXBZ
2       2.P.C 2.L.V 2.N.C 2.I.V 2.0 1.5 1.75 0.8        0       47      75
#
137     give (V)        Russian datʲ    d a tʲ  TAT     1 7 1   AXN
4       4.T.C 4.A.V 4.T.c       2.0 1.5 0.8     0       48      76
```

This illustrates a very important additional aspect of the `Wordlist` `csv`-format: Apart from the raw word entries, it can contain additional information, such as, in this case, multiple sequence alignments. But this is not all, `Wordlist` reads and outputs additional "meta"-information using the following strategies:

- `@key: value`: Use this construct in separate lines *before* the `csv`-data to pass simple key-value pairs to the `Wordlist` that are accessible as simple attributes of the `Wordlist` object.
- `<specific_data> ... </specific_data>`: Use this construct on multiple lines for predefined additional data structures, such as multiple alignments (`<msa>`), distance matrices (`<dst>`), or taxa (`<taxa>`). More constructs will probably be added in the future.
- `<json> JSON data </json>`: Use JSON datastructures on multiple lines for all kinds of data structures that are not yet implemented as specific constructs.

Let's have a look at these specific ways of meta-data modeling by calculating some additional data for our `Alignments` object and writing it to file.

```
In [60]: alm.calculate('dst',ref='lexstatid')
         alm.calculate('tree',ref='lexstatid')
         alm.calculate('groups',threshold=0.2,ref='lexstatid')
         alm.output('qlc',filename='slavic_metadata')
         lines = open('slavic_metadata.qlc').readlines()
         print('\n...\n')
         print(''.join([x for x in lines if '@tree' in x]))
         print('...\n')
         print(''.join(lines[lines.index('<json>\n'):lines.index('</json>\n')+1]))
         print('...\n')
         print(''.join(lines[lines.index('<dst>\n'):lines.index('</dst>\n')+1]))
```

```
...

@tree:(Bulgarian,(Russian,(Czech,Polish)));

...

<json>
{
    "params": {
        "cscorer": {
            "ratio": [
                2,
                1
            ],
            "vscale": 0.5,
            "runs": 1000,
            "modestring": "global-2-0.50:local-1-0.50",
            "restricted_chars": "_T",
            "factor": 0.3,
            "method": "shuffle",
            "threshold": 0.7,
            "preprocessing": "True:upgma:-2"
        },
        "cluster": "lexstat_single_0.50"
    },
    "groups": {
        "Russian": "G_4",
        "Bulgarian": "G_1",
        "Polish": "G_3",
        "Czech": "G_2"
    },
    "doculect": [
        "Bulgarian",
        "Czech",
        "Polish",
        "Russian"
    ]
}
</json>

...

<dst>
 4
Bulgarian 0.000000 0.348624 0.318182 0.336364
Czech     0.348624 0.000000 0.201835 0.284404
Polish    0.318182 0.201835 0.000000 0.227273
Russian   0.336364 0.284404 0.227273 0.000000
</dst>
```

As can be seen from this example, the extended `csv` format is very flexible, and gives the user a lot of freedom to handle all kinds of different datatypes.

However, for alignment analyses and cognate judgments, one usually wants some other output format which comes closer to publication plots than to txt-based formats. In order to convert our data into a colorful HTML file that can be conveniently used for publication or manual evaluation, we can use the plot() function defined for the Alignment class:

```
In [63]: alm.plot(ref='lexstatid',filename='slavic_alignments',show=True)
```

Getting Preliminary Reconstructions

With help of alignments, some more interesting things can be done. For example, we can calculate preliminary quick-and-dirty "reconstructions" by calculating the consensus strings of all alignments. A consensus string represents an alignment by its most frequently occuring characters. That it is not far away from a real reconstruction is reflected in the fact that on a Polynesian testset the consensus strings are only 2.15 edit operations away from expert reconstructions. Comparing this score to the 1.9 edit operations that the method by Bouchard-Côté et al. (2013) yields, this is not too bad for a very, very simple approach. Consensus strings can be calculated with help of the `get_consensus()` method. The consensus string is stored in an additional column which is named "consensus" as a default.

```
In [64]: alm.get_consensus(ref='lexstatid')
         alm.output('qlc',filename='slavic_consensus')
```

Let's check the consensus string for the first ten basic concepts in the Russian list.

```
In [53]: cons = alm.get_dict(col='Russian')
         for key in sorted(cons)[0:11]:
             print('\t'.join(['{0:10}'.format(k) for k in [key,alm[cons[key][0],'ipa'],a
```

```
all             fsʲe            fʃeʧnʲ
ashes           zɐla            zɐla
bark            kɐra            kuːra
belly           ʒɨvot           ʒɨvot
big             bɐlʲʃoj         bɐlʲʃoj
bird            ptʲiʦə          ptiʦ
bite (V)        kusatʲ          kowsatʲ
black           ʧornɨj          ʧornɨ
blood           krofʲ           krɛf
bone            kosʲtʲ          kost
breast          grutʲ           grutʲ
```

PhyBo

As a last short example and outlook, we have a quick look at the borrowing detection method which is available in the PhyBo class of LingPy. This method is pretty complex, and it offers very many methods and parameters which have not yet been fully documented and explored, we make this example very, very quick. In order to load the class, we need an input file (just like the output file we just created from our alignments, and we need to explicitly import the PhyBo class, since it is not included in the regular imports of LingPy.

In [65]:
```
from lingpy.compare.phylogeny import PhyBo
```

Now we load the file into PhyBo. Note that we don't need to specify the file extension (`csv`), since PhyBo expects a "dataset" as input. Additional data can be passed to PhyBo in additional files with different extensions that have the same basic file name.

In [67]:
```
phybo = PhyBo('slavic_consensus',ref='lexstatid')
```

The easiest way to use PhyBo is to use the `analyze()` method without parameters. This will evoke some default settings which guarantee a more or less satisfying coverage of analyses to be run.

In [68]:
```
phybo.analyze()
```

Calculating "Minimal Lateral Networks" (MLN) following the method by Nelson-Sathi et al. (2011), is done by first computing the MLN using the `get_MLN()` method. Here, we need to pass an additional argument which is usually the best model of the analysis. After using the `analyze()` method, this model is automatically stored in the `best_model` attribute of a PhyBo object.

In [69]:
```
phybo.get_MLN(phybo.best_model)
```

Plotting the data to a PDF file, can then be done with help of the `plot_MLN()` method.

In [70]:
```
phybo.plot_MLN(phybo.best_model)
```

The result of this analysis is the file "slavic_consensus.pdf" (and a lot more data). For four language, the network is not very impressive, but it should give a first impression on what can be done with LingPy.