Tiger Lily Documentation

Release 0.1-dev

Erich Blume <blume.erich@gmail.com>
Euclid Sun <euclidsun@gmail.com>

September 23, 2011

CONTENTS

1 About Tiger Lily **1**
- 1.1 Priorities . . . 1
- 1.2 Tiger Lily and Bioinformatics . . . 1
- 1.3 Tiger Lily and Python . . . 1
- 1.4 What about Biopython? . . . 2
- 1.5 Why 'Tiger Lily'? . . . 2

2 Project Modules **3**
- 2.1 tigerlily Package . . . 3
  - 2.1.1 `tigerlily` Package . . . 3
  - 2.1.2 Subpackages . . . 3
    - grc Package . . . 3
      - `grc` Package . . . 3
      - `genome` Module . . . 3
    - index Package . . . 5
      - `fixedtree` Module . . . 5
      - `index` Module . . . 7
    - sequences Package . . . 7
      - `sequences` Package . . . 7
      - `fasta` Module . . . 7
      - `genomic` Module . . . 8
      - `mixed` Module . . . 11
      - `raw` Module . . . 12
      - `sequence` Module . . . 13
    - utility Package . . . 14
      - `utility` Package . . . 14
      - `string_relations` Module . . . 14

Python Module Index **17**

Index **19**

CHAPTER

ONE

ABOUT TIGER LILY

Tiger Lily is a package written for Python 3 which provides tools commonly needed in bioinformatics. It does this by providing an easy-to-use package called `tigerlily`, which has modules written with the intention of being high performance and specific in purpose while being pluggable to fit a wide variety of purposes.

1.1 Priorities

Tiger Lily aims to be the go-to tool for any bioinformatics application. This means Tiger Lily should:

1. Be well documented.
2. Be well tested.
3. Have *Batteries Included*, and provide tools for every common task.
4. Be extensible, allowing other packages to modify, extend, and enhance Tiger Lily with ease and grace.
5. Be pluggable, allowing applications to pick and choose what tools it will need and use them together in whatever order it pleases.
6. Be *fast*, and meet the often staggering needs of high-throughput sequencing.
7. Be *scalable* through parallelism and design, allowing the user to get the most out of her hardware.

1.2 Tiger Lily and Bioinformatics

As of this early writing, Tiger Lily is still in its infancy. As we move closer to the goals listed above we will expand the breadth and depth of Tiger Lily.

For now, Tiger Lily is mostly applicable towards aligning short reads against a reference genome, as well as providing a flexible tool for converting between common formats. While most of the early developmental focus will be expended on short read sequence alignment, this is by no means the full scope of Tiger Lily. (It just happens to be the domain of the main author’s experience.)

1.3 Tiger Lily and Python

Tiger Lily was specifically built for Python 3 with an eye towards current and emerging standards. As much as possibly, Tiger Lily will incorporate existing features in the standard py3k library.

What this means to you, the user, is that you should feel comfortable in knowing that Tiger Lily (if we are doing our job) is staying current and relevant.

1.4 What about Biopython?

As the reader may or may not be aware, there is already a similar project to Tiger Lily called Biopython. Biopython supports a very wide range of sequence formats including coverage of nearly all of the most commonly used web resources (NCBI, UCSC Genome Browser, SwissProt, UniProt, Sanger Institute, etc.). It also has the benefit of having a very wide international support base and is well respected.

We believe that Tiger Lily still has a place beside Biopython for the following reasons:

1. Tiger Lily is targeting Python 3, which Biopython does not currently support.
2. Tiger Lily is aimed at high speed computing for short read sequencing applications where biopython is aimed more at more traditional forms of bioinformatics, sometimes at the cost of performance.
3. Tiger Lily has a different 'flavor' that the authors find more palatable.

That being said, long-term it is very likely that the fruits of the Tiger Lily project will end up being submitted to Biopython as a feature enhancement for their consideration. Still, Tiger Lily will be developed as though the goal were to eventually have Tiger Lily be the go-to resource for python bioinformatics.

1.5 Why 'Tiger Lily'?

The name wasn't chosen for any particular reason - a name was needed and the original author felt that tiger lillies might make for a good logo some day.

CHAPTER

TWO

PROJECT MODULES

2.1 tigerlily Package

2.1.1 `tigerlily` Package

2.1.2 Subpackages

grc Package

`grc` Package

tigerlily.grc - functions and extensions for common GRC-related tasks.

The members of this package generally interact with online or downloaded resources from the members of the GRC (Genome Reference Consortium) and other online resource entities such as NCBI, Sanger Institute, and the UCSC Genome Browser.

`genome` Module

class `tigerlily.grc.genome.`**`GRCGenome`**

Bases: `tigerlily.grc.genome.ReferenceGenome`

Fetch, store, load, parse, and extract sequences from a GCR ref assembly

NCBI has released many top-quality reference genomes. As of late 2010, these assemblies are produced under the brand of the GRC - the Genome Reference Consortium. The UCSC Genome Browser mirrors each of these assemblies in such a way that makes it very conveniant to download and extract data that is useful to Tiger Lily.

This class provides an interface to automatically download, store, load, parse, and extract sequences from the UCSC Genome Browser's copies of the GRC reference genomes.

Support for different assemblies will be added manually to this class. For a list of supported assemblies by their name, see SUPPORTED_ASSEMBLIES . The default (most current) assembly will be stored in DEFAULT_ASSEMBLY

For the convenience of faster non-networked tests, extremely small made-up reference genomes are provided inside of this package in a folder called 'test_assemblies'. They can be loaded either by using the GRCGenome.load method (as normal), or else by using GRCGenome.download with names that start with 'test' (eg. 'test1', 'test2', 'testnomask', etc.).

classmethod `download`(*name='h19'*, *store=False*, *silent=True*)

Download a reference genome of the given name, and return a GRCGenome

Fetches the named reference assembly (default is DEFAULT_ASSEMBLY) from the web, and creates a new GRCGenome object to handle it.

If store is False (default), the data will be kept entirely in memory, and will be destroyed as soon as the object is released. If True, the .tar.gz of the entire assembly will be saved in the current directory - ValueError will be raised if this file seems to already exist. If store is a string, it will be assumed to be a path to a directory (trailing slash optional) in which the .tar.gz archive should be stored. (Again, ValueError will be raised if the file already exists.)

If silent is False, status messages will be printed using print() to keep the user informed of the progress. This is usually very important in command line applications as the reference archives are about 900 MB in size and may take minutes or hours to download depending on the internet connection.

Because of the large size of these files, it is highly recommended that the store option be set. Please do not use Tiger Lily to abuse the UCSC Genome Browser group's generosity in hosting these large files to the general public.

```
>>> refgen = GRCGenome.download('test1')
>>> refgen2 = GRCGenome.download('test1',store=True)
>>> import os
>>> os.path.isfile('test1.tar.gz')
True
>>> os.unlink('test1.tar.gz')
```

Only supported reference genome assemblies are allowed, otherwise ValueError will be raised.

```
>>> GRCGenome.download('invalid')
Traceback (most recent call last):
    ...
ValueError: Unknown or unsupported reference genome specified
```

classmethod `load`(*path*)

Load the given .tar.gz archive file as a downloaded GRC Genome.

It is expected that this file will be named <assembly>.tar.gz

```
>>> import os
>>> refgen = GRCGenome.download('test1',store=True)
>>> os.path.isfile('test1.tar.gz')
True
>>> refgen2 = GRCGenome.load('test1.tar.gz')
>>> len(refgen2.sequences()) == len(refgen.sequences())
True
>>> os.unlink('test1.tar.gz')
```

classmethod `load_archive`(*archive*)

Given an instance of tarfile.TarFile, load it as a ref. assembly.

`sequences`()

Create a MixedSequenceGroup of the FASTA sequences from this ref.

```
>>> from tigerlily.sequences import PolymerSequenceGroup
>>> refgen = GRCGenome.download('test1')
>>> seqs = refgen.sequences()
>>> isinstance(seqs,PolymerSequenceGroup)
True
```

class `tigerlily.grc.genome.`**`ReferenceGenome`**

Bases: `builtins.object`

Abstract base class for all Genome objects.

ReferenceGenome objects represent an underlying file or data structure that encodes a Reference Genome assembly (often called simply an 'assembly', although in Tiger Lily they will normally use the full name Reference Genome to help clarity).

ReferenceGenome objects convert such file structures in to sequence groups, normally of the type tigerlily.sequences.MixedGroup, with the contained sequences being tigerlily.sequences.NucleicSequence - one per each assembly unit (nominally chromosomes, but often with other meta-units included), with the identifier set appropriately.

`sequences``()`

Return a PolymerSequenceGroup representing this Genome.

Normally this will be a MixedSequenceGroup with NucleicSequence members.

index Package

`fixedtree` Module

class `tigerlily.index.fixedtree.`**`FixedTree`**`(`*sequence_group*, *width*, *reverse=False*`)`

Bases: `tigerlily.index.index.GroupIndex`

GroupIndex wrapper which implements a Fixed-Width Substring Tree.

The resulting tree supports alignments of a fixed width only. It supports memory and time efficient lookups including mismatches. It does not support gaps, indels, arbitrary width lookups, etc.

As an example, the following block of code will create a MixedSequenceGroup that we can index.

```
>>> from tigerlily.sequences import NucleicSequence
>>> from tigerlily.sequences import createNucleicSequenceGroup
>>> s1 = NucleicSequence('ACGTACTTAGCATCATACGTCAGTACGCAGTCAGTCAGTCAT')
>>> s2 = NucleicSequence('CGAGCGACGCAGTACGTACTGGCAGACGTGTATACCTGC')
>>> group = createNucleicSequenceGroup(s1,s2)
>>> index = FixedTree(group,5)
```

`add_sequence``(`*sequence*, *reverse*`)`

Add the given sequence to this index.

The sequence must be a NucleicSequence, and it will be split in to subsequences of the length specified by this index.

If reverse is True, each individual subsequence will also be reversed.

```
>>> from tigerlily.sequences import NucleicSequence
>>> from tigerlily.sequences import createNucleicSequenceGroup
>>> s1 = NucleicSequence('ACGTACTTAGCATCATACGTCAGTACGCAGTCAGTCAGTCAT')
>>> s2 = NucleicSequence('CGAGCGACGCAGTACGTACTGGCAGACGTGTATACCTGC')
>>> group = createNucleicSequenceGroup(s1,s2)
>>> index = FixedTree(group,5)
>>> seq = NucleicSequence('CCCCC')
>>> index.add_sequence(seq,False)
```

`alignments``(`*sequence*, *mismatches=0*, *maximum_alignments=None*, *best_alignments=False*`)`

Returns a list of all alignments produced by the given input.

Retrurns a list of tuples, each tuple containing three values. They are: 0: str - 'chromosome name' 1: int - 'position' 2: boolean - 'strand' (True means 'reported strand', False means

'opposing strand'. Opposing strand matches do not alter the original position.)

Each reported alignment will have a computed 'Hamming Distance' (see http://en.wikipedia.org/wiki/Hamming_distance) of no greater than the mismatches argument. If left at 0, no actual edit distance calculations are performed.

If maximum_alignments is an integer greater than 0, then only that many alignments will be found - after that many are found, the search ends.

If best_alignments is True, then every possible alignment will be found even if maximum_alignments is set. Then, alignments are reported in increasing order of their edit distance from the search sequence. In general, this function will slow down the search considerably, particularly if maximum_alignments is set (since the primary benefit of maximim_alignments will be negated.) If left at False, no sorting is performed and the search can end as soon maximum_alignments has been reached.

This will raise ValueError if the given sequence does not match the pre-specified width of the index.

```
>>> from tigerlily.sequences import NucleicSequence
>>> from tigerlily.sequences import createNucleicSequenceGroup
>>> seq = NucleicSequence('GGAATTCC',identifier='foo')
>>> seqgroup = createNucleicSequenceGroup(seq)
>>> index = FixedTree(seqgroup,2)
>>> index.alignments('GG')
[('foo', 0, True)]
```

Note that for the next example with mismatches, the order of the result is unspecified, so we will just wrap it with len() to avoid testing errors. You don't need to wrap the result in len() in your own code.

```
>>> len(index.alignments('GG',mismatches=1))
2
```

For the next example we enable best_alignments, so the order is gaurunteed.

```
>>> index.alignments('GG',mismatches=1,best_alignments=True)
[('foo', 0, True), ('foo', 1, True)]
```

class `tigerlily.index.fixedtree.`**`FixedTreeNode`**(*alignment=None*)

Bases: `builtins.object`

Implicit tree data structure for storing a fixed read length index.

Each node in the graph may contain alignments (although the Root node is gaurunteed to not store any alignments). Any input sequence that lands on a node with a set of alignments may report those alignments.

Each node in the graph may contain a dictionary that maps strings to other nodes - these strings represent labeled edges. Moving along an edge consumes the corresponding prefix from the input sequence.

Hamming distance may be used to move along an edge that doesn't exactly match to a prefix of the input sequence. If the Hamming distance between an edge and a prefix of the input sequence is less than or equal to the number of remaining mismatches, then travel may proceed across that edge, although the mismatch count is decremented by that hamming distance.

Great care is made to ensure that the following properties are maintained in this structure at all times:

- For any given node N, for all of N's edges E1, there exists no edge in N called E2 that is not E1 but for which E2 is an exact prefix of E1.
- For any given node N, for all of N's children M, there is no way to re-distribute the edges between N and the M's (even by creating new children nodes in-between) in order to increase the length of an edge's label without violating the first property.

In other words, the edges of each node are ‘maximally uncommon’ - they are chosen to be as long as possible without sharing any prefixes.

Because of this property, we can be sure of correctness and optimality.

Well, I’m mostly hoping about the optimality part. I haven’t done the math.

`alignments` (*sequence*, *original_mismatches*, *remaining_mismatches=0*, *maximum_alignments=None*)

`insert` (*sequence*, *alignment*)

`index` Module

class `tigerlily.index.index.`**`GroupIndex`** (*sequence_group*, ***kwargs*)

Bases: `builtins.object`

Abstract base class for all genomic indexes.

`alignments` (*sequence*, ***kwargs*)

Gather all alignments for the given sequence. (See __contains__).

Implementing subclasses may (and almost certainly will) provide additional arguments to constrain the alignment set, such as to allow for a certain number of mismatches or to only generate the closest fit, etc.

The return value will always be a list or list-like object, with each item corresponding to an alignment. (The list may be empty.) However, each alignment’s representation is *undefind*. This is because different indexing methods may or may not be able to provide the different details allowed by other indexing methods. Consult the implementing subclasses’ documentation for the structure of an alignment.

sequences Package

`sequences` Package

`fasta` Module

class `tigerlily.sequences.fasta.`**`FASTA`** (*file=None*, *data=None*)

Bases: `tigerlily.sequences.sequence.PolymerSequenceGroup`

Implementation of an NCBI-compatible FASTA format.

classmethod **`load_sequences`** (**sequences*)

Create a new FASTA object from a list of arbitrary sequences.

```
>>> from tigerlily.sequences import RawSequence,NucleicSequence
>>> seq1 = RawSequence('AACGGTTACGATCAGGACTACGGGAGGAGAGA')
>>> seq2 = NucleicSequence('ACGGACTTACCAGGACTACGGACTCAGACG')
>>> fasta = FASTA.load_sequences(seq1,seq2)
>>> len(fasta)
2
```

`write` (*file*)

Write this FASTA group to a file, producing a conforming FASTA file.

```
>>> data = r'''>SEQUENCE_1
... MTEITAAMVKELRESTGAGMMDCKNALSETNGDFDKAVQLLREKGLGKAAKKADRLAAEG
... LVSVKVSDDFTIAAMRPSYLSYEDLDMTFVENEYKALVAELEKENEERRRLKDPNKPEHK
... IPQFASRKQLSDAILKEAEEKIKEELKAQGKPEKIWDNIIPGKMNSFIADNSQLDSKLTL
... MGQFYVMDDKKTVEQVIAEKEKEFGGKIKIVEFICFEVGEGLEKKTEDFAAEVAAQL
... >SEQUENCE_2
... SATVSEINSETDFVAKNDQFIALTKDTTAHIQSNSLQSVEELHSSTINGVKFEEYLKSQI
... ATIGENLVVRRFATLKAGANGVVNGYIHTNGRVGVVIAAACDSAEVASKSRDLLRQICMH
... '''
```

We could use the file-based loading initializer for FASTA, but let's just leave it as a string for simplicity.

```
>>> seqs = FASTA(data=data)
>>> len(seqs)
2
```

Now we write the sequences out to a buffer.

```
>>> import io
>>> buffer = io.StringIO()
>>> seqs.write(buffer)
```

You'll just have to trust that the output is consistent, and that it only differs from data in terms of superficial formatting.

class `tigerlily.sequences.fasta.`**`FASTASequence`**`(`*sequence*, *identifier*`)`

Bases: `tigerlily.sequences.sequence.FormattedSequence`

Container for a single FASTA Sequence.

Do not use this object. Instead, use the FASTA object, which subclasses PolymerSequenceGroup. The FASTA format is intrinsically a set, so most of the parsing logic is left in the FASTA object.

Nothing is *stopping* you from using this object, but you will probably find that it doesn't do very much work for you.

`MAX_LINE_WIDTH` **= 79**

`identifier`

`sequence`

`write``(`*file*`)`

Write this FASTA sequence to the opened file object.

Note that this function has the same result as writing the output of .format(). However, this function will outperform* that approach for large sequences (such as reference genomes stored in FASTA format), because this function doesn't store the entire sequence in memory a second time like .format() does (for text wrapping purposes).

***: OK, the performance will be mostly identical, it's just the** memory footprint should be much smaller.

`genomic` Module

class `tigerlily.sequences.genomic.`**`AminoSequence`**`(`*sequence*, *identifier=None*`)`

Bases: `tigerlily.sequences.sequence.PolymerSequence`

PolymerSequence for aminoacid sequences (e.g. protein sequences).

Each member of the sequence must be one of ABCDEFGHIKLMNOPQRSTUVWYZX*

`identifier`

Returns the identifier, if any.

If the identifier has not been explicitly set, the default identifier will be used instead.

sequence

translations()

Generate every NucleicSequence that could translate to this.

Because any given AminoSequence might have multiple nucleic acid sequences that could translate in to it, this generator function will iterate over every possible NucleicSequence object that could make this AminoSequence.

Note that the generated NucleicSequence objects do not use any sort of genomic signaling or encoding parameters - in other words, stop and start codons are not implicitly added unless the AminoSequence contained them to begin with.

```
>>> s1 = AminoSequence('NDC')
>>> trans = [x.sequence for x in s1.translations()]
>>> len(trans)
8
>>> 'AATGATTGT' in trans
True
```

class `tigerlily.sequences.genomic.`**`NucleicSequence`**(*sequence*, *identifier=None*)

Bases: `tigerlily.sequences.sequence.PolymerSequence`

Container for nucleic (chromosomal DNA) sequences.

In general this is not an 'input' or 'output' format, but rather an internal format that constrains the sequence to a certain set of allowed characters in the sequence.

The set of characters allowed is the following: ATGC

If a sequence is converted to a NucleicSequence and doesn't fit that set, ValueError will be raised.

NucleicSequence objects do not have a valid write() method, nor do they have a valid format() method. (Instead they will raise NotImplementedError.)

```
>>> import tigerlily.sequences.raw as raw
>>> seqdata = 'CTAGCATACTCACAGT'
>>> seq = raw.RawSequence(seqdata)
>>> nucleic = seq.convert(NucleicSequence)
>>> nucleic.sequence
'CTAGCATACTCACAGT'
```

An example showing that NucleicSequence objects can be instantiated directly.

```
>>> seq2 = NucleicSequence(seqdata)
>>> seq2.sequence == nucleic.sequence
True
```

An example where the constraint fails:

```
>>> seq = raw.RawSequence('CAGTTACTm')
>>> nucleic = seq.convert(NucleicSequence)
Traceback (most recent call last):
    ...
ValueError: Invalid nucleic sequence format: CAGTTACTm
```

complement()

Return the purine<->pyrimidine complement of the sequence as a new sequence.

```
>>> seq1 = NucleicSequence('AATGCC')
>>> cseq1 = seq1.complement()
>>> cseq1.sequence
```

```
'TTACGG'
>>> seq2 = cseq1.complement()
>>> seq2.sequence == seq1.sequence
True
```

identifier
The identifier for this read.

reverse()
Return a new sequence that is the reverse of this sequence.

```
>>> seq1 = NucleicSequence('AATGCC')
>>> rseq1 = seq1.reverse()
>>> rseq1.sequence
'CCGTAA'
>>> seq2 = rseq1.reverse()
>>> seq2.sequence == seq1.sequence
True
```

reverse_complement()
Return the reverse complement of the sequence as a new sequence.

Has the same result as sequence.reverse().complement(), but slightly more efficient.

```
>>> seq1 = NucleicSequence('AATGCC')
>>> rcseq1 = seq1.reverse_complement()
>>> rcseq1.sequence
'GGCATT'
>>> seq2 = rcseq1.reverse_complement()
>>> seq2.sequence == seq1.sequence
True
>>> rcseq1.sequence == seq1.reverse().complement().sequence
True
```

sequence

translate(*reading_frame=1*, *use_control_codes=False*)
Convert this NucleicSequence to a corresponding AminoSequence.

Conversion is performed by taking each codon and replacing it with the corresponding amino acid. In biological terms, translate() assumes that the NucleicSequence object's sequence is the coding sequence (and not the template). Incidentally, to get the coding sequence of a template, just call template.reverse_complement().

reading_frame is either 1, 2, or 3, and corresponds to the 1-based offset into the sequence from which to begin collecting codons. In other words, reading_frame=1 (the default) will start from the first base in the sequence, and reading_frame=2 from the 2nd, and so on. Higher values are allowed and will start further in to the sequence by skipping codons that would otherwise have been in the reading frame.

use_control_codes is a flag which, when True, enables an alternative processing algorithm. The new algorithm is like the first one (including the reading_frame), but after processing has finished the following constraint is placed upon the strand: all amino acids prior to and including the first Methionine (M/ATG) will be removed, and all amino acids after the first STOP codon (*/TAA,TGA,TAG) will be removed. If no Methionine is detected, ValueError will be raised. (It is not necessary for there to be a STOP codon.)

```
>>> nucleic = NucleicSequence('CATGGTATGTTTTGGGTTTAGAAACGT')
>>> amino = nucleic.translate()
>>> amino.sequence
'HGMFWV*KR'
```

We can also specify a reading_frame even though it will mean that the given input sequence doesn't cleanly subdivide in to codons - this is not an error.

```
>>> amino = nucleic.translate(reading_frame=2)
>>> amino.sequence
'MVCFGFRN'
```

Here is an example using use_control_codes mode.

```
>>> amino = nucleic.translate(use_control_codes=True)
>>> amino.sequence
'FWV'
>>> amino = nucleic.translate(use_control_codes=True,reading_frame=2)
>>> amino.sequence
'VCFGFRN'
>>> amino = nucleic.translate(use_control_codes=True,reading_frame=3)
Traceback (most recent call last):
    ...
ValueError: No Methionine found in translated nucleic sequence
```

`tigerlily.sequences.genomic.`**`createNucleicSequenceGroup`**`(`**sequences*`)`

Convert any group of sequences in to a nucleic MixedSequence group.

```
>>> from tigerlily.sequences.raw import Raw
>>> sequences = Raw(data='AGTACGTATTTCAT\nTTCATACGACTAC\n')
>>> len(sequences)
2
>>> nucleic = createNucleicSequenceGroup(*[s for s in sequences])
>>> len(nucleic)
2
>>> for seq in nucleic:
...     isinstance(seq,NucleicSequence)
...
True
True
```

`tigerlily.sequences.genomic.`**`reverse_complement`**`(`*sequence*`)`

Compute the reverse complement of the input string.

Input is assumed to be a string that could be the sequence of a NucleicSequence.

```
>>> reverse_complement('ACGGTC')
'GACCGT'
>>> reverse_complement('GACCGT')
'ACGGTC'
```

`mixed` Module

class `tigerlily.sequences.mixed.`**`MixedSequenceGroup`**`(`*sequence_group=None*`)`

Bases: `tigerlily.sequences.sequence.PolymerSequenceGroup`

PolymerSequenceGroup that allows sequences of any type.

What this gains in flexibility, it loses in representation. This subclass of PolymerSequenceGroup does not have a .write() method, since there probably isn't a good way to represent the sequences it contains.

We also gain an add() method, which allows additional sequences to be added to the sequence group after initialization - perhaps the key benefit of a MixedSequenceGroup.

```
>>> import tigerlily.sequences as tigseq
>>> seq1 = tigseq.FASTASequence(sequence='aCGTAtagcATCA',identifier='seq1')
>>> seq2 = tigseq.RawSequence(sequence='GGCATACGGCAatacgaCATN')
>>> sequences = tigseq.Raw(data='GGCATACT\nGAGcgaACT\n')
>>> genomic = tigseq.MixedSequenceGroup(sequences)
>>> len(genomic)
2
>>> genomic.add(seq1)
>>> genomic.add(seq2)
>>> len(genomic)
4
>>> genomic = tigseq.MixedSequenceGroup()
>>> len(genomic)
0
>>> genomic.add(seq1)
>>> len(genomic)
1
```

add(*sequence*)

`raw` Module

class `tigerlily.sequences.raw.`**`Raw`**(*file=None*, *data=None*)

Bases: `tigerlily.sequences.sequence.PolymerSequenceGroup`

Container for a set of raw (unformatted) sequences, one per line.

Use this object to parse data that stores sequences one per line without any additional formatting or information beyond the read and a newline character for each read (except possibly the last one).

Additionally, blank lines are skipped without an error (even if they contain white space characters).

```
>>> data = r'''
... CGTATACGCTCAGTC
... CGGGGCATCAGACTA
... CACGTACGACTACGTACGACTGACTGACTGCATCACATG
...
... LAGVVGALVUIALKT
... '''
>>> sequences = Raw(data=data)
>>> len(sequences)
4
```

write(*file*)

class `tigerlily.sequences.raw.`**`RawSequence`**(*sequence*, *identifier=None*)

Bases: `tigerlily.sequences.sequence.FormattedSequence`

Container for a 'raw' sequence.

Since raw reads are essentially just a sequence, this is a fairly useless class. It is more or less just a holder for the Raw sequence group object.

```
>>> seq = RawSequence(sequence='ATCGCGAGTCAGTCAGCATGACTACGCACAGTAC')
```

One possible use of this container is to create Raw sequences that know their identifier. Raw sequences that are given an identifier don't forget it and will happily send it when being converted to and from different formats. They just don't ever use them in output.

```
>>> seq = RawSequence(sequence='LALL',identifier='seq1')
```

An example converting from Raw to FASTA and back, to prove it works:

```
>>> import tigerlily.sequences as sequence
>>> fasta = seq.convert(sequence.FASTASequence)
>>> seq2 = fasta.convert(RawSequence)
>>> seq.sequence == fasta.sequence == seq2.sequence == 'LALL'
True
>>> seq.identifier == fasta.identifier == seq2.identifier == 'seq1'
True
```

`identifier`

`sequence`

`write`(*file*)

`sequence` Module

Module providing an abstraction of nucleic or amino acid polymer sequences.

class `tigerlily.sequences.sequence.`**`FormattedSequence`**(*sequence=None*, *identifier=None*)

Bases: `tigerlily.sequences.sequence.PolymerSequence`

Abstract base class for PolymerSequence objects which can be formatted.

Children of this class are sequences with some sort of character-encoded format. In particular, they support a .format() method and a .write() method.

`format`(*to_type=None*)

Return a string representing this sequence in the perscribed format.

The last character of the format is gaurunteed to be a newline.

To return this string in the format that the Sequence is already in, you may omit to_type.

Note that some FormattedSequence descendants will be representing a binary format. These descendants can choose whether they wish to return a 'bytes' object instead of a string, or simply raise an exception.

`write`(*file*)

Write the formatted output of this sequence in to the file.

For some formats this function may not make sense (such as for tile-and-cycle based outputs like Illumina's BCL format), and may return NotImplementedError instead.

The result, otherwise, will be the same as if you had called: file.write(sequence.format())

However, note that for some formats, calling this method is considerably faster than using .format(), particularly for large sequences.

class `tigerlily.sequences.sequence.`**`PolymerSequence`**(*sequence=None*, *identifier=None*)

Bases: `builtins.object`

Abstract base class for representing genomic sequences.

See the documentation for this module for examples on using this class.

`convert`(*to_type*)

Return this sequence as an instance of to_type.

This can be useful in changing a sequence from one format to another. For this purpose, see format()

No checking is made to ensure that to_type is a subclass of PolymerSequence, but it would probably be bad to pass in a type that isn't a PolymerSequence type.

`identifier`

Return a str object identifying the sequence.

Ideally this will come straight from the underlying source data (e.g. the header row on FASTA sequences), however, some data sources have no given identifier. In this case it is better to give SOME sort of contextualizing identifier than to give up and use a placeholder like 'Unknown'. An example would be to return the line number the sequence occured on, or the file that the sequence was contained in, or the date and time the sequence was processed, etc.

If all else fails and you can't identify the sequence properly, just use 'return super()'. Currently this will return the string 'Unknown'.

`sequence`

Return a str object representing the sequence.

class `tigerlily.sequences.sequence.`**`PolymerSequenceGroup`**

Bases: `_abcoll.Iterable`

Abstract base class for representing groups of genomic sequences.

This could be used for any purpose you like, but was originally intended for use when the underlying format implies a grouping of PolymerSequence objects, such as in the FASTA format.

utility Package

`utility` Package

`string_relations` Module

`tigerlily.utility.string_relations.`**`greatest_common_prefix`**`(`*s1*, *s2*`)`

Return the length of the longest common prefix between s1 and s2.

```
>>> greatest_common_prefix('banana','bandit')
3
>>> greatest_common_prefix('apple','sour apple')
0
>>> greatest_common_prefix('tree','tree')
4
```

`tigerlily.utility.string_relations.`**`hamming_distance`**`(`*s1*, *s2*`)`

Return the Hamming edit distance between s1 and s2.

The Hamming edit distance is defined as the number of individual alterations performed on characters in one string in order to turn it in to another string of the same length.

If s1 and s2 are not the same length, ValueError will be raised.

```
>>> hamming_distance('party','party')
0
>>> hamming_distance('zebra','cobra')
2
>>> hamming_distance('one','three')
Traceback (most recent call last):
    ...
ValueError: Cannot compute Hamming distance of strings of unequal length
```

With apologies to the fine editors of wikipedia.com for kernel of this code.

`tigerlily.utility.string_relations.`**`levenshtein_distance`**`(`*s1*, *s2*`)`

Return the Levenshtein edit distance (integer) between s1 and s2.

The Levenshtein is defined as the minimum number of substitutions, additions, and deletions needed to transform s1 in to s2.

```
>>> levenshtein_distance('kitten','sitten')
1
>>> levenshtein_distance('sitten','sittin')
1
>>> levenshtein_distance('sittin','sitting')
1
>>> levenshtein_distance('Alabama','Hell') # Surprisingly, not 0!
7
```

Thanks to hetland.org for this code: hetland.org/coding/python/levenshtein.py (no copyright information was found)

PYTHON MODULE INDEX

t

tigerlily.__init__, 3
tigerlily.grc, 3
tigerlily.grc.genome, 3
tigerlily.index.fixedtree, 5
tigerlily.index.index, 7
tigerlily.sequences, 7
tigerlily.sequences.fasta, 7
tigerlily.sequences.genomic, 8
tigerlily.sequences.mixed, 11
tigerlily.sequences.raw, 12
tigerlily.sequences.sequence, 13
tigerlily.utility, 14
tigerlily.utility.string_relations, 14

INDEX

A

add() (tigerlily.sequences.mixed.MixedSequenceGroup method), 12
add_sequence() (tigerlily.index.fixedtree.FixedTree method), 5
alignments() (tigerlily.index.fixedtree.FixedTree method), 5
alignments() (tigerlily.index.fixedtree.FixedTreeNode method), 7
alignments() (tigerlily.index.index.GroupIndex method), 7
AminoSequence (class in tigerlily.sequences.genomic), 8

C

complement() (tigerlily.sequences.genomic.NucleicSequence method), 9
convert() (tigerlily.sequences.sequence.PolymerSequence method), 13
createNucleicSequenceGroup() (in module tigerlily.sequences.genomic), 11

D

download() (tigerlily.grc.genome.GRCGenome class method), 3

F

FASTA (class in tigerlily.sequences.fasta), 7
FASTASequence (class in tigerlily.sequences.fasta), 8
FixedTree (class in tigerlily.index.fixedtree), 5
FixedTreeNode (class in tigerlily.index.fixedtree), 6
format() (tigerlily.sequences.sequence.FormattedSequence method), 13
FormattedSequence (class in tigerlily.sequences.sequence), 13

G

GRCGenome (class in tigerlily.grc.genome), 3
greatest_common_prefix() (in module tigerlily.utility.string_relations), 14
GroupIndex (class in tigerlily.index.index), 7

H

hamming_distance() (in module tigerlily.utility.string_relations), 14

I

identifier (tigerlily.sequences.fasta.FASTASequence attribute), 8
identifier (tigerlily.sequences.genomic.AminoSequence attribute), 8
identifier (tigerlily.sequences.genomic.NucleicSequence attribute), 10
identifier (tigerlily.sequences.raw.RawSequence attribute), 13
identifier (tigerlily.sequences.sequence.PolymerSequence attribute), 14
insert() (tigerlily.index.fixedtree.FixedTreeNode method), 7

L

levenshtein_distance() (in module tigerlily.utility.string_relations), 15
load() (tigerlily.grc.genome.GRCGenome class method), 4
load_archive() (tigerlily.grc.genome.GRCGenome class method), 4
load_sequences() (tigerlily.sequences.fasta.FASTA class method), 7

M

MAX_LINE_WIDTH (tigerlily.sequences.fasta.FASTASequence attribute), 8
MixedSequenceGroup (class in tigerlily.sequences.mixed), 11

N

NucleicSequence (class in tigerlily.sequences.genomic), 9

P

PolymerSequence (class in tigerlily.sequences.sequence), 13

PolymerSequenceGroup (class in tigerlily.sequences.sequence), 14

R

Raw (class in tigerlily.sequences.raw), 12
RawSequence (class in tigerlily.sequences.raw), 12
ReferenceGenome (class in tigerlily.grc.genome), 4
reverse() (tigerlily.sequences.genomic.NucleicSequence method), 10
reverse_complement() (in module tigerlily.sequences.genomic), 11
reverse_complement() (tigerlily.sequences.genomic.NucleicSequence method), 10

S

sequence (tigerlily.sequences.fasta.FASTASequence attribute), 8
sequence (tigerlily.sequences.genomic.AminoSequence attribute), 8
sequence (tigerlily.sequences.genomic.NucleicSequence attribute), 10
sequence (tigerlily.sequences.raw.RawSequence attribute), 13
sequence (tigerlily.sequences.sequence.PolymerSequence attribute), 14
sequences() (tigerlily.grc.genome.GRCGenome method), 4
sequences() (tigerlily.grc.genome.ReferenceGenome method), 5

T

tigerlily.__init__ (module), 3
tigerlily.grc (module), 3
tigerlily.grc.genome (module), 3
tigerlily.index.fixedtree (module), 5
tigerlily.index.index (module), 7
tigerlily.sequences (module), 7
tigerlily.sequences.fasta (module), 7
tigerlily.sequences.genomic (module), 8
tigerlily.sequences.mixed (module), 11
tigerlily.sequences.raw (module), 12
tigerlily.sequences.sequence (module), 13
tigerlily.utility (module), 14
tigerlily.utility.string_relations (module), 14
translate() (tigerlily.sequences.genomic.NucleicSequence method), 10
translations() (tigerlily.sequences.genomic.AminoSequence method), 9

W

write() (tigerlily.sequences.fasta.FASTA method), 7
write() (tigerlily.sequences.fasta.FASTASequence method), 8
write() (tigerlily.sequences.raw.Raw method), 12
write() (tigerlily.sequences.raw.RawSequence method), 13
write() (tigerlily.sequences.sequence.FormattedSequence method), 13