pyaln

Release 0.1.4

Marco Mariotti

Aug 06, 2021

CONTENTS:

The module pyaln is centered around the Alignment class, which provides access to convenient methods for reading, processing, and writing multiple sequence alignments.

- First time? After *Installation*, check the *Tutorial of pyaln*
- Find the documentation at *Alignment class*
- Check additional methods in submodule *Sequtils submodule*
- Here's a *Index* of all methods and objects.

CHAPTER

ONE

INSTALLATION

We recommend to use the conda package manager to install pyaln (check this page to install conda / miniconda). With conda installed, **run this in a terminal to install pyaln**:

```
conda install -c mmariotti pyaln
```

Alternatively, you can use pip:

```
pip install pyaln
```

To check that the pyaln installation was successful, run this:

```
python -c 'import pyaln'
```

CHAPTER

TWO

TUTORIAL OF PYALN

Welcome to the tutorial of pyaln. Here, you will learn how to use the Alignment class to read, write, process and characterize key features of multiple sequence alignments.

Contents of Tutorial

2.1 Introducing the Alignment class

This class is the core of pyaln, and represents a multiple alignment of homologous sequences. Sequences can be of any type (nucleotide, protein, or custom characters). Gaps **must** be encoded as dashes, i.e. `"-"`.

Each entry is uniquely identified by a *name*, with an optional *description*.

In many file formats (e.g. aligned fasta), an extensive *title* is associated to each sequence. This will include some form of identifier, plus other information such as gene/protein name, source etc. When reading alignment files, such *titles* are split into *name* (the first word, must be unique per alignment) and *description* (the remainder of the title).

The Alignment class comes into play when you have already aligned sequences. These may have been generated by any of the numerous aligner methods out there (for example: ClustalOmega, Mafft, T-coffee).

The rationale of pyaln Alignment is to provide a convenient and efficient interface for reading, writing, manipulating, and profiling alignments. Under the hood, pyaln employs Numpy and Pandas for computationally intensive tasks.

2.2 Tutorial set-up

For the examples below to work correctly, after *installing pyaln*, open python and run this before anything else:

```
>>> from pyaln import Alignment, pyaln_folder
```

2.3 Reading and writing alignments

Aligned sequences are **loaded** at the time at the creation of an Alignment object. In the next few examples, we load alignment files located in pyaln examples folder:

```
>>> filename=pyaln_folder + '/examples/fep15_protein.fa'
>>> fep_ali=Alignment(filename, fileformat='fasta')
```

Many file formats are supported, thanks to Bio.AlignIO (see a full list here). For a few common cases, extensions are recognized so it is not compulsory to specify format:

```
>>> fep_ali2=Alignment(pyaln_folder+'/examples/fep15_protein.stockholm')
>>> sbp2_ali=Alignment(pyaln_folder+'/examples/SBP2_protein.aln')
```

Alignments can also be instanced with a IO buffer rather than filename:

```
>>> fb=open(pyaln_folder+'/examples/SBP2_protein.aln')
>>> sbp2_ali2=Alignment(fb)
```

You may also initialize an alignment manually by providing aligned sequences and their identifiers. Alignment accepts any iterable of `(title, sequence)`:

```
>>> ex_ali=Alignment([ ('seq1 description1', 'ATTCG-'), ('seq2 desc2', '--TTGG'), ('seq3
↪', 'ATTCG-')])
```

Various options are available for **writing** alignments. If you print an Alignment, you will obtain a reduced representation, showing its number of sequences and length:

```
>>> print(fep_ali)
# Alignment of 6 sequences and 138 positions
MWLTLVALLALCATGRTAENLSESTTDQDKLVIARGKLVAPSVVGUSIKKMPELYNFLM...L Fep15_danio_rerio
MWAFLLLTLAFSATGMTEE-DVTDTAIEERPVIAKGILKAPSVVGUAIKKMPALYMFLM...L Fep15_S_salar
MWIFLLLTLAFSATGMTEE-NVTDTAIEERPVIAKGILKAPSVVGUAIKKMPELYTFLM...L Fep15_O_mykiss
MWAFLVLTFAVAA-GASET-VDNHTAAEEKLLIARGKLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_rubripes
MWALLVLTFAVTV-GASEE-VKNQTAAEEKLVIARGTLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_nigroviridis
MWAFVLIAFSV---GASDS--SNSTAE----VIARGKLMAPSVVGUAIKKLPELNRFLM...L Fep15_O_latipes
```

However, note that this representation may not include the full sequence, and omits descriptions.

On the other hand, *write()* method of Alignment offers a variety of output formats (again through Bio.AlignIO, see the full list here). The most common, *fasta*, includes sequence descriptions:

```
>>> ex_ali=Alignment([ ('seq1 description1', 'ATTCG-'), ('seq2 desc2', '--TTGG'), ('seq3
↪', 'ATTCG-')])
>>> print( ex_ali.write('fasta') )
>seq1 description1
```

(continues on next page)

(continued from previous page)

```
ATTCG-
>seq2 desc2
--TTGG
>seq3
ATTCG-
```

The *write* method also accepts a *to_file* argument to write directly to a file:

```
>>> ex_ali.write('clustal', to_file='ali_file.aln')
```

2.4 Indexing and transversing alignments

Alignments have two dimensions. By *length* of the alignment, we refer to its width, meaning the number of alignment columns (aka alignment positions). The other dimension is the *number of sequences* in the alignment (i.e. its height). These features can be inspected by the methods *ali_length()*, and *n_seqs()*, or at once through the property *shape()*, as illustrated below:

```
>>> ali=Alignment([ ('seq1 this is a seq', 'ATTCG-'), ('seq2 another seq', '--TTGG'), (
↪'seq3', 'ATTCG-')])
>>> print( [ali.ali_length(), ali.n_seqs(), ali.shape] )
[6, 3, (3, 6)]
```

You can slice portions of an Alignment (i.e. take on some sequences and/or some columns) by **indexing** it. The format is `Alignment[rows_selector, column_selector]`, where:

- The `rows_selector` can be an integer (i.e., the vertical position of the sequence in the alignment), or a slice thereof (e.g. `2:5`), or a list of sequence names.
- The `column_selector` is a integer index (i.e. the horizontal position in the alignment), or a slice thereof, or a list of (start, end) indices, or a Numpy boolean array.

Warning: As customary in python, in pyaln all positions are 0-based, and intervals are specified with their start included and their end excluded.

For example, we load this small alignment:

```
>>> ali=Alignment([ ('seq1 this is a seq', 'ATTCG-'), ('seq2 another seq', '--TTGG'), (
↪'seq3', 'ATTCG-')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
ATTCG- seq3
```

Let's get the alignment of first two sequences only:

```
>>> ali[:2,:]
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
```

We could have done the same by specifying sequences by name:

```
>>> ali[ ['seq1', 'seq2'], : ]
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
```

Now let's take the alignment without the first and last columns:

```
>>> ali[:,1:-1]
# Alignment of 3 sequences and 4 positions
TTCG seq1
-TTG seq2
TTCG seq3
```

We can take non-contigous alignment regions by indexing columns with a list of `(start, end)` elements. For example, to get the 1st, 2nd, and 6th position in a single step:

```
>>> ali[:, [(0,2), (5, 6)]]
# Alignment of 3 sequences and 3 positions
AT- seq1
--G seq2
AT- seq3
```

Indexing by row and column at once, to get the 1st character of all sequences except the last:

```
>>> ali[:-1, 0:1]
# Alignment of 2 sequences and 1 positions
A seq1
- seq2
```

Complex column selection can be performed by providing a Numpy boolean array. For example, take all columns except for the 3rd and 4th:

```
>>> import numpy as np
>>> colsel=np.array( [True, True, False, False, True, True] )
>>> ali[:, colsel]
# Alignment of 3 sequences and 4 positions
ATG- seq1
--GG seq2
ATG- seq3
```

To **iterate** through the sequences in the alignment (i.e. its rows), use a `for` loop. This will yield tuples like `(name, sequence)`. To get the description of a sequence, use *get_desc()*.

For example, here we print the name, sequence length, and description of each sequence (in the same order as they are found in the alignment):

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'), (
↪'seq3', 'ATTCG-')])
>>> for name, seq in ali:
...   print(  (name, len(seq),     ali.get_desc(name)) )
('seq1', 6, 'this is first')
('seq2', 6, 'this is 2nd')
('seq3', 6, '')
```

To iterate over alignment positions instead (i.e. its columns) use the *positions()* method.

For example, here we check at each position whether the two sequences ('seq1' and 'seq2') have the same character:

```
>>> for i in ali.positions():
...    print(  (i, ali.get_seq('seq1')[i]   ==  ali.get_seq('seq2')[i])  )
(0, False)
(1, False)
(2, True)
(3, False)
(4, True)
(5, False)
```

2.5 Working with alignment columns

You may want to determine the composition of each column, meaning the frequencies of observed characters at each specific position. Since alignment columns represent homologous positions in the aligned sequences and frequencies represent the conservation at those positions, this is referred to as *conservation map*.

Like some other methods of the Alignment class, *conservation_map()* returns a Pandas DataFrame:

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'), (
↪'seq3', 'ATTCG-')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
ATTCG- seq3


>>> ali.conservation_map()
          0         1    2         3    4         5
-  0.333333  0.333333  0.0  0.000000  0.0  0.666667
A  0.666667  0.000000  0.0  0.000000  0.0  0.000000
C  0.000000  0.000000  0.0  0.666667  0.0  0.000000
G  0.000000  0.000000  0.0  0.000000  1.0  0.333333
T  0.000000  0.666667  1.0  0.333333  0.0  0.000000
```

Pandas Series (basically the data type of each column of a DataFrame) may be used to index the columns of pyaln Alignment. This may be convenient, for example, to take all alignment columns at least one "T" character:

```
>>> ali[:,   ali.conservation_map().loc['T']>0 ]
# Alignment of 3 sequences and 3 positions
TTC seq1
-TT seq2
TTC seq3
```

A very common operation with alignments involves removing those columns featuring too many gaps. This is often referred to as **trimming alignments**, and it is achieved through the function *trim_gaps()*.

For example, let's remove all columns with at least 50% gaps:

```
>>> ali.trim_gaps(0.5)
# Alignment of 3 sequences and 5 positions
ATTCG seq1
--TTG seq2
ATTCG seq3
```

Another common operation is **alignment concatenation**: two or more alignments corresponding to different gene families, but coming from the same set of species, are combined into one. Visually, alignment concatenation corresponds to stacking one alignment next to the other horizontally. This is achieved in pyaln by adding two Alignment instances using with a + operator (or analogously, calling the *concatenate()* function).

```
>>> ali2=Alignment([ ('seq1', 'AAATAAAA'), ('seq2'  , '-AAGAAAG'), ('seq3', 'ACATAAAC')])
>>> ali + ali2
# Alignment of 3 sequences and 14 positions
ATTCG-AAATAAAA seq1
--TTGG-AAGAAAG seq2
ATTCG-ACATAAAC seq3
```

Note that if the two alignments being added do not have exaclty the same *names*, an error occurs.

Adding a string to an Alignment is equivalent to adding its content to each sequence of the alignment:

```
>>> ali + 'NNNN' + ali2
# Alignment of 3 sequences and 18 positions
ATTCG-NNNNAAATAAAA seq1
--TTGGNNNN-AAGAAAG seq2
ATTCG-NNNNACATAAAC seq3
```

2.6 Sequence identity

There are various methods implemented in pyaln to estimate the degree of similarity of sequences in the alignment. In general, they are based on **sequence identity**. At first glance, this is a very straightforward concept: the sequence identity of two sequences is the number of identical positions, divided by their length. In this example, 4/5 -> 80%

```
>>> from pyaln.sequtils import sequence_identity
>>> sequence_identity('ATGCA',
...                   'ATGCC')
0.8
```

However, when gaps come into the picture, things get a little more complicated, as you may choose to score them in a few different ways. Pyaln offers four options in this regard, each identified by a single letter `gaps` code:

1. `gaps='y'`: gaps are considered and considered mismatches. This is the **default** behaviour.
2. `gaps='n'`: gaps are ignored
3. `gaps='t'`: terminal gaps (those at the beginning or the end of sequences) are ignored. Others are considered as in `'y'`.
4. `gaps='a'`: gaps are considered as any other character; even gap-to-gap matches are scored as identities

These options can be provided to *sequence_identity()* and other pyaln methods. Let's see a few examples of their behavior:

```
>>> from pyaln.sequtils import sequence_identity
>>> seq1='--ATC-GGG-'
>>> seq2='AAATCGGGGC'
>>> seq3='--ACC-CCGC'
>>> ali=Alignment( [('seq1', seq1), ('seq2', seq2), ('seq3', seq3)] )
```

The first two sequences are identical, but *seq2* has three insertions (i.e. gapped regions) compared to *seq1*. Comparing them with `gaps='y'` will consider all positions (including gaps) as total sequence length, effectively scoring negatively gaps:

```
>>> sequence_identity(seq1, seq2, gaps='y')
0.6
```

On the other hand, if we ignore gaps with `gaps='n'`, we obtain 100% sequence identity:

```
>>> sequence_identity(seq1, seq2, gaps='n')
1.0
```

In certain applications, you may want to ignore terminal gaps with `gaps='t'`. In this case, this means that the *seq1* subsequence `ATC-GGG` is effectively compared to the corresponding region of *seq2*, resulting in 6/7 –> ~0.86

```
>>> sequence_identity(seq1, seq2, gaps='t')
0.8571428571428571
```

The option `gaps='a'` is not recommended for biological alignments. This behaves similarly to `gaps='y'`, but with an important difference. When comparing two sequences coming an alignment that contains many additional ones, it is possible that the two sequences both have a gap in one or more positions:

```
>>> print ( seq1+'\n'+seq3 )
--ATC-GGG-
--ACC-CCGC
```

If we compare them naively, counting all identical characters without differentiating gaps (i.e., the behavior of `gaps='a'`), we end up scoring shared gaps positively, with 6/10 matches:

```
>>> sequence_identity(seq1, seq3, gaps='a')
0.6
```

Shared gaps should be ignored in any pairwise comparison, which is the behavior followed under any other value of `gaps` (`'y'`, `'n'`, `'t'`):

```
>>> sequence_identity(seq1, seq3, gaps='y')   # 3/7
0.42857142857142855

>>> sequence_identity(seq1, seq3, gaps='n')   # 3/6
0.5

>>> sequence_identity(seq1, seq3, gaps='t')   # 3/6
0.5
```

The function *score_similarity()* allows to compute the **Average Sequence Identity (ASI)** of each sequence, when compared to the whole alignment. This is equivalent to calling the function *sequence_identity()* introduced above in all-against-all fashion (but it is implemented differently for better performance). This measure is instrumental estimate the overall similarity of sequence in the alignment.

```
>>> fep_ali=Alignment(pyaln_folder + '/examples/fep15_protein.fa', fileformat='fasta')
>>> fep_ali.score_similarity()
metrics                    ASI
Fep15_danio_rerio     0.777778
Fep15_S_salar         0.826334
Fep15_O_mykiss        0.822684
Fep15_T_rubripes      0.829599
Fep15_T_nigroviridis  0.815000
Fep15_O_latipes       0.767438
```

The *score_similarity()* method accepts the `gaps` parameter to define how to treat gaps. You may provide a single `gaps` argument, or provide multiple ones at once to assess how results would differ:

```
>>> fep_ali.score_similarity(gaps=['y', 'n', 't', 'a'])
gaps                          y         n         t         a
metrics                     ASI       ASI       ASI       ASI
Fep15_danio_rerio     0.777778  0.793051  0.777778  0.777778
Fep15_S_salar         0.826334  0.838283  0.826334  0.827295
Fep15_O_mykiss        0.822684  0.834522  0.822684  0.823671
Fep15_T_rubripes      0.829599  0.842566  0.829599  0.830918
Fep15_T_nigroviridis  0.815000  0.835351  0.815000  0.816425
Fep15_O_latipes       0.767438  0.805693  0.767438  0.769324
```

Besides ASI, this method may also return a variant called **Average Weighted Sequence Identity (AWSI)**, wherein the most conserved positions in the alignment are given higher weight. For details, see *score_similarity()*.

```
>>> fep_ali.score_similarity(metrics=['i', 'w'],  gaps='y')
metrics                    ASI      AWSI
Fep15_danio_rerio     0.777778  0.847123
Fep15_S_salar         0.826334  0.885040
Fep15_O_mykiss        0.822684  0.882183
Fep15_T_rubripes      0.829599  0.887255
Fep15_T_nigroviridis  0.815000  0.874389
Fep15_O_latipes       0.767438  0.834809
```

These sequence metrics may be employed to assess how some external sequences *fit* in a core alignment. This may be instrumental to check whether some candidate sequences appear to belong to a certain gene family. In the following example, we load an alignment containing the same sequences as *fep_ali* above, with the addition of an extra candidate sequence. We want to test whether this sequence resembles other sequences in a similar degree as they resemble each other.

```
>>> cand_ali=Alignment(pyaln_folder + '/examples/fep15_protein.with_candidate.fa',␣
↪fileformat='fasta')
>>> cand_ali
# Alignment of 7 sequences and 163 positions
MWLTLVALLALCATGRTAENLSESTTDQDKLVIARGKLVAPSVVGUSIKKMPELYNFLM...L Fep15_danio_rerio
MWAFLLLTLAFSATGMTEE-DVTDTAIEERPVIAKGILKAPSVVGUAIKKMPALYMFLM...L Fep15_S_salar
MWIFLLLTLAFSATGMTEE-NVTDTAIEERPVIAKGILKAPSVVGUAIKKMPELYTFLM...L Fep15_O_mykiss
MWAFLVLTFAVAA-GASET-VDNHTAAEEKLLIARGKLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_rubripes
MWALLVLTFAVTV-GASEE-VKNQTAAEEKLVIARGTLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_nigroviridis
MWAFVLIAFSV---GASDS--SNSTAE----VIARGKLMAPSVVGUAIKKLPELNRFLM...L Fep15_O_latipes
-------------------------------------QSCGGUQLNRLREVKAFVT...L Fep15_candidate
```

Let's see the ASI and AWSI metrics for the core alignment (all sequences except the last one):

```
>>> cand_ali[:-1,:].score_similarity( metrics='iw', gaps='yn' )
gaps                         y                   n
metrics                    ASI      AWSI       ASI      AWSI
Fep15_danio_rerio     0.777778  0.847123  0.793051  0.856044
Fep15_S_salar         0.826334  0.885040  0.838283  0.893412
Fep15_O_mykiss        0.822684  0.882183  0.834522  0.890497
Fep15_T_rubripes      0.829599  0.887255  0.842566  0.896094
Fep15_T_nigroviridis  0.815000  0.874389  0.835351  0.891288
Fep15_O_latipes       0.767438  0.834809  0.805693  0.860639
```

Now let's see the same metrics but comparing the candidate to the same set of sequences. This is achieved through the `targets` argument of *score_similarity()*:

```
>>> cand_ali[:-1,:].score_similarity( targets=cand_ali[ ['Fep15_candidate'] ,:], metrics=
↪'iw', gaps='yn' )
gaps                      y                   n
metrics                 ASI      AWSI       ASI      AWSI
Fep15_candidate  0.213043  0.282854  0.349844  0.362332
```

We can see that the metrics are well outside the range of the similarity metrics of the core alignments, indicating that the sequence does not fit in the family just as well. Indeed, this protein is from another family.

2.7 Biopython, Numpy, and Pandas

Sequences are stored in pyaln Alignment objects in form of built-in string types. This ensures the most common operations are as fast as possible. For certain procedures, however, an alternative representation is generated on the fly.

Specifically, pyaln transforms Alignment objects into MultipleSeqAlignments from Biopython AlignIO to access a variety of Input / Output capabilities.

On the other hand, fast vectorized operations on alignment columns are performed using alignment representations as Numpy array (one row per sequence, one column per alignment position). A similar representation, slightly slower but more versatile, is also employed: the Pandas DataFrame.

Conversions back and forth from these alternative representations of alignments automatically occur under the hood of pyaln when they are convenient for efficient computation. If you wish to build on top of pyaln and may find these representation useful, then check the documentation of these methods:

- *to_biopython()*
- *to_numpy()*
- *to_pandas()*
- *from_numpy()*

CHAPTER

THREE

ALIGNMENT CLASS

class pyaln.**Alignment**(*file_or_iter=None*, *fileformat=None*)

Represents a multiple sequence alignment.

Alignment can contain sequences of any type (nucleotide, protein, or custom). Gaps **must** be encoded as -.

Each entry is uniquely identified by a *name*, with an optional *description*. When reading alignment files, sequence *titles* are split into the name (the first word) and descriptions (the remainder of the title).

An Alignment can be instanced with a filename (or file buffer), or from any iterable of *[title, sequence]*. A variety of file formats are supported, through Bio.AlignIO (see a full list). When a filename or buffer is provided but not the file format, Alignment tries to guess it from the extension.

You can take portions of an Alignment (i.e. take some sequences and/or some columns) by **indexing** it.

The format is *Alignment[rows_selector, column_selector]*, where:

- The *rows_selector* can be an integer (i.e., the vertical position of the sequence in the alignment), or a slice thereof (e.g. *2:5*), or a list of sequence names.
- The *column_selector* is a integer index (i.e. the horizontal position in the alignment), or a slice thereof, or a boolean Numpy array / Pandas Series. See examples below.

Iterating over an Alignment will yield tuples like *(name, sequence)*. To get the description of a sequence, use *Alignment.get_desc(name)*.

Parameters

- **file_or_iter** (*str | TextIO | iterable*) – Filename to sequence file to be loaded, or TextIO buffer already opened on it, or iterable of [title, seq] objects.
- **fileformat** (*str, optional*) – When a filename or TextIO is provided, specifies the file format (e.g. fasta, clustal, stockholm ..)

Examples

```
>>> ali=Alignment(pyaln_folder+'/examples/fep15_protein.fa')
```

Default representation (note, it does not contain descriptions):

```
>>> ali
# Alignment of 6 sequences and 138 positions
MWLTLVALLALCATGRTAENLSESTTDQDKLVIARGKLVAPSVVGUSIKKMPELYNFLM...L Fep15_danio_rerio
MWAFLLLTLAFSATGMTEE-DVTDTAIEERPVIAKGILKAPSVVGUAIKKMPALYMFLM...L Fep15_S_salar
MWIFLLLTLAFSATGMTEE-NVTDTAIEERPVIAKGILKAPSVVGUAIKKMPELYTFLM...L Fep15_O_mykiss
MWAFLVLTFAVAA-GASET-VDNHTAAEEKLLIARGKLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_rubripes
```

(continues on next page)

(continued from previous page)

```
MWALLVLTFAVTV-GASEE-VKNQTAAEEKLVIARGTLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_nigroviridis
MWAFVLIAFSV---GASDS--SNSTAE----VIARGKLMAPSVVGUAIKKLPELNRFLM...L Fep15_O_latipes
```

Many file formats are supported:

```
>>> ali=Alignment(pyaln_folder+'/examples/fep15_protein.stockholm', fileformat=
↪'stockholm')
>>> ali
# Alignment of 6 sequences and 138 positions
MWLTLVALLALCATGRTAENLSESTTDQDKLVIARGKLVAPSVVGUSIKKMPELYNFLM...L Fep15_danio_rerio
MWAFLLLTLAFSATGMTEE-DVTDTAIEERPVIAKGILKAPSVVGUAIKKMPALYMFLM...L Fep15_S_salar
MWIFLLLTLAFSATGMTEE-NVTDTAIEERPVIAKGILKAPSVVGUAIKKMPELYTFLM...L Fep15_O_mykiss
MWAFLVLTFAVAA-GASET-VDNHTAAEEKLLIARGKLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_rubripes
MWALLVLTFAVTV-GASEE-VKNQTAAEEKLVIARGTLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_nigroviridis
MWAFVLIAFSV---GASDS--SNSTAE----VIARGKLMAPSVVGUAIKKLPELNRFLM...L Fep15_O_latipes
```

Initializing from iterable (in this case a list):

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
ATTCG- seq3
```

To visualize sequence descriptions, use the fasta format:

```
>>> ali=Alignment([ ('seq1 this is a seq', 'ATTCG-'), ('seq2 another seq', '--TTGG
↪'), ('seq3', 'ATTCG-')])
>>> print(ali.fasta())
>seq1 this is a seq
ATTCG-
>seq2 another seq
--TTGG
>seq3
ATTCG-
```

Indexing an alignment

Get alignment of first two sequences only:

```
>>> ali[:2,:]
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
```

Trim off the first and last alignment columns:

```
>>> ali[:,1:-1]
# Alignment of 3 sequences and 4 positions
TTCG seq1
-TTG seq2
TTCG seq3
```

Get subalignment of two sequences, by their name:

```
>>> ali[ ['seq1', 'seq3'], : ]
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
ATTCG- seq3
```

Index columns by providing list of (start, end) elements:

```
>>> ali[:, [(0,2), (5, 6)]]
# Alignment of 3 sequences and 3 positions
AT- seq1
--G seq2
AT- seq3
```

Iterating over an alignment:

```
>>> [(name, len(seq))   for name, seq in ali]
[('seq1', 6), ('seq2', 6), ('seq3', 6)]
```

add_seq(*title*, *sequence*, *desc=None*, *index=None*)

Add a sequence to the alignment.

The sequence name (i.e., its unique id) is derived from title, taking its first word. The rest of title is taken as sequence description. By default, the sequence is added to the bottom of the alignment.

Parameters

- **title** (*str*) – Sequence title, from which name and description are derived
- **sequence** (*str*) – Actual sequence, with gaps encoded as "-" characters
- **desc** (*str, optional*) – The description can be directly provided here. If so, title is taken as name instead
- **index** (*int, optional*) – The position at which the sequence is inserted. If not provided, it goes last

Returns None

Return type None

Examples

```
>>> ali=Alignment()
>>> ali.add_seq('seq1 custom nt seq', 'ATTCG-')
>>> ali.add_seq('seq2 another seq',   '--TTGG')
>>> print(ali.fasta())
>seq1 custom nt seq
ATTCG-
>seq2 another seq
--TTGG
```

```
>>> ali.add_seq('seq3', 'ATT---', desc='some desc')
>>> ali.add_seq('seq4', 'ATTGG-', index=0)
>>> print(ali.fasta())
```

(continues on next page)

(continued from previous page)

```
>seq4
ATTGG-
>seq1 custom nt seq
ATTCG-
>seq2 another seq
--TTGG
>seq3 some desc
ATT---
```

ali_length()

Returns the number of columns in the alignment (i.e., its width)

Returns The number of columns in the alignment

Return type int

Examples

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali.ali_length()
6
```

Warning: For best performance, the Alignment class does not check that all sequences have the same length. This method simply returns the length of the first sequence. To check for homogenous sequence length, see *same_length()*

See also:

same_length check that all sequences are truly aligned, i.e. have the same length

column_weights(*method='m'*)

Computes weights indicating the relative importance of the different alignment columns, based on their level of conservation.

Parameters **method** (*str*) – One of these arguments: - 'm' : maximum frequency of non-gap character in self - 'i' : information content, i.e. 2- sum(p*log2(p)) where p is frequency of non-gap characters - 'q' : quadratic sum, i.e. sum(p*p) where p is frequency of non-gap characters

Returns Numpy array of float numbers, of the same length as the alignment (n. of columns) indicating the different weights

Return type np.ndarray

See also:

score_similarity

concatenate(*other*)

Concatenate two alignments, i.e., add their sequences one next to the other

The two alignments must have the same names in the same order or an AlignmentError exception is raised. The sequence descriptions in returned alignment are taken from self

Parameters **other** (*Alignment or str*) – alignment that will be concatenated to the right of the self in the returned Alignment. If a string is provided, this same sequence is added to each sequence in self

Returns alignment with same names as inputs, and sequences resulting from their concatenation

Return type *Alignment*

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', 'ATTCG-')])
>>> ali2=Alignment([ ('seq1 first', 'TTGC-TAG'), ('seq2 this is 2nd'  , '-
↪ATGGGGC'), ('seq3', 'AATCGGCC')])
>>> ali.concatenate(ali2)
# Alignment of 3 sequences and 14 positions
ATTCG-TTGC-TAG seq1
--TTGG-ATGGGGC seq2
ATTCG-AATCGGCC seq3
```

Note that descriptions in the second alignment are ignored:

```
>>> ali3= Alignment([ ('seq1 this desc is ignored', 'TTGC-TAG'), ('seq2'  , '-
↪ATGGGGC'), ('seq3 this also', 'AATCGGCC')])
>>> print( ali.concatenate(ali3).fasta() )
>seq1 first
ATTCG-TTGC-TAG
>seq2 this is 2nd
--TTGG-ATGGGGC
>seq3
ATTCG-AATCGGCC
```

consensus(*ignore_gaps=None*)

Compute the consensus sequence, taking the most represented character for each column

Parameters **ignore_gaps** (*float, optional*) – By default, gaps are treated as any other character, so that they are returned for columns in which they are the most common character. If you provide ignore_gaps with a float from 0.0 to 1.0, gaps are not present on the output except for columns with a frequency equal or greater than the value provided. For example, a value of 1.0 implies gaps are included only if a column is entirely made of gaps

Returns The consensus sequence

Return type str

Examples

```
>>> ali= Alignment([ ('seq1', 'ATTCG-'), ('seq2'  , '-TTCGT'), ('seq3', 'ACGCG-
↪'),  ('seq4', 'CTTGGT'), ('seq5', '-TGCT-'), ('seq6', '-TGGG-')])
>>> ali
# Alignment of 6 sequences and 6 positions
ATTCG- seq1
-TTCGT seq2
ACGCG- seq3
```

(continues on next page)

(continued from previous page)

```
CTTGGT seq4
-TGCT- seq5
-TGGG- seq6
```

```
>>> ali.conservation_map()
          0         1    2         3         4         5
-  0.500000  0.000000  0.0  0.000000  0.000000  0.666667
A  0.333333  0.000000  0.0  0.000000  0.000000  0.000000
C  0.166667  0.166667  0.0  0.666667  0.000000  0.000000
G  0.000000  0.000000  0.5  0.333333  0.833333  0.000000
T  0.000000  0.833333  0.5  0.000000  0.166667  0.333333
```

```
>>> ali.consensus()
'-TGCG-'
```

```
>>> ali.consensus(0.6)
'ATGCG-'
```

conservation_map(*counts=None*)

Computes the frequency of characters (nucleotides/amino acids) at each column of the alignment

Gaps are considered as any other character during computation. The returned object reports frequencies at each position, for all characters which are observed at least once in the alignment. This may not correspond to the full nucleotide or protein alphabet, if some characters are not present in the alignment.

Returns The returned dataframe has one row per observed character (i.e., nucleotide / amino acid) and one column per alignment position. Each value is a float ranging from 0 to 1 representing the frequency of that character in that alignment column.

Return type pd.DataFrame

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', '--TT--')])
>>> ali.conservation_map()
          0         1    2         3         4         5
-  0.666667  0.666667  0.0  0.000000  0.333333  0.666667
A  0.333333  0.000000  0.0  0.000000  0.000000  0.000000
C  0.000000  0.000000  0.0  0.333333  0.000000  0.000000
G  0.000000  0.000000  0.0  0.000000  0.666667  0.333333
T  0.000000  0.333333  1.0  0.666667  0.000000  0.000000
```

Warning: This function is cached for best performance. Thus, do not directly modify the returned object. The hash key for caching is derived from sequences only: names are not considered.

copy()

Returns a copy of the alignment

Returns copy of the self alignment

Return type *Alignment*

descriptions()
Returns the descriptions for all sequences in the alignment as list

Returns An ordered list of descriptions for each sequence in the alignment

Return type list of str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.descriptions()
['this is first', 'this is 2nd', '']
```

See also:

names get all sequence names (their unique identifiers, without description)

titles get all sequence titles (name and description separated by space)

fasta(*nchar=60*)
Returns the alignment in (aligned) fasta format

Parameters **nchar** (*int, default=60*) – The number of characters per line for sequences

Returns A multiline string with the alignment in fasta format, including sequence descriptions

Return type str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> print(ali.fasta())
>seq1 this is first
ATTCG-
>seq2 this is 2nd
--TTGG
>seq3
ATTCG-
```

See also:

write generic function supporting many output formats

classmethod from_numpy(*nparray*, *names*, *descriptions=None*)
Class method to instance an Alignment object from a numpy array.

Parameters

- **nparray** (*np.ndarray*) – analogous to object returned by Alignment.to_numpy(), it must have one row per sequence, and one column per alignment position. Its dtype must be is np.str_
- **names** (*list of str*) – ordered list of sequence names (i.e. identifiers)

- **descriptions** (*list of str, optional*) – ordered list of sequence description. If not provided, all descriptions are set to ''

Returns new alignment object

Return type *Alignment*

See also:

to_numpy

get_desc(*name*)

Returns the description for a sequence entry

If no sequence with that name is present in the alignment, an AlignmentError exception is raised.

Parameters **name** (*str*) – The name (i.e. identifier) of the sequence

Returns The description stored for the sequence

Return type str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.get_desc('seq1')
'this is first'
>>> ali.get_desc('seq3')
''
```

See also:

set_desc

get_seq(*name*)

Returns the sequence with the requested name

If no sequence with that name is present in the alignment, an AlignmentError exception is raised.

Parameters **name** (*str*) – The name (i.e. identifier) of the sequence requested

Returns The sequence requested, including any gaps it may contain

Return type str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.get_seq('seq1')
'ATTCG-'
>>> ali.get_seq('seq2')
'--TTGG'
```

See also:

set_seq

has_name(*name*)

Checks whether the alignment contains a sequence with the name provided

Parameters **name** (*str*) – The name (i.e. identifier) searched in the alignment.

Returns A bool indicating whether the name is present

Return type bool

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.has_name('seq1')
True
>>> ali.has_name('seq2 this is 2nd')  # note: the name would be 'seq2' only
False
```

n_seqs()

Returns the number of sequences in the alignment (i.e. the number of rows, or alignment height)

Returns Number of sequences in the alignment

Return type int

Examples

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali.n_seqs()
3
```

See also:

ali_length length of the alignment (i.e. number of columns)

names()

Returns a list of all sequence names in the alignment

The names returned do not include the sequence descriptions.

Returns An ordered list of sequence names (identifiers) in the alignment

Return type list of str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.names()
['seq1', 'seq2', 'seq3']
```

See also:

titles get all sequence titles, including their name and description

position_in_ali(*name*, *pos_in_seq*)

Maps an position in a certain sequence (without counting gaps) to its corresponding position in the alignment

If the requested position is invalid, raise an IndexError

Parameters

- **name** (*str*) – the name of the sequence
- **pos_in_seq** (*int*) – 0-based sequence position, i.e. the index mapping to the requested sequence without gaps

Returns 0-based alignment position , i.e. the column index where the requested sequence position is found

Return type int

Examples

```
>>> ali= Alignment([ ('seq1', 'ATTCG-'), ('seq2'  , '--TTG-'), ('seq3', 'AT-CCG
↪')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTG- seq2
AT-CCG seq3
```

```
>>> ali.position_in_ali('seq1', 4)
4
>>> ali.position_in_ali('seq2', 0)
2
>>> ali.position_in_ali('seq3', 2)
3
```

See also:

position_in_seq maps an alignment position to sequence position for a single sequence

position_map maps all alignment positions for all sequences

position_in_seq(*name*, *pos_in_ali*)

Maps an alignment column position to the corresponding position in a certain sequence

Parameters

- **name** (*str*) – the name of the sequence to map to
- **pos_in_ali** (*int*) – 0-based alignment position, i.e. the column index

Returns

0-based position in sequence, i.e. the index of the sequence without counting gaps at the requested column.

Note that for gap positions, the position of the closest non-gap to the left is reported. For gap positions, the position of the closest non-gap to the left is reported.

Return type int

Examples

```
>>> ali= Alignment([ ('seq1', 'ATTCG-'), ('seq2'  , '--TTG-'), ('seq3', 'ATTCCG
↪')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTG- seq2
ATTCCG seq3
```

```
>>> ali.position_in_seq('seq1', 4)
4
>>> ali.position_in_seq('seq2', 2)
0
```

Checking a left terminal gap returns -1:

```
>>> ali.position_in_seq('seq2', 0)
-1
```

Checking a gap position returns the index of the closest non gap char on the left:

```
>>> ali.position_in_seq('seq1', 5)
4
```

See also:

position_in_ali maps a sequence position to alignment position for a single sequence

position_map maps all alignment positions for all sequences

position_map()

Compute a numerical matrix instrumental to map alignment positions to sequence positions (and reverse)

Returns Returns a Pandas DataFrame with one row per sequence (indexed by name), and one column per alignment position. Each value is the index of that particular sequence (without gaps) in that alignment column. All positions are 0-based. For gap positions, the position of the closest non-gap to the left is reported. For left terminal gaps, the value reported is -1

Return type pd.DataFrame

Examples

```
>>> ali= Alignment([ ('seq1', 'ATTCG-'), ('seq2'  , '--TTG-'), ('seq3', 'ATTCCG
↪')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTG- seq2
ATTCCG seq3
```

```
>>> ali.position_map()
     0  1  2  3  4  5
```

(continues on next page)

(continued from previous page)

```
seq1  0  1  2  3  4  4
seq2 -1 -1  0  1  2  2
seq3  0  1  2  3  4  5
```

Note: Computing this matrix makes sense only if you will use positions for many or all sequences. For the corresponding operations on single sequences, see functions *position_in_seq()* and *position_in_ali()*

See also:

position_in_seq maps an alignment position to sequence position for a single sequence

position_in_ali maps a sequence position to alignment position for a single sequence

positions()

Returns an iterator over the column indices of the alignment

This is qquivalent to range(self.ali_length())

Returns An iterator of int

Return type range

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> for i in ali.positions():      print( ali.get_seq('seq1')[i] )
A
T
T
C
G
-
```

remove_by_index(**seqindices*)

Remove one or more sequences in the alignment by their index, in-place.

The input indices refer to the position of the sequence in the alignment, i.e. their row number. Note that the modification is done in place. To obtain a new object instead, see examples below.

Parameters ***seqindices** (*tuple*) – index or indices of sequences to be removed from the alignment

Returns None

Return type None

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', 'ATTCG-')])
>>> ali.remove_by_index(0)
>>> ali
# Alignment of 2 sequences and 6 positions
--TTGG seq2
ATTCG- seq3
```

To return a new alignment without certain sequences, do not use this function. Instead, use indexing by rows:

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', 'ATTCG-')])
>>> indices_to_omit=set( [0, 2] )
>>> ali[ [n for i,n in enumerate(ali.names()) if not i in indices_to_omit], :]
# Alignment of 1 sequences and 6 positions
--TTGG seq2
```

See also:

remove_by_name

remove_by_name(**names*)

Remove one or more sequences in the alignment by name in-place.

Note that the modification is done in place. To obtain a new object instead, see examples below.

Parameters ***names** (*tuple*) – name or names of sequences to be removed from the alignment

Returns None

Return type None

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', 'ATTCG-')])
>>> ali.remove_by_name('seq1')
>>> ali
# Alignment of 2 sequences and 6 positions
--TTGG seq2
ATTCG- seq3
```

```
>>> ali.remove_by_name('seq2', 'seq3')
>>> ali
# Empty alignment
```

To return a new alignment without certain names, do not use this function. Instead, use indexing by rows:

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', 'ATTCG-')])
>>> names_to_omit=set(  ['seq2']  )
>>> ali[ [n for n in ali.names() if not n in names_to_omit], :]
```

(continues on next page)

(continued from previous page)

```
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
ATTCG- seq3
```

See also:

remove_by_index

remove_empty_seqs(*inplace=True*)

Remove all sequences which are entirely made of gaps or that are empty.

By default, removal is done in place.

Parameters **inplace** (*bool, default:True*) – whether the removal should be done in place. If not, a new Alignment is returned instead

Returns If inplace==True, None is returned; otherwise, a new Alignment without empty sequences is returned

Return type None or *Alignment*

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', '------')])
>>> ali.remove_empty_seqs()
>>> ali
# Alignment of 2 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
```

See also:

trim_gaps, *remove_by_name*, *remove_by_index*

same_length()

Check whether sequences are aligned, i.e. they have the same length

Returns Stating if all sequences have the same lengths

Return type bool

Examples

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali.same_length()
True
```

```
>>> ali.add_seq('seqX', 'TATTCGGT-')
>>> ali.same_length()
False
```

See also:

ali_length length of the alignment (i.e. number of columns)

score_similarity(*targets=None*, *gaps='y'*, *metrics='i'*, *weights='m'*, *method=2*)

Computes metrics of similarity between some target sequences and sequences in the (self) alignment

The rationale of the function is to quantify how much target sequences 'fit' in the alignment. By default, similarity metrics are computed for all sequences in self, meaning targets=self. In that case, they provide a measure of how much sequences resemble each other, i.e. the global agreement of alignment, and it can be instrumental to identify outliers.

The default use of this function is to compute the Average Sequence Identity (*ASI*) of targets, obtained by performing pairwise comparisons between each target and all sequences in self, and averaging. The definition of sequence identity varies as **gaps may be counted in different ways**, as specified by the gaps argument. For explanatory examples, see this page .

Besides *ASI*, a variant called *AWSI* (Average Weighted Sequence Identity) is available, wherein different alignment columns are given different *weight* when averaging. Various built-in methods to define weights are available, all based on the concept that conserved alignment positions are given more weight. Custom weights may also be provided.

Parameters

- **targets** (*Alignment, optional*) – sequences for which the similarity metrics are requested, provided as an Alignment instance. The sequences must be aligned to the self Alignment. If not provided, self is taken as targets, meaning that metrics are reported for each sequence in self, compared to the full alignment.
- **gaps** (*str, default:'y'*) – defines how to take into account gaps when comparing sequences pairwise. Possible values: - 'y' : gaps are considered and considered mismatches. Positions that are gaps in both sequences are ignored. - 'n' : gaps are not considered. Positions that are gaps in either sequences compared are ignored. - 't' : terminal gaps are trimmed. Terminal gap positions in either sequences are ignored, others are considered as in 'y'. - 'a' : gaps are considered as any other character; even gap-to-gap matches are scored as identities.

  Multiple arguments may be concatenated (e.g. 'yn') to compute all of the possibilities.
- **metrics** (*str, default:'i'*) – defines which metrics are computed. Possible values: - 'i' : average sequence identity, aka ASI - 'w' : weighted sequence identity, aka AWSI

  Multiple arguments may be concatenated (e.g. 'iw') to compute all of the possibilities.
- **weights** (*str or list or np.ndarray, default: 'm'*) – if AWSI is computed, defines how weights are computed for each alignment column. Possible values: - 'm' : maximum frequency of non-gap character in self - 'i' : information content, i.e. 2-sum(p*log2(p)) where p is frequency of non-gap characters - 'q' : quadratic sum, i.e. sum(p*p) where p is frequency of non-gap characters

  Multiple arguments may be concatenated (e.g. 'mi') to compute all of the possibilities.

  Alternatively, custom weights may be provided as an iterable (e.g. list or Numpy ndarray) of numbers (the weights), with as many elements as the alignment columns.

Returns a DataFrame with one row per target (indexed by sequence names), and one column for each sequence metrics requested. If multiple parameters were specified for *gaps*, *metrics* and/or *weights*, the output columns are a MultiIndex which covers all combinations requested.

Return type pd.DataFrame

See also:

sequtils.sequence_identity function that compares sequences pairwise and returns their sequence identity. Accepts the same gaps argument as score_similarity. Though sequtils.sequence_identity is

not run internally by score_similarity, it can be used to replicate its results.

Examples

```
>>> ali= Alignment([ ('seq1', 'ATTCG-'), ('seq2'  , '--TTG-'), ('seq3', 'AT-CCG
↪')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTG- seq2
AT-CCG seq3
```

```
>>> fep_ali=Alignment(pyaln_folder + '/examples/fep15_protein.fa', fileformat=
↪'fasta')
>>> fep_ali
# Alignment of 6 sequences and 138 positions
MWLTLVALLALCATGRTAENLSESTTDQDKLVIARGKLVAPSVVGUSIKKMPELYNFLM...L Fep15_danio_
↪rerio
MWAFLLLTLAFSATGMTEE-DVTDTAIEERPVIAKGILKAPSVVGUAIKKMPALYMFLM...L Fep15_S_salar
MWIFLLLTLAFSATGMTEE-NVTDTAIEERPVIAKGILKAPSVVGUAIKKMPELYTFLM...L Fep15_O_mykiss
MWAFLVLTFAVAA-GASET-VDNHTAAEEKLLIARGKLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_rubripes
MWALLVLTFAVTV-GASEE-VKNQTAAEEKLVIARGTLLAPSVVGUGIKKMPELHHFLM...L Fep15_T_
↪nigroviridis
MWAFVLIAFSV---GASDS--SNSTAE----VIARGKLMAPSVVGUAIKKLPELNRFLM...L Fep15_O_latipes
```

```
>>> fep_ali.score_similarity()
metrics                    ASI
Fep15_danio_rerio     0.777778
Fep15_S_salar         0.826334
Fep15_O_mykiss        0.822684
Fep15_T_rubripes      0.829599
Fep15_T_nigroviridis  0.815000
Fep15_O_latipes       0.767438
```

If we choose to not consider gap positions, the score increases:

```
>>> fep_ali.score_similarity(gaps='n')
metrics                    ASI
Fep15_danio_rerio     0.793051
Fep15_S_salar         0.838283
Fep15_O_mykiss        0.834522
Fep15_T_rubripes      0.842566
Fep15_T_nigroviridis  0.835351
Fep15_O_latipes       0.805693
```

Requesting AWSI as well as ASI, and testing both considering and omitting gaps:

```
>>> fep_ali.score_similarity(gaps='yn', metrics='iw')
gaps                        y                    n
metrics                   ASI      AWSI        ASI      AWSI
Fep15_danio_rerio     0.777778  0.847123  0.793051  0.856044
Fep15_S_salar         0.826334  0.885040  0.838283  0.893412
```

(continues on next page)

(continued from previous page)

```
Fep15_O_mykiss        0.822684  0.882183  0.834522  0.890497
Fep15_T_rubripes      0.829599  0.887255  0.842566  0.896094
Fep15_T_nigroviridis  0.815000  0.874389  0.835351  0.891288
Fep15_O_latipes       0.767438  0.834809  0.805693  0.860639
```

Computing AWSI with all possible weights available, considering only internal gaps:

```
>>> fep_ali.score_similarity(gaps='t', metrics='w', weights='iqm')
metrics                AWSI.i    AWSI.q    AWSI.m
Fep15_danio_rerio     0.881664  0.913531  0.847123
Fep15_S_salar         0.912174  0.937773  0.885040
Fep15_O_mykiss        0.910436  0.936245  0.882183
Fep15_T_rubripes      0.915546  0.938932  0.887255
Fep15_T_nigroviridis  0.901011  0.929572  0.874389
Fep15_O_latipes       0.872316  0.903712  0.834809
```

sequences()

Returns a list of the all sequences in the alignment

Returns An ordered list of sequences in the alignment (without names or descriptions)

Return type list of str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.sequences()
['ATTCG-', '--TTGG', 'ATTCG-']
```

set_desc(*name*, *desc*)

Change the description of an entry in-place.

Parameters

- **name** (*str*) – The name (i.e. identifier) of the entry to be altered
- **desc** (*str*) – The new description to be used

Returns None

Return type None

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3 not sure',   'ATTCG-')])
>>> print(ali.fasta())
>seq1 this is first
ATTCG-
>seq2 this is 2nd
--TTGG
>seq3 not sure
ATTCG-
```

```
>>> ali.set_desc('seq3', 'obviously third')
>>> print(ali.fasta())
>seq1 this is first
ATTCG-
>seq2 this is 2nd
--TTGG
>seq3 obviously third
ATTCG-
```

set_seq(*name*, *sequence*)
Change the sequence of an entry in-place.

Parameters

- **name** (*str*) – The name (i.e. identifier) of the sequence to be altered
- **sequence** (*str*) – The new sequence to be set

Returns None

Return type None

Examples

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTGG seq2
ATTCG- seq3
```

```
>>> ali.set_seq('seq1', 'CHANGE')
>>> ali
# Alignment of 3 sequences and 6 positions
CHANGE seq1
--TTGG seq2
ATTCG- seq3
```

See also:

add_seq, *get_seq*

property shape
Return the size of the alignment in its two dimensions, i.e. the number of sequences and alignment columns

Returns (height, width), where height is the number of sequences in the alignment and width the number of columns

Return type tuple of int

Examples

```
>>> ali=Alignment([ ('seq1', 'ATTCG-'), ('seq2', '--TTGG'), ('seq3', 'ATTCG-')])
>>> ali.shape
(3, 6)
```

Note: This method is presented as property for symmetry with Numpy array `.shape`. However, this Alignment property is read-only.

titles()

Returns a list of all sequence titles in the alignment

Each title is the concatenation of sequence name and description, separated by a space. If the description is empty for an entry, only the name is returned

Returns An ordered list of sequence titles in the alignment

Return type list of str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> ali.titles()
['seq1 this is first', 'seq2 this is 2nd', 'seq3']
```

See also:

names get all sequence names (their unique identifiers, without description)

to_biopython()

Returns a copy of the alignment as a Bio.Align.MultipleSeqAlignment object

The SeqRecord instances in the returned MultipleSeqAlignment has their id and name attributes set to sequence names, and also possess the description attribute.

Returns Alignment in biopython format (Bio.Align.MultipleSeqAlignment)

Return type MultipleSeqAlignment

See also:

to_numpy, *to_pandas*

to_numpy()

Returns a numpy 2-D array representation of the alignment, useful for vectorized sequence methods

Returns The returned array has one row per sequence and one column per alignment position. Each value is a single character. The dtype is np.str_ Note that rows are not indexed by sequence names, just by their order index

Return type np.ndarray

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', '--TT--')])
>>> print(ali.to_numpy())
[['A' 'T' 'T' 'C' 'G' '-']
 ['-' '-' 'T' 'T' 'G' 'G']
 ['-' '-' 'T' 'T' '-' '-']]
```

See also:

to_biopython, *to_pandas*

Warning: This function is cached for best performance. Thus, do not directly modify the returned object. The hash key for caching is derived from sequences only: names are not considered.

`to_pandas`(*use_names=False*)

Returns a pandas DataFrame representation of the alignment

Parameters **use_names** (*bool, optional*) – Normally, the returned DataFrame has a simply RangeIndex as index. Specify this to instead use sequence names as the index.

Returns The returned dataframe has one row per sequence and one column per alignment position. Each value is a single character. The dtype is object Rows are indexed by the sequence names if use_names==True, or by a RangeIndex by default

Return type pd.DataFrame

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTGG'),
↪ ('seq3', '--TT--')])
>>> ali.to_pandas()
   0  1  2  3  4  5
0  A  T  T  C  G  -
1  -  -  T  T  G  G
2  -  -  T  T  -  -
```

```
>>> ali.to_pandas(use_names=True)
      0  1  2  3  4  5
seq1  A  T  T  C  G  -
seq2  -  -  T  T  G  G
seq3  -  -  T  T  -  -
```

See also:

to_biopython, *to_numpy*

`trim_gaps`(*pct=1.0*, *count=None*, *inplace=False*)

Removes the alignment columns with more gaps than specified

By default, a new alignment without the columns identified as 'too gappy' is returned.

Parameters

- **pct** (*float, default:1.0*) – minimal gap frequency (from 0.0 to 1.0) for a column to be removed
- **count** (*int, optional*) – defines the minimum absolute number of gaps for a column to be removed. It is an alternative way to select columns, which overrides the pct argument
- **inplace** (*bool, default:False*) – whether the column removal should be done in place. If not, a new Alignment is returned instead

Returns By default, a new Alignment without empty sequences is returned; if inplace==True, None is returned

Return type None or *Alignment*

Examples

```
>>> ali= Alignment([ ('seq1 first', 'ATTCG-'), ('seq2 this is 2nd'  , '--TTG-'),
↪ ('seq3', 'ATTCCG')])
>>> ali
# Alignment of 3 sequences and 6 positions
ATTCG- seq1
--TTG- seq2
ATTCCG seq3
```

```
>>> ali.trim_gaps(0.5)
# Alignment of 3 sequences and 5 positions
ATTCG seq1
--TTG seq2
ATTCC seq3
```

```
>>> ali.trim_gaps(count=1)
# Alignment of 3 sequences and 3 positions
TCG seq1
TTG seq2
TCC seq3
```

write(*fileformat='fasta', to_file=None*)

Returns a string representation of the alignment in a format of choice

Internally uses Bio.Align to generate output. Supported fileformat arguments include clustal, stockholm, phylip and many others. The full list of supported fileformat arguments is provided here).

Parameters

- **fileformat** (*str, default='fasta'*) – text format requested
- **to_file** (*str | TextIO, optional*) – filename or buffer to write into. If not specified, the output is returned instead

Returns String representation of alignment in the requested format

Return type str

Examples

```
>>> ali=Alignment([ ('seq1 this is first', 'ATTCG-'), ('seq2 this is 2nd'  , '--
↪TTGG'), ('seq3', 'ATTCG-')])
>>> print(ali.write('phylip'))
 3 6
seq1       ATTCG-
seq2       --TTGG
seq3       ATTCG-
```

CHAPTER

FOUR

SEQUTILS SUBMODULE

pyaln.sequtils.**sequence_identity**(*a*, *b*, *gaps='y'*)

Compute the sequence identity between two sequences.

The definition of sequence_identity is ambyguous as it depends on how gaps are treated, here defined by the *gaps* argument. For details and examples, see this page

Parameters

- **a** (*str*) – first sequence, with gaps encoded as "-"
- **b** (*str*) – second sequence, with gaps encoded as "-"
- **gaps** (*str*) – defines how to take into account gaps when comparing sequences pairwise. Possible values: - 'y' : gaps are considered and considered mismatches. Positions that are gaps in both sequences are ignored. - 'n' : gaps are not considered. Positions that are gaps in either sequences compared are ignored. - 't' : terminal gaps are trimmed. Terminal gap positions in either sequences are ignored, others are considered as in 'y'. - 'a' : gaps are considered as any other character; even gap-to-gap matches are scored as identities.

Returns sequence identity between the two sequences

Return type float

Examples

```
>>> sequence_identity('ATGCA',
...                   'ATGCC')
0.8
```

```
>>> sequence_identity('--ATC-GGG-',
                      'AAATCGGGGC',
                      gaps='y')
0.6
```

Note: To compute sequence identity efficiently among many sequences, use *score_similarity()* instead.

See also:

pyaln.Alignment.score_similarity, *weighted_sequence_identity*

pyaln.sequtils.**weighted_sequence_identity**(*a*, *b*, *weights*, *gaps='y'*)

Compute the sequence identity between two sequences, different positions differently

The definition of sequence_identity is ambyguous as it depends on how gaps are treated, here defined by the *gaps* argument. For details and examples, see this page

Parameters

- **a** (*str*) – first sequence, with gaps encoded as "-"
- **b** (*str*) – second sequence, with gaps encoded as "-"
- **weights** (*list of float*) – list of weights. Any iterable with the same length as the two input sequences (including gaps) is accepted. The final score is divided by their sum (except for positions not considered, as defined by the gaps argument).
- **gaps** (*str*) – defines how to take into account gaps when comparing sequences pairwise. Possible values: - 'y' : gaps are considered and considered mismatches. Positions that are gaps in both sequences are ignored. - 'n' : gaps are not considered. Positions that are gaps in either sequences compared are ignored. - 't' : terminal gaps are trimmed. Terminal gap positions in either sequences are ignored, others are considered as in 'y'. - 'a' : gaps are considered as any other character; even gap-to-gap matches are scored as identities.

Returns sequence identity between the two sequences

Return type float

Examples

```
>>> weighted_sequence_identity('ATGCA',
...                            'ATGCC', weights=[1, 1, 1, 1, 6])
0.4
```

```
>>> weighted_sequence_identity('ATGCA',
...                            'ATGCC', weights=[1, 1, 1, 1, 1])
0.8
```

Note: To compute sequence identity efficiently among many sequences, use *score_similarity()* instead.

See also:

pyaln.Alignment.score_similarity, *weighted_sequence_identity*

CHAPTER

FIVE

INDEX

PYTHON MODULE INDEX

INDEX