LingPy Documentation

Release 2.6.4

Johann-Mattis List, Simon Greenhill, Tiago Tresoldi, and Robert Forkel

2018-11-26

CONTENTS

CHAPTER

ONE

SEQUENCE MODELLING

1.1 Sound Classes (`sound_classes`)

This module provides functions and classes to deal with sound-class sequences. Sound classes go back to an approach `Dolgopolsky1964`. The basic idea behind sound classes is to reduce the IPA space of possible phonetic segments in order to guarantee both the comparability of sounds between languages, but also to give some assessment regarding the probability that sounds belonging to the same class occur in correspondence relations in genetically related languages. More recently, sound classes have been applied in a couple of approaches, including phonetic alignment (see `List2012a`), and automatic cognate detection (see `Turchin2012`, `List2012b`).

1.1.1 Functions

ipa2tokens(istring, **keywords)	Tokenize IPA-encoded strings.
tokens2class(tokens, model[, stress,])	Convert tokenized IPA strings into their respective class strings.
prosodic_string(string[, _output])	Create a prosodic string of the sonority profile of a sequence.
prosodic_weights(prostring[, _transform])	Calculate prosodic weights for each position of a sequence.
class2tokens(tokens, classes[, gap_char, local])	Turn aligned sound-class sequences into an aligned sequences of IPA tokens.
pid(almA, almB[, mode])	Calculate the Percentage Identity (PID) score for aligned sequence pairs.
get_all_ngrams(sequence[, sort])	Function returns all possible n-grams of a given sequence.
sampa2uni(seq)	Convert sequence in IPA-sampa-format to IPA-unicode.

1.1.2 Classes

Model(model[, path])	Class for the handling of sound-class models.

1.2 Generate Random Sequences (`generate`)

1.2.1 Classes

MCBasic(seqs)	Basic class for creating Markov chains from sequence training data.
MCPhon(words[, tokens, prostrings, classes,])	Class for the creation of phonetic sequences (pseudo words).

1.3 Generate Orthography Profiles (`profile`)

1.3.1 Functions

simple_profile(wordlist[, ref,])	Create an initial Orthography Profile using Lingpys clean_string procedure.
context_profile(wordlist[, ref, col,])	Create an advanced Orthography Profile with context and doculect information.

1.4 Methods for generating and manipulating ngrams (`ngrams`)

1.4.1 Classes

NgramModel([pre_order, post_order,])	Class for operation upon sequences using ngrams models.

1.4.2 Functions

get_n_ngrams(sequence, order[, pad_symbol])	Build an iterator for collecting all ngrams of a given order.
get_all_ngrams_by_order(sequence[, orders,])	Build an iterator for collecting all ngrams of a given set of orders.
get_skipngrams(sequence, order, max_gaps[,])	Build an iterator for collecting all skip ngrams of a given length.
get_posngrams(sequence[, pre_order,])	Build an iterator for collecting all positional ngrams of a sequence.
get_all_posngrams(sequence, pre_orders,)	Build an iterator for collecting all positional ngrams of a sequence.
get_all_ngrams(sequence[, sort])	Function returns all possible n-grams of a given sequence.

1.5 Sound Class Models (`Model`)

class `lingpy.data.model.`**`Model`**(*model*, *path=None*)

Class for the handling of sound-class models.

Parameters **model** : { sca, dolgo, asjp, art, _color }

A string indicating the name of the model which shall be loaded. Select between:

- sca - the SCA sound-class model (see `List2012a`),
- dolgo - the DOLGO sound-class model (see: :evobib:‘Dolgopolsky1986),

- asjp - the ASJP sound-class model (see Brown2008 and Brown2011),
- art - the sound-class model which is used for the calculation of sonority profiles and prosodic strings (see List2012), and
- _color - the sound-class model which is used for the coloring of sound-tokens when creating html-output.

See also:

lingpy.data.derive.compile_model, *lingpy.data.derive.compile_dvt*

Notes

Models are loaded from binary files which can be found in the `data/models/` folder of the LingPy package. A model has two essential attributes:

- `converter` – a dictionary with IPA-tokens as keys and sound-class characters as values, and
- `scorer` – a scoring dictionary with tuples of sound-class characters as keys and scores (integers or floats) as values.

Examples

When loading LingPy, the models `sca`, `asjp`, `dolgo`, and `art` are automatically loaded, and they are accessible via the *rc()* function for global settings:

```
>>> from lingpy import *
>>> rc('asjp')
<sca-model "asjp">
```

Define variables for the standard models for convenience:

```
>>> asjp = rc('asjp')
>>> sca = rc('sca')
>>> dolgo = rc('dolgo')
>>> art = rc('art')
```

Check how the letter `a` is converted in the various models:

```
>>> for m in [asjp,sca,dolgo,art]:
...     print('{0} > {1} ({2})'.format('a',m.converter['a'],m.name))
...
a > a (asjp)
a > A (sca)
a > V (dolgo)
a > 7 (art)
```

Retrieve basic information of a given model:

```
>>> print(sca)
Model:     sca
Info:      Extended sound class model based on Dolgopolsky (1986)
Source:    List (2012)
Compiler: Johann-Mattis List
Date:      2012-03
```

Attributes

converter	dict	A dictionary with IPA tokens as keys and sound-class characters as values.
scorer	dict	A scoring dictionary with tuples of sound-class characters as keys and similarity scores as values.
info	dict	A dictionary storing the key-value pairs defined in the `INFO`.
name	str	The name of the model which is identical with the name of the folder from wich the model is loaded.

1.6 Predefined Datasets (`data`)

LingPy comes along with many different kinds of predefined data. When loading the library, the following dictionary is automatically loaded and employed by all LingPy modules:

`rcParams : dict`

As an alternative to all global variables, this dictionary contains all these variables, and additional ones. This dictionary is used for internal coding purposes and stores parameters that are globally set (if not defined otherwise by the user), such as

- specific debugging messages (warnings, messages, errors)
- default values, such as gop (gap opening penalty), scale (scaling factor
- by which extended gaps are penalized), or figsize (the default size of
- figures if data is plotted using matplotlib).

These default values can be changed with help of the `rc` function that takes any keyword and any variable as input and adds or modifies the specific key of the rcParams dictionary, but also provides more complex functions that change whole sets of variables, such as the following statement:

```
>>> rc(schema="asjp")
```

which switches the variables asjp, dolgo, etc. to the ASCII-based transcription system of the ASJP project.

If you want to change the content of c{rcParams} directly, you need to import the dictionary explicitly:

```
>>> from lingpy.settings import rcParams
```

However, changing the values in the dictionary randomly can produce unexpected behavior and we recommend to use the regular `rc` function for this purpose.

`lingpy.settings.`**`rc`**`(`*rval=None*, ***keywords*`)`

Function changes parameters globally set for LingPy sessions.

Parameters rval : string (default=None)

Use this keyword to specify a return-value for the rc-function.

schema : {ipa, asjp}

Change the basic schema for sequence comparison. When switching to asjp, this means that sequences will be treated as sequences in ASJP code, otherwise, they will be treated as sequences written in basic IPA.

Notes

This function is the standard way to communicate with the *rcParams* dictionary which is not imported as a default. If you want to see which parameters there are, you can load the rcParams dictonary directly:

```
>>> from lingpy.settings import rcParams
```

However, be careful when changing the values. They might produce some unexpected behavior.

Examples

Import LingPy:

```
>>> from lingpy import *
```

Switch from IPA transcriptions to ASJP transcriptions:

```
>>> rc(schema="asjp")
```

You can check which basic orthography is currently loaded:

```
>>> rc(basic_orthography)
'asjp'
>>> rc(schema='ipa')
>>> rc(basic_orthography)
'fuzzy'
```

1.6.1 Functions

rc([rval])	Function changes parameters globally set for LingPy sessions.

1.7 Creating Sound-Class Models (`derive`)

1.7.1 Functions

compile_model(model[, path])	Function compiles customized sound-class models.
compile_dvt([path])	Function compiles diacritics, vowels, and tones.

CHAPTER

TWO

DATASET HANDLING

2.1 Word Lists (`wordlist`)

Word lists represent the core of LingPys data model, and a proper understanding of how we deal with word lists is important for automatic cognate detection, alignments, and borrowing detection. The basic class that handles word lists, is the *`Wordlist`* class, which is also the base class of the *`LexStat`* class for automatic cognate detection and the *`Alignments`* class for multiple alignment of cognate words.

2.1.1 Functions

`get_wordlist`(path[, delimiter, quotechar,])	Load a wordlist from a normal CSV file.

2.1.2 Classes

`Wordlist`(filename[, row, col, conf])	Basic class for the handling of multilingual word lists.

2.2 Cognate Detection (`sanity`)

2.2.1 Functions

`mutual_coverage`(wordlist[, concepts])	Compute mutual coverage for all language pairs in your data.
`mutual_coverage_check`(wordlist, threshold[,])	Check whether a given mutual coverage is fulfilled by the dataset.
`mutual_coverage_subset`(wordlist, threshold)	Compute maximal mutual coverage for all language in a wordlist.
`synonymy`(wordlist[, concepts, languages])	Check the number of synonyms per language and concept.

CHAPTER

THREE

DATA EXPORT

3.1 Converting Data to Strings (`strings`)

The strings module provides some general and some specific functions which allow to convert data into strings which can then be imported by other software tools. You can import it by typing:

```
>>> from lingpy.convert.strings import *
```

Or by typing:

```
>>> from lingpy.convert import strings
```

Most of the functions are used internally, being triggered when writing, for example, data from a ~lingpy.basic.wordlist.Wordlist object to file. They can, however, also be used directly, and especially the ~lingpy.convert.strings.write_nexus function may prove useful to get a more flexible nexus-output of wordlist data.

3.1.1 Functions

`scorer2str`(scorer)	Convert a scoring function to a string.
`msa2str`(msa[, wordlist, comment, _arange, merge])	Function converts an MSA object into a string.
`matrix2dst`(matrix[, taxa, stamp, filename,])	Convert matrix to dst-format.
`pap2nex`(taxa, paps[, missing, filename,])	Function converts a list of paps into nexus file format.
`pap2csv`(taxa, paps[, filename])	Write paps created by the Wordlist class to a csv-file.
`multistate2nex`(taxa, matrix[, filename, missing])	Convert the data in a given wordlist to NEXUS-format for multistate analyses in PAUP.
`write_nexus`(wordlist[, mode, filename, ref,])	Write a nexus file for phylogenetic analyses.

3.2 Converting Data to CLDF (`cldf`)

3.2.1 Functions

`to_cldf`(wordlist[, path, source_path, ref,])	Convert a wordlist in LingPy to CLDF.
`from_cldf`(path[, to, concept, concepticon,])	Load data from CLDF into a LingPy Wordlist object or similar.

CHAPTER

FOUR

SEQUENCE COMPARISON

4.1 Helper Functions for SCA Alignment (`calign`, and `misc`)

The helper functions and classes below play an important role in all SCA alignment algorithms in LingPy (`List2012b`). They are implemented both in pure Python and in Cython (only supported for Python 3), in order to allow for faster implementations of the core alignment functions. Instead of using these functions directly, we recommend to use the more general functions which you can find in the *`pairwise`* and the *`multiple`* module of LingPy, and which are based on the helper functions we list below.

4.1.1 Functions

`globalign`	Carry out global alignment of two sequences.
`secondary_globalign`	Carry out global alignment of two sequences with secondary sequence structures.
`localign`	Carry out semi-global alignment of two sequences.
`secondary_localign`	Carry out lobal alignment of two sequences with sensitivity to secondary sequence structures.
`semi_globalign`	Carry out semi-global alignment of two sequences.
`secondary_semi_globalign`	Carry out semi-global alignment of two sequences with sensitivity to secondary sequence structures.
`dialign`	Carry out dialign alignment of two sequences.
`secondary_dialign`	Carry out dialign alignment of two sequences with sensitivity for secondary sequence structures.
`align_pair`	Align a pair of sequences.
`align_pairwise`	Align a list of sequences pairwise.
`align_pairs`	Align multiple sequence pairs.
`align_profile`	Align two profiles using the basic modes.
`score_profile`	Basic function for the scoring of profiles.
`swap_score_profile`	Basic function for the scoring of profiles which contain swapped sequences.
`corrdist`	Create a correspondence distribution for a given language pair.

4.1.2 Classes

ScoreDict	Class allows quick access to scoring functions using dictionary syntax.

4.2 Miscellaneous Helper Functions (`malign`)

The helper functions below are miscellaneous deep implementations of alignment and string similarity algorithms. They are implemented both in pure Python and in Cython (only supported for Python 3), in order to allow for faster implementations of the core alignment functions. Instead of using these functions directly, we recommend to use the more general functions which you can find in the *pairwise* and the *multiple* module of LingPy, and which are based on the helper functions we list below.

4.2.1 Functions

edit_dist	Return the edit-distance between two strings.
nw_align	Align two sequences using the Needleman-Wunsch algorithm.
restricted_edit_dist	Return the restricted edit-distance between two strings.
structalign	Carry out a structural alignment analysis using Dijkstras algorithm.
sw_align	Align two sequences using the Smith-Waterman algorithm.
we_align	Align two sequences using the Waterman-Eggert algorithm.

4.3 Helper Functions for Traditional Alignment (`talign`)

The helper functions and classes below play an important role in traditional alignment algorithms in LingPy which do not make use of sound classes. They are implemented both in pure Python and in Cython (only supported for Python 3), in order to allow for faster implementations of the core alignment functions. Instead of using these functions directly, we recommend to use the more general functions which you can find in the *pairwise* and the *multiple* module of LingPy, and which are based on the helper functions we list below.

4.3.1 Functions

globalign	Carry out global alignment of two sequences.
localign	Carry out semi-global alignment of two sequences.
semi_globalign	Carry out semi-global alignment of two sequences.
dialign	Carry out dialign alignment of two sequences.
align_pair	Align a pair of sequences.
align_pairwise	Align all sequences pairwise.
align_pairs	Align multiple sequence pairs.
align_profile	Align two profiles using the basic modes.
score_profile	Basic function for the scoring of profiles.
swap_score_profile	Basic function for the scoring of profiles in swapped sequences.

4.4 Pairwise Alignment (`pairwise`)

4.4.1 Functions

nw_align(seqA, seqB[, scorer, gap])	Carry out the traditional Needleman-Wunsch algorithm.
sw_align(seqA, seqB[, scorer, gap])	Carry out the traditional Smith-Waterman algorithm.
we_align(seqA, seqB[, scorer, gap])	Carry out the traditional Waterman-Eggert algorithm.
edit_dist(seqA, seqB[, normalized, restriction])	Return the edit distance between two strings.
SCA(infile, **keywords)	Method returns alignment objects depending on input file or input data.

4.4.2 Classes

Pairwise(seqs[, seqB])	Basic class for the handling of pairwise sequence alignments (PSA).
PSA(infile, **keywords)	Basic class for dealing with the pairwise alignment of sequences.

4.5 Multiple Alignment (`multiple`)

4.5.1 Functions

mult_align(seqs[, gop, scale, tree_calc,])	A short-cut method for multiple alignment analyses.
SCA(infile, **keywords)	Method returns alignment objects depending on input file or input data.

4.5.2 Classes

Multiple(seqs, **keywords)	Basic class for multiple sequence alignment analyses.
MSA(infile, **keywords)	Basic class for carrying out multiple sequence alignment analyses.
Alignments(infile[, row, col, conf,])	Class handles Wordlists for the purpose of alignment analyses.

CHAPTER

FIVE

LANGUAGE COMPARISON

5.1 Cluster Algorithms (`clustering` and `extra`)

5.1.1 Functions

flat_cluster(method, threshold, matrix[,])	Carry out a flat cluster analysis based on linkage algorithms.
flat_upgma(threshold, matrix[, taxa, revert])	Carry out a flat cluster analysis based on the UPGMA algorithm (Sokal1958).
fuzzy(threshold, matrix, taxa[, method, revert])	Create fuzzy cluster of a given distance matrix.
link_clustering(threshold, matrix, taxa[,])	Carry out a link clustering analysis using the method by Ahn2010.
mcl(threshold, matrix, taxa[, max_steps,])	Carry out a clustering using the MCL algorithm (Dongen2000).
neighbor(matrix, taxa[, distances])	Function clusters data according to the Neighbor-Joining algorithm (Saitou1987).
upgma(matrix, taxa[, distances])	Carry out a cluster analysis based on the UPGMA algorithm (Sokal1958).
infomap_clustering(threshold, matrix[,])	Compute the Infomap clustering analysis of the data.
affinity_propagation(threshold, matrix, taxa)	Compute affinity propagation from the matrix.
valid_cluster(sequence)	Only allow to have sequences which have consecutive ordering of elements.
generate_all_clusters(numbers)	Generate all possible clusters for a number of elements.
generate_random_cluster(numbers[, bias])	Generate a random cluster for a number of elements.
order_cluster(clr)	Order a cluster into the form of a valid cluster.
mutate_cluster(clr[, chance])	Mutate a cluster.

5.2 Cognate Detection (`LexStat`)

class lingpy.compare.lexstat.**LexStat**(*filename*, ***keywords*)

Basic class for automatic cognate detection.

Parameters **filename** : str

The name of the file that shall be loaded.

model : *Model*

The sound-class model that shall be used for the analysis. Defaults to the SCA sound-class model.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged into single tokens or kept apart as separate tokens.

transform : dict

A dictionary that indicates how prosodic strings should be simplified (or generally transformed), using a simple key-value structure with the key referring to the original prosodic context and the value to the new value. Currently, prosodic strings (see *prosodic_string()*) offer 11 different prosodic contexts. Since not all these are helpful in preliminary analyses for cognate detection, it is useful to merge some of these contexts into one. The default settings distinguish only 5 instead of 11 available contexts, namely:

- `C` for all consonants in prosodically ascending position,
- `c` for all consonants in prosodically descending position,
- `V` for all vowels,
- `T` for all tones, and
- `_` for word-breaks.

Make sure to check also the vowel keyword when initialising a LexStat object, since the symbols you use for vowels and tones should be identical with the ones you define in your transform dictionary.

vowels : str (default=**VT_**)

For scoring function creation using the *get_scorer* function, you have the possibility to use reduced scores for the matching of tones and vowels by modifying the vscale parameter, which is set to 0.5 as a default. In order to make sure that vowels and tones are properly detected, make sure your prosodic string representation of vowels matches the one in this keyword. Thus, if you change the prosodic strings using the transform keyword, you also need to change the vowel string, to make sure that vscale works as wanted in the *get_scorer* function.

check : bool (default=False)

If set to **True**, the input file will first be checked for errors before the calculation is carried out. Errors will be written to the file **errors**, defaulting to `errors.log`. See also `apply_checks` apply_checks : bool (default=False) If set to **True**, any errors identified by *check* will be handled silently.

no_bscorer: bool (default=False) :

If set to **True**, this will suppress the creation of a language-specific scoring function (which may become quite large and is additional ballast if the method lexstat is not used after all. If you use the lexstat method, however, this needs to be set to **False**.

errors : str

The name of the error log.

segments : str (default=tokens)

The name of the column in your data which contains the segmented transcriptions, or in which the segmented transcriptions should be placed.

transcription : str (default=ipa)

The name of the column in your data which contains the unsegmented transcriptions.

classes : str (default=classes)

The name of the column in the data which contains the sound class representation of the transcriptions, or in which this information shall be placed after automatic conversion.

numbers : str (default=numbers)

The language-specific triples consisting of language id (numeric), sound class string (one character only), and prosodic string (one character only). Usually, numbers are automatically created from the columns classes, prostrings, and langid, but you can also provide them in your data.

langid : str (default=langid)

Name of the column that contains a numerical language identifier, needed to produce the language-specific character triples (numbers). Unless specific explicitly, this is automatically created.

prostrings : str (default=prostrings)

Name of the column containing prosodic strings (see `List2014d` for more details) of the segmented transcriptions, containing one character per prosodic string. Prostrings add a contextual component to phonetic sequences. They are automatically created, but can likewise be submitted from the initial data.

weights : str (default=weights)

The name of the column which stores the individual gap-weights for each sequence. Gap weights are positive floats for each segment in a string, which modify the gap opening penalty during alignment.

tokenize : function (default=ipa2tokens)

The function which should be used to tokenize the entries in the column storing the transcriptions in case no segmentation is provided by the user.

get_prostring : function (default=prosodic_string)

The function which should be used to create prosodic strings from the segmented transcription data. If you want to completely ignore prosodic strings in LexStat calculations, you could just pass the following function:

```
>>> lex = LexStat('inputfile.tsv', get_prostring=lambda x: ["x"␣
↪for
    y in x])
```

cldf : bool (default=True)

If set to True, as by default, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved when internally converting tokens to classes (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a ``source/target`` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Notes

Instantiating this class does not require a lot of parameters. However, the user may modify its behaviour by providing additional attributes in the input file.

Attributes

pairs	dict	A dictionary with tuples of language names as key and indices as value, pointing to unique combinations of words with the same meaning in all language pairs.
model	*Model*	The sound class model instance which serves to convert the phonetic data into sound classes.
chars	list	A list of all unique language-specific character types in the instantiated LexStat object. The characters in this list consist of • the language identifier (numeric, referenced as langid as a default, but customizable via the keyword langid) • the sound class symbol for the respective IPA transcription value • the prosodic class value All values are represented in the above order as one string, separated by a dot. Gaps are also included in this collection. They are traditionally represented as X for the sound class and - for the prosodic string.
rchars	list	A list containing all unique character types across languages. In contrast to the chars-attribute, the rchars (raw chars) do not contain the language identifier, thus they only consist of two values, separated by a dot, namely, the sound class symbol, and the prosodic class value.
scorer	dict	A collection of *ScoreDict* objects, which are used to score the strings. LexStat distinguishes two different scoring functions: • rscorer: A raw scorer that is not language-specific and consists only of sound class values and prosodic string values. This scorer is traditionally used to carry out the first alignment in order to calculate the language-specific scorer. It is directly accessible as an attribute of the LexStat class (`rscorer`). The characters which constitute the values of this scorer are accessible via the rchars attribue of each lexstat class. • bscorer: The language-

Methods

align_pairs(idxA, idxB[, concept])	Align all or some words of a given pair of languages.
cluster([method, cluster_method, threshold,])	Function for flat clustering of words into cognate sets.
get_distances([method, mode, gop, scale,])	Method calculates different distance estimates for language pairs.
get_random_distances([method, runs, mode,])	Method calculates randoms scores for unrelated words in a dataset.
get_scorer(**keywords)	Create a scoring function based on sound correspondences.
output(fileformat, **keywords)	Write data to file.

Inherited WordList Methods

pickle([filename])	Store the QLCParser instance in a pickle file.
get_entries(entry)	Return all entries matching the given entry-type as a two-dimensional list.
add_entries(entry, source, function[, override])	Add new entry-types to the word list by modifying given ones.
calculate(data[, taxa, concepts, ref])	Function calculates specific data.
export(fileformat[, sections, entries,])	Export the wordlist to specific fileformats.
export(fileformat[, sections, entries,])	Export the wordlist to specific fileformats.
get_dict([col, row, entry])	Function returns dictionaries of the cells matched by the indices.
get_dict([col, row, entry])	Function returns dictionaries of the cells matched by the indices.
get_etymdict([ref, entry, modify_ref])	Return an etymological dictionary representation of the word list.
get_etymdict([ref, entry, modify_ref])	Return an etymological dictionary representation of the word list.
get_list([row, col, entry, flat])	Function returns lists of rows and columns specified by their name.
get_list([row, col, entry, flat])	Function returns lists of rows and columns specified by their name.
get_paps([ref, entry, missing, modify_ref])	Function returns a list of present-absent-patterns of a given word list.
get_paps([ref, entry, missing, modify_ref])	Function returns a list of present-absent-patterns of a given word list.
output(fileformat, **keywords)	Write wordlist to file.
renumber(source[, target, override])	Renumber a given set of string identifiers by replacing the ids by integers.

5.3 Partial Cognate Detection (`Partial`)

class `lingpy.compare.partial.`**`Partial`**(*infile*, ***keywords*)

Extended class for automatic detection of partial cognates.

Parameters **filename** : str

The name of the file that shall be loaded.

model : *Model*

The sound-class model that shall be used for the analysis. Defaults to the SCA sound-class model.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged into single tokens or kept apart as separate tokens.

transform : dict

A dictionary that indicates how prosodic strings should be simplified (or generally transformed), using a simple key-value structure with the key referring to the original prosodic context and the value to the new value. Currently, prosodic strings (see *prosodic_string()*) offer 11 different prosodic contexts. Since not all these are helpful in preliminary analyses for cognate detection, it is useful to merge some of these contexts into one. The default settings distinguish only 5 instead of 11 available contexts, namely:

- `C` for all consonants in prosodically ascending position,
- `c` for all consonants in prosodically descending position,
- `V` for all vowels,
- `T` for all tones, and
- _ for word-breaks.

Make sure to check also the vowel keyword when initialising a LexStat object, since the symbols you use for vowels and tones should be identical with the ones you define in your transform dictionary.

vowels : str (default=**VT_**)

For scoring function creation using the *get_scorer* function, you have the possibility to use reduced scores for the matching of tones and vowels by modifying the vscale parameter, which is set to 0.5 as a default. In order to make sure that vowels and tones are properly detected, make sure your prosodic string representation of vowels matches the one in this keyword. Thus, if you change the prosodic strings using the transform keyword, you also need to change the vowel string, to make sure that vscale works as wanted in the *get_scorer* function.

check : bool (default=False)

If set to **True**, the input file will first be checked for errors before the calculation is carried out. Errors will be written to the file **errors**, defaulting to `errors.log`. See also `apply_checks`

apply_checks : bool (default=False)

If set to **True**, any errors identified by *check* will be handled silently.

no_bscorer: bool (default=False) :

If set to **True**, this will suppress the creation of a language-specific scoring function (which may become quite large and is additional ballast if the method lexstat is not used after all. If you use the lexstat method, however, this needs to be set to **False**.

errors : str

The name of the error log.

Notes

This method automatically infers partial cognate sets from data which was previously morphologically segmented.

Attributes

pairs	dict	A dictionary with tuples of language names as key and indices as value, pointing to unique combinations of words with the same meaning in all language pairs.
model	*Model*	The sound class model instance which serves to convert the phonetic data into sound classes.
chars	list	A list of all unique language-specific character types in the instantiated LexStat object. The characters in this list consist of • the language identifier (numeric, referenced as langid as a default, but customizable via the keyword langid) • the sound class symbol for the respective IPA transcription value • the prosodic class value All values are represented in the above order as one string, separated by a dot. Gaps are also included in this collection. They are traditionally represented as X for the sound class and - for the prosodic string.
rchars	list	A list containing all unique character types across languages. In contrast to the chars-attribute, the rchars (raw chars) do not contain the language identifier, thus they only consist of two values, separated by a dot, namely, the sound class symbol, and the prosodic class value.
scorer	dict	A collection of *ScoreDict* objects, which are used to score the strings. LexStat distinguishes two different scoring functions: • rscorer: A raw scorer that is not language-specific and consists only of sound class values and prosodic string values. This scorer is traditionally used to carry out the first alignment in order to calculate the language-specific scorer. It is directly accessible as an attribute of the LexStat class (`rscorer`). The characters which constitute the values in this scorer are accessible via the rchars attribue of each lexstat class. • bscorer: The language-

Methods

`partial_cluster`([method, threshold, scale,])	Cluster the words into partial cognate sets.
`add_cognate_ids`(source, target[, idtype,])	Compute normal cognate identifiers from partial cognate sets.

Inherited LexStat Methods

`align_pairs`(idxA, idxB[, concept])	Align all or some words of a given pair of languages.
`cluster`([method, cluster_method, threshold,])	Function for flat clustering of words into cognate sets.
`get_distances`([method, mode, gop, scale,])	Method calculates different distance estimates for language pairs.
`get_random_distances`([method, runs, mode,])	Method calculates randoms scores for unrelated words in a dataset.
`get_scorer`(**keywords)	Create a scoring function based on sound correspondences.
`output`(fileformat, **keywords)	Write data to file.

Inherited WordList Methods

`pickle`([filename])	Store the QLCParser instance in a pickle file.
`get_entries`(entry)	Return all entries matching the given entry-type as a two-dimensional list.
`add_entries`(entry, source, function[, override])	Add new entry-types to the word list by modifying given ones.
`calculate`(data[, taxa, concepts, ref])	Function calculates specific data.
`export`(fileformat[, sections, entries,])	Export the wordlist to specific fileformats.
`export`(fileformat[, sections, entries,])	Export the wordlist to specific fileformats.
`get_dict`([col, row, entry])	Function returns dictionaries of the cells matched by the indices.
`get_dict`([col, row, entry])	Function returns dictionaries of the cells matched by the indices.
`get_etymdict`([ref, entry, modify_ref])	Return an etymological dictionary representation of the word list.
`get_etymdict`([ref, entry, modify_ref])	Return an etymological dictionary representation of the word list.
`get_list`([row, col, entry, flat])	Function returns lists of rows and columns specified by their name.
`get_list`([row, col, entry, flat])	Function returns lists of rows and columns specified by their name.
`get_paps`([ref, entry, missing, modify_ref])	Function returns a list of present-absent-patterns of a given word list.
`get_paps`([ref, entry, missing, modify_ref])	Function returns a list of present-absent-patterns of a given word list.
`output`(fileformat, **keywords)	Write wordlist to file.
`renumber`(source[, target, override])	Renumber a given set of string identifiers by replacing the ids by integers.

5.4 Borrowing Detection (`phylogeny`)

class `lingpy.compare.phylogeny.`**PhyBo**(*dataset*, *tree=None*, *paps='pap'*, *ref='cogid'*, *tree_calc='neighbor'*, *output_dir=None*, ***keywords*)

Basic class for calculations using the TreBor method.

Parameters **dataset** : string

Name of the dataset that shall be analyzed.

tree : {None, string}

Name of the tree file.

paps : string (default=pap)

Name of the column that stores the specific cognate IDs consisting of an arbitrary integer key and a key for the concept.

ref : string (default=cogid)

Name of the column that stores the general cognate ids (the reference of the analysis).

tree_calc : {neighbor,upgma} (default=neighbor)

Select the algorithm to be used for the tree calculation if no tree is passed with the file.

missing : int (default=-1)

Specify how missing data should be handled. If set to -1, missing data can account for both presence or absence of a cognate set in the given language. If set to 0, missing data is treated as absence.

degree : int (default=100)

The degree which is chosen for the projection of the tree layout.

Methods

analyze([runs, mixed, output_gml, tar,])	Carry out a full analysis using various parameters.
get_AVSD(glm, **keywords)	Function retrieves all pap s for ancestor languages in a given tree.
get_CVSD()	Calculate the Contemporary Vocabulary Size Distribution (CVSD).
get_GLS([mode, ratio, restriction,])	Create gain-loss-scenarios for all non-singleton paps in the data.
get_IVSD([output_gml, output_plot, tar,])	Calculate VSD on the basis of each item.
get_MLN(glm[, threshold, method])	Compute an Minimal Lateral Network for a given model.
get_MSN([glm, external_edges, deep_nodes])	Plot the Minimal Spatial Network.
get_PDC(glm, **keywords)	Calculate Patchily Distributed Cognates.
get_edge(glm, nodeA, nodeB[, entries, msn])	Return the edge data for a given gain-loss model.
get_stats(glm[, subset, filename])	Calculate basic statistics for a given gain-loss model.
plot_MLN([glm, fileformat, threshold,])	Plot the MLN with help of Matplotlib.
plot_MSN([glm, fileformat, threshold,])	Plot a minimal spatial network.
plot_concept_evolution(glm[, concept,])	Plot the evolution of specific concepts along the reference tree.
plot_two_concepts(concept, cogA, cogB[,])	Plot the evolution of two concepts in space.

Inherited Methods

pickle([filename])	Store the QLCParser instance in a pickle file.
get_entries(entry)	Return all entries matching the given entry-type as a two-dimensional list.
add_entries(entry, source, function[, override])	Add new entry-types to the word list by modifying given ones.
calculate(data[, taxa, concepts, ref])	Function calculates specific data.
export(fileformat[, sections, entries,])	Export the wordlist to specific fileformats.
get_dict([col, row, entry])	Function returns dictionaries of the cells matched by the indices.
get_etymdict([ref, entry, modify_ref])	Return an etymological dictionary representation of the word list.
get_list([row, col, entry, flat])	Function returns lists of rows and columns specified by their name.
get_paps([ref, entry, missing, modify_ref])	Function returns a list of present-absent-patterns of a given word list.
output(fileformat, **keywords)	Write wordlist to file.
renumber(source[, target, override])	Renumber a given set of string identifiers by replacing the ids by integers.

CHAPTER

SIX

HANDLING PHYLOGENETIC TREES

6.1 Trees (`Tree`)

6.1.1 Functions

random_tree(taxa[, branch_lengths])	Create a random tree from a list of taxa.

6.1.2 Classes

Tree(tree, **keywords)	Basic class for the handling of phylogenetic trees.

CHAPTER

SEVEN

PLOTTING DATA

7.1 Plotting Data and Results (plot)

The plot-module provides some general and some specific functions for the plotting of data and results. This module is not imported as a default, so you need to import it explicitly by typing:

```
>>> from lingpy.convert.plot import *
```

Or by typing:

```
>>> from lingpy.convert import plot
```

7.1.1 Functions

plot_gls(gls, treestring[, degree, fileformat])	Plot a gain-loss scenario for a given reference tree.
plot_tree(treestring[, degree, fileformat, root])	Plot a Newick tree to PDF or other graphical formats.
plot_concept_evolution(scenarios, tree[,])	Plot the evolution according to the MLN method of all words for a given concept.
plot_heatmap(wordlist[, filename,])	Create a heatmap-representation of shared cognates for a given wordlist.

CHAPTER

EIGHT

EVALUATION

8.1 Automatic Cognate Detection (`acd`)

This module provides functions that can be used to evaluate how well algorithms perform in the task of automatic cognate detection.

8.1.1 Functions

bcubes(wordlist[, gold, test, modify_ref,])	Compute B-Cubed scores for test and reference datasets.
partial_bcubes(wordlist, gold, test[, pprint])	Compute B-Cubed scores for test and reference datasets for partial cognate detection.
pairs(lex[, gold, test, modify_ref, pprint,])	Compute pair scores for the evaluation of cognate detection algorithms.
diff(wordlist[, gold, test, modify_ref,])	Write differences in classifications on an item-basis to file.
npoint_ap(scores, cognates[, reverse])	Calculate the n-point average precision.
random_cognates(wordlist[, ref, bias])	Populate a wordlist with random cognates for each entry.
extreme_cognates(wordlist[, ref, bias])	Return extreme cognates, either lump all words together or split them.

8.2 Automatic Linguistic Reconstruction (`acd`)

This module provides functions that can be used to evaluate how well algorithms perform in the task of automatic linguistic reconstruction.

8.2.1 Functions

mean_edit_distance(wordlist[, gold, test,])	Function computes the edit distance between gold standard and test set.

8.3 Automatic Phonetic Alignment (`apa`)

This module provides functions that can be used to evaluate how well algorithms perform in the task of automatic phonetic alignment analyses.

8.3.1 Classes

`EvalPSA`(gold, test)	Base class for the evaluation of automatic pairwise sequence analyses.
`EvalMSA`(gold, test)	Base class for the evaluation of automatic multiple sequence analyses.

CHAPTER

NINE

REFERENCE

9.1 Reference

9.1.1 lingpy package

Subpackages

lingpy.algorithm package

Subpackages

lingpy.algorithm.cython package

Submodules

lingpy.algorithm.cython.calign module

`lingpy.algorithm.cython.calign.`**`align_pair`**`()`

Align a pair of sequences.

Parameters **seqA, seqB** : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

mode : { global, local, overlap, dialign }

Select one of the four basic modes for alignment analyses.

restricted_chars : str

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

distance : int (default=0)

Select whether you want to calculate the normalized distance or the similarity between two strings (following `Downey2008` for normalization).

Returns alignment : tuple

The aligned sequences and the similarity or distance.

Notes

This is a utility function that allows calls any of the four classical alignment functions (*`lingpy.algorithm.cython.calign.globalign`* *`lingpy.algorithm.cython.calign.semi_globalign`*, *`lingpy.algorithm.cython.calign.localign`*, *`lingpy.algorithm.cython.calign.dialign`*,) and their secondary counterparts.

`lingpy.algorithm.cython.calign.`**`align_pairs`**`()`

Align multiple sequence pairs.

Parameters seqs : list

A two-dimensional list containing one pair of sequences each.

gops : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

pros : list

The prosodic strings which have the same length as seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

mode : { global, local, overlap, dialign }

Select one of the four basic modes for alignment analyses.

restricted_chars : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

distance : int (default=0)

Select whether you want to calculate the normalized distance or the similarity between two strings (following Downey2008 for normalization). If you set this value to 2, both distances and similarities will be returned.

Returns alignments : list

A list of tuples of size 3 or 4, containing the alignments, and the similarity or the distance (or both, if distance is set to 2).

Notes

This function computes alignments of all pairs passed in the list of sequence pairs (a two-dimensional list with two sequences each) and is basically used in LingPys module for cognate detection (*lingpy.compare.lexstat.LexStat*).

`lingpy.algorithm.cython.calign.`**`align_pairwise`**`()`

Align a list of sequences pairwise.

Parameters seqs : list

The list containing the sequences.

gops : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

pros : list

The prosodic strings which have the same length as seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring function which needs to provide scores for all segments in seqA and seqB.

mode : { global, local, overlap, dialign }

Select one of the four basic modes for alignment analyses.

r : str

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

Returns alignments : list

A list of tuples of size 4, containing the alignment, the similarity and the distance for each sequence pair.

Notes

This function computes alignments of all possible pairs passed in the list of sequences and is basically used in LingPys module for multiple alignment analyses (*lingpy.align.multiple*).

`lingpy.algorithm.cython.calign.`**`align_profile`**`()`

Align two profiles using the basic modes.

Parameters **profileA, profileB** : list

Two-dimensional list for each of the profiles.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as profileA and profileB.

gop : int

The general gap opening penalty which will be used to introduce a gap between the two profiles.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in the two profiles.

restricted_chars : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence. They need to be computed by computing a consensus string from all prosodic strings in the profile.

mode : { global, local, overlap, dialign }

Select one of the four basic modes for alignment analyses.

gap_weight : float

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

Returns **alignment** : tuple

The aligned profiles, and the overall similarity of the profiles.

Notes

This function computes alignments of two profiles of multiple sequences (see Durbin2002 for details on profiles) and is basically used in LingPys module for multiple alignment (*lingpy.align.multiple*).

`lingpy.algorithm.cython.calign.`**`corrdist`**`()`

Create a correspondence distribution for a given language pair.

Parameters **threshold** : float

The threshold of sequence distance which determines whether a sequence pair is included or excluded from the calculation of the distribution.

seqs : list

The sequences passed as a two-dimensional list of sequence pairs.

gops : list

The gap opening penalties, passed as individual lists of penalties for each sequence.

pros : list

The list of prosodic strings for each sequence.

gop : int

The general gap opening penalty which will be used to introduce a gap between the two profiles.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in the two profiles.

mode : { global, local, overlap, dialign }

Select one of the four basic modes for alignment analyses.

restricted_chars : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence. They need to be computed by computing a consensus string from all prosodic strings in the profile.

Returns **results** : tuple

A dictionary containing the distribution, and the number of included sequences.

Notes

This function is the core of the *`LexStat`* function to compute distributions of aligned segment pairs.

`lingpy.algorithm.cython.calign.`**`dialign`**`()`

Carry out dialign alignment of two sequences.

Parameters **seqA, seqB** : list

The list containing the sequences.

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

Returns alignment : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out local dialign alignment analyses (keyword dialign) when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *Pairwise*, *Multiple*, or *LexStat*. Dialign (see `Morgenstern1996`) is an alignment algorithm that does not require gap penalties and generally works in a rather local fashion.

`lingpy.algorithm.cython.calign.`**`globalign`**`()`

Carry out global alignment of two sequences.

Parameters seqA, seqB : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

Returns alignment : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out global alignment analyses when using many of LingPys classes for alignment analyses, like *Pairwise*, *Multiple*, or *LexStat*. It differs from classical Needleman-Wunsch alignment (compare Needleman1970) in a couple of aspects. These include, among others, the use of a gap extension *scale* rather than a gap extension penalty (the scale consecutively reduces the gap penalty and thus lets gap penalties approach zero if gapped regions are large), the use of individual gap opening penalties for all positions of a sequence, and the use of prosodic strings, and prosodic factors that raise scores when segments occur in the same prosodic environment.

If one sets certain of these parameters to zero or one and uses the same gap opening penalties, however, the function will behave like the traditional Needleman-Wunsch algorithm, and since it is implemented in Cython, it will work faster than a pure Python implementation for alignment algorithms.

Examples

We show that the Needleman-Wunsch algorithms yields the same result as the globalign algorithm, provided we adjust the parameters:

```
>>> from lingpy.algorithm.cython.calign import globalign
>>> from lingpy.align.pairwise import nw_align
>>> nw_align('abab', 'baba')
(['a', 'b', 'a', 'b', '-'], ['-', 'b', 'a', 'b', 'a'], 1)

>>> globalign(list('abab'), list('baba'), 4 * [-1], 4 * [-1], 'aaaa', 'aaaa', 4,␣
↪4, 1, 0, {("a","b"):-1, ("b","a"): -1, ("a","a"): 1, ("b", "b"): 1})
(['a', 'b', 'a', 'b', '-'], ['-', 'b', 'a', 'b', 'a'], 1.0)
```

`lingpy.algorithm.cython.calign.`**`localign`**`()`

Carry out semi-global alignment of two sequences.

Parameters **seqA, seqB** : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

Returns alignment : tuple

A tuple of the two alignments and the alignment score. The alignments are each a list of suffix, alignment, and prefix.

Notes

This is the function that is called to carry out local alignment analyses when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *Pairwise*, *Multiple*, or *LexStat*. Local alignment means that only the best matching substring between two sequences is returned (compare `Smith1981`), also called the Smith-Waterman algorithm.

`lingpy.algorithm.cython.calign.`**`score_profile`**`()`

Basic function for the scoring of profiles.

Parameters colA, colB : list

The two columns of a profile.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in the two profiles.

gap_weight : float (default=0.0)

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

Returns score : float

The score for the profile

Notes

This function handles how profiles are scored.

`lingpy.algorithm.cython.calign.`**`secondary_dialign`**`()`

Carry out dialign alignment of two sequences with sensitivity for secondary sequence structures.

Parameters seqA, seqB : list

The list containing the sequences.

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

r : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

Returns alignment : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out local dialign alignment analyses (keyword dialign) when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *Pairwise*, *Multiple*, or *LexStat*. Dialign (see `Morgenstern1996`) is an alignment algorithm that does not require gap penalties and generally works in a rather local fashion.

`lingpy.algorithm.cython.calign.`**`secondary_globalign`**`()`

Carry out global alignment of two sequences with secondary sequence structures.

Parameters seqA, seqB : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

r : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

Returns alignment : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out global alignment analyses when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in List2014d for details), like *Pairwise*, *Multiple*, or *LexStat*. It differs from classical Needleman-Wunsch alignment (compare Needleman1970) in a couple of aspects. These include, among others, the use of a gap extension *scale* rather than a gap extension penalty (the scale consecutively reduces the gap penalty and thus lets gap penalties approach zero if gapped regions are large), the use of individual gap opening penalties for all positions of a sequence, and the use of prosodic strings, and prosodic factors that raise scores when segments occur in the same prosodic environment.

If one sets certain of these parameters to zero or one and uses the same gap opening penalties, however, the function will behave like the traditional Needleman-Wunsch algorithm, and since it is implemented in Cython, it will work faster than a pure Python implementation for alignment algorithms.

Examples

We compare globalign with secondary_globalign::

```
>>> from lingpy.algorithm.cython.calign import globalign, secondary_globalign
>>> globalign(list('abab'), list('baba'), 4 * [-1], 4 * [-1], 'aaaa', 'aaaa',␣
↪4, 4, 1, 0, {("a","b"):-1, ("b","a"): -1, ("a","a"): 1, ("b", "b"): 1})
(['a', 'b', 'a', 'b', '-'], ['-', 'b', 'a', 'b', 'a'], 1.0)
>>> secondary_globalign(list('ab.ab'), list('ba.ba'), 5 * [-1], 5 * [-1], 'ab.
↪ab', 'ba.ba', 5, 5, 1, 0, {("a","b"):-1, ("b","a"): -1, ("a","a"): 1, ("b",
↪"b"): 1, ("a",".") : -1, ("b","."):-1, (".","."):0, (".", "b"): -1, (".", "a
↪"):-1}, '.')
(['a', 'b', '-', '.', 'a', 'b', '-'],
['-', 'b', 'a', '.', '-', 'b', 'a'],
-2.0)
```

lingpy.algorithm.cython.calign.**secondary_localign**()

Carry out lobal alignment of two sequences with sensitivity to secondary sequence structures.

Parameters **seqA, seqB** : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

r : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

Returns alignment : tuple

A tuple of the two alignments and the alignment score. The alignments are each a list of suffix, alignment, and prefix.

Notes

This is the function that is called to carry out local alignment analyses when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *Pairwise*, *Multiple*, or *LexStat*. Local alignment means that only the best matching substring between two sequences is returned (compare `Smith1981`), also called the Smith-Waterman algorithm.

`lingpy.algorithm.cython.calign.`**`secondary_semi_globalign`**`()`

Carry out semi-global alignment of two sequences with sensitivity to secondary sequence structures.

Parameters seqA, seqB : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

r : { str }

The string containing restricted characters. Restricted characters occur, as a rule, in the prosodic strings, not in the normal sequence.

Returns alignment : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out semi-global alignment analyses (keyword overlap) when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *`Pairwise`*, *`Multiple`*, or *`LexStat`*. Semi-global alignment means that the suffixes or prefixes in one of the words are not penalized.

`lingpy.algorithm.cython.calign.`**`semi_globalign`**`()`

Carry out semi-global alignment of two sequences.

Parameters **seqA, seqB** : list

The list containing the sequences.

gopA, gopB : list

The gap opening penalties (individual for each sequence, therefore passed as a list of floats or integers).

proA, proB : str

The prosodic strings which have the same length as seqA and seqB.

M, N : int

The lengths of seqA and seqB.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

factor : float

The factor by which matches are increased when two segments occur in the same prosodic position of an alignment.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in seqA and seqB.

Returns **alignment** : tuple

A tuple of the two alignments and the alignment score.

Notes

This is the function that is called to carry out semi-global alignment analyses (keyword overlap) when using many of LingPys classes for alignment analyses which is at the same time sensitive for secondary sequence structures (see the description of secondary alignment in `List2014d` for details), like *`Pairwise`*, *`Multiple`*, or *`LexStat`*. Semi-global alignment means that the suffixes or prefixes in one of the words are not penalized.

Examples

We compare globalign with semi_globalign::

```
>>> from lingpy.algorithm.cython.calign import globalign, semi_globalign
>>> globalign(list('abab'), list('baba'), 4 * [-1], 4 * [-1], 'aaaa', 'aaaa',␣
↪4, 4, 1, 0, {("a","b"):-1, ("b","a"): -1, ("a","a"): 1, ("b", "b"): 1})
(['a', 'b', 'a', 'b', '-'], ['-', 'b', 'a', 'b', 'a'], 1.0)
```

(continues on next page)

(continued from previous page)

```
>>> semi_globalign(list('abab'), list('baba'), 4 * [-1], 4 * [-1], 'aaaa',
↪'aaaa', 4, 4, 1, 0, {("a","b"):-1, ("b","a"): -1, ("a","a"): 1, ("b", "b"):␣
↪1})
(['a', 'b', 'a', 'b', '-'], ['-', 'b', 'a', 'b', 'a'], 3.0)
```

`lingpy.algorithm.cython.calign.`**`swap_score_profile`**`()`

Basic function for the scoring of profiles which contain swapped sequences.

Parameters **colA, colB** : list

The two columns of a profile.

scorer : { dict, *lingpy.algorithm.cython.misc.ScoreDict* }

The scoring function which needs to provide scores for all segments in the two profiles.

gap_weight : float (default=0.0)

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

swap_penalty : int (default=-5)

The swap penalty applied to swapped columns.

Returns **score** : float

The score for the profile.

Notes

This function handles how profiles with swapped segments are scored.

lingpy.algorithm.cython.cluster module

`lingpy.algorithm.cython.cluster.`**`flat_cluster`**`()`

Carry out a flat cluster analysis based on the UPGMA algorithm.

Parameters **method** : str { upgma, single, complete }

Select between ugpma, single, and complete.

threshold : float

The threshold which terminates the algorithm.

matrix : list or `numpy.array`

A two-dimensional list containing the distances.

taxa : list (default = [])

A list containing the names of the taxa. If the list is left empty, the indices of the taxa will be returned instead of their names.

Returns **clusters** : dict

A dictionary with cluster-IDs as keys and a list of the taxa corresponding to the respective ID as values.

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
array([[ 0.  ,  0.5 ,  0.67,  0.8 ,  0.2 ],
       [ 0.5 ,  0.  ,  0.4 ,  0.7 ,  0.6 ],
       [ 0.67,  0.4 ,  0.  ,  0.8 ,  0.8 ],
       [ 0.8 ,  0.7 ,  0.8 ,  0.  ,  0.3 ],
       [ 0.2 ,  0.6 ,  0.8 ,  0.3 ,  0.  ]])
```

Carry out the flat cluster analysis.

```
>>> flat_upgma(0.5,matrix,taxa)
{0: ['German', 'Dutch', 'English'], 1: ['Swedish', 'Icelandic']}
```

`lingpy.algorithm.cython.cluster.`**`flat_upgma`**`()`

Carry out a flat cluster analysis based on the UPGMA algorithm (`Sokal1958`).

Parameters threshold : float

The threshold which terminates the algorithm.

matrix : list or `numpy.array`

A two-dimensional list containing the distances.

taxa : list (default = [])

A list containing the names of the taxa. If the list is left empty, the indices of the taxa will be returned instead of their names.

Returns clusters : dict

A dictionary with cluster-IDs as keys and a list of the taxa corresponding to the respective ID as values.

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
array([[ 0.  ,  0.5 ,  0.67,  0.8 ,  0.2 ],
       [ 0.5 ,  0.  ,  0.4 ,  0.7 ,  0.6 ],
       [ 0.67,  0.4 ,  0.  ,  0.8 ,  0.8 ],
       [ 0.8 ,  0.7 ,  0.8 ,  0.  ,  0.3 ],
       [ 0.2 ,  0.6 ,  0.8 ,  0.3 ,  0.  ]])
```

Carry out the flat cluster analysis.

```
>>> flat_upgma(0.5,matrix,taxa)
{0: ['German', 'Dutch', 'English'], 1: ['Swedish', 'Icelandic']}
```

`lingpy.algorithm.cython.cluster.`**`neighbor`**`()`

Function clusters data according to the Neighbor-Joining algorithm (`Saitou1987`).

Parameters **matrix** : list or `numpy.array`

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

distances : bool

If set to `False`, only the topology of the tree will be returned.

Returns **newick** : str

A string in newick-format which can be further used in biological software packages to view and plot the tree.

Examples

Function is automatically imported when importing lingpy.

```
>>> from lingpy import *
```

Create an arbitrary list of taxa.

```
>>> taxa = ['Norwegian','Swedish','Icelandic','Dutch','English']
```

Create an arbitrary matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
```

Carry out the cluster analysis.

```
>>> neighbor(matrix,taxa)
'(((Norwegian,(Swedish,Icelandic)),English),Dutch);'
```

`lingpy.algorithm.cython.cluster.`**`upgma`**`()`

Carry out a cluster analysis based on the UPGMA algorithm (`Sokal1958`).

Parameters **matrix** : list or `numpy.array`

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

distances : bool

If set to `False`, only the topology of the tree will be returned.

Returns newick : str

A string in newick-format which can be further used in biological software packages to view and plot the tree.

Examples

Function is automatically imported when importing lingpy.

```
>>> from lingpy import *
```

Create an arbitrary list of taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
```

Carry out the cluster analysis.

```
>>> upgma(matrix,taxa,distances=False)
'((Swedish,Icelandic),(English,(German,Dutch)));'
```

lingpy.algorithm.cython.compilePYX module

Script handles compilation of Cython files to C and also to C-Extension modules.

`lingpy.algorithm.cython.compilePYX.`**`main`**`()`

`lingpy.algorithm.cython.compilePYX.`**`pyx2py`**`(`*infile*, *debug=False*`)`

lingpy.algorithm.cython.malign module

This module provides various alignment functions in an optimized version.

`lingpy.algorithm.cython.malign.`**`edit_dist`**`()`

Return the edit-distance between two strings.

Parameters seqA, seqB : list

The sequences to be aligned, passed as list.

normalized : bool

Indicate whether you want the normalized or the unnormalized edit distance to be returned.

Returns dist : { int, float }

Either the normalized or the unnormalized edit distance.

`lingpy.algorithm.cython.malign.`**`nw_align`**`()`

Align two sequences using the Needleman-Wunsch algorithm.

Parameters **seqA, seqB** : list

The sequences to be aligned, passed as list.

scorer : dict

A dictionary containing tuples of two segments as key and numbers as values.

gap : int

The gap penalty.

Returns **alignment** : tuple

A tuple of the two aligned sequences, and the similarity score.

Notes

This function is a very straightforward implementation of the Needleman-Wunsch algorithm (`Needleman1970`). We recommend to use the function if you want to test your own scoring dictionaries and profit from a fast implementation (as we use Cython, the implementation is indeed faster than pure Python implementations, as long as you use Python 3 and have Cython installed). If you want to test the NW algorithm without specifying a scoring dictionary, we recommend to have a look at our wrapper function with the same name in the *`pairwise`* module.

`lingpy.algorithm.cython.malign.`**`restricted_edit_dist`**`()`

Return the restricted edit-distance between two strings.

Parameters **seqA, seqB** : list

The two sequences passed as list.

resA, resB : str

The restrictions passed as a string with the same length as the corresponding sequence. We note a restriction if the strings show different symbols in their restriction string. If the symbols are identical, it is modeled as a non-restriction.

normalized : bool

Determine whether you want to return the normalized or the unnormalized edit distance.

Notes

Restrictions follow the definition of `Heeringa2006`: Segments that are not allowed to match are given a penalty of ∞. We model restrictions as strings, for example consisting of letters c and v. So the sequence woldemort could be modeled as cvccvcvcc, and when aligning it with the sequence walter and its restriction string cvccvc, the matching of those segments in the sequences in which the segments of the restriction string differ, would be heavily penalized, thus prohibiting an alignment of vowels and consonants (v and c).

`lingpy.algorithm.cython.malign.`**`structalign`**`()`

Carry out a structural alignment analysis using Dijkstras algorithm.

Parameters **seqA,seqB** : str

The input sequences.

restricted_chars : str (default =)

The characters which are used to separate secondary from primary segments in the input sequences. Currently, the use of restricted chars may fail to yield an alignment.

Notes

Structural alignment is hereby understood as an alignment of two sequences whose alphabets differ. The algorithm returns all alignments with minimal edit distance. Edit distance in this context refers to the number of edit operations that are needed in order to convert one sequence into the other, with repeated edit operations being penalized only once.

`lingpy.algorithm.cython.malign.`**`sw_align`**`()`

Align two sequences using the Smith-Waterman algorithm.

Parameters seqA, seqB : list

The sequences to be aligned, passed as list.

scorer : dict

A dictionary containing tuples of two segments as key and numbers as values.

gap : int

The gap penalty.

Returns alignment : tuple

A tuple of the two aligned sequences, and the similarity score.

Notes

This function is a very straightforward implementation of the Smith-Waterman algorithm (`Smith1981`). We recommend to use the function if you want to test your own scoring dictionaries and profit from a fast implementation (as we use Cython, the implementation is indeed faster than pure Python implementations, as long as you use Python 3 and have Cython installed). If you want to test the SW algorithm without specifying a scoring dictionary, we recommend to have a look at our wrapper function with the same name in the *`pairwise`* module.

`lingpy.algorithm.cython.malign.`**`we_align`**`()`

Align two sequences using the Waterman-Eggert algorithm.

Parameters seqA, seqB : list

The input sequences passed as a list.

scorer : dict

A dictionary containing tuples of two segments as key and numbers as values.

gap : int

The gap penalty.

Returns alignments : list

A list consisting of tuples. Each tuple gives the alignment of one of the subsequences of the input sequences. Each tuple contains the aligned part of the first, the aligned part of the second sequence, and the score of the alignment.

Notes

This function is a very straightforward implementation of the Waterman-Eggert algorithm (`Waterman1987`). We recommend to use the function if you want to test your own scoring dictionaries and profit from a fast implementation (as we use Cython, the implementation is indeed faster than pure Python implementations, as long as you use Python 3 and have Cython installed). If you want to test the WE algorithm without specifying a scoring dictionary, we recommend to have a look at our wrapper function with the same name in the *`pairwise`* module.

lingpy.algorithm.cython.misc module

class `lingpy.algorithm.cython.misc.`**`ScoreDict`**
Bases: `object`

Class allows quick access to scoring functions using dictionary syntax.

Parameters **chars** : list

The list of all character tokens for the scoring dictionary.

matrix : list

A two-dimensional scoring matrix.

Notes

Since this class has dictionary syntax, you can always also just create a dictionary in order to store your scoring functions. Scoring dictionaries should contain a tuple of segments to be compared as a key, and a float or integer as a value, with negative values indicating dissimilarity, and positive values similarity.

Examples

Initialize a ScoreDict object::

```
>>> from lingpy.algorith.cython.misc import ScoreDict
>>> scorer = ScoreDict(['a', 'b'], [1, -1, -1, 1])
```

Retrieve scores::

```
>>> scorer['a', 'b']
-1
>>> scorer['a', 'a']
1
>>> scorer['a', 'X']
-22.5
```

`lingpy.algorithm.cython.misc.`**`squareform`**`()`
A simplified version of the `scipy.spatial.distance.squareform()` function.

Parameters **x** : `numpy.array` or list

The one-dimensional flat representation of a symmetrix distance matrix.

Returns **matrix** : `numpy.array`

The two-dimensional redundant representation of a symmetric distance matrix.

`lingpy.algorithm.cython.misc.`**`transpose`**`()`

Transpose a matrix along its two dimensions.

Parameters **matrix** : list

A two-dimensional list.

lingpy.algorithm.cython.talign module

`lingpy.algorithm.cython.talign.`**`align_pair`**`()`

Align a pair of sequences.

Parameters **seqA, seqB** : list

The sequences to be aligned, passed as lists.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

mode : { global, local, overlap, dialign }

Select the mode for the alignment analysis (overlap refers to semi-global alignments).

distance : int (default=0)

Select whether you want distances or similarities to be returned (0 indicates similarities, 1 indicates distances, 2 indicates both).

Returns **alignment** : tuple

The aligned sequences and the similarity or distance scores, or both.

Notes

This is a utility function that allows calls any of the four classical alignment functions (*`lingpy.algorithm.cython.talign.globalign`* *`lingpy.algorithm.cython.talign.semi_globalign`*, `lingpy.algorithm.cython.talign.lotalign`, *`lingpy.algorithm.cython.talign.dialign`*,) and their secondary counterparts.

`lingpy.algorithm.cython.talign.`**`align_pairs`**`()`

Align multiple sequence pairs.

Parameters **seqs** : list

The sequences to be aligned, passed as lists.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

mode : { global, local, overlap, dialign }

Select the mode for the alignment analysis (overlap refers to semi-global alignments).

distance : int (default=0)

Indicate whether distances or similarities should be returned.

Returns alignments : list

A list of tuples, containing the aligned sequences, and the similarity or the distance scores.

Notes

This function aligns all pairs which are passed to it.

`lingpy.algorithm.cython.talign.`**`align_pairwise`**`()`

Align all sequences pairwise.

Parameters seqs : list

The sequences to be aligned, passed as lists.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

mode : { global, local, overlap, dialign }

Select the mode for the alignment analysis (overlap refers to semi-global alignments).

Returns alignments : list

A list of tuples, containing the aligned sequences, the similarity and the distance scores.

Notes

This function aligns all possible pairs between the sequences you pass to it. It is important for multiple alignment, where it can be used to construct the guide tree.

`lingpy.algorithm.cython.talign.`**`align_profile`**`()`

Align two profiles using the basic modes.

Parameters profileA, profileB : list

Two-dimensional list for each of the profiles.

gop : int

The gap opening penalty.

scale : float

The gap extension scale by which consecutive gaps are reduced. LingPy uses a scale rather than a constant gap extension penalty.

scorer : { dict, `lingpy.algorithm.cython.misc.ScoreDict` }

The scoring function which needs to provide scores for all segments in the two profiles.

mode : { global, overlap, dialign }

Select one of the four basic modes for alignment analyses.

gap_weight : float

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

Returns alignment : tuple

The aligned profiles, and the overall similarity of the profiles.

Notes

This function computes alignments of two profiles of multiple sequences (see `Durbin2002` for details on profiles) and is important for multiple alignment analyses.

`lingpy.algorithm.cython.talign.`**`dialign`**`()`

Carry out dialign alignment of two sequences.

Parameters seqA, seqB : list

The sequences to be aligned, passed as lists.

M, N : int

The length of the two sequences.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

Returns alignment : tuple

The aligned sequences and the similarity score.

Notes

This algorithm carries out dialign alignment (`Morgenstern1996`).

`lingpy.algorithm.cython.talign.`**`globalign`**`()`

Carry out global alignment of two sequences.

Parameters seqA, seqB : list

The sequences to be aligned, passed as lists.

M, N : int

The length of the two sequences.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

Returns alignment : tuple

The aligned sequences and the similarity score.

Notes

This algorithm carries out classical Needleman-Wunsch alignment (`Needleman1970`).

`lingpy.algorithm.cython.talign.`**`localign`**`()`

Carry out semi-global alignment of two sequences.

Parameters seqA, seqB : list

The sequences to be aligned, passed as lists.

M, N : int

The length of the two sequences.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

Returns alignment : tuple

The aligned sequences and the similarity score.

Notes

This algorithm carries out local alignment (`Smith1981`).

`lingpy.algorithm.cython.talign.`**`score_profile`**`()`

Basic function for the scoring of profiles.

Parameters colA, colB : list

The two columns of a profile.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in the two profiles.

gap_weight : float (default=0.0)

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

Returns **score** : float

The score for the profile

Notes

This function handles how profiles are scored.

`lingpy.algorithm.cython.talign.`**`semi_globalign`**`()`

Carry out semi-global alignment of two sequences.

Parameters **seqA, seqB** : list

The sequences to be aligned, passed as lists.

M, N : int

The length of the two sequences.

gop : int

The gap opening penalty.

scale : float

The gap extension scale.

scorer : { dict, ~lingpy.algorithm.cython.misc.ScoreDict }

The scoring dictionary containing scores for all possible segment combinations in the two sequences.

Returns **alignment** : tuple

The aligned sequences and the similarity score.

Notes

This algorithm carries out semi-global alignment (`Durbin2002`).

`lingpy.algorithm.cython.talign.`**`swap_score_profile`**`()`

Basic function for the scoring of profiles in swapped sequences.

Parameters **colA, colB** : list

The two columns of a profile.

scorer : { dict, *`lingpy.algorithm.cython.misc.ScoreDict`* }

The scoring function which needs to provide scores for all segments in the two profiles.

gap_weight : float (default=0.0)

This handles the weight that is given to gaps in a column. If you set it to 0, for example, this means that all gaps will be ignored when determining the score for two columns in the profile.

swap_penalty : int (default=-5)

The swap penalty applied to swapped columns.

Returns **score** : float

The score for the profile.

Notes

This function handles how profiles with swapped segments are scored.

Module contents

Package provides modules for time-consuming routines.

Submodules

lingpy.algorithm.cluster_util module

Various utility functions which are useful for algorithmic operations

`lingpy.algorithm.cluster_util.`**`generate_all_clusters`**`(`*numbers*`)`

Generate all possible clusters for a number of elements.

Returns **clr** : iterator

An iterator that will yield the next of all possible clusters.

See also:

`valid_cluster`, *`generate_random_cluster`*, *`order_cluster`*, *`mutate_cluster`*

`lingpy.algorithm.cluster_util.`**`generate_random_cluster`**`(`*numbers*, *bias=False*`)`

Generate a random cluster for a number of elements.

Parameters **numbers** : int

Number of separate entities which should be clustered.

bias : str (default=False)

When set to lumper will tend to create larger groups, when set to splitter it will tend to produce smaller groups.

Returns **cluster** : list

A list with consecutive ordering of clusters, starting from zero.

See also:

`valid_cluster`, *`generate_all_clusters`*, *`order_cluster`*, *`mutate_cluster`*

`lingpy.algorithm.cluster_util.`**`mutate_cluster`**`(`*clr*, *chance=0.5*`)`

Mutate a cluster.

Parameters **clr** : cluster

A list with ordered clusters.

chance : float (default=0.5)

The mutation rate for each element in a cluster. If set to 0.5, this means that in 50% of the cases, an element will be assigned to another cluster or a new cluster.

Returns valid_cluster : list

A newly clustered list in consecutive order.

See also:

valid_cluster, *generate_all_clusters*, *generate_random_cluster*, *order_cluster*

`lingpy.algorithm.cluster_util.`**`order_cluster`**`(`*clr*`)`

Order a cluster into the form of a valid cluster.

Parameters clr : list

A list with clusters assigned by given each element a specific clusuter ID.

Returns valid_cluster : list

A list in which the IDs start from zero and increase consecutively with each new cluster introduced.

See also:

valid_cluster, *generate_all_clusters*, *generate_random_cluster*, *mutate_cluster*

`lingpy.algorithm.cluster_util.`**`valid_cluster`**`(`*sequence*`)`

Only allow to have sequences which have consecutive ordering of elements.

Parameters sequence : list

A cluster sequence in which elements should be consecutively ordered, starting from 0, and identical segments in the sequence retrieve the same number.

Returns valid_cluster : bool

True, if the cluster is valid, and False if it judged to be invalid.

See also:

generate_all_clusters, *generate_random_cluster*, *order_cluster*, *mutate_cluster*

Examples

We define a valid and an invalid cluster sequence:

```
>>> clrA = [0, 1, 2, 3]
>>> clrB = [1, 1, 2, 3] # should be [0, 0, 1, 2]
>>> from lingpy.algorithm.utils import valid_cluster
>>> valid_cluster(clrA)
True
>>> valid_cluster(clrB)
False
```

lingpy.algorithm.clustering module

Module provides general clustering functions for LingPy.

`lingpy.algorithm.clustering.`**`best_threshold`**`(`*matrix*, *trange=(0.3, 0.7, 0.05)*`)`

Calculate the best threshold by maximizing partition density for a given range of thresholds.

Notes

This method makes use of the idea of partition density proposed in Ahn2010.

`lingpy.algorithm.clustering.`**`check_taxon_names`**(*taxa*)

`lingpy.algorithm.clustering.`**`find_threshold`**(*matrix, thresholds=[0.9, 0.8500000000000001, 0.8, 0.75, 0.7000000000000001, 0.65, 0.6000000000000001, 0.55, 0.5, 0.45, 0.4, 0.3500000000000003, 0.30000000000000004, 0.25, 0.2, 0.15000000000000002, 0.1, 0.05], logs=True*)

Use a variant of the method by Apeltsin2011 in order to find an optimal threshold.

Parameters **matrix** : list

The distance matrix for which the threshold shall be determined.

thresholds : list (default=[i*0.05 for i in range(1,19)[::-1])

The range of thresholds that shall be tested.

logs : {bool,builtins.function} (default=True)

If set to **True**, the logarithm of the score beyond the threshold will be assigned as weight to the graph. If set to c{False} all weights will be set to 1. Use a custom function to define individual ways to calculate the weights.

Returns **threshold** : {float,None}

If a float is returned, this is the threshold identified by the method. If **None** is returned, no threshold could be identified.

Notes

This is a very simple method that may not work well depending on the dataset. So we recommend to use it with great care.

`lingpy.algorithm.clustering.`**`flat_cluster`**(*method, threshold, matrix, taxa=None, revert=False*)

Carry out a flat cluster analysis based on linkage algorithms.

Parameters **method** : { upgma, single, complete, ward}

Select between ugpma, single, and complete. You can also test ward, but theres no guarantee that this is the correct algorithm.

threshold : float

The threshold which terminates the algorithm.

matrix : list

A two-dimensional list containing the distances.

taxa : list (default=None)

A list containing the names of the taxa. If the list is left empty, the indices of the taxa will be returned instead of their names.

Returns **clusters** : dict

A dictionary with cluster-IDs as keys and a list of the taxa corresponding to the respective ID as values.

See also:

flat_cluster, *flat_upgma*, *fuzzy*, *link_clustering*, *mcl*

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
[[0.0, 0.5, 0.67, 0.8, 0.2],
 [0.5, 0.0, 0.4, 0.7, 0.6],
 [0.67, 0.4, 0.0, 0.8, 0.8],
 [0.8, 0.7, 0.8, 0.0, 0.3],
 [0.2, 0.6, 0.8, 0.3, 0.0]]
```

Carry out the flat cluster analysis.

```
>>> flat_cluster('upgma',0.6,matrix,taxa)
{0: ['German', 'Dutch', 'English'], 1: ['Swedish', 'Icelandic']}
```

`lingpy.algorithm.clustering.`**`flat_upgma`**(*threshold*, *matrix*, *taxa=None*, *revert=False*)

Carry out a flat cluster analysis based on the UPGMA algorithm (Sokal1958).

Parameters threshold : float

The threshold which terminates the algorithm.

matrix : list

A two-dimensional list containing the distances.

taxa : list (default=None)

A list containing the names of the taxa. If the list is left empty, the indices of the taxa will be returned instead of their names.

Returns clusters : dict

A dictionary with cluster-IDs as keys and a list of the taxa corresponding to the respective ID as values.

See also:

flat_cluster, *flat_upgma*, *fuzzy*, *link_clustering*, *mcl*

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
[[0.0, 0.5, 0.67, 0.8, 0.2],
 [0.5, 0.0, 0.4, 0.7, 0.6],
 [0.67, 0.4, 0.0, 0.8, 0.8],
 [0.8, 0.7, 0.8, 0.0, 0.3],
 [0.2, 0.6, 0.8, 0.3, 0.0]]
```

Carry out the flat cluster analysis.

```
>>> flat_upgma(0.6,matrix,taxa)
{0: ['German', 'Dutch', 'English'], 1: ['Swedish', 'Icelandic']}
```

`lingpy.algorithm.clustering.`**`fuzzy`**(*threshold*, *matrix*, *taxa*, *method='upgma'*, *revert=False*)

Create fuzzy cluster of a given distance matrix.

Parameters **threshold** : float

The threshold that shall be used for the basic clustering of the data.

matrix : list

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

method : { upgma, single, complete } (default=upgma)

Select the method for the flat cluster analysis.

distances : bool

If set to False, only the topology of the tree will be returned.

revert : bool (default=False)

Specify whether a reverted dictionary should be returned.

Returns **cluster** : dict

A dictionary with cluster-IDs as keys and a list as value, containing the taxa that are assigned to a given cluster-ID.

See also:

`link_clustering`

Notes

This is a very simple fuzzy clustering algorithm. It basically does nothing else than removing taxa successively from the matrix, flat-clustering the remaining taxa with the corresponding threshold, and then returning a combined consensus cluster in which taxa may be assigned to multiple clusters.

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
from lingpy.algorithm import squareform
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
[[0.0, 0.5, 0.67, 0.8, 0.2],
 [0.5, 0.0, 0.4, 0.7, 0.6],
 [0.67, 0.4, 0.0, 0.8, 0.8],
 [0.8, 0.7, 0.8, 0.0, 0.3],
 [0.2, 0.6, 0.8, 0.3, 0.0]]
```

Carry out the fuzzy flat cluster analysis.

```
>>> fuzzy(0.5,matrix,taxa)
{1: ['Swedish', 'Icelandic'], 2: ['Dutch', 'German'], 3: ['Dutch', 'English']}
```

`lingpy.algorithm.clustering.`**`link_clustering`**(*threshold*, *matrix*, *taxa*, *link_threshold=False*, *revert=False*, *matrix_type='distances'*, *fuzzy=True*)

Carry out a link clustering analysis using the method by `Ahn2010`.

Parameters **threshold** : {float, bool}

The threshold that shall be used for the initial selection of links assigned to the data. If set to c{False}, the weights from the matrix will be used directly.

matrix : list

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

link_threshold : float (default=0.5)

The threshold that shall be used for the internal clustering of the data.

matrix_type : {distances,similarities,weights} (default=distances)

Specify the type of the matrix. If the matrix contains distance data, it will be adapted to similarity data. If it contains similarities, no adaptation is needed. If it contains weights, a weighted version of link clustering (see the supplementary in `Ahn2010` for details)]will be carried out.

Returns **cluster** : dict

A dictionary with cluster-IDs as keys and a list as value, containing the taxa that are assigned to a given cluster-ID.

See also:

fuzzy

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
[[0.0, 0.5, 0.67, 0.8, 0.2],
 [0.5, 0.0, 0.4, 0.7, 0.6],
 [0.67, 0.4, 0.0, 0.8, 0.8],
 [0.8, 0.7, 0.8, 0.0, 0.3],
 [0.2, 0.6, 0.8, 0.3, 0.0]]
```

Carry out the link-clustering analysis.

```
>>> link_clustering(0.5,matrix,taxa)
{1: ['Dutch', 'English', 'German'], 2: ['Icelandic', 'Swedish']}
```

`lingpy.algorithm.clustering.`**`matrix2groups`**(*threshold*, *matrix*, *taxa*, *cluster_method='upgma'*)

Calculate flat cluster of distance matrix.

Parameters **threshold** : float

The threshold to be used for the calculation.

matrix : list

The distance matrix to be used.

taxa : list

A list of the taxa in the distance matrix.

cluster_method : {upgma, mcl, single, complete} (default=upgma)

Returns **groups** : dict

A dictionary with the taxa as keys and the group assignment as values.

Notes

This function is important for internal calculations within wordlist. It is not recommended for further use.

`lingpy.algorithm.clustering.`**`matrix2tree`**(*matrix*, *taxa*, *tree_calc='neighbor'*, *distances=True*, *filename=''*)

Calculate a tree of a given distance matrix.

Parameters **matrix** : list

The distance matrix to be used.

taxa : list

A list of the taxa in the distance matrix.

tree_calc : str (default=neighbor)

The method for tree calculation that shall be used. Select between:

- neighbor: Neighbor-joining method (`Saitou1987`)
- upgma : UPGMA method (`Sokal1958`)

distances : bool (default=True)

If set to c{True}, distances will be included in the tree-representation.

filename : str (default=)

If a filename is specified, the data will be written to that file.

Returns **tree** : ~lingpy.thirdparty.cogent.tree.PhyloNode

A ~lingpy.thirdparty.cogent.tree.PhyloNode object for handling tree files.

`lingpy.algorithm.clustering.`**`mcl`**(*threshold*, *matrix*, *taxa*, *max_steps=1000*, *inflation=2*, *expansion=2*, *add_self_loops=True*, *revert=False*, *logs=True*, *matrix_type='distances'*)

Carry out a clustering using the MCL algorithm (`Dongen2000`).

Parameters **threshold** : {float, bool}

The threshold that shall be used for the initial selection of links assigned to the data. If set to c{False}, the weights from the matrix will be used directly.

matrix : list

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

max_steps : int (default=1000)

Maximal number of iterations.

inflation : int (default=2)

Inflation parameter for the MCL algorithm.

expansion : int (default=2)

Expansion parameter of the MCL algorithm.

add_self_loops : {True, False, builtins.function} (default=True)

Determine whether self-loops should be added, and if so, how they should be weighted. If a function for the calculation of self-loops is given, it will take the whole column of the matrix for each taxon as input.

logs : { bool, function } (default=True)

If set to c{True}, the logarithm of the score beyond the threshold will be assigned as weight to the graph. If set to c{False} all weights will be set to 1. Use a custom function to define individual ways to calculate the weights.

matrix_type : { distances, similarities }

Specify the type of the matrix. If the matrix contains distance data, it will be adapted to similarity data. If it contains similarities, no adaptation is needed.

Examples

The function is automatically imported along with LingPy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create a list of arbitrary taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary distance matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
>>> matrix
[[0.0, 0.5, 0.67, 0.8, 0.2],
 [0.5, 0.0, 0.4, 0.7, 0.6],
 [0.67, 0.4, 0.0, 0.8, 0.8],
 [0.8, 0.7, 0.8, 0.0, 0.3],
 [0.2, 0.6, 0.8, 0.3, 0.0]]
```

Carry out the link-clustering analysis.

```
>>> mcl(0.5,matrix,taxa)
{1: ['German', 'English', 'Dutch'], 2: ['Swedish', 'Icelandic']}
```

`lingpy.algorithm.clustering.`**`neighbor`**(*matrix*, *taxa*, *distances=True*)

Function clusters data according to the Neighbor-Joining algorithm (`Saitou1987`).

Parameters **matrix** : list

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

distances : bool (default=True)

If set to **False**, only the topology of the tree will be returned.

Returns **newick** : str

A string in newick-format which can be further used in biological software packages to view and plot the tree.

See also:

`upgma`

Examples

Function is automatically imported when importing lingpy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create an arbitrary list of taxa.

```
>>> taxa = ['Norwegian','Swedish','Icelandic','Dutch','English']
```

Create an arbitrary matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
```

Carry out the cluster analysis.

```
>>> neighbor(matrix,taxa)
'(((Norwegian,(Swedish,Icelandic)),English),Dutch);'
```

lingpy.algorithm.clustering.**partition_density**(*matrix*, *t*)

Calculate partition density for a given threshold on a distance matrix.

Notes

See Ahn2012 for details on the calculation of partition density in a given network.

lingpy.algorithm.clustering.**upgma**(*matrix*, *taxa*, *distances=True*)

Carry out a cluster analysis based on the UPGMA algorithm (Sokal1958).

Parameters **matrix** : list

A two-dimensional list containing the distances.

taxa : list

An list containing the names of all taxa corresponding to the distances in the matrix.

distances : bool (default=True)

If set to **False**, only the topology of the tree will be returned.

Returns **newick** : str

A string in newick-format which can be further used in biological software packages to view and plot the tree.

See also:

neighbor

Examples

Function is automatically imported when importing lingpy.

```
>>> from lingpy import *
>>> from lingpy.algorithm import squareform
```

Create an arbitrary list of taxa.

```
>>> taxa = ['German','Swedish','Icelandic','English','Dutch']
```

Create an arbitrary matrix.

```
>>> matrix = squareform([0.5,0.67,0.8,0.2,0.4,0.7,0.6,0.8,0.8,0.3])
```

Carry out the cluster analysis.

```
>>> upgma(matrix,taxa,distances=False)
'((Swedish,Icelandic),(English,(German,Dutch)));'
```

lingpy.algorithm.extra module

Adapting specific cluster algorithms from scikit-learn to LingPy.

`lingpy.algorithm.extra.`**`affinity_propagation`**(*threshold*, *matrix*, *taxa*, *revert=False*)

Compute affinity propagation from the matrix.

Parameters **threshold** : float

The threshold for clustering you want to use.

matrix : list

The two-dimensional matrix passed as list or array.

taxa : list

The list of taxon names. If set to False a fake list of taxon names will be created, giving a positive numerical ID in increasing order for each column in the matrix.

revert : bool

If set to False, dont return taxon names but simply the language identifiers and their labels as a dictionary. Otherwise returns a dictionary with labels as keys and list of taxon names as values.

Returns **clusters** : dict

Either a dictionary of taxon identifiers and labels, or a dictionary of labels and taxon names.

Notes

Affinity propagation is a clustering method originally proposed by `Frey2007`.

Requires the scikitlearn package, downloadable from http://scikit-learn.org/.

`lingpy.algorithm.extra.`**`dbscan`**(*threshold*, *matrix*, *taxa*, *revert=False*, *min_samples=1*)

Compute DBSCAN cluster analysis.

Parameters **threshold** : float

The threshold for clustering you want to use.

matrix : list

The two-dimensional matrix passed as list or array.

taxa : list

The list of taxon names. If set to False a fake list of taxon names will be created, giving a positive numerical ID in increasing order for each column in the matrix.

revert : bool

If set to False, dont return taxon names but simply the language identifiers and their labels as a dictionary. Otherwise returns a dictionary with labels as keys and list of taxon names as values.

min_samples : int (default=1)

The minimal samples parameter of the DBCSCAN method from the SKLEARN package.

Returns clusters : dict

Either a dictionary of taxon identifiers and labels, or a dictionary of labels and taxon names.

Notes

This method does not work as expected, probably since it normally requires distances between points as input. We list it only for completeness here, but urge to be careful when using the code and checking properly our implementation in the source code.

Requires the scikitlearn package, downloadable from http://scikit-learn.org/.

`lingpy.algorithm.extra.`**`infomap_clustering`**(*threshold*, *matrix*, *taxa=False*, *revert=False*)

Compute the Infomap clustering analysis of the data.

Parameters threshold : float

The threshold for clustering you want to use.

matrix : list

The two-dimensional matrix passed as list or array.

taxa : list

The list of taxon names. If set to False a fake list of taxon names will be created, giving a positive numerical ID in increasing order for each column in the matrix.

revert : bool

If set to False, dont return taxon names but simply the language identifiers and their labels as a dictionary. Otherwise returns a dictionary with labels as keys and list of taxon names as values.

Returns clusters : dict

Either a dictionary of taxon identifiers and labels, or a dictionary of labels and taxon names.

Notes

Infomap clustering is a community detection method originally proposed by `Rosvall2008`. This method requires the igraph package, downloadable from http://igraph.org/.

Module contents

Package for specific algorithms and time-intensive routines.

lingpy.align package

Submodules

lingpy.align.multiple module

Module provides classes and functions for multiple alignment analyses.

class `lingpy.align.multiple.`**`Multiple`**(*seqs*, ***keywords*)
Bases: `clldutils.misc.UnicodeMixin`

Basic class for multiple sequence alignment analyses.

Parameters **seqs** : list

List of sequences that shall be aligned.

Notes

Depending on the structure of the sequences, further keywords can be specified that manage how the items get tokenized.

`align`(*method*, ***kw*)

`get_local_peaks`(*threshold=2*, *gap_weight=0.0*)
Return all peaks in a given alignment.

Parameters **threshold** : { int, float } (default=2)

The threshold to determine whether a given column is a peak or not.

gap_weight : float (default=0.0)

The weight for gaps.

`get_pairwise_alignments`(***keywords*)
Function creates a dictionary of all pairwise alignments scores.

Parameters **new_calc** : bool (default=True)

Specify, whether the analysis should be repeated from the beginning, or whether already conducted analyses should be carried out.

model : string (default=sca)

A string indicating the name of the *`Model`* object that shall be used for the analysis. Currently, three models are supported:

- dolgo – a sound-class model based on `Dolgopolsky1986`,
- sca – an extension of the dolgo sound-class model based on `List2012b`, and
- asjp – an independent sound-class model which is based on the sound-class model of `Brown2008` and the empirical data of `Brown2011` (see the description in `List2012`.

mode : string (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm `Needleman1970`,
- dialign – global alignment analysis which seeks to maximize local similarities `Morgenstern1996`.

gop : int (default=-3)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.6)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1982`.

factor : float (default=1)

The factor by which the initial and the descending position shall be modified.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

restricted_chars : string (default=T)

Define which characters of the prosodic string of a sequence reflect its secondary structure (cf. `List2012b`) and should therefore be aligned specifically. This defaults to T, since this is the character that represents tones in the prosodic strings of sequences.

`get_peaks`(*gap_weight=0*)

Calculate the profile score for each column of the alignment.

Parameters **gap_weight** : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

Returns **peaks** : list

A list containing the profile scores for each column of the given alignment.

`get_pid`(*mode=1*)

Return the Percentage Identity (PID) score of the calculated MSA.

Parameters **mode** : { 1, 2, 3, 4, 5 } (default=1)

Indicate which of the four possible PID scores described in `Raghava2006` should be calculated, the fifth possibility is added for linguistic purposes:

1. identical positions / (aligned positions + internal gap positions),
2. identical positions / aligned positions,
3. identical positions / shortest sequence, or
4. identical positions / shortest sequence (including internal gap pos.)
5. identical positions / (aligned positions + 2 * number of gaps)

Returns **score** : float

The PID score of the given alignment as a floating point number between 0 and 1.

See also:

lingpy.sequence.sound_classes.pid

iterate_all_sequences(*check='final'*, *mode='global'*, *gop=-3*, *scale=0.5*, *factor=0*, *gap_weight=1*, *restricted_chars='T_'*)

Iterative refinement based on a complete realignment of all sequences.

Parameters **check** : { final, immediate } (default=final)

Specify when to check for improved sum-of-pairs scores: After each iteration (immediate) or after all iterations have been carried out (final).

mode : { global, overlap, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm Needleman1970,
- dialign – global alignment analysis which seeks to maximize local similarities Morgenstern1996.
- overlap – semi-global alignment, where gaps introduced in the beginning and the end of a sequence do not score.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.5)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties Gotoh1981.

factor : float (default=0.3)

The factor by which the initial and the descending position shall be modified.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

See also:

Multiple.iterate_clusters, *Multiple.iterate_similar_gap_sites*, *Multiple.iterate_orphans*

Notes

This method essentially follows the iterative method of Barton1987 with the exception that an MSA has already been calculated.

iterate_clusters(*threshold*, *check='final'*, *mode='global'*, *gop=-3*, *scale=0.5*, *factor=0*, *gap_weight=1*, *restricted_chars='T_'*)

Iterative refinement based on a flat cluster analysis of the data.

Parameters **threshold** : float

The threshold for the flat cluster analysis.

check : string (default=final)

Specify when to check for improved sum-of-pairs scores: After each iteration (immediate) or after all iterations have been carried out (final).

mode : { global, overlap, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm `Needleman1970`,
- dialign – global alignment analysis which seeks to maximize local similarities `Morgenstern1996`.
- overlap – semi-global alignment, where gaps introduced in the beginning and the end of a sequence do not score.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.6)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1981`.

factor : float (default=0.3)

The factor by which the initial and the descending position shall be modified.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

See also:

`Multiple.iterate_similar_gap_sites`, *`Multiple.iterate_all_sequences`*

Notes

This method uses the *`lingpy.algorithm.clustering.flat_upgma()`* function in order to retrieve a flat cluster of the data.

`iterate_orphans`(*check='final'*, *mode='global'*, *gop=-3*, *scale=0.5*, *factor=0*, *gap_weight=1.0*, *restricted_chars='T_'*)

Iterate over the most divergent sequences in the sample.

Parameters **check** : string (default=final)

Specify when to check for improved sum-of-pairs scores: After each iteration (immediate) or after all iterations have been carried out (final).

mode : { global, overlap, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm `Needleman1970`,
- dialign – global alignment analysis which seeks to maximize local similarities `Morgenstern1996`.
- overlap – semi-global alignment, where gaps introduced in the beginning and the end of a sequence do not score.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.6)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1981`.

factor : float (default=0.3)

The factor by which the initial and the descending position shall be modified.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

See also:

`Multiple.iterate_clusters`, *`Multiple.iterate_similar_gap_sites`*, *`Multiple.iterate_all_sequences`*

Notes

The most divergent sequences are those whose average distance to all other sequences is above the average distance of all sequence pairs.

`iterate_similar_gap_sites`(*check='final'*, *mode='global'*, *gop=-3*, *scale=0.5*, *factor=0*, *gap_weight=1*, *restricted_chars='T_'*)

Iterative refinement based on the *Similar Gap Sites* heuristic.

Parameters **check** : { final, immediate } (default=final)

Specify when to check for improved sum-of-pairs scores: After each iteration (immediate) or after all iterations have been carried out (final).

mode : { global, overlap, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm `Needleman1970`,
- dialign – global alignment analysis which seeks to maximize local similarities `Morgenstern1996`.
- overlap – semi-global alignment, where gaps introduced in the beginning and the end of a sequence do not score.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.5)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties Gotoh1982.

factor : float (default=0.3)

The factor by which the initial and the descending position shall be modified.

gap_weight : float (default=1)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When, e.g., set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

See also:

Multiple.iterate_clusters, *Multiple.iterate_all_sequences*, *Multiple.iterate_orphans*

Notes

This heuristic is fairly simple. The idea is to try to split a given MSA into partitions with identical gap sites.

lib_align(***keywords*)

Carry out a library-based progressive alignment analysis of the sequences.

Parameters **model** : { dolgo, sca, asjp } (default=sca)

A string indicating the name of the *Model* object that shall be used for the analysis. Currently, three models are supported:

- dolgo – a sound-class model based on Dolgopolsky1986,
- sca – an extension of the dolgo sound-class model based on List2012b, and
- asjp – an independent sound-class model which is based on the sound-class model of Brown2008 and the empirical data of Brown2011 (see the description in List2012.

mode : { global, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm Needleman1970,
- dialign – global alignment analysis which seeks to maximize local similarities Morgenstern1996.

modes : list (default=[(global,-10,0.6),(local,-1,0.6)])

Indicate the mode, the gap opening penalties (GOP), and the gap extension scale (GEP scale), of the pairwise alignment analyses which are used to create the library.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

gep_scale : float (default=0.6)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1982`.

factor : float (default=1)

The factor by which the initial and the descending position shall be modified.

tree_calc : { neighbor, upgma } (default=upgma)

The cluster algorithm which shall be used for the calculation of the guide tree. Select between `neighbor`, the Neighbor-Joining algorithm (`Saitou1987`), and `upgma`, the UPGMA algorithm (`Sokal1958`).

guide_tree : tree_matrix

Use a custom guide tree instead of performing a cluster algorithm for constructing one based on the input similarities. The use of this option makes the tree_calc option irrelevant.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

restricted_chars : string (default=T)

Define which characters of the prosodic string of a sequence reflect its secondary structure (cf. `List2012b`) and should therefore be aligned specifically. This defaults to T, since this is the character that represents tones in the prosodic strings of sequences.

Notes

In contrast to traditional progressive multiple sequence alignment approaches such as `Feng1981` and `Thompson1994`, library-based progressive alignment `Notredame2000` is based on a pre-processing of the data where the information given in global and local pairwise alignments of the input sequences is used to derive a refined scoring function (*library*) which is later used in the progressive phase.

`prog_align`(***keywords*)

Carry out a progressive alignment analysis of the input sequences.

Parameters **model** : { dolgo, sca, asjp } (defaul=sca)

A string indicating the name of the *`Model`* object that shall be used for the analysis. Currently, three models are supported:

- dolgo – a sound-class model based on `Dolgopolsky1986`,
- sca – an extension of the dolgo sound-class model based on `List2012b`, and
- asjp – an independent sound-class model which is based on the sound-class model of `Brown2008` and the empirical data of `Brown2011` (see the description in `List2012`.

mode : { global, dialign } (default=global)

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm `Needleman1970`,

- dialign – global alignment analysis which seeks to maximize local similarities `Morgenstern1996`.

gop : int (default=-2)

The gap opening penalty (GOP) used in the analysis.

scale : float (default=0.5)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1982`.

factor : float (default=0.3)

The factor by which the initial and the descending position shall be modified.

tree_calc : { neighbor, upgma } (default=upgma)

The cluster algorithm which shall be used for the calculation of the guide tree. Select between `neighbor`, the Neighbor-Joining algorithm (`Saitou1987`), and `upgma`, the UPGMA algorithm (`Sokal1958`).

guide_tree : tree_matrix

Use a custom guide tree instead of performing a cluster algorithm for constructing one based on the input similarities. The use of this option makes the tree_calc option irrelevant.

gap_weight : float (default=0.5)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

restricted_chars : string (default=T)

Define which characters of the prosodic string of a sequence reflect its secondary structure (cf. `List2012b`) and should therefore be aligned specifically. This defaults to T, since this is the character that represents tones in the prosodic strings of sequences.

`sum_of_pairs`(*alm_matrix='self'*, *mat=None*, *gap_weight=0.0*, *gop=-1*)
Calculate the sum-of-pairs score for a given alignment analysis.

Parameters **alm_matrix** : { self, other } (default=self)

Indicate for which MSA the sum-of-pairs score shall be calculated.

mat : { None, list }

If other is chosen as an option for **alm_matrix**, define for which matrix the sum-of-pairs score shall be calculated.

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

Returns **The sum-of-pairs score of the alignment.** :

`swap_check`(*swap_penalty=-3*, *score_mode='classes'*)
Check for possibly swapped sites in the alignment.

Parameters **swap_penalty** : { int, float } (default=-3)

Specify the penalty for swaps in the alignment.

score_mode : { classes, library } (default=classes)

Define the score-mode of the calculation which is either based on sound classes proper, or on the specific scores derived from the library approach.

Returns **result** : bool

Returns `True`, if a swap was identified, and `False` otherwise. The information regarding the position of the swap is stored in the attribute `swap_index`.

Notes

The method for swap detection is described in detail in `List2012b`.

Examples

Define a set of strings whose alignment contans a swap.

```
>>> from lingpy import *
>>> mult = Multiple(["woldemort", "waldemar", "wladimir"])
```

Align the data, using the progressive approach.

```
>>> mult.prog_align()
```

Check for swaps.

```
>>> mult.swap_check()
True
```

Print the alignment

```
>>> print(mult)
w   o   l   -   d   e   m   o   r   t
w   a   l   -   d   e   m   a   r   -
v   -   l   a   d   i   m   i   r   -
```

`lingpy.align.multiple.`**`dotjoin`**(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.align.multiple.`**`mult_align`**(*seqs*, *gop=-1*, *scale=0.5*, *tree_calc='upgma'*, *scoredict=False*, *pprint=False*)

A short-cut method for multiple alignment analyses.

Parameters **seqs** : list

The input sequences.

gop = int (default=-1) :

The gap opening penalty.

scale : float (default=0.5)

The scaling factor by which penalties for gap extensions are decreased.

tree_calc : { upgma neighbor } (default=upgma)

The algorithm which is used for the calculation of the guide tree.

pprint : bool (default=False)

Indicate whether results shall be printed onto screen.

Returns alignments : list

A two-dimensional list in which alignments are represented as a list of tokens.

Examples

```
>>> m = mult_align(["woldemort", "waldemar", "vladimir"], pprint=True)
w    o    l    -    d    e    m    o    r    t
w    a    l    -    d    e    m    a    r    -
-    v    l    a    d    i    m    i    r    -
```

lingpy.align.pairwise module

Module provides classes and functions for pairwise alignment analyses.

class lingpy.align.pairwise.**Pairwise**(*seqs*, *seqB=False*, ***keywords*)
Bases: object

Basic class for the handling of pairwise sequence alignments (PSA).

Parameters seqs : string list

Either the first string of a sequence pair that shall be aligned, or a list of sequence tuples.

seqB : string or bool (default=None)

Define the second sequence that shall be aligned with the first sequence, if only two sequences shall be compared.

align(***keywords*)
Align a pair of sequences or multiple sequence pairs.

Parameters gop : int (default=-1)

The gap opening penalty (GOP).

scale : float (default=0.5)

The gap extension penalty (GEP), calculated with help of a scaling factor.

mode : {global,local,overlap,dialign}

The alignment mode, see List2012a for details.

factor : float (default = 0.3)

The factor by which matches in identical prosodic position are increased.

restricted_chars : str (default=T_)

The restricted chars that function as an indicator of syllable or morpheme breaks for secondary alignment, see List2012c for details.

distance : bool (default=False)

If set to *True*, return the distance instead of the similarity score. Distance is calculated using the formula by `Downey2008`.

model : { None, ~lingpy.data.model.Model }

Specify the sound class model that shall be used for the analysis. If no model is specified, the default model of `List2012a` will be used.

pprint : bool (default=False)

If set to *True*, the alignments are printed to the screen.

`lingpy.align.pairwise.`**`edit_dist`**(*seqA*, *seqB*, *normalized=False*, *restriction=”*)

Return the edit distance between two strings.

Parameters **seqA,seqB** : str

The strings that shall be compared.

normalized : bool (default=False)

Specify whether the normalized edit distance shall be returned. If no restrictions are chosen, the edit distance is normalized by dividing by the length of the longer string. If *restriction* is set to *cv* (consonant-vowel), the edit distance is normalized by the length of the alignment.

restriction : {cv} (default=)

Specify the restrictions to be used. Currently, only `cv` is supported. This prohibits matches of vowels with consonants.

Returns **dist** : {int float}

The edit distance, which is a float if normalized is set to c{True}, and an integer otherwise.

Notes

The edit distance was first formally defined by V. I. Levenshtein (`Levenshtein1965`). The first algorithm to compute the edit distance was proposed by Wagner and Fisher (`Wagner1974`).

Examples

Align two sequences::

```
>>> seqA = 'fat cat'
>>> seqB = 'catfat'
>>> edit_dist(seqA, seqB)
3
```

`lingpy.align.pairwise.`**`nw_align`**(*seqA*, *seqB*, *scorer=False*, *gap=-1*)

Carry out the traditional Needleman-Wunsch algorithm.

Parameters **seqA, seqB** : {str, list, tuple}

The input strings. These should be iterables, so you can use tuples, lists, or strings.

scorer [dict (default=False)] If set to c{False} a scorer will automatically be calculated, otherwise, the scorer needs to be passed as a dictionary that covers all segment matches between the input strings (segment matches need to be passed as tuples of two segments, following the order of the input sequences). Note also that the scorer can well be asymmetric, so you could also use it for two completely different alphabets. All you need to make sure is that the tuples representing the segment matches follow the order of your input sequences.

gap [int (default=-1)] The gap penalty.

Returns alm : tuple

A tuple consisting of the aligments of the first and the second sequence, and the alignment score.

Notes

The Needleman-Wunsch algorithm (see `Needleman1970`) returns a global alignment of two sequences.

+ .join(almB), (sim={0}).format(sim))

a b a b - - b a b a (sim=1)

Nothing unexpected so far, you could reach the same result without the scorer. But now lets make a scorer that favors mismatches for our little two-letter alphabet:

```
>>> scorer = { ('a','b'): 1, ('a','a'):-1, ('b','b'):-1, ('b', 'a'): 1}
>>> seqA, seqB = 'abab', 'baba'
>>> almA, almB, sim = nw_align(seqA, seqB, scorer=scorer)
>>> print(' '.join(almA)+'
```

+ .join(almB), (sim={0}).format(sim))

a b a b b a b a (sim=4)

Now, lets analyse two strings which are completely different, but where we use the scorer to define mappings between the segments. We simply do this by using lower case letters in one and upper case letters in the other case, which will, of course, be treated as different symbols in Python:

```
>>> scorer = { ('A','a'): 1, ('A','b'):-1, ('B','a'):-1, ('B', 'B'): 1}
>>> seqA, seqB = 'ABAB', 'aa'
>>> almA, almB, sim = nw_align(seqA, seqB, scorer=scorer)
>>> print(' '.join(almA)+'
```

+ .join(almB), (sim={0}).format(sim)) A B A B a - a - (sim=0)

`lingpy.align.pairwise.`**`pw_align`**(*seqA*, *seqB*, *gop=-1*, *scale=0.5*, *scorer=False*, *mode='global'*, *distance=False*, ***keywords*)

Align two sequences in various ways.

Parameters seqA, seqB : {text_type, list, tuple}

The input strings. These should be iterables, so you can use tuples, lists, or strings.

scorer : dict (default=False)

If set to c{False} a scorer will automatically be calculated, otherwise, the scorer needs to be passed as a dictionary that covers all segment matches between the input strings.

gop : int (default=-1)

The gap opening penalty.

scale : float (default=0.5)

The gap extension scale. This scale is similar to the gap extension penalty, but in contrast to the traditional GEP, it scales the gap opening penalty.

mode : {global, local, dialign, overlap} (default=global)

Select between one of the four different alignment modes regularly implemented in LingPy, see `List2012a` for details.

distance : bool (default=False)

If set to c{True} return the distance score following the formula by `Downey2008`. Otherwise, return the basic similarity score.

Examples

Align two words using the dialign algorithm::

```
>>> seqA = 'fat cat'
>>> seqB = 'catfat'
>>> pw_align(seqA, seqB, mode='dialign')
(['f', 'a', 't', ' ', 'c', 'a', 't', '-', '-', '-'],
 ['-', '-', '-', '-', 'c', 'a', 't', 'f', 'a', 't'],
 3.0)
```

`lingpy.align.pairwise.`**`structalign`**(*seqA*, *seqB*)

Experimental function for testing structural alignment algorithms.

`lingpy.align.pairwise.`**`sw_align`**(*seqA*, *seqB*, *scorer=False*, *gap=-1*)

Carry out the traditional Smith-Waterman algorithm.

Parameters **seqA, seqB** : {str, list, tuple}

The input strings. These should be iterables, so you can use tuples, lists, or strings.

scorer : dict (default=False)

If set to c{False} a scorer will automatically be calculated, otherwise, the scorer needs to be passed as a dictionary that covers all segment matches between the input strings.

gap : int (default=-1)

The gap penalty.

Returns **alm** : tuple

A tuple consisting of prefix, alignment, and suffix of the first and the second sequence, and the alignment score.

Notes

The Smith-Waterman algorithm (see `Smith1981`) returns a local alignment between two sequences. A local alignment is an alignment of those subsequences of the input sequences that yields the highest score.

Examples

Align two sequences::

```
>>> seqA = 'fat cat'
>>> seqB = 'catfat'
>>> sw_align(seqA, seqB)
(([], ['f', 'a', 't'], [' ', 'c', 'a', 't']),
 (['c', 'a', 't'], ['f', 'a', 't'], []),
 3.0)
```

`lingpy.align.pairwise.`**`turchin`**(*seqA*, *seqB*, *model='dolgo'*, ***keywords*)

Return cognate judgment based on the method by `Turchin2010`.

Parameters **seqA, seqB** : {str, list, tuple}

The input strings. These should be iterables, so you can use tuples, lists, or strings.

model : {asjp, sca, dolgo} (default=dolgo)

A sound-class model instance or a string that denotes one of the standard sound class models used in LingPy.

Returns **cognacy** : {0, 1}

The cognacy assertion which is either 0 (words are probably cognate) or 1 (words are not likely to be cognate).

`lingpy.align.pairwise.`**`we_align`**(*seqA*, *seqB*, *scorer=False*, *gap=-1*)

Carry out the traditional Waterman-Eggert algorithm.

Parameters **seqA, seqB** : {str, list, tuple}

The input strings. These should be iterables, so you can use tuples, lists, or strings.

scorer : dict (default=False)

If set to c{False} a scorer will automatically be calculated, otherwise, the scorer needs to be passed as a dictionary that covers all segment matches between the input strings.

gap : int (default=-1)

The gap penalty.

Returns **alms** : list

A list consisting of tuples. Each tuple gives the alignment of one of the subsequences of the input sequences. Each tuple contains the aligned part of the first, the aligned part of the second sequence, and the score of the alignment.

Notes

The Waterman-Eggert algorithm (see `Waterman1987`) returns *all* local matches between two sequences.

Examples

Align two sequences::

```
>>> seqA = 'fat cat'
>>> seqB = 'catfat'
>>> we_align(seqA, seqB)
[(['f', 'a', 't'], ['f', 'a', 't'], 3.0),
 (['c', 'a', 't'], ['c', 'a', 't'], 3.0)]
```

lingpy.align.sca module

Basic module for pairwise and multiple sequence comparison.

The module consists of four classes which deal with pairwise and multiple sequence comparison from the *sequence* and the *alignment* perspective. The sequence perspective deals with unaligned sequences. The *alignment* perspective deals with aligned sequences.

class lingpy.align.sca.**Alignments**(*infile*, *row='concept'*, *col='doculect'*, *conf="*, *modify_ref=False*, *_interactive=True*, *split_on_tones=True*, *ref='cogid'*, ***keywords*)

Bases: `lingpy.basic.wordlist.Wordlist`

Class handles Wordlists for the purpose of alignment analyses.

Parameters infile : str

The name of the input file that should conform to the basic format of the *~lingpy.basic.wordlist.Wordlist* class and define a specific ID for cognate sets.

row : str (default = concept)

A string indicating the name of the row that shall be taken as the basis for the tabular representation of the word list.

col : str (default = doculect)

A string indicating the name of the column that shall be taken as the basis for the tabular representation of the word list.

conf : string (default=)

A string defining the path to the configuration file.

ref : string (default=cogid)

The name of the column that stores the cognate IDs.

modify_ref : function (default=False)

Use a function to modify the reference. If your cognate identifiers are numerical, for example, and negative values are assigned as loans, but you want to suppress this behaviour, just set this keyword to abs, and all cognate IDs will be converted to their absolute value.

split_on_tones : bool (default=True)

If set to True, this means that in the case of fuzzy alignment mode, the algorithm will attempt to split words into morphemes by tones if no explicit morpheme markers can be found.

Notes

This class inherits from *Wordlist* and additionally creates instances of the *Multiple* class for all cognate sets that are specified by the *ref* keyword.

Attributes

msa	dict	A dictionary storing multiple alignments as dictionaries which can be directly opened and aligned with help of the ~lingpy.align.sca.SCA function. The alignment objects are referenced by a key which is identical with the reference (ref-keyword) of the alignment, that is the name of the column which contains the cognate identifiers.

add_alignments(*ref=False*, *modify_ref=False*, *fuzzy=False*, *split_on_tones=True*, *override=False*)
Function adds a new set of alignments to the data.

Parameters **ref: str (default=False)** :

Use this to set the name of the column which contains the cognate sets.

fuzzy: bool (default=False) :

If set to true, force the algorithm to treat the cognate sets as fuzzy cognate sets, i.e., as multiple cognate sets which are in order assigned to a word (proper partial cognates).

align(***keywords*)
Carry out a multiple alignment analysis of the data.

Parameters **method** : { progressive, library } (default=progressive)

Select the method to use for the analysis.

iteration : bool (default=False)

Set to c{True} in order to use iterative refinement methods.

swap_check : bool (default=False)

Set to c{True} in order to carry out a swap-check.

model : { dolgo, sca, asjp }

A string indicating the name of the *Model* object that shall be used for the analysis. Currently, three models are supported:

- dolgo – a sound-class model based on Dolgopolsky1986,
- sca – an extension of the dolgo sound-class model based on List2012b, and
- asjp – an independent sound-class model which is based on the sound-class model of Brown2008 and the empirical data of Brown2011 (see the description in List2012.

mode : { global, dialign }

A string indicating which kind of alignment analysis should be carried out during the progressive phase. Select between:

- global – traditional global alignment analysis based on the Needleman-Wunsch algorithm Needleman1970,
- dialign – global alignment analysis which seeks to maximize local similarities Morgenstern1996.

modes : list (default=[(global,-2,0.5),(local,-1,0.5)])

Indicate the mode, the gap opening penalties (GOP), and the gap extension scale (GEP scale), of the pairwise alignment analyses which are used to create the library.

gop : int (default=-5)

The gap opening penalty (GOP) used in the analysis.

scale : float (default=0.6)

The factor by which the penalty for the extension of gaps (gap extension penalty, GEP) shall be decreased. This approach is essentially inspired by the exension of the basic alignment algorithm for affine gap penalties `Gotoh1982`.

factor : float (default=1)

The factor by which the initial and the descending position shall be modified.

tree_calc : { neighbor, upgma } (default=upgma)

The cluster algorithm which shall be used for the calculation of the guide tree. Select between `neighbor`, the Neighbor-Joining algorithm (`Saitou1987`), and `upgma`, the UPGMA algorithm (`Sokal1958`).

gap_weight : float (default=0)

The factor by which gaps in aligned columns contribute to the calculation of the column score. When set to 0, gaps will be ignored in the calculation. When set to 0.5, gaps will count half as much as other characters.

restricted_chars : string (default=T)

Define which characters of the prosodic string of a sequence reflect its secondary structure (cf. `List2012b`) and should therefore be aligned specifically. This defaults to T, since this is the character that represents tones in the prosodic strings of sequences.

`get_confidence`(*scorer*, *ref='lexstatid'*, *gap_weight=0.25*)

Function creates confidence scores for a given set of alignments.

Parameters scorer : `ScoreDict`

A *ScoreDict* object which gives similarity scores for all segments in the alignment.

ref : str (default=lexstatid)

The reference entry-type, referring to the cognate-set to be used for the analysis.

gap_weight : {loat} (default=1.0)

Determine the weight assigned to matches containing gaps.

`get_consensus`(*tree=False*, *gaps=False*, *classes=False*, *consensus='consensus'*, *counterpart='ipa'*, *weights=[]*, *return_data=False*, ***keywords*)

Calculate a consensus string of all MSAs in the wordlist.

Parameters msa : {c{list} ~lingpy.align.multiple.Multiple}

Either an MSA object or an MSA matrix.

tree : {c{str} ~lingpy.thirdparty.cogent.PhyloNode}

A tree object or a Newick string along which the consensus shall be calculated.

gaps : c{bool} (default=False)

If set to c{True}, return the gap positions in the consensus.

classes : c{bool} (default=False)

Specify whether sound classes shall be used to calculate the consensus.

model : ~lingpy.data.model.Model

A sound class model according to which the IPA strings shall be converted to sound-class strings.

return_data : c{bool} (default=False)

Return the data instead of adding it in a column to the wordlist object.

`get_msa` (*ref*)

`output` (*fileformat*, ***keywords*)
Write wordlist to file.

Parameters **fileformat** : {tsv, msa, tre, nwk, dst, taxa, starling, paps.nex,

paps.csv html} The format that is written to file. This corresponds to the file extension, thus tsv creates a file in tsv-format, dst creates a file in Phylip-distance format, etc. Specific output is created for the formats html and msa:

- msa will create a folder containing all alignments of all cognate sets in msa-format
- html will create html-output in which words are sorted according to meaning, cognate set, and all cognate words are aligned

filename : str

Specify the name of the output file (defaults to a filename that indicates the creation date).

subset : bool (default=False)

If set to c{True}, return only a subset of the data. Which subset is specified in the keywords cols and rows.

cols : list

If *subset* is set to c{True}, specify the columns that shall be written to the csv-file.

rows : dict

If *subset* is set to c{True}, use a dictionary consisting of keys that specify a column and values that give a Python-statement in raw text, such as, e.g., == hand. The content of the specified column will then be checked against statement passed in the dictionary, and if it is evaluated to c{True}, the respective row will be written to file.

ref : str

Name of the column that contains the cognate IDs if starling is chosen as an output format.

missing : { str, int } (default=0)

If paps.nex or paps.csv is chosen as fileformat, this character will be inserted as an indicator of missing data.

tree_calc : {neighbor, upgma}

If no tree has been calculated and tre or nwk is chosen as output format, the method that is used to calculate the tree.

threshold : float (default=0.6)

The threshold that is used to carry out a flat cluster analysis if groups or cluster is chosen as output format.

style : str (default=id)

If msa is chosen as output format, this will write the alignments for each msa-file in a specific format in which the first column contains a direct reference to the word via its ID in the wordlist.

ignore : { list, all }

Modifies the output format in tsv output and allows to ignore certain blocks in extended tsv, like msa, taxa, json, etc., which should be passed as a list. If you choose all as a plain string and not a list, this will ignore all additional blocks and output only plain tsv.

prettify : bool (default=True)

Inserts comment characters between concepts in the tsv file output format, which makes it easier to see blocks of words denoting the same concept. Switching this off will output the file in plain tsv.

`reduce_alignments`(*alignment=False*, *ref=False*)
Function reduces alignments which contain columns that are marked to be ignored by the user.

Notes

This function changes the data only internally: All alignments are checked as to whether they contain data that should be ignored. If this is the case, the alignments are then reduced, and stored in a specific item of the alignment string. If the method doesnt find any instances for reduction, it still makes the copies of the alignments in order to guarantee that the alignments with with we want to work are at the same place in the dictionary.

`class` `lingpy.align.sca.`**`MSA`**(*infile*, ***keywords*)
Bases: *`lingpy.align.multiple.Multiple`*

Basic class for carrying out multiple sequence alignment analyses.

Parameters **infile** : file

A file in `msq`-format or `msa`-format.

merge_vowels : bool (default=True)

Indicate, whether neighboring vowels should be merged into diphtongs, or whether they should be kept separated during the analysis.

comment : char (default=#)

The comment character which, inserted in the beginning of a line, prevents that line from being read.

normalize : bool (default=True)

Normalize the alignment, that is, add gap characters for all sequences which are shorter than the longest sequence, and delete all columns from the alignment in which only gaps occur.

Notes

There are two possible input formats for this class: the MSQ-format, and the MSA-format (see msa_formats for details). This class directly inherits all methods of the *`Multiple`* class.

Examples

Get the path to a file from the testset.

```
>>> from lingpy import *
>>> path = rc("test_path")+'harry.msq'
```

Load the file into the Multiple class.

```
>>> mult = Multiple(path)
```

Carry out a progressive alignment analysis of the sequences.

```
>>> mult.prog_align()
```

Print the result to the screen:

```
>>> print(mult)
w    o    l    -    d    e    m    o    r    t
w    a    l    -    d    e    m    a    r    -
v    -    l    a    d    i    m    i    r    -
```

ipa2cls (***keywords*)

Retrieve sound-class strings from aligned IPA sequences.

Parameters **model** : str (default=sca)

The sound-class model according to which the sequences shall be converted.

Notes

This function is only useful when an `msa`-file with already conducted alignment analyses was loaded.

output (*fileformat='msa', filename=None, sorted_seqs=False, unique_seqs=False, **keywords*)

Write data to file.

Parameters **fileformat** : { psa, msa, msq }

Indicate which data should be written to file. Select between:

- psa – output of all pairwise alignments in `psa`-format,
- msa – output of the multiple alignment in `msa`-format, or
- msq – output of the multiple sequences in `msq`-format.
- html – output of the multiple alignment in `html`-format.

filename : str

Select a specific name for the outfile, otherwise, the name of the infile will be taken by default.

sorted_seqs : bool

Indicate whether the sequences should be sorted or not (applys only to msa and msq output.

unique_seqs : bool

Indicate whether only unique sequences should be written to file or not.

class lingpy.align.sca.**PSA**(*infile*, ***keywords*)
Bases: *lingpy.align.pairwise.Pairwise*

Basic class for dealing with the pairwise alignment of sequences.

Parameters **infile** : file

A file in psq-format.

merge_vowels : bool (default=True)

Indicate, whether neighboring vowels should be merged into diphtongs, or whether they should be kept separated during the analysis.

comment : char (default=#)

The comment character which, inserted in the beginning of a line, prevents that line from being read.

Notes

In order to read in data from text files, two different file formats can be used along with this class: the PSQ-format, and the PSA-format (see psa_formats for details). This class inherits the methods of the *Pairwise* class.

Attributes

taxa	list	A list of tuples containing the taxa of all sequence pairs.
seqs	list	A list of tuples containing all sequence pairs.
tokens	list	A list of tuples containing all sequence pairs in a tokenized form.

output(*fileformat='psa'*, *filename=None*, ***keywords*)
Write the results of the analyses to a text file.

Parameters **fileformat** : { psa, psq }

Indicate which data should be written to file. Select between:

- psa – output of all pairwise alignments in psa-format,
- psq – output of the multiple sequences in psq-format.

filename : str

Select a specific name for the outfile, otherwise, the name of the infile will be taken by default.

lingpy.align.sca.**SCA**(*infile*, ***keywords*)
Method returns alignment objects depending on input file or input data.

Notes

This method checks for the type of an alignment object and returns an alignment object of the respective type.

lingpy.align.sca.**get_consensus**(*msa*, *gaps=False*, *taxa=False*, *classes=False*, ***keywords*)
Calculate a consensus string of a given MSA.

Parameters **msa** : {c{list} ~lingpy.align.multiple.Multiple}

Either an MSA object or an MSA matrix.

gaps : c{bool} (default=False)

If set to c{True}, return the gap positions in the consensus.

taxa : {c{list} bool} (default=False)

If *tree* is chosen as a parameter, specify the taxa in order of the aligned strings.

classes : c{bool} (default=False)

Specify whether sound classes shall be used to calculate the consensus.

model : ~lingpy.data.model.Model

A sound class model according to which the IPA strings shall be converted to sound-class strings.

local : { c{bool}, peaks, gaps }(default=False)

Specify whether local pre-processing should be applied to the data. If set to c{peaks}, the average alignment score of each column is taken as reference to remove low-scoring columns from the alignment. If set to gaps, the columns with the highest proportion of gaps will be excluded.

Returns cons : c{str}

A consensus string of the given MSA.

Module contents

Package provides basic modules for alignment analyses.

lingpy.basic package

Submodules

lingpy.basic.ops module

Module provides basic operations on Wordlist-Objects.

`lingpy.basic.ops.`**`calculate_data`**(*wordlist*, *data*, *taxa='taxa'*, *concepts='concepts'*, *ref='cogid'*, ***keywords*)

Manipulate a wordlist object by adding different kinds of data.

Parameters data : str

The type of data that shall be calculated. Currently supports

- tree: calculate a reference tree based on shared cognates
- dst: get distances between taxa based on shared cognates
- cluster: cluster the taxa into groups using different methods

`lingpy.basic.ops.`**`clean_taxnames`**(*wordlist*, *column='doculect'*, *f=<function <lambda>>*)

Function cleans taxon names for use in Newick files.

`lingpy.basic.ops.`**`coverage`**(*wordlist*)

Determine the average coverage of a wordlist.

`lingpy.basic.ops.`**`get_score`**(*wl*, *ref*, *mode*, *taxA*, *taxB*, *concepts_attr='concepts'*, *ignore_missing=False*)

`lingpy.basic.ops.`**`iter_rows`**(*wordlist*, **values*)

Function generates a list of the specified values in a wordlist.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

A wordlist object or one of the daughter classes of wordlists.

value : str

A value as defined in the header of the wordlist.

Returns **list** : list

A generator object that generates list containing the key of each row in the wordlist and the corresponding cells, as specified in the headers.

Notes

Use this function to quickly iterate over specified fields in the wordlist. For example, when trying to access all pairs of language names and concepts, you may write:

```
>>> for k, language, concept in iter_rows(wl, 'language', 'concept'):
        print(k, language, concept)
```

Note that this function returns the key of the given row as a first value. So if you do not specify anything, the output will just be the key.

`lingpy.basic.ops.`**`renumber`**(*wordlist*, *source*, *target=''*, *override=False*)

Create numerical identifiers from string identifiers.

`lingpy.basic.ops.`**`triple2tsv`**(*triples_or_fname*, *output='table'*)

Function reads a triple file and converts it to a tabular data structure.

`lingpy.basic.ops.`**`tsv2triple`**(*wordlist*, *outfile=None*)

Function converts a wordlist to a triple data structure.

Notes

The basic values of which the triples consist are:

- ID (the ID in the TSV file)
- COLUMN (the column in the TSV file)
- VALUE (the entry in the TSV file)

`lingpy.basic.ops.`**`wl2dict`**(*wordlist*, *sections*, *entries*, *exclude=None*)

Convert a wordlist to a complex dictionary with headings as keys.

`lingpy.basic.ops.`**`wl2dst`**(*wl*, *taxa='taxa'*, *concepts='concepts'*, *ref='cogid'*, *refB=''*, *mode='swadesh'*, *ignore_missing=False*, ***keywords*)

Function converts wordlist to distance matrix.

`lingpy.basic.ops.`**`wl2multistate`**(*wordlist*, *ref*, *missing*)

Function converts a wordlist to multistate format (compatible with PAUP).

`lingpy.basic.ops.`**`wl2qlc`**(*header*, *data*, *filename=''*, *formatter='concept'*, ***keywords*)

Write the basic data of a wordlist to file.

lingpy.basic.parser module

Basic parser for text files in QLC format.

class lingpy.basic.parser.**QLCParser**(*filename*, *conf=''*)

Bases: object

Basic class for the handling of text files in QLC format.

add_entries(*entry*, *source*, *function*, *override=False*, ***keywords*)

Add new entry-types to the word list by modifying given ones.

Parameters **entry** : string

A string specifying the name of the new entry-type to be added to the word list.

source : string

A string specifying the basic entry-type that shall be modified. If multiple entry-types shall be used to create a new entry, they should be passed in a simple string separated by a comma.

function : function

A function which is used to convert the source into the target value.

keywords : {dict}

A dictionary of keywords that are passed as parameters to the function.

Notes

This method can be used to add new entry-types to the data by converting given ones. There are a lot of possibilities for adding new entries, but the most basic procedure is to use an existing entry-type and to modify it with help of a function.

pickle(*filename=None*)

Store the QLCParser instance in a pickle file.

Notes

The function stores a binary file called `FILENAME.pkl` with `FILENAME` corresponding to the name of the original file in the user cache dir for lingpy on your system. To restore the instance from the pickle call *unpickle()*.

static unpickle(*filename*)

class lingpy.basic.parser.**QLCParserWithRowsAndCols**(*filename*, *row*, *col*, *conf*)

Bases: *lingpy.basic.parser.QLCParser*

get_entries(*entry*)

Return all entries matching the given entry-type as a two-dimensional list.

Parameters **entry** : string

The entry-type of the data that shall be returned in tabular format.

lingpy.basic.parser.**confirm**(*question*, ***, *default=False*)

Ask a yes/no question interactively.

Parameters **question** – The text of the question to ask.

Returns True if the answer was yes, False otherwise.

lingpy.basic.tree module

Basic module for the handling of language trees.

class lingpy.basic.tree.**Tree**(*tree*, ***keywords*)

Bases: *lingpy.thirdparty.cogent.tree.PhyloNode*

Basic class for the handling of phylogenetic trees.

Parameters **tree** : {str file list}

A string or a file containing trees in Newick format. As an alternative, you can also simply pass a list containing taxon names. In that case, a random tree will be created from the list of taxa.

branch_lengths : bool (default=False)

When set to *True*, and a list of taxa is passed instead of a Newick string or a file containing a Newick string, a random tree with random branch lengths will be created with the branch lengths being in order of the maximum number of the total number of internal branches.

getDistanceToRoot(*node*)

Return the distance from the given node to the root.

Parameters **node** : str

The name of a given node in the tree.

Returns **distance** : int

The distance of the given node to the root of the tree.

get_distance(*other*, *distance='grf'*, *debug=False*)

Function returns the Robinson-Foulds distance between the two trees.

Parameters **other** : lingpy.basic.tree.Tree

A tree object. It should have the same number of taxa as the intitial tree.

distance : { grf, rf, branch, symmetric} (default=grf)

The distance which shall be calculated. Select between:

- grf: the generalized Robinson-Foulds Distance
- rf: the Robinson-Foulds Distance
- **symmetric: the symmetric difference between all partitions of** the trees

lingpy.basic.tree.**random_tree**(*taxa*, *branch_lengths=False*)

Create a random tree from a list of taxa.

Parameters **taxa** : list

The list containing the names of the taxa from which the tree will be created.

branch_lengths : bool (default=False)

When set to *True*, a random tree with random branch lengths will be created with the branch lengths being in order of the maximum number of the total number of internal branches.

Returns tree_string : str

A string representation of the random tree in Newick format.

lingpy.basic.wordlist module

This module provides a basic class for the handling of word lists.

class lingpy.basic.wordlist.**BounceAsID**

Bases: object

A helper class for CLDF conversion when tables are missing.

A class with trivial item lookup:

```
>>> b = BounceAsID()
>>> b[5]
{"ID": 5}
>>> b["long_id"]
{"ID": "long_id"}
```

class lingpy.basic.wordlist.**Wordlist**(*filename*, *row='concept'*, *col='doculect'*, *conf=None*)

Bases: *lingpy.basic.parser.QLCParserWithRowsAndCols*

Basic class for the handling of multilingual word lists.

Parameters filename : { string, dict }

The input file that contains the data. Otherwise a dictionary with consecutive integers as keys and lists as values with the key 0 specifying the header.

row : str (default = concept)

A string indicating the name of the row that shall be taken as the basis for the tabular representation of the word list.

col : str (default = doculect)

A string indicating the name of the column that shall be taken as the basis for the tabular representation of the word list.

conf : string (default=)

A string defining the path to the configuration file (more information in the notes).

Notes

A word list is created from a dictionary containing the data. The idea is a three-dimensional representation 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). The LingPy website offers some tutorials for word lists which we recommend to read in case you are looking for more information.

A couple of methods is provided along with the word list class in order to access the multi-dimensional input data. The main idea is to provide an easy way to access two-dimensional slices of the data by specifying which entry type should be returned. Thus, if a word list consists not only of simple orthographical entries but also of IPA encoded phonetic transcriptions, both the orthographical source and the IPA transcriptions can be easily accessed as two separate two-dimensional lists.

add_entries (*entry*, *source*, *function*, *override=False*, ***keywords*)
Add new entry-types to the word list by modifying given ones.

Parameters **entry** : string

A string specifying the name of the new entry-type to be added to the word list.

source : string

A string specifying the basic entry-type that shall be modified. If multiple entry-types shall be used to create a new entry, they should be passed in a simple string separated by a comma.

function : function

A function which is used to convert the source into the target value.

keywords : {dict}

A dictionary of keywords that are passed as parameters to the function.

Notes

This method can be used to add new entry-types to the data by converting given ones. There are a lot of possibilities for adding new entries, but the most basic procedure is to use an existing entry-type and to modify it with help of a function.

calculate (*data*, *taxa='taxa'*, *concepts='concepts'*, *ref='cogid'*, ***keywords*)
Function calculates specific data.

Parameters **data** : str

The type of data that shall be calculated. Currently supports

- tree: calculate a reference tree based on shared cognates
- dst: get distances between taxa based on shared cognates
- cluster: cluster the taxa into groups using different methods

coverage (*stats='absolute'*)
Function determines the coverage of a wordlist.

export (*fileformat*, *sections=None*, *entries=None*, *entry_sep="*, *item_sep="*, *template="*, ***keywords*)
Export the wordlist to specific fileformats.

Notes

The difference between export and output is that the latter mostly serves for internal purposes and formats, while the former serves for publication of data, using specific, nested statements to create, for example, HTML or LaTeX files from the wordlist data.

classmethod from_cldf (*path*, *columns=[]*, *filter=<function Wordlist.<lambda>>*, **args*, ***kwargs*)
Load a CLDF dataset.

Open a CLDF Dataset – with metadata or metadata-free – (only Wordlist datasets are supported for now, because other modules dont seem to make sense for LingPy) and transform it into this Class. Columns from the FormTable are imported in lowercase, columns from LanguageTable, ParameterTable and CognateTable are prefixed with *langage_*, *concept_* and ‘cogid_‘and converted to lowercase.

Parameters columns: list of strings :

The list of columns to import. (default: all columns)

filter: function: rowdict → bool :

A condition function for importing only some rows. (default: lambda row: row[form])

All other parameters are passed on to the ‘cls‘ :

Returns A ‘cls‘ object representing the CLDF dataset :

Notes

CLDFs default column names for wordlists are different from LingPys, so you probably have to use:

```
>>> lingpy.Wordlist.from_cldf(
```

Wordlist-metadata.json, col=language_id, row=parameter_id, segments=segments, transcription=form)

in order to avoid errors from LingPy not finding required columns.

get_dict (*col=”*, *row=”*, *entry=”*, ***keywords*)

Function returns dictionaries of the cells matched by the indices.

Parameters col : string (default=)

The column index evaluated by the method. It should contain one of the values in the row of the `Wordlist` instance, usually a taxon (language) name.

row : string (default=)

The row index evaluated by the method. It should contain one of the values in the row of the `Wordlist` instance, usually a taxon (language) name.

entry : string (default=)

The index for the entry evaluated by the method. It can be used to specify the datatype of the rows or columns selected. As a default, the indices of the entries are returned.

Returns entries : dict

A dictionary of keys and values specifying the selected part of the data. Typically, this can be a dictionary of a given language with keys for the concept and values as specified in the entry keyword.

See also:

`Wordlist.get_list`, *`Wordlist.add_entries`*

Notes

The col and row keywords in the function are all aliased according to the description in the `wordlist.rc` file. Thus, instead of using these attributes, the aliases can also be taken. For selecting a language, one may type something like:

```
>>> Wordlist.get_dict(language='LANGUAGE')
```

and for the selection of a concept, one may type something like:

```
>>> Wordlist.get_dict(concept='CONCEPT')
```

See the examples below for details.

Examples

Load the `harry_potter.csv` file:

```
>>> wl = Wordlist('harry_potter.csv')
```

Select all IPA-entries for the language German:

```
>>> wl.get_dict(language='German',entry='ipa')
{'Harry': ['haralt'], 'hand': ['hant'], 'leg': ['bain']}
```

Select all words (orthographical representation) for the concept Harry:

```
>>> wl.get_dict(concept="Harry",entry="words")
{'English': ['hæri'], 'German': ['haralt'], 'Russian': ['gari'],
↪'Ukrainian': ['gari']}
```

Note that the values of the dictionary that is returned are always lists, since it is possible that the original file contains synonyms (multiple words corresponding to the same concept).

get_distances(***kw*)

get_etymdict(*ref='cogid'*, *entry="*, *modify_ref=False*)
Return an etymological dictionary representation of the word list.

Parameters **ref** : string (default = cogid)

The reference entry which is used to store the cognate ids.

entry : string (default =)

The entry-type which shall be selected.

modify_ref : function (default=False)

Use a function to modify the reference. If your cognate identifiers are numerical, for example, and negative values are assigned as loans, but you want to suppress this behaviour, just set this keyword to abs, and all cognate IDs will be converted to their absolute value.

Returns **etym_dict** : dict

An etymological dictionary representation of the data.

Notes

In contrast to the word-list representation of the data, an etymological dictionary representation sorts the counterparts according to the cognate sets of which they are reflexes. If more than one cognate ID are assigned to an entry, for example in cases of fuzzy cognate IDs or partial cognate IDs, the etymological dictionary will return one cognate set for each of the IDs.

get_list (*row=”*, *col=”*, *entry=”*, *flat=False*, ***keywords*)

Function returns lists of rows and columns specified by their name.

Parameters **row: string (default =)** :

The row name whose entries are selected from the data.

col : string (default =)

The column name whose entries are selected from the data.

entry: string (default =) :

The entry-type which is selected from the data.

flat : bool (default = False)

Specify whether the returned list should be one- or two-dimensional, or whether it should contain gaps or not.

Returns **data** : list

A list representing the selected part of the data.

See also:

Wordlist.get_list, *Wordlist.add_entries*

Notes

The col and row keywords in the function are all aliased according to the description in the `wordlist.rc` file. Thus, instead of using these attributes, the aliases can also be taken. For selecting a language, one may type something like:

```
>>> Wordlist.get_list(language='LANGUAGE')
```

and for the selection of a concept, one may type something like:

```
>>> Wordlist.get_list(concept='CONCEPT')
```

See the examples below for details.

Examples

Load the `harry_potter.csv` file:

```
>>> wl = Wordlist('harry_potter.csv')
```

Select all IPA-entries for the language German:

```
>>> wl.get_list(language='German',entry='ipa'
['bain', 'hant', 'haralt']
```

Note that this function returns 0 for missing values (concepts that dont have a word in the given language). If one wants to avoid this, the flat keyword should be set to *True*.

Select all words (orthographical representation) for the concept Harry:

```
>>> wl.get_list(concept="Harry",entry="words")
[['Harry', 'Harald', '', 'i']]
```

Note that the values of the list that is returned are always two-dimensional lists, since it is possible that the original file contains synonyms (multiple words corresponding to the same concept). If one wants to have a flat representation of the entries, the flat keyword should be set to *True*:

```
>>> wl.get_list(concept="Harry",entry="words",flat=True)
['hæri', 'haralt', 'gari', 'hari']
```

get_paps (*ref='cogid'*, *entry='concept'*, *missing=0*, *modify_ref=False*)
Function returns a list of present-absent-patterns of a given word list.

Parameters **ref** : string (default = cogid)

The reference entry which is used to store the cognate ids.

entry : string (default = concept)

The field which is used to check for missing data.

missing : string,int (default = 0)

The marker for missing items.

get_tree (***kw*)

iter_rows (**entries*)
Iterate over the columns in a wordlist.

Parameters **entries** : list

The name of the columns which shall be iterated.

Returns **iterator** : iterator

An iterator yielding lists in which the first entry is the ID of the wordlist row and the following entries are the content of the columns as specified.

Examples

Load a wordlist from LingPys test data:

```
>>> from lingpy.tests.util import test_data
>>> from lingpy import Wordlist
>>> wl = Wordlist(test_data("KSL.qlc"))
>>> list(wl.iter_rows('ipa'))[:10]
[[1, 'iθ'],
 [2, 'l'],
 [3, 'tut'],
 [4, 'al'],
 [5, 'apa.u'],
 [6, 'ayo'],
 [7, 'bytyn'],
 [8, 'e'],
 [9, 'and'],
 [10, 'e']]
```

So as you can see, the function returns the key of the wordlist as well as the specified entry.

output (*fileformat*, ***keywords*)
Write wordlist to file.

Parameters **fileformat** : {tsv,tre,nwk,dst, taxa, starling, paps.nex, paps.csv}

The format that is written to file. This corresponds to the file extension, thus tsv creates a file in extended tsv-format, dst creates a file in Phylip-distance format, etc.

filename : str

Specify the name of the output file (defaults to a filename that indicates the creation date).

subset : bool (default=False)

If set to c{True}, return only a subset of the data. Which subset is specified in the keywords cols and rows.

cols : list

If *subset* is set to c{True}, specify the columns that shall be written to the csv-file.

rows : dict

If *subset* is set to c{True}, use a dictionary consisting of keys that specify a column and values that give a Python-statement in raw text, such as, e.g., == hand. The content of the specified column will then be checked against statement passed in the dictionary, and if it is evaluated to c{True}, the respective row will be written to file.

ref : str

Name of the column that contains the cognate IDs if starling is chosen as an output format.

missing : { str, int } (default=0)

If paps.nex or paps.csv is chosen as fileformat, this character will be inserted as an indicator of missing data.

tree_calc : {neighbor, upgma}

If no tree has been calculated and tre or nwk is chosen as output format, the method that is used to calculate the tree.

threshold : float (default=0.6)

The threshold that is used to carry out a flat cluster analysis if groups or cluster is chosen as output format.

ignore : { list, all (default=all)}

Modifies the output format in tsv output and allows to ignore certain blocks in extended tsv, like msa, taxa, json, etc., which should be passed as a list. If you choose all as a plain string and not a list, this will ignore all additional blocks and output only plain tsv.

prettify : bool (default=False)

Inserts comment characters between concepts in the tsv file output format, which makes it easier to see blocks of words denoting the same concept. Switching this off will output the file in plain tsv.

`renumber`(*source*, *target=''*, *override=False*)

Renumber a given set of string identifiers by replacing the ids by integers.

Parameters **source** : str

The source column to be manipulated.

target : str (default=)

The name of the target colummn. If no name is chosen, the target column will be manipulated by adding id to the name of the source column.

override : bool (default=False)

Force to overwrite the data if the target column already exists.

Notes

In addition to a new column, an further entry is added to the _meta attribute of the wordlist by which newly coined ids can be retrieved from the former string attributes. This attribute is called source2target and can be accessed either via the _meta dictionary or directly as an attribute of the wordlist.

`lingpy.basic.wordlist.`**`from_cldf`**(*path*, *to=<class 'lingpy.basic.wordlist.Wordlist'>*, *concept='Name'*, *concepticon='Concepticon_ID'*, *glottocode='Glottocode'*, *language='Name'*)

Load data from CLDF into a LingPy Wordlist object or similar.

Parameters **path** : str

The path to the metadata-file of your CLDF dataset.

to : ~lingpy.basic.wordlist.Wordlist

A ~lingpy.basic.wordlist.Wordlist object or one of the descendants (LexStat, Alignmnent).

concept : str (default=gloss)

The name used for the basic gloss in the *parameters.csv* table.

glottocode : str (default=glottocode)

The default name for the column storing the Glottolog ID in the *languages.csv* table.

language : str (default=name)

The default name for the language name in the *languages.csv* table.

concepticon : str (default=conceptset)

The default name for the concept set in the *paramters.csv* table.

Notes

This function does not offer absolute flexibility regarding the data you can input so far. However, it can regularly read CLDF-formatted data into LingPy and thus allow you to use CLDF data in LingPy analyses.

`lingpy.basic.wordlist.`**`get_wordlist`**(*path*, *delimiter=','*, *quotechar='"'*, *normalization_form='NFC'*, ***keywords*)

Load a wordlist from a normal CSV file.

Parameters **path** : str

The path to your CSV file.

delimiter : str

The delimiter in the CSV file.

quotechar : str

The quote character in your data.

row : str (default = concept)

A string indicating the name of the row that shall be taken as the basis for the tabular representation of the word list.

col : str (default = doculect)

A string indicating the name of the column that shall be taken as the basis for the tabular representation of the word list.

conf : string (default=)

A string defining the path to the configuration file.

Notes

This function returns a *Wordlist* object. In contrast to the normal way to load a wordlist from a tab-separated file, however, this allows to directly load a wordlist from any normal csv-file, with your own specified delimiters and quote characters. If the first cell in the first row of your CSV file is not named ID, the integer identifiers, which are required by LingPy will be automatically created.

Module contents

This module provides basic classes for the handling of linguistic data.

The basic idea is to provide classes that allow the user to handle basic linguistic datatypes (spreadsheets, wordlists) in a consistent way.

lingpy.compare package

Submodules

lingpy.compare.lexstat module

class lingpy.compare.lexstat.**LexStat**(*filename*, ***keywords*)

Bases: *lingpy.basic.wordlist.Wordlist*

Basic class for automatic cognate detection.

Parameters filename : str

The name of the file that shall be loaded.

model : *Model*

The sound-class model that shall be used for the analysis. Defaults to the SCA sound-class model.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged into single tokens or kept apart as separate tokens.

transform : dict

A dictionary that indicates how prosodic strings should be simplified (or generally transformed), using a simple key-value structure with the key referring to the original prosodic context and the value to the new value. Currently, prosodic strings (see `prosodic_string()`) offer 11 different prosodic contexts. Since not all these are helpful in preliminary analyses for cognate detection, it is useful to merge some of these contexts into one. The default settings distinguish only 5 instead of 11 available contexts, namely:

- `C` for all consonants in prosodically ascending position,
- `c` for all consonants in prosodically descending position,
- `V` for all vowels,
- `T` for all tones, and
- `_` for word-breaks.

Make sure to check also the vowel keyword when initialising a LexStat object, since the symbols you use for vowels and tones should be identical with the ones you define in your transform dictionary.

vowels : str (default=**VT_**)

For scoring function creation using the `get_scorer` function, you have the possibility to use reduced scores for the matching of tones and vowels by modifying the vscale parameter, which is set to 0.5 as a default. In order to make sure that vowels and tones are properly detected, make sure your prosodic string representation of vowels matches the one in this keyword. Thus, if you change the prosodic strings using the transform keyword, you also need to change the vowel string, to make sure that vscale works as wanted in the `get_scorer` function.

check : bool (default=False)

If set to **True**, the input file will first be checked for errors before the calculation is carried out. Errors will be written to the file **errors**, defaulting to `errors.log`. See also `apply_checks` apply_checks : bool (default=False) If set to **True**, any errors identified by *check* will be handled silently.

no_bscorer: bool (default=False) :

If set to **True**, this will suppress the creation of a language-specific scoring function (which may become quite large and is additional ballast if the method lexstat is not used after all. If you use the lexstat method, however, this needs to be set to **False**.

errors : str

The name of the error log.

segments : str (default=tokens)

The name of the column in your data which contains the segmented transcriptions, or in which the segmented transcriptions should be placed.

transcription : str (default=ipa)

The name of the column in your data which contains the unsegmented transcriptions.

classes : str (default=classes)

The name of the column in the data which contains the sound class representation of the transcriptions, or in which this information shall be placed after automatic conversion.

numbers : str (default=numbers)

The language-specific triples consisting of language id (numeric), sound class string (one character only), and prosodic string (one character only). Usually, numbers are automatically created from the columns classes, prostrings, and langid, but you can also provide them in your data.

langid : str (default=langid)

Name of the column that contains a numerical language identifier, needed to produce the language-specific character triples (numbers). Unless specific explicitly, this is automatically created.

prostrings : str (default=prostrings)

Name of the column containing prosodic strings (see `List2014d` for more details) of the segmented transcriptions, containing one character per prosodic string. Prostrings add a contextual component to phonetic sequences. They are automatically created, but can likewise be submitted from the initial data.

weights : str (default=weights)

The name of the column which stores the individual gap-weights for each sequence. Gap weights are positive floats for each segment in a string, which modify the gap opening penalty during alignment.

tokenize : function (default=ipa2tokens)

The function which should be used to tokenize the entries in the column storing the transcriptions in case no segmentation is provided by the user.

get_prostring : function (default=prosodic_string)

The function which should be used to create prosodic strings from the segmented transcription data. If you want to completely ignore prosodic strings in LexStat calculations, you could just pass the following function:

```
>>> lex = LexStat('inputfile.tsv', get_prostring=lambda x: ["x
↪" for
    y in x])
```

cldf : bool (default=True)

If set to True, as by default, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved when internally converting tokens to classes (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a `` `source/target` `` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Notes

Instantiating this class does not require a lot of parameters. However, the user may modify its behaviour by providing additional attributes in the input file.

Attributes

pairs	dict	A dictionary with tuples of language names as key and indices as value, pointing to unique combinations of words with the same meaning in all language pairs.
model	*Model*	The sound class model instance which serves to convert the phonetic data into sound classes.
chars	list	A list of all unique language-specific character types in the instantiated LexStat object. The characters in this list consist of • the language identifier (numeric, referenced as langid as a default, but customizable via the keyword langid) • the sound class symbol for the respective IPA transcription value • the prosodic class value All values are represented in the above order as one string, separated by a dot. Gaps are also included in this collection. They are traditionally represented as X for the sound class and - for the prosodic string.
rchars	list	A list containing all unique character types across languages. In contrast to the chars-attribute, the rchars (raw chars) do not contain the language identifier, thus they only consist of two values, separated by a dot, namely, the sound class symbol, and the prosodic class value.
scorer	dict	A collection of *ScoreDict* objects, which are used to score the strings. LexStat distinguishes two different scoring functions: • rscorer: A raw scorer that is not language-specific and consists only of sound class values and prosodic string values. This scorer is traditionally used to carry out the first alignment in order to calculate the language-specific scorer. It is directly accessible as an attribute of the LexStat class (`rscorer`). The characters which constitute the values in this scorer are accessible via the rchars attribue of each lexstat class.

`align_pairs`(*idxA*, *idxB*, *concept=None*, ***keywords*)
Align all or some words of a given pair of languages.

Parameters **idxA,idxB** : {int, str}

Use an integer to refer to the words by their unique internal ID, use language names to select all words for a given language.

method : {lexstat,sca}

Define the method to be used for the alignment of the words.

mode : {global,local,overlap,dialign} (default=overlap)

Select the mode for the alignment analysis.

gop : int (default=-2)

If sca is selected as a method, define the gap opening penalty.

scale : float (default=0.5)

Select the scale for the gap extension penalty.

factor : float (default=0.3)

Select the factor for extra scores for identical prosodic segments.

restricted_chars : str (default=**T_**)

Select the restricted chars (boundary markers) in the prosodic strings in order to enable secondary alignment.

distance : bool (default=True)

If set to c{True}, return the distance instead of the similarity score.

pprint : bool (default=True)

If set to c{True}, print the results to the terminal.

return_distance : bool (default=False)

If set to c{True}, return the distance score, otherwise, nothing will be returned.

`cluster`(*method='sca'*, *cluster_method='upgma'*, *threshold=0.3*, *scale=0.5*, *factor=0.3*, *restricted_chars='_T'*, *mode='overlap'*, *gop=-2*, *restriction="*, *ref="*, *external_function=None*, ***keywords*)
Function for flat clustering of words into cognate sets.

Parameters **method** : {sca,lexstat,edit-dist,turchin} (default=sca)

Select the method that shall be used for the calculation.

cluster_method : {upgma,single,complete, mcl} (default=upgma)

Select the cluster method. upgma (`Sokal1958`) refers to average linkage clustering, mcl refers to the Markov Clustering Algorithm (`Dongen2000`).

threshold : float (default=0.3)

Select the threshold for the cluster approach. If set to c{False}, an automatic threshold will be calculated by calculating the average distance of unrelated sequences (use with care).

scale : float (default=0.5)

Select the scale for the gap extension penalty.

factor : float (default=0.3)

Select the factor for extra scores for identical prosodic segments.

restricted_chars : str (default=**T_**)

Select the restricted chars (boundary markers) in the prosodic strings in order to enable secondary alignment.

mode : {global,local,overlap,dialign} (default=overlap)

Select the mode for the alignment analysis.

verbose : bool (default=False)

Define whether verbose output should be used or not.

gop : int (default=-2)

If sca is selected as a method, define the gap opening penalty.

restriction : {cv} (default=)

Specify the restriction for calculations using the edit-distance. Currently, only cv is supported. If *edit-dist* is selected as *method* and *restriction* is set to *cv*, consonant-vowel matches will be prohibited in the calculations and the edit distance will be normalized by the length of the alignment rather than the length of the longest sequence, as described in `Heeringa2006`.

inflation : {int, float} (default=2)

Specify the inflation parameter for the use of the MCL algorithm.

expansion : int (default=2)

Specify the expansion parameter for the use of the MCL algorithm.

`get_distances`(*method='sca', mode='overlap', gop=-2, scale=0.5, factor=0.3, restricted_chars='T_', aggregate=True*)

Method calculates different distance estimates for language pairs.

Parameters **method** : {sca,lexstat,edit-dist,turchin} (default=sca)

Select the method that shall be used for the calculation.

runs : int (default=100)

Select the number of random alignments for each language pair.

mode : {global,local,overlap,dialign} (default=overlap)

Select the mode for the alignment analysis.

gop : int (default=-2)

If sca is selected as a method, define the gap opening penalty.

scale : float (default=0.5)

Select the scale for the gap extension penalty.

factor : float (default=0.3)

Select the factor for extra scores for identical prosodic segments.

restricted_chars : str (default=**T_**)

Select the restricted chars (boundary markers) in the prosodic strings in order to enable secondary alignment.

aggregate : bool (default=True)

Return aggregated distances in form of a distance matrix for all taxa in the data.

Returns **D** : c{numpy.array}

An array with all distances calculated for each sequence pair.

get_frequencies (*ftype='sounds'*, *ref='tokens'*, *aggregated=False*)

Computes the frequencies of a given wordlist.

Parameters **ftype: str (default=sounds)** :

The type of frequency which shall be calculated. Select between sounds (type-token frequencies of sounds), and wordlength (average word length per taxon or in aggregated form), or diversity for the diversity index (requires that you have carried out cognate judgments, and make sure to set the ref keyword to the column in which your cognates are).

ref : str (default=tokens)

The reference column, with the column for tokens as a default. Make sure to modify this keyword in case you want to check for the diversity.

aggregated : bool (default=False)

Determine whether frequencies should be calculated in an aggregated way, for all languages, or on a language-per-language basis.

Returns **freqs** : {dict, float}

Depending on the selection of the datatype you chose, this returns either a dictionary containing the frequencies or a float indicating the ratio.

get_random_distances (*method='lexstat'*, *runs=100*, *mode='overlap'*, *gop=-2*, *scale=0.5*, *factor=0.3*, *restricted_chars='T_'*)

Method calculates randoms scores for unrelated words in a dataset.

Parameters **method** : {sca,lexstat,edit-dist,turchin} (default=sca)

Select the method that shall be used for the calculation.

runs : int (default=100)

Select the number of random alignments for each language pair.

mode : {global,local,overlap,dialign} (default=overlap)

Select the mode for the alignment analysis.

gop : int (default=-2)

If sca is selected as a method, define the gap opening penalty.

scale : float (default=0.5)

Select the scale for the gap extension penalty.

factor : float (default=0.3)

Select the factor for extra scores for identical prosodic segments.

restricted_chars : str (default=T_)

Select the restricted chars (boundary markers) in the prosodic strings in order to enable secondary alignment.

Returns **D** : c{numpy.array}

An array with all distances calculated for each sequence pair.

get_scorer(***keywords*)

Create a scoring function based on sound correspondences.

Parameters **method** : str (default=shuffle)

Select between markov, for automatically generated random strings, and shuffle, for random strings taken directly from the data.

ratio : tuple (default=3,2)

Define the ratio between derived and original score for sound-matches.

vscale : float (default=0.5)

Define a scaling factor for vowels, in order to decrease their score in the calculations.

runs : int (default=1000)

Choose the number of random runs that shall be made in order to derive the random distribution.

threshold : float (default=0.7)

The threshold which used to select those words that are compared in order to derive the attested distribution.

modes : list (default = [(global,-2,0.5),(local,-1,0.5)])

The modes which are used in order to derive the distributions from pairwise alignments.

factor : float (default=0.3)

The scaling factor for sound segments with identical prosodic environment.

force : bool (default=False)

Force recalculation of existing distribution.

preprocessing: bool (default=False) :

Select whether SCA-analysis shall be used to derive a preliminary set of cognates from which the attested distribution shall be derived.

rands : int (default=1000)

If method is set to markov, this parameter defines the number of strings to produce for the calculation of the random distribution.

limit : int (default=10000)

If method is set to markov, this parameter defines the limit above which no more search for unique strings will be carried out.

cluster_method : {upgma single complete} (default=upgma)

Select the method to be used for the calculation of cognates in the preprocessing phase, if preprocessing is set to c{True}.

gop : int (default=-2)

If preprocessing is selected, define the gap opening penalty for the preprocessing calculation of cognates.

unattested : {int, float} (default=-5)

If a pair of sounds is not attested in the data, but expected by the alignment algorithm that computes the expected distribution, the score would be -infinity. Yet in order to allow to smooth this behaviour and to reduce the strictness, we set a default negative value which does not necessarily need to be too high, since it may well be that we miss a potentially good pairing in the first runs of alignment analyses. Use this keyword to adjust this parameter.

unexpected : {int, float} (default=0.000001)

If a pair is encountered in a given alignment but not expected according to the randomized alignments, the score would be not calculable, since we had to divide by zero. For this reason, we set a very small constant, by which the score is divided in this case. Not that this constant is only relevant in those cases where the shuffling procedure was not carried out long enough.

`get_subset` (*sublist*, *ref='concept'*)

Function creates a specific subset of all word pairs.

Parameters **sublist** : list

A list which contains those items which should be considered for the subset creation, for example, a list of concepts.

ref : string (default=concept)

The reference point to compare the given sublist.

Notes

This function can be used to consider only a smaller part of word pairs when creating a scorer. Normally, all words are compared, but defining a subset allows to compare only those belonging to a specific concept list (Swadesh list).

`output` (*fileformat*, ***keywords*)

Write data to file.

Parameters **fileformat** : {tsv, tre,nwk,dst, taxa,starling, paps.nex, paps.csv}

The format that is written to file. This corresponds to the file extension, thus tsv creates a file in tsv-format, dst creates a file in Phylip-distance format, etc.

filename : str

Specify the name of the output file (defaults to a filename that indicates the creation date).

subset : bool (default=False)

If set to c{True}, return only a subset of the data. Which subset is specified in the keywords cols and rows.

cols : list

If *subset* is set to c{True}, specify the columns that shall be written to the csv-file.

rows : dict

If *subset* is set to c{True}, use a dictionary consisting of keys that specify a column and values that give a Python-statement in raw text, such as, e.g., == hand. The content of the specified column will then be checked against statement passed in the dictionary, and if it is evaluated to c{True}, the respective row will be written to file.

ref : str

Name of the column that contains the cognate IDs if starling is chosen as an output format.

missing : { str, int } (default=0)

If paps.nex or paps.csv is chosen as fileformat, this character will be inserted as an indicator of missing data.

tree_calc : {neighbor, upgma}

If no tree has been calculated and tre or nwk is chosen as output format, the method that is used to calculate the tree.

threshold : float (default=0.6)

The threshold that is used to carry out a flat cluster analysis if groups or cluster is chosen as output format.

ignore : { list, all }

Modifies the output format in tsv output and allows to ignore certain blocks in extended tsv, like msa, taxa, json, etc., which should be passed as a list. If you choose all as a plain string and not a list, this will ignore all additional blocks and output only plain tsv.

prettify : bool (default=True)

Inserts comment characters between concepts in the tsv file output format, which makes it easier to see blocks of words denoting the same concept. Switching this off will output the file in plain tsv.

`lingpy.compare.lexstat.`**`char_from_charstring`**(*cstring*)

`lingpy.compare.lexstat.`**`get_score_dict`**(*chars*, *model*)

lingpy.compare.partial module

Module provides a class for partial cognate detection, expanding the LexStat class.

`class` `lingpy.compare.partial.`**`Partial`**(*infile*, ***keywords*)

Bases: *`lingpy.compare.lexstat.LexStat`*

Extended class for automatic detection of partial cognates.

Parameters **filename** : str

The name of the file that shall be loaded.

model : *`Model`*

The sound-class model that shall be used for the analysis. Defaults to the SCA sound-class model.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged into single tokens or kept apart as separate tokens.

transform : dict

A dictionary that indicates how prosodic strings should be simplified (or generally transformed), using a simple key-value structure with the key referring to the original prosodic context and the value to the new value. Currently, prosodic strings (see `prosodic_string()`) offer 11 different prosodic contexts. Since not all these are helpful in preliminary analyses for cognate detection, it is useful to merge some of these contexts into one. The default settings distinguish only 5 instead of 11 available contexts, namely:

- `C` for all consonants in prosodically ascending position,
- `c` for all consonants in prosodically descending position,
- `V` for all vowels,
- `T` for all tones, and
- `_` for word-breaks.

Make sure to check also the vowel keyword when initialising a LexStat object, since the symbols you use for vowels and tones should be identical with the ones you define in your transform dictionary.

vowels : str (default=**VT_**)

For scoring function creation using the `get_scorer` function, you have the possibility to use reduced scores for the matching of tones and vowels by modifying the vscale parameter, which is set to 0.5 as a default. In order to make sure that vowels and tones are properly detected, make sure your prosodic string representation of vowels matches the one in this keyword. Thus, if you change the prosodic strings using the transform keyword, you also need to change the vowel string, to make sure that vscale works as wanted in the `get_scorer` function.

check : bool (default=False)

If set to **True**, the input file will first be checked for errors before the calculation is carried out. Errors will be written to the file **errors**, defaulting to `errors.log`. See also `apply_checks`

apply_checks : bool (default=False)

If set to **True**, any errors identified by *check* will be handled silently.

no_bscorer: bool (default=False) :

If set to **True**, this will suppress the creation of a language-specific scoring function (which may become quite large and is additional ballast if the method lexstat is not used after all. If you use the lexstat method, however, this needs to be set to **False**.

errors : str

The name of the error log.

Notes

This method automatically infers partial cognate sets from data which was previously morphologically segmented.

Attributes

pairs	dict	A dictionary with tuples of language names as key and indices as value, pointing to unique combinations of words with the same meaning in all language pairs.
model	*Model*	The sound class model instance which serves to convert the phonetic data into sound classes.
chars	list	A list of all unique language-specific character types in the instantiated LexStat object. The characters in this list consist of • the language identifier (numeric, referenced as langid as a default, but customizable via the keyword langid) • the sound class symbol for the respective IPA transcription value • the prosodic class value All values are represented in the above order as one string, separated by a dot. Gaps are also included in this collection. They are traditionally represented as X for the sound class and - for the prosodic string.
rchars	list	A list containing all unique character types across languages. In contrast to the chars-attribute, the rchars (raw chars) do not contain the language identifier, thus they only consist of two values, separated by a dot, namely, the sound class symbol, and the prosodic class value.
scorer	dict	A collection of *ScoreDict* objects, which are used to score the strings. LexStat distinguishes two different scoring functions: • rscorer: A raw scorer that is not language-specific and consists only of sound class values and prosodic string values. This scorer is traditionally used to carry out the first alignment in order to calculate the language-specific scorer. It is directly accessible as an attribute of the LexStat class (`rscorer`). The characters which constitute the values in this scorer are accessible via the rchars attribue of each lexstat class.

add_cognate_ids(*source*, *target*, *idtype='strict'*, *override=False*)
Compute normal cognate identifiers from partial cognate sets.

Parameters **source: str** :

Name of the source column in your wordlist file.

target : str

Name of the target column in your wordlist file.

idtype : str (default=strict)

Select between strict and loose.

override: bool (default=False) :

Specify whether you want to override existing columns.

Notes

While the computation of strict cognate IDs from partial cognate IDs is straightforward and just judges those words as cognate which are identical in all their parts, the computation of loose cognate IDs constructs a network between all words, draws lines between all words that share a common morpheme, and judges all connected components in this network as cognate.

partial_cluster(*method='sca'*, *threshold=0.45*, *scale=0.5*, *factor=0.3*, *restricted_chars='_T'*, *mode='overlap'*, *cluster_method='infomap'*, *gop=-1*, *restriction="*, *ref="*, *external_function=None*, *split_on_tones=True*, ***keywords*)
Cluster the words into partial cognate sets.

Function for flat clustering of words into cognate sets.

Parameters **method** : {sca,lexstat,edit-dist,turchin} (default=sca)

Select the method that shall be used for the calculation.

cluster_method : {upgma,single,complete, mcl} (default=upgma)

Select the cluster method. upgma (Sokal1958) refers to average linkage clustering, mcl refers to the Markov Clustering Algorithm (Dongen2000).

threshold : float (default=0.3)

Select the threshold for the cluster approach. If set to c{False}, an automatic threshold will be calculated by calculating the average distance of unrelated sequences (use with care).

scale : float (default=0.5)

Select the scale for the gap extension penalty.

factor : float (default=0.3)

Select the factor for extra scores for identical prosodic segments.

restricted_chars : str (default=**T_**)

Select the restricted chars (boundary markers) in the prosodic strings in order to enable secondary alignment.

mode : {global,local,overlap,dialign} (default=overlap)

Select the mode for the alignment analysis.

verbose : bool (default=False)

Define whether verbose output should be used or not.

gop : int (default=-2)

If sca is selected as a method, define the gap opening penalty.

restriction : {cv} (default=)

Specify the restriction for calculations using the edit-distance. Currently, only cv is supported. If *edit-dist* is selected as *method* and *restriction* is set to *cv*, consonant-vowel matches will be prohibited in the calculations and the edit distance will be normalized by the length of the alignment rather than the length of the longest sequence, as described in `Heeringa2006`.

inflation : {int, float} (default=2)

Specify the inflation parameter for the use of the MCL algorithm.

expansion : int (default=2)

Specify the expansion parameter for the use of the MCL algorithm.

lingpy.compare.phylogeny module

Phylogeny-based detection of borrowings in lexicostatistical wordlists.

class `lingpy.compare.phylogeny.`**PhyBo**(*dataset*, *tree=None*, *paps='pap'*, *ref='cogid'*, *tree_calc='neighbor'*, *output_dir=None*, ***keywords*)

Bases: *`lingpy.basic.wordlist.Wordlist`*

Basic class for calculations using the TreBor method.

Parameters **dataset** : string

Name of the dataset that shall be analyzed.

tree : {None, string}

Name of the tree file.

paps : string (default=pap)

Name of the column that stores the specific cognate IDs consisting of an arbitrary integer key and a key for the concept.

ref : string (default=cogid)

Name of the column that stores the general cognate ids (the reference of the analysis).

tree_calc : {neighbor,upgma} (default=neighbor)

Select the algorithm to be used for the tree calculation if no tree is passed with the file.

missing : int (default=-1)

Specify how missing data should be handled. If set to -1, missing data can account for both presence or absence of a cognate set in the given language. If set to 0, missing data is treated as absence.

degree : int (default=100)

The degree which is chosen for the projection of the tree layout.

analyze(*runs='default'*, *mixed=False*, *output_gml=False*, *tar=False*, *full_analysis=True*, *plot_dists=False*, *output_plot=False*, *plot_mln=False*, *plot_msn=False*, ***keywords*)
Carry out a full analysis using various parameters.

Parameters **runs** : {str list} (default=default)

Define a couple of different models to be analyzed. Select between:

- default: weighted analysis, using parsimony and weights for gains and losses
- topdown: use the traditional approach by `Nelson-Sathi2011`
- restriction: use the restriction approach

You can also define your own mix of models.

usetex : bool (default=True)

Specify whether you want to use LaTeX to render plots.

mixed : bool (default=False)

If set to c{True}, calculate a mixed model by selecting the best model for each item separately.

output_gml : bool (default=False)

Set to c{True} in order to output every gain-loss-scenario in GML-format.

full_analysis : bool (default=True)

Specifies whether a full analysis is carried out or not.

plot_mln : bool (default=True)

Select or unselect output plot for the MLN.

plot_msn : bool (default=False)

Select or unselect output plot for the MSN.

get_ACS(*glm*, ***keywords*)
Compute the ancestral character states (ACS) for all internal nodes.

get_AVSD(*glm*, ***keywords*)
Function retrieves all pap s for ancestor languages in a given tree.

get_CVSD()
Calculate the Contemporary Vocabulary Size Distribution (CVSD).

get_GLS(*mode='weighted'*, *ratio=(1, 1)*, *restriction=3*, *output_gml=False*, *output_plot=False*, *tar=False*, ***keywords*)
Create gain-loss-scenarios for all non-singleton paps in the data.

Parameters **mode** : string (default=weighted)

Select between weighted, restriction and topdown. The three modes refer to the following frameworks:

- weighted refers to the weighted parsimony framework described in `List2014b` and `List2014a`. Weights are specified with help of a ratio for the scoring of gain and loss events. The ratio can be defined with help of the *ratio* keyword.
- restrictino refers to a simple method in which only a specific amount of gain events is allowed. The maximally allowed number of gain events can be defined with help of the *restriction* keyword.

- topdown refers to the top-down method outlined in `Dagan2007` and first applied to linguistic data in `Nelson-Sathi2011`. This method also defines a maximal number of gain events, but in contrast to the restriction approach, it starts from the top of the tree and stops if the maximal number of restrictions has been reached. The maximally allowed number of gain events can, again, be specified with help of the *restriction* keyword.

ratio : tuple (default=(1,1))

If weighted mode is selected, define the ratio between the weights for gains and losses.

restriction : int (default=3)

If restriction is selected as mode, define the maximal number of gains.

output_gml : bool (default=False)

If set to c{True}, the decisions for each GLS are stored in a separate file in GML-format.

tar : bool (default=False)

If set to c{True}, the GML-files will be added to a compressed tar-file.

gpl : int (default=1)

Specifies the maximal number of gains per lineage. This parameter specifies how cases should be handled in which a character is first gained, then lost, and then gained again. By setting this parameter to 1 (the default setting), such cases are prohibited, since only one gain per lineage is allowed.

missing_data : int (default=0)

Currently, we offer two ways to handle missing data. The first case just treats missing data in the same way in which the absence of a character is handled and can be evoked by setting this parameter to 0. The second case will treat missing data as either absent or present characters, based on how well each option coincides with the overall evolutionary scenario. This behaviour can be evoked by setting this parameter to -1.

push_gains: bool (default=True) :

In bottom-up calculations, there will often be multiple scenarios upon which only one is selected by the method. In order to define consistent criteria for scenario selection, we follow `Mirkin2003` in allowing to force the algorithm to prefer those scenarios in which gains are pushed to the leaves. This behaviour is handle by this parameter. Setting it to *True* will force the algorithm to push gain events to the leaves of the tree. Setting it to *False* will force it to prefer those scenarios where the gains are closer to the root.

`get_IVSD`(*output_gml=False*, *output_plot=False*, *tar=True*, *leading_model=False*, *mixed_threshold=0.0*, *evaluation='mwu'*, ***keywords*)
Calculate VSD on the basis of each item.

`get_MLN`(*glm*, *threshold=1*, *method='mr'*)
Compute an Minimal Lateral Network for a given model.

Parameters **glm** : str

The dictionary key for the gain-loss-model.

threshold : int (default=1)

The threshold used to exclude edges.

method : str (default=mr)

Select the method for MLN calculation. Choose between: * mr: majority-rule, multiple links are resolved by selecting

those which occur most frequently

- td: tree-distance, multiple links are resolved by selecting those which are closest on the tree
- bc: betweenness-centrality, multiple links are resolved by selecting those which have the highest betweenness centrality

get_MSN(*glm=”*, *external_edges=False*, *deep_nodes=False*, ***keywords*)
Plot the Minimal Spatial Network.

Parameters **glm** : str (default=)

A string that encodes which model should be plotted.

filename : str

The name of the file to which the plot shall be written.

fileformat : str

The output format of the plot.

threshold : int (default=1)

The threshold for the minimal amount of shared links that shall be plotted.

usetex : bool (default=True)

Specify whether LaTeX shall be used for the plot.

get_PDC(*glm*, ***keywords*)
Calculate Patchily Distributed Cognates.

get_edge(*glm*, *nodeA*, *nodeB*, *entries=”*, *msn=False*)
Return the edge data for a given gain-loss model.

get_stats(*glm*, *subset=”*, *filename=”*)
Calculate basic statistics for a given gain-loss model.

plot_ACS(*glm*, ***keywords*)
Plot a tree in which the node size correlates with the size of the ancestral node.

plot_GLS(*glm*, ***keywords*)
Plot the inferred scenarios for a given model.

plot_MLN(*glm=”*, *fileformat=’pdf’*, *threshold=1*, *usetex=False*, *taxon_labels=’taxon_short_labels’*, *alphat=False*, *alpha=0.75*, ***keywords*)
Plot the MLN with help of Matplotlib.

glm [str (default=)] Identifier for the gain-loss model that is plotted. Defaults to the model that had the best scores in terms of probability.

filename [str (default=)] If no filename is selected, the filename is identical with the dataset.

fileformat [{svg,png,jpg,pdf} (default=pdf)] Select the format of the output plot.

threshold [int (default=1)] Select the threshold for drawing lateral edges.

usetex [bool (default=True)] Specify whether you want to use LaTeX to render plots.

colormap [{None matplotlib.cm}] A `matplotlib.colormap` instance. If set to c{None}, this defaults to `jet`.

taxon_labels [str (default=taxon.short_labels)] Specify the taxon labels that should be included in the plot.

plot_MLN_3d(*glm=”*, *filename=”*, *fileformat='pdf'*, *threshold=1*, *usetex=True*, *colormap=None*, *taxon_labels='taxon_short_labels'*, *alphat=False*, *alpha=0.75*, ***keywords*)
Plot the MLN with help of Matplotlib in 3d.

glm [str (default=)] Identifier for the gain-loss model that is plotted. Defaults to the model that had the best scores in terms of probability.

filename [str (default=)] If no filename is selected, the filename is identical with the dataset.

fileformat [{svg,png,jpg,pdf} (default=pdf)] Select the format of the output plot.

threshold [int (default=1)] Select the threshold for drawing lateral edges.

usetex [bool (default=True)] Specify whether you want to use LaTeX to render plots.

colormap [{None matplotlib.cm}] A `matplotlib.colormap` instance. If set to c{None}, this defaults to `jet`.

taxon_labels [str (default=taxon.short_labels)] Specify the taxon labels that should be included in the plot.

plot_MSN(*glm=”*, *fileformat='pdf'*, *threshold=1*, *usetex=False*, *alphat=False*, *alpha=0.75*, *only=[]*, ***keywords*)
Plot a minimal spatial network.

plot_concept_evolution(*glm*, *concept=”*, *fileformat='png'*, ***keywords*)
Plot the evolution of specific concepts along the reference tree.

plot_two_concepts(*concept*, *cogA*, *cogB*, *labels={1: '1', 2: '2', 3: '3', 4: '4'}*, *tcolor={1: 'white', 2: 'black', 3: '0.5', 4: '0.1'}*, *filename='pdf'*, *fileformat='pdf'*, *usetex=True*)
Plot the evolution of two concepts in space.

Notes

This function may be useful to contrast patterns of different words in geographic space.

`lingpy.compare.phylogeny.`**TreBor**
alias of *`lingpy.compare.phylogeny.PhyBo`*

`lingpy.compare.phylogeny.`**get_gls**(*paps*, *taxa*, *tree*, *gpl=1*, *weights=(1, 1)*, *push_gains=True*, *missing_data=0*)
Calculate a gain-loss scenario.

Parameters **paps** : list

A list containing the presence-absence patterns for all leaves of the reference tree. Presence is indicated by 1, and absence by 0. Missing characters are indicated by -1.

taxa : list

The list of taxa (leaves of the tree).

tree : str

A tree in Newick-format. Taxon names should (of course) be identical with the names in the list of taxa.

gpl : int

Gains per lineage. Specify the maximal amount of gains per lineage. One lineage is hereby defined as one path in the tree. If set to 0, only one gain per lineage is allowed, if set to 1, one additional gain is allowed, and so on. Use with care, since this will lead to larger computation costs (more possibilities have to be taken care of) and can also be quite unrealistic.

weights : tuple (default=(1,1))

Specify the weights for gains and losses. Setting this parameter to (2,1) will penalize gain events with 2 and loss events with 1.

push_gains : bool (default=True)

Determine whether of a set of equally parsimonious patterns those should be retained that show gains closer to the leaves of the tree or not.

missing_data : int (default=0)

Determine how missing data should be represented. If set to 0 (default), missing data will be treated in the same way as absence character states. If you want missing data to be accounted for in the algorithm, set this parameter to -1.

Notes

This is an enhanced version of the older approach to parsimony-based gain-loss mapping. The algorithm is much faster than the previous one and also written much clearer as to the code. In most tests I ran so far, it also outperformed other approaches by finding more parsimonious solutions.

lingpy.compare.sanity module

Module provides basic checks for wordlists.

`lingpy.compare.sanity.`**`average_coverage`**(*wordlist*, *concepts='concepts'*)
Compute average mutual coverage for a given wordlist.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

Your Wordlist object (or a descendant class).

concepts : str (default=concept)

The column which stores your concepts.

Returns **coverage** : dict

A dictionary of dictionaries whose value is the number of items two languages share.

See also:

mutual_coverage_check, *mutual_coverage_subset*, *mutual_coverage*

Examples

Compute coverage for the KSL.qlc dataset:

```
>>> from lingpy.compare.sanity import average_coverage
>>> from lingpy import *
>>> from lingpy.tests.util import test_data
>>> wl = Wordlist(test_data('KSL.qlc'))
```

(continues on next page)

(continued from previous page)

```
>>> average_coverage(wl)
1.0
```

`lingpy.compare.sanity.`**`mutual_coverage`**(*wordlist*, *concepts='concept'*)

Compute mutual coverage for all language pairs in your data.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

Your Wordlist object (or a descendant class).

concepts : str (default=concept)

The column which stores your concepts.

Returns **coverage** : dict

A dictionary of dictionaries whose value is the number of items two languages share.

See also:

mutual_coverage_check, *mutual_coverage_subset*, *average_coverage*

Examples

Compute coverage for the KSL.qlc dataset:

```
>>> from lingpy.compare.sanity import mutual_coverage
>>> from lingpy import *
>>> from lingpy.tests.util import test_data
>>> wl = Wordlist(test_data('KSL.qlc'))
>>> cov = mutual_coverage(wl)
>>> cov['English']['German']
200
```

`lingpy.compare.sanity.`**`mutual_coverage_check`**(*wordlist*, *threshold*, *concepts='concept'*)

Check whether a given mutual coverage is fulfilled by the dataset.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

Your Wordlist object (or a descendant class).

concepts : str (default=concept)

The column which stores your concepts.

threshold : int

The threshold which should be checked.

Returns **c: bool** :

True, if coverage is fulfilled for all language pairs, False if otherwise.

See also:

mutual_coverage, *mutual_coverage_subset*, *average_coverage*

Examples

Compute minimal mutual coverage for the KSL dataset:

```
>>> from lingpy.compare.sanity import mutual_coverage
>>> from lingpy import *
>>> from lingpy.tests.util import test_data
>>> wl = Wordlist(test_data('KSL.qlc'))
>>> for i in range(wl.height, 1, -1):
        if mutual_coverage_check(wl, i):
            print('mutual coverage is {0}'.format(i))
            break
    200
```

lingpy.compare.sanity.**mutual_coverage_subset**(*wordlist*, *threshold*, *concepts='concept'*)

Compute maximal mutual coverage for all language in a wordlist.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

Your Wordlist object (or a descendant class).

concepts : str (default=concept)

The column which stores your concepts.

threshold : int

The threshold which should be checked.

Returns **coverage** : tuple

A tuple consisting of the number of languages for which the coverage could be found as well as a list of all pairings in which this coverage is possible. The list itself contains the mutual coverage inside each pair and the list of languages.

See also:

mutual_coverage, *mutual_coverage_check*, *average_coverage*

Examples

Compute all sets of languages with coverage at 200 for the KSL dataset:

```
>>> from lingpy.compare.sanity import mutual_coverage_subset
>>> from lingpy import *
>>> from lingpy.tests.util import test_data
>>> wl = Wordlist(test_data('KSL.qlc'))
>>> number_of_languages, pairs = mutual_coverage_subset(wl, 200)
>>> for number_of_items, languages in pairs:
        print(number_of_items, ','.join(languages))
    200 Albanian,English,French,German,Hawaiian,Navajo,Turkish
```

lingpy.compare.sanity.**synonymy**(*wordlist*, *concepts='concept'*, *languages='doculect'*)

Check the number of synonyms per language and concept.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

Your Wordlist object (or a descendant class).

concepts : str (default=concept)

The column which stores your concepts.

languages : str (default=doculect)

The column which stores your language names.

Returns **synonyms** : dict

A dictionary with language and concept as key and the number of synonyms as value.

Examples

Calculate synonymy in KSL.qlc dataset:

```
>>> from lingpy.compare.sanity import synonymy
>>> from lingpy import *
>>> from lingpy.tests.util import test_data
>>> wl = Wordlist(test_data('KSL.qlc'))
>>> syns = synonymy(wl)
>>> for a, b in syns.items():
        if b > 1:
            print(a[0], a[1], b)
```

There is no case where synonymy exceeds 1 word per concept per language, since Kessler2001 was paying particular attention to avoid synonyms.

lingpy.compare.strings module

Module provides various string similarity metrics.

`lingpy.compare.strings.`**`bidist1`**(*a*, *b*, *normalized=True*)
Computes bigram-based distance.

Notes

The binary version. Checks if two bigrams are equal or not.

`lingpy.compare.strings.`**`bidist2`**(*a*, *b*, *normalized=True*)
Computes bigram based distance.

Notes

The comprehensive version of the bigram distance.

`lingpy.compare.strings.`**`bidist3`**(*a*, *b*, *normalized=True*)
Computes bigram based distance.

Notes

Computes the positional version of the bigrams. Assigns a partial distance between two bigrams based on positional similarity of bigrams.

`lingpy.compare.strings.`**`bisim1`**(*a*, *b*, *normalized=True*)
computes the binary version of bigram similarity.

`lingpy.compare.strings.`**`bisim2`**(*a*, *b*, *normalized=True*)
Computes bigram similarity the comprehensive version.

Notes

Computes the number of common 1-grams between two n-grams.

`lingpy.compare.strings.`**`bisim3`**(*a*, *b*, *normalized=True*)
Computes bi-sim the positional version.

Notes

The partial similarity between two bigrams is defined as the number of matching 1-grams at each position.

`lingpy.compare.strings.`**`dice`**(*a*, *b*, *normalized=True*)
Computes the Dice measure that measures the number of common bigrams.

`lingpy.compare.strings.`**`ident`**(*a*, *b*)
Computes the identity between two strings. If yes, returns 1, else, returns 0.

`lingpy.compare.strings.`**`jcd`**(*a*, *b*, *normalized=True*)
Computes the bigram-based Jaccard Index.

`lingpy.compare.strings.`**`jcdn`**(*a*, *b*, *normalized=True*)
Computes the bigram and trigram-based Jaccard Index

`lingpy.compare.strings.`**`lcs`**(*a*, *b*, *normalized=True*)
Computes the longest common subsequence between two strings.

`lingpy.compare.strings.`**`ldn`**(*a*, *b*, *normalized=True*)
Basic Levenshtein distance without swap operation (all operations are equal costs).

See also:

`lingpy.align.pairwise.edit_dist`, *`lingpy.compare.strings.ldn_swap`*

`lingpy.compare.strings.`**`ldn_swap`**(*a*, *b*, *normalized=True*)
Basic Levenshtein distance with swap operation included (identifies metathesis).

`lingpy.compare.strings.`**`prefix`**(*a*, *b*, *normalized=True*)
Computes the longest common prefix between two strings.

`lingpy.compare.strings.`**`tridist1`**(*a*, *b*, *normalized=True*)
Computes trigram-based distance.

Notes

The binary version. Checks if two trigrams are equal or not.

`lingpy.compare.strings.`**`tridist2`**(*a*, *b*, *normalized=True*)
Computes bigram based distance.

Notes

The comprehensive version of the bigram distance.

`lingpy.compare.strings.`**`tridist3`**(*a*, *b*, *normalized=True*)
Computes trigram based distance.

Notes

Computes the positional version of the trigrams. Assigns a partial distance between two trigrams based on positional similarity of trigrams.

`lingpy.compare.strings.`**`trigram`**(*a*, *b*, *normalized=True*)
Computes the number of common trigrams between two strings.

`lingpy.compare.strings.`**`trisim1`**(*a*, *b*, *normalized=True*)
Computes the binary version of trigram similarity.

`lingpy.compare.strings.`**`trisim2`**(*a*, *b*, *normalized=True*)
Computes tri-sim the comprehensive version.

Notes

Simply computes the number of common 1-grams between two n-grams instead of calling LCS as should be done in `Kondrak2005` paper. Note that the LCS for a trigram can be computed in O(n) time if we asssume that list lookup is in constant time.

`lingpy.compare.strings.`**`trisim3`**(*a*, *b*, *normalized=True*)
Computes tri-sim the positional version.

Notes

Simply computes the number of matching 1-grams in each position.

`lingpy.compare.strings.`**`xdice`**(*a*, *b*, *normalized=True*)
Computes the skip 1 character version of Dice.

`lingpy.compare.strings.`**`xxdice`**(*a*, *b*, *normalized=True*)
Returns the XXDice between two strings.

Notes

Taken from `Brew1996`.

lingpy.compare.util module

Module contents

Basic module for language comparison.

lingpy.convert package

Submodules

lingpy.convert.cldf module

Basic functions for the conversion from LingPy to CLDF and vice versa.

`lingpy.convert.cldf.`**`template_path`**(**comps*) `lingpy.convert.cldf.`**`to_cldf`**`(`*wordlist*, *path='cldf'*, *source_path=None*, *ref='cogid'*, *segments='tokens'*, *form='ipa'*, *note='note'*, *form_in_source='value'*, *source=None*, *alignment=None*`)`

Convert a wordlist in LingPy to CLDF.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

A regular Wordlist object (or similar).

path : str (default=cldf)

The name of the directory to which the files will be written.

source_path : str (default=None)

If available, specify the path of your BibTex file with the sources.

ref : str (default=cogid)

The column in which the cognate sets are stored.

segments : str (default=tokens)

The column in which the segmented phonetic strings are stored.

form : str (default=ipa)

The column in which the unsegmented phonetic strings are stored.

note : str (default=None)

The column in which you store your comments.

form_in_source : str (default=None)

The column in which you store the original form in the source.

source : str (default=None)

The column in which you store your source information.

alignment : str (default=alignment)

The column in which you store the alignments.

lingpy.convert.graph module

Conversion routines for the GML format.

`lingpy.convert.graph.`**`gls2gml`**`(`*gls*, *graph*, *tree*, *filename=''*`)`

Create GML-representation of a given gain-loss-scenario (GLS).

Parameters **gls** : list

A list of tuples, indicating the origins of characters along a tree.

graph : networkx.graph

A graph that serves as a template for the plotting of the GLS.

tree : cogent.tree.PhyloNode

A tree object.

`lingpy.convert.graph.`**`igraph2networkx`**`(`*graph*`)`

`lingpy.convert.graph.`**`networkx2igraph`**`(`*graph*`)`

Helper function converts networkx graph to igraph graph object.

`lingpy.convert.graph.`**`nwk2gml`**(*treefile*, *filename=''*)

Function converts a tree in newick format to a network in gml-format.

treefile [str] Either a str defining the path to a file containing the tree in Newick-format, or the tree-string itself.

filename [str (default=lingpy)] The name of the output GML-file. If filename is set to c{None}, the function returns a `Graph`.

Returns graph : networkx.Graph

`lingpy.convert.graph.`**`radial_layout`**(*treestring*, *change=<function <lambda>>*, *degree=100*, *filename=''*, *start=0*, *root='root'*)

Function calculates a simple radial tree layout.

Parameters treefile : str

Either a str defining the path to a file containing the tree in Newick-format, or the tree-string itself.

filename : str (default=None)

The name of the output file (GML-format). If set to c{None}, no output will be written to file.

change : function (default = lambda x:2 * x**2)

The function used to modify the radius in the polar projection of the tree.

Returns graph : networkx.Graph

A graph representation of the tree with coordinates specified in the graphics-attribute of the nodes.

Notes

This function creates a radial tree-layout from a given tree specified in Newick format.

lingpy.convert.html module

Basic functions for HTML-plots.

`lingpy.convert.html.`**`alm2html`**(*infile*, *title=''*, *shorttitle=''*, *filename=''*, *colored=False*, *main_template=''*, *table_template=''*, *dataset=''*, *confidence=False*, ***keywords*)

Convert files in `alm`-format into colored `html`-format.

Parameters title : str

Define the title of the output file. If no title is provided, the default title `LexStat - Automatic Cognate Judgments` will be used.

shorttitle : str

Define the shorttitle of the `html`-page. If no title is provided, the default title `LexStat` will be used.

See also:

`lingpy.convert.html.msa2html`, *`lingpy.convert.html.msa2tex`*

Notes

The coloring of sound segments with respect to the sound class they belong to is based on the definitions given in the `color` *Model*. It can easily be changed and adapted.

`lingpy.convert.html.`**`colorRange`**(*number*, *brightness=300*)

Function returns different colors for the given range.

Notes

Idea taken from http://stackoverflow.com/questions/876853/generating-color-ranges-in-python .

`lingpy.convert.html.`**`msa2html`**(*msa*, *shorttitle="*, *filename="*, *template="*, ***keywords*)

Convert files in `msa`-format into colored `html`-format.

Parameters **msa** : dict

A dictionary object that contains all the information of an MSA object.

shorttitle : str

Define the shorttitle of the `html`-page. If no title is provided, the default title `SCA` will be used.

filename : str (default=)

Define the name of the output file. If no name is defined, the name of the input file will be taken as a default.

template : str (default=)

The path to the template file. If no name is defined, the basic template will be used. The basic template currently used can be found under `lingpy/data/templates/msa2html.html`.

See also:

`lingpy.convert.html.alm2html`

Notes

The coloring of sound segments with respect to the sound class they belong to is based on the definitions given in the `color` *Model*. It can easily be changed and adapted.

Examples

Load the libary.

```
>>> from lingpy import *
```

Load an `msq`-file from the test-sets.

```
>>> msa = MSA('harry.msq')
```

Align the data progressively and carry out a check for swapped sites.

```
>>> msa.prog_align()
>>> msa.swap_check()
>>> print(msa)
w    o    l    -    d    e    m    o    r    t
w    a    l    -    d    e    m    a    r    -
v    -    l    a    d    i    m    i    r    -
```

Save the data to the file `harry.msa`.

```
>>> msa.output('msa',filename='harry')
```

Save the `msa`-object as `html`.

```
>>> msa.output('html',filename='harry')
```

`lingpy.convert.html.`**`msa2tex`**(*infile*, *template=''*, *filename=''*, ***keywords*)
Convert an MSA to a tabular representation which can easily be used in LaTeX documents.

`lingpy.convert.html.`**`psa2html`**(*infile*, ***kw*)
Function converts a PSA-file into colored html-format.

`lingpy.convert.html.`**`string2html`**(*taxon*, *string*, *swaps=[]*, *tax_len=None*)
Function converts an (aligned) string into colored html-format.

@deprecated

`lingpy.convert.html.`**`template_path`**(**comps*)

`lingpy.convert.html.`**`tokens2html`**(*string*, *swaps=[]*, *tax_len=None*)
Function converts an (aligned) string into colored html-format.

Notes

This function is currently not used by any other program. So it might be useful to just deprecate it.

@deprecated

lingpy.convert.plot module

Module provides functions for the transformation of text data into visually appealing format.

`lingpy.convert.plot.`**`plot_concept_evolution`**(*scenarios*, *tree*, *fileformat='pdf'*, *degree=90*, ***keywords*)
Plot the evolution according to the MLN method of all words for a given concept.

Parameters **tree** : str

A tree representation in Newick format.

fileformat : str (default=pdf)

A valid fileformat according to Matplotlib.

degree : int (default=90)

The degree by which the tree is drawn. 360 yields a circular tree, 180 yields a tree filling half of the space of a circle.

`lingpy.convert.plot.`**`plot_gls`**(*gls*, *treestring*, *degree=90*, *fileformat='pdf'*, ***keywords*)
Plot a gain-loss scenario for a given reference tree.

`lingpy.convert.plot.`**`plot_heatmap`**(*wordlist*, *filename='heatmap'*, *fileformat='pdf'*, *ref='cogid'*, *normalized=False*, *refB=''*, ***keywords*)

Create a heatmap-representation of shared cognates for a given wordlist.

Parameters **wordlist** : lingpy.basic.wordlist.Wordlist

A Wordlist object containing cognate IDs.

filename : str (default=heatmap)

Name of the file to which the heatmap will be written.

fileformat : str (default=pdf)

A regular matplotlib-fileformat (pdf, png, pgf, svg).

ref : str (default=cogid)

The name of the column that contains the cognate identifiers.

normalized : {bool str} (default=True)

If set to c{False}, dont normalize the data. Otherwise, select the normalization method, choose between:

- jaccard for the Jaccard-distance (see `Bategelj1995` for details), and
- swadesh for traditional lexicostatistical calculation of shared cognate percentages.

cmap : matplotlib.cm (default=matplotlib.cm.jet)

The color scheme to be used for the heatmap.

steps : int (default=5)

The number of steps in which names of taxa will be written to the axes.

xrotation : int (default=45)

The rotation of the taxon-names on the x-axis.

colorbar : bool (default=True)

Specify, whether a colorbar should be added to the plot.

figsize : tuple (default=(10,10))

Specify the size of the figure.

tree : str (default=)

A tree passed for the taxa in Newick-format. If no tree is specified, the method looks for a tree object in the Wordlist.

Notes

This function plots shared cognate percentages.

`lingpy.convert.plot.`**`plot_tree`**(*treestring*, *degree=90*, *fileformat='pdf'*, *root='root'*, ***keywords*)

Plot a Newick tree to PDF or other graphical formats.

Parameters **treestring** : str

A string in Newick format.

degree : int

Determine the degree of the tree (this determines how circular the tree will be).

fileformat : str (default=pdf)

Select the fileformat to which the tree shall be written.

filename : str

Determine the name of the file to which the data shall be written. Defaults to a timestamp.

figsize : tuple (default=(10,10))

Determine the size of the figure.

lingpy.convert.strings module

Basic functions for the conversion of Python-internal data into strings.

`lingpy.convert.strings.`**`matrix2dst`**(*matrix*, *taxa=None*, *stamp=”*, *filename=”*, *taxlen=10*, *comment=’#’*)

Convert matrix to dst-format.

Parameters **taxa** : {None, list}

List of taxon names corresponding to the distances. Make sure that you only use alphanumeric characters and the understroke for assigning the taxon names. Especially avoid the usage of brackets, since this will confuse many phylogenetic programs.

stamp : str (default=)

Convenience stamp passed as a comment that can be used to indicate how the matrix was created.

filename : str

If you specify a filename, the data will be written to file.

taxlen : int (default=10)

Indicate how long the taxon names are allowed to be. The Phylip package only allows taxon names consisting of maximally 10 characters. Other packages, however, allow more. If Phylip compatibility is not important for you and you just want to allow for as long taxon names as possible, set this value to 0.

comment : str (default = #)

The comment character to be used when adding additional information in the stamp.

Returns **output** : {str or file}

Depending on your settings, this function returns a string in DST (=Phylip) format, or a file containing the string.

`lingpy.convert.strings.`**`msa2str`**(*msa*, *wordlist=False*, *comment=’#’*, *_arange=’{stamp}{comment}\n{meta}{comment}\n{body}’*, *merge=False*)

Function converts an MSA object into a string.

`lingpy.convert.strings.`**`multistate2nex`**(*taxa*, *matrix*, *filename=”*, *missing=’?’*)

Convert the data in a given wordlist to NEXUS-format for multistate analyses in PAUP.

Parameters **taxa** : list

The list of taxa that shall be written to file.

matrix : list

The multi-state matrix with the first dimension indicating the taxa, and the second their states.

filename : str (default=)

If not specified, the filename of the Wordlist will be taken, otherwise, it specifies the name of the file to which the data will be written.

`lingpy.convert.strings.`**`pap2csv`**(*taxa*, *paps*, *filename=''*)

Write paps created by the Wordlist class to a csv-file.

`lingpy.convert.strings.`**`pap2nex`**(*taxa*, *paps*, *missing=0*, *filename=''*, *datatype='STANDARD'*)

Function converts a list of paps into nexus file format.

Parameters **taxa** : list

List of taxa.

paps : {list, dict}

A two-dimensional list with the first dimension being identical to the number of taxa and the second dimension being identical to the number of paps. If a dictionary is passed, each key represents a given pap. The following two structures will thus be treated identically:

```
>>> paps = [[1,0],[1,0],[1,0]] # two languages, three paps
>>> paps = {1:[1,0], 2:[1,0], 3:[1,0]} # two languages, three␣
↪paps
```

missing : {str, int} (default=0)

Indicate how missing characters are represented in the original data.

`lingpy.convert.strings.`**`scorer2str`**(*scorer*)

Convert a scoring function to a string.

`lingpy.convert.strings.`**`template_path`**(**comps*)

`lingpy.convert.strings.`**`write_nexus`**(*wordlist*, *mode='mrbayes'*, *filename='mrbayes.nex'*, *ref='cogid'*, *missing='?'*, *gap='-'*, *custom=None*, *custom_name='lingpy'*, *commands=None*, *commands_name='mrbayes'*)

Write a nexus file for phylogenetic analyses.

Parameters **wordlist** : lingpy.basic.wordlist.Wordlist

A Wordlist object containing cognate IDs.

mode : str (default=mrbayes)

The name of the output nexus style. Valid values are:

- MRBAYES: a MrBayes formatted nexus file.
- SPLITSTREE: a SPLITSTREE formatted nexus file.
- BEAST: a BEAST formatted nexus file.
- **BEASTWORDS: a BEAST formatted nexus for word-partitioned** analyses.
- TRAITLAB: a TRAITLab formatted nexus.

filename : str (default=None)

Name of the file to which the nexus file will be written. If set to c{None}, then this function will not write the nexus ontent to a file, but simply return the content as a string.

ref: str (default=cogid) :

Column in which you store the cognate sets in your data.

gap : str (default=-)

The symbol for gaps (not relevant for linguistic analyses).

missing : str (default=?)

The symbol for missing characters.

custom : list {default=None)

This information allows to add custom information to the nexus file, like, for example, the structure of the characters, their original concept, or their type, and it will be written into a custom block in the nexus file. The name of the custom block can be specified with help of the *custom_name* keyword. The content is a list of strings which will be written line by line into the custom block.

custom_name : str (default=lingpy)

The name of the custom block which will be written to the file.

commands : list (default=None)

If specified, will write an additional block containing commands for phylogenetic software. The commands are passed as a list, containing strings. The name of the block is given by the keywords commands_name.

commands_name : str (default=mrbayes)

Determines how the block will be called to which the commands will be written.

Returns **nexus** : str

A string containing nexus file output

lingpy.convert.tree module

Functions for tree calculations and working with trees.

`lingpy.convert.tree.`**`nwk2tree_matrix`**`(`*newick*`)`

Convert a newick file to a tree matrix.

Notes

This is an additional function that can be used for plots with help of matplotlibs functions. The tree_matrix is compatible with those matrices that scipys linkage functions create.

Module contents

Package provides different methods for file conversion.

lingpy.data package

Subpackages

lingpy.data.ipa package

Submodules

lingpy.data.ipa.sampa module

The regular expression used in the sampa2unicode-converter is taken from an algorithm for the conversion of XSAMPA to IPA (Unicode) by Peter Kleiweg <http://www.let.rug.nl/~kleiweg/L04/devel/python/xsampa.html>. @author: Peter Kleiweg @date: 2007/07/19

`lingpy.data.ipa.sampa.`**`data_path`**`(`**comps*`)`

Module contents

Submodules

lingpy.data.derive module

Module for the derivation of sound class models.

The module provides functions for the customized compilation of sound-class models. All models are defined in simple text files. In order to guarantee their quick access when loading the library, the models are compiled and stored in binary files.

`lingpy.data.derive.`**`compile_dvt`**`(`*path=''*`)`

Function compiles diacritics, vowels, and tones.

See also:

`lingpy.data.model.Model`, *`lingpy.data.derive.compile_model`*

Notes

Diacritics, vowels, and tones are defined in the `data/models/dv/` directory of the LingPy package and automatically loaded when loading the LingPy library. The values are defined as the constants `rcParams['vowels']`, `rcParams['diacritics']`, and `rcParams['tones']`. Their core purpose is to guide the tokenization of IPA strings (cf. *`ipa2tokens()`*). In order to change the variables, one simply has to change the text files `diacritics`, `tones`, and `vowels` in the `data/models/dv` directory. The structure of these files is fairly simple: Each line contains a vowel or a diacritic character, whereas diacritics are preceded by a dash.

`lingpy.data.derive.`**`compile_model`**`(`*model*, *path=None*`)`

Function compiles customized sound-class models.

Parameters **model** : str

A string indicating the name of the model which shall be created.

path : str

A string indication the path where the model-folder is stored.

See also:

lingpy.data.model.Model, *compile_dvt*

Notes

A model is defined by a folder placed in `data/models` directory of the LingPy package. The name of the folder reflects the name of the model. It contains three files: the file `converter`, the file `INFO`, and the optional file `scorer`. The format requirements for these files are as follows:

INFO The `INFO`-file serves as a reference for a given sound-class model. It can contain arbitrary information (and also be empty). If one wants to define specific characteristics, like the `source`, the `compiler`, the `date`, or a `description` of a given model, this can be done by employing a key-value structure in which the key is preceded by an `@` and followed by a colon and the value is written right next to the key in the same line, e.g.:

```
@source: Dolgopolsky (1986)
```

This information will then be read from the `INFO` file and rendered when printing the model to screen with help of the `print()` function.

converter The `converter` file contains all sound classes which are matched with their respective sound values. Each line is reserved for one class, precede by the key (preferably an ASCII-letter) representing the class:

```
B : , β, f, pf, pf,
E : , æ, , , , e, , , , , è, é, , , ê,
D : θ, ð, , þ,
G : x, , χ
...
```

matrix A scoring matrix indicating the alignment scores of all sound-class characters defined by the model. The scoring is structured as a simple tab-delimited text file. The first cell contains the character names, the following cells contain the scores in redundant form (with both triangles being filled):

```
B  10.0 -10.0   5.0 ...
E -10.0   5.0 -10.0 ...
F   5.0 -10.0  10.0 ...
...
```

scorer The `scorer` file (which is optional) contains the graph of class-transitions which is used for the calculation of the scoring dictionary. Each class is listed in a separate line, followed by the symbols `v`,‘‘c‘‘, or `t` (indicating whether the class represents vowels, consonants, or tones), and by the classes it is directly connected to. The strength of this connection is indicated by digits (the smaller the value, the shorter the path between the classes):

```
A : v, E:1, O:1
C : c, S:2
B : c, W:2
E : v, A:1, I:1
D : c, S:2
...
```

The information in such a file is automatically converted into a scoring dictionary (see `List2012b` for details).

Based on the information provided by the files, a dictionary for the conversion of IPA-characters to sound classes and a scoring dictionary are created and stored as a binary. The model can be loaded with help of the *Model* class and used in the various classes and functions provided by the library.

lingpy.data.model module

Module for handling sequence models.

class lingpy.data.model.**Model**(*model*, *path=None*)

Bases: `object`

Class for the handling of sound-class models.

Parameters model : { sca, dolgo, asjp, art, _color }

A string indicating the name of the model which shall be loaded. Select between:

- sca - the SCA sound-class model (see `List2012a`),
- dolgo - the DOLGO sound-class model (see: :evobib:'Dolgopolsky1986),
- asjp - the ASJP sound-class model (see `Brown2008` and `Brown2011`),
- art - the sound-class model which is used for the calculation of sonority profiles and prosodic strings (see `List2012`), and
- _color - the sound-class model which is used for the coloring of sound-tokens when creating html-output.

See also:

lingpy.data.derive.compile_model, *lingpy.data.derive.compile_dvt*

Notes

Models are loaded from binary files which can be found in the `data/models/` folder of the LingPy package. A model has two essential attributes:

- `converter` – a dictionary with IPA-tokens as keys and sound-class characters as values, and
- `scorer` – a scoring dictionary with tuples of sound-class characters as keys and scores (integers or floats) as values.

Examples

When loading LingPy, the models `sca`, `asjp`, `dolgo`, and `art` are automatically loaded, and they are accessible via the *rc()* function for global settings:

```
>>> from lingpy import *
>>> rc('asjp')
<sca-model "asjp">
```

Define variables for the standard models for convenience:

```
>>> asjp = rc('asjp')
>>> sca = rc('sca')
>>> dolgo = rc('dolgo')
>>> art = rc('art')
```

Check how the letter `a` is converted in the various models:

```
>>> for m in [asjp,sca,dolgo,art]:
...     print('{0} > {1} ({2})'.format('a',m.converter['a'],m.name))
...
a > a (asjp)
a > A (sca)
a > V (dolgo)
a > 7 (art)
```

Retrieve basic information of a given model:

```
>>> print(sca)
Model:    sca
Info:     Extended sound class model based on Dolgopolsky (1986)
Source:   List (2012)
Compiler: Johann-Mattis List
Date:     2012-03
```

Attributes

con-verter	dict	A dictionary with IPA tokens as keys and sound-class characters as values.
scorer	dict	A scoring dictionary with tuples of sound-class characters as keys and similarity scores as values.
info	dict	A dictionary storing the key-value pairs defined in the `INFO`.
name	str	The name of the model which is identical with the name of the folder from wich the model is loaded.

`lingpy.data.model.`**`load_dvt`**`(`*path=''*`)`

Function loads the default characters for IPA diacritics and IPA vowels of LingPy.

Module contents

LingPy comes along with many different kinds of predefined data. When loading the library, the following dictionary is automatically loaded and employed by all LingPy modules:

`rcParams : dict`

As an alternative to all global variables, this dictionary contains all these variables, and additional ones. This dictionary is used for internal coding purposes and stores parameters that are globally set (if not defined otherwise by the user), such as

- specific debugging messages (warnings, messages, errors)
- default values, such as gop (gap opening penalty), scale (scaling factor
- by which extended gaps are penalized), or figsize (the default size of
- figures if data is plotted using matplotlib).

These default values can be changed with help of the `rc` function that takes any keyword and any variable as input and adds or modifies the specific key of the rcParams dictionary, but also provides more complex functions that change whole sets of variables, such as the following statement:

```
>>> rc(schema="asjp")
```

which switches the variables asjp, dolgo, etc. to the ASCII-based transcription system of the ASJP project.

If you want to change the content of c{rcParams} directly, you need to import the dictionary explicitly:

```
>>> from lingpy.settings import rcParams
```

However, changing the values in the dictionary randomly can produce unexpected behavior and we recommend to use the regular `rc` function for this purpose.

`lingpy.settings.`**`rc`**(*rval=None*, ***keywords*)

Function changes parameters globally set for LingPy sessions.

Parameters **rval** : string (default=None)

Use this keyword to specify a return-value for the rc-function.

schema : {ipa, asjp}

Change the basic schema for sequence comparison. When switching to asjp, this means that sequences will be treated as sequences in ASJP code, otherwise, they will be treated as sequences written in basic IPA.

Notes

This function is the standard way to communicate with the *rcParams* dictionary which is not imported as a default. If you want to see which parameters there are, you can load the rcParams dictonary directly:

```
>>> from lingpy.settings import rcParams
```

However, be careful when changing the values. They might produce some unexpected behavior.

Examples

Import LingPy:

```
>>> from lingpy import *
```

Switch from IPA transcriptions to ASJP transcriptions:

```
>>> rc(schema="asjp")
```

You can check which basic orthography is currently loaded:

```
>>> rc(basic_orthography)
'asjp'
>>> rc(schema='ipa')
>>> rc(basic_orthography)
'fuzzy'
```

lingpy.evaluate package

Submodules

lingpy.evaluate.acd module

Evaluation methods for automatic cognate detection.

`lingpy.evaluate.acd.`**`bcubes`**(*wordlist*, *gold='cogid'*, *test='lexstatid'*, *modify_ref=False*, *pprint=True*, *per_concept=False*)

Compute B-Cubed scores for test and reference datasets.

Parameters **lex** : `lingpy.basic.wordlist.Wordlist`

A `lingpy.basic.wordlist.Wordlist` class or a daughter class, (like the `LexStat` class used for the computation). It should have two columns indicating cognate IDs.

gold : str (default=cogid)

The name of the column containing the gold standard cognate assignments.

test : str (default=lexstatid)

The name of the column containing the automatically implemented cognate assignments.

modify_ref : function (default=False)

Use a function to modify the reference. If your cognate identifiers are numerical, for example, and negative values are assigned as loans, but you want to suppress this behaviour, just set this keyword to abs, and all cognate IDs will be converted to their absolute value.

pprint : bool (default=True)

Print out the results

per_concept : bool (default=False)

Compute b-cubed scores per concep and not for the whole data in one piece.

Returns **t** : tuple

A tuple consisting of the precision, the recall, and the harmonic mean (F-scores).

See also:

`diff`, `pairs`

Notes

B-Cubed scores were first described by `Bagga1998` as part of an algorithm. Later on, `Amigo2009` showed that they can also used as to compare cluster decisions. `Hauer2011` applied the B-Cubed scores first to the task of automatic cognate detection.

`lingpy.evaluate.acd.`**`diff`**(*wordlist*, *gold='cogid'*, *test='lexstatid'*, *modify_ref=False*, *pprint=True*, *filename=''*, *tofile=True*, *transcription='ipa'*, *concepts=False*)

Write differences in classifications on an item-basis to file.

lex [`lingpy.compare.lexstat.LexStat`] The `LexStat` class used for the computation. It should have two columns indicating cognate IDs.

gold [str (default=cogid)] The name of the column containing the gold standard cognate assignments.

test [str (default=lexstatid)] The name of the column containing the automatically implemented cognate assignments.

modify_ref [function (default=False)] Use a function to modify the reference. If your cognate identifiers are numerical, for example, and negative values are assigned as loans, but you want to suppress this behaviour, just set this keyword to abs, and all cognate IDs will be converted to their absolute value.

pprint [bool (default=True)] Print out the results

filename [str (default=)] Name of the output file. If not specified, it is identical with the name of the `LexStat`, but with the extension `diff`.

tofile [bool (default=True)] If set to c{False}, no data will be written to file, but instead, the data will be returned.

transcription [str (default=ipa)] The file in which the transcriptions are located (should be a string, no segmentized version, for convenience of writing to file).

Returns **t** : tuple

A nested tuple consisting of two further tuples. The first containing precision, recall, and harmonic mean (F-scores), the second containing the same values for the pair-scores.

See also:

`bcubes`, `pairs`

Notes

If the **tofile** option is chosen, the results are written to a specific file with the extension `diff`. This file contains all cognate sets in which there are differences between gold standard and test sets. It also gives detailed information regarding false positives, false negatives, and the words involved in these wrong decisions.

`lingpy.evaluate.acd.`**`extreme_cognates`**(*wordlist*, *ref='extremeid'*, *bias='lumper'*)

Return extreme cognates, either lump all words together or split them.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

A ~lingpy.basic.wordlist.Wordlist object.

ref : str (default=extremeid)

The name of the table in your wordlist to which the new IDs should be written.

bias : str (default=lumper)

If set to lumper, all words with a certain meaning will be given the same cognate set ID, if set to splitter, all will be given a separate ID.

`lingpy.evaluate.acd.`**`npoint_ap`**(*scores*, *cognates*, *reverse=False*)

Calculate the n-point average precision.

Parameters **scores** : list

The scores of your algorithm for pairwise string comparison.

cognates : list

The cognate codings of the word pairs you compared. 1 indicates that the pair is cognate, 0 indicates that it is not cognate.

reverse : bool (default=False)

The order of your ranking mechanism. If your algorithm yields high scores for words which are probably cognate, and low scores for non-cognate words, you should set this keyword to True.

Notes

This follows the description in Kondrak2002. The n-point average precision is useful to compare the discriminative force of different algorithms for string similarity, or to train the parameters of a given algorithm.

Examples

```
>>> scores = [1, 2, 3, 4, 5]
>>> cognates = [1, 1, 1, 0, 0]
>>> from lingpy.evaluate.acd import npoint_ap
>>> npoint_ap(scores, cognates)
1.0
```

`lingpy.evaluate.acd.`**`pairs`**(*lex*, *gold='cogid'*, *test='lexstatid'*, *modify_ref=False*, *pprint=True*, *_return_string=False*)

Compute pair scores for the evaluation of cognate detection algorithms.

Parameters **lex** : *lingpy.compare.lexstat.LexStat*

The *LexStat* class used for the computation. It should have two columns indicating cognate IDs.

gold : str (default=cogid)

The name of the column containing the gold standard cognate assignments.

test : str (default=lexstatid)

The name of the column containing the automatically implemented cognate assignments.

modify_ref : function (default=False)

Use a function to modify the reference. If your cognate identifiers are numerical, for example, and negative values are assigned as loans, but you want to suppress this behaviour, just set this keyword to abs, and all cognate IDs will be converted to their absolute value.

pprint : bool (default=True)

Print out the results

Returns **t** : tuple

A tuple consisting of the precision, the recall, and the harmonic mean (F-scores).

See also:

diff, *bcubes*

Notes

Pair-scores can be computed in different ways, with often different results. This variant follows the description by Bouchard-Cote2013.

`lingpy.evaluate.acd.`**`partial_bcubes`**(*wordlist*, *gold*, *test*, *pprint=True*)

Compute B-Cubed scores for test and reference datasets for partial cognate detection.

Parameters **wordlist** : `Wordlist`

A `Wordlist`, or one of its daughter classes (like, e.g., the `Partial` class used for computation of partial cognates. It should have two columns indicating cognate IDs.

gold : str (default=cogid)

The name of the column containing the gold standard cognate assignments.

test : str (default=lexstatid)

The name of the column containing the automatically implemented cognate assignments.

pprint : bool (default=True)

Print out the results

Returns **t** : tuple

A tuple consisting of the precision, the recall, and the harmonic mean (F-scores).

See also:

`bcubes`, `diff`, `pairs`

Notes

B-Cubed scores were first described by `Bagga1998` as part of an algorithm. Later on, `Amigo2009` showed that they can also used as to compare cluster decisions. `Hauer2011` applied the B-Cubed scores first to the task of automatic cognate detection.

`lingpy.evaluate.acd.`**`random_cognates`**(*wordlist*, *ref='randomid'*, *bias=False*)

Populate a wordlist with random cognates for each entry.

Parameters **ref** : str (default=randomid)

Cognate set identifier for the newly created random cognate sets.

bias : str (default=False)

When set to lumper this will tend to create less cognate sets and larger clusters, when set to splitter it will tend to create smaller clusters.

Notes

When using this method for evaluation, you should be careful to overestimate the results. The function which creates the random clusters is based on simple functions for randomization and thus probably

lingpy.evaluate.alr module

Module provides methods for the evaluation of automatic linguistic reconstruction analyses.

`lingpy.evaluate.alr.`**`mean_edit_distance`**(*wordlist*, *gold='proto'*, *test='consensus'*, *ref='cogid'*, *tokens=True*, *classes=False*, ***keywords*)

Function computes the edit distance between gold standard and test set.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

The wordlist object containing the data for a given analysis.

gold : str (default=proto)

The name of the column containing the gold-standard solutions.

test = consensus :

The name of the column containing the test solutions.

stress : str (default=rcParams[stress])

A string containing the stress symbols used in the sound-class conversion. Defaults to the stress as defined in ~lingpy.settings.rcParams.

diacritics : str (default=rcParams[diacritics])

A string containing diacritic symbols used in the sound-class conversion. Defaults to the diacritic symbolds defined in ~lingpy.settings.rcParams.

cldf : bool (default=False)

If set to True, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a `` `source/target` `` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Returns **dist** : float

The mean edit distance between gold and test reconstructions.

Notes

This function has an alias (med). Calling it will produce the same results.

`lingpy.evaluate.alr.`**`med`**(*wordlist*, ***keywords*)

lingpy.evaluate.apa module

Basic module for the comparison of automatic phonetic alignments.

`class` `lingpy.evaluate.apa.`**`Eval`**(*gold*, *test*)

Bases: `object`

Base class for evaluation objects.

`class` `lingpy.evaluate.apa.`**`EvalMSA`**(*gold*, *test*)

Bases: *`lingpy.evaluate.apa.Eval`*

Base class for the evaluation of automatic multiple sequence analyses.

Parameters **gold, test** : *`MSA`*

The `Multiple` objects which shall be compared. The first object should be the gold standard and the second object should be the test set.

Notes

Most of the scores which can be calculated with help of this class are standard evaluation scores in evolutionary biology. For a close description on how these scores are calculated, see, for example, Thompson1999, List2012, and Rosenberg2009b.

c_score(*mode=1*)
Calculate the column (C) score.

Parameters **mode** : { 1, 2, 3, 4 }

Indicate, which mode to compute. Select between:

1. divide the number of common columns in reference and test alignment by the total number of columns in the test alignment (the traditional C score described in Thompson1999, also known as precision score in applications of information retrieval),
2. divide the number of common columns in reference and test alignment by the total number of columns in the reference alignment (also known as recall score in applications of information retrieval),
3. divide the number of common columns in reference and test alignment by the average number of columns in reference and test alignment, or
4. combine the scores of mode `1` and mode `2` by computing their F-score, using the formula $2 * \frac{pr}{p+r}$, where *p* is the precision (mode `1`) and *r* is the recall (mode `2`).

Returns **score** : float

The C score for reference and test alignments.

Notes

The different c-

c_scores
Calculate the c-scores.

check_swaps()
Check for possibly identical swapped sites.

Returns **swap** : { -2, -1, 0, 1, 2 }

Information regarding the identity of swap decisions is coded by integers, whereas

1 – indicates that swaps are detected in both gold standard and testset, whereas a negative value indicates that the positions are not identical,

2 – indicates that swap decisions are not identical in gold standard and testset, whereas a negative value indicates that there is a false positive in the testset, and

0 – indicates that there are no swaps in the gold standard and the testset.

jc_score()
Calculate the Jaccard (JC) score.

Returns **score** : float

The JC score.

See also:

`lingpy.test.evaluate.EvalPSA.jc_score`

Notes

The Jaccard score (see `List2012`) is calculated by dividing the size of the intersection of residue pairs in reference and test alignment by the size of the union of residue pairs in reference and test alignment.

r_score()
Compute the rows (R) score.

Returns **score** : float

The PIR score.

Notes

The R score is the number of identical rows (sequences) in reference and test alignment divided by the total number of rows.

sp_score(*mode=1*)
Calculate the sum-of-pairs (SP) score.

Parameters **mode** : { 1, 2, 3 }

Indicate, which mode to compute. Select between:

1. divide the number of common residue pairs in reference and test alignment by the total number of residue pairs in the test alignment (the traditional SP score described in `Thompson1999`, also known as precision score in applications of information retrieval),
2. divide the number of common residue pairs in reference and test alignment by the total number of residue pairs in the reference alignment (also known as recall score in applications of information retrieval),
3. divide the number of common residue pairs in reference and test alignment by the average number of residue pairs in reference and test alignment.

Returns **score** : float

The SP score for gold standard and test alignments.

Notes

The SP score (see `Thompson1999`) is calculated by dividing the number of identical residue pairs in reference and test alignment by the total number of residue pairs in the reference alignment.

class lingpy.evaluate.apa.**EvalPSA**(*gold*, *test*)
Bases: *`lingpy.evaluate.apa.Eval`*

Base class for the evaluation of automatic pairwise sequence analyses.

Parameters **gold, test** : *`lingpy.align.sca.PSA`*

The `Pairwise` objects which shall be compared. The first object should be the gold standard and the second object should be the test set.

Notes

Moste of the scores which can be calculated with help of this class are standard evaluation scores in evolutionary biology. For a close description on how these scores are calculated, see, for example, Thompson1999, List2012, and Rosenberg2009b.

c_score()
Calculate column (C) score.

Returns **score** : float

The C score for reference and test alignments.

Notes

The C score, as it is described in Thompson1999, is calculated by dividing the number of columns which are identical in the gold standarad and the test alignment by the total number of columns in the test alignment.

diff(***keywords*)
Write all differences between two sets to a file.

Parameters **filename** : str (default=eval_psa_diff)

Default

jc_score()
Calculate the Jaccard (JC) score.

Returns **score** : float

The JC score.

Notes

The Jaccard score (see List2012) is calculated by dividing the size of the intersection of residue pairs in reference and test alignment by the size of the union of residue pairs in reference and test alignment.

pairwise_column_scores
Compute the different column scores for pairwise alignments. The method returns the precision, the recall score, and the f-score, following the proposal of Bergsma and Kondrak (2007), and the column score proposed by Thompson et al. (1999).

r_score(*mode=1*)
Compute the percentage of identical rows (PIR) score.

Parameters **mode** : { 1, 2 }

Select between mode `1`, where all sequences are compared with each other, and mode `2`, where only whole alignments are compared.

Returns **score** : float

The PIR score.

Notes

The PIR score is the number of identical rows (sequences) in reference and test alignment divided by the total number of rows.

sp_score()

Calculate the sum-of-pairs (SP) score.

Returns score : float

The SP score for reference and test alignments.

Notes

The SP score (see Thompson1999) is calculated by dividing the number of identical residue pairs in reference and test alignment by the total number of residue pairs in the reference alignment.

Module contents

Basic module for the evaluation of algorithms.

lingpy.meaning package

Submodules

lingpy.meaning.colexification module

Module offers methods to handle colexification patterns in wordlist objects.

lingpy.meaning.colexification.**colexification_network**(*wordlist*, *entry='ipa'*, *concept='concept'*, *output=''*, *filename='network'*, *bipartite=False*, ***keywords*)

Calculate a colexification network from a given wordlist object.

Parameters wordlist : ~lingpy.basic.wordlist.Wordlist

The wordlist object containing the data.

entry : str (default=ipa)

The reference point for the language entry. We use ipa as a default.

concept : str (default=concept)

The reference point for the name of the row containing the concepts. We use concept as a default.

output: str (default=) :

If output is set to gml, the resulting network will be written to a textfile in GML format.

Returns G : networkx.Graph

A networkx.Graph object.

lingpy.meaning.colexification.**compare_colexifications**(*wordlist*, *entry='ipa'*, *concept='concept'*)

Compare colexification patterns for a given wordlist.

lingpy.meaning.colexification.**dotjoin**(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.meaning.colexification.`**`evaluate_colexifications`**(*G*, *weight='wordWeight'*, *outfile=None*)

Function calculates most frequent colexifications in a wordlist.

Module contents

lingpy.read package

Submodules

lingpy.read.csv module

Module provides functions for reading csv-files.

`lingpy.read.csv.`**`csv2dict`**(*filename*, *fileformat=None*, *dtype=None*, *comment='#'*, *sep='\t'*, *strip_lines=True*, *header=False*)

Very simple function to get quick access to CSV-files.

Parameters **filename** : str

Name of the input file.

fileformat : {None str}

If not specified the file <filename> will be loaded. Otherwise, the fileformat is interpreted as the specific extension of the input file.

dtype : {None list}

If not specified, all data will be loaded as strings. Otherwise, a list specifying the data for each line should be provided.

comment : string (default=#)

Comment character in the begin of a line forces this line to be ignored.

sep : string (default =)

Specify the separator for the CSV-file.

strip_lines : bool (default=True)

Specify whether empty cells in the input file should be preserved. If set to c{False}, each line will be stripped first, and all whitespace will be cleaned. Otherwise, each line will be separated using the specified separator, and no stripping of whitespace will be carried out.

header : bool (default=False)

Indicate, whether the data comes along with a header.

Returns **d** : dict

A dictionary-representation of the CSV file, with the first row being used as key and the rest of the rows as values.

`lingpy.read.csv.`**`csv2list`**(*filename*, *fileformat=''*, *dtype=None*, *comment='#'*, *sep='\t'*, *strip_lines=True*, *header=False*)

Very simple function to get quick (and somewhat naive) access to CSV-files.

Parameters **filename** : str

Name of the input file.

fileformat : {None str}

If not specified the file <filename> will be loaded. Otherwise, the fileformat is interpreted as the specific extension of the input file.

dtype : {list}

If not specified, all data will be loaded as strings. Otherwise, a list specifying the data for each line should be provided.

comment : string (default=#)

Comment character in the begin of a line forces this line to be ignored (set to None if you want to parse all lines of your file).

sep : string (default =)

Specify the separator for the CSV-file.

strip_lines : bool (default=True)

Specify whether empty cells in the input file should be preserved. If set to c{False}, each line will be stripped first, and all whitespace will be cleaned. Otherwise, each line will be separated using the specified separator, and no stripping of whitespace will be carried out.

header : bool (default=False)

Indicate, whether the data comes along with a header.

Returns **l** : list

A list-representation of the CSV file.

`lingpy.read.csv.`**`csv2multidict`**(*filename*, *comment='#'*, *sep='\t'*)

Function reads a csv-file into a multi-dimensional dictionary structure.

`lingpy.read.csv.`**`read_asjp`**(*infile*, *family='Indo-European'*, *classification='hh'*, *max_synonyms=2*, *min_population=<function <lambda>>*, *merge_vowels=True*, *evaluate=False*)

lingpy.read.phylip module

Module provides functions to read in various formats from the Phylip package.

`lingpy.read.phylip.`**`read_dst`**(*filename*, *taxlen=10*, *comment='#'*)

Function reads files in Phylip dst-format.

Parameters **filename** : string

Name of the file which should have the extension `dst`.

taxlen : int (default=10)

Indicate how long the taxon names are allowed to be in the file from which you want to read. The Phylip package only allows taxon names consisting of maximally 10 characters (this is the default). Other packages, however, allow more. If Phylip compatibility is not important for you and you just want to allow for as long taxon names as possible, set this value to 0 and make sure to use tabstops as separators between values in your matrix file.

comment : str (default = #)

The comment character to be used if your file contains additional information which should be ignored.

Returns data : tuple

A tuple consisting of a list of taxa and a matrix.

`lingpy.read.phylip.`**`read_scorer`**`(`*infile*`)`

Read a scoring function in a file into a ScoreDict object.

Parameters infile : str

The path to the input file that shall be read as a scoring dictionary. The matrix format is a simple csv-file in which the scoring matrix is displayed, with negative values indicating high differences between sound segments (or sound classes) and positive values indicating high similarity. The matrix should be symmetric, columns should be separated by tabstops, and the first column should provide the alphabet for which the scoring function is defined.

Returns scoredict : ~lingpy.algorithm.misc.ScoreDict

A ScoreDict instance which can be directly passed to LingPys alignment functions.

lingpy.read.qlc module

`lingpy.read.qlc.`**`normalize_alignment`**`(`*alignment*`)`

Function normalizes an alignment.

Normalization here means that columns consisting only of gaps will be deleted, and all sequences will be stretched to equal length by adding additional gap characters in the end of smaller sequences.

`lingpy.read.qlc.`**`read_msa`**`(`*infile*, *comment='#'*, *ids=False*, *header=True*, *normalize=True*, ***keywords*`)`

Simple function to load an MSA object.

Parameters infile : str

The name of the input file.

comment : str (default=#)

The comment character. If a line starts with this character, it will be ignored.

ids : bool (default=False)

Indicate whether the MSA file contains unique IDs for all sequences or not.

Returns d : dict

A dictionary in which keys correspond to specific parts of a multiple alignment. This dictionary can be directly passed to alignment functions, such as `lingpy.sca.MSA`.

`lingpy.read.qlc.`**`read_qlc`**`(`*infile*, *comment='#'*`)`

Simple function that loads qlc-format into a dictionary.

Parameters infile : str

The name of the input file.

comment : str (default=#)

The comment character. If a line starts with this character, it will be ignored.

Returns **d** : dict

A dictionary with integer keys corresponding to the order of the lines of the input file. The header is given 0 as a specific key.

`lingpy.read.qlc.`**`reduce_alignment`**(*alignment*)

Function reduces a given alignment.

Notes

Reduction here means that the output alignment consists only of those parts which have not been marked to be ignored by the user (parts in brackets). It requires that all data is properly coded. If reduction fails, this will throw a warning, and all brackets are simply removed in the output alignment.

lingpy.read.starling module

Basic parser for Starling data.

`lingpy.read.starling.`**`star2qlc`**(*filename*, *clean_taxnames=False*, *debug=False*)

Converts a file directly output from starling to LingPy-QLC format.

Module contents

lingpy.sequence package

Submodules

lingpy.sequence.generate module

Module provides simple basic classes for sequence generation using Markov models.

`class` `lingpy.sequence.generate.`**`MCBasic`**(*seqs*)

Bases: `object`

Basic class for creating Markov chains from sequence training data.

Parameters **seq** : list

A list of sequences. Sequences are assumed to be tokenized, i.e. they should be either passed as lists or as tuples.

`walk`()

Create random sequence from the distribution.

`class` `lingpy.sequence.generate.`**`MCPhon`**(*words*, *tokens=False*, *prostrings=[]*, *classes=False*, *class_model=<sca-model "sca">*, ***keywords*)

Bases: *`lingpy.sequence.generate.MCBasic`*

Class for the creation of phonetic sequences (pseudo words).

Parameters **words** : list

List of phonetic sequences. This list can contain tokenized sequences (lists or tuples), or simple untokenized IPA strings.

tokens : bool (default=False)

If set to True, no tokenization of input sequences is carried out.

prostring : list (default=[])

List containing the prosodic profiles of the input sequences. If the list is empty, the profiles are generated automatically.

evaluate_string(*string*, *tokens=False*, ***keywords*)

get_string(*new=True*, *tokens=False*)

Generate a string from the Markov chain created from the training data.

Parameters **new** : bool (default=True)

Determine whether the string created should be different from the training data or not.

tokens : bool (default=False)

If set to *True* he full list of tokens that was internally used to represent the sequences as a Markov chain is returned.

lingpy.sequence.ngrams module

This modules provides methods for generating and collecting ngrams.

The methods allow to collect different kind of subsequences, such as standard ngrams (preceding context), skip ngrams with both single or multiple gap openings (both preceding and following context), and positional ngrams (both preceding and following context).

class lingpy.sequence.ngrams.**NgramModel**(*pre_order=0*, *post_order=0*, *pad_symbol='$$$'*, *sequences=None*)

Bases: object

Class for operation upon sequences using ngrams models.

This class allows different operations upon sequences after training ngram models, such as sequence relative likelihood computation (both per state and overall), random sequence generation, computation of a model entropy and of cross-entropy/perplexity of a sequence, etc. As model training is computationally and time consuming for large datasets, trained models can be saved and loaded (serialized) from disk.

add_sequences(*sequences*)

Adds sequences to a model, collecting their ngrams.

This method does not return any value, but cleans the internal matrix probability, if one was previously computed, and automatically updates the ngram counters. The actual training, with the computation of smoothed log-probabilities, is not performed automatically, and must be requested by the user by calling the *.train()* method.

Parameters **sequences: list** :

A list of sequences to be added to the model.

entropy(*sequence*, *base=2.0*)

Calculates the cross-entropy of a sequence.

Parameters **sequence: list** :

The sequence whose cross-entropy will be calculated.

base: float :

The logarithmic base for the cross-entropy calculation. Defaults to 2.0, following the standard approach set by Shannon that allows to consider entropy in terms of bits needed for unique representation.

Returns ch: float :

The cross-entropy calculated for the sequence, a real number.

model_entropy()

Return the model entropy.

This methods collects the P x log(P) for all contexts, returning their sum. This is different from a sequence cross-entropy, and should be used to estimate the complexity of a model.

Please note that for very large models the computation of this entropy might run into underflow problems.

Returns h: float :

The model entropy.

perplexity(*sequence*)

Calculates the perplexity of a model.

As per definition, this is simply 2.0 to the cross-entropy of the given sequence on logarithmic base of 2.0.

Parameters sequence: list :

The sequence whose perplexity should be calculated.

Returns perplexity: float :

The calculated perplexity for the sequence.

random_seqs(*k=1*, *seq_len=None*, *scale=2*, *only_longest=False*, *attempts=10*, *seed=None*)

Return a set of random sequences based in the observed transition frequencies.

This function tries to generate a set of *k* random sequences from the internal model. Given that the random selection and the parameters might lead to a long or infinite search loop, the number of attempts for each word generation is limited, meaning that there is no guarantee that the returned list will be of length *k*, but only that it will be at most of length *k*.

Parameters k: int :

The desired and maximum number of random sequences to be returned. While the algorithm should be robust enough for most cases, there is no guarantee that the desired number or even that a single random sequence will be returned. In case of missing sequences, try increasing the parameter *attempts*.

seq_len: int or list :

An optional integer with length of the sequences to be generated or a list of lengths to be uniformly drawn for the generated sequences. If the parameter is not specified, the length of the sequences will be drawn by the sequence lengths observed in training according to their frequencies.

scale: numeric :

The exponent used for weighting ngram probabilities according to their length in number of states. The higher this value, the less likely the algorithm will be to drawn shorter ngrams, which contribute to a higher variety in words but can also result in less likely sequences. Defaults to 2.

only_longest: bool :

Whether the algorithm should only collect the longest possible ngrams when computing the search space from which each new random character is obtained. This usually translates into less variation in the generated sequences and a longer searching time, which might need to be increased via the *attempts* parameters. Defaults to False.

tries: int :

The number of times the algorithm will try to generate a random sequence. If the algorithm is unable to generate a suitable random sequence after the specified number of *attempts*, the loop will advance to the following sequence (if any). Defaults to 10.

seed: obj :

Any hasheable object, used to feed the random number generator and thus reproduce the generated set of random sequences.

Returns seqs: list :

A list of size *k* with random sequences.

score (*sequence*, *use_length=True*)

Returns the relative likelihood of a sequence.

The model must have been trained before using this function.

Parameters sequence: list :

A list of states to be scored.

use_length: bool :

Whether to correct the sequence relative likelihood by using length probability. Defaults to True.

Returns prob: list :

A list of floats, of the same length of *sequence*, with the individual log-probability for each state.

state_score (*sequence*)

Returns the relative likelihood for each state in a sequence.

Please note that this does not perform correction due to sequence length, as optionally and by default performed by the *.score()* method. The model must have been trained in advance.

Parameters sequence: list :

A list of states to be scored.

Returns prob: list :

A list of floats, of the same length of *sequence*, with the individual log-probability for each state.

train (*method='laplace'*, *normalize=False*, *bins=None*, ***kwargs*)

Train a model after ngrams have been collected.

This method does not return any value, but sets the internal variables with smoothed probabilities (such as *self._p* and *self._p0*) and internally marks the model as having been trained.

Parameters method: str :

The name of the smoothing method to be used, as used by *smooth_dist()*. Either uniform, random, mle, lidstone, laplace, ele, wittenbell, certaintydegree, or sgt. Defaults to laplace.

normalize: boolean :

Whether to normalize the log-probabilities for each ngram in the model after smoothing, i.e., to guarantee that the probabilities (with the probability for unobserved transitions counted a single time) sum to 1.0. This is computationally expansive, and should be only used if the model is intended for later serialization. While experiments with real data demonstrated that this normalization does not improve the results or performance of the methods, the computational cost of normalizing the probabilities might be justified if descriptive statistics of the model, like samples from the matrix of transition probabilities or the entropy/perplexity of a sequence, are needed (such as for publication), as they will be more in line with what is generally expected and will facilitate the comparison of different models.

bins: int :

The number of bins to be assumed when smoothing, for the smoothing methods that use this information. Defaults to the number of unique states observed, as gathered from the count of ngrams with no context.

`lingpy.sequence.ngrams.`**`bigrams`**(*sequence*, *, *order=2*, *pad_symbol='$$$'*)

Build an iterator for collecting all bigrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the bigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the bigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in bigrams(sent):
...     print(ngram)
...
('$$$', 'Insurgents')
('Insurgents', 'killed')
('killed', 'in')
('in', 'ongoing')
('ongoing', 'fighting')
('fighting', '$$$')
```

`lingpy.sequence.ngrams.`**`confirm`**(*question*, *, *default=False*)

Ask a yes/no question interactively.

Parameters **question** – The text of the question to ask.

Returns True if the answer was yes, False otherwise.

`lingpy.sequence.ngrams.`**`data_path`**`(`**comps*`)`

`lingpy.sequence.ngrams.`**`dotjoin`**`(`**args*, ***kw*`)`

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.sequence.ngrams.`**`fourgrams`**`(`*sequence*, ***, *order=4*, *pad_symbol='$$$'*`)`

Build an iterator for collecting all fourgrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the fourgrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the fourgrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in fourgrams(sent):
...     print(ngram)
...
('$$$', '$$$', '$$$', 'Insurgents')
('$$$', '$$$', 'Insurgents', 'killed')
('$$$', 'Insurgents', 'killed', 'in')
('Insurgents', 'killed', 'in', 'ongoing')
('killed', 'in', 'ongoing', 'fighting')
('in', 'ongoing', 'fighting', '$$$')
('ongoing', 'fighting', '$$$', '$$$')
('fighting', '$$$', '$$$', '$$$')
```

`lingpy.sequence.ngrams.`**`get_all_ngrams`**`(`*sequence*, *sort=False*`)`

Function returns all possible n-grams of a given sequence.

Parameters **sequence** : list or str

The sequence that shall be converted into its ngram-representation.

Returns **out** : list

A list of all ngrams of the input word, sorted in decreasing order of length.

Examples

```
>>> get_all_ngrams('abcde')
['abcde', 'bcde', 'abcd', 'cde', 'abc', 'bcd', 'ab', 'de', 'cd', 'bc', 'a', 'e',
↪'b', 'd', 'c']
```

lingpy.sequence.ngrams.**get_all_ngrams_by_order**(*sequence*, *orders=None*, *pad_symbol='$$$'*)

Build an iterator for collecting all ngrams of a given set of orders.

If no set of orders (i.e., lengths) is provided, this will collect all possible ngrams in the sequence.

Parameters sequence: list or str :

The sequence from which the ngrams will be collected.

orders: list :

An optional list of the orders of the ngrams to be collected. Can be larger than the length of the sequence, in which case the latter will be padded accordingly if requested. Defaults to the collection of all possible ngrams in the sequence with the minimum padding.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns out: iterable :

An iterable over the ngrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents were killed"
>>> for ngram in get_all_ngrams_by_order(sent):
...     print(ngram)
...
('Insurgents',)
('were',)
('killed',)
('$$$', 'Insurgents')
('Insurgents', 'were')
('were', 'killed')
('killed', '$$$')
('$$$', '$$$', 'Insurgents')
('$$$', 'Insurgents', 'were')
('Insurgents', 'were', 'killed')
('were', 'killed', '$$$')
('killed', '$$$', '$$$')
```

lingpy.sequence.ngrams.**get_all_posngrams**(*sequence*, *pre_orders*, *post_orders*, *pad_symbol='$$$'*, *elm_symbol='###'*)

Build an iterator for collecting all positional ngrams of a sequence.

The elements of the iterator, as returned by get_posngrams(), include a tuple of the context, which can be hashed (as any tuple), the transition symbol, and the position of the symbol in the sequence. Such output is primarily intended for state-by-state relative likelihood computations with stochastics models, and can be approximated to a collection of shingles.

Parameters **sequence: list or str** :

The sequence from which the ngrams will be collected.

pre-orders: int or list :

An integer with the maximum length of the preceding context or a list with all preceding context lengths to be collected. If an integer is passed, all lengths from zero to the informed one will be collected.

post-orders: int or list :

An integer with the maximum length of the following context or a list with all following context lengths to be collected. If an integer is passed, all lengths from zero to the informed one will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

elm_symbol: object :

An optional symbol to be used as transition symbol replacement in the context tuples (the first element in the returned iterator). Defaults to ###.

Returns **out: iterable** :

An iterable over the positional ngrams of the sequence, returned as tuples whose elements are: (1) a tuple representing the context (thus including preceding context, the transition symbol, and the following context), (2) an object with the value of the transition symbol, and (3) the index of the transition symbol in the sequence.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents were killed"
>>> for ngram in get_all_posngrams(sent, 2, 1):
...     print(ngram)
...
(('###',), 'Insurgents', 0)
(('###',), 'were', 1)
(('###',), 'killed', 2)
(('###', 'were'), 'Insurgents', 0)
(('###', 'killed'), 'were', 1)
(('###', '$$$'), 'killed', 2)
(('$$$', '###'), 'Insurgents', 0)
(('Insurgents', '###'), 'were', 1)
(('were', '###'), 'killed', 2)
(('$$$', '###', 'were'), 'Insurgents', 0)
(('Insurgents', '###', 'killed'), 'were', 1)
(('were', '###', '$$$'), 'killed', 2)
(('$$$', '$$$', '###'), 'Insurgents', 0)
(('$$$', 'Insurgents', '###'), 'were', 1)
(('Insurgents', 'were', '###'), 'killed', 2)
(('$$$', '$$$', '###', 'were'), 'Insurgents', 0)
```

(continues on next page)

(continued from previous page)

```
(('$$$', 'Insurgents', '###', 'killed'), 'were', 1)
(('Insurgents', 'were', '###', '$$$'), 'killed', 2)
```

`lingpy.sequence.ngrams.`**`get_n_ngrams`**(*sequence*, *order*, *pad_symbol='$$$'*)

Build an iterator for collecting all ngrams of a given order.

The sequence can optionally be padded with boundary symbols which are equal for before and and after sequence boundaries.

Parameters **sequence: list or str** :

The sequence from which the ngrams will be collected.

order: int :

The order of the ngrams to be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the ngrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in get_n_ngrams(sent, 2):
...     print(ngram)
...
('$$$', 'Insurgents')
('Insurgents', 'killed')
('killed', 'in')
('in', 'ongoing')
('ongoing', 'fighting')
('fighting', '$$$')
```

```
>>> for ngram in get_n_ngrams(sent, 1):
...     print(ngram)
...
('Insurgents',)
('killed',)
('in',)
('ongoing',)
('fighting',)
```

```
>>> for ngram in get_n_ngrams(sent, 0):
...     print(ngram)
...
```

`lingpy.sequence.ngrams.`**`get_posngrams`**(*sequence*, *pre_order=0*, *post_order=0*, *pad_symbol='$$$'*, *elm_symbol='###'*)

Build an iterator for collecting all positional ngrams of a sequence.

The preceding and a following orders (i.e., contexts) must always be informed. The elements of the iterator include a tuple of the context, which can be hashed as any tuple, the transition symbol, and the position of the symbol in the sequence. Such output is primarily intended for state-by-state relative likelihood computations with stochastics models.

Parameters **sequence: list or str** :

The sequence from which the ngrams will be collected.

pre_order: int :

An optional integer specifying the length of the preceding context. Defaults to zero.

post_order: int :

An optional integer specifying the length of the following context. Defaults to zero.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

elm_symbol: object :

An optional symbol to be used as transition symbol replacement in the context tuples (the first element in the returned iterator). Defaults to ###.

Returns **out: iterable** :

An iterable over the positional ngrams of the sequence, returned as tuples whose elements are: (1) a tuple representing the context (thus including preceding context, the transition symbol, and the following context), (2) an object with the value of the transition symbol, and (3) the index of the transition symbol in the sequence.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in get_posngrams(sent, 2, 1):
...     print(ngram)
...
(('$$$', '$$$', '###', 'killed'), 'Insurgents', 0)
(('$$$', 'Insurgents', '###', 'in'), 'killed', 1)
(('Insurgents', 'killed', '###', 'ongoing'), 'in', 2)
(('killed', 'in', '###', 'fighting'), 'ongoing', 3)
(('in', 'ongoing', '###', '$$$'), 'fighting', 4)
```

`lingpy.sequence.ngrams.`**`get_skipngrams`**(*sequence*, *order*, *max_gaps*, *pad_symbol='$$$'*, *single_gap=True*)

Build an iterator for collecting all skip ngrams of a given length.

The function requires an information of the length of the skip ngrams to be collected, allowing to either collect ngrams with an unlimited number of gap openings (as described and implemented in Guthrie et al. 2006) or with at most one gap opening.

Parameters **sequence: list or str** :

The sequence from which the ngrams will be collected. Must not include None as an element, as it is used as a sentinel during skip ngram collection following the implementation offered by Bird et al. 2018 (NLTK), which is a de facto standard.

order: int :

The order of the ngrams to be collected (parameter n in Guthrie et al. 2006).

max_gaps: int :

The maximum number of gaps in the ngrams to be collected (parameter k in Guthrie et al. 2006).

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

single_gap: boolean :

An optional logic value indicating if multiple gap openings are to be allowed, as in Guthrie et al. (2006) and Bird et al. (2018), or if at most one gap_opening is to be allowed. Defaults to True.

Returns out: iterable :

An iterable over the ngrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in get_skipngrams(sent, 2, 2):
...     print(ngram)
...
('$$$', 'Insurgents')
('Insurgents', 'killed')
('killed', 'in')
('in', 'ongoing')
('ongoing', 'fighting')
('fighting', '$$$')
('$$$', 'killed')
('Insurgents', 'in')
('killed', 'ongoing')
('in', 'fighting')
('ongoing', '$$$')
('$$$', 'in')
('Insurgents', 'ongoing')
('killed', 'fighting')
('in', '$$$')
>>> for ngram in get_skipngrams(sent, 2, 2, single_gap=False):
...     print(ngram)
...
('$$$', 'Insurgents')
('$$$', 'killed')
('$$$', 'in')
('Insurgents', 'killed')
('Insurgents', 'in')
('Insurgents', 'ongoing')
('killed', 'in')
('killed', 'ongoing')
('killed', 'fighting')
```

(continues on next page)

(continued from previous page)

```
('in', 'ongoing')
('in', 'fighting')
('in', '$$$')
('ongoing', 'fighting')
('ongoing', '$$$')
('fighting', '$$$')
```

`lingpy.sequence.ngrams.`**`tabjoin`**(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.sequence.ngrams.`**`trigrams`**(*sequence*, ***, *order=3*, *pad_symbol='$$$'*)

Build an iterator for collecting all trigrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the trigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the trigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in trigrams(sent):
...     print(ngram)
...
('$$$', '$$$', 'Insurgents')
('$$$', 'Insurgents', 'killed')
('Insurgents', 'killed', 'in')
('killed', 'in', 'ongoing')
('in', 'ongoing', 'fighting')
('ongoing', 'fighting', '$$$')
('fighting', '$$$', '$$$')
```

lingpy.sequence.profile module

Module provides methods for the handling of orthography profiles.

`lingpy.sequence.profile.`**`context_profile`**`(`*wordlist*, *ref='ipa'*, *col='doculect'*, *semi_diacritics='hsw'*, *merge_vowels=False*, *brackets=None*, *splitters='/, ;~'*, *merge_geminates=True*, *clts=False*, *bad_word='<???>'*, *bad_sound='<?>'*, *unknown_sound='!{0}'*, *examples=2*`)`

Create an advanced Orthography Profile with context and doculect information.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

A wordlist from which you want to derive an initial orthography profile.

ref : str (default=ipa)

The name of the reference column in which the words are stored.

col : str (default=doculect)

Indicate in which column the information on the language variety is stored.

semi_diacritics : str

Indicate characters which can occur both as diacritics (second part in a sound) or alone.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged.

brackets : dict

A dictionary with opening brackets as key and closing brackets as values. Defaults to a pre-defined set of frequently occurring brackets.

splitters : str

The characters which force the automatic splitting of an entry.

clts : dict (default=None)

A dictionary(like) object that converts a given source sound into a potential target sound, using the get()-method of the dictionary. Normally, we think of a CLTS instance here (that is: a cross-linguistic transcription system as defined in the pyclts package).

bad_word : str (default=ń???ż)

Indicate how words that could not be parsed should be handled. Note that both bad_word and bad_sound are format-strings, so you can add formatting information here.

bad_sound : str (default=ń?ż)

Indicate how sounds that could not be converted to a sound class be handled. Note that both bad_word and bad_sound are format-strings, so you can add formatting information here.

unknown_sound : str (default=!{0})

If with_clts is set to True, use this string to indicate that sounds are classified as unknown sound in the CLTS framework.

examples : int(default=2)

Indicate the number of examples that should be printed out.

Returns **profile** : generator

A generator of tuples (three items), indicating the segment, its frequency, the conversion to sound classes in the Dolgopolsky sound-class model, and the unicode-codepoints.

`lingpy.sequence.profile.`**`simple_profile`**(*wordlist*, *ref='ipa'*, *semi_diacritics='hsw'*, *merge_vowels=False*, *brackets=None*, *splitters='/, ;~'*, *merge_geminates=True*, *bad_word='<???>'*, *bad_sound='<?>'*, *clts=None*, *unknown_sound='!{0}'*)

Create an initial Orthography Profile using Lingpys clean_string procedure.

Parameters **wordlist** : ~lingpy.basic.wordlist.Wordlist

A wordlist from which you want to derive an initial orthography profile.

ref : str (default=ipa)

The name of the reference column in which the words are stored.

semi_diacritics : str

Indicate characters which can occur both as diacritics (second part in a sound) or alone.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged.

brackets : dict

A dictionary with opening brackets as key and closing brackets as values. Defaults to a pre-defined set of frequently occurring brackets.

splitters : str

The characters which force the automatic splitting of an entry.

clts : dict (default=None)

A dictionary(like) object that converts a given source sound into a potential target sound, using the get()-method of the dictionary. Normally, we think of a CLTS instance here (that is: a cross-linguistic transcription system as defined in the pyclts package).

bad_word : str (default=ń???ż)

Indicate how words that could not be parsed should be handled. Note that both bad_word and bad_sound are format-strings, so you can add formatting information here.

bad_sound : str (default=ń?ż)

Indicate how sounds that could not be converted to a sound class be handled. Note that both bad_word and bad_sound are format-strings, so you can add formatting information here.

unknown_sound : str (default=!{0})

If with_clts is set to True, use this string to indicate that sounds are classified as unknown sound in the CLTS framework.

Returns **profile** : generator

A generator of tuples (three items), indicating the segment, its frequency, the conversion to sound classes in the Dolgopolsky sound-class model, and the unicode-codepoints.

lingpy.sequence.smoothing module

Module providing various methods for using Ngram models.

The smoothing methods are implemented to be as compatible as possible with those offered by NLTK. In fact, both implementation and comments try to follow Bird at al. as close as possible.

`lingpy.sequence.smoothing.`**`certaintydegree_dist`** (*freqdist*, ***kwargs*)

Returns a log-probability distribution based on the degree of certainty.

In this distribution a mass probability is reserved for unobserved samples from a computation of the degree of certainty that the are no unobserved samples.

Under development and test by Tiago Tresoldi, this is an experimental probability distribution that should not be used as the sole or main distribution for the time being.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

bins: int :

The optional number of sample bins that can be generated by the experiment that is described by the probability distribution. If not specified, it will default to the number of samples in the frequency distribution.

unobs_prob : float

An optional mass probability to be reserved for unobserved states, from 0.0 to 1.0.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`ele_dist`** (*freqdist*, ***kwargs*)

Returns an Expected-Likelihood estimate log-probability distribution.

In an Expected-Likelihood estimate log-probability the frequency distribution of observed samples is used to estimate the probability distribution of the experiment that generated such observation, following a parameter given by a real number *gamma* set by definition to 0.5. As such, it is a generalization of the Lidstone estimate.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

bins: int :

The optional number of sample bins that can be generated by the experiment that is described by the probability distribution. If not specified, it will default to the number of samples in the frequency distribution.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`laplace_dist`**(*freqdist*, ***kwargs*)

Returns a Laplace estimate log-probability distribution.

In a Laplace estimate log-probability the frequency distribution of observed samples is used to estimate the probability distribution of the experiment that generated such observation, following a parameter given by a real number *gamma* set by definition to 1. As such, it is a generalization of the Lidstone estimate.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

bins: int :

The optional number of sample bins that can be generated by the experiment that is described by the probability distribution. If not specified, it will default to the number of samples in the frequency distribution.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`lidstone_dist`**(*freqdist*, ***kwargs*)

Returns a Lidstone estimate log-probability distribution.

In a Lidstone estimate log-probability the frequency distribution of observed samples is used to estimate the probability distribution of the experiment that generated such observation, following a parameter given by a real number *gamma* typycally randing from 0.0 to 1.0. The Lidstone estimate approximates the probability of a sample with count *c* from an experiment with *N* outcomes and *B* bins as *(c+gamma)/(N+B*gamma)*. This is equivalent to adding *gamma* to the count of each bin and taking the Maximum-Likelihood estimate of the resulting frequency distribution, with the corrected space of observation; the probability for an unobserved sample is given by frequency of a sample with gamma observations.

Also called additive smoothing, this estimation method is frequently used with a *gamma* of 1.0 (the so-called Laplace smoothing) or of 0.5 (the so-called Expected likelihood estimate, or ELE).

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

gamma : float

A real number used to parameterize the estimate.

bins: int :

The optional number of sample bins that can be generated by the experiment that is described by the probability distribution. If not specified, it will default to the number of samples in the frequency distribution.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`mle_dist`**(*freqdist*, ***kwargs*)

Returns a Maximum-Likelihood Estimation log-probability distribution.

In an MLE log-probability distribution the probability of each sample is approximated as the frequency of the same sample in the frequency distribution of observed samples. It is the distribution people intuitively adopt when thinking of probability distributions. A mass probability can optionally be reserved for unobserved samples.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

unobs_prob : float

An optional mass probability to be reserved for unobserved states, from 0.0 to 1.0.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`random_dist`**(*freqdist*, ***kwargs*)

Returns a random log-probability distribution.

In a random log-probability distribution all samples, no matter the observed counts, will have a random log-probability computed from a set of randomly drawn floating point values. A mass probability can optionally be reserved for unobserved samples.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

unobs_prob : float

An optional mass probability to be reserved for unobserved states, from 0.0 to 1.0.

seed : any hasheable value

An optional seed for the random number generator, defaulting to None.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`sgt_dist`**(*freqdist*, ***kwargs*)

Returns a Simple Good-Turing log-probability distribution.

The returned log-probability distribution is based on the Good-Turing frequency estimation, as first developed by Alan Turing and I. J. Good and implemented in a more easily computable way by Gale and Sampsons (1995/2001 reprint) in the so-called Simple Good-Turing.

This implementation is based mostly in the one by maxbane (2011) (https://github.com/maxbane/simplegoodturing/blob/master/sgt.py), as well as in the original one in C by Geoffrey Sampson (1995; 2000; 2005; 2008) (https://www.grsampson.net/Resources.html), and in the one by Loper, Bird et al. (2001-2018, NLTK Project) (http://www.nltk.org/_modules/nltk/probability.html). Please note that due to minor differences in implementation intended to guarantee non-zero probabilities even in cases of expected underflow, as well as our relience on scipys libraries for speed and our way of handling probabilities that are not computable when the assumptions of SGT are not met, most results will not exactly match those of the gold standard of Gale and Sampson, even though the differences are never expected to be significative and are equally distributed across the samples.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

p_value : float

The p-value for calculating the confidence interval of the empirical Turing estimate, which guides the decision of using either the Turing estimate x or the loglinear smoothed y. Defaults to 0.05, as per the reference implementation by Sampson, but consider that the authors, both in their paper and in the code following suggestions credited to private communication with Fan Yang, consider using a value of 0.1.

allow_fail : bool

A logic value informing if the function is allowed to fail, throwing RuntimeWarning exceptions, if the essential assumptions on the frequency distribution are not met, i.e., if the slope of the loglinear regression is > -1.0 or if an unobserved count is reached before we are able to cross the smoothing threshold. If set to False, the estimation might result in an unreliable probability distribution; defaults to True.

default_p0 : float

An optional value indicating the probability for unobserved samples (p0) in cases where no samples with a single count are observed; if this value is not specified, p0 will default to a Laplace estimation for the current frequency distribution. Please note that this is intended change from the reference implementation by Gale and Sampson.

Returns **state_prob: dict** :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`smooth_dist`**(*freqdist*, *method*, ***kwargs*)

Returns a smoothed log-probability distribution from a named method.

This method is used to generalize over all implemented smoothing methods, especially in terms of serialization. The *method* argument informs which smoothing mehtod to use and passes all the arguments to the appropriate function.

Parameters **freqdist** : dict

Frequency distribution of samples (keys) and counts (values) from which the log-probability distribution will be calculated.

method: str :

The name of the probability smoothing method to use. Either uniform, random, mle, lidstone, laplace, ele, wittenbell, certaintydegree, or sgt.

kwargs: additional arguments :

Additional arguments passed to the appropriate smoothing method function.

Returns state_prob: dict :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`uniform_dist`**(*freqdist*, ***kwargs*)

Returns a uniform log-probability distribution.

In a uniform log-probability distribution all samples, no matter the observed counts, will have the same log-probability. A mass probability can optionally be reserved for unobserved samples.

Parameters freqdist : dict

Frequency distribution of samples (keys) and counts (values) from which the log-probability distribution will be calculated.

unobs_prob : float

An optional mass probability to be reserved for unobserved states, from 0.0 to 1.0.

Returns state_prob: dict :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

`lingpy.sequence.smoothing.`**`wittenbell_dist`**(*freqdist*, ***kwargs*)

Returns a Witten-Bell estimate log-probability distribution.

In a Witten-Bell estimate log-probability a uniform probability mass is allocated to yet unobserved samples by using the number of samples that have only been observed once. The probability mass reserved for unobserved samples is equal to *T / (N +T)*, where *T* is the number of observed samples and *N* the number of total observations. This equates to the Maximum-Likelihood Estimate of a new type of sample occurring. The remaining probability mass is discounted such that all probability estimates sum to one, yielding:

- *p = T / Z (N + T)*, if count == 0
- *p = c / (N + T)*, otherwise

Parameters freqdist : dict

Frequency distribution of samples (keys) and counts (values) from which the probability distribution will be calculated.

bins: int :

The optional number of sample bins that can be generated by the experiment that is described by the probability distribution. If not specified, it will default to the number of samples in the frequency distribution.

Returns state_prob: dict :

A dictionary of sample to log-probabilities for all the samples in the frequency distribution.

unobserved_prob: float :

The log-probability for samples not found in the frequency distribution.

lingpy.sequence.sound_classes module

Module provides various methods for the handling of sound classes.

`lingpy.sequence.sound_classes.`**`asjp2tokens`**(*seq*, *merge_vowels=True*)

`lingpy.sequence.sound_classes.`**`check_tokens`**(*tokens*, ***keywords*)
Function checks whether tokens are given in a consistent input format.

`lingpy.sequence.sound_classes.`**`class2tokens`**(*tokens*, *classes*, *gap_char='-'*, *local=False*)
Turn aligned sound-class sequences into an aligned sequences of IPA tokens.

Parameters **tokens** : list

The list of tokens corresponding to the unaligned IPA string.

classes : string or list

The aligned class string.

gap_char : string (default=-)

The character which indicates gaps in the output string.

local : bool (default=False)

If set to *True* a local alignment with prefix and suffix can be converted.

Returns **alignment** : list

A list of tokens with gaps at the positions where they occured in the alignment of the class string.

See also:

`ipa2tokens`, *`tokens2class`*

Examples

```
>>> from lingpy import *
>>> tokens = ipa2tokens('tsy')
>>> aligned_sequence = 'CU-KE'
>>> print ', '.join(class2tokens(tokens,aligned_sequence))
ts, y, -, ,
```

`lingpy.sequence.sound_classes.`**`clean_string`**(*sequence*, *semi_diacritics='hsw'*, *merge_vowels=False*, *segmentized=False*, *rules=None*, *ignore_brackets=True*, *brackets=None*, *split_entries=True*, *splitters='/, ;~'*, *preparse=None*, *merge_geminates=True*, *normalization_form='NFC'*)
Function exhaustively checks how well a sequence is understood by LingPy.

Parameters **semi_diacritics** : str

Indicate characters which can occur both as diacritics (second part in a sound) or alone.

merge_vowels : bool (default=True)

Indicate whether consecutive vowels should be merged.

segmentized : False

Indicate whether the input string is already segmentized or not. If set to True, items in brackets can no longer be ignored.

rules : dict

Replacement rules to be applied to a segmentized string.

ignore_brackets : bool

If set to True, ignore all content within a given bracket.

brackets : dict

A dictionary with opening brackets as key and closing brackets as values. Defaults to a pre-defined set of frequently occurring brackets.

split_entries : bool (default=True)

Indicate whether multiple entries (with a comma etc.) should be split into separate entries.

splitters : str

The characters which force the automatic splitting of an entry.

preparse : list

List of tuples, giving simple replacement patterns (source and target), which are applied before every processing starts.

Returns cleaned_strings : list

A list of cleaned strings which are segmented by space characters. If splitters are encountered, indicating that the entry contains two variants, the list will contain one for each element in a separate entry. If there are no splitters, the list has only size one.

`lingpy.sequence.sound_classes.`**`codepoint`**(*s*)

Return unicode codepoint(s) for a character set.

`lingpy.sequence.sound_classes.`**`get_all_ngrams`**(*sequence*, *sort=False*)

Function returns all possible n-grams of a given sequence.

Parameters sequence : list or str

The sequence that shall be converted into its ngram-representation.

Returns out : list

A list of all ngrams of the input word, sorted in decreasing order of length.

Examples

```
>>> get_all_ngrams('abcde')
['abcde', 'bcde', 'abcd', 'cde', 'abc', 'bcd', 'ab', 'de', 'cd', 'bc', 'a', 'e',
↪'b', 'd', 'c']
```

`lingpy.sequence.sound_classes.`**`ipa2tokens`**(*istring*, ***keywords*)

Tokenize IPA-encoded strings.

Parameters seq : str

The input sequence that shall be tokenized.

diacritics : {str, None} (default=None)

A string containing all diacritics which shall be considered in the respective analysis. When set to *None*, the default diacritic string will be used.

vowels : {str, None} (default=None)

A string containing all vowel symbols which shall be considered in the respective analysis. When set to *None*, the default vowel string will be used.

tones : {str, None} (default=None)

A string indicating all tone letter symbals which shall be considered in the respective analysis. When set to *None*, the default tone string will be used.

combiners : str (default=)

A string with characters that are used to combine two separate characters (compare affricates such as ts).

breaks : str (default=-.)

A string containing the characters that indicate that a new token starts right after them. These can be used to indicate that two consecutive vowels should not be treated as diphtongs or for diacritics that are put before the following letter.

merge_vowels : bool (default=False)

Indicate, whether vowels should be merged into diphtongs (default=True), or whether each vowel symbol should be considered separately.

merge_geminates : bool (default=False)

Indicate, whether identical symbols should be merged into one token, or rather be kept separate.

expand_nasals : bool (default=False)

semi_diacritics: str (default=) :

Indicate which symbols shall be treated as semi-diacritics, that is, as symbols which can occur on their own, but which eventually, when preceded by a consonant, will form clusters with it. If you want to disable this features, just set the keyword to an empty string.

clean_string : bool (default=False)

Conduct a rough string-cleaning strategy by which all items between brackets are removed along with the brackets, and

Returns tokens : list

A list of IPA tokens.

See also:

`tokens2class`, `class2tokens`

Examples

```
>>> from lingpy import *
>>> myseq = 'tsy'
>>> ipa2tokens(myseq)
['ts', 'y', '', '']
```

`lingpy.sequence.sound_classes.`**`ono_parse`**(*word*, *output=''*, ***keywords*)

Carry out a rough onset-nucleus-offset parse of a word in IPA.

Notes

Method is an approximation and not supposed to do without flaws. It is, however, rather helpful in most instances. It defines a so far simple model in which 7 different contexts for each word are distinguished:

- #: onset cluster in a words initial
- C: onset cluster in a words non-initial
- V: nucleus vowel in a words initial syllable
- v: nucleus vowel in a words non-initial and non-final syllable
- >: nucleus vowel in a words final syllable
- c: offset cluster in a words non-final syllable
- $: offset cluster in a words final syllable

`lingpy.sequence.sound_classes.`**`pgrams`**(*sequence*, ***keywords*)

Convert a given sequence into bigrams consisting of prosodic string symbols and the tokens of the original sequence.

`lingpy.sequence.sound_classes.`**`pid`**(*almA*, *almB*, *mode=2*)

Calculate the Percentage Identity (PID) score for aligned sequence pairs.

Parameters **almA, almB** : string or list

The aligned sequences which can be either a string or a list.

mode : { 1, 2, 3, 4, 5 }

Indicate which of the four possible PID scores described in `Raghava2006` should be calculated, the fifth possibility is added for linguistic purposes:

1. identical positions / (aligned positions + internal gap positions),
2. identical positions / aligned positions,
3. identical positions / shortest sequence, or
4. identical positions / shortest sequence (including internal gap pos.)
5. identical positions / (aligned positions + 2 * number of gaps)

Returns **score** : float

The PID score of the given alignment as a floating point number between 0 and 1.

See also:

`lingpy.compare.Multiple.get_pid`,

Notes

The PID score is a common measure for the diversity of a given alignment. The implementation employed by LingPy follows the description of `Raghava2006` where four different variants of PID scores are distinguished. Essentially, the PID score is based on the comparison of identical residue pairs with the total number of residue pairs in a given alignment.

Examples

Load an alignment from the test suite.

```
>>> from lingpy import *
>>> pairs = PSA(get_file('test.psa'))
```

Extract the alignments of the first aligned sequence pair.

```
>>> almA,almB,score = pairs.alignments[0]
```

Calculate the PID score of the alignment.

```
>>> pid(almA,almB)
0.44444444444444442
```

`lingpy.sequence.sound_classes.`**`prosodic_string`**(*string*, *_output=True*, ***keywords*)

Create a prosodic string of the sonority profile of a sequence.

Parameters **seq** : list

A list of integers indicating the sonority of the tokens of the underlying sequence.

stress : str (default=rcParams[stress])

A string containing the stress symbols used in the analysis. Defaults to the stress as defined in ~lingpy.settings.rcParams.

diacritics : str (default=rcParams[diacritics])

A string containing diacritic symbols used in the analysis. Defaults to the diacritic symbolds defined in ~lingpy.settings.rcParams.

cldf : bool (default=False)

If set to True, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a ``source/target`` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Returns **prostring** : string

A prosodic string corresponding to the sonority profile of the underlying sequence.

See also:

`prosodic`

Notes

A prosodic string is a sequence of specific characters which indicating their resprective prosodic context (see `List2012` or `List2012a` for a detailed description). In contrast to the previous model, the current implementation allows for a more fine-graded distinction between different prosodic segments. The current scheme distinguishes 9 prosodic positions:

- `A`: sequence-initial consonant
- `B`: syllable-initial, non-sequence initial consonant in a context of ascending sonority
- `C`: non-syllable, non-initial consonant in ascending sonority context
- `L`: non-syllable-final consonant in descending environment
- `M`: syllable-final consonant in descending environment
- `N`: word-final consonant
- `X`: first vowel in a word
- `Y`: non-final vowel in a word
- `Z`: vowel occuring in the last position of a word
- `T`: tone
- `_`: word break

Examples

```
>>> prosodic_string(ipa2tokens('tsy')
'AXBZ'
```

`lingpy.sequence.sound_classes.`**`prosodic_weights`**(*prostring*, *_transform={}*)
Calculate prosodic weights for each position of a sequence.

Parameters **prostring** : string

A prosodic string as it is returned by *`prosodic_string()`*.

_transform : dict

A dictionary that determines how prosodic strings should be transformed into prosodic weights. Use this dictionary to adjust the prosodic strings to your own user-defined prosodic weight schema.

Returns **weights** : list

A list of floats reflecting the modification of the weight for each position.

See also:

`prosodic_string`

Notes

Prosodic weights are specific scaling factors which decrease or increase the gap score of a given segment in alignment analyses (see `List2012` or `List2012a` for a detailed description).

Examples

```
>>> from lingpy import *
>>> prostring = '#vC>'
>>> prosodic_weights(prostring)
[2.0, 1.3, 1.5, 0.7]
```

`lingpy.sequence.sound_classes.`**`sampa2uni`**`(`*seq*`)`

Convert sequence in IPA-sampa-format to IPA-unicode.

Notes

This function is based on code taken from Peter Kleiweg (http://www.let.rug.nl/~kleiweg/L04/devel/python/xsampa.html).

`lingpy.sequence.sound_classes.`**`syllabify`**`(`*seq*, *output='flat'*, ***keywords*`)`

Carry out a simple syllabification of a sequence, using sonority as a proxy.

Parameters output: {flat, breakpoints, nested} (default=flat) :

Define how to output the syllabification. Select between: * flat: A syllable separator is introduced to mark the syllable boundaries * breakpoins: A tuple consisting of indices that slice the original sequence into syllables is returned. * nested: A nested list reflecting the syllable structure is returned.

sep : str (default=)

Select your preferred syllable separator.

Returns syllable : list

Either a flat list containing a morpheme separator, or a nested list, reflecting the syllable structure, or a list of tuples containing the indices indicating where the input sequence should be sliced in order to split it into syllables.

Notes

When analyzing the sequence, we start a new syllable in all cases where we reach a deepest point in the sonority hierarchy of the sonority profile of the sequence. When passing an aligned string to this function, the gaps will be ignored when computing boundaries, but later on re-introduced, if the alignment is passed in segmented form.

`lingpy.sequence.sound_classes.`**`token2class`**`(`*token*, *model*, *stress=None*, *diacritics=None*, *cldf=None*`)`

Convert a single token into a sound-class.

tokens [str] A token (phonetic segment).

model [*`Model`*] A *`Model`* object.

stress [str (default=rcParams[stress])] A string containing the stress symbols used in the analysis. Defaults to the stress as defined in ~lingpy.settings.rcParams.

diacritics [str (default=rcParams[diacritics])] A string containing diacritic symbols used in the analysis. Defaults to the diacritic symbolds defined in ~lingpy.settings.rcParams.

cldf [bool (default=False)] If set to True, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a `` `source/target` `` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Returns **sound_class** : str

A sound-class representation of the phonetic segment. If the segment cannot be resolved, the respective string will be rendered as 0 (zero).

See also:

`ipa2tokens`, `class2tokens`, `token2class`

`lingpy.sequence.sound_classes.`**`tokens2class`**(*tokens*, *model*, *stress=None*, *diacritics=None*, *cldf=True*)

Convert tokenized IPA strings into their respective class strings.

Parameters **tokens** : list

A list of tokens as they are returned from `ipa2tokens()`.

model : `Model`

A `Model` object.

stress : str (default=rcParams[stress])

A string containing the stress symbols used in the analysis. Defaults to the stress as defined in ~lingpy.settings.rcParams.

diacritics : str (default=rcParams[diacritics])

A string containing diacritic symbols used in the analysis. Defaults to the diacritic symbolds defined in ~lingpy.settings.rcParams.

cldf : bool (default=True)

If set to True, as by default, this will allow for a specific treatment of phonetic symbols which cannot be completely resolved (e.g., laryngeal h_2 in Indo-European). Following the CLDF specifications (in particular the specifications for writing transcriptions in segmented strings, as employed by the CLTS initiative), in cases of insecurity of pronunciation, users can adopt a ``source/target`` style, where the source is the symbol used, e.g., in a reconstruction system, and the target is a proposed phonetic interpretation. This practice is also accepted by the EDICTOR tool.

Returns **classes** : list

A sound-class representation of the tokenized IPA string in form of a list. If sound classes cannot be resolved, the respective string will be rendered as 0 (zero).

See also:

`ipa2tokens`, `class2tokens`, `token2class`

Notes

The function ~lingpy.sequence.sound_classes.token2class returns a 0 (zero) if the sound is not recognized by LingPys sound class models. While an unknown sound in a longer sequence is no problem for alignment algorithms, we have some unwanted and often even unforeseeable behavior, if the sequence is completely unknown. For this reason, this function raises a ValueError, if a resulting sequence only contains unknown sounds.

Examples

```
>>> from lingpy import *
>>> tokens = ipa2tokens('tsy')
>>> classes = tokens2class(tokens,'sca')
>>> print(classes)
CUKE
```

`lingpy.sequence.sound_classes.`**`tokens2morphemes`**(*tokens*, ***keywords*)

Split a string into morphemes if it contains separators.

Parameters **sep** : str (default=)

Select your morpheme separator.

word_sep: str (default=_) :

Select your word separator.

Returns **morphemes** : list

A nested list of the original segments split into morphemes.

Notes

Function splits a list of tokens into subsequent lists of morphemes if the list contains morpheme separators. If no separators are found, but tonemarkers, it will still split the string according to the tones. If you want to avoid this behavior, set the keyword **split_on_tones** to False.

Module contents

Module provides methods and functions for dealing with linguistic sequences.

`lingpy.sequence.`**`bigrams`**(*sequence*, ***, *order=2*, *pad_symbol='$$$'*)

Build an iterator for collecting all bigrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the bigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the bigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in bigrams(sent):
...     print(ngram)
```

(continues on next page)

(continued from previous page)

```
...
('$$$', 'Insurgents')
('Insurgents', 'killed')
('killed', 'in')
('in', 'ongoing')
('ongoing', 'fighting')
('fighting', '$$$')
```

`lingpy.sequence.confirm`(*question*, ***, *default=False*)

Ask a yes/no question interactively.

Parameters **`question`** – The text of the question to ask.

Returns True if the answer was yes, False otherwise.

`lingpy.sequence.data_path`(**comps*)

`lingpy.sequence.dotjoin`(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.sequence.fourgrams`(*sequence*, ***, *order=4*, *pad_symbol='$$$'*)

Build an iterator for collecting all fourgrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the fourgrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the fourgrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in fourgrams(sent):
...     print(ngram)
...
('$$$', '$$$', '$$$', 'Insurgents')
('$$$', '$$$', 'Insurgents', 'killed')
('$$$', 'Insurgents', 'killed', 'in')
('Insurgents', 'killed', 'in', 'ongoing')
('killed', 'in', 'ongoing', 'fighting')
('in', 'ongoing', 'fighting', '$$$')
('ongoing', 'fighting', '$$$', '$$$')
('fighting', '$$$', '$$$', '$$$')
```

`lingpy.sequence.`**`tabjoin`**`(`**args*, ***kw*`)`

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.sequence.`**`trigrams`**`(`*sequence*, ***, *order=3*, *pad_symbol='$$$'*`)`

Build an iterator for collecting all trigrams of a sequence.

The sequence is padded by default.

Parameters sequence: list or str :

The sequence from which the trigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns out: iterable :

An iterable over the trigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in trigrams(sent):
...     print(ngram)
...
('$$$', '$$$', 'Insurgents')
('$$$', 'Insurgents', 'killed')
('Insurgents', 'killed', 'in')
('killed', 'in', 'ongoing')
('in', 'ongoing', 'fighting')
('ongoing', 'fighting', '$$$')
('fighting', '$$$', '$$$')
```

lingpy.tests package

Subpackages

lingpy.tests.algorithm package

Submodules

lingpy.tests.algorithm.test__tree module

class `lingpy.tests.algorithm.test__tree.`**`Tests`**`(`*methodName='runTest'*`)`

Bases: `unittest.case.TestCase`

setUp()
Hook method for setting up the test fixture before exercising it.

test_grf()

lingpy.tests.algorithm.test_cluster_util module

class lingpy.tests.algorithm.test_cluster_util.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_generate_all_clusters()

test_generate_random_clusters()

test_mutate_cluster()

test_order_cluster()

test_valid_cluster()

lingpy.tests.algorithm.test_clustering module

class lingpy.tests.algorithm.test_clustering.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_best_threshold()

test_check_taxa()

test_check_taxon_names()

test_find_threshold()

test_flat_cluster()

test_fuzzy()

test_link_clustering()

test_matrix2groups()

test_matrix2tree()

test_neighbor()

test_partition_density()

test_upgma()

lingpy.tests.algorithm.test_cython module

class lingpy.tests.algorithm.test_cython.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test__calign()

test__malign()

test__talign()

test_corrdist()

lingpy.tests.algorithm.test_extra module

class lingpy.tests.algorithm.test_extra.**Cluster**(**args*, ***kw*)

Bases: mock.mock.MagicMock

class AffinityPropagation(**args*, ***kw*)

Bases: object

static fit_predict(*arg*)

static dbscan(**args*, ***kw*)

class lingpy.tests.algorithm.test_extra.**Components**(*nodes*)

Bases: object

subgraphs()

class lingpy.tests.algorithm.test_extra.**Igraph**(**args*, ***kw*)

Bases: mock.mock.MagicMock

class Graph(*vs=[]*)

Bases: object

add_edge(*a*, *b*)

add_vertex(*vertex*)

community_infomap(**args*, ***kw*)

class lingpy.tests.algorithm.test_extra.**Tests**(*methodName='runTest'*)

Bases: unittest.case.TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

test_affinity_propagation()

test_clustering()

test_dbscan()

test_infomap_clustering()

Module contents

lingpy.tests.align package

Submodules

lingpy.tests.align.test_multiple module

Testing multiple module.

class lingpy.tests.align.test_multiple.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test___get__()

test_get_local_peaks()

test_get_pairwise_alignments()

test_get_peaks()

test_get_pid()

test_iterate_all_sequences()

test_iterate_clusters()

test_iterate_orphans()

test_iterate_similar_gap_sites()

test_lib_align()

test_mult_align()

test_prog_align()

test_sum_of_pairs()

test_swap_check()

lingpy.tests.align.test_pairwise module

class lingpy.tests.align.test_pairwise.**TestPairwise**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_align()

test_basics()

lingpy.tests.align.test_pairwise.**test_editdist**()

lingpy.tests.align.test_pairwise.**test_nw_align**()

lingpy.tests.align.test_pairwise.**test_pw_align**()

lingpy.tests.align.test_pairwise.**test_structalign**()

lingpy.tests.align.test_pairwise.**test_sw_align**()

lingpy.tests.align.test_pairwise.**test_turchin**()

lingpy.tests.align.test_pairwise.**test_we_align**()

lingpy.tests.align.test_sca module

Test the SCA module.

class lingpy.tests.align.test_sca.**TestAlignments**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_align()

test_get_confidence()

test_get_consensus()

test_ipa2tokens()

test_output()

class lingpy.tests.align.test_sca.**TestMSA**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_output()

class lingpy.tests.align.test_sca.**TestPSA**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_output()

lingpy.tests.align.test_sca.**test_get_consensus**()

lingpy.tests.align.test_sca.**test_partial_alignments_with_lexstat**()

Module contents

lingpy.tests.basic package

Submodules

lingpy.tests.basic.test_ops module

Test wordlist module.

class lingpy.tests.basic.test_ops.**TestOps**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_calculate_data()

test_clean_taxnames()

test_coverage()

test_iter_rows()

test_renumber()

test_tsv2triple()

test_wl2dict()

test_wl2dst()

test_wl2multistate()

test_wl2qlc()

lingpy.tests.basic.test_parser module

class lingpy.tests.basic.test_parser.**TestParser**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_add_entries()

test_cache()

test_get_entries()

test_getattr()

test_getitem()

test_init()

test_len()

lingpy.tests.basic.test_parser.**data_path**(**comps*)

lingpy.tests.basic.test_tree module

class lingpy.tests.basic.test_tree.**TestTree**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_getDistanceToRoot()

test_get_LCA()

test_get_distance()

test_get_distance_unknown()
test failure with unknown distance

test_init_from_file()

test_init_from_list()

lingpy.tests.basic.test_tree.**test_random_tree**()

lingpy.tests.basic.test_tree.**test_star_tree**()

lingpy.tests.basic.test_wordlist module

Test wordlist module.

class lingpy.tests.basic.test_wordlist.**TestWordlist**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test___len__()

test_calculate()

test_coverage()

test_export()

test_get_dict()

test_get_entries()

test_get_etymdict()

test_get_list()

test_get_paps()

test_get_wordlist()

test_output()

test_renumber()

Module contents

lingpy.tests.compare package

Submodules

lingpy.tests.compare.test__phylogeny module

class lingpy.tests.compare.test__phylogeny.**Graph**(**args*, ***kw*)
Bases: mock.mock.MagicMock

nodes(***kw*)

class lingpy.tests.compare.test__phylogeny.**Nx**(**args*, ***kw*)
Bases: mock.mock.MagicMock

Graph(**args*, ***kw*)

generate_gml(**args*)

class lingpy.tests.compare.test__phylogeny.**Plt**(**args*, ***kw*)
Bases: mock.mock.MagicMock

Polygon(**args*, ***kw*)

fill(**args*, ***kw*)

gca(**args*, ***kw*)

plot(**args*, ***kw*)

text(**args*, ***kw*)

class lingpy.tests.compare.test__phylogeny.**SPS**(**args*, ***kw*)
Bases: mock.mock.MagicMock

mstats = <MagicMock id='139781404517264'>

class lingpy.tests.compare.test__phylogeny.**TestUtils**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_utils()

lingpy.tests.compare.test__phylogeny.**test_convex_hull**()

lingpy.tests.compare.test__phylogeny.**test_get_convex_hull**()

lingpy.tests.compare.test__phylogeny.**test_get_polygon_from_nodes**()

lingpy.tests.compare.test__phylogeny.**test_seg_intersect**()

lingpy.tests.compare.test__phylogeny.**test_settings**()

lingpy.tests.compare.test_lexstat module

class lingpy.tests.compare.test_lexstat.**TestLexStat**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test__get_matrices()

test_align_pairs()

test_cluster()

test_correctness()

test_get_distances()

test_get_frequencies()

test_get_scorer()

test_get_subset()

test_getitem()

test_init()

test_init2()

test_init3()

test_output()

lingpy.tests.compare.test_lexstat.**test_char_from_charstring**()

lingpy.tests.compare.test_lexstat.**test_get_score_dict**()

lingpy.tests.compare.test_partial module

class lingpy.tests.compare.test_partial.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test__get_slices()

test_add_cognate_ids()

test_get_partial_matrices()

test_partial_cluster()

lingpy.tests.compare.test_phylogeny module

Test the TreBor borrowing detection algorithm.

class lingpy.tests.compare.test_phylogeny.**Bmp**(**args*, ***kw*)
Bases: mock.mock.MagicMock

Basemap(**args*, ***kw*)

class lingpy.tests.compare.test_phylogeny.**Graph**(**args*, ***kw*)
Bases: mock.mock.MagicMock

static nodes()

class lingpy.tests.compare.test_phylogeny.**Plt**(**args*, ***kw*)
Bases: mock.mock.MagicMock

static plot(**args*, ***kw*)

class lingpy.tests.compare.test_phylogeny.**Sp**(**args*, ***kw*)
Bases: mock.mock.MagicMock

stats = <MagicMock id='139781404577408'>

class lingpy.tests.compare.test_phylogeny.**TestPhyBo**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_get_GLS()

test_plot()

lingpy.tests.compare.test_sanity module

class lingpy.tests.compare.test_sanity.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test__get_concepts()

test__mutual_coverage()

test_mutual_coverage()

test_mutual_coverage_check()

test_mutual_coverage_subset()

test_synonymy()

lingpy.tests.compare.test_strings module

class lingpy.tests.compare.test_strings.**TestStrings**(*methodName='runTest'*)

Bases: unittest.case.TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

test_bidist1()

test_bidist2()

test_bidist3()

test_bisim1()

test_bisim2()

test_bisim3()

test_dice()

test_ident()

test_jcd()

test_jcdn()

test_lcs()

test_ldn()

test_ldn_swap()

test_prefix()

test_tridist1()

test_tridist2()

test_tridist3()

test_trigram()

test_trisim1()

test_trisim2()

test_trisim3()

test_xdice()

test_xxdice()

Module contents

lingpy.tests.convert package

Submodules

lingpy.tests.convert.test_cldf module

class lingpy.tests.convert.test_cldf.**Tests**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_from_cldf()

lingpy.tests.convert.test_cldf_methods module

class lingpy.tests.convert.test_cldf_methods.**CLDFWordlistWriteTest**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_load_cldf_and_write()

class lingpy.tests.convert.test_cldf_methods.**FailTests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_load_noexisting_cldf(***kw*)

test_load_non_wordlist_cldf(***kw*)

class lingpy.tests.convert.test_cldf_methods.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_load_from_cldf_metadata()

test_load_from_cldf_metadatafree()

lingpy.tests.convert.test_graph module

class lingpy.tests.convert.test_graph.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_igraph2networkx()

test_networkx2igraph()

lingpy.tests.convert.test_html module

class lingpy.tests.convert.test_html.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_alm2html()

test_color_range()

test_msa2html()

test_psa2html()

test_strings_and_tokens2html()

lingpy.tests.convert.test_html.**template_path**(**comps*)

lingpy.tests.convert.test_plot module

class lingpy.tests.convert.test_plot.**Plt**(**args, **kw*)
Bases: mock.mock.MagicMock

static plot(**args, **kw*)

class lingpy.tests.convert.test_plot.**Sch**(**args, **kw*)
Bases: mock.mock.MagicMock

static dendrogram(**args*, ***kw*)

class lingpy.tests.convert.test_plot.**TestPlot**(*methodName='runTest'*)

Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()

Hook method for setting up the test fixture before exercising it.

test_plots()

lingpy.tests.convert.test_strings module

Test conversions involving strings.

class lingpy.tests.convert.test_strings.**TestIsConstant**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_all_absent()

test_all_present()

test_not_constant()

class lingpy.tests.convert.test_strings.**TestWriteNexus**(*methodName='runTest'*)

Bases: *lingpy.tests.util_testing.WithTempDir*

Tests for *write_nexus*

assertRegexWorkaround(*a*, *b*)

setUp()

Hook method for setting up the test fixture before exercising it.

test_beast()

test_beastwords()

test_error_on_unknown_mode()

test_error_on_unknown_ref()

test_merge_custom_statements()

test_mrbayes()

test_splitstree()

test_traitlab()

class lingpy.tests.convert.test_strings.**Tests**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_matrix2dst()

test_msa2str()

test_pap2csv()

test_pap2nex()

test_scorer2str()

Test conversion of scorers to strings.

lingpy.tests.convert.test_tree module

class lingpy.tests.convert.test_tree.**TestTree**(*methodName='runTest'*)
    Bases: *lingpy.tests.util_testing.WithTempDir*

    setUp()
        Hook method for setting up the test fixture before exercising it.

    test__nwk_format()

    test_nwk2tree_matrix()

Module contents

lingpy.tests.data package

Submodules

lingpy.tests.data.test_derive module

class lingpy.tests.data.test_derive.**TestDerive**(*methodName='runTest'*)
    Bases: *lingpy.tests.util_testing.WithTempDir*

    setUp()
        Hook method for setting up the test fixture before exercising it.

    test_compile_dvt()

    test_compile_model()

lingpy.tests.data.test_sound_class_models module

class lingpy.tests.data.test_sound_class_models.**Tests**
    Bases: object

    failures = {}

    model = 'cv'

    models = ['sca', 'dolgo', 'art', 'color', 'asjp', 'cv']

    segment = ''

    segments = {'!', '!', '#', "'", '+', '-', '_', 'a', 'b', 'bv', 'bv', 'c', 'd', 'dz', '

    values = ['', '', '', 'p', '$_{14}$', '', '$_{6}$', '', 'ʋ', 'f', '', '', '^{53}', '', '', 'd', '',

Module contents

lingpy.tests.evaluate package

Submodules

lingpy.tests.evaluate.test_acd module

class lingpy.tests.evaluate.test_acd.**Tests**(*methodName='runTest'*)
  Bases: *lingpy.tests.util_testing.WithTempDir*

  setUp()
    Hook method for setting up the test fixture before exercising it.

  test_bcubes()

  test_diff()

  test_extreme_cognates()

  test_pairs()

  test_partial_bcubes()

  test_random_cognates()

lingpy.tests.evaluate.test_acd.**test_npoint_ap**()

lingpy.tests.evaluate.test_alr module

class lingpy.tests.evaluate.test_alr.**Tests**(*methodName='runTest'*)
  Bases: *lingpy.tests.util_testing.WithTempDir*

  setUp()
    Hook method for setting up the test fixture before exercising it.

  test_med()

lingpy.tests.evaluate.test_apa module

class lingpy.tests.evaluate.test_apa.**Tests**(*methodName='runTest'*)
  Bases: *lingpy.tests.util_testing.WithTempDir*

  test_EvalMSA()

  test_EvalPSA()

Module contents

lingpy.tests.meaning package

Submodules

lingpy.tests.meaning.test_colexification module

Tests for colexification module.

class lingpy.tests.meaning.test_colexification.**TestColexifications**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test__get_colexifications()

test__get_colexifications_by_taxa()

test__get_statistics()

test__make_graph()

test__make_matrix()

test_colexification_network()

test_compare_colexifications()

test_evaluate_colexifications()

lingpy.tests.meaning.test_colexification.**dotjoin**(**args*, ***kw*)
Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

Module contents

lingpy.tests.read package

Submodules

lingpy.tests.read.test_csv module

Tests for the read.csv module.

class lingpy.tests.read.test_csv.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_csv2dict()

test_csv2list()

test_csv2multidict()

test_read_asjp()

lingpy.tests.read.test_phylip module

Basic tests for the Phylip module.

class lingpy.tests.read.test_phylip.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_read_dst()

test_read_scorer()

lingpy.tests.read.test_qlc module

class lingpy.tests.read.test_qlc.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_normalize_alignment()

test_read_msa()

test_read_qlc()

test_reduce_msa()

lingpy.tests.read.test_starling module

class lingpy.tests.read.test_starling.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_star2qlc()

Module contents

lingpy.tests.sequence package

Submodules

lingpy.tests.sequence.test_generate module

class lingpy.tests.sequence.test_generate.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_evaluate_string()

test_get_string()

lingpy.tests.sequence.test_ngrams module

class lingpy.tests.sequence.test_ngrams.**Tests**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_all_ngrams()

test_bigrams()

test_fourgrams()

test_get_all_ngrams_by_order()

test_get_all_posngrams()

test_get_n_grams()

test_get_posngrams()

test_get_skipngrams()

test_ngram_class()

test_trigrams()

lingpy.tests.sequence.test_ngrams.**bigrams**(*sequence*, ***, *order=2*, *pad_symbol='$$$'*)
Build an iterator for collecting all bigrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the bigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the bigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in bigrams(sent):
...     print(ngram)
...
('$$$', 'Insurgents')
('Insurgents', 'killed')
('killed', 'in')
('in', 'ongoing')
```

(continues on next page)

(continued from previous page)

```
('ongoing', 'fighting')
('fighting', '$$$')
```

`lingpy.tests.sequence.test_ngrams.`**`fourgrams`**`(`*sequence*, *, *order=4*, *pad_symbol='$$$'*`)`

Build an iterator for collecting all fourgrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the fourgrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the fourgrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in fourgrams(sent):
...     print(ngram)
...
('$$$', '$$$', '$$$', 'Insurgents')
('$$$', '$$$', 'Insurgents', 'killed')
('$$$', 'Insurgents', 'killed', 'in')
('Insurgents', 'killed', 'in', 'ongoing')
('killed', 'in', 'ongoing', 'fighting')
('in', 'ongoing', 'fighting', '$$$')
('ongoing', 'fighting', '$$$', '$$$')
('fighting', '$$$', '$$$', '$$$')
```

`lingpy.tests.sequence.test_ngrams.`**`trigrams`**`(`*sequence*, *, *order=3*, *pad_symbol='$$$'*`)`

Build an iterator for collecting all trigrams of a sequence.

The sequence is padded by default.

Parameters **sequence: list or str** :

The sequence from which the trigrams will be collected.

pad_symbol: object :

An optional symbol to be used as start-of- and end-of-sequence boundaries. The same symbol is used for both boundaries. Must be a value different from None, defaults to $$$.

Returns **out: iterable** :

An iterable over the trigrams of the sequence, returned as tuples.

Examples

```
>>> from lingpy.sequence import *
>>> sent = "Insurgents killed in ongoing fighting"
>>> for ngram in trigrams(sent):
...     print(ngram)
...
('$$$', '$$$', 'Insurgents')
('$$$', 'Insurgents', 'killed')
('Insurgents', 'killed', 'in')
('killed', 'in', 'ongoing')
('in', 'ongoing', 'fighting')
('ongoing', 'fighting', '$$$')
('fighting', '$$$', '$$$')
```

lingpy.tests.sequence.test_profile module

class `lingpy.tests.sequence.test_profile.`**`Tests`**`(`*methodName='runTest'*`)`
Bases: `unittest.case.TestCase`

`setUp``()`
Hook method for setting up the test fixture before exercising it.

`test_context_profile``()`

`test_simple_profile``()`

lingpy.tests.sequence.test_smoothing module

class `lingpy.tests.sequence.test_smoothing.`**`Tests`**`(`*methodName='runTest'*`)`
Bases: `unittest.case.TestCase`

`setUp``()`
Hook method for setting up the test fixture before exercising it.

`test_certaintydegree_dist``()`
Test for the Degree of Certainty distribution.

`test_ele_dist``()`
Test for the Expected-Likelihood estimation distribution.

`test_laplace_dist``()`
Test for the Laplace distribution.

`test_mle_dist``()`
Test for the Maximum-Likelihood Estimation distribution.

`test_random_dist``()`
Test for the random distribution.

`test_sgt_dist``()`
Test for the Simple Good-Turing distribution.

`test_uniform_dist``()`
Test for the uniform distribution.

`test_wittenbell_dist``()`
Test for the Witten-Bell distribution.

lingpy.tests.sequence.test_sound_classes module

class lingpy.tests.sequence.test_sound_classes.**Tests**(*methodName='runTest'*)

Bases: unittest.case.TestCase

setUp()

Hook method for setting up the test fixture before exercising it.

test_check_tokens()

test_class2tokens()

test_clean_string()

test_codepoint()

test_ipa2tokens()

test_onoparse()

test_pgrams()

test_pid()

test_prosodic_string()

test_prosodic_weights()

test_sampa2uni()

test_syllabify()

test_token2class()

test_tokens2class()

test_tokens2morphemes()

Module contents

lingpy.tests.thirdparty package

Submodules

lingpy.tests.thirdparty.test_cogent module

Test thirdparty modules.

class lingpy.tests.thirdparty.test_cogent.**Tests**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_PhyloNode()

test_Tree()

test_load_tree()

test_more_trees()

lingpy.tests.thirdparty.test_linkcomm module

class lingpy.tests.thirdparty.test_linkcomm.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_hlc()

Module contents

Submodules

lingpy.tests.test_basictypes module

class lingpy.tests.test_basictypes.**Tests**
Bases: object

app = ['1', '2', '3']

f = [1.0, 2.0, 3.0, 1.0, 2.0, 3.0]

i = [1, 2, 3]

l = ['1', '2', '3', '+', '1', '2', '3']

s = ['1', '2', '3']

string1 = '1 2 3 + 1 2 3'

string2 = '1 2 3 1 2 3'

lingpy.tests.test_cache module

class lingpy.tests.test_cache.**TestCache**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_cache()

lingpy.tests.test_cli module

class lingpy.tests.test_cli.**Tests**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

static run_cli(**args*)

test_alignments()

test_lexstat()

test_multiple()

test_ortho_profile()

test_pairwise()

test_profile()

test_settings()

test_wordlist()

lingpy.tests.test_cli.**capture**(**args*)

lingpy.tests.test_config module

class lingpy.tests.test_config.**ConfigTest**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

setUp()
Hook method for setting up the test fixture before exercising it.

test_default()

test_existing_config()

test_new_config()

lingpy.tests.test_log module

class lingpy.tests.test_log.**LogTest**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

tearDown()
Hook method for deconstructing the test fixture after testing it.

test_Logging_context_manager()

static test_convenience()

test_default_config()

test_new_config()

lingpy.tests.test_util module

class lingpy.tests.test_util.**Test**(*methodName='runTest'*)
Bases: *lingpy.tests.util_testing.WithTempDir*

test_TextFile()

test_write_text_file()

class lingpy.tests.test_util.**TestCombinations**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_combinations2()

class lingpy.tests.test_util.**TestJoin**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_as_string()

test_dotjoin()

test_join()

lingpy.tests.util module

Utilities used in lingpy tests

`lingpy.tests.util.`**`get_log`**`()`

A mock object for *lingpy.log* to test whether log messages have been emitted.

Returns Mock instance.

`lingpy.tests.util.`**`test_data`**`(`**comps*`)`

Access test data files.

Parameters **`comps`** – Path components of the data file path relative to

the test_data dir. :return: Absolute path to the specified test data file.

lingpy.tests.util_testing module

`class` `lingpy.tests.util_testing.`**`WithTempDir`**`(`*methodName='runTest'*`)`

Bases: *`lingpy.tests.util_testing.WithTempDirMixin`*, `unittest.case.TestCase`

Backwards compatible test base class.

`class` `lingpy.tests.util_testing.`**`WithTempDirMixin`**

Bases: `object`

Composable test fixture providing access to a temporary directory.

http://nedbatchelder.com/blog/201210/multiple_inheritance_is_hard.html

`setUp``()`

`tearDown``()`

`tmp_path``(`**comps*`)`

`lingpy.tests.util_testing.`**`capture`**`(`*func*, **args*, ***kw*`)`

`lingpy.tests.util_testing.`**`capture_all`**`(`*func*, **args*, ***kw*`)`

Module contents

lingpy.thirdparty package

Subpackages

lingpy.thirdparty.cogent package

Submodules

lingpy.thirdparty.cogent.newick module

Newick format with all features as per the specs at: http://evolution.genetics.washington.edu/phylip/newick_doc.html http://evolution.genetics.washington.edu/phylip/newicktree.html

ie: Unquoted label underscore munging Quoted labels Inner node labels Lengths [ ] Comments (discarded) Unlabeled tips

also: Double quotes can be used. Spaces and quote marks are OK inside unquoted labels.

exception `lingpy.thirdparty.cogent.newick.`**`TreeParseError`**
Bases: `ValueError`

`lingpy.thirdparty.cogent.newick.`**`parse_string`**(*text*, *constructor*, ***kw*)
Parses a Newick-format string, using specified constructor for tree.

Calls constructor(children, name, attributes)

Note: underscore_unmunge, if True, replaces underscores with spaces in the data thats read in. This is part of the Newick format, but it is often useful to suppress this behavior.

lingpy.thirdparty.cogent.tree module

`lingpy.thirdparty.cogent.tree.`**`LoadTree`**(*filename=None*, *treestring=None*, *tip_names=None*, *underscore_unmunge=False*)
Constructor for tree.

Arguments, use only one of:

- filename: a file containing a newick or xml formatted tree.
- treestring: a newick or xml formatted tree string.
- tip_names: a list of tip names.

Notes

Underscore_unmunging is turned off by default, although it is part of the Newick format. Set underscore_unmunge to True to replace underscores with spaces in all names read.

class `lingpy.thirdparty.cogent.tree.`**`PhyloNode`**(**args*, ***kwargs*)
Bases: *`lingpy.thirdparty.cogent.tree.TreeNode`*

`Length`

`balanced`()
Tree rooted here with no neighbour having > 50% of the edges.

Notes

Using a balanced tree can substantially improve performance of the likelihood calculations. Note that the resulting tree has a different orientation with the effect that specifying clades or stems for model parameterisation should be done using the outgroup_name argument.

`bifurcating`(*constructor=None*)

`compareByPartitions`(*other*, *debug=False*)

`distance`(*other*)
Returns branch length between self and other.

`getDistances`(*endpoints=None*)
The distance matrix as a dictionary.

Usage: Grabs the branch lengths (evolutionary distances) as a complete matrix (i.e. a,b and b,a).

`getNewick`(*with_distances=False*, *semicolon=True*, *escape_name=True*)
Return the newick string for this tree.

Arguments:

- with_distances: whether branch lengths are included.
- semicolon: end tree string with a semicolon
- **escape_name: if any of these characters [](),:;_ exist in a** nodes name, wrap the name in single quotes

NOTE: This method returns the Newick representation of this node and its descendents. This method is a modification of an implementation by Zongzhi Liu

prune()
Reconstructs correct tree after nodes have been removed.

Internal nodes with only one child will be removed and new connections and Branch lengths will be made to reflect change.

rootAtMidpoint()
return a new tree rooted at midpoint of the two tips farthest apart

this fn doesnt preserve the internal node naming or structure, but does keep tip to tip distances correct. uses unrootedDeepcopy()

rootedAt(*edge_name*)
Return a new tree rooted at the provided node.

Usage: This can be useful for drawing unrooted trees with an orientation that reflects knowledge of the true root location.

rootedWithTip(*outgroup_name*)
A new tree with the named tip as one of the roots children

sameTopology(*other*)
Tests whether two trees have the same topology.

scaleBranchLengths(*max_length=100*, *ultrametric=False*)
Scales BranchLengths in place to integers for ascii output.

Warning: tree might not be exactly the length you specify.

Set ultrametric=True if you want all the root-tip distances to end up precisely the same.

setTipDistances()
Sets distance from each node to the most distant tip.

tipToTipDistances(*endpoints=None*, *default_length=1*)
Returns distance matrix between all pairs of tips, and a tip order.

Warning: .__start and .__stop added to self and its descendants.

tip_order contains the actual node objects, not their names (may be confusing in some cases).

totalDescendingBranchLength()
Returns total descending branch length from self

unrooted()
A tree with at least 3 children at the root.

unrootedDeepcopy(*constructor=None*, *parent=None*)

class lingpy.thirdparty.cogent.tree.**TreeBuilder**(*mutable=False*, *constructor=<class 'lingpy.thirdparty.cogent.tree.PhyloNode'>*)
Bases: object

createEdge(*children*, *name*, *params*, *nameLoaded=True*)
Callback for newick parser

edgeFromEdge(*edge*, *children*, *params=None*)
Callback for tree-to-tree transforms like getSubTree

exception `lingpy.thirdparty.cogent.tree.`**TreeError**
Bases: `Exception`

class `lingpy.thirdparty.cogent.tree.`**TreeNode**(*Name=None*, *Children=None*, *Parent=None*, *Params=None*, *NameLoaded=True*, ***kwargs*)
Bases: `object`

Store information about a tree node. Mutable.

Parameters: Name: label for the node, assumed to be unique. Children: list of the nodes children. Params: dict containing arbitrary parameters for the node. NameLoaded: ?

Parent
Accessor for parent.

If using an algorithm that accesses Parent a lot, it will be much faster to access self._parent directly, but dont do it if mutating self._parent! (or, if you must, remember to clean up the refs).

ancestors()
Returns all ancestors back to the root. Dynamically calculated.

append(*i*)
Appends i to self.Children, in-place, cleaning up refs.

asciiArt(*show_internal=True*, *compact=False*, *labels=False*)
Returns a string containing an ascii drawing of the tree.

Parameters **show_internal: bool** :

includes internal edge names.

compact: bool :

use exactly one line per tip.

labels: {bool, list} :

specify specific labels for all nodes in the tree.

Notes

The labels-keyword was added to the function by JML.

childGroups()
Returns list containing lists of children sharing a state.

In other words, returns runs of tip and nontip children.

compareByNames(*other*)
Equality test for trees by name

compareBySubsets(*other*, *exclude_absent_taxa=False*)
Returns fraction of overlapping subsets where self and other differ.

Other is expected to be a tree object compatible with PhyloNode.

Notes

Names present in only one of the two trees will count as mismatches: if you dont want this behavior, strip out the non-matching tips first.

compareName(*other*)
Compares TreeNode by name

copy(*memo=None*, *_nil=[]*, *constructor='ignored'*)
Returns a copy of self using an iterative approach

copyRecursive(*memo=None*, *_nil=[]*, *constructor='ignored'*)
Returns copy of selfs structure, including shallow copy of attrs.

constructor is ignored; required to support old tree unit tests.

copyTopology(*constructor=None*)
Copies only the topology and labels of a tree, not any extra data.

Useful when you want another copy of the tree with the same structure and labels, but want to e.g. assign different branch lengths and environments. Does not use deepcopy from the copy module, so _much_ faster than the copy() method.

deepcopy(*memo=None*, *_nil=[]*, *constructor='ignored'*)
Returns a copy of self using an iterative approach

descendantArray(*tip_list=None*)
Returns numpy array with nodes in rows and descendants in columns.

A value of 1 indicates that the decendant is a descendant of that node/ A value of 0 indicates that it is not

Also returns a list of nodes in the same order as they are listed in the array.

tip_list is a list of the names of the tips that will be considered, in the order they will appear as columns in the final array. Internal nodes will appear as rows in preorder traversal order.

extend(*items*)
Extends self.Children by items, in-place, cleaning up refs.

getConnectingEdges(*name1*, *name2*)
returns a list of edges connecting two nodes

includes self and other in the list

getConnectingNode(*name1*, *name2*)
Finds the last common ancestor of the two named edges.

getDistances(*endpoints=None*)
The distance matrix as a dictionary.

Usage: Grabs the branch lengths (evolutionary distances) as a complete matrix (i.e. a,b and b,a).

getEdgeNames(*tip1name*, *tip2name*, *getclade*, *getstem*, *outgroup_name=None*)
Return the list of stem and/or sub tree (clade) edge name(s). This is done by finding the common intersection, and then getting the list of names. If the clade traverses the root, then use the outgroup_name argument to ensure valid specification.

Arguments:

- tip1/2name: edge 1/2 names
- getstem: whether the name of the clade stem edge is returned.
- getclade: whether the names of the edges within the clade are returned

- outgroup_name: if provided the calculation is done on a version of the tree re-rooted relative to the provided tip.

Usage: The returned list can be used to specify subtrees for special parameterisation. For instance, say you want to allow the primates to have a different value of a particular parameter. In this case, provide the results of this method to the parameter controller method *setParamRule()* along with the parameter name etc..

getEdgeVector()
Collect the list of edges in postfix order

getMaxTipTipDistance()
Returns the max tip tip distance between any pair of tips

Returns (dist, tip_names, internal_node)

getNewick(*with_distances=False*, *semicolon=True*, *escape_name=True*)
Return the newick string for this tree.

Arguments:

- with_distances: whether branch lengths are included.
- semicolon: end tree string with a semicolon
- **escape_name: if any of these characters [](),:;_ exist in a** nodes name, wrap the name in single quotes

NOTE: This method returns the Newick representation of this node and its descendents. This method is a modification of an implementation by Zongzhi Liu

getNewickRecursive(*with_distances=False*, *semicolon=True*, *escape_name=True*)
Return the newick string for this edge.

Arguments:

- with_distances: whether branch lengths are included.
- semicolon: end tree string with a semicolon
- **escape_name: if any of these characters [](),:;_ exist in a** nodes name, wrap the name in single quotes

getNodeMatchingName(*name*)

getNodeNames(*includeself=True*, *tipsonly=False*)
Return a list of edges from this edge - may or may not include self. This node (or first connection) will be the first, and then they will be listed in the natural traverse order.

getNodesDict()
Returns a dict keyed by node name, value is node

Will raise TreeError if non-unique names are encountered

getParamValue(*param*, *edge*)
returns the parameter value for named edge

getSubTree(*name_list*, *ignore_missing=False*, *keep_root=False*)
A new instance of a sub tree that contains all the otus that are listed in name_list.

ignore_missing: if False, getSubTree will raise a ValueError if name_list contains names that arent nodes in the tree keep_root: if False, the root of the subtree will be the last common ancestor of all nodes kept in the subtree. Root to tip distance is then (possibly) different from the original tree If True, the root to tip distance remains constant, but root may only have one child node.

getTipNames(*includeself=False*)
return the list of the names of all tips contained by this edge

get_LCA(**nodes*)
Find lowest common ancestor of a given number of nodes.

Notes

This function is supposed to yield the same output as lowestCommonAncestor does. It was added in order to overcome certain problems in the original function, resulting from attributes added to a PhyloNode-object that make the use at time unsecure. Furthermore, it works with an arbitrary list of nodes (including tips and internal nodes).

indexInParent()
Returns index of self in parent.

insert(*index*, *i*)
Inserts an item at specified position in self.Children.

isRoot()
Returns True if the current is a root, i.e. has no parent.

isTip()
Returns True if the current node is a tip, i.e. has no children.

isroot()
Returns True if root of a tree, i.e. no parent.

istip()
Returns True if is tip, i.e. no children.

iterNontips(*include_self=False*)
Iterates over nontips descended from self, [] if none.

include_self, if True (default is False), will return the current node as part of the list of nontips if it is a nontip.

iterTips(*include_self=False*)
Iterates over tips descended from self, [] if self is a tip.

lastCommonAncestor(*other*)
Finds last common ancestor of self and other, or None.

Always tests by identity.

lca(*other*)
Finds last common ancestor of self and other, or None.

Always tests by identity.

levelorder(*include_self=True*)
Performs levelorder iteration over tree

lowestCommonAncestor(*tipnames*)
Lowest common ancestor for a list of tipnames

This should be around O(H sqrt(n)), where H is height and n is the number of tips passed in.

makeTreeArray(*dec_list=None*)
Makes an array with nodes in rows and descendants in columns.

A value of 1 indicates that the decendant is a descendant of that node/ A value of 0 indicates that it is not

also returns a list of nodes in the same order as they are listed in the array

maxTipTipDistance()
returns the max distance between any pair of tips

Also returns the tip names that it is between as a tuple

nameUnnamedNodes()
sets the Data property of unnamed nodes to an arbitrary value

Internal nodes are often unnamed and so this function assigns a value for referencing.

nonTipChildren()
Returns direct children in self that have descendants.

nontips(*include_self=False*)
Returns nontips descended from self.

pop(*index=-1*)
Returns and deletes child of self at index (default: -1)

postorder(*include_self=True*)
Performs postorder iteration over tree.

This is somewhat inelegant compared to saving the node and its index on the stack, but is 30% faster in the average case and 3x faster in the worst case (for a comb tree).

Zongzhi Lius slower but more compact version is:

def postorder_zongzhi(self): stack = [[self, 0]] while stack:

curr, child_idx = stack[-1] if child_idx < len(curr.Children):

stack[-1][1] += 1 stack.append([curr.Children[child_idx], 0])

else: yield stack.pop()[0]

pre_and_postorder(*include_self=True*)
Performs iteration over tree, visiting node before and after.

preorder(*include_self=True*)
Performs preorder iteration over tree.

prune()
Reconstructs correct topology after nodes have been removed.

Internal nodes with only one child will be removed and new connections will be made to reflect change.

reassignNames(*mapping*, *nodes=None*)
Reassigns node names based on a mapping dict

mapping : dict, old_name -> new_name nodes : specific nodes for renaming (such as just tips, etc)

remove(*target*)
Removes node by name instead of identity.

Returns True if node was present, False otherwise.

removeNode(*target*)
Removes node by identity instead of value.

Returns True if node was present, False otherwise.

root()
Returns root of the tree self is in. Dynamically calculated.

sameShape(*other*)
Ignores lengths and order, so trees should be sorted first

separation(*other*)
Returns number of edges separating self and other.

setMaxTipTipDistance()
Propagate tip distance information up the tree

This method was originally implemented by Julia Goodrich with the intent of being able to determine max tip to tip distances between nodes on large trees efficiently. The code has been modified to track the specific tips the distance is between

setParamValue(*param*, *edge*, *value*)
sets the value for param at named edge

siblings()
Returns all nodes that are children of the same parent as self.

Notes

Excludes self from the list. Dynamically calculated.

sorted(*sort_order=[]*)
An equivalent tree sorted into a standard order. If this is not specified then alphabetical order is used. At each node starting from root, the algorithm will try to put the descendant which contains the lowest scoring tip on the left.

subset()
Returns set of names that descend from specified node

subsets()
Returns all sets of names that come from specified node and its kids

tipChildren()
Returns direct children of self that are tips.

tips(*include_self=False*)
Returns tips descended from self, [] if self is a tip.

traverse(*self_before=True*, *self_after=False*, *include_self=True*)
Returns iterator over descendants. Iterative: safe for large trees.

Notes

self_before includes each node before its descendants if True. self_after includes each node after its descendants if True. include_self includes the initial node if True.

self_before and self_after are independent. If neither is True, only terminal nodes will be returned.

Note that if self is terminal, it will only be included once even if self_before and self_after are both True.

This is a depth-first traversal. Since the trees are not binary, preorder and postorder traversals are possible, but inorder traversals would depend on the data in the tree and are not handled here.

traverse_recursive (*self_before=True*, *self_after=False*, *include_self=True*)
Returns iterator over descendants. IMPORTANT: read notes below.

Notes

traverse_recursive is slower than traverse, and can lead to stack errors. However, you _must_ use traverse_recursive if you plan to modify the tree topology as you walk over it (e.g. in post-order), because the iterative methods use their own stack that is not updated if you alter the tree.

self_before includes each node before its descendants if True. self_after includes each node after its descendants if True. include_self includes the initial node if True.

self_before and self_after are independent. If neither is True, only terminal nodes will be returned.

Note that if self is terminal, it will only be included once even if self_before and self_after are both True.

This is a depth-first traversal. Since the trees are not binary, preorder and postorder traversals are possible, but inorder traversals would depend on the data in the tree and are not handled here.

writeToFile (*filename*, *with_distances=True*, *format=None*)
Save the tree to filename

Arguments:

- filename: self-evident
- with_distances: whether branch lengths are included in string.
- format: default is newick, xml is alternate. Argument overrides the filename suffix. All attributes are saved in the xml format.

lingpy.thirdparty.cogent.tree.**cmp** (*a*, *b*)

lingpy.thirdparty.cogent.tree.**comb** (*items*, *n=None*)
Yields each successive combination of n items.

items: a slicable sequence. n: number of items in each combination This version from Raymond Hettinger, 2006/03/23

Module contents

Simple py3-port of PyCogents (http://pycogent.sourceforge.net) Tree classes.

lingpy.thirdparty.linkcomm package

Submodules

lingpy.thirdparty.linkcomm.link_clustering module

changes 2010-08-27:

- all three output files now contain the same community id numbers
- comm2nodes and comm2edges both present the cid as the first

entry of each line. Previously only comm2nodes did this. * implemented weighted version, added -w switch * expanded help string to explain input and outputs

`lingpy.thirdparty.linkcomm.link_clustering.`**`Dc`**(*m*, *n*)
partition density

`class` `lingpy.thirdparty.linkcomm.link_clustering.`**`HLC`**(*adj*, *edges*)
Bases: `object`

`initialize_edges`()

`merge_comms`(*edge1*, *edge2*)

`single_linkage`(*threshold=None*, *w=None*)

`lingpy.thirdparty.linkcomm.link_clustering.`**`similarities_unweighted`**(*adj*)
Get all the edge similarities. Input dict maps nodes to sets of neighbors. Output is a list of decorated edge-pairs, (1-sim,eij,eik), ordered by similarity.

`lingpy.thirdparty.linkcomm.link_clustering.`**`similarities_weighted`**(*adj*, *ij2wij*)
Same as similarities_unweighted but using tanimoto coefficient. ‘adj is a dict mapping nodes to sets of neighbors, ij2wij is a dict mapping an edge (ni,nj) tuple to the weight wij of that edge.

`lingpy.thirdparty.linkcomm.link_clustering.`**`swap`**(*a*, *b*)

Module contents

Module provides a simple py3 port for link community analyses, following the algorithm by James Bagrow, Yong-Yeol Ahn.

Module contents

Submodules

lingpy.basictypes module

`class` `lingpy.basictypes.`**`aligned`**(*iterable*)
Bases: `lingpy.basictypes._strings`

`a`

`class` `lingpy.basictypes.`**`lists`**(*iterable*, *sep=' + '*)
Bases: `lingpy.basictypes._strings`

`change`(*i*, *item*)

`extend`(*iterable*) → None – extend list by appending elements from the iterable

lingpy.cache module

Implements the lingpy cache.

Some operations in lingpy may be time consuming, so we provide a mechanism to cache the results of these operations.

`lingpy.cache.`**`dump`**(*data*, *filename*)

`lingpy.cache.`**`load`**(*filename*)

`lingpy.cache.`**`path`**(*filename*)

lingpy.cli module

`class` `lingpy.cli.`**`Command`**

Bases: `object`

Base class for subcommands of the lingpy command line interface.

`help = None`

`output`(*args*, *content*)

`classmethod subparser`(*parser*)

Hook to define subcommand arguments.

`class` `lingpy.cli.`**`CommandMeta`**(*name*, *bases*, *dct*)

Bases: `type`

A metaclass which keeps track of subclasses, if they have all-lowercase names.

`lingpy.cli.`**`add_align_method_option`**(*p*)

`lingpy.cli.`**`add_cognate_identifier_option`**(*p*, *default*)

`lingpy.cli.`**`add_format_option`**(*p*, *default*, *choices*)

`lingpy.cli.`**`add_method_option`**(*p*, *default*, *choices*, *spec=''*)

`lingpy.cli.`**`add_mode_option`**(*p*, *choices*)

`lingpy.cli.`**`add_option`**(*parser*, *name_*, *default_*, *help_*, *short_opt=None*, ***kw*)

`lingpy.cli.`**`add_shared_args`**(*p*)

`lingpy.cli.`**`add_strings_option`**(*p*, *n*)

`lingpy.cli.`**`add_tree_calc_option`**(*p*)

`class` `lingpy.cli.`**`alignments`**

Bases: *`lingpy.cli.Command`*

Carry out alignment analysis of a wordlist file with readily detected cognates.

`classmethod subparser`(*p*)

Hook to define subcommand arguments.

`lingpy.cli.`**`get_parser`**()

`class` `lingpy.cli.`**`help`**

Bases: *`lingpy.cli.Command`*

Show help for commands.

`classmethod subparser`(*parser*)

Hook to define subcommand arguments.

`class` `lingpy.cli.`**`lexstat`**

Bases: *`lingpy.cli.Command`*

`classmethod subparser`(*p*)

Hook to define subcommand arguments.

`lingpy.cli.`**`main`**(**args*)

LingPy command line interface.

class lingpy.cli.**multiple**
Bases: *lingpy.cli.Command*

Multiple alignment console interface for LingPy.

classmethod subparser(*p*)
Hook to define subcommand arguments.

class lingpy.cli.**pairwise**
Bases: *lingpy.cli.Command*

Run pairwise analyses from command line in LingPy

Notes

Currently, the following options are supported:

- run normal analyses without sound class strings
- run sound-class based analyses

Furthermore, input output is handled as follows:

- define user input using psa-formats in lingpy
- define user output (stdout, file)

classmethod subparser(*p*)
Hook to define subcommand arguments.

class lingpy.cli.**profile**
Bases: *lingpy.cli.Command*

classmethod subparser(*p*)
Hook to define subcommand arguments.

class lingpy.cli.**settings**
Bases: *lingpy.cli.Command*

classmethod subparser(*p*)
Hook to define subcommand arguments.

class lingpy.cli.**wordlist**
Bases: *lingpy.cli.Command*

Load a wordlist and carry out simple checks.

classmethod subparser(*p*)
Hook to define subcommand arguments.

lingpy.compat module

Functionality to provide compatibility across supported python versions

lingpy.config module

Configuration management for lingpy.

Various aspects of lingpy can be configured and customized by the user. This is done with configuration files in the users config dir.

See also:

https://pypi.python.org/pypi/appdirs/

class `lingpy.config.`**`Config`**(*name*, *default=None*, ***kw*)

Bases: `configparser.RawConfigParser`

lingpy.log module

Logging utilities

class `lingpy.log.`**`CustomFilter`**(*name=”*)

Bases: `logging.Filter`

`filter`(*record*)

Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class `lingpy.log.`**`Logging`**(*level=10*, *logger=None*)

Bases: `object`

A context manager to execute a block of code at a specific logging level.

`lingpy.log.`**`debug`**(*msg*, ***kw*)

`lingpy.log.`**`deprecated`**(*old*, *new*)

`lingpy.log.`**`error`**(*msg*, ***kw*)

`lingpy.log.`**`file_written`**(*fname*, *logger=None*)

`lingpy.log.`**`get_level`**()

`lingpy.log.`**`get_logger`**(*config_dir=None*, *force_default_config=False*, *test=False*)

Get a logger configured according to the lingpy log config file.

Note: If no logging configuration file exists, it will be created.

Parameters

- **`config_dir`** – Directory in which to look for/create the log config file.
- **`force_default_config`** – Configure the logger using the default config.
- **`test`** – Force reconfiguration of the logger.

Returns A logger.

`lingpy.log.`**`info`**(*msg*, ***kw*)

`lingpy.log.`**`missing_module`**(*name*, *logger=None*)

`lingpy.log.`**`warning`**(*msg*)

lingpy.settings module

Module handels all global parameters used in a LingPy session.

`lingpy.settings.`**`rc`**(*rval=None*, ***keywords*)

Function changes parameters globally set for LingPy sessions.

Parameters **rval** : string (default=None)

Use this keyword to specify a return-value for the rc-function.

schema : {ipa, asjp}

Change the basic schema for sequence comparison. When switching to asjp, this means that sequences will be treated as sequences in ASJP code, otherwise, they will be treated as sequences written in basic IPA.

Notes

This function is the standard way to communicate with the *rcParams* dictionary which is not imported as a default. If you want to see which parameters there are, you can load the rcParams dictonary directly:

```
>>> from lingpy.settings import rcParams
```

However, be careful when changing the values. They might produce some unexpected behavior.

Examples

Import LingPy:

```
>>> from lingpy import *
```

Switch from IPA transcriptions to ASJP transcriptions:

```
>>> rc(schema="asjp")
```

You can check which basic orthography is currently loaded:

```
>>> rc(basic_orthography)
'asjp'
>>> rc(schema='ipa')
>>> rc(basic_orthography)
'fuzzy'
```

lingpy.util module

class lingpy.util.**TemporaryPath**(*suffix=”*)
Bases: object

class lingpy.util.**TextFile**(*path, log=True*)
Bases: object

lingpy.util.**accumulate_purepy**(*iterable, func=<built-in function add>*)
Return running totals.

This implementation replaces itertools.accumulate for compatibility with Python 2.7.

lingpy.util.**as_string**(*obj, pprint=False*)

lingpy.util.**charstring**(*id_, char=’X’, cls=’-’*)

lingpy.util.**combinations2**(*iterable*)
Convenience shortcut

lingpy.util.**confirm**(*question, *, default=False*)
Ask a yes/no question interactively.

Parameters **question** – The text of the question to ask.

Returns True if the answer was yes, False otherwise.

`lingpy.util.`**`data_path`**(**comps*)

`lingpy.util.`**`dotjoin`**(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.util.`**`identity`**(*x*)

`lingpy.util.`**`join`**(*sep*, **args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.util.`**`lines_to_text`**(*lines*)

`lingpy.util.`**`lingpy_path`**(**comps*)

`lingpy.util.`**`multicombinations2`**(*iterable*)

Convenience shortcut, for the name, see the Wikipedia article on Combination.

https://en.wikipedia.org/wiki/Combination#Number_of_combinations_with_repetition

`lingpy.util.`**`nexus_slug`**(*s*)

Converts a string to a nexus safe representation (i.e. removes many unicode characters and removes some punctuation characters).

Parameters **s** : str

A string to convert to a nexus safe format.

Returns **s** : str

A string containing a nexus safe label.

`lingpy.util.`**`random_choices`**(*population*, *weights=None*, *cum_weights=None*, *k=1*)

Return a population sample from weighted elements.

In particular, return a *k* sized list of elements chosen from *population* with replacement and according to a list of weights. If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights. For example, the relative weights *[10, 5, 30, 5]* are equivalent to the cumulative weights *[10, 15, 45, 50]*. Internally, the relative weights are converted to the cumulative weights before making selections, so supplying the cumulative weights saves work.

This function is compatible with the random.choices() function available in Pythons standard library from version 3.6 on. It can be replaced by the standard implementation once the version requirement is updated.

Parameters **population: list** :

A list of elements from which the element(s) will be drawn.

weights: list :

A list of any numeric type with the relative weight of each element. Either *weights* or *cum_weights* must be provided.

cum_weights: list :

A list of any numeric type with the accumulated weight of each element. Either *weights* or *cum_weights* must be provided.

k: int :

The number of elements to be drawn, with replacement.

Returns sample: list :

A list of elements randomly drawn according to the specified weights.

`lingpy.util.`**`read_config_file`**(*path*, ***kw*)

Read lines of a file ignoring commented lines and empty lines.

`lingpy.util.`**`read_text_file`**(*path*, *normalize=None*, *lines=False*)

Read a text file encoded in utf-8.

Parameters path : { Path, str }

File-system path of the file.

normalize : { None, NFC, NFC }

If not *None* a valid unicode normalization mode must be passed.

lines : bool (default=False)

Flag signalling whether to return a list of lines (without the line-separation character).

Returns file_content : { list, str }

File content as unicode object or list of lines as unicode objects.

Notes

The whole file is read into memory.

`lingpy.util.`**`setdefaults`**(*d*, ***kw*)

Shortcut for a common idiom, setting multiple default values at once.

Parameters d : dict

Dictionary to be updated.

kw : dict

Dictionary with default values.

`lingpy.util.`**`tabjoin`**(**args*, ***kw*)

Convenience shortcut. Strings to be joined do not have to be passed as list or tuple.

Notes

An implicit conversion of objects to strings is performed as well.

`lingpy.util.`**`write_text_file`**(*path*, *content*, *normalize=None*, *log=True*)

Write a text file encoded in utf-8.

Parameters path : str

File-system path of the file.

content : str

The text content to be written.

normalize : { None, NFC, NFD } (default=False)

If not *None* a valid unicode normalization mode must be passed.

log : bool (default=True)

Indicate whether you want to log the result of the file writing process.

Module contents

LingPy package for quantitative tasks in historical linguistics.

Documentation is available in the docstrings. Online documentation is available at http://lingpy.org

Subpackages

algorithm Basic Algorithms for Sequence Comparison align Specific Algorithms Alignment Analyses basic Basic Classes for Language Comparison compare Basic Modules for Language Comparison convert Functions for Format Conversion data Data Handling evaluate Basic Classes and Functions for Algorithm Evaluation read Basic Functions for Data Input sequence Basic Functions for Sequence Modeling thirdparty Temporary Forks of Third-Party-Modules

CHAPTER

TEN

DOWNLOAD

10.1 Download

10.1.1 Current Version

The current stable release of LingPy is version 2.6.4. This (and older versions) can be downloaded from:

- **PyPi**: https://pypi.python.org/pypi/lingpy

We are regularly developing LingPy. You can always download the most recent version at our GIT repository:

- **GIT-Repository**: https://github.com/lingpy/lingpy

10.1.2 Older Versions

Older versions of Lingpy that work only with Python 2, including source code and documentation, are still available for download, but they will no longer be modified.

- **LingPy-1.0 (Python2)**: http://pypi.python.org/pypi/lingpy/1.0
- **Documentation (PDF)**: http://lingpy.org/download/lingpy_doc.pdf
- **Documentation (HTML)**: http://lingpy.org/download/lingpy-1.0-doc.zip

PYTHON MODULE INDEX