# Abydos Documentation

***Release 0.3.0***

**Christopher C. Little**

**Oct 15, 2018**

# Contents:

CHAPTER 1

# Introduction

## 1.1 Abydos


Abydos NLP/IR library


Abydos is a library of phonetic algorithms, string distance measures & metrics, stemmers, and string fingerprinters including:

- **Phonetic algorithms**
  - Robert C. Russell's Index
  - American Soundex
  - Refined Soundex
  - Daitch-Mokotoff Soundex

  - Kölner Phonetik
  - NYSIIS
  - Match Rating Algorithm
  - Metaphone
  - Double Metaphone
  - Caverphone
  - Alpha Search Inquiry System
  - Fuzzy Soundex
  - Phonex
  - Phonem
  - Phonix
  - SfinxBis
  - phonet
  - Standardized Phonetic Frequency Code
  - Statistics Canada
  - Lein
  - Roger Root
  - Oxford Name Compression Algorithm (ONCA)
  - Eudex phonetic hash
  - Haase Phonetik
  - Reth-Schek Phonetik
  - FONEM
  - Parmar-Kumbharana
  - Davidson's Consonant Code
  - SoundD
  - PSHP Soundex/Viewex Coding
  - an early version of Henry Code
  - Norphone
  - Dolby Code
  - Phonetic Spanish
  - Spanish Metaphone
  - MetaSoundex
  - SoundexBR
  - NRL English-to-phoneme
  - Beider-Morse Phonetic Matching
- **String distance metrics**

- Levenshtein distance
- Optimal String Alignment distance
- Levenshtein-Damerau distance
- Hamming distance
- Tversky index
- Sørensen–Dice coefficient & distance
- Jaccard similarity coefficient & distance
- overlap similarity & distance
- Tanimoto coefficient & distance
- Minkowski distance & similarity
- Manhattan distance & similarity
- Euclidean distance & similarity
- Chebyshev distance
- cosine similarity & distance
- Jaro distance
- Jaro-Winkler distance (incl. the strcmp95 algorithm variant)
- Longest common substring
- Ratcliff-Obershelp similarity & distance
- Match Rating Algorithm similarity
- Normalized Compression Distance (NCD) & similarity
- Monge-Elkan similarity & distance
- Matrix similarity
- Needleman-Wunsch score
- Smither-Waterman score
- Gotoh score
- Length similarity
- Prefix, Suffix, and Identity similarity & distance
- Modified Language-Independent Product Name Search (MLIPNS) similarity & distance
- Bag distance
- Editex distance
- Eudex distances
- Sift4 distance
- Baystat distance & similarity
- Typo distance
- Indel distance
- Synoname

- **Stemmers**
  - the Lovins stemmer
  - the Porter and Porter2 (Snowball English) stemmers
  - Snowball stemmers for German, Dutch, Norwegian, Swedish, and Danish
  - CLEF German, German plus, and Swedish stemmers
  - Caumann's German stemmer
  - UEA-Lite Stemmer
  - Paice-Husk Stemmer
  - Schinke Latin stemmer
  - S stemmer
- **String Fingerprints**
  - string fingerprint
  - q-gram fingerprint
  - phonetic fingerprint
  - Pollock & Zomora's skeleton key
  - Pollock & Zomora's omission key
  - Cisłak & Grabowski's occurrence fingerprint
  - Cisłak & Grabowski's occurrence halved fingerprint
  - Cisłak & Grabowski's count fingerprint
  - Cisłak & Grabowski's position fingerprint
  - Synoname Toolcode

---

## 1.2 Installation

Required libraries:

- Numpy
- Six

Recommended libraries:

- PylibLZMA (Python 2 only–for LZMA compression string distance metric)

To install Abydos (master) from Github source:

```
git clone https://github.com/chrislit/abydos.git --recursive
cd abydos
python setup install
```

If your default python command calls Python 2.7 but you want to install for Python 3, you may instead need to call:

```
python3 setup install
```

To install Abydos (latest release) from PyPI using pip:

```
pip install abydos
```

To install from conda-forge:

```
conda install abydos
```

It should run on Python 2.7 and Python 3.3-3.7.

## 1.3 Testing & Contributing

To run the whole test-suite just call tox:

```
tox
```

The tox setup has the following environments: py27, py36, doctest, py27-regression, py36-regression, pylint, pycodestyle, flake8, doc8, badges, docs, py27-fuzz, & py36-fuzz. So if only want to generate documentation (in HTML, EPUB, & PDF formats), just call:

```
tox -e docs
```

In order to only run & generate Flake8 reports, call:

```
tox -e flake8
```

Contributions such as bug reports, PRs, suggestions, desired new features, etc. are welcome through the Github Issues & Pull requests.

---

## 1.4 License

CHAPTER 2

# Tutorial

Coming soon. . .

# CHAPTER 3

# abydos

## 3.1 abydos package

abydos.

Abydos NLP/IR library by Christopher C. Little

### 3.1.1 Submodules

#### 3.1.1.1 abydos.clustering module

abydos.clustering.

**The clustering module implements clustering algorithms such as:**

- mean pair-wise similarity

abydos.clustering.**mean_pairwise_similarity**(*collection*, *metric=<function sim>*, *mean_func=<function hmean>*, *symmetric=False*)

Calculate the mean pairwise similarity of a collection of strings.

Takes the mean of the pairwise similarity between each member of a collection, optionally in both directions (for asymmetric similarity metrics.

**Parameters**

- **collection** (*list*) – a collection of terms or a string that can be split
- **metric** (*function*) – a similarity metric function
- **mean_func** (*function*) – a mean function that takes a list of values and returns a float
- **symmetric** (*bool*) – set to True if all pairwise similarities should be calculated in both directions

**Returns** the mean pairwise similarity of a collection of strings

**Return type** float

```
>>> round(mean_pairwise_similarity(['Christopher', 'Kristof',
... 'Christobal']), 12)
0.519801980198
>>> round(mean_pairwise_similarity(['Niall', 'Neal', 'Neil']), 12)
0.545454545455
```

abydos.clustering.**pairwise_similarity_statistics**(*src_collection*, *tar_collection*, *metric=<function sim>*, *mean_func=<function amean>*, *symmetric=False*)

Calculate the pairwise similarity statistics a collection of strings.

Calculate pairwise similarities among members of two collections, returning the maximum, minimum, mean (according to a supplied function, arithmetic mean, by default), and (population) standard deviation of those similarities.

**Parameters**

- **src_collection** (*list*) – a collection of terms or a string that can be split
- **tar_collection** (*list*) – a collection of terms or a string that can be split
- **metric** (*function*) – a similarity metric function
- **mean_func** (*function*) – a mean function that takes a list of values and returns a float
- **symmetric** (*bool*) – set to True if all pairwise similarities should be calculated in both directions

**Returns** the max, min, mean, and standard deviation of similarities

**Return type** tuple

```
>>> tuple(round(_, 12) for _ in pairwise_similarity_statistics(
... ['Christopher', 'Kristof', 'Christobal'], ['Niall', 'Neal', 'Neil']))
(0.2, 0.0, 0.118614718615, 0.075070477184)
```

#### 3.1.1.2 abydos.compression module

abydos.compression.

The compression module defines compression and compression-related functions for use within Abydos, including implementations of the following:

- arithmetic coding functions (ac_train, ac_encode, & ac_decode)
- Burrows-Wheeler transform encoder/decoder (bwt_encode & bwt_decode)
- Run-Length Encoding encoder/decoder (rle_encode & rle_decode)

abydos.compression.**ac_decode**(*longval*, *nbits*, *probs*)

Decode the number to a string using the given statistics.

This is based on Andrew Dalke's public domain implementation [Dal05]. It has been ported to use the fractions.Fraction class.

**Parameters**

- **longval** (*int*) – The first part of an encoded tuple from ac_encode
- **nbits** (*int*) – The second part of an encoded tuple from ac_encode

- **probs** (*dict*) – A probability statistics dictionary generated by ac_train

**Returns** The arithmetically decoded text

**Return type** str

```
>>> pr = ac_train('the quick brown fox jumped over the lazy dog')
>>> ac_decode(16720586181, 34, pr)
'align'
```

abydos.compression.**ac_encode**(*text*, *probs*)

Encode a text using arithmetic coding with the provided probabilities.

Text and the 0-order probability statistics -> longval, nbits

The encoded number is Fraction(longval, 2**nbits)

This is based on Andrew Dalke's public domain implementation [Dal05]. It has been ported to use the fractions.Fraction class.

**Parameters**

- **text** (*str*) – A string to encode
- **probs** (*dict*) – A probability statistics dictionary generated by ac_train

**Returns** The arithmetically coded text

**Return type** tuple

```
>>> pr = ac_train('the quick brown fox jumped over the lazy dog')
>>> ac_encode('align', pr)
(16720586181, 34)
```

abydos.compression.**ac_train**(*text*)

Generate a probability dict from the provided text.

Text -> 0-order probability statistics as a dict

This is based on Andrew Dalke's public domain implementation [Dal05]. It has been ported to use the fractions.Fraction class.

**Parameters** **text** (*str*) – The text data over which to calculate probability statistics. This must not contain the NUL (0x00) character because that's used to indicate the end of data.

**Returns** a probability dict

**Return type** dict

```
>>> ac_train('the quick brown fox jumped over the lazy dog')
{' ': (Fraction(0, 1), Fraction(8, 45)),
 'o': (Fraction(8, 45), Fraction(4, 15)),
 'e': (Fraction(4, 15), Fraction(16, 45)),
 'u': (Fraction(16, 45), Fraction(2, 5)),
 't': (Fraction(2, 5), Fraction(4, 9)),
 'r': (Fraction(4, 9), Fraction(22, 45)),
 'h': (Fraction(22, 45), Fraction(8, 15)),
 'd': (Fraction(8, 15), Fraction(26, 45)),
 'z': (Fraction(26, 45), Fraction(3, 5)),
 'y': (Fraction(3, 5), Fraction(28, 45)),
 'x': (Fraction(28, 45), Fraction(29, 45)),
 'w': (Fraction(29, 45), Fraction(2, 3)),
 'v': (Fraction(2, 3), Fraction(31, 45)),
```

(continues on next page)

(continued from previous page)

```
 'q': (Fraction(31, 45), Fraction(32, 45)),
 'p': (Fraction(32, 45), Fraction(11, 15)),
 'n': (Fraction(11, 15), Fraction(34, 45)),
 'm': (Fraction(34, 45), Fraction(7, 9)),
 'l': (Fraction(7, 9), Fraction(4, 5)),
 'k': (Fraction(4, 5), Fraction(37, 45)),
 'j': (Fraction(37, 45), Fraction(38, 45)),
 'i': (Fraction(38, 45), Fraction(13, 15)),
 'g': (Fraction(13, 15), Fraction(8, 9)),
 'f': (Fraction(8, 9), Fraction(41, 45)),
 'c': (Fraction(41, 45), Fraction(14, 15)),
 'b': (Fraction(14, 15), Fraction(43, 45)),
 'a': (Fraction(43, 45), Fraction(44, 45)),
 '\x00': (Fraction(44, 45), Fraction(1, 1))}
```

abydos.compression.**bwt_decode**(*code*, *terminator='\x00'*)

Return a word decoded from BWT form.

The Burrows-Wheeler transform is an attempt at placing similar characters together to improve compression. This function reverses the transform. Cf. [BW94].

**Parameters**

- **code** (*str*) – the word to transform from BWT form
- **terminator** (*str*) – a character added to word to signal the end of the string

**Returns** word decoded by BWT

**Return type** str

```
>>> bwt_decode('n\x00ilag')
'align'
>>> bwt_decode('annb\x00aa')
'banana'
>>> bwt_decode('annb@aa', '@')
'banana'
```

abydos.compression.**bwt_encode**(*word*, *terminator='\x00'*)

Return the Burrows-Wheeler transformed form of a word.

The Burrows-Wheeler transform is an attempt at placing similar characters together to improve compression. Cf. [BW94].

**Parameters**

- **word** (*str*) – the word to transform using BWT
- **terminator** (*str*) – a character to add to word to signal the end of the string

**Returns** word encoded by BWT

**Return type** str

```
>>> bwt_encode('align')
'n\x00ilag'
>>> bwt_encode('banana')
'annb\x00aa'
>>> bwt_encode('banana', '@')
'annb@aa'
```

abydos.compression.**rle_decode**(*text*, *use_bwt=True*)

Perform decoding of run-length-encoding (RLE).

Cf. [RC67].

Based on http://rosettacode.org/wiki/Run-length_encoding#Python [Cod18b]. This is licensed GFDL 1.2.

Digits 0-9 cannot have been in the original text.

**Parameters**

- **text** (*str*) – a text string to decode
- **use_bwt** (*bool*) – boolean indicating whether to perform BWT decoding after RLE decoding

**Returns** word decoded by BWT

**Return type** str

```
>>> rle_decode('n\x00ilag')
'align'
>>> rle_decode('align', use_bwt=False)
'align'
```

```
>>> rle_decode('annb\x00aa')
'banana'
>>> rle_decode('banana', use_bwt=False)
'banana'
```

```
>>> rle_decode('ab\x00abbab5a')
'aaabaababababa'
>>> rle_decode('3abaababababa', False)
'aaabaababababa'
```

abydos.compression.**rle_encode**(*text*, *use_bwt=True*)

Perform encoding of run-length-encoding (RLE).

Cf. [RC67].

Based on http://rosettacode.org/wiki/Run-length_encoding#Python [Cod18b]. This is licensed GFDL 1.2.

Digits 0-9 cannot be in text.

**Parameters**

- **text** (*str*) – a text string to encode
- **use_bwt** (*bool*) – boolean indicating whether to perform BWT encoding before RLE encoding

**Returns** word decoded by BWT

**Return type** str

```
>>> rle_encode('align')
'n\x00ilag'
>>> rle_encode('align', use_bwt=False)
'align'
```

```
>>> rle_encode('banana')
'annb\x00aa'
>>> rle_encode('banana', use_bwt=False)
'banana'
```

```
>>> rle_encode('aaabaababababa')
'ab\x00abbab5a'
>>> rle_encode('aaabaababababa', False)
'3abaababababa'
```

#### 3.1.1.3 abydos.corpus module

abydos.corpus.

The corpus class is a container for linguistic corpora and includes various functions for corpus statistics, language modeling, etc.

**class** abydos.corpus.**Corpus**(*corpus_text=''*, *doc_split='nn'*, *sent_split='n'*, *filter_chars=''*, *stop_words=None*)

Bases: object

Corpus class.

Internally, this is a list of lists or lists. The corpus itself is a list of documents. Each document is an ordered list of sentences in those documents. And each sentence is an ordered list of words that make up that sentence.

**docs**()

Return the docs in the corpus.

Each list within a doc represents the sentences in that doc, each of which is in turn a list of words within that sentence.

**Returns** the paragraphs in the corpus as a list of lists of lists of strs

**Return type** [[[str]]]

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> corp.docs()
[[['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.'], ['And', 'then', 'it', 'slept.'], ['And', 'the', 'dog', 'ran',
'off.']]]
>>> len(corp.docs())
1
```

**docs_of_words**()

Return the docs in the corpus, with sentences flattened.

Each list within the corpus represents all the words of that document. Thus the sentence level of lists has been flattened.

**Returns** the docs in the corpus as a list of list of strs

**Return type** [[str]]

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
```

(continues on next page)

(continued from previous page)

```
>>> corp.docs_of_words()
[['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.', 'And', 'then', 'it', 'slept.', 'And', 'the', 'dog', 'ran',
'off.']]
>>> len(corp.docs_of_words())
1
```

**idf**(*term*, *transform=None*)

Calculate the Inverse Document Frequency of a term in the corpus.

**Parameters**

- **term** (*str*) – the term to calculate the IDF of
- **transform** (*function*) – a function to apply to each document term before checking for the presence of term

**Returns** the IDF

**Return type** float

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n\n'
>>> tqbf += 'And then it slept.\n\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> print(corp.docs())
[[['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.']],
[['And', 'then', 'it', 'slept.']],
[['And', 'the', 'dog', 'ran', 'off.']]]
>>> round(corp.idf('dog'), 10)
0.4771212547
>>> round(corp.idf('the'), 10)
0.1760912591
```

**paras**()

Return the paragraphs in the corpus.

Each list within a paragraph represents the sentences in that doc, each of which is in turn a list of words within that sentence. This is identical to the docs() member function and exists only to mirror part of NLTK's API for corpora.

**Returns** the paragraphs in the corpus as a list of lists of lists of strs

**Return type** [[[str]]]

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> corp.paras()
[[['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.'], ['And', 'then', 'it', 'slept.'], ['And', 'the', 'dog', 'ran',
'off.']]]
>>> len(corp.paras())
1
```

**raw**()

Return the raw corpus.

This is reconstructed by joining sub-components with the corpus' split characters

**Returns** the raw corpus

**Return type** str

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> print(corp.raw())
The quick brown fox jumped over the lazy dog.
And then it slept.
And the dog ran off.
>>> len(corp.raw())
85
```

**sents**()

Return the sentences in the corpus.

Each list within a sentence represents the words within that sentence.

**Returns** the sentences in the corpus as a list of lists of strs

**Return type** [[str]]

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> corp.sents()
[['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.'], ['And', 'then', 'it', 'slept.'], ['And', 'the', 'dog', 'ran',
'off.']]
>>> len(corp.sents())
3
```

**words**()

Return the words in the corpus as a single list.

**Returns** the words in the corpus as a list of strs

**Return type** [str]

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> corp = Corpus(tqbf)
>>> corp.words()
['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy',
'dog.', 'And', 'then', 'it', 'slept.', 'And', 'the', 'dog', 'ran',
'off.']
>>> len(corp.words())
18
```

#### 3.1.1.4 abydos.distance module

abydos.distance.

The distance module implements string edit distance functions including:

- Levenshtein distance
- Optimal String Alignment distance
- Levenshtein-Damerau distance

- Hamming distance
- Tversky index
- Sørensen–Dice coefficient & distance
- Jaccard similarity coefficient & distance
- overlap similarity & distance
- Tanimoto coefficient & distance
- Minkowski distance & similarity
- Manhattan distance & similarity
- Euclidean distance & similarity
- Chebyshev distance
- cosine similarity & distance
- Jaro distance
- Jaro-Winkler distance (incl. the strcmp95 algorithm variant)
- Longest common substring
- Ratcliff-Obershelp similarity & distance
- Match Rating Algorithm similarity
- Normalized Compression Distance (NCD) & similarity
- Monge-Elkan similarity & distance
- Matrix similarity
- Needleman-Wunsch score
- Smith-Waterman score
- Gotoh score
- Length similarity
- Prefix, Suffix, and Identity similarity & distance
- Modified Language-Independent Product Name Search (MLIPNS) similarity & distance
- Bag distance
- Editex distance
- Eudex distances
- Sift4 distance
- Baystat distance & similarity
- Typo distance
- Indel distance
- Synoname

Functions beginning with the prefixes 'sim' and 'dist' are guaranteed to be in the range [0, 1], and sim_X = 1 - dist_X since the two are complements. If a sim_X function is supplied identical src & tar arguments, it is guaranteed to return 1; the corresponding dist_X function is guaranteed to return 0.

abydos.distance.**bag**(*src*, *tar*)

Return the bag distance between two strings.

Bag distance is proposed in [BCP02]. It is defined as: $max(|multiset(src) - multiset(tar)|, |multiset(tar) - multiset(src)|)$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** bag distance

**Return type** int

```
>>> bag('cat', 'hat')
1
>>> bag('Niall', 'Neil')
2
>>> bag('aluminum', 'Catalan')
5
>>> bag('ATCG', 'TAGC')
0
>>> bag('abcdefg', 'hijklm')
7
>>> bag('abcdefg', 'hijklmno')
8
```

abydos.distance.**chebyshev**(*src*, *tar*, *qval=2*, *normalized=False*, *alphabet=None*)

Return the Chebyshev distance between two strings.

Euclidean distance is the chessboard distance, equivalent to Minkowski distance in $L^{\infty} - space$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **normalized** – normalizes to [0, 1] if True
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the Chebyshev distance

**Return type** float

```
>>> chebyshev('cat', 'hat')
1.0
>>> chebyshev('Niall', 'Neil')
1.0
>>> chebyshev('Colin', 'Cuilen')
1.0
>>> chebyshev('ATCG', 'TAGC')
1.0
>>> chebyshev('ATCG', 'TAGC', qval=1)
0.0
>>> chebyshev('ATCGATTCGGAATTTC', 'TAGCATAATCGCCG', qval=1)
3.0
```

abydos.distance.**damerau_levenshtein**(*src*, *tar*, *cost=(1, 1, 1, 1)*)

Return the Damerau-Levenshtein distance between two strings.

This computes the Damerau-Levenshtein distance [Dam64]. Damerau-Levenshtein code is based on Java code by Kevin L. Stern [Ste14], under the MIT license: https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/string/DamerauLevenshteinAlgorithm.java

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** the Damerau-Levenshtein distance between src & tar

**Return type** int (may return a float if cost has float values)

```
>>> damerau_levenshtein('cat', 'hat')
1
>>> damerau_levenshtein('Niall', 'Neil')
3
>>> damerau_levenshtein('aluminum', 'Catalan')
7
>>> damerau_levenshtein('ATCG', 'TAGC')
2
```

abydos.distance.**dist**(*src*, *tar*, *method=<function sim_levenshtein>*)

Return a distance between two strings.

This is a generalized function for calling other distance functions.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **method** (*function*) – specifies the similarity metric (Levenshtein by default) – Note that this takes a similarity metric function, not a distance metric function.

**Returns** distance according to the specified function

**Return type** float

```
>>> round(dist('cat', 'hat'), 12)
0.333333333333
>>> round(dist('Niall', 'Neil'), 12)
0.6
>>> dist('aluminum', 'Catalan')
0.875
>>> dist('ATCG', 'TAGC')
0.75
```

abydos.distance.**dist_bag**(*src*, *tar*)

Return the normalized bag distance between two strings.

Bag distance is normalized by dividing by $max(|src|, |tar|)$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** normalized bag distance

**Return type** float

```
>>> dist_bag('cat', 'hat')
0.3333333333333333
>>> dist_bag('Niall', 'Neil')
0.4
>>> dist_bag('aluminum', 'Catalan')
0.625
>>> dist_bag('ATCG', 'TAGC')
0.0
```

abydos.distance.**dist_baystat**(*src*, *tar*, *min_ss_len=None*, *left_ext=None*, *right_ext=None*)

Return the Baystat distance.

Normalized Baystat similarity is the complement of normalized Baystat distance: $sim_{Baystat} = 1 - dist_{Baystat}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **min_ss_len** (*int*) – minimum substring length to be considered
- **left_ext** (*int*) – left-side extension length
- **right_ext** (*int*) – right-side extension length

**Returns** the Baystat distance

**Return type** float

```
>>> round(dist_baystat('cat', 'hat'), 12)
0.333333333333
>>> dist_baystat('Niall', 'Neil')
0.6
>>> round(dist_baystat('Colin', 'Cuilen'), 12)
0.833333333333
>>> dist_baystat('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_compression**(*src*, *tar*, *compressor='bz2'*, *probs=None*)

Return the normalized compression distance between two strings.

Normalized compression distance (NCD) [CV05].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **compressor** (*str*) – a compression scheme to use for the similarity calculation, from the following:
  - *zlib* – standard zlib/gzip
  - *bz2* – bzip2 (default)
  - *lzma* – Lempel–Ziv–Markov chain algorithm

- *arith* – arithmetic coding
- *rle* – run-length encoding
- *bwtrle* – Burrows-Wheeler transform followed by run-length encoding

- **probs** (*dict*) – a dictionary trained with ac_train (for the arith compressor only)

**Returns** compression distance

**Return type** float

```
>>> dist_compression('cat', 'hat')
0.08
>>> dist_compression('Niall', 'Neil')
0.037037037037037035
>>> dist_compression('aluminum', 'Catalan')
0.20689655172413793
>>> dist_compression('ATCG', 'TAGC')
0.037037037037037035
```

```
>>> dist_compression('Niall', 'Neil', compressor='zlib')
0.45454545454545453
>>> dist_compression('Niall', 'Neil', compressor='bz2')
0.037037037037037035
>>> dist_compression('Niall', 'Neil', compressor='lzma')
0.16
>>> dist_compression('Niall', 'Neil', compressor='arith')
0.6875
>>> dist_compression('Niall', 'Neil', compressor='rle')
1.0
>>> dist_compression('Niall', 'Neil', compressor='bwtrle')
0.8333333333333334
```

abydos.distance.**dist_cosine**(*src*, *tar*, *qval=2*)

Return the cosine distance between two strings.

Cosine distance is the complement of cosine similarity: $dist_{cosine} = 1 - sim_{cosine}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** cosine distance

**Return type** float

```
>>> dist_cosine('cat', 'hat')
0.5
>>> dist_cosine('Niall', 'Neil')
0.6348516283298893
>>> dist_cosine('aluminum', 'Catalan')
0.882148869802242
>>> dist_cosine('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_damerau**(*src*, *tar*, *cost=(1, 1, 1, 1)*)

Return the Damerau-Levenshtein similarity of two strings.

Damerau-Levenshtein distance normalized to the interval [0, 1].

The Damerau-Levenshtein distance is normalized by dividing the Damerau-Levenshtein distance by the greater of the number of characters in src times the cost of a delete and the number of characters in tar times the cost of an insert. For the case in which all operations have $cost = 1$, this is equivalent to the greater of the length of the two strings src & tar.

The arguments are identical to those of the levenshtein() function.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** normalized Damerau-Levenshtein distance

**Return type** float

```
>>> round(dist_damerau('cat', 'hat'), 12)
0.333333333333
>>> round(dist_damerau('Niall', 'Neil'), 12)
0.6
>>> dist_damerau('aluminum', 'Catalan')
0.875
>>> dist_damerau('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_dice**(*src*, *tar*, *qval=2*)

Return the Sørensen–Dice distance between two strings.

Sørensen–Dice distance is the complemenjt of the Sørensen–Dice coefficient: $dist_{dice} = 1 - sim_{dice}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Sørensen–Dice distance

**Return type** float

```
>>> dist_dice('cat', 'hat')
0.5
>>> dist_dice('Niall', 'Neil')
0.6363636363636364
>>> dist_dice('aluminum', 'Catalan')
0.8823529411764706
>>> dist_dice('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_editex**(*src*, *tar*, *cost=(0, 1, 2)*, *local=False*)

Return the normalized Editex distance between two strings.

The Editex distance is normalized by dividing the Editex distance (calculated by any of the three supported methods) by the greater of the number of characters in src times the cost of a delete and the number of characters in tar times the cost of an insert. For the case in which all operations have $cost = 1$, this is equivalent to the greater of the length of the two strings src & tar.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 3-tuple representing the cost of the four possible edits: match, same-group, and mismatch respectively (by default: (0, 1, 2))
- **local** (*bool*) – if True, the local variant of Editex is used

**Returns** normalized Editex distance

**Return type** float

```
>>> round(dist_editex('cat', 'hat'), 12)
0.333333333333
>>> round(dist_editex('Niall', 'Neil'), 12)
0.2
>>> dist_editex('aluminum', 'Catalan')
0.75
>>> dist_editex('ATCG', 'TAGC')
0.75
```

abydos.distance.**dist_euclidean**(*src*, *tar*, *qval=2*, *alphabet=None*)

Return the normalized Euclidean distance between two strings.

The normalized Euclidean distance is a distance metric in $L^2 - space$, normalized to [0, 1].

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Euclidean distance

**Return type** float

```
>>> round(dist_euclidean('cat', 'hat'), 12)
0.57735026919
>>> round(dist_euclidean('Niall', 'Neil'), 12)
0.683130051064
>>> round(dist_euclidean('Colin', 'Cuilen'), 12)
0.727606875109
>>> dist_euclidean('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_eudex**(*src*, *tar*, *weights='exponential'*, *max_length=8*)

Return normalized Hamming distance between Eudex hashes of two terms.

This is Eudex distance normalized to [0, 1].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **iterable, or generator function weights** (*str,*) – the weights or weights generator function

- **max_length** – the number of characters to encode as a eudex hash

**Returns** the normalized Eudex distance

**Return type** float

```
>>> round(dist_eudex('cat', 'hat'), 12)
0.062745098039
>>> round(dist_eudex('Niall', 'Neil'), 12)
0.000980392157
>>> round(dist_eudex('Colin', 'Cuilen'), 12)
0.004901960784
>>> round(dist_eudex('ATCG', 'TAGC'), 12)
0.197549019608
```

abydos.distance.**dist_hamming**(*src*, *tar*, *diff_lens=True*)

Return the normalized Hamming distance between two strings.

Hamming distance normalized to the interval [0, 1].

The Hamming distance is normalized by dividing it by the greater of the number of characters in src & tar (unless diff_lens is set to False, in which case an exception is raised).

The arguments are identical to those of the hamming() function.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **diff_lens** (*bool*) – If True (default), this returns the Hamming distance for those characters that have a matching character in both strings plus the difference in the strings' lengths. This is equivalent to extending the shorter string with obligatorily non-matching characters. If False, an exception is raised in the case of strings of unequal lengths.

**Returns** normalized Hamming distance

**Return type** float

```
>>> round(dist_hamming('cat', 'hat'), 12)
0.333333333333
>>> dist_hamming('Niall', 'Neil')
0.6
>>> dist_hamming('aluminum', 'Catalan')
1.0
>>> dist_hamming('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_ident**(*src*, *tar*)

Return the identity distance between two strings.

This is 0 if the two strings are identical, otherwise 1, i.e. $dist_{identity} = 1 - sim_{identity}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** identity distance

**Return type** int

```
>>> dist_ident('cat', 'hat')
1
>>> dist_ident('cat', 'cat')
0
```

abydos.distance.**dist_indel**(*src*, *tar*)

Return the indel distance between two strings.

This is equivalent to levenshtein distance, when only inserts and deletes are possible.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** indel distance

**Return type** float

```
>>> round(dist_indel('cat', 'hat'), 12)
0.333333333333
>>> round(dist_indel('Niall', 'Neil'), 12)
0.333333333333
>>> round(dist_indel('Colin', 'Cuilen'), 12)
0.454545454545
>>> dist_indel('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_jaccard**(*src*, *tar*, *qval=2*)

Return the Jaccard distance between two strings.

Jaccard distance is the complement of the Jaccard similarity coefficient: $dist_{Jaccard} = 1 - sim_{Jaccard}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Jaccard distance

**Return type** float

```
>>> dist_jaccard('cat', 'hat')
0.6666666666666667
>>> dist_jaccard('Niall', 'Neil')
0.7777777777777778
>>> dist_jaccard('aluminum', 'Catalan')
0.9375
>>> dist_jaccard('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_jaro_winkler**(*src*, *tar*, *qval=1*, *mode='winkler'*, *long_strings=False*, *boost_threshold=0.7*, *scaling_factor=0.1*)

Return the Jaro or Jaro-Winkler distance between two strings.

Jaro(-Winkler) similarity is the complement of Jaro(-Winkler) distance: $sim_{Jaro(-Winkler)} = 1 - dist_{Jaro(-Winkler)}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **qval** (*int*) – the length of each q-gram (defaults to 1: character-wise matching)
- **mode** (*str*) – indicates which variant of this distance metric to compute:
  - 'winkler' – computes the Jaro-Winkler distance (default) which increases the score for matches near the start of the word
  - 'jaro' – computes the Jaro distance

The following arguments apply only when mode is 'winkler':

**Parameters**

- **long_strings** (*bool*) – set to True to "Increase the probability of a match when the number of matched characters is large. This option allows for a little more tolerance when the strings are large. It is not an appropriate test when comparing fixed length fields such as phone and social security numbers."
- **boost_threshold** (*float*) – a value between 0 and 1, below which the Winkler boost is not applied (defaults to 0.7)
- **scaling_factor** (*float*) – a value between 0 and 0.25, indicating by how much to boost scores for matching prefixes (defaults to 0.1)

**Returns** Jaro or Jaro-Winkler distance

**Return type** float

```
>>> round(dist_jaro_winkler('cat', 'hat'), 12)
0.222222222222
>>> round(dist_jaro_winkler('Niall', 'Neil'), 12)
0.195
>>> round(dist_jaro_winkler('aluminum', 'Catalan'), 12)
0.39880952381
>>> round(dist_jaro_winkler('ATCG', 'TAGC'), 12)
0.166666666667
```

```
>>> round(dist_jaro_winkler('cat', 'hat', mode='jaro'), 12)
0.222222222222
>>> round(dist_jaro_winkler('Niall', 'Neil', mode='jaro'), 12)
0.216666666667
>>> round(dist_jaro_winkler('aluminum', 'Catalan', mode='jaro'), 12)
0.39880952381
>>> round(dist_jaro_winkler('ATCG', 'TAGC', mode='jaro'), 12)
0.166666666667
```

abydos.distance.**dist_lcsseq**(*src*, *tar*)

Return the longest common subsequence distance between two strings.

Longest common subsequence distance ($dist_{LCSseq}$).

This employs the LCSseq function to derive a similarity metric: $dist_{LCSseq}(s,t) = 1 - sim_{LCSseq}(s,t)$

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** LCSseq distance

**Return type** float

```
>>> dist_lcsseq('cat', 'hat')
0.33333333333333337
>>> dist_lcsseq('Niall', 'Neil')
0.4
>>> dist_lcsseq('aluminum', 'Catalan')
0.625
>>> dist_lcsseq('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_lcsstr**(*src*, *tar*)

Return the longest common substring distance between two strings.

Longest common substring distance ($dist_{LCSstr}$).

This employs the LCS function to derive a similarity metric: $dist_{LCSstr}(s,t) = 1 - sim_{LCSstr}(s,t)$

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** LCSstr distance

**Return type** float

```
>>> dist_lcsstr('cat', 'hat')
0.33333333333333337
>>> dist_lcsstr('Niall', 'Neil')
0.8
>>> dist_lcsstr('aluminum', 'Catalan')
0.75
>>> dist_lcsstr('ATCG', 'TAGC')
0.75
```

abydos.distance.**dist_length**(*src*, *tar*)

Return the length distance between two strings.

Length distance is the complement of length similarity: $dist_{length} = 1 - sim_{length}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** length distance

**Return type** float

```
>>> dist_length('cat', 'hat')
0.0
>>> dist_length('Niall', 'Neil')
0.19999999999999996
>>> dist_length('aluminum', 'Catalan')
0.125
>>> dist_length('ATCG', 'TAGC')
0.0
```

abydos.distance.**dist_levenshtein**(*src*, *tar*, *mode='lev'*, *cost=(1, 1, 1, 1)*)

Return the normalized Levenshtein distance between two strings.

The Levenshtein distance is normalized by dividing the Levenshtein distance (calculated by any of the three supported methods) by the greater of the number of characters in src times the cost of a delete and the number of characters in tar times the cost of an insert. For the case in which all operations have $cost = 1$, this is equivalent to the greater of the length of the two strings src & tar.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **mode** (*str*) – specifies a mode for computing the Levenshtein distance:
  - 'lev' (default) computes the ordinary Levenshtein distance, in which edits may include inserts, deletes, and substitutions
  - 'osa' computes the Optimal String Alignment distance, in which edits may include inserts, deletes, substitutions, and transpositions but substrings may only be edited once
  - 'dam' computes the Damerau-Levenshtein distance, in which edits may include inserts, deletes, substitutions, and transpositions and substrings may undergo repeated edits
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** normalized Levenshtein distance

**Return type** float

```
>>> round(dist_levenshtein('cat', 'hat'), 12)
0.333333333333
>>> round(dist_levenshtein('Niall', 'Neil'), 12)
0.6
>>> dist_levenshtein('aluminum', 'Catalan')
0.875
>>> dist_levenshtein('ATCG', 'TAGC')
0.75
```

abydos.distance.**dist_manhattan**(*src*, *tar*, *qval=2*, *alphabet=None*)

Return the normalized Manhattan distance between two strings.

The normalized Manhattan distance is a distance metric in $L^1 - space$, normalized to [0, 1].

This is identical to Canberra distance.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Manhattan distance

**Return type** float

```
>>> dist_manhattan('cat', 'hat')
0.5
>>> round(dist_manhattan('Niall', 'Neil'), 12)
0.636363636364
>>> round(dist_manhattan('Colin', 'Cuilen'), 12)
```

(continues on next page)

(continued from previous page)

```
0.692307692308
>>> dist_manhattan('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_minkowski**(*src*, *tar*, *qval=2*, *pval=1*, *alphabet=None*)

Return normalized Minkowski distance of two strings.

The normalized Minkowski distance [Min10] is a distance metric in $L^p - space$, normalized to [0, 1].

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or float pval** (*int*) – the $p$-value of the $L^p$-space.
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Minkowski distance

**Return type** float

```
>>> dist_minkowski('cat', 'hat')
0.5
>>> round(dist_minkowski('Niall', 'Neil'), 12)
0.636363636364
>>> round(dist_minkowski('Colin', 'Cuilen'), 12)
0.692307692308
>>> dist_minkowski('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_mlipns**(*src*, *tar*, *threshold=0.25*, *max_mismatches=2*)

Return the MLIPNS distance between two strings.

MLIPNS distance is the complement of MLIPNS similarity: $dist_{MLIPNS} = 1 - sim_{MLIPNS}$. This function returns only 0.0 (distant) or 1.0 (not distant).

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **threshold** (*float*) – a number [0, 1] indicating the maximum similarity score, below which the strings are considered 'similar' (0.25 by default)
- **max_mismatches** (*int*) – a number indicating the allowable number of mismatches to remove before declaring two strings not similar (2 by default)

**Returns** MLIPNS distance

**Return type** float

```
>>> dist_mlipns('cat', 'hat')
0.0
>>> dist_mlipns('Niall', 'Neil')
1.0
>>> dist_mlipns('aluminum', 'Catalan')
1.0
```

(continues on next page)

(continued from previous page)

```
>>> dist_mlipns('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_monge_elkan**(*src*, *tar*, *sim_func=<function sim_levenshtein>*, *symmetric=False*)

Return the Monge-Elkan distance between two strings.

Monge-Elkan distance is the complement of Monge-Elkan similarity: $dist_{Monge-Elkan} = 1 - sim_{Monge-Elkan}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **sim_func** (*function*) – the internal similarity metric to employ
- **symmetric** (*bool*) – return a symmetric similarity measure

**Returns** Monge-Elkan distance

**Return type** float

```
>>> dist_monge_elkan('cat', 'hat')
0.25
>>> round(dist_monge_elkan('Niall', 'Neil'), 12)
0.333333333333
>>> round(dist_monge_elkan('aluminum', 'Catalan'), 12)
0.611111111111
>>> dist_monge_elkan('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_mra**(*src*, *tar*)

Return the normalized MRA distance between two strings.

MRA distance is the complement of MRA similarity: $dist_{MRA} = 1 - sim_{MRA}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** normalized MRA distance

**Return type** float

```
>>> dist_mra('cat', 'hat')
0.16666666666666663
>>> dist_mra('Niall', 'Neil')
0.0
>>> dist_mra('aluminum', 'Catalan')
1.0
>>> dist_mra('ATCG', 'TAGC')
0.16666666666666663
```

abydos.distance.**dist_overlap**(*src*, *tar*, *qval=2*)

Return the overlap distance between two strings.

Overlap distance is the complement of the overlap coefficient: $sim_{overlap} = 1 - dist_{overlap}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** overlap distance

**Return type** float

```
>>> dist_overlap('cat', 'hat')
0.5
>>> dist_overlap('Niall', 'Neil')
0.6
>>> dist_overlap('aluminum', 'Catalan')
0.875
>>> dist_overlap('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_prefix**(*src*, *tar*)

Return the prefix distance between two strings.

Prefix distance is the complement of prefix similarity: $dist_{prefix} = 1 - sim_{prefix}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** prefix distance

**Return type** float

```
>>> dist_prefix('cat', 'hat')
1.0
>>> dist_prefix('Niall', 'Neil')
0.75
>>> dist_prefix('aluminum', 'Catalan')
1.0
>>> dist_prefix('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_ratcliff_obershelp**(*src*, *tar*)

Return the Ratcliff-Obershelp distance between two strings.

Ratcliff-Obsershelp distance the complement of Ratcliff-Obershelp similarity: $dist_{Ratcliff-Obershelp} = 1 - sim_{Ratcliff-Obershelp}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** Ratcliff-Obershelp distance

**Return type** float

```
>>> round(dist_ratcliff_obershelp('cat', 'hat'), 12)
0.333333333333
>>> round(dist_ratcliff_obershelp('Niall', 'Neil'), 12)
0.333333333333
```

(continues on next page)

(continued from previous page)

```
>>> round(dist_ratcliff_obershelp('aluminum', 'Catalan'), 12)
0.6
>>> dist_ratcliff_obershelp('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_sift4**(*src*, *tar*, *max_offset=5*, *max_distance=0*)

Return the normalized "common" Sift4 distance between two terms.

This is Sift4 distance, normalized to [0, 1].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **max_offset** – the number of characters to search for matching letters
- **max_distance** – the distance at which to stop and exit

**Returns** the normalized Sift4 distance

**Return type** float

```
>>> round(dist_sift4('cat', 'hat'), 12)
0.333333333333
>>> dist_sift4('Niall', 'Neil')
0.4
>>> dist_sift4('Colin', 'Cuilen')
0.5
>>> dist_sift4('ATCG', 'TAGC')
0.5
```

abydos.distance.**dist_strcmp95**(*src*, *tar*, *long_strings=False*)

Return the strcmp95 distance between two strings.

strcmp95 distance is the complement of strcmp95 similarity: $dist_{strcmp95} = 1 - sim_{strcmp95}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **long_strings** (*bool*) – set to True to "Increase the probability of a match when the number of matched characters is large. This option allows for a little more tolerance when the strings are large. It is not an appropriate test when comparing fixed length fields such as phone and social security numbers."

**Returns** strcmp95 distance

**Return type** float

```
>>> round(dist_strcmp95('cat', 'hat'), 12)
0.222222222222
>>> round(dist_strcmp95('Niall', 'Neil'), 12)
0.1545
>>> round(dist_strcmp95('aluminum', 'Catalan'), 12)
0.345238095238
>>> round(dist_strcmp95('ATCG', 'TAGC'), 12)
0.166666666667
```

abydos.distance.**dist_suffix**(*src*, *tar*)

Return the suffix distance between two strings.

Suffix distance is the complement of suffix similarity: $dist_{suffix} = 1 - sim_{suffix}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** suffix distance

**Return type** float

```
>>> dist_suffix('cat', 'hat')
0.3333333333333337
>>> dist_suffix('Niall', 'Neil')
0.75
>>> dist_suffix('aluminum', 'Catalan')
1.0
>>> dist_suffix('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_tversky**(*src*, *tar*, *qval=2*, *alpha=1*, *beta=1*, *bias=None*)

Return the Tversky distance between two strings.

Tversky distance is the complement of the Tvesrsky index (similarity): $dist_{Tversky} = 1 - sim_{Tversky}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **alpha** (*float*) – the Tversky index's alpha parameter
- **beta** (*float*) – the Tversky index's beta parameter
- **bias** (*float*) – The symmetric Tversky index bias parameter

**Returns** Tversky distance

**Return type** float

```
>>> dist_tversky('cat', 'hat')
0.6666666666666667
>>> dist_tversky('Niall', 'Neil')
0.7777777777777778
>>> dist_tversky('aluminum', 'Catalan')
0.9375
>>> dist_tversky('ATCG', 'TAGC')
1.0
```

abydos.distance.**dist_typo**(*src*, *tar*, *metric='euclidean'*, *cost=(1, 1, 0.5, 0.5)*)

Return the normalized typo distance between two strings.

This is typo distance, normalized to [0, 1].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

- **metric** (*str*) – supported values include: 'euclidean', 'manhattan', 'log-euclidean', and 'log-manhattan'
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and shift, respectively (by default: (1, 1, 0.5, 0.5)) The substitution & shift costs should be significantly less than the cost of an insertion & deletion unless a log metric is used.

**Returns** normalized typo distance

**Return type** float

```
>>> round(dist_typo('cat', 'hat'), 12)
0.527046283086
>>> round(dist_typo('Niall', 'Neil'), 12)
0.565028142929
>>> round(dist_typo('Colin', 'Cuilen'), 12)
0.569035609563
>>> dist_typo('ATCG', 'TAGC')
0.625
```

abydos.distance.**editex**(*src*, *tar*, *cost=(0, 1, 2)*, *local=False*)

Return the Editex distance between two strings.

As described on pages 3 & 4 of [ZD96].

The local variant is based on [RU09].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 3-tuple representing the cost of the four possible edits: match, same-group, and mismatch respectively (by default: (0, 1, 2))
- **local** (*bool*) – if True, the local variant of Editex is used

**Returns** Editex distance

**Return type** int

```
>>> editex('cat', 'hat')
2
>>> editex('Niall', 'Neil')
2
>>> editex('aluminum', 'Catalan')
12
>>> editex('ATCG', 'TAGC')
6
```

abydos.distance.**euclidean**(*src*, *tar*, *qval=2*, *normalized=False*, *alphabet=None*)

Return the Euclidean distance between two strings.

Euclidean distance is the straigh-line or as-the-crow-flies distance, equivalent to Minkowski distance in $L^2$-space.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison

- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **normalized** – normalizes to [0, 1] if True
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the Euclidean distance

**Return type** float

```
>>> euclidean('cat', 'hat')
2.0
>>> round(euclidean('Niall', 'Neil'), 12)
2.645751311065
>>> euclidean('Colin', 'Cuilen')
3.0
>>> round(euclidean('ATCG', 'TAGC'), 12)
3.162277660168
```

abydos.distance.**eudex**(*word*, *max_length=8*)

Return the eudex phonetic hash of a word.

This implementation of eudex phonetic hashing is based on the specification (not the reference implementation) at [Tic].

Further details can be found at [Tic16].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length in bits of the code returned (default 8)

**Returns** the eudex hash

**Return type** int

```
>>> eudex('Colin')
432345564238053650
>>> eudex('Christopher')
433648490138894409
>>> eudex('Niall')
648518346341351840
>>> eudex('Smith')
720575940412906756
>>> eudex('Schmidt')
720589151732307997
```

abydos.distance.**eudex_hamming**(*src*, *tar*, *weights='exponential'*, *max_length=8*, *normalized=False*)

Calculate the Hamming distance between the Eudex hashes of two terms.

Cf. [Tic].

- If weights is set to None, a simple Hamming distance is calculated.
- If weights is set to 'exponential', weight decays by powers of 2, as proposed in the eudex specification: https://github.com/ticki/eudex.
- If weights is set to 'fibonacci', weight decays through the Fibonacci series, as in the eudex reference implementation.
- If weights is set to a callable function, this assumes it creates a generator and the generator is used to populate a series of weights.

- If weights is set to an iterable, the iterable's values should be integers and will be used as the weights.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **iterable, or generator function weights** (*str,*) – the weights or weights generator function
- **max_length** – the number of characters to encode as a eudex hash
- **normalized** (*bool*) – normalizes to [0, 1] if True

**Returns** the Eudex Hamming distance

**Return type** int

```
>>> eudex_hamming('cat', 'hat')
128
>>> eudex_hamming('Niall', 'Neil')
2
>>> eudex_hamming('Colin', 'Cuilen')
10
>>> eudex_hamming('ATCG', 'TAGC')
403
```

```
>>> eudex_hamming('cat', 'hat', weights='fibonacci')
34
>>> eudex_hamming('Niall', 'Neil', weights='fibonacci')
2
>>> eudex_hamming('Colin', 'Cuilen', weights='fibonacci')
7
>>> eudex_hamming('ATCG', 'TAGC', weights='fibonacci')
117
```

```
>>> eudex_hamming('cat', 'hat', weights=None)
1
>>> eudex_hamming('Niall', 'Neil', weights=None)
1
>>> eudex_hamming('Colin', 'Cuilen', weights=None)
2
>>> eudex_hamming('ATCG', 'TAGC', weights=None)
9
```

```
>>> # Using the OEIS A000142:
>>> eudex_hamming('cat', 'hat', [1, 1, 2, 6, 24, 120, 720, 5040])
1
>>> eudex_hamming('Niall', 'Neil', [1, 1, 2, 6, 24, 120, 720, 5040])
720
>>> eudex_hamming('Colin', 'Cuilen', [1, 1, 2, 6, 24, 120, 720, 5040])
744
>>> eudex_hamming('ATCG', 'TAGC', [1, 1, 2, 6, 24, 120, 720, 5040])
6243
```

abydos.distance.**gotoh**(*src*, *tar*, *gap_open=1*, *gap_ext=0.4*, *sim_func=<function sim_ident>*)

Return the Gotoh score of two strings.

The Gotoh score [Got82] is essentially Needleman-Wunsch with affine gap penalties.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **gap_open** (*float*) – the cost of an open alignment gap (1 by default)
- **gap_ext** (*float*) – the cost of an alignment gap extension (0.4 by default)
- **sim_func** (*function*) – a function that returns the similarity of two characters (identity similarity by default)

**Returns** Gotoh score

**Return type** float

```
>>> gotoh('cat', 'hat')
2.0
>>> gotoh('Niall', 'Neil')
1.0
>>> round(gotoh('aluminum', 'Catalan'), 12)
-0.4
>>> gotoh('cat', 'hat')
2.0
```

abydos.distance.**hamming**(*src*, *tar*, *diff_lens=True*)

Return the Hamming distance between two strings.

Hamming distance [Ham50] equals the number of character positions at which two strings differ. For strings of unequal lengths, it is not normally defined. By default, this implementation calculates the Hamming distance of the first n characters where n is the lesser of the two strings' lengths and adds to this the difference in string lengths.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **diff_lens** (*bool*) – If True (default), this returns the Hamming distance for those characters that have a matching character in both strings plus the difference in the strings' lengths. This is equivalent to extending the shorter string with obligatorily non-matching characters. If False, an exception is raised in the case of strings of unequal lengths.

**Returns** the Hamming distance between src & tar

**Return type** int

```
>>> hamming('cat', 'hat')
1
>>> hamming('Niall', 'Neil')
3
>>> hamming('aluminum', 'Catalan')
8
>>> hamming('ATCG', 'TAGC')
4
```

abydos.distance.**lcsseq**(*src*, *tar*)

Return the longest common subsequence of two strings.

Longest common subsequence (LCSseq) is the longest subsequence of characters that two strings have in common.

Based on the dynamic programming algorithm from http://rosettacode.org/wiki/Longest_common_subsequence#Dynamic_Programming_6 [Cod18a]. This is licensed GFDL 1.2.

**Modifications include:** conversion to a numpy array in place of a list of lists

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** the longest common subsequence

**Return type** str

```
>>> lcsseq('cat', 'hat')
'at'
>>> lcsseq('Niall', 'Neil')
'Nil'
>>> lcsseq('aluminum', 'Catalan')
'aln'
>>> lcsseq('ATCG', 'TAGC')
'AC'
```

abydos.distance.**lcsstr**(*src*, *tar*)

Return the longest common substring of two strings.

Longest common substring (LCSstr).

Based on the code from https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Longest_common_substring#Python [Wik18]. This is licensed Creative Commons: Attribution-ShareAlike 3.0.

Modifications include:

- conversion to a numpy array in place of a list of lists
- conversion to Python 2/3-safe range from xrange via six

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** the longest common substring

**Return type** str

```
>>> lcsstr('cat', 'hat')
'at'
>>> lcsstr('Niall', 'Neil')
'N'
>>> lcsstr('aluminum', 'Catalan')
'al'
>>> lcsstr('ATCG', 'TAGC')
'A'
```

abydos.distance.**levenshtein**(*src*, *tar*, *mode='lev'*, *cost=(1, 1, 1, 1)*)

Return the Levenshtein distance between two strings.

This is the standard edit distance measure. Cf. [65][Lev66].

Two additional variants: optimal string alignment (aka restricted Damerau-Levenshtein distance) [Boy11] and the Damerau-Levenshtein [Dam64] distance are also supported.

The ordinary Levenshtein & Optimal String Alignment distance both employ the Wagner-Fischer dynamic programming algorithm [WF74].

Levenshtein edit distance ordinarily has unit insertion, deletion, and substitution costs.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **mode** (*str*) – specifies a mode for computing the Levenshtein distance:
  - 'lev' (default) computes the ordinary Levenshtein distance, in which edits may include inserts, deletes, and substitutions
  - 'osa' computes the Optimal String Alignment distance, in which edits may include inserts, deletes, substitutions, and transpositions but substrings may only be edited once
  - 'dam' computes the Damerau-Levenshtein distance, in which edits may include inserts, deletes, substitutions, and transpositions and substrings may undergo repeated edits
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** the Levenshtein distance between src & tar

**Return type** int (may return a float if cost has float values)

```
>>> levenshtein('cat', 'hat')
1
>>> levenshtein('Niall', 'Neil')
3
>>> levenshtein('aluminum', 'Catalan')
7
>>> levenshtein('ATCG', 'TAGC')
3
```

```
>>> levenshtein('ATCG', 'TAGC', mode='osa')
2
>>> levenshtein('ACTG', 'TAGC', mode='osa')
4
```

```
>>> levenshtein('ATCG', 'TAGC', mode='dam')
2
>>> levenshtein('ACTG', 'TAGC', mode='dam')
3
```

abydos.distance.**manhattan**(*src*, *tar*, *qval=2*, *normalized=False*, *alphabet=None*)

Return the Manhattan distance between two strings.

Manhattan distance is the city-block or taxi-cab distance, equivalent to Minkowski distance in $L^1$-space.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

- **normalized** – normalizes to [0, 1] if True
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the Manhattan distance

**Return type** float

```
>>> manhattan('cat', 'hat')
4.0
>>> manhattan('Niall', 'Neil')
7.0
>>> manhattan('Colin', 'Cuilen')
9.0
>>> manhattan('ATCG', 'TAGC')
10.0
```

abydos.distance.**minkowski**(*src*, *tar*, *qval=2*, *pval=1*, *normalized=False*, *alphabet=None*)

Return the Minkowski distance ($L^p - norm$) of two strings.

The Minkowski distance [Min10] is a distance metric in $L^p - space$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or float pval** (*int*) – the $p$-value of the $L^p$-space.
- **normalized** (*bool*) – normalizes to [0, 1] if True
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the Minkowski distance

**Return type** float

```
>>> minkowski('cat', 'hat')
4.0
>>> minkowski('Niall', 'Neil')
7.0
>>> minkowski('Colin', 'Cuilen')
9.0
>>> minkowski('ATCG', 'TAGC')
10.0
```

abydos.distance.**mra_compare**(*src*, *tar*)

Return the MRA comparison rating of two strings.

The Western Airlines Surname Match Rating Algorithm comparison rating, as presented on page 18 of [MKTM77].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** MRA comparison rating

**Return type** int

```
>>> mra_compare('cat', 'hat')
5
>>> mra_compare('Niall', 'Neil')
6
>>> mra_compare('aluminum', 'Catalan')
0
>>> mra_compare('ATCG', 'TAGC')
5
```

abydos.distance.**needleman_wunsch**(*src*, *tar*, *gap_cost=1*, *sim_func=<function sim_ident>*)

Return the Needleman-Wunsch score of two strings.

The Needleman-Wunsch score [NW70] is a standard edit distance measure.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **gap_cost** (*float*) – the cost of an alignment gap (1 by default)
- **sim_func** (*function*) – a function that returns the similarity of two characters (identity similarity by default)

**Returns** Needleman-Wunsch score

**Return type** float

```
>>> needleman_wunsch('cat', 'hat')
2.0
>>> needleman_wunsch('Niall', 'Neil')
1.0
>>> needleman_wunsch('aluminum', 'Catalan')
-1.0
>>> needleman_wunsch('ATCG', 'TAGC')
0.0
```

abydos.distance.**sift4_common**(*src*, *tar*, *max_offset=5*, *max_distance=0*)

Return the "common" Sift4 distance between two terms.

This is an approximation of edit distance, described in [Zac14].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **max_offset** – the number of characters to search for matching letters
- **max_distance** – the distance at which to stop and exit

**Returns** the Sift4 distance according to the common formula

**Return type** int

```
>>> sift4_common('cat', 'hat')
1
>>> sift4_common('Niall', 'Neil')
2
>>> sift4_common('Colin', 'Cuilen')
3
```

(continues on next page)

(continued from previous page)

```
>>> sift4_common('ATCG', 'TAGC')
2
```

abydos.distance.**sift4_simplest**(*src*, *tar*, *max_offset=5*)

Return the "simplest" Sift4 distance between two terms.

This is an approximation of edit distance, described in [Zac14].

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **max_offset** – the number of characters to search for matching letters

**Returns** the Sift4 distance according to the simplest formula

**Return type** int

```
>>> sift4_simplest('cat', 'hat')
1
>>> sift4_simplest('Niall', 'Neil')
2
>>> sift4_simplest('Colin', 'Cuilen')
3
>>> sift4_simplest('ATCG', 'TAGC')
2
```

abydos.distance.**sim**(*src*, *tar*, *method=<function sim_levenshtein>*)

Return a similarity of two strings.

This is a generalized function for calling other similarity functions.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **method** (*function*) – specifies the similarity metric (Levenshtein by default)

**Returns** similarity according to the specified function

**Return type** float

```
>>> round(sim('cat', 'hat'), 12)
0.666666666667
>>> round(sim('Niall', 'Neil'), 12)
0.4
>>> sim('aluminum', 'Catalan')
0.125
>>> sim('ATCG', 'TAGC')
0.25
```

abydos.distance.**sim_bag**(*src*, *tar*)

Return the normalized bag similarity of two strings.

Normalized bag similarity is the complement of normalized bag distance: $sim_{bag} = 1 - dist_{bag}$.

**Parameters**

- **src** (*str*) – source string for comparison

- **tar** (*str*) – target string for comparison

**Returns** normalized bag similarity

**Return type** float

```
>>> round(sim_bag('cat', 'hat'), 12)
0.666666666667
>>> sim_bag('Niall', 'Neil')
0.6
>>> sim_bag('aluminum', 'Catalan')
0.375
>>> sim_bag('ATCG', 'TAGC')
1.0
```

abydos.distance.**sim_baystat**(*src*, *tar*, *min_ss_len=None*, *left_ext=None*, *right_ext=None*)

Return the Baystat similarity.

Good results for shorter words are reported when setting min_ss_len to 1 and either left_ext OR right_ext to 1.

The Baystat similarity is defined in [FurnrohrRvR02].

This is ostensibly a port of the R module PPRL's implementation: https://github.com/cran/PPRL/blob/master/src/MTB_Baystat.cpp [Ruk18]. As such, this could be made more pythonic.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **min_ss_len** (*int*) – minimum substring length to be considered
- **left_ext** (*int*) – left-side extension length
- **right_ext** (*int*) – right-side extension length

**Returns** the Baystat similarity

**Return type** float

```
>>> round(sim_baystat('cat', 'hat'), 12)
0.666666666667
>>> sim_baystat('Niall', 'Neil')
0.4
>>> round(sim_baystat('Colin', 'Cuilen'), 12)
0.166666666667
>>> sim_baystat('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_compression**(*src*, *tar*, *compressor='bz2'*, *probs=None*)

Return the normalized compression similarity of two strings.

Normalized compression similarity is the complement of normalized compression distance: $sim_{NCS} = 1 - dist_{NCD}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **compressor** (*str*) – a compression scheme to use for the similarity calculation:
  - *zlib* – standard zlib/gzip

- *bz2* – bzip2 (default)
- *lzma* – Lempel–Ziv–Markov chain algorithm
- *arith* – arithmetic coding
- *rle* – run-length encoding
- *bwtrle* – Burrows-Wheeler transform followed by run-length encoding

- **probs** (*dict*) – a dictionary trained with ac_train (for the arith compressor only)

**Returns** compression similarity

**Return type** float

```
>>> sim_compression('cat', 'hat')
0.92
>>> sim_compression('Niall', 'Neil')
0.962962962962963
>>> sim_compression('aluminum', 'Catalan')
0.7931034482758621
>>> sim_compression('ATCG', 'TAGC')
0.962962962962963
```

```
>>> sim_compression('Niall', 'Neil', compressor='zlib')
0.5454545454545454
>>> sim_compression('Niall', 'Neil', compressor='bz2')
0.962962962962963
>>> sim_compression('Niall', 'Neil', compressor='lzma')
0.84
>>> sim_compression('Niall', 'Neil', compressor='arith')
0.3125
>>> sim_compression('Niall', 'Neil', compressor='rle')
0.0
>>> sim_compression('Niall', 'Neil', compressor='bwtrle')
0.16666666666666663
```

abydos.distance.**sim_cosine**(*src*, *tar*, *qval=2*)

Return the cosine similarity of two strings.

For two sets X and Y, the cosine similarity, Otsuka-Ochiai coefficient, or Ochiai coefficient [Ots36][Och57] is:
$sim_{cosine}(X, Y) =$
$frac|XcapY|sqrt|X|cdot|Y|$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** cosine similarity

**Return type** float

```
>>> sim_cosine('cat', 'hat')
0.5
>>> sim_cosine('Niall', 'Neil')
0.3651483716701107
>>> sim_cosine('aluminum', 'Catalan')
```

(continues on next page)

(continued from previous page)

```
0.11785113019775793
>>> sim_cosine('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_damerau**(*src*, *tar*, *cost=(1, 1, 1, 1)*)

Return the Damerau-Levenshtein similarity of two strings.

Normalized Damerau-Levenshtein similarity the complement of normalized Damerau-Levenshtein distance: $sim_{Damerau} = 1 - dist_{Damerau}$.

The arguments are identical to those of the levenshtein() function.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** normalized Damerau-Levenshtein similarity

**Return type** float

```
>>> round(sim_damerau('cat', 'hat'), 12)
0.666666666667
>>> round(sim_damerau('Niall', 'Neil'), 12)
0.4
>>> sim_damerau('aluminum', 'Catalan')
0.125
>>> sim_damerau('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_dice**(*src*, *tar*, *qval=2*)

Return the Sørensen–Dice coefficient of two strings.

For two sets X and Y, the Sørensen–Dice coefficient [Dic45][Sorensen48] is $sim_{dice}(X, Y) = frac2cdot|XcapY||X| + |Y|$.

This is identical to the Tanimoto similarity coefficient [Tan58] and the Tversky index [Tve77] for
$alpha =$
$beta = 0.5$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Sørensen–Dice similarity

**Return type** float

```
>>> sim_dice('cat', 'hat')
0.5
>>> sim_dice('Niall', 'Neil')
0.36363636363636365
>>> sim_dice('aluminum', 'Catalan')
0.11764705882352941
```

(continues on next page)

(continued from previous page)

```
>>> sim_dice('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_editex**(*src*, *tar*, *cost=(0, 1, 2)*, *local=False*)

Return the normalized Editex similarity of two strings.

The Editex similarity is the complement of Editex distance: $sim_{Editex} = 1 - dist_{Editex}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **cost** (*tuple*) – a 3-tuple representing the cost of the four possible edits: match, same-group, and mismatch respectively (by default: (0, 1, 2))
- **local** (*bool*) – if True, the local variant of Editex is used

**Returns** normalized Editex similarity

**Return type** float

```
>>> round(sim_editex('cat', 'hat'), 12)
0.666666666667
>>> round(sim_editex('Niall', 'Neil'), 12)
0.8
>>> sim_editex('aluminum', 'Catalan')
0.25
>>> sim_editex('ATCG', 'TAGC')
0.25
```

abydos.distance.**sim_euclidean**(*src*, *tar*, *qval=2*, *alphabet=None*)

Return the normalized Euclidean similarity of two strings.

Euclidean similarity is the complement of Euclidean distance: $sim_{Euclidean} = 1 - dist_{Euclidean}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Euclidean similarity

**Return type** float

```
>>> round(sim_euclidean('cat', 'hat'), 12)
0.42264973081
>>> round(sim_euclidean('Niall', 'Neil'), 12)
0.316869948936
>>> round(sim_euclidean('Colin', 'Cuilen'), 12)
0.272393124891
>>> sim_euclidean('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_eudex**(*src*, *tar*, *weights='exponential'*, *max_length=8*)

Return normalized Hamming similarity between Eudex hashes of two terms.

Normalized Eudex similarity is the complement of normalized Eudex distance: $sim_{Eudex} = 1 - dist_{Eudex}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **iterable, or generator function weights** (*str,*) – the weights or weights generator function
- **max_length** – the number of characters to encode as a eudex hash

**Returns** the normalized Eudex similarity

**Return type** float

```
>>> round(sim_eudex('cat', 'hat'), 12)
0.937254901961
>>> round(sim_eudex('Niall', 'Neil'), 12)
0.999019607843
>>> round(sim_eudex('Colin', 'Cuilen'), 12)
0.995098039216
>>> round(sim_eudex('ATCG', 'TAGC'), 12)
0.802450980392
```

abydos.distance.**sim_hamming**(*src*, *tar*, *diff_lens=True*)

Return the normalized Hamming similarity of two strings.

Hamming similarity normalized to the interval [0, 1].

Hamming similarity is the complement of normalized Hamming distance: $sim_{Hamming} = 1 - distHamming$.

Provided that diff_lens==True, the Hamming similarity is identical to the Language-Independent Product Name Search (LIPNS) similarity score. For further information, see the sim_mlipns documentation.

The arguments are identical to those of the hamming() function.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **diff_lens** (*bool*) – If True (default), this returns the Hamming distance for those characters that have a matching character in both strings plus the difference in the strings' lengths. This is equivalent to extending the shorter string with obligatorily non-matching characters. If False, an exception is raised in the case of strings of unequal lengths.

**Returns** normalized Hamming similarity

**Return type** float

```
>>> round(sim_hamming('cat', 'hat'), 12)
0.666666666667
>>> sim_hamming('Niall', 'Neil')
0.4
>>> sim_hamming('aluminum', 'Catalan')
0.0
>>> sim_hamming('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_ident**(*src*, *tar*)

Return the identity similarity of two strings.

Identity similarity is 1 if the two strings are identical, otherwise 0.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** identity similarity

**Return type** int

```
>>> sim_ident('cat', 'hat')
0
>>> sim_ident('cat', 'cat')
1
```

abydos.distance.**sim_indel**(*src*, *tar*)

Return the indel similarity of two strings.

This is equivalent to levenshtein similarity, when only inserts and deletes are possible.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** indel similarity

**Return type** float

```
>>> round(sim_indel('cat', 'hat'), 12)
0.666666666667
>>> round(sim_indel('Niall', 'Neil'), 12)
0.666666666667
>>> round(sim_indel('Colin', 'Cuilen'), 12)
0.545454545455
>>> sim_indel('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_jaccard**(*src*, *tar*, *qval=2*)

Return the Jaccard similarity of two strings.

For two sets X and Y, the Jaccard similarity coefficient [Jac01] is $sim_{jaccard}(X, Y) = frac|XcapY||XcupY|$.

This is identical to the Tanimoto similarity coefficient [Tan58] and the Tversky index [Tve77] for $alpha =$
$beta = 1$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Jaccard similarity

**Return type** float

```
>>> sim_jaccard('cat', 'hat')
0.3333333333333333
>>> sim_jaccard('Niall', 'Neil')
0.2222222222222222
>>> sim_jaccard('aluminum', 'Catalan')
0.0625
>>> sim_jaccard('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_jaro_winkler**(*src*, *tar*, *qval=1*, *mode='winkler'*, *long_strings=False*, *boost_threshold=0.7*, *scaling_factor=0.1*)

Return the Jaro or Jaro-Winkler similarity of two strings.

Jaro(-Winkler) distance is a string edit distance initially proposed by Jaro and extended by Winkler [Jar89][Win90].

This is Python based on the C code for strcmp95: http://web.archive.org/web/20110629121242/http://www.census.gov/geo/msb/stand/strcmp.c [WMJL94]. The above file is a US Government publication and, accordingly, in the public domain.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **qval** (*int*) – the length of each q-gram (defaults to 1: character-wise matching)
- **mode** (*str*) – indicates which variant of this distance metric to compute:
  - 'winkler' – computes the Jaro-Winkler distance (default) which increases the score for matches near the start of the word
  - 'jaro' – computes the Jaro distance

The following arguments apply only when mode is 'winkler':

**Parameters**

- **long_strings** (*bool*) – set to True to "Increase the probability of a match when the number of matched characters is large. This option allows for a little more tolerance when the strings are large. It is not an appropriate test when comparing fixed length fields such as phone and social security numbers."
- **boost_threshold** (*float*) – a value between 0 and 1, below which the Winkler boost is not applied (defaults to 0.7)
- **scaling_factor** (*float*) – a value between 0 and 0.25, indicating by how much to boost scores for matching prefixes (defaults to 0.1)

**Returns** Jaro or Jaro-Winkler similarity

**Return type** float

```
>>> round(sim_jaro_winkler('cat', 'hat'), 12)
0.777777777778
>>> round(sim_jaro_winkler('Niall', 'Neil'), 12)
0.805
>>> round(sim_jaro_winkler('aluminum', 'Catalan'), 12)
0.60119047619
>>> round(sim_jaro_winkler('ATCG', 'TAGC'), 12)
0.833333333333
```

```
>>> round(sim_jaro_winkler('cat', 'hat', mode='jaro'), 12)
0.777777777778
>>> round(sim_jaro_winkler('Niall', 'Neil', mode='jaro'), 12)
0.783333333333
>>> round(sim_jaro_winkler('aluminum', 'Catalan', mode='jaro'), 12)
0.60119047619
>>> round(sim_jaro_winkler('ATCG', 'TAGC', mode='jaro'), 12)
0.833333333333
```

abydos.distance.**sim_lcsseq**(*src*, *tar*)

Return the longest common subsequence similarity of two strings.

Longest common subsequence similarity ($sim_{LCSseq}$).

This employs the LCSseq function to derive a similarity metric: $sim_{LCSseq}(s,t) = frac|LCSseq(s,t)|max(|s|,|t|)$

**Parameters**

- **src** ($str$) – source string for comparison
- **tar** ($str$) – target string for comparison

**Returns** LCSseq similarity

**Return type** float

```
>>> sim_lcsseq('cat', 'hat')
0.6666666666666666
>>> sim_lcsseq('Niall', 'Neil')
0.6
>>> sim_lcsseq('aluminum', 'Catalan')
0.375
>>> sim_lcsseq('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_lcsstr**(*src*, *tar*)

Return the longest common substring similarity of two strings.

Longest common substring similarity ($sim_{LCSstr}$).

This employs the LCS function to derive a similarity metric: $sim_{LCSstr}(s,t) = frac|LCSstr(s,t)|max(|s|,|t|)$

**Parameters**

- **src** ($str$) – source string for comparison
- **tar** ($str$) – target string for comparison

**Returns** LCSstr similarity

**Return type** float

```
>>> sim_lcsstr('cat', 'hat')
0.6666666666666666
>>> sim_lcsstr('Niall', 'Neil')
0.2
>>> sim_lcsstr('aluminum', 'Catalan')
0.25
>>> sim_lcsstr('ATCG', 'TAGC')
0.25
```

abydos.distance.**sim_length**(*src*, *tar*)

Return the length similarity of two strings.

Length similarity is the ratio of the length of the shorter string to the longer.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** length similarity

**Return type** float

```
>>> sim_length('cat', 'hat')
1.0
>>> sim_length('Niall', 'Neil')
0.8
>>> sim_length('aluminum', 'Catalan')
0.875
>>> sim_length('ATCG', 'TAGC')
1.0
```

abydos.distance.**sim_levenshtein**(*src*, *tar*, *mode='lev'*, *cost=(1, 1, 1, 1)*)

Return the Levenshtein similarity of two strings.

Normalized Levenshtein similarity is the complement of normalized Levenshtein distance: $sim_{Levenshtein} = 1 - dist_{Levenshtein}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **mode** (*str*) – specifies a mode for computing the Levenshtein distance:
  - 'lev' (default) computes the ordinary Levenshtein distance, in which edits may include inserts, deletes, and substitutions
  - 'osa' computes the Optimal String Alignment distance, in which edits may include inserts, deletes, substitutions, and transpositions but substrings may only be edited once
  - 'dam' computes the Damerau-Levenshtein distance, in which edits may include inserts, deletes, substitutions, and transpositions and substrings may undergo repeated edits
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and transpositions, respectively (by default: (1, 1, 1, 1))

**Returns** normalized Levenshtein similarity

**Return type** float

```
>>> round(sim_levenshtein('cat', 'hat'), 12)
0.666666666667
>>> round(sim_levenshtein('Niall', 'Neil'), 12)
0.4
>>> sim_levenshtein('aluminum', 'Catalan')
0.125
>>> sim_levenshtein('ATCG', 'TAGC')
0.25
```

abydos.distance.**sim_manhattan**(*src*, *tar*, *qval=2*, *alphabet=None*)

Return the normalized Manhattan similarity of two strings.

Manhattan similarity is the complement of Manhattan distance: $sim_{Manhattan} = 1 - dist_{Manhattan}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Manhattan similarity

**Return type** float

```
>>> sim_manhattan('cat', 'hat')
0.5
>>> round(sim_manhattan('Niall', 'Neil'), 12)
0.363636363636
>>> round(sim_manhattan('Colin', 'Cuilen'), 12)
0.307692307692
>>> sim_manhattan('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_matrix**(*src*, *tar*, *mat=None*, *mismatch_cost=0*, *match_cost=1*, *symmetric=True*, *alphabet=None*)

Return the matrix similarity of two strings.

With the default parameters, this is identical to sim_ident. It is possible for sim_matrix to return values outside of the range $[0, 1]$, if values outside that range are present in mat, mismatch_cost, or match_cost.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **mat** (*dict*) – a dict mapping tuples to costs; the tuples are (src, tar) pairs of symbols from the alphabet parameter
- **mismatch_cost** (*float*) – the value returned if (src, tar) is absent from mat when src does not equal tar
- **match_cost** (*float*) – the value returned if (src, tar) is absent from mat when src equals tar
- **symmetric** (*bool*) – True if the cost of src not matching tar is identical to the cost of tar not matching src; in this case, the values in mat need only contain (src, tar) or (tar, src), not both
- **alphabet** (*str*) – a collection of tokens from which src and tar are drawn; if this is defined a ValueError is raised if either tar or src is not found in alphabet

**Returns** matrix similarity

**Return type** float

```
>>> sim_matrix('cat', 'hat')
0
>>> sim_matrix('hat', 'hat')
1
```

abydos.distance.**sim_minkowski**(*src*, *tar*, *qval=2*, *pval=1*, *alphabet=None*)

Return normalized Minkowski similarity of two strings.

Minkowski similarity is the complement of Minkowski distance: $sim_{Minkowski} = 1 - dist_{Minkowski}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **or float pval** (*int*) – the $p$-value of the $L^p$-space.
- **or int alphabet** (*collection*) – the values or size of the alphabet

**Returns** the normalized Minkowski similarity

**Return type** float

```
>>> sim_minkowski('cat', 'hat')
0.5
>>> round(sim_minkowski('Niall', 'Neil'), 12)
0.363636363636
>>> round(sim_minkowski('Colin', 'Cuilen'), 12)
0.307692307692
>>> sim_minkowski('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_mlipns**(*src*, *tar*, *threshold=0.25*, *max_mismatches=2*)

Return the MLIPNS similarity of two strings.

Modified Language-Independent Product Name Search (MLIPNS) is described in [SA10]. This function returns only 1.0 (similar) or 0.0 (not similar). LIPNS similarity is identical to normalized Hamming similarity.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **threshold** (*float*) – a number [0, 1] indicating the maximum similarity score, below which the strings are considered 'similar' (0.25 by default)
- **max_mismatches** (*int*) – a number indicating the allowable number of mismatches to remove before declaring two strings not similar (2 by default)

**Returns** MLIPNS similarity

**Return type** float

```
>>> sim_mlipns('cat', 'hat')
1.0
>>> sim_mlipns('Niall', 'Neil')
0.0
>>> sim_mlipns('aluminum', 'Catalan')
0.0
>>> sim_mlipns('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_monge_elkan**(*src*, *tar*, *sim_func=<function sim_levenshtein>*, *symmetric=False*)

Return the Monge-Elkan similarity of two strings.

Monge-Elkan is defined in [ME96].

Note: Monge-Elkan is NOT a symmetric similarity algoritm. Thus, the similarity of src to tar is not necessarily equal to the similarity of tar to src. If the sym argument is True, a symmetric value is calculated, at the cost of doubling the computation time (since the $sim_{Monge-Elkan}(src, tar)$ and $sim_{Monge-Elkan}(tar, src)$ are both calculated and then averaged).

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **sim_func** (*function*) – the internal similarity metric to employ
- **symmetric** (*bool*) – return a symmetric similarity measure

**Returns** Monge-Elkan similarity

**Return type** float

```
>>> sim_monge_elkan('cat', 'hat')
0.75
>>> round(sim_monge_elkan('Niall', 'Neil'), 12)
0.666666666667
>>> round(sim_monge_elkan('aluminum', 'Catalan'), 12)
0.388888888889
>>> sim_monge_elkan('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_mra**(*src*, *tar*)

Return the normalized MRA similarity of two strings.

This is the MRA normalized to $[0, 1]$, given that MRA itself is constrained to the range $[0, 6]$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** normalized MRA similarity

**Return type** float

```
>>> sim_mra('cat', 'hat')
0.8333333333333334
>>> sim_mra('Niall', 'Neil')
1.0
>>> sim_mra('aluminum', 'Catalan')
0.0
>>> sim_mra('ATCG', 'TAGC')
0.8333333333333334
```

abydos.distance.**sim_overlap**(*src*, *tar*, *qval=2*)

Return the overlap coefficient of two strings.

For two sets X and Y, the overlap coefficient [Szy34][Sim49], also called the Szymkiewicz-Simpson coefficient, is $sim_{overlap}(X, Y) =$
$frac|XcapY|min(|X|, |Y|)$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison

- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** overlap similarity

**Return type** float

```
>>> sim_overlap('cat', 'hat')
0.5
>>> sim_overlap('Niall', 'Neil')
0.4
>>> sim_overlap('aluminum', 'Catalan')
0.125
>>> sim_overlap('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_prefix**(*src*, *tar*)

Return the prefix similarity of two strings.

Prefix similarity is the ratio of the length of the shorter term that exactly matches the longer term to the length of the shorter term, beginning at the start of both terms.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** prefix similarity

**Return type** float

```
>>> sim_prefix('cat', 'hat')
0.0
>>> sim_prefix('Niall', 'Neil')
0.25
>>> sim_prefix('aluminum', 'Catalan')
0.0
>>> sim_prefix('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_ratcliff_obershelp**(*src*, *tar*)

Return the Ratcliff-Obershelp similarity of two strings.

This follows the Ratcliff-Obershelp algorithm [RM88] to derive a similarity measure:

1. Find the length of the longest common substring in src & tar.
2. Recurse on the strings to the left & right of each this substring in src & tar. The base case is a 0 length common substring, in which case, return 0. Otherwise, return the sum of the current longest common substring and the left & right recursed sums.
3. Multiply this length by 2 and divide by the sum of the lengths of src & tar.

Cf. http://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** Ratcliff-Obershelp similarity

**Return type** float

```
>>> round(sim_ratcliff_obershelp('cat', 'hat'), 12)
0.666666666667
>>> round(sim_ratcliff_obershelp('Niall', 'Neil'), 12)
0.666666666667
>>> round(sim_ratcliff_obershelp('aluminum', 'Catalan'), 12)
0.4
>>> sim_ratcliff_obershelp('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_sift4**(*src*, *tar*, *max_offset=5*, *max_distance=0*)

Return the normalized "common" Sift4 similarity of two terms.

Normalized Sift4 similarity is the complement of normalized Sift4 distance: $sim_{Sift4} = 1 - dist_{Sift4}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **max_offset** – the number of characters to search for matching letters
- **max_distance** – the distance at which to stop and exit

**Returns** the normalized Sift4 similarity

**Return type** float

```
>>> round(sim_sift4('cat', 'hat'), 12)
0.666666666667
>>> sim_sift4('Niall', 'Neil')
0.6
>>> sim_sift4('Colin', 'Cuilen')
0.5
>>> sim_sift4('ATCG', 'TAGC')
0.5
```

abydos.distance.**sim_strcmp95**(*src*, *tar*, *long_strings=False*)

Return the strcmp95 similarity of two strings.

This is a Python translation of the C code for strcmp95: http://web.archive.org/web/20110629121242/http://www.census.gov/geo/msb/stand/strcmp.c [WMJL94]. The above file is a US Government publication and, accordingly, in the public domain.

This is based on the Jaro-Winkler distance, but also attempts to correct for some common typos and frequently confused characters. It is also limited to uppercase ASCII characters, so it is appropriate to American names, but not much else.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **long_strings** (*bool*) – set to True to "Increase the probability of a match when the number of matched characters is large. This option allows for a little more tolerance when the strings are large. It is not an appropriate test when comparing fixed length fields such as phone and social security numbers."

**Returns** strcmp95 similarity

**Return type** float

```
>>> sim_strcmp95('cat', 'hat')
0.7777777777777777
>>> sim_strcmp95('Niall', 'Neil')
0.8454999999999999
>>> sim_strcmp95('aluminum', 'Catalan')
0.6547619047619048
>>> sim_strcmp95('ATCG', 'TAGC')
0.8333333333333334
```

abydos.distance.**sim_suffix**(*src*, *tar*)

Return the suffix similarity of two strings.

Suffix similarity is the ratio of the length of the shorter term that exactly matches the longer term to the length of the shorter term, beginning at the end of both terms.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison

**Returns** suffix similarity

**Return type** float

```
>>> sim_suffix('cat', 'hat')
0.6666666666666666
>>> sim_suffix('Niall', 'Neil')
0.25
>>> sim_suffix('aluminum', 'Catalan')
0.0
>>> sim_suffix('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_tanimoto**(*src*, *tar*, *qval=2*)

Return the Tanimoto similarity of two strings.

For two sets X and Y, the Tanimoto similarity coefficient [Tan58] is $sim_{Tanimoto}(X, Y) = frac|XcapY||XcupY|$.

This is identical to the Jaccard similarity coefficient [Jac01] and the Tversky index [Tve77] for $alpha =$
$beta = 1$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Tanimoto similarity

**Return type** float

```
>>> sim_tanimoto('cat', 'hat')
0.3333333333333333
>>> sim_tanimoto('Niall', 'Neil')
0.2222222222222222
>>> sim_tanimoto('aluminum', 'Catalan')
0.0625
```

(continues on next page)

(continued from previous page)

```
>>> sim_tanimoto('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_tversky**(*src*, *tar*, *qval=2*, *alpha=1*, *beta=1*, *bias=None*)

Return the Tversky index of two strings.

The Tversky index [Tve77] is defined as: For two sets X and Y: $sim_{Tversky}(X, Y) = frac|XcapY||XcapY| + alpha|X - Y| + beta|Y - X|$.

$alpha =$
$beta = 1$ is equivalent to the Jaccard & Tanimoto similarity coefficients.

$alpha =$
$beta = 0.5$ is equivalent to the Sørensen-Dice similarity coefficient [Dic45][Sorensen48].

Unequal $\alpha$ and $\beta$ will tend to emphasize one or the other set's contributions:

- $alpha >$
  $beta$ emphasizes the contributions of X over Y
- $alpha <$
  $beta$ emphasizes the contributions of Y over X)

Parameter values' relation to 1 emphasizes different types of contributions:

- $alphaand$
  $beta > 1$ emphsize unique contributions over the intersection
- $alphaand$
  $beta < 1$ emphsize the intersection over unique contributions

The symmetric variant is defined in [JBG13]. This is activated by specifying a bias parameter.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version
- **alpha** (*float*) – Tversky index parameter as described above
- **beta** (*float*) – Tversky index parameter as described above
- **bias** (*float*) – The symmetric Tversky index bias parameter

**Returns** Tversky similarity

**Return type** float

```
>>> sim_tversky('cat', 'hat')
0.3333333333333333
>>> sim_tversky('Niall', 'Neil')
0.2222222222222222
>>> sim_tversky('aluminum', 'Catalan')
```

(continues on next page)

(continued from previous page)

```
0.0625
>>> sim_tversky('ATCG', 'TAGC')
0.0
```

abydos.distance.**sim_typo**(*src*, *tar*, *metric='euclidean'*, *cost=(1, 1, 0.5, 0.5)*)

Return the normalized typo similarity between two strings.

Normalized typo similarity is the complement of normalized typo distance: $sim_{typo} = 1 - dist_{typo}$.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **metric** (*str*) – supported values include: 'euclidean', 'manhattan', 'log-euclidean', and 'log-manhattan'
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and shift, respectively (by default: (1, 1, 0.5, 0.5)) The substitution & shift costs should be significantly less than the cost of an insertion & deletion unless a log metric is used.

**Returns** normalized typo similarity

**Return type** float

```
>>> round(sim_typo('cat', 'hat'), 12)
0.472953716914
>>> round(sim_typo('Niall', 'Neil'), 12)
0.434971857071
>>> round(sim_typo('Colin', 'Cuilen'), 12)
0.430964390437
>>> sim_typo('ATCG', 'TAGC')
0.375
```

abydos.distance.**smith_waterman**(*src*, *tar*, *gap_cost=1*, *sim_func=<function sim_ident>*)

Return the Smith-Waterman score of two strings.

The Smith-Waterman score [SW81] is a standard edit distance measure, differing from Needleman-Wunsch in that it focuses on local alignment and disallows negative scores.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **gap_cost** (*float*) – the cost of an alignment gap (1 by default)
- **sim_func** (*function*) – a function that returns the similarity of two characters (identity similarity by default)

**Returns** Smith-Waterman score

**Return type** float

```
>>> smith_waterman('cat', 'hat')
2.0
>>> smith_waterman('Niall', 'Neil')
1.0
>>> smith_waterman('aluminum', 'Catalan')
```

(continues on next page)

(continued from previous page)

```
0.0
>>> smith_waterman('ATCG', 'TAGC')
1.0
```

abydos.distance.**synoname**(*src*, *tar*, *word_approx_min=0.3*, *char_approx_min=0.73*, *tests=4095*, *ret_name=False*)

Return the Synoname similarity type of two words.

Cf. [JPGTrust91][Gro91]

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **ret_name** (*bool*) – return the name of the match type rather than the int value
- **word_approx_min** (*float*) – the minimum word approximation value to signal a 'word_approx' match
- **char_approx_min** (*float*) – the minimum character approximation value to signal a 'char_approx' match
- **or Iterable tests** (*int*) – either an integer indicating tests to perform or a list of test names to perform (defaults to performing all tests)
- **ret_name** – if True, returns the match name rather than its integer equivalent

**Returns** Synoname value

**Return type** int (or str if ret_name is True)

```
>>> synoname(('Breghel', 'Pieter', ''), ('Brueghel', 'Pieter', ''))
2
>>> synoname(('Breghel', 'Pieter', ''), ('Brueghel', 'Pieter', ''),
... ret_name=True)
'omission'
>>> synoname(('Dore', 'Gustave', ''),
... ('Dore', 'Paul Gustave Louis Christophe', ''),
... ret_name=True)
'inclusion'
>>> synoname(('Pereira', 'I. R.', ''), ('Pereira', 'I. Smith', ''),
... ret_name=True)
'word_approx'
```

abydos.distance.**tanimoto**(*src*, *tar*, *qval=2*)

Return the Tanimoto distance between two strings.

Tanimoto distance is $-log_2 sim_{Tanimoto}$.

**Parameters**

- **src** (*str*) – source string (or QGrams/Counter objects) for comparison
- **tar** (*str*) – target string (or QGrams/Counter objects) for comparison
- **qval** (*int*) – the length of each q-gram; 0 for non-q-gram version

**Returns** Tanimoto distance

**Return type** float

```
>>> tanimoto('cat', 'hat')
-1.5849625007211563
>>> tanimoto('Niall', 'Neil')
-2.1699250014423126
>>> tanimoto('aluminum', 'Catalan')
-4.0
>>> tanimoto('ATCG', 'TAGC')
-inf
```

abydos.distance.**typo**(*src*, *tar*, *metric='euclidean'*, *cost=(1, 1, 0.5, 0.5)*, *layout='QWERTY'*)

Return the typo distance between two strings.

This is inspired by Typo-Distance [Son11], and a fair bit of this was copied from that module. Compared to the original, this supports different metrics for substitution.

**Parameters**

- **src** (*str*) – source string for comparison
- **tar** (*str*) – target string for comparison
- **metric** (*str*) – supported values include: 'euclidean', 'manhattan', 'log-euclidean', and 'log-manhattan'
- **cost** (*tuple*) – a 4-tuple representing the cost of the four possible edits: inserts, deletes, substitutions, and shift, respectively (by default: (1, 1, 0.5, 0.5)) The substitution & shift costs should be significantly less than the cost of an insertion & deletion unless a log metric is used.
- **layout** (*str*) – name of the keyboard layout to use (Currently supported: QWERTY, Dvorak, AZERTY, QWERTZ)

**Returns** typo distance

**Return type** float

```
>>> typo('cat', 'hat')
1.5811388
>>> typo('Niall', 'Neil')
2.8251407
>>> typo('Colin', 'Cuilen')
3.4142137
>>> typo('ATCG', 'TAGC')
2.5
```

```
>>> typo('cat', 'hat', metric='manhattan')
2.0
>>> typo('Niall', 'Neil', metric='manhattan')
3.0
>>> typo('Colin', 'Cuilen', metric='manhattan')
3.5
>>> typo('ATCG', 'TAGC', metric='manhattan')
2.5
```

```
>>> typo('cat', 'hat', metric='log-manhattan')
0.804719
>>> typo('Niall', 'Neil', metric='log-manhattan')
2.2424533
>>> typo('Colin', 'Cuilen', metric='log-manhattan')
```

(continues on next page)

(continued from previous page)

```
2.2424533
>>> typo('ATCG', 'TAGC', metric='log-manhattan')
2.3465736
```

#### 3.1.1.5 abydos.fingerprint module

abydos.fingerprint.

**The fingerprint module implements string fingerprints such as:**

- string fingerprint
- q-gram fingerprint
- phonetic fingerprint
- Pollock & Zomora's skeleton key
- Pollock & Zomora's omission key
- Cisłak & Grabowski's occurrence fingerprint
- Cisłak & Grabowski's occurrence halved fingerprint
- Cisłak & Grabowski's count fingerprint
- Cisłak & Grabowski's position fingerprint
- Synoname Toolcode

abydos.fingerprint.**count_fingerprint**(*word*, *n_bits=16*, *most_common=('e', 't', 'a', 'o', 'i', 'n', 's', 'h', 'r', 'd', 'l', 'c', 'u', 'm', 'w', 'f')*)

Return the count fingerprint.

Based on the count fingerprint from [CislakG17].

**Parameters**

- **word** (*str*) – the word to fingerprint
- **n_bits** (*int*) – number of bits in the fingerprint returned
- **most_common** (*list*) – the most common tokens in the target language, ordered by frequency

**Returns** the count fingerprint

**Return type** int

```
>>> bin(count_fingerprint('hat'))
'0b1010000000001'
>>> bin(count_fingerprint('niall'))
'0b10001010000'
>>> bin(count_fingerprint('colin'))
'0b101010000'
>>> bin(count_fingerprint('atcg'))
'0b1010000000000'
>>> bin(count_fingerprint('entreatment'))
'0b1111010000100000'
```

abydos.fingerprint.**occurrence_fingerprint**(*word*, *n_bits=16*, *most_common=('e', 't', 'a', 'o', 'i', 'n', 's', 'h', 'r', 'd', 'l', 'c', 'u', 'm', 'w', 'f')*)

Return the occurrence fingerprint.

Based on the occurrence fingerprint from [CislakG17].

**Parameters**

- **word** (*str*) – the word to fingerprint
- **n_bits** (*int*) – number of bits in the fingerprint returned
- **most_common** (*list*) – the most common tokens in the target language, ordered by frequency

**Returns** the occurrence fingerprint

**Return type** int

```
>>> bin(occurrence_fingerprint('hat'))
'0b1100000100000000'
>>> bin(occurrence_fingerprint('niall'))
'0b10110000100000'
>>> bin(occurrence_fingerprint('colin'))
'0b1110000110000'
>>> bin(occurrence_fingerprint('atcg'))
'0b1100000000010000'
>>> bin(occurrence_fingerprint('entreatment'))
'0b1110010010000100'
```

abydos.fingerprint.**occurrence_halved_fingerprint**(*word*, *n_bits=16*, *most_common=('e', 't', 'a', 'o', 'i', 'n', 's', 'h', 'r', 'd', 'l', 'c', 'u', 'm', 'w', 'f')*)

Return the occurrence halved fingerprint.

Based on the occurrence halved fingerprint from [CislakG17].

**Parameters**

- **word** (*str*) – the word to fingerprint
- **n_bits** (*int*) – number of bits in the fingerprint returned
- **most_common** (*list*) – the most common tokens in the target language, ordered by frequency

**Returns** the occurrence halved fingerprint

**Return type** int

```
>>> bin(occurrence_halved_fingerprint('hat'))
'0b1010000000010'
>>> bin(occurrence_halved_fingerprint('niall'))
'0b10010100000'
>>> bin(occurrence_halved_fingerprint('colin'))
'0b1001010000'
>>> bin(occurrence_halved_fingerprint('atcg'))
'0b101000000000000'
>>> bin(occurrence_halved_fingerprint('entreatment'))
'0b1111010000110000'
```

abydos.fingerprint.**omission_key**(*word*)

Return the omission key.

The omission key of a word is defined in [PZ84].

**Parameters** **word** (*str*) – the word to transform into its omission key

**Returns** the omission key

**Return type** str

```
>>> omission_key('The quick brown fox jumped over the lazy dog.')
'JKQXZVWYBFMGPDHCLNTREUIOA'
>>> omission_key('Christopher')
'PHCTSRIOE'
>>> omission_key('Niall')
'LNIA'
```

abydos.fingerprint.**phonetic_fingerprint**(*phrase*, *phonetic_algorithm=<function double_metaphone>*, *joiner=' '*, **args*)

Return the phonetic fingerprint of a phrase.

A phonetic fingerprint is identical to a standard string fingerprint, as implemented in abydos.clustering.fingerprint(), but performs the fingerprinting function after converting the string to its phonetic form, as determined by some phonetic algorithm. This fingerprint is described at [Ope12].

**Parameters**

- **phrase** (*str*) – the string from which to calculate the phonetic fingerprint
- **phonetic_algorithm** (*function*) – a phonetic algorithm that takes a string and returns a string (presumably a phonetic representation of the original string) By default, this function uses abydos.phonetic.double_metaphone()
- **joiner** (*str*) – the string that will be placed between each word
- **args** – additional arguments to pass to the phonetic algorithm, along with the phrase itself

**Returns** the phonetic fingerprint of the phrase

**Return type** str

```
>>> phonetic_fingerprint('The quick brown fox jumped over the lazy dog.')
'0 afr fks jmpt kk ls prn tk'
>>> from abydos.phonetic import soundex
>>> phonetic_fingerprint('The quick brown fox jumped over the lazy dog.',
... phonetic_algorithm=soundex)
'b650 d200 f200 j513 l200 o160 q200 t000'
```

abydos.fingerprint.**position_fingerprint**(*word*, *n_bits=16*, *most_common=('e', 't', 'a', 'o', 'i', 'n', 's', 'h', 'r', 'd', 'l', 'c', 'u', 'm', 'w', 'f')*, *bits_per_letter=3*)

Return the position fingerprint.

Based on the position fingerprint from [CislakG17].

**Parameters**

- **word** (*str*) – the word to fingerprint
- **n_bits** (*int*) – number of bits in the fingerprint returned
- **most_common** (*list*) – the most common tokens in the target language, ordered by frequency

- **bits_per_letter** (*int*) – the bits to assign for letter position

**Returns** the position fingerprint

**Return type** int

```
>>> bin(position_fingerprint('hat'))
'0b1110100011111111'
>>> bin(position_fingerprint('niall'))
'0b1111110101110010'
>>> bin(position_fingerprint('colin'))
'0b1111111110010111'
>>> bin(position_fingerprint('atcg'))
'0b1110010001111111'
>>> bin(position_fingerprint('entreatment'))
'0b101011111111'
```

abydos.fingerprint.**qgram_fingerprint**(*phrase*, *qval=2*, *start_stop=''*, *joiner=''*)

Return Q-Gram fingerprint.

A q-gram fingerprint is a string consisting of all of the unique q-grams in a string, alphabetized & concatenated. This fingerprint is described at [Ope12].

**Parameters**

- **phrase** (*str*) – the string from which to calculate the q-gram fingerprint
- **qval** (*int*) – the length of each q-gram (by default 2)
- **start_stop** (*str*) – the start & stop symbol(s) to concatenate on either end of the phrase, as defined in abydos.util.qgram()
- **joiner** (*str*) – the string that will be placed between each word

**Returns** the q-gram fingerprint of the phrase

**Return type** str

```
>>> qgram_fingerprint('The quick brown fox jumped over the lazy dog.')
'azbrckdoedeleqerfoheicjukblampnfogovowoxpequrortthuiumvewnxjydzy'
>>> qgram_fingerprint('Christopher')
'cherhehrisopphristto'
>>> qgram_fingerprint('Niall')
'aliallni'
```

abydos.fingerprint.**skeleton_key**(*word*)

Return the skeleton key.

The skeleton key of a word is defined in [PZ84].

**Parameters** **word** (*str*) – the word to transform into its skeleton key

**Returns** the skeleton key

**Return type** str

```
>>> skeleton_key('The quick brown fox jumped over the lazy dog.')
'THQCKBRWNFXJMPDVLZYGEUIOA'
>>> skeleton_key('Christopher')
'CHRSTPIOE'
>>> skeleton_key('Niall')
'NLIA'
```

abydos.fingerprint.**str_fingerprint**(*phrase*, *joiner=' '*)
Return string fingerprint.

The fingerprint of a string is a string consisting of all of the unique words in a string, alphabetized & concatenated with intervening joiners. This fingerprint is described at [Ope12].

**Parameters**

- **phrase** (*str*) – the string from which to calculate the fingerprint
- **joiner** (*str*) – the string that will be placed between each word

**Returns** the fingerprint of the phrase

**Return type** str

```
>>> str_fingerprint('The quick brown fox jumped over the lazy dog.')
'brown dog fox jumped lazy over quick the'
```

abydos.fingerprint.**synoname_toolcode**(*lname*, *fname=''*, *qual=''*, *normalize=0*)
Build the Synoname toolcode.

Cf. [JPGTrust91][Gro91].

**Parameters**

- **lname** (*str*) – last name
- **fname** (*str*) – first name (can be blank)
- **qual** (*str*) – qualifier
- **normalize** (*int*) – normalization mode (0, 1, or 2)

**Returns** the transformed last and first names and the synoname toolcode

**Return type** tuple

```
>>> synoname_toolcode('hat')
('hat', '', '0000000003$$h')
>>> synoname_toolcode('niall')
('niall', '', '0000000005$$n')
>>> synoname_toolcode('colin')
('colin', '', '0000000005$$c')
>>> synoname_toolcode('atcg')
('atcg', '', '0000000004$$a')
>>> synoname_toolcode('entreatment')
('entreatment', '', '0000000011$$e')
```

```
>>> synoname_toolcode('Ste.-Marie', 'Count John II', normalize=2)
('ste.-marie ii', 'count john', '0200491310$015b049a127c$smcji')
>>> synoname_toolcode('Michelangelo IV', '', 'Workshop of')
('michelangelo iv', '', '3000550015$055b$mi')
```

#### 3.1.1.6 abydos.ngram module

abydos.ngram.

The NGram class is a container for an n-gram corpus

**class** abydos.ngram.**NGramCorpus**(*corpus=None*)
Bases: object

The NGramCorpus class.

Internally, this is a set of recursively embedded dicts, with n layers for a corpus of n-grams. E.g. for a trigram corpus, this will be a dict of dicts of dicts. More precisely, collections.Counter is used in place of dict, making multiset operations valid and allowing unattested n-grams to be queried.

The key at each level is a word. The value at the most deeply embedded level is a numeric value representing the frequency of the trigram. E.g. the trigram frequency of 'colorless green ideas' would be the value stored in self.ngcorpus['colorless']['green']['ideas'][None].

**corpus_importer**(*corpus*, *n_val=1*, *bos='_START_'*, *eos='_END_'*)

Fill in self.ngcorpus from a Corpus argument.

**Parameters**

- **corpus** (Corpus) – The Corpus from which to initialize the n-gram corpus
- **n_val** (*int*) – maximum n value for n-grams
- **bos** (*str*) – string to insert as an indicator of beginning of sentence
- **eos** (*str*) – string to insert as an indicator of end of sentence

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> ngcorp = NGramCorpus()
>>> ngcorp.corpus_importer(Corpus(tqbf))
```

**get_count**(*ngram*, *corpus=None*)

Get the count of an n-gram in the corpus.

**Parameters**

- **ngram** (*list, tuple, or string*) – The n-gram to retrieve the count of from the n-gram corpus
- **corpus** (Corpus) – The corpus

**Returns** The n-gram count

**Return type** int

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> ngcorp = NGramCorpus(Corpus(tqbf))
>>> NGramCorpus(Corpus(tqbf)).get_count('the')
2
>>> NGramCorpus(Corpus(tqbf)).get_count('fox')
1
```

**gng_importer**(*corpus_file*)

Fill in self.ngcorpus from a Google NGram corpus file.

**Parameters** **corpus_file** (*file*) – The Google NGram file from which to initialize the n-gram corpus

**tf**(*term*)

Return term frequency.

**Parameters** **term** (*str*) – The term for which to calculate tf

**Returns** The term frequency (tf)

**Return type** float

```
>>> tqbf = 'The quick brown fox jumped over the lazy dog.\n'
>>> tqbf += 'And then it slept.\n And the dog ran off.'
>>> ngcorp = NGramCorpus(Corpus(tqbf))
>>> NGramCorpus(Corpus(tqbf)).tf('the')
1.3010299956639813
>>> NGramCorpus(Corpus(tqbf)).tf('fox')
1.0
```

#### 3.1.1.7 abydos.phones module

abydos.phones.

The phones module implements ...

abydos.phones.**cmp_features**(*feat1*, *feat2*)

Compare features.

This returns a number in the range [0, 1] representing a comparison of two feature bundles.

If one of the bundles is negative, -1 is returned (for unknown values)

If the bundles are identical, 1 is returned.

If they are inverses of one another, 0 is returned.

Otherwise, a float representing their similarity is returned.

**Parameters**

- **feat1** (*int*) – a feature bundle
- **feat2** (*int*) – a feature bundle

**Returns** a comparison of the feature bundles

**Return type** float

```
>>> cmp_features(ipa_to_features('l')[0], ipa_to_features('l')[0])
1.0
>>> cmp_features(ipa_to_features('l')[0], ipa_to_features('n')[0])
0.8709677419354839
>>> cmp_features(ipa_to_features('l')[0], ipa_to_features('z')[0])
0.8709677419354839
>>> cmp_features(ipa_to_features('l')[0], ipa_to_features('i')[0])
0.564516129032258
```

abydos.phones.**get_feature**(*vector*, *feature*)

Get a feature vector.

**This returns a list of ints, equal in length to the vector input,** representing presence/absence/neutrality with respect to a particular phonetic feature.

**Parameters**

- **vector** (*list*) – a tuple or list of ints representing the phonetic features of a phone or series of phones (such as is returned by the ipa_to_features function)
- **feature** (*str*) – a feature name from the set: 'consonantal', 'sonorant', 'syllabic', 'labial', 'round', 'coronal', 'anterior', 'distributed', 'dorsal', 'high', 'low', 'back', 'tense', 'pharyngeal', 'ATR', 'voice', 'spread_glottis', 'constricted_glottis', 'continuant', 'strident', 'lateral', 'delayed_release', 'nasal'

**Returns** a list indicating presence/absence/neutrality with respect to the feature

**Return type** [int]

```
>>> tails = ipa_to_features('telz')
>>> get_feature(tails, 'consonantal')
[1, -1, 1, 1]
>>> get_feature(tails, 'sonorant')
[-1, 1, 1, -1]
>>> get_feature(tails, 'nasal')
[-1, -1, -1, -1]
>>> get_feature(tails, 'coronal')
[1, -1, 1, 1]
```

abydos.phones.**ipa_to_features**(*ipa*)

Convert IPA to features.

This translates an IPA string of one or more phones to a list of ints representing the features of the string.

**Parameters** **ipa** (*str*) – the IPA representation of a phone or series of phones

**Returns** a representation of the features of the input string

**Return type** [int]

```
>>> ipa_to_features('mut')
[2709662981243185770, 1825831513894594986, 2783230754502126250]
>>> ipa_to_features('fon')
[2781702983095331242, 1825831531074464170, 2711173160463936106]
>>> ipa_to_features('telz')
[2783230754502126250, 1826957430176000426, 2693158761954453926,
2783230754501863834]
```

#### 3.1.1.8 abydos.phonetic module

abydos.phonetic.

The phonetic module implements phonetic algorithms including:

- Robert C. Russell's Index
- American Soundex
- Refined Soundex
- Daitch-Mokotoff Soundex
- Kölner Phonetik
- NYSIIS
- Match Rating Algorithm
- Metaphone
- Double Metaphone
- Caverphone
- Alpha Search Inquiry System
- Fuzzy Soundex

- Phonex
- Phonem
- Phonix
- SfinxBis
- phonet
- Standardized Phonetic Frequency Code
- Statistics Canada
- Lein
- Roger Root
- Oxford Name Compression Algorithm (ONCA)
- Eudex phonetic hash
- Haase Phonetik
- Reth-Schek Phonetik
- FONEM
- Parmar-Kumbharana
- Davidson's Consonant Code
- SoundD
- PSHP Soundex/Viewex Coding
- an early version of Henry Code
- Norphone
- Dolby Code
- Phonetic Spanish
- Spanish Metaphone
- MetaSoundex
- SoundexBR
- NRL English-to-phoneme
- Beider-Morse Phonetic Matching

abydos.phonetic.**alpha_sis**(*word*, *max_length=14*)

Return the IBM Alpha Search Inquiry System code for a word.

The Alpha Search Inquiry System code is defined in [IBMCorporation73]. This implementation is based on the description in [MKTM77].

A collection is necessary since there can be multiple values for a single word. But the collection must be ordered since the first value is the primary coding.

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 14)

**Returns** the Alpha SIS value

**Return type** tuple

```
>>> alpha_sis('Christopher')
('06401840000000', '07040184000000', '04018400000000')
>>> alpha_sis('Niall')
('02500000000000',)
>>> alpha_sis('Smith')
('03100000000000',)
>>> alpha_sis('Schmidt')
('06310000000000',)
```

abydos.phonetic.**bmpm**(*word*, *language_arg=0*, *name_mode='gen'*, *match_mode='approx'*, *concat=False*, *filter_langs=False*)

Return the Beider-Morse Phonetic Matching algorithm code for a word.

The Beider-Morse Phonetic Matching algorithm is described in [BM08]. The reference implementation is licensed under GPLv3.

**Parameters**

- **word** (*str*) – the word to transform
- **language_arg** (*str*) – the language of the term; supported values include:
  - 'any'
  - 'arabic'
  - 'cyrillic'
  - 'czech'
  - 'dutch'
  - 'english'
  - 'french'
  - 'german'
  - 'greek'
  - 'greeklatin'
  - 'hebrew'
  - 'hungarian'
  - 'italian'
  - 'latvian'
  - 'polish'
  - 'portuguese'
  - 'romanian'
  - 'russian'
  - 'spanish'
  - 'turkish'
- **name_mode** (*str*) – the name mode of the algorithm:
  - 'gen' – general (default)

- 'ash' – Ashkenazi
- 'sep' – Sephardic

- **match_mode** (*str*) – matching mode: 'approx' or 'exact'
- **concat** (*bool*) – concatenation mode
- **filter_langs** (*bool*) – filter out incompatible languages

**Returns** the BMPM value(s)

**Return type** tuple

```
>>> bmpm('Christopher')
'xrQstopir xrQstYpir xristopir xristYpir xrQstofir xrQstYfir xristofir
xristYfir xristopi xritopir xritopi xristofi xritofir xritofi tzristopir
tzristofir zristopir zristopi zritopir zritopi zristofir zristofi zritofir
zritofi'
>>> bmpm('Niall')
'nial niol'
>>> bmpm('Smith')
'zmit'
>>> bmpm('Schmidt')
'zmit stzmit'
```

```
>>> bmpm('Christopher', language_arg='German')
'xrQstopir xrQstYpir xristopir xristYpir xrQstofir xrQstYfir xristofir
xristYfir'
>>> bmpm('Christopher', language_arg='English')
'tzristofir tzrQstofir tzristafir tzrQstafir xristofir xrQstofir xristafir
xrQstafir'
>>> bmpm('Christopher', language_arg='German', name_mode='ash')
'xrQstopir xrQstYpir xristopir xristYpir xrQstofir xrQstYfir xristofir
xristYfir'
```

```
>>> bmpm('Christopher', language_arg='German', match_mode='exact')
'xriStopher xriStofer xristopher xristofer'
```

abydos.phonetic.**caverphone**(*word*, *version=2*)

Return the Caverphone code for a word.

A description of version 1 of the algorithm can be found in [Hoo02].

A description of version 2 of the algorithm can be found in [Hoo04].

**Parameters**

- **word** (*str*) – the word to transform
- **version** (*int*) – the version of Caverphone to employ for encoding (defaults to 2)

**Returns** the Caverphone value

**Return type** str

```
>>> caverphone('Christopher')
'KRSTFA1111'
>>> caverphone('Niall')
'NA11111111'
>>> caverphone('Smith')
'SMT1111111'
```

(continues on next page)

(continued from previous page)

```
>>> caverphone('Schmidt')
'SKMT111111'
```

```
>>> caverphone('Christopher', 1)
'KRSTF1'
>>> caverphone('Niall', 1)
'N11111'
>>> caverphone('Smith', 1)
'SMT111'
>>> caverphone('Schmidt', 1)
'SKMT11'
```

abydos.phonetic.**davidson**(*lname*, *fname='.'*, *omit_fname=False*)

Return Davidson's Consonant Code.

This is based on the name compression system described in [Dav62].

[Dol70] identifies this as having been the name compression algorithm used by SABRE.

**Parameters**

- **lname** (*str*) – Last name (or word) to be encoded
- **fname** (*str*) – First name (optional), of which the first character is included in the code.
- **omit_fname** (*bool*) – Set to True to completely omit the first character of the first name

**Returns** Davidson's Consonant Code

**Return type** str

```
>>> davidson('Gough')
'G   .'
>>> davidson('pneuma')
'PNM .'
>>> davidson('knight')
'KNGT.'
>>> davidson('trice')
'TRC .'
>>> davidson('judge')
'JDG .'
>>> davidson('Smith', 'James')
'SMT J'
>>> davidson('Wasserman', 'Tabitha')
'WSRMT'
```

abydos.phonetic.**dm_soundex**(*word*, *max_length=6*, *zero_pad=True*)

Return the Daitch-Mokotoff Soundex code for a word.

Based on Daitch-Mokotoff Soundex [Mok97], this returns values of a word as a set. A collection is necessary since there can be multiple values for a single word.

**Parameters**

- **word** – the word to transform
- **max_length** – the length of the code returned (defaults to 6; must be between 6 and 64)
- **zero_pad** – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Daitch-Mokotoff Soundex value

**Return type** str

```
>>> sorted(dm_soundex('Christopher'))
['494379', '594379']
>>> dm_soundex('Niall')
{'680000'}
>>> dm_soundex('Smith')
{'463000'}
>>> dm_soundex('Schmidt')
{'463000'}
```

```
>>> sorted(dm_soundex('The quick brown fox', max_length=20,
... zero_pad=False))
['35457976754', '3557976754']
```

abydos.phonetic.**dolby**(*word*, *max_length=-1*, *keep_vowels=False*, *vowel_char='*'*)

Return the Dolby Code of a name.

This follows "A Spelling Equivalent Abbreviation Algorithm For Personal Names" from [Dol70] and [C+69].

**Parameters**

- **word** – the word to encode
- **max_length** – maximum length of the returned Dolby code – this also activates the fixed-length code mode if it is greater than 0
- **keep_vowels** – if True, retains all vowel markers
- **vowel_char** – the vowel marker character (default to *)

**Returns** the Dolby Code

**Return type** str

```
>>> dolby('Hansen')
'H*NSN'
>>> dolby('Larsen')
'L*RSN'
>>> dolby('Aagaard')
'*GR'
>>> dolby('Braaten')
'BR*DN'
>>> dolby('Sandvik')
'S*NVK'
>>> dolby('Hansen', max_length=6)
'H*NS*N'
>>> dolby('Larsen', max_length=6)
'L*RS*N'
>>> dolby('Aagaard', max_length=6)
'*G*R  '
>>> dolby('Braaten', max_length=6)
'BR*D*N'
>>> dolby('Sandvik', max_length=6)
'S*NF*K'
```

```
>>> dolby('Smith')
'SM*D'
>>> dolby('Waters')
'W*DRS'
```

(continues on next page)

(continued from previous page)

```
>>> dolby('James')
'J*MS'
>>> dolby('Schmidt')
'SM*D'
>>> dolby('Ashcroft')
'*SKRFD'
>>> dolby('Smith', max_length=6)
'SM*D  '
>>> dolby('Waters', max_length=6)
'W*D*RS'
>>> dolby('James', max_length=6)
'J*M*S '
>>> dolby('Schmidt', max_length=6)
'SM*D  '
>>> dolby('Ashcroft', max_length=6)
'*SKRFD'
```

abydos.phonetic.**double_metaphone**(*word*, *max_length=-1*)

Return the Double Metaphone code for a word.

Based on Lawrence Philips' (Visual) C++ code from 1999 [Phi00].

**Parameters**

- **word** – the word to transform
- **max_length** – the maximum length of the returned Double Metaphone codes (defaults to 64, but in Philips' original implementation this was 4)

**Returns** the Double Metaphone value(s)

**Return type** tuple

```
>>> double_metaphone('Christopher')
('KRSTFR', '')
>>> double_metaphone('Niall')
('NL', '')
>>> double_metaphone('Smith')
('SM0', 'XMT')
>>> double_metaphone('Schmidt')
('XMT', 'SMT')
```

abydos.phonetic.**eudex**(*word*, *max_length=8*)

Return the eudex phonetic hash of a word.

This implementation of eudex phonetic hashing is based on the specification (not the reference implementation) at [Tic].

Further details can be found at [Tic16].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length in bits of the code returned (default 8)

**Returns** the eudex hash

**Return type** int

```
>>> eudex('Colin')
432345564238053650
>>> eudex('Christopher')
433648490138894409
>>> eudex('Niall')
648518346341351840
>>> eudex('Smith')
720575940412906756
>>> eudex('Schmidt')
720589151732307997
```

abydos.phonetic.**fonem**(*word*)

Return the FONEM code of a word.

FONEM is a phonetic algorithm designed for French (particularly surnames in Saguenay, Canada), defined in [BBL81].

Guillaume Plique's Javascript implementation [Pli18] at https://github.com/Yomguithereal/talisman/blob/master/src/phonetics/french/fonem.js was also consulted for this implementation.

**Parameters** **word** (*str*) – the word to transform

**Returns** the FONEM code

**Return type** str

```
>>> fonem('Marchand')
'MARCHEN'
>>> fonem('Beaulieu')
'BOLIEU'
>>> fonem('Beaumont')
'BOMON'
>>> fonem('Legrand')
'LEGREN'
>>> fonem('Pelletier')
'PELETIER'
```

abydos.phonetic.**fuzzy_soundex**(*word*, *max_length=5*, *zero_pad=True*)

Return the Fuzzy Soundex code for a word.

Fuzzy Soundex is an algorithm derived from Soundex, defined in [HM02].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Fuzzy Soundex value

**Return type** str

```
>>> fuzzy_soundex('Christopher')
'K6931'
>>> fuzzy_soundex('Niall')
'N4000'
>>> fuzzy_soundex('Smith')
'S5300'
>>> fuzzy_soundex('Smith')
'S5300'
```

abydos.phonetic.**haase_phonetik**(*word*, *primary_only=False*)

Return the Haase Phonetik (numeric output) code for a word.

Based on the algorithm described at [Pra15].

Based on the original [HH00].

While the output code is numeric, it is nevertheless a str.

**Parameters**

- **word** (*str*) – the word to transform
- **primary_only** (*bool*) – if True, only the primary code is returned

**Returns** the Haase Phonetik value as a numeric string

**Return type** tuple

```
>>> haase_phonetik('Joachim')
('9496',)
>>> haase_phonetik('Christoph')
('4798293', '8798293')
>>> haase_phonetik('Jörg')
('974',)
>>> haase_phonetik('Smith')
('8692',)
>>> haase_phonetik('Schmidt')
('8692', '4692')
```

abydos.phonetic.**henry_early**(*word*, *max_length=3*)

Calculate the early version of the Henry code for a word.

The early version of Henry coding is given in [LegareLC72]. This is different from the later version defined in [Hen76].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 3)

**Returns** the early Henry code

**Return type** str

```
>>> henry_early('Marchand')
'MRC'
>>> henry_early('Beaulieu')
'BL'
>>> henry_early('Beaumont')
'BM'
>>> henry_early('Legrand')
'LGR'
>>> henry_early('Pelletier')
'PLT'
```

abydos.phonetic.**koelner_phonetik**(*word*)

Return the Kölner Phonetik (numeric output) code for a word.

Based on the algorithm defined by [Pos69].

While the output code is numeric, it is still a str because 0s can lead the code.

**Parameters** **word** (*str*) – the word to transform

**Returns** the Kölner Phonetik value as a numeric string

**Return type** str

```
>>> koelner_phonetik('Christopher')
'478237'
>>> koelner_phonetik('Niall')
'65'
>>> koelner_phonetik('Smith')
'862'
>>> koelner_phonetik('Schmidt')
'862'
>>> koelner_phonetik('Müller')
'657'
>>> koelner_phonetik('Zimmermann')
'86766'
```

abydos.phonetic.**koelner_phonetik_alpha**(*word*)

Return the Kölner Phonetik (alphabetic output) code for a word.

**Parameters** **word** (*str*) – the word to transform

**Returns** the Kölner Phonetik value as an alphabetic string

**Return type** str

```
>>> koelner_phonetik_alpha('Smith')
'SNT'
>>> koelner_phonetik_alpha('Schmidt')
'SNT'
>>> koelner_phonetik_alpha('Müller')
'NLR'
>>> koelner_phonetik_alpha('Zimmermann')
'SNRNN'
```

abydos.phonetic.**koelner_phonetik_num_to_alpha**(*num*)

Convert a Kölner Phonetik code from numeric to alphabetic.

**Parameters** **num** (*str*) – a numeric Kölner Phonetik representation (can be a str or an int)

**Returns** an alphabetic representation of the same word

**Return type** str

```
>>> koelner_phonetik_num_to_alpha('862')
'SNT'
>>> koelner_phonetik_num_to_alpha('657')
'NLR'
>>> koelner_phonetik_num_to_alpha('86766')
'SNRNN'
```

abydos.phonetic.**lein**(*word*, *max_length=4*, *zero_pad=True*)

Return the Lein code for a word.

This is Lein name coding, described in [MKTM77].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length (default 4) of the code to return

- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Lein code

**Return type** str

```
>>> lein('Christopher')
'C351'
>>> lein('Niall')
'N300'
>>> lein('Smith')
'S210'
>>> lein('Schmidt')
'S521'
```

abydos.phonetic.**metaphone**(*word*, *max_length=-1*)

Return the Metaphone code for a word.

Based on Lawrence Philips' Pick BASIC code from 1990 [Phi90b], as described in [Phi90a]. This incorporates some corrections to the above code, particularly some of those suggested by Michael Kuhn in [Kuh95].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length of the returned Metaphone code (defaults to 64, but in Philips' original implementation this was 4)

**Returns** the Metaphone value

**Return type** str

```
>>> metaphone('Christopher')
'KRSTFR'
>>> metaphone('Niall')
'NL'
>>> metaphone('Smith')
'SM0'
>>> metaphone('Schmidt')
'SKMTT'
```

abydos.phonetic.**metasoundex**(*word*, *lang='en'*)

Return the MetaSoundex code for a word.

This is based on [KV17]. Only English ('en') and Spanish ('es') languages are supported, as in the original.

**Parameters**

- **word** (*str*) – the word to transform
- **lang** (*str*) – either 'en' for English or 'es' for Spanish

**Returns** the MetaSoundex code

**Return type** str

```
>>> metasoundex('Smith')
'4500'
>>> metasoundex('Waters')
'7362'
>>> metasoundex('James')
'1520'
>>> metasoundex('Schmidt')
```

(continues on next page)

(continued from previous page)

```
'4530'
>>> metasoundex('Ashcroft')
'0261'
>>> metasoundex('Perez', lang='es')
'094'
>>> metasoundex('Martinez', lang='es')
'69364'
>>> metasoundex('Gutierrez', lang='es')
'83994'
>>> metasoundex('Santiago', lang='es')
'4638'
>>> metasoundex('Nicolás', lang='es')
'6754'
```

abydos.phonetic.**mra**(*word*)

Return the MRA personal numeric identifier (PNI) for a word.

A description of the Western Airlines Surname Match Rating Algorithm can be found on page 18 of [MKTM77].

**Parameters** **word** (*str*) – the word to transform

**Returns** the MRA PNI

**Return type** str

```
>>> mra('Christopher')
'CHRPHR'
>>> mra('Niall')
'NL'
>>> mra('Smith')
'SMTH'
>>> mra('Schmidt')
'SCHMDT'
```

abydos.phonetic.**norphone**(*word*)

Return the Norphone code.

The reference implementation by Lars Marius Garshol is available in [Gar15].

Norphone was designed for Norwegian, but this implementation has been extended to support Swedish vowels as well. This function incorporates the "not implemented" rules from the above file's rule set.

**Parameters** **word** (*str*) – the word to transform

**Returns** the Norphone code

**Return type** str

```
>>> norphone('Hansen')
'HNSN'
>>> norphone('Larsen')
'LRSN'
>>> norphone('Aagaard')
'ÅKRT'
>>> norphone('Braaten')
'BRTN'
>>> norphone('Sandvik')
'SNVK'
```

abydos.phonetic.**nrl**(*word*)

Return the Naval Research Laboratory phonetic encoding of a word.

This is defined by [EJMS76].

**Parameters** **word** (*str*) – the word to transform

**Returns** the NRL phonetic encoding

**Return type** str

```
>>> nrl('the')
'DHAX'
>>> nrl('round')
'rAWnd'
>>> nrl('quick')
'kwIHk'
>>> nrl('eaten')
'IYtEHn'
>>> nrl('Smith')
'smIHTH'
>>> nrl('Larsen')
'lAArsEHn'
```

abydos.phonetic.**nysiis**(*word*, *max_length=6*, *modified=False*)

Return the NYSIIS code for a word.

The New York State Identification and Intelligence System algorithm is defined in [Taf70].

The modified version of this algorithm is described in Appendix B of [LA77].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length (default 6) of the code to return
- **modified** (*bool*) – indicates whether to use USDA modified NYSIIS

**Returns** the NYSIIS value

**Return type** str

```
>>> nysiis('Christopher')
'CRASTA'
>>> nysiis('Niall')
'NAL'
>>> nysiis('Smith')
'SNAT'
>>> nysiis('Schmidt')
'SNAD'
```

```
>>> nysiis('Christopher', max_length=-1)
'CRASTAFAR'
```

```
>>> nysiis('Christopher', max_length=8, modified=True)
'CRASTAFA'
>>> nysiis('Niall', max_length=8, modified=True)
'NAL'
>>> nysiis('Smith', max_length=8, modified=True)
'SNAT'
```

(continues on next page)

(continued from previous page)

```
>>> nysiis('Schmidt', max_length=8, modified=True)
'SNAD'
```

abydos.phonetic.**onca**(*word*, *max_length=4*, *zero_pad=True*)

Return the Oxford Name Compression Algorithm (ONCA) code for a word.

This is the Oxford Name Compression Algorithm, based on [Gil97].

I can find no complete description of the "anglicised version of the NYSIIS method" identified as the first step in this algorithm, so this is likely not a precisely correct implementation, in that it employs the standard NYSIIS algorithm.

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length (default 5) of the code to return
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the ONCA code

**Return type** str

```
>>> onca('Christopher')
'C623'
>>> onca('Niall')
'N400'
>>> onca('Smith')
'S530'
>>> onca('Schmidt')
'S530'
```

abydos.phonetic.**parmar_kumbharana**(*word*)

Return the Parmar-Kumbharana encoding of a word.

This is based on the phonetic algorithm proposed in [PK14].

**Parameters** **word** (*str*) – the word to transform

**Returns** the Parmar-Kumbharana encoding

**Return type** str

```
>>> parmar_kumbharana('Gough')
'GF'
>>> parmar_kumbharana('pneuma')
'NM'
>>> parmar_kumbharana('knight')
'NT'
>>> parmar_kumbharana('trice')
'TRS'
>>> parmar_kumbharana('judge')
'JJ'
```

abydos.phonetic.**phonem**(*word*)

Return the Phonem code for a word.

Phonem is defined in [GM88].

This version is based on the Perl implementation documented at [Wil05]. It includes some enhancements presented in the Java port at [dcm4che].

Phonem is intended chiefly for German names/words.

**Parameters** **word** (*str*) – the word to transform

**Returns** the Phonem value

**Return type** str

```
>>> phonem('Christopher')
'CRYSDOVR'
>>> phonem('Niall')
'NYAL'
>>> phonem('Smith')
'SMYD'
>>> phonem('Schmidt')
'CMYD'
```

abydos.phonetic.**phonet**(*word*, *mode=1*, *lang='de'*)

Return the phonet code for a word.

phonet ("Hannoveraner Phonetik") was developed by Jörg Michael and documented in [Mic99].

This is a port of Jesper Zedlitz's code, which is licensed LGPL [Zed15].

That is, in turn, based on Michael's C code, which is also licensed LGPL [Mic07].

**Parameters**

- **word** (*str*) – the word to transform
- **mode** (*int*) – the ponet variant to employ (1 or 2)
- **lang** (*str*) – 'de' (default) for German 'none' for no language

**Returns** the phonet value

**Return type** str

```
>>> phonet('Christopher')
'KRISTOFA'
>>> phonet('Niall')
'NIAL'
>>> phonet('Smith')
'SMIT'
>>> phonet('Schmidt')
'SHMIT'
```

```
>>> phonet('Christopher', mode=2)
'KRIZTUFA'
>>> phonet('Niall', mode=2)
'NIAL'
>>> phonet('Smith', mode=2)
'ZNIT'
>>> phonet('Schmidt', mode=2)
'ZNIT'
```

```
>>> phonet('Christopher', lang='none')
'CHRISTOPHER'
>>> phonet('Niall', lang='none')
'NIAL'
>>> phonet('Smith', lang='none')
```

(continues on next page)

(continued from previous page)

```
'SMITH'
>>> phonet('Schmidt', lang='none')
'SCHMIDT'
```

abydos.phonetic.**phonetic_spanish**(*word*, *max_length=-1*)

Return the PhoneticSpanish coding of word.

This follows the coding described in [AmonME12] and [delPAngelesEGGM15].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to unlimited)

**Returns** the PhoneticSpanish code

**Return type** str

```
>>> phonetic_spanish('Perez')
'094'
>>> phonetic_spanish('Martinez')
'69364'
>>> phonetic_spanish('Gutierrez')
'83994'
>>> phonetic_spanish('Santiago')
'4638'
>>> phonetic_spanish('Nicolás')
'6454'
```

abydos.phonetic.**phonex**(*word*, *max_length=4*, *zero_pad=True*)

Return the Phonex code for a word.

Phonex is an algorithm derived from Soundex, defined in [LR96].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Phonex value

**Return type** str

```
>>> phonex('Christopher')
'C623'
>>> phonex('Niall')
'N400'
>>> phonex('Schmidt')
'S253'
>>> phonex('Smith')
'S530'
```

abydos.phonetic.**phonix**(*word*, *max_length=4*, *zero_pad=True*)

Return the Phonix code for a word.

Phonix is a Soundex-like algorithm defined in [Gad90].

This implementation is based on: - [Pfe00] - [Chr11] - [Kollar]

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Phonix value

**Return type** str

```
>>> phonix('Christopher')
'K683'
>>> phonix('Niall')
'N400'
>>> phonix('Smith')
'S530'
>>> phonix('Schmidt')
'S530'
```

abydos.phonetic.**pshp_soundex_first**(*fname*, *max_length=4*, *german=False*)

Calculate the PSHP Soundex/Viewex Coding of a first name.

This coding is based on [HBD76].

Reference was also made to the German version of the same: [HBD79].

A separate function, pshp_soundex_last() is used for last names.

**Parameters**

- **fname** (*str*) – the first name to encode
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **german** (*bool*) – set to True if the name is German (different rules apply)

**Returns** the PSHP Soundex/Viewex Coding

**Return type** str

```
>>> pshp_soundex_first('Smith')
'S530'
>>> pshp_soundex_first('Waters')
'W352'
>>> pshp_soundex_first('James')
'J700'
>>> pshp_soundex_first('Schmidt')
'S500'
>>> pshp_soundex_first('Ashcroft')
'A220'
>>> pshp_soundex_first('John')
'J500'
>>> pshp_soundex_first('Colin')
'K400'
>>> pshp_soundex_first('Niall')
'N400'
>>> pshp_soundex_first('Sally')
'S400'
>>> pshp_soundex_first('Jane')
'J500'
```

abydos.phonetic.**pshp_soundex_last**(*lname*, *max_length=4*, *german=False*)
Calculate the PSHP Soundex/Viewex Coding of a last name.

This coding is based on [HBD76].

Reference was also made to the German version of the same: [HBD79].

A separate function, pshp_soundex_first() is used for first names.

**Parameters**

- **lname** (*str*) – the last name to encode
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **german** (*bool*) – set to True if the name is German (different rules apply)

**Returns** the PSHP Soundex/Viewex Coding

**Return type** str

```
>>> pshp_soundex_last('Smith')
'S530'
>>> pshp_soundex_last('Waters')
'W350'
>>> pshp_soundex_last('James')
'J500'
>>> pshp_soundex_last('Schmidt')
'S530'
>>> pshp_soundex_last('Ashcroft')
'A225'
```

abydos.phonetic.**refined_soundex**(*word*, *max_length=-1*, *zero_pad=False*, *retain_vowels=False*)
Return the Refined Soundex code for a word.

This is Soundex, but with more character classes. It was defined at [Boy98].

**Parameters**

- **word** – the word to transform
- **max_length** – the length of the code returned (defaults to unlimited)
- **zero_pad** – pad the end of the return value with 0s to achieve a max_length string
- **retain_vowels** – retain vowels (as 0) in the resulting code

**Returns** the Refined Soundex value

**Return type** str

```
>>> refined_soundex('Christopher')
'C393619'
>>> refined_soundex('Niall')
'N87'
>>> refined_soundex('Smith')
'S386'
>>> refined_soundex('Schmidt')
'S386'
```

abydos.phonetic.**reth_schek_phonetik**(*word*)
Return Reth-Schek Phonetik code for a word.

This algorithm is proposed in [vonRethS77].

Since I couldn't secure a copy of that document (maybe I'll look for it next time I'm in Germany), this implementation is based on what I could glean from the implementations published by German Record Linkage Center (www.record-linkage.de):

- Privacy-preserving Record Linkage (PPRL) (in R) [Ruk18]
- Merge ToolBox (in Java) [SBB04]

Rules that are unclear:

- Should 'C' become 'G' or 'Z'? (PPRL has both, 'Z' rule blocked)
- Should 'CC' become 'G'? (PPRL has blocked 'CK' that may be typo)
- Should 'TUI' -> 'ZUI' rule exist? (PPRL has rule, but I can't think of a German word with '-tui-' in it.)
- Should we really change 'SCH' -> 'CH' and then 'CH' -> 'SCH'?

**Parameters** **word** (*str*) – the word to transform

**Returns** the Reth-Schek Phonetik code

**Return type** str

```
>>> reth_schek_phonetik('Joachim')
'JOAGHIM'
>>> reth_schek_phonetik('Christoph')
'GHRISDOF'
>>> reth_schek_phonetik('Jörg')
'JOERG'
>>> reth_schek_phonetik('Smith')
'SMID'
>>> reth_schek_phonetik('Schmidt')
'SCHMID'
```

abydos.phonetic.**roger_root**(*word*, *max_length=5*, *zero_pad=True*)

Return the Roger Root code for a word.

This is Roger Root name coding, described in [MKTM77].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length (default 5) of the code to return
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Roger Root code

**Return type** str

```
>>> roger_root('Christopher')
'06401'
>>> roger_root('Niall')
'02500'
>>> roger_root('Smith')
'00310'
>>> roger_root('Schmidt')
'06310'
```

abydos.phonetic.**russell_index**(*word*)

Return the Russell Index (integer output) of a word.

This follows Robert C. Russell's Index algorithm, as described in [Rus18].

**Parameters** **word** (*str*) – the word to transform

**Returns** the Russell Index value

**Return type** int

```
>>> russell_index('Christopher')
3813428
>>> russell_index('Niall')
715
>>> russell_index('Smith')
3614
>>> russell_index('Schmidt')
3614
```

abydos.phonetic.**russell_index_alpha**(*word*)

Return the Russell Index (alphabetic output) for the word.

This follows Robert C. Russell's Index algorithm, as described in [Rus18].

**Parameters** **word** (*str*) – the word to transform

**Returns** the Russell Index value as an alphabetic string

**Return type** str

```
>>> russell_index_alpha('Christopher')
'CRACDBR'
>>> russell_index_alpha('Niall')
'NAL'
>>> russell_index_alpha('Smith')
'CMAD'
>>> russell_index_alpha('Schmidt')
'CMAD'
```

abydos.phonetic.**russell_index_num_to_alpha**(*num*)

Convert the Russell Index integer to an alphabetic string.

This follows Robert C. Russell's Index algorithm, as described in [Rus18].

**Parameters** **num** (*int*) – a Russell Index integer value

**Returns** the Russell Index as an alphabetic string

**Return type** str

```
>>> russell_index_num_to_alpha(3813428)
'CRACDBR'
>>> russell_index_num_to_alpha(715)
'NAL'
>>> russell_index_num_to_alpha(3614)
'CMAD'
```

abydos.phonetic.**sfinxbis**(*word*, *max_length=-1*)

Return the SfinxBis code for a word.

SfinxBis is a Soundex-like algorithm defined in [Axe09].

This implementation follows the reference implementation: [Sjoo09].

SfinxBis is intended chiefly for Swedish names.

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to unlimited)

**Returns** the SfinxBis value

**Return type** tuple

```
>>> sfinxbis('Christopher')
('K68376',)
>>> sfinxbis('Niall')
('N4',)
>>> sfinxbis('Smith')
('S53',)
>>> sfinxbis('Schmidt')
('S53',)
```

```
>>> sfinxbis('Johansson')
('J585',)
>>> sfinxbis('Sjöberg')
('#162',)
```

abydos.phonetic.**sound_d**(*word*, *max_length=4*)

Return the SoundD code.

SoundD is defined in [VB12].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)

**Returns** the SoundD code

**Return type** str

```
>>> sound_d('Gough')
'2000'
>>> sound_d('pneuma')
'5500'
>>> sound_d('knight')
'5300'
>>> sound_d('trice')
'3620'
>>> sound_d('judge')
'2200'
```

abydos.phonetic.**soundex**(*word*, *max_length=4*, *var='American'*, *reverse=False*, *zero_pad=True*)

Return the Soundex code for a word.

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **var** (*str*) – the variant of the algorithm to employ (defaults to 'American'):
  - 'American' follows the American Soundex algorithm, as described at [UnitedStates07] and in [Knu98]; this is also called Miracode

  - 'special' follows the rules from the 1880-1910 US Census retrospective re-analysis, in which h & w are not treated as blocking consonants but as vowels. Cf. [Rep13].
  - 'Census' follows the rules laid out in GIL 55 [UnitedStates97] by the US Census, including coding prefixed and unprefixed versions of some names
- **reverse** (*bool*) – reverse the word before computing the selected Soundex (defaults to False); This results in "Reverse Soundex", which is useful for blocking in cases where the initial elements may be in error.
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the Soundex value

**Return type** str

```
>>> soundex("Christopher")
'C623'
>>> soundex("Niall")
'N400'
>>> soundex('Smith')
'S530'
>>> soundex('Schmidt')
'S530'
```

```
>>> soundex('Christopher', max_length=-1)
'C623160000000000000000000000000000000000000000000000000000000000000000000000'
>>> soundex('Christopher', max_length=-1, zero_pad=False)
'C62316'
```

```
>>> soundex('Christopher', reverse=True)
'R132'
```

```
>>> soundex('Ashcroft')
'A261'
>>> soundex('Asicroft')
'A226'
>>> soundex('Ashcroft', var='special')
'A226'
>>> soundex('Asicroft', var='special')
'A226'
```

abydos.phonetic.**soundex_br**(*word*, *max_length=4*, *zero_pad=True*)

Return the SoundexBR encoding of a word.

This is based on [Mar15].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 4)
- **zero_pad** (*bool*) – pad the end of the return value with 0s to achieve a max_length string

**Returns** the SoundexBR code

**Return type** str

```
>>> soundex_br('Oliveira')
'O416'
>>> soundex_br('Almeida')
'A453'
>>> soundex_br('Barbosa')
'B612'
>>> soundex_br('Araújo')
'A620'
>>> soundex_br('Gonçalves')
'G524'
>>> soundex_br('Goncalves')
'G524'
```

abydos.phonetic.**spanish_metaphone**(*word*, *max_length=6*, *modified=False*)

Return the Spanish Metaphone of a word.

This is a quick rewrite of the Spanish Metaphone Algorithm, as presented at https://github.com/amsqr/Spanish-Metaphone and discussed in [MLM12].

Modified version based on [delPAngelesBailonM16].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the length of the code returned (defaults to 6)
- **modified** (*bool*) – Set to True to use del Pilar Angeles & Bailón-Miguel's modified version of the algorithm

**Returns** the Spanish Metaphone code

**Return type** str

```
>>> spanish_metaphone('Perez')
'PRZ'
>>> spanish_metaphone('Martinez')
'MRTNZ'
>>> spanish_metaphone('Gutierrez')
'GTRRZ'
>>> spanish_metaphone('Santiago')
'SNTG'
>>> spanish_metaphone('Nicolás')
'NKLS'
```

abydos.phonetic.**spfc**(*word*)

Return the Standardized Phonetic Frequency Code (SPFC) of a word.

Standardized Phonetic Frequency Code is roughly Soundex-like. This implementation is based on page 19-21 of [MKTM77].

**Parameters** **word** (*str*) – the word to transform

**Returns** the SPFC value

**Return type** str

```
>>> spfc('Christopher Smith')
'01160'
>>> spfc('Christopher Schmidt')
'01160'
```

(continues on next page)

(continued from previous page)

```
>>> spfc('Niall Smith')
'01660'
>>> spfc('Niall Schmidt')
'01660'
```

```
>>> spfc('L.Smith')
'01960'
>>> spfc('R.Miller')
'65490'
```

```
>>> spfc(('L', 'Smith'))
'01960'
>>> spfc(('R', 'Miller'))
'65490'
```

abydos.phonetic.**statistics_canada**(*word*, *max_length=4*)

Return the Statistics Canada code for a word.

The original description of this algorithm could not be located, and may only have been specified in an unpublished TR. The coding does not appear to be in use by Statistics Canada any longer. In its place, this is an implementation of the "Census modified Statistics Canada name coding procedure".

**The modified version of this algorithm is described in Appendix B of** [MKTM77].

**Parameters**

- **word** (*str*) – the word to transform
- **max_length** (*int*) – the maximum length (default 4) of the code to return

**Returns** the Statistics Canada name code value

**Return type** str

```
>>> statistics_canada('Christopher')
'CHRS'
>>> statistics_canada('Niall')
'NL'
>>> statistics_canada('Smith')
'SMTH'
>>> statistics_canada('Schmidt')
'SCHM'
```

#### 3.1.1.9 abydos.qgram module

abydos.qgram.

The qgram module defines the QGrams multi-set class

**class** abydos.qgram.**QGrams**(*term*, *qval=2*, *start_stop='$#'*, *skip=0*)

Bases: collections.Counter

A q-gram class, which functions like a bag/multiset.

A q-gram is here defined as all sequences of q characters. Q-grams are also known as k-grams and n-grams, but the term n-gram more typically refers to sequences of whitespace-delimited words in a string, where q-gram refers to sequences of characters in a word or string.

**count**()

Return q-grams count.

**Returns** the total count of q-grams in a QGrams object

**Return type** int

```
>>> qg = QGrams('AATTATAT')
>>> qg.count()
9
```

```
>>> qg = QGrams('AATTATAT', qval=1, start_stop='')
>>> qg.count()
8
```

```
>>> qg = QGrams('AATTATAT', qval=3, start_stop='')
>>> qg.count()
6
```

**ordered_list = []**

**term = ''**

**term_ss = ''**

#### 3.1.1.10 abydos.stats module

abydos.stats.

The stats module defines functions for calculating various statistical data about linguistic objects.

This includes the ConfusionTable object, which includes members capable of calculating the following data based on a confusion table:

- population counts
- precision, recall, specificity, negative predictive value, fall-out, false discovery rate, accuracy, balanced accuracy, informedness, and markedness
- various means of the precision & recall, including: arithmetic, geometric, harmonic, quadratic, logarithmic, contraharmonic, identric (exponential), & Hölder (power/generalized) means
- $F_\beta$-scores, $E$-scores, $G$-measures, along with special functions for $F_1$, $F_{0.5}$, & $F_2$ scores
- significance & Matthews correlation coefficient calculation

Functions are provided for calculating the following means:

- arithmetic
- geometric
- harmonic
- quadratic
- contraharmonic
- logarithmic
- identric (exponential)
- Seiffert's

- Lehmer
- Heronian
- Hölder (power/generalized)
- Stolkarsky
- arithmetic-geometric
- geometric-harmonic
- arithmetic-geometric-harmonic

And for calculating:

- midrange
- median
- mode
- variance
- standard deviation

**class** abydos.stats.**ConfusionTable**(*tp=0*, *tn=0*, *fp=0*, *fn=0*)

Bases: object

ConfusionTable object.

This object is initialized by passing either four integers (or a tuple of four integers) representing the squares of a confusion table: true positives, true negatives, false positives, and false negatives

The object possesses methods for the calculation of various statistics based on the confusion table.

**accuracy**()

Return accuracy.

Accuracy is defined as $\frac{tp+tn}{population}$

Cf. https://en.wikipedia.org/wiki/Accuracy

**Returns** The accuracy of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.accuracy()
0.782608695652174
```

**accuracy_gain**()

Return gain in accuracy.

The gain in accuracy is defined as: $G(accuracy) = \frac{accuracy}{random\ accuracy}$

Cf. https://en.wikipedia.org/wiki/Gain_(information_retrieval)

**Returns** The gain in accuracy of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.accuracy_gain()
1.4325259515570934
```

**balanced_accuracy**()
Return balanced accuracy.

Balanced accuracy is defined as $\frac{sensitivity+specificity}{2}$

Cf. https://en.wikipedia.org/wiki/Accuracy

**Returns** The balanced accuracy of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.balanced_accuracy()
0.775
```

**cond_neg_pop**()
Return condition negative population.

**Returns** The condition negative population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.cond_neg_pop()
80
```

**cond_pos_pop**()
Return condition positive population.

**Returns** The condition positive population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.cond_pos_pop()
150
```

**correct_pop**()
Return correct population.

**Returns** the correct population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.correct_pop()
180
```

**e_score**(*beta=1*)
Return $E$-score.

This is Van Rijsbergen's effectiveness measure

Cf. https://en.wikipedia.org/wiki/Information_retrieval#F-measure

**Returns** The $E$-score of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.e_score()
0.17241379310344818
```

**error_pop**()
Return error population.

**Returns** The error population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.error_pop()
50
```

**f1_score**()
Return $F_1$ score.

$F_1$ score is the harmonic mean of precision and recall: $2 \cdot \frac{precision \cdot recall}{precision + recall}$

Cf. https://en.wikipedia.org/wiki/F1_score

**Returns** The $F_1$ of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.f1_score()
0.8275862068965516
```

**f2_score**()
Return $F_2$.

The $F_2$ score emphasizes recall over precision in comparison to the $F_1$ score

Cf. https://en.wikipedia.org/wiki/F1_score

**Returns** The $F_2$ of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.f2_score()
0.8108108108108109
```

**f_measure**()
Return $F$-measure.

$F$-measure is the harmonic mean of precision and recall: $2 \cdot \frac{precision \cdot recall}{precision + recall}$

Cf. https://en.wikipedia.org/wiki/F1_score

**Returns** The math:$F$-measure of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.f_measure()
0.8275862068965516
```

**fallout**()
Return fall-out.

Fall-out is defined as $\frac{fp}{fp + tn}$

AKA false positive rate (FPR)

Cf. https://en.wikipedia.org/wiki/Information_retrieval#Fall-out

**Returns** The fall-out of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.fallout()
0.25
```

**false_neg**()
Return false negatives.

**Returns** the false negatives of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.false_neg()
30
```

**false_pos**()
Return false positives.

**Returns** the false positives of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.false_pos()
20
```

**fbeta_score**(*beta=1*)
Return $F_\beta$ score.

$F_\beta$ for a positive real value $\beta$ "measures the effectiveness of retrieval with respect to a user who attaches $\beta$ times as much importance to recall as precision" (van Rijsbergen 1979)

$F_\beta$ score is defined as: $(1 + \beta^2) \cdot \frac{precision \cdot recall}{((\beta^2 \cdot precision) + recall)}$

Cf. https://en.wikipedia.org/wiki/F1_score

**Params numeric beta** The $\beta$ parameter in the above formula

**Returns** The $F_\beta$ of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.fbeta_score()
0.8275862068965518
>>> ct.fbeta_score(beta=0.1)
0.8565371024734982
```

**fdr**()
Return false discovery rate (FDR).

False discovery rate is defined as $\frac{fp}{fp+tp}$

Cf. https://en.wikipedia.org/wiki/False_discovery_rate

**Returns** The false discovery rate of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.fdr()
0.14285714285714285
```

**fhalf_score**()
Return $F_{0.5}$ score.

The $F_{0.5}$ score emphasizes precision over recall in comparison to the $F_1$ score

Cf. https://en.wikipedia.org/wiki/F1_score

**Returns** The $F_{0.5}$ score of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.fhalf_score()
0.8450704225352114
```

**g_measure**()
Return G-measure.

$G$-measure is the geometric mean of precision and recall: $\sqrt{precision \cdot recall}$

This is identical to the Fowlkes–Mallows (FM) index for two clusters.

Cf. https://en.wikipedia.org/wiki/F1_score#G-measure

Cf. https://en.wikipedia.org/wiki/Fowlkes%E2%80%93Mallows_index

**Returns** The $G$-measure of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.g_measure()
0.828078671210825
```

**informedness**()
Return informedness.

Informedness is defined as $sensitivity + specificity - 1$.

AKA Youden's J statistic

AKA DeltaP'

Cf. https://en.wikipedia.org/wiki/Youden%27s_J_statistic

Cf. http://dspace.flinders.edu.au/xmlui/bitstream/handle/2328/27165/Powers%20Evaluation.pdf

**Returns** The informedness of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.informedness()
0.55
```

**kappa_statistic**()
Return $\kappa$ statistic.

The $\kappa$ statistic is defined as: $\kappa = \frac{accuracy - random\ accuracy}{1 - random\ accuracy}$

The $\kappa$ statistic compares the performance of the classifier relative to the performance of a random classifier. $\kappa = 0$ indicates performance identical to random. $\kappa = 1$ indicates perfect predictive success. $\kappa = -1$ indicates perfect predictive failure.

**Returns** The $\kappa$ statistic of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.kappa_statistic()
0.5344129554655871
```

**markedness**()

Return markedness.

Markedness is defined as $precision + npv - 1$

AKA DeltaP

Cf. https://en.wikipedia.org/wiki/Youden%27s_J_statistic

Cf. http://dspace.flinders.edu.au/xmlui/bitstream/handle/2328/27165/Powers%20Evaluation.pdf

**Returns** The markedness of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.markedness()
0.5238095238095237
```

**mcc**()

Return Matthews correlation coefficient (MCC).

The Matthews correlation coefficient is defined as: $\frac{(tp \cdot tn) - (fp \cdot fn)}{\sqrt{(tp+fp)(tp+fn)(tn+fp)(tn+fn)}}$

This is equivalent to the geometric mean of informedness and markedness, defined above.

Cf. https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

**Returns** The Matthews correlation coefficient of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.mcc()
0.5367450401216932
```

**npv**()

Return negative predictive value (NPV).

NPV is defined as $\frac{tn}{tn+fn}$

Cf. https://en.wikipedia.org/wiki/Negative_predictive_value

**Returns** The negative predictive value of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.npv()
0.6666666666666666
```

**population**()
Return population, N.

**Returns** The population (N) of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.population()
230
```

**pr_aghmean**()
Return arithmetic-geometric-harmonic mean of precision & recall.

Iterates over arithmetic, geometric, & harmonic means until they converge to a single value (rounded to 12 digits), following the method described by Raïssouli, Leazizi, & Chergui: http://www.emis.de/journals/JIPAM/images/014_08_JIPAM/014_08.pdf

**Returns** The arithmetic-geometric-harmonic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_aghmean()
0.8280786712108288
```

**pr_agmean**()
Return arithmetic-geometric mean of precision & recall.

Iterates between arithmetic & geometric means until they converge to a single value (rounded to 12 digits)

Cf. https://en.wikipedia.org/wiki/Arithmetic-geometric_mean

**Returns** The arithmetic-geometric mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_agmean()
0.8283250315702829
```

**pr_amean**()
Return arithmetic mean of precision & recall.

The arithmetic mean of precision and recall is defined as: $\frac{precision \cdot recall}{2}$

Cf. https://en.wikipedia.org/wiki/Arithmetic_mean

**Returns** The arithmetic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_amean()
0.8285714285714285
```

**pr_cmean**()
Return contraharmonic mean of precision & recall.

The contraharmonic mean is: $\frac{precision^2+recall^2}{precision+recall}$

Cf. https://en.wikipedia.org/wiki/Contraharmonic_mean

**Returns** The contraharmonic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_cmean()
0.8295566502463055
```

**pr_ghmean**()

Return geometric-harmonic mean of precision & recall.

Iterates between geometric & harmonic means until they converge to a single value (rounded to 12 digits)

Cf. https://en.wikipedia.org/wiki/Geometric-harmonic_mean

**Returns** The geometric-harmonic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_ghmean()
0.8278323841238441
```

**pr_gmean**()

Return geometric mean of precision & recall.

The geometric mean of precision and recall is defined as: $\sqrt{precision \cdot recall}$

Cf. https://en.wikipedia.org/wiki/Geometric_mean

**Returns** The geometric mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_gmean()
0.828078671210825
```

**pr_heronian_mean**()

Return Heronian mean of precision & recall.

The Heronian mean of precision and recall is defined as: $\frac{precision+\sqrt{precision \cdot recall}+recall}{3}$

Cf. https://en.wikipedia.org/wiki/Heronian_mean

**Returns** The Heronian mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_heronian_mean()
0.8284071761178939
```

**pr_hmean**()

Return harmonic mean of precision & recall.

The harmonic mean of precision and recall is defined as: $\frac{2 \cdot precision \cdot recall}{precision+recall}$

Cf. https://en.wikipedia.org/wiki/Harmonic_mean

**Returns** The harmonic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_hmean()
0.8275862068965516
```

**pr_hoelder_mean**(*exp=2*)

Return Hölder (power/generalized) mean of precision & recall.

The power mean of precision and recall is defined as: $\frac{1}{2} \cdot \sqrt[exp]{precision^{exp} + recall^{exp}}$ for $exp \neq 0$, and the geometric mean for $exp = 0$

Cf. https://en.wikipedia.org/wiki/Generalized_mean

**Parameters** **exp** (*numeric*) – The exponent of the Hölder mean

**Returns** The Hölder mean for the given exponent of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_hoelder_mean()
0.8290638930598233
```

**pr_imean**()

Return identric (exponential) mean of precision & recall.

The identric mean is: precision if precision = recall, otherwise $\frac{1}{e} \cdot \sqrt[precision-recall]{\frac{precision^{precision}}{recall^{recall}}}$

Cf. https://en.wikipedia.org/wiki/Identric_mean

**Returns** The identric mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_imean()
0.8284071826325543
```

**pr_lehmer_mean**(*exp=2*)

Return Lehmer mean of precision & recall.

The Lehmer mean is: $\frac{precision^{exp}+recall^{exp}}{precision^{exp-1}+recall^{exp-1}}$

Cf. https://en.wikipedia.org/wiki/Lehmer_mean

**Parameters** **exp** (*numeric*) – The exponent of the Lehmer mean

**Returns** The Lehmer mean for the given exponent of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_lehmer_mean()
0.8295566502463055
```

**pr_lmean**()

Return logarithmic mean of precision & recall.

The logarithmic mean is: 0 if either precision or recall is 0, the precision if they are equal, otherwise $\frac{precision-recall}{ln(precision)-ln(recall)}$

Cf. https://en.wikipedia.org/wiki/Logarithmic_mean

**Returns** The logarithmic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_lmean()
0.8282429171492667
```

**pr_qmean**()

Return quadratic mean of precision & recall.

The quadratic mean of precision and recall is defined as: $\sqrt{\frac{precision^2+recall^2}{2}}$

Cf. https://en.wikipedia.org/wiki/Quadratic_mean

**Returns** The quadratic mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_qmean()
0.8290638930598233
```

**pr_seiffert_mean**()

Return Seiffert's mean of precision & recall.

Seiffert's mean of precision and recall is: $\frac{precision-recall}{4 \cdot arctan\sqrt{\frac{precision}{recall}}-\pi}$

Cf. http://www.helsinki.fi/~hasto/pp/miaPreprint.pdf

**Returns** Seiffer's mean of the confusion table's precision & recall

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.pr_seiffert_mean()
0.8284071696048312
```

**precision**()

Return precision.

Precision is defined as $\frac{tp}{tp+fp}$

AKA positive predictive value (PPV)

Cf. https://en.wikipedia.org/wiki/Precision_and_recall

Cf. https://en.wikipedia.org/wiki/Information_retrieval#Precision

**Returns** The precision of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.precision()
0.8571428571428571
```

**precision_gain**()

Return gain in precision.

The gain in precision is defined as: $G(precision) = \frac{precision}{random\ precision}$

Cf. https://en.wikipedia.org/wiki/Gain_(information_retrieval)

**Returns** The gain in precision of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.precision_gain()
1.3142857142857143
```

**recall**()

Return recall.

Recall is defined as $\frac{tp}{tp+fn}$

AKA sensitivity

AKA true positive rate (TPR)

Cf. https://en.wikipedia.org/wiki/Precision_and_recall

Cf. https://en.wikipedia.org/wiki/Sensitivity_(test)

Cf. https://en.wikipedia.org/wiki/Information_retrieval#Recall

**Returns** The recall of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.recall()
0.8
```

**significance**()

Return the significance, $\chi^2$.

Significance is defined as: $\chi^2 = \frac{(tp \cdot tn - fp \cdot fn)^2 (tp+tn+fp+fn)}{((tp+fp)(tp+fn)(tn+fp)(tn+fn)}$

Also: $\chi^2 = MCC^2 \cdot n$

Cf. https://en.wikipedia.org/wiki/Pearson%27s_chi-square_test

**Returns** The significance of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.significance()
66.26190476190476
```

**specificity**()

Return specificity.

Specificity is defined as $\frac{tn}{tn+fp}$

AKA true negative rate (TNR)

Cf. https://en.wikipedia.org/wiki/Specificity_(tests)

**Returns** The specificity of the confusion table

**Return type** float

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.specificity()
0.75
```

**test_neg_pop**()
Return test negative population.

**Returns** The test negative population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.test_neg_pop()
90
```

**test_pos_pop**()
Return test positive population.

**Returns** The test positive population of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.test_pos_pop()
140
```

**to_dict**()
Cast to dict.

**Returns** the confusion table as a dict

**Return type** dict

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> import pprint
>>> pprint.pprint(ct.to_dict())
{'fn': 30, 'fp': 20, 'tn': 60, 'tp': 120}
```

**to_tuple**()
Cast to tuple.

**Returns** the confusion table as a 4-tuple (tp, tn, fp, fn)

**Return type** tuple

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.to_tuple()
(120, 60, 20, 30)
```

**true_neg**()
Return true negatives.

**Returns** the true negatives of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.true_neg()
60
```

**true_pos**()
Return true positives.

**Returns** the true positives of the confusion table

**Return type** int

```
>>> ct = ConfusionTable(120, 60, 20, 30)
>>> ct.true_pos()
120
```

abydos.stats.**aghmean**(*nums*)

Return arithmetic-geometric-harmonic mean.

Iterates over arithmetic, geometric, & harmonic means until they converge to a single value (rounded to 12 digits), following the method described by Raïssouli, Leazizi, & Chergui: http://www.emis.de/journals/JIPAM/images/014_08_JIPAM/014_08.pdf

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The arithmetic-geometric-harmonic mean of nums

**Return type** float

```
>>> aghmean([1, 2, 3, 4])
2.198327159900212
>>> aghmean([1, 2])
1.4142135623731884
>>> aghmean([0, 5, 1000])
335.0
```

abydos.stats.**agmean**(*nums*)

Return arithmetic-geometric mean.

Iterates between arithmetic & geometric means until they converge to a single value (rounded to 12 digits).

Cf. https://en.wikipedia.org/wiki/Arithmetic-geometric_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The arithmetic-geometric mean of nums

**Return type** float

```
>>> agmean([1, 2, 3, 4])
2.3545004777751077
>>> agmean([1, 2])
1.4567910310469068
>>> agmean([0, 5, 1000])
2.9753977059954195e-13
```

abydos.stats.**amean**(*nums*)

Return arithmetic mean.

The arithmetic mean is defined as: $\frac{\sum nums}{|nums|}$

Cf. https://en.wikipedia.org/wiki/Arithmetic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The arithmetic mean of nums

**Return type** float

```
>>> amean([1, 2, 3, 4])
2.5
>>> amean([1, 2])
1.5
```

(continues on next page)

(continued from previous page)

```
>>> amean([0, 5, 1000])
335.0
```

abydos.stats.**cmean**(*nums*)

Return contraharmonic mean.

The contraharmonic mean is: $\frac{\sum_i x_i^2}{\sum_i x_i}$

Cf. https://en.wikipedia.org/wiki/Contraharmonic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The contraharmonic mean of nums

**Return type** float

```
>>> cmean([1, 2, 3, 4])
3.0
>>> cmean([1, 2])
1.6666666666666667
>>> cmean([0, 5, 1000])
995.0497512437811
```

abydos.stats.**ghmean**(*nums*)

Return geometric-harmonic mean.

Iterates between geometric & harmonic means until they converge to a single value (rounded to 12 digits).

Cf. https://en.wikipedia.org/wiki/Geometric-harmonic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The geometric-harmonic mean of nums

**Return type** float

```
>>> ghmean([1, 2, 3, 4])
2.058868154613003
>>> ghmean([1, 2])
1.3728805006183502
>>> ghmean([0, 5, 1000])
0.0
```

```
>>> ghmean([0, 0])
0.0
>>> ghmean([0, 0, 5])
nan
```

abydos.stats.**gmean**(*nums*)

Return geometric mean.

The geometric mean is defined as: $\sqrt[|nums|]{\prod_i nums_i}$

Cf. https://en.wikipedia.org/wiki/Geometric_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The geometric mean of nums

**Return type** float

```
>>> gmean([1, 2, 3, 4])
2.213363839400643
>>> gmean([1, 2])
1.4142135623730951
>>> gmean([0, 5, 1000])
0.0
```

abydos.stats.**heronian_mean**(*nums*)

Return Heronian mean.

The Heronian mean is: $\frac{\sum_{i,j} \sqrt{x_i \cdot x_j}}{|nums| \cdot \frac{|nums|+1}{2}}$ for $j \geq i$

Cf. https://en.wikipedia.org/wiki/Heronian_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The Heronian mean of nums

**Return type** float

```
>>> heronian_mean([1, 2, 3, 4])
2.3888282852609093
>>> heronian_mean([1, 2])
1.4714045207910316
>>> heronian_mean([0, 5, 1000])
179.28511301977582
```

abydos.stats.**hmean**(*nums*)

Return harmonic mean.

The harmonic mean is defined as: $\frac{|nums|}{\sum_i \frac{1}{nums_i}}$

Following the behavior of Wolfram|Alpha: - If one of the values in nums is 0, return 0. - If more than one value in nums is 0, return NaN.

Cf. https://en.wikipedia.org/wiki/Harmonic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The harmonic mean of nums

**Return type** float

```
>>> hmean([1, 2, 3, 4])
1.9200000000000004
>>> hmean([1, 2])
1.3333333333333333
>>> hmean([0, 5, 1000])
0
```

abydos.stats.**hoelder_mean**(*nums*, *exp=2*)

Return Hölder (power/generalized) mean.

The Hölder mean is defined as: $\sqrt[p]{\frac{1}{|nums|} \cdot \sum_i x_i^p}$ for $p \neq 0$, and the geometric mean for $p = 0$

Cf. https://en.wikipedia.org/wiki/Generalized_mean

**Parameters**

- **nums** (*list*) – A series of numbers

- **exp** (*numeric*) – The exponent of the Hölder mean

**Returns** The Hölder mean of nums for the given exponent

**Return type** float

```
>>> hoelder_mean([1, 2, 3, 4])
2.7386127875258306
>>> hoelder_mean([1, 2])
1.5811388300841898
>>> hoelder_mean([0, 5, 1000])
577.3574860228857
```

abydos.stats.**imean**(*nums*)

Return identric (exponential) mean.

The identric mean of two numbers x and y is: x if x = y otherwise $\frac{1}{e} \sqrt[x-y]{\frac{x^x}{y^y}}$

Cf. https://en.wikipedia.org/wiki/Identric_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The identric mean of nums

**Return type** float

```
>>> imean([1, 2])
1.4715177646857693
>>> imean([1, 0])
nan
>>> imean([2, 4])
2.9430355293715387
```

abydos.stats.**lehmer_mean**(*nums*, *exp=2*)

Return Lehmer mean.

The Lehmer mean is: $\frac{\sum_i x_i^p}{\sum_i x_i^{(p-1)}}$

Cf. https://en.wikipedia.org/wiki/Lehmer_mean

**Parameters**

- **nums** (*list*) – A series of numbers
- **exp** (*numeric*) – The exponent of the Lehmer mean

**Returns** The Lehmer mean of nums for the given exponent

**Return type** float

```
>>> lehmer_mean([1, 2, 3, 4])
3.0
>>> lehmer_mean([1, 2])
1.6666666666666667
>>> lehmer_mean([0, 5, 1000])
995.0497512437811
```

abydos.stats.**lmean**(*nums*)

Return logarithmic mean.

The logarithmic mean of an arbitrarily long series is defined by http://www.survo.fi/papers/logmean.pdf as: $L(x_1, x_2, ..., x_n) = (n-1)! \sum_{i=1}^{n} \frac{x_i}{\prod_{\substack{j=1 \\ j \neq i}}^{n} ln \frac{x_i}{x_j}}$

Cf. https://en.wikipedia.org/wiki/Logarithmic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The logarithmic mean of nums

**Return type** float

```
>>> lmean([1, 2, 3, 4])
2.2724242417489258
>>> lmean([1, 2])
1.4426950408889634
```

abydos.stats.**median**(*nums*)

Return median.

With numbers sorted by value, the median is the middle value (if there is an odd number of values) or the arithmetic mean of the two middle values (if there is an even number of values).

Cf. https://en.wikipedia.org/wiki/Median

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The median of nums

**Return type** int or float

```
>>> median([1, 2, 3])
2
>>> median([1, 2, 3, 4])
2.5
>>> median([1, 2, 2, 4])
2
```

abydos.stats.**midrange**(*nums*)

Return midrange.

The midrange is the arithmetic mean of the maximum & minimum of a series.

Cf. https://en.wikipedia.org/wiki/Midrange

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The midrange of nums

**Return type** float

```
>>> midrange([1, 2, 3])
2.0
>>> midrange([1, 2, 2, 3])
2.0
>>> midrange([1, 2, 1000, 3])
500.5
```

abydos.stats.**mode**(*nums*)

Return the mode.

The mode of a series is the most common element of that series

Cf. https://en.wikipedia.org/wiki/Mode_(statistics)

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The mode of nums

**Return type** float

```
>>> mode([1, 2, 2, 3])
2
```

abydos.stats.**qmean**(*nums*)

Return quadratic mean.

The quadratic mean of precision and recall is defined as: $\sqrt{\sum_i \frac{num_i^2}{|nums|}}$

Cf. https://en.wikipedia.org/wiki/Quadratic_mean

**Parameters** **nums** (*list*) – A series of numbers

**Returns** The quadratic mean of nums

**Return type** float

```
>>> qmean([1, 2, 3, 4])
2.7386127875258306
>>> qmean([1, 2])
1.5811388300841898
>>> qmean([0, 5, 1000])
577.3574860228857
```

abydos.stats.**seiffert_mean**(*nums*)

Return Seiffert's mean.

Seiffert's mean of two numbers x and y is: $\frac{x-y}{4 \cdot arctan\sqrt{\frac{x}{y}}-\pi}$

Cf. http://www.helsinki.fi/~hasto/pp/miaPreprint.pdf

**Parameters** **nums** (*list*) – A series of numbers

**Returns** Sieffert's mean of nums

**Return type** float

```
>>> seiffert_mean([1, 2])
1.4712939827611637
>>> seiffert_mean([1, 0])
0.3183098861837907
>>> seiffert_mean([2, 4])
2.9425879655223275
>>> seiffert_mean([2, 1000])
336.84053300118825
```

abydos.stats.**std**(*nums*, *mean_func=<function amean>*, *ddof=0*)

Return the standard deviation.

**Parameters**

- **nums** (*list*) – A series of numbers
- **mean_func** (*function*) – A mean function (amean by default)
- **ddof** (*int*) – The degrees of freedom (0 by default)

**Returns** The standard deviation of the values in the series

**Return type** float

```
>>> std([1, 1, 1, 1])
0.0
>>> round(std([1, 2, 3, 4]), 12)
1.11803398875
>>> round(std([1, 2, 3, 4], ddof=1), 12)
1.290994448736
```

abydos.stats.**var**(*nums*, *mean_func=<function amean>*, *ddof=0*)

Calculate the variance.

**Parameters**

- **nums** (*list*) – A series of numbers
- **mean_func** (*function*) – A mean function (amean by default)
- **ddof** (*int*) – The degrees of freedom (0 by default)

**Returns** The variance of the values in the series

**Return type** float

```
>>> var([1, 1, 1, 1])
0.0
>>> var([1, 2, 3, 4])
1.25
>>> round(var([1, 2, 3, 4], ddof=1), 12)
1.666666666667
```

#### 3.1.1.11 abydos.stemmer module

abydos.stemmer.

The stemmer module defines word stemmers including:

- the Lovins stemmer
- the Porter and Porter2 (Snowball English) stemmers
- Snowball stemmers for German, Dutch, Norwegian, Swedish, and Danish
- CLEF German, German plus, and Swedish stemmers
- Caumanns German stemmer
- UEA-Lite Stemmer
- Paice-Husk Stemmer
- Schinke Latin stemmer
- S stemmer

abydos.stemmer.**caumanns**(*word*)

Return Caumanns German stem.

Jörg Caumanns' stemmer is described in his article in [Cau99].

This implementation is based on the GermanStemFilter described at [Lan13].

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> caumanns('lesen')
'les'
>>> caumanns('graues')
'grau'
>>> caumanns('buchstabieren')
'buchstabier'
```

abydos.stemmer.**clef_german**(*word*)
Return CLEF German stem.

The CLEF German stemmer is defined at [Sav05].

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> clef_german('lesen')
'lese'
>>> clef_german('graues')
'grau'
>>> clef_german('buchstabieren')
'buchstabier'
```

abydos.stemmer.**clef_german_plus**(*word*)
Return 'CLEF German stemmer plus' stem.

The CLEF German stemmer plus is defined at [Sav05].

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> clef_german_plus('lesen')
'les'
>>> clef_german_plus('graues')
'grau'
>>> clef_german_plus('buchstabieren')
'buchstabi'
```

abydos.stemmer.**clef_swedish**(*word*)
Return CLEF Swedish stem.

The CLEF Swedish stemmer is defined at [Sav05].

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> clef_swedish('undervisa')
'undervis'
>>> clef_swedish('suspension')
'suspensio'
```

(continues on next page)

(continued from previous page)

```
>>> clef_swedish('visshet')
'viss'
```

abydos.stemmer.**lovins**(*word*)

Return Lovins stem.

Lovins stemmer

The Lovins stemmer is described in Julie Beth Lovins's article [Lov68].

**Parameters** **word** (*str*) – the word to stem

**Returns** word stem

**Return type** str

```
>>> lovins('reading')
'read'
>>> lovins('suspension')
'suspens'
>>> lovins('elusiveness')
'elus'
```

abydos.stemmer.**paice_husk**(*word*)

Return Paice-Husk stem.

Implementation of the Paice-Husk Stemmer, also known as the Lancaster Stemmer, developed by Chris Paice, with the assistance of Gareth Husk

This is based on the algorithm's description in [Pai90].

**Parameters** **word** (*str*) – the word to stem

**Returns** the stemmed word

**Return type** str

```
>>> paice_husk('assumption')
'assum'
>>> paice_husk('verifiable')
'ver'
>>> paice_husk('fancies')
'fant'
>>> paice_husk('fanciful')
'fancy'
>>> paice_husk('torment')
'tor'
```

abydos.stemmer.**porter**(*word*, *early_english=False*)

Return Porter stem.

The Porter stemmer is described in [Por80].

**Parameters**

- **word** (*str*) – the word to calculate the stem of
- **early_english** (*bool*) – set to True in order to remove -eth & -est (2nd & 3rd person singular verbal agreement suffixes)

**Returns** word stem

**Return type** str

```
>>> porter('reading')
'read'
>>> porter('suspension')
'suspens'
>>> porter('elusiveness')
'elus'
```

```
>>> porter('eateth', early_english=True)
'eat'
```

abydos.stemmer.**porter2**(*word*, *early_english=False*)

Return the Porter2 (Snowball English) stem.

The Porter2 (Snowball English) stemmer is defined in [Por02].

**Parameters**

- **word** (*str*) – the word to calculate the stem of
- **early_english** (*bool*) – set to True in order to remove -eth & -est (2nd & 3rd person singular verbal agreement suffixes)

**Returns** word stem

**Return type** str

```
>>> porter2('reading')
'read'
>>> porter2('suspension')
'suspens'
>>> porter2('elusiveness')
'elus'
```

```
>>> porter2('eateth', early_english=True)
'eat'
```

abydos.stemmer.**s_stemmer**(*word*)

Return the S-stemmed form of a word.

The S stemmer is defined in [Har91].

**Parameters** **word** (*str*) – the word to stem

**Returns** the stemmed word

**Return type** str

```
>>> s_stemmer('summaries')
'summary'
>>> s_stemmer('summary')
'summary'
>>> s_stemmer('towers')
'tower'
>>> s_stemmer('reading')
'reading'
>>> s_stemmer('census')
'census'
```

abydos.stemmer.**sb_danish**(*word*)

Return Snowball Danish stem.

The Snowball Danish stemmer is defined at: http://snowball.tartarus.org/algorithms/danish/stemmer.html

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> sb_danish('underviser')
'undervis'
>>> sb_danish('suspension')
'suspension'
>>> sb_danish('sikkerhed')
'sikker'
```

abydos.stemmer.**sb_dutch**(*word*)

Return Snowball Dutch stem.

The Snowball Dutch stemmer is defined at: http://snowball.tartarus.org/algorithms/dutch/stemmer.html

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> sb_dutch('lezen')
'lez'
>>> sb_dutch('opschorting')
'opschort'
>>> sb_dutch('ongrijpbaarheid')
'ongrijp'
```

abydos.stemmer.**sb_german**(*word*, *alternate_vowels=False*)

Return Snowball German stem.

The Snowball German stemmer is defined at: http://snowball.tartarus.org/algorithms/german/stemmer.html

**Parameters**

- **word** (*str*) – the word to calculate the stem of
- **alternate_vowels** (*bool*) – composes ae as ä, oe as ö, and ue as ü before running the algorithm

**Returns** word stem

**Return type** str

```
>>> sb_german('lesen')
'les'
>>> sb_german('graues')
'grau'
>>> sb_german('buchstabieren')
'buchstabi'
```

abydos.stemmer.**sb_norwegian**(*word*)

Return Snowball Norwegian stem.

The Snowball Norwegian stemmer is defined at: http://snowball.tartarus.org/algorithms/norwegian/stemmer.html

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> sb_norwegian('lese')
'les'
>>> sb_norwegian('suspensjon')
'suspensjon'
>>> sb_norwegian('sikkerhet')
'sikker'
```

abydos.stemmer.**sb_swedish**(*word*)

Return Snowball Swedish stem.

The Snowball Swedish stemmer is defined at: http://snowball.tartarus.org/algorithms/swedish/stemmer.html

**Parameters** **word** (*str*) – the word to calculate the stem of

**Returns** word stem

**Return type** str

```
>>> sb_swedish('undervisa')
'undervis'
>>> sb_swedish('suspension')
'suspension'
>>> sb_swedish('visshet')
'viss'
```

abydos.stemmer.**schinke**(*word*)

Return the stem of a word according to the Schinke stemmer.

This is defined in [SGRW96].

**Parameters** **word** (*str*) – the word to stem

**Returns** a dict of the noun- and verb-stemmed word

**Return type** dict

```
>>> schinke('atque')
{'n': 'atque', 'v': 'atque'}
>>> schinke('census')
{'n': 'cens', 'v': 'censu'}
>>> schinke('virum')
{'n': 'uir', 'v': 'uiru'}
>>> schinke('populusque')
{'n': 'popul', 'v': 'populu'}
>>> schinke('senatus')
{'n': 'senat', 'v': 'senatu'}
```

abydos.stemmer.**uealite**(*word*, *max_word_length=20*, *max_acro_length=8*, *return_rule_no=False*, *var=None*)

Return UEA-Lite stem.

The UEA-Lite stemmer is discussed in [JS05].

This is chiefly based on the Java implementation of the algorithm, with variants based on the Perl implementation and Jason Adams' Ruby port.

Java version: [Chu] Perl version: [JS05] Ruby version: [Ada17]

**Parameters**

- **word** (*str*) – the word to calculate the stem of
- **max_word_length** (*int*) – the maximum word length allowed
- **max_acro_length** (*int*) – the maximum acryonym length allowed
- **return_rule_no** (*bool*) – if True, returns the stem along with rule number
- **var** (*str*) – variant to use (set to 'Adams' to use Jason Adams' rules, or 'Perl' to use the original Perl set of rules)

**Returns** word stem

**Return type** str or (str, int)

```
>>> uealite('readings')
'read'
>>> uealite('insulted')
'insult'
>>> uealite('cussed')
'cuss'
>>> uealite('fancies')
'fancy'
>>> uealite('eroded')
'erode'
```

#### 3.1.1.12 abydos.util module

abydos.util.

The util module defines various utility functions for other modules within Abydos, including:

- prod – computes the product of a collection of numbers (akin to sum)

abydos.util.**prod**(*nums*)

Return the product of nums.

The product is Π(nums).

Cf. https://en.wikipedia.org/wiki/Product_(mathematics)

**Parameters** **nums** – a collection (list, tuple, set, etc.) of numbers

**Returns** the product of a nums

**Return type** numeric

```
>>> prod([1,1,1,1])
1
>>> prod((2,4,8))
64
>>> prod({1,2,3,4})
24
>>> prod(2**i for i in range(5))
1024
```

CHAPTER 4

# Release History

## 4.1 0.3.0 (2018-10-15)

- Fixed implementation of Bag distance
- Updated BMPM to version 3.10
- Fixed Sphinx documentation on readthedocs.org
- Split string fingerprints out of clustering into their own module
- Added support for q-grams to skip-n characters
- **New phonetic algorithms:**
  - Statistics Canada
  - Lein
  - Roger Root
  - Oxford Name Compression Algorithm (ONCA)
  - Eudex phonetic hash
  - Haase Phonetik
  - Reth-Schek Phonetik
  - FONEM
  - Parmar-Kumbharana
  - Davidson's Consonant Code
  - SoundD
  - PSHP Soundex/Viewex Coding
  - an early version of Henry Code
  - Norphone

  - Dolby Code
  - Phonetic Spanish
  - Spanish Metaphone
  - MetaSoundex
  - SoundexBR
  - NRL English-to-phoneme
- **New string fingerprints:**
  - Cisłak & Grabowski's occurrence fingerprint
  - Cisłak & Grabowski's occurrence halved fingerprint
  - Cisłak & Grabowski's count fingerprint
  - Cisłak & Grabowski's position fingerprint
  - Synoname Toolcode
- **New distance measures:**
  - Minkowski distance & similarity
  - Manhattan distance & similarity
  - Euclidean distance & similarity
  - Chebyshev distance & similarity
  - Eudex distances
  - Sift4 distance
  - Baystat distance & similarity
  - Typo distance
  - Indel distance
  - Synoname
- **New stemmers:**
  - UEA-Lite Stemmer
  - Paice-Husk Stemmer
  - Schinke Latin stemmer
- Eliminated ._compat submodule in favor of six
- Transitioned from PEP8 to flake8, etc.
- Phonetic algorithms now consistently use max_length=-1 to indicate that there should be no length limit
- Added example notebooks in binder directory

## 4.2 0.2.0 (2015-05-27)

- Added Caumanns' German stemmer
- Added Lovins' English stemmer

- Updated Beider-Morse Phonetic Matching to 3.04
- Added Sphinx documentation

## 4.3 0.1.1 (2015-05-12)

- First Beta release to PyPI

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

## 5.1 Bibliography

### References

# Bibliography


[Ada17] Jason Adams. Ruby port of uealite stemmer. 2017. URL: https://github.com/ealdent/uea-stemmer.

[AmonME12] Iván Amón, Francisco Moreno, and Jaime Echeverri. Algoritmo fonético para detección de cadenas de texto duplicadas en el idioma español. *Revista Ingenierías Universidad de Medellín*, 11(20):127–138, June 2012. URL: http://www.scielo.org.co/scielo.php?pid=S1692-33242012000100011&script=sci_abstract&tlng=es.

[Axe09] Pål Axelsson. Sfinxbis. Technical Report, Swedish Alliance for Middleware Infrastructure, April 2009. URL: http://www.swami.se/download/18.248ad5af12aa81365338000091/SfinxBis.pdf.

[BCP02] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. String matching with metric trees using an approximate distance. In Alberto H. F. Laender and Arlindo L. Oliveira, editors, *SPIRE 2002: String Processing and Information Retrieval*, 271–283. Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. URL: http://www-db.disi.unibo.it/research/papers/SPIRE02.pdf, doi:10.1007/3-540-45735-6_24.

[BM08] Alexander Beider and Stephen P. Morse. Beider-morse phonetic matching: an alternative to soundex with fewer false hits. *International Review of Jewish Genealogy*, Summer 2008. URL: https://stevemorse.org/phonetics/bmpm.htm.

[BBL81] Gérard Bouchard, Patrick Brard, and Yolande Lavoie. Fonem: un code de transcription phonétique pour la reconstitution automatique des familles saguenayennes. *Population*, 1981. URL: http://www.persee.fr/doc/pop_0032-4663_1981_num_36_6_17248, doi:10.2307/1532326.

[Boy98] Carolyn B. Boyce. Information on the refined soundex algorithm. November 1998. URL: https://web.archive.org/web/20010513121003/http://www.bluepoof.com:80/Soundex/info2.html.

[Boy11] Leonid Boytsov. Indexing methods for approximate dictionary searching: comparative analysis. *Journal of Experimental Algorithmics*, 16:1.1:1.1–1.1:1.91, May 2011. doi:10.1145/1963190.1963191.

[BW94] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation, Palo Alto, May 1994. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html.

[Cau99] Jörg Caumanns. A fast and simple stemming algorithm for german words. Technical Report, Free University of Berlin, 1999. URL: https://refubium.fu-berlin.de/bitstream/handle/fub188/18405/tr-b-99-16.pdf.

[Chr11] Peter Christen. Febrl (freely extensible biomedical record linkage) – encode.py. December 2011. URL: https://sourceforge.net/projects/febrl/.

[Chu] Richard Churchill. Ueastem.java. URL: http://lemur.cmp.uea.ac.uk/Research/stemmer/UEAstem.java.

[CV05] Rudi Cilibrasi and Paul Michael Béla Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005. URL: https://ieeexplore.ieee.org/document/1412045, doi:10.1109/TIT.2005.844059.

[CislakG17] Aleksander Cisłak and Szymon Grabowski. Lightweight fingerprints for fast approximate keyword matching using bitwise operations. *CoRR*, 2017. URL: http://arxiv.org/abs/1711.08475, arXiv:1711.08475.

[Cod18a] Rosetta Code. Longest common subsequence. 2018. URL: http://rosettacode.org/wiki/Longest_common_subsequence#Dynamic_Programming_6.

[Cod18b] Rosetta Code. Run-length encoding. 2018. URL: https://rosettacode.org/wiki/Run-length_encoding#Python.

[C+69] Jay L. Cunningham and others. A study of the organization and search of bibliographic holdings in online computer systems: phase i. Technical Report, University of California, Berkleley, Institute of Library Research, mar 1969. URL: https://files.eric.ed.gov/fulltext/ED029679.pdf.

[Dal05] Andrew Dalke. Arithmetic coder (python recipe). 2005. URL: http://code.activestate.com/recipes/306626/.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, March 1964. doi:10.1145/363958.363994.

[Dav62] Leon Davidson. Retrieval of misspelled names in an airlines passenger record system. *Communications of the ACM*, 5(3):169–171, March 1962. doi:10.1145/366862.366913.

[dcm4che] dcm4che. DICOM toolkit & library: phonem.java. URL: https://github.com/dcm4che/dcm4che/blob/master/dcm4che-soundex/src/main/java/org/dcm4che3/soundex/Phonem.java.

[Dic45] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945. URL: https://www.jstor.org/stable/1932409, doi:10.2307/1932409.

[Dol70] James L. Dolby. An algorithm for variable-length proper-name compression. *Journal of Library Automation*, 3(4):257–275, 1970. URL: https://ejournals.bc.edu/ojs/index.php/ital/article/download/5259/4734, doi:10.6017/ital.v3i4.5259.

[EJMS76] Honey S. Elovitz, Rodney W. Johnson, Astrid McHugh, and John E. Shore. Automatic translation of english text to pphonetic by means of letter-to-sound rules. NRL Report 7948, document AD/A021 929, Naval Research Laboratory, Washington, D.C., 1976.

[FurnrohrRvR02] Michael Fürnrohr, Birgit Rimmelspacher, and Tilman von Roncador. Zusammenführung von datenbeständen ohne numerische identifikatoren: ein verfahren im rahmen der testuntersuchungen zu einem registergestützten zensus. *Bayern in Zahlen*, 2002(7):308–321, 2002. URL: https://www.statistik.bayern.de/medien/statistik/zensus/zusammenf__hrung_von_datenbest__nden_ohne_numerische_identifikatoren.pdf.

[Gad90] T. N. Gadd. Phonix: the algorithm. *Program*, 24(4):363–366, 1990. doi:10.1108/eb047069.

[Gar15] Lars Marius Garshol. Norphone comparator. 2015. URL: https://github.com/larsga/Duke/blob/master/duke-core/src/main/java/no/priv/garshol/duke/comparators/NorphoneComparator.java.

[GM88] Wilde Georg and Carsten Meyer. Nicht wörtlich genommen, 'schreibweisentolerante' suchroutine in dbase implementiert. *c't Magazin für Computer Technik*, pages 126–131, October 1988.

[Gil97] Leicester E. Gill. Ox-link: the oxford medical record linkage system. In *Record Linkage Techniques*. Washington, D.C., March 1997. Federal Committee on Statistical Methodology, Office of Management and Budget. URL: https://pdfs.semanticscholar.org/fff7/02a3322e05c282a84064ee085e589ef74584.pdf.

[Got82] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982. URL: http://www.sciencedirect.com/science/article/pii/0022283682903989, doi:10.1016/0022-2836(82)90398-9.

[Gro91] Aaron D. Gross. Getty synoname: the development of software for personal name pattern matching. In *Intelligent Text and Image Handling - Volume 2*, RIAO '91, 754–763. Paris, France, France, 1991. LE CENTRE DE HAUTES ETUDES INTERNATIONALES D'INFORMATIQUE DOCUMENTAIRE. URL: http://dl.acm.org/citation.cfm?id=3171004.3171021.

[HH00] Martin Haase and Kai Heitmann. Die erweiterte kölner phonetik. 2000.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950. URL: https://ieeexplore.ieee.org/document/6772729/, doi:10.1002/j.1538-7305.1950.tb00463.x.

[Har91] Donna Harman. How effective is stemming? *Journal of the American Society for Information Science*, 42(1):7–15, 1991. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.9828&rep=rep1&type=pdf, doi:10.1002/(SICI)1097-4571(199101)42:1%3C7::AID-ASI2%3E3.0.CO;2-P.

[Hen76] Louis Henry. Projet de transcription phonétique des noms de famille. *Annales de Démographie Historique*, 1976:201–214, 1976. URL: https://www.persee.fr/doc/adh_0066-2062_1976_num_1976_1_1313.

[HBD76] Theodore Hershberg, Alan Burstein, and Robert Dockhorn. Record linkage. *Historical Methods Newsletter*, 9(2–3):137–163, 1976. doi:10.1080/00182494.1976.10112639.

[HBD79] Theodore Hershberg, Alan Burstein, and Robert Dockhorn. Verkettung von daten: record linkage am beispiel des philadelphia social history project. In Wilhelm Heinz Schröder, editor, *Moderne Stadtgeschichte*, volume 8, pages 35–73. Klett-Cotta, 1979. URL: https://www.ssoar.info/ssoar/handle/document/32782.

[HM02] David Holmes and M. Catherine McCabe. Improving precision and recall for soundex retrieval. In *Proceedings. International Conference on Information Technology: Coding and Computing*, 22–26. April 2002. URL: https://ieeexplore.ieee.org/document/1000354/, doi:10.1109/ITCC.2002.1000354.

[Hoo02] David Hood. Cavesystem: phonetic matching algorithm. Technical Report CTP060902, University of Otago, Dunedin, New Zealand, September 2002. URL: https://caversham.otago.ac.nz/files/working/ctp060902.pdf.

[Hoo04] David Hood. Caverphone revisited. Technical Report CTP150804, University of Otago, Dunedin, New Zealand, December 2004. URL: https://caversham.otago.ac.nz/files/working/ctp150804.pdf.

[Jac01] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:241–272, 1901. URL: https://core.ac.uk/download/pdf/20654241.pdf.

[Jar89] Matthew A. Jaro. Advances in record linkage methodology as applied to the 1985 census of tampa florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989. doi:10.1080/01621459.1989.10478785.

[JS05] Marie-Claire Jenkins and Dan Smith. Conservative stemming for search and indexing. Technical Report, University of East-Anglia, Norwich, UK, 2005. URL: http://lemur.cmp.uea.ac.uk/Research/stemmer/stemmer25feb.pdf.

[JBG13] Sergio Jiminez, Claudio Becerra, and Alexander Gelbukh. SOFTCARDINALITY-CORE: improving text overlap with distributional measures for semantic textual similarity. In *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 1: Proceedings of the Main Conference and the Shared Task*, 194–201. Atlanta, GA, June 2013. Association for Computational Linguistics. URL: http://www.aclweb.org/anthology/S13-1028.

[Knu98] Donald E. Knuth. *The Art of Computer Programming: Volume 3, Sorting and Searching*, pages 394. Addison-Wesley, 1998.

[Kollar] Maroš Kollár. Text::phonetic::phonix. URL: https://github.com/maros/Text-Phonetic/blob/master/lib/Text/Phonetic/Phonix.pm.

[KV17] Kerrthi Koneru and Cihan Varol. Privacy preserving record linkage using metasoundex algorithm. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 443–447. December 2017. URL: https://ieeexplore.ieee.org/document/8260671/, doi:10.1109/ICMLA.2017.0-121.

[Kuh95] Michael Kuhn. Metaphone searches. November 1995. URL: http://aspell.net/metaphone/metaphone-kuhn.txt.

[LR96] Andrew J. Lait and Brian Randell. An assessment of name matching algorithms. Technical Report, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 1996. URL: http://homepages.cs.ncl.ac.uk/brian.randell/Genealogy/NameMatching.pdf.

[Lan13] Joerg Lang. Inner wworking of the german analyzer in lucene. November 2013. URL: http://www.evelix.ch/unternehmen/Blog/evelix/2013/11/11/inner-workings-of-the-german-analyzer-in-lucene.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966. URL: https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

[Lov68] Julie Beth Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11(1–2):22–31, June 1968. URL: http://www.mt-archive.info/MT-1968-Lovins.pdf.

[LA77] Billy T. Lynch and William L. Arends. Selection of a surname coding procedure for the srs record linkage system. Technical Report, Statistical Reporting Service, US Department of Agriculture, Washington, D.C., February 1977. URL: https://naldc.nal.usda.gov/download/27833/PDF.

[LegareLC72] Jacques Légaré, Yolande Lavoie, and Hubert Charbonneau. The early canadian population: problems in automatic record linkage. *Canadian Historical Review*, 53(4):427–442, December 1972. doi:10.3138/CHR-053-04-03.

[Mar15] Daniel Marcelino. Soundexbr: soundex (phonetic) algorithm for Brazilian portuguese. jul 2015. URL: https://github.com/danielmarcelino/SoundexBR.

[Mic99] Jörg Michael. Doppelgänger gesucht – ein programm für die kontextsensitive phonetische stringumwandlung. *c't Magazin für Computer Technik*, pages 252, 1999. URL: http://www.heise.de/ct/ftp/99/25/252/.

[Mic07] Jörg Michael. Phonet.c. August 2007. URL: ftp://ftp.heise.de/pub/ct/listings/phonet.zip.

[Min10] Hermann Minkowski. *Geometrie der Zahlen*. R. G. Teubner, Leipzig, 1910. URL: https://archive.org/stream/geometriederzahl00minkrich.

[Mok97] Gary Mokotoff. Soundexing and genealogy. 1997. URL: http://www.avotaynu.com/soundex.htm.

[ME96] Alvaro E. Monge and Charles P. Elkan. The field matching problem: algorithms and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, 267–270. AAAI Press, 1996. URL: http://dl.acm.org/citation.cfm?id=3001460.3001516.

[MKTM77] Gwendolyn B. Moore, John L. Kuhns, Jeffrey L. Trefftzs, and Christine A. Montgomery. *Accessing Individual Records from Personal Data Files Using Non-Unique Identifiers*. Number 500-2 in Special Publication. National Bureau of Standards, Washington, D.C., February 1977. URL: https://archive.org/details/accessingindivid00moor.

[MLM12] Alejandro Mosquera, Elena Lloret, and Paloma Moreda. Towards facilitating the accessibility of web 2.0~Texts through text normalisation. In *Proceedings of the LREC workshop: Natural Language Processing for Improving Textual Accessibility (NLP4ITA) ; Istanbul, Turkey.*, 9–14. 2012. URL: http://www.taln.upf.edu/pages/nlp4ita/pdfs/mosquera-nlp4ita2012.pdf.

[NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574, doi:10.1016/0022-2836(70)90057-4.

[Och57] Akira Ochiai. Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions-ii. *Bulletin of the Japanese Society of Scientific Fisheries*, 22(9):526–530, 1957. URL: https://www.jstage.jst.go.jp/article/suisan1932/22/9/22_9_526/_pdf/-char/en, doi:10.2331/suisan.22.526.

[Ope12] OpenRefine. Clustering in depth. 2012. URL: https://github.com/OpenRefine/OpenRefine/wiki/Clustering-In-Depth.

[Ots36] Yanosuke Otsuka. The faunal character of the Japanese pleistocene marine mollusca, as evidence of the climate having become colder during the pleistocene in Japan. *Bulletin of the Biogeographical Society of Japan*, 6(16):165–170, 1936.

[Pai90] Chris D. Paice. Another stemmer. In *ACM SIGIR Forum*, volume 24, 56–61. Fall 1990. URL: https://dl.acm.org/citation.cfm?id=101310, doi:10.1145/101306.101310.

[PK14] Vimal P. Parmar and CK Kumbharana. Study existing various phonetic algorithms and designing and development of a working model for the new developed algorithm and comparison by implementing ti with existing algorithm(s). *International Journal of Computer Applications*, 98(19):45–49, 2014. doi:10.5120/17295-7795.

[Pfe00] Ulrich Pfeifer. Wait 1.8 - soundex.c. 2000. URL: https://fastapi.metacpan.org/source/ULPFR/WAIT-1.800/soundex.c.

[Phi90a] Lawrence Philips. Hanging on the metaphone. *Computer Language Magazine*, 7(12):39–44, December 1990.

[Phi90b] Lawrence Philips. Metaphone. December 1990. URL: http://aspell.net/metaphone/metaphone.basic.

[Phi00] Lawrence Philips. The double metaphone search algorithm. *C/C++ Users Journal*, 18(6):38–43, June 2000.

[Pli18] Guillaume Plique. Talisman. 2018. URL: https://github.com/Yomguithereal/talisman.

[PZ84] Joseph J. Pollock and Antonio Zamora. Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4):358–368, April 1984. URL: http://dl.acm.org/citation.cfm?id=358048, doi:10.1145/358027.358048.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, July 1980. URL: http://snowball.tartarus.org/algorithms/porter/stemmer.html, doi:10.1108/eb046814.

[Por02] Martin F. Porter. The english (porter2) stemming algorithm. September 2002. URL: http://snowball.tartarus.org/algorithms/english/stemmer.html.

[Pos69] Hans Joachim Postel. Die kölner phonetik: ein verfahren zur identifizierung von personennamen auf der grundlage der gestaltanalyse. *IBM-Nachrichten*, 19:925–931, 1969.

[Pra15] Jörg Prante. Elasticsearch – haasephonetik.java. 2015. URL: https://github.com/elastic/elasticsearch/blob/master/plugins/analysis-phonetic/src/main/java/org/elasticsearch/index/analysis/phonetic/HaasePhonetik.java.

[RM88] John W. Ratcliff and David E. Metzener. Pattern matching: the gestalt approach. *Dr. Dobbs Journal*, 1988. URL: http://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970.

[Rep13] Dominic John Repici. Understanding classic soundex algorithms. 2013. URL: http://creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#SoundExAndCensus.

[RU09] Nicholas Ring and Alexandra L. Uitdenbogerd. Finding 'lucy in disguise': the misheard lyric matching problem. In Gary Geunbae Lee, Dawei Song, Chin-Yew Lin, Akiko Aizawa, Kazuko Kuriyama, Masaharu Yoshioka, and Tetsuya Sakai, editors, *Information Retrieval Technology*, 157–167. Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04769-5_14.

[RC67] A. H. Robinson and Colin Cherry. Results of a prototype television bandwidth compression scheme. In *Proceedings of the IEEE*, volume 55, 356–364. IEEE, 1967. doi:10.1109/PROC.1967.5493.

[Ruk18] Dorothea Rukasz. Pprl – privacy preserving record linkage. 2018. URL: https://github.com/cran/PPRL.

[Rus18] Robert C. Russell. Index. 1918. URL: https://patentimages.storage.googleapis.com/31/35/a1/f697a3ab85ced6/US1261167.pdf.

[Sav05] Jacques Savoy. IR multilingual resources at unine. 2005. URL: http://members.unine.ch/jacques.savoy/clef/.

[SGRW96] Robyn Schinke, Mark Greengrass, Alexander M. Robertson, and Peter Willett. A stemming algorithm for latin text databases. *Journal of Documentation*, 52(2):172–187, 1996. doi:10.1108/eb026966.

[SBB04] Rainer Schnell, Tobias Bachteler, and Stefan Bender. A toolbox for record linkage. *Australian Journal of Statistics*, 33(1-2):125–133, 2004. URL: https://pdfs.semanticscholar.org/2353/21c24ed0401cd05d7752c2c8a8da5b7a4dc0.pdf.

[SA10] Boumedyen A. N. Shannaq and Victor V. Alexandrov. Using product similarity for adding business. *Global Journal of Computer Science and Technology*, 10(12):2–8, October 2010. URL: https://www.sial.iias.spb.su/files/386-386-1-PB.pdf.

[Sim49] Edward H. Simpson. Measurement of diversity. *Nature*, 163:688, April 1949. URL: https://www.nature.com/articles/163688a0, doi:10.1038/163688a0.

[Sjoo09] Allan Sjöö. Swamisfinxbix. 2009. URL: http://www.swami.se/download/18.248ad5af12aa8136533800093/swamiSfinxBis.java.txt.

[SW81] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. URL: http://www.sciencedirect.com/science/article/pii/0022283681900875, doi:10.1016/0022-2836(81)90087-5.

[Son11] Wayne Song. Typo-distance. 2011. URL: https://github.com/wsong/Typo-Distance.

[Ste14] Kevin L. Stern. Dameraulevenshteinalgorithm.java. 2014. URL: https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/string/DamerauLevenshteinAlgorithm.java.

[Szy34] Dezydery Szymkiewicz. Une contribution statistique à la géographie floristique. *Acta Societatis Botanicorum Poloniae*, 11(3):249–265, 1934. URL: https://pbsociety.org.pl/journals/index.php/asbp/article/download/asbp.1934.012/6710, doi:10.5586/asbp.1934.012.

[Sorensen48] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Kongelige Danske Videnskabernes Selskab*, 5(4):1–34, 1948. URL: http://www.royalacademy.dk/Publications/High/295_S%C3%B8rensen,%20Thorvald.pdf.

[Taf70] Robert L. Taft. *Name Search Techniques*. Special report (New York State Identification and Intelligence System). Bureau of Systems Development, New York State Identification and Intelligence System, 1970.

[Tan58] T. T. Tanimoto. An elementary mathematical theory of classification and prediction. Technical Report, IBM, 1958.

[Tic] Ticki. Eudex: a blazingly fast phonetic reduction/hashing algorithm. URL: https://github.com/ticki/eudex.

[Tic16] Ticki. The eudex algorithm. December 2016. URL: http://ticki.github.io/blog/the-eudex-algorithm/.

[Tve77] Amos Tversky. Features of similarity. *Psychological Review*, 84(4):327–352, 1977. URL: http://www.cogsci.ucsd.edu/~coulson/203/tversky-features.pdf, doi:10.1037/0033-295x.84.4.327.

[VB12] Cihan Varol and Coskun Bayrak. Hybrid matching algorithm for personal names. *Journal of Data and Information Quality*, 3(4):8:1–8:18, September 2012. doi:10.1145/2348828.2348830.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974. doi:10.1145/321796.321811.

[Wik18] Wikibooks. Algorithm implementation/strings/longest common substring. 2018. URL: https://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Longest_common_substring#Python.

[Wil05] Martin Wilz. Aspekte der kodierung phonetischer Ähnlichkeiten in deutschen eigennamen. Master's thesis, Universität zu Köln, Köln, 2005. URL: http://ifl.phil-fak.uni-koeln.de/sites/linguistik/Phonetik/import/Phonetik_Files/Allgemeine_Dateien/Martin_Wilz.pdf.

[Win90] William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. Technical Report, U.S. Bureau of the Census, Statistical Research Division, Washington, D.C., 1990. URL: https://files.eric.ed.gov/fulltext/ED325505.pdf.

[WMJL94] William E. Winkler, George McLaughlin, Matthew A. Jaro, and Maureen Lync. Strcmp95.c. January 1994. URL: https://web.archive.org/web/20110629121242/http://www.census.gov/geo/msb/stand/strcmp.c.

[Zac14] Siderite Zackwehdex. Super fast and accurate string distance algorithm: sift4. 2014. URL: https://siderite.blogspot.com/2014/11/super-fast-and-accurate-string-distance.html.

[Zed15] Jesper Zedlitz. Phonet4java phonet.java. 2015. URL: https://github.com/jze/phonet4java/blob/master/src/main/java/de/zedlitz/phonet4java/Phonet.java.

[ZD96] Justin Zobel and Philip Dart. Phonetic string matching: lessons from information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, 166–172. New York, NY, USA, 1996. ACM. doi:10.1145/243199.243258.

[delPAngelesBailonM16] Mar'ıa del Pilar Angeles and Noemi Bailón-Miguel. Performance of spanish encoding functions during record linkage. In *DATA ANALYTICS 2016: The Fifth International Conference on Data Analysis*, 1–7. 2016. URL: https://core.ac.uk/download/pdf/55855695.pdf#page=14.

[delPAngelesEGGM15] Mar'ıa del Pilar Angeles, Adrián Espino-Gamez, and Jonathan Gil-Moncada. Comparison of a modified spanish phonetic, soundex, and phonex coding functions during data matching process. In *2015 International Conference on Informatics, Electronics Vision (ICIEV)*, 1–5. June 2015. URL: https://www.researchgate.net/publication/285589803_Comparison_of_a_Modified_Spanish_Phonetic_Soundex_and_Phonex_coding_functions_during_data_matching_process, doi:10.1109/ICIEV.2015.7334028.

[IBMCorporation73] IBM Corporation. *Alpha Search Inquiry System, General Information Manual*. White Plains, NY, 1973.

[JPGTrust91] The J. Paul Getty Trust. Synoname. 1991. URL: http://www.cs.cmu.edu/Groups/AI/areas/nlp/misc/synoname/synoname.zip.

[UnitedStates97] United States. *Using the Census Soundex*. Number 55 in General Information Leaflet. National Archives and Records Administration, Washington, D.C., 1997. URL: https://hdl.handle.net/2027/pur1.32754067050041.

[UnitedStates07] United States. Soundex system: the soundex indexing system. 2007. URL: https://www.archives.gov/research/census/soundex.html.

[vonRethS77] Hans-Peter von Reth and Hans-Jörg Schek. Eine zugriffsmethode für die phonetische Ähnlichkeitssuche. Technical Report 77.03.002, IBM Deutschland GmbH., 1977.

[65] . . \cyrchar \cyrk , \cyrchar \cyrk . *CCP*, 163(4):845–848, 1965. URL: http://mi.mathnet.ru/dan31411.

# Python Module Index

## a

# Index

## P



## Q



## R



## S

## T



## U



## V



## W