Tiger Lily Documentation

Release 0.1-dev

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

October 01, 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 About the Authors . . . 2
- 1.6 Why ‘Tiger Lily’? . . . 2

2 Tiger Lily Cookbook **3**
- 2.1 Downloading a Reference Genome . . . 3
- 2.2 Using Sequences . . . 4
  - 2.2.1 Creating Sequences . . . 4
  - 2.2.2 Converting Sequences . . . 5
  - 2.2.3 Manipulating `NucleicSequence` objects . . . 5
    - Translation . . . 5
    - Reverse and Complement . . . 5
  - 2.2.4 Writing in Another Format . . . 6

3 tigerlily Package **7**
- 3.1 `tigerlily` Package . . . 7
- 3.2 Subpackages . . . 7
  - 3.2.1 grc Package . . . 7
    - `grc` Package . . . 7
    - `genome` Module . . . 7
  - 3.2.2 index Package . . . 9
    - `fixedtree` Module . . . 9
    - `fixedtree_test` Module . . . 10
    - `index` Module . . . 10
  - 3.2.3 sequences Package . . . 11
    - `sequences` Package . . . 11
    - `fasta` Module . . . 11
    - `genomic` Module . . . 13
    - `raw` Module . . . 16
    - `sequence` Module . . . 17
  - 3.2.4 utility Package . . . 18
    - `utility` Package . . . 18
    - `archive` Module . . . 18
    - `archive_test` Module . . . 19
    - `download` Module . . . 19

`download_test` Module 20
`string_relations` Module 20

Python Module Index **23**

Index **25**

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 About the Authors

Tiger Lily was started by Erich Blume, an undergraduate Computer Science student at San Jose State University. Having some experience at a San Francisco Bay area biotech company with short read sequence alignment, Erich wanted to make Tiger Lily as the focus of a Bioinformatics class project. Euclid Sun is a classmate of Erich's studying Biochemistry and a fellow member of the SJSU computer science club.

1.6 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.

TIGER LILY COOKBOOK

Common procedures, recipes, operations, and usages of Tiger Lily, with clear documentation and example code.

2.1 Downloading a Reference Genome

Reference genome assemblies are carefully compiled consensus sequences for an *ideal* human genome. Often times, sequenced genetic material in the lab comes fragmented and without any notion of where in the specimen's genome it came from. Using a reference genome, you can hope to get a good idea of where the sequence came from by searching the reference for similar sequences.

With **Tiger Lily**, downloading and using a reference genome is very simple. To do this, we use the `tigerlily.grc` package to download reference genomes provided by the UCSC Genome Browser.

First, let's download a genome and store it so that we can use it later.

> **Remember: while Tiger Lily makes it easy to download a reference genome, these files are often very large and are served at the public expense from the UCSC Genome Browser web page. Please do not download reference genomes too often, and please *do* store genomes that you have downloaded.**

```
>>> import tigerlily.grc as grc
>>> ref_genome = grc.GRCGenome.download('hg19',store=True,silent=False)
Downloading http://.../chromFa.tar.gz
05:26 - 05:26 |==============================| 100% (926504K / 926501K)
```

The actual message printed may vary, but this should be close to what you see. After the download is complete, the prompt may 'hang' for up to a few minutes as the sequences are loaded from the downloaded archive.

Because we set `store=True`, a file called `hg19.assembly` will have been created.

```
>>> import os
>>> os.path.isfile('hg19.assembly')
True
```

Also, note that we can in the future load the file you just downloaded without hitting the poor UCSC Genome Browser's web server. **(You *do not* need to run this command now for this recipe - ‘‘ref_genome‘‘ is already correctly set after downloading the genome from the UCSC Genome Browser.)**

```
>>> ref_genome = grc.GRCGenome.load('hg19.assembly')
```

Finally, we can manipulate each individual sequence in the genome however we want. Every sequence generated by the `sequences()` method will be a `FASTASequence` object.

```
>>> sequences = [s for s in grc.sequences()]
```

2.2 Using Sequences

In bioinformatics, it is common to want to work on sequences of characters that represent either neucleotides or amino acids in order to represent genetic and proteomic units. The Tiger Lily package `tigerlily.sequences` provides support for these operations.

It's important to note that in Tiger Lily, all sequences descend from the common base class `tigerlily.sequences.PolymerSequence`, but you will generally never use that class directly. Also, some sequences are purely scientific abstractions (like `tigerlily.sequences.NucleicSequence`) while others are tightly related to a representation (like `tigerlily.sequences.FASTASequence`). This second class of sequences (that map to a format) all inherit from a the base class `tigerlily.sequences.FormattedSequence`, which are themselves just `PolymerSequence` objects that support a `format()` method and a `write` method.

Background aside, let's dive in to some examples of using Tiger Lily sequences.

2.2.1 Creating Sequences

For this example, lets pretend we have a *flat* text file of sequences. In other words, we have a text file which has one sequence per line, with nothing else at all in the file (except possibly empty lines). In Tiger Lily terms, this is called a Raw File, and it is supported with the classes and functions in `tigerlily.sequences.raw`, namely `Raw` and `parseRaw`, a class and a function respectively.

First, lets create a 'fake' file using the multi-line string syntax of Python.

```
>>> raw_reads = """CATGGCTTTGTGACTGAGTCCAGTAC
... ACTTGGATTATTCGATCGTAGCTATTATCGAC
... CACATTTAGGGAGGAGGAGACGATGCTAGCTAGCTGATTGTTATTATTATTATAGCGGGGCGCATGACT
... AAGAGAAAAAAAAAAAACGATACGACTACG
... ACGGCTAGCGTACGCTAGCCAAAACCACACATTTCATCATCA
... """
```

Keep in mind that as far as the RAW sequence format is concerned, sequences can be absolutely any string of characters except for the newline character - we are using only A, T, G, and C so that we can convert these reads in to `NucleicSequence` objects later.

Next, let's read these sequences in.

```
>>> from tigerlily.sequences import parseRaw
>>> seqs = [s for s in parseRaw(data=raw_reads)]
>>> len(seqs)
5
```

We can access the `sequence` attribute of each `RawSequence` object in the list thusly:

```
>>> seqs[0].sequence
'CATGGCTTTGTGACTGAGTCCAGTAC'
```

Additionally, the `Raw` class pre-loaded each sequence with an `identifier` attribute by using the line number of each sequence.

```
>>> seqs[2].identifier
'RawSeq_Line3'
>>> seqs[3].identifier
'RawSeq_Line4'
```

2.2.2 Converting Sequences

Now that we have our sequences as a bunch of `RawSequence` objects, we might wonder what we can do with that beyond having a somewhat indirect method of accessing each individual sequence as a string? Well, let's say that each sequence is the **coding sequence** of a protein subunit. We then might want to translate each sequence using the genetic code. With Tiger Lily, it's easy!

First, convert each `RawSequence` object to a `NucleicSequence` object:

```
>>> nucleic_seqs = [s.convert(NucleicSequence) for s in seqs]
```

Note: if any of the sequences didn't have just A, T, G, or C in it, that would have raised an error.

2.2.3 Manipulating `NucleicSequence` objects

Now, let's use the `translate` method of `NucleicSequence` objects to get the first coding frame translation of the first nucleic sequence:

```
>>> subunit = nucleic_seqs[0].translate()
>>> subunit.sequence
'HGVTESS'
>>> type(subunit)
<class 'tigerlily.sequences.genomic.AminoSequence'>
```

Translation

As you can see, the `translate` method changed our `NucleicSequence` object in to an `AminoSequence` object. In fact we can also go in the reverse direction, although due to the highly redundant nature of the genetic code, reverse translation produces many possible translations of amino acid sequences in to their corresponding nucleic sequences.

```
>>> len([t for t in subunit.translations()])
18432
```

We can also specify different *reading frames* of the nucleic sequence to get different translations:

```
>>> subunit = nucleic_seqs[0].translate(reading_frame=2)
>>> subunit.sequence
'MAL*LSPV'
```

We can even tell the translation function to honor control codes (START and STOP codons):

```
>>> subunit = nucleic_seqs[0].translate(reading_frame=2,use_control_codes=True)
>>> subunit.sequence
'AL'
```

Reverse and Complement

Another common task with `NucleicSequence` objects is to convert the sequence in to an equivalent sequence on the opposing strand and/or in the opposite direction. This is also simple to do in Tiger Lily:

```
>>> seq3 = nucleic_seqs[3]
>>> seq3.sequence
'AAGAGAAAAAAAAAAACGATACGACTACG'
>>> seq3.reverse().sequence
'GCATCAGCATAGCAAAAAAAAAAAGAGAA'
```

```
>>> seq3.complement().sequence
'TTCTCTTTTTTTTTTTGCTATGCTGATGC'
>>> seq3.reverse_complement().sequence
'CGTAGTCGTATCGTTTTTTTTTTTCTCTT'
```

Note that each command is generating an entirely new ‘‘NucleicSequence‘‘ object.

2.2.4 Writing in Another Format

Let’s say we want to take the original raw sequences we created earlier (in `seqs`) and write them in the FASTA file format. But that’s not all - we first want to find each of their reverse complements. You guessed it - all very simple!

First a little bit of setup:

```
>>> from tigerlily.sequences import writeFASTA
>>> from io import StringIO
>>> output = StringIO()
```

And now the business end of the operation:

```
>>> conv_seqs=[s.convert(NucleicSequence).reverse_complement() for s in seqs]
>>> writeFASTA(output,*conv_seqs)
```

If you examine the output, you should see:

```
>>> print(output.getvalue())
>RawSeq_Line1
GTACTGGACTCAGTCACAAAGCCATG
>RawSeq_Line2
GTCGATAATAGCTACGATCGAATAATCCAAGT
>RawSeq_Line3
AGTCATGCGCCCCGCTATAATAATAATAACAATCAGCTAGCTAGCATCGTCTCCTCCTCCCTAAATGTG
>RawSeq_Line4
CGTAGTCGTATCGTTTTTTTTTTTCTCTT
>RawSeq_Line5
TGATGATGAAATGTGTGGTTTTGGCTAGCGTACGCTAGCCGT
```

CHAPTER

THREE

TIGERLILY PACKAGE

3.1 `tigerlily` Package

tigerlily - a Python 3 package for bioinformatics

3.2 Subpackages

3.2.1 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: `builtins.object`

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 in a temporary file, and will be destroyed as soon as the object is released. If True, the entire assembly will be saved in the current directory - ValueError will be raised if this file seems to already exist. The resulting file will have the '.assembly' suffix, and may be either a .zip, a .tar, a .tar.gz, or a .tar.bz2 file. See `tigerlily.utility.archive.Archive` for more information. 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 necessary, any intermiediate directories will be created.

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.assembly')
True
>>> os.unlink('test1.assembly')
```

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`(*filename*)

Load the file given by filename as an `Archive` of a ref genome.

```
>>> import os
>>> refgen = GRCGenome.download('test1',store=True)
>>> os.path.isfile('test1.assembly')
True
>>> refgen2 = GRCGenome.load('test1.assembly')
>>> for seq1,seq2 in zip(refgen.sequences(),refgen2.sequences()):
...     seq1.sequence == seq2.sequence
True
True
True
>>> os.unlink('test1.assembly')
```

classmethod `load_archive`(*archive*)

Load the given `tigerlily.utility.archive.Archive` object as a reference genome assembly.

`sequences`()

Generates each sequence in the genome as a FASTASequence

```
>>> from tigerlily.sequences import PolymerSequence
>>> refgen = GRCGenome.download('test1')
>>> for seq in refgen.sequences():
```

```
...     isinstance(seq,PolymerSequence)
True
True
True
```

3.2.2 index Package

`fixedtree` Module

class `tigerlily.index.fixedtree.`**`FixedTree`**(*width*, *genome=None*, *reverse=False*)

Bases: `tigerlily.index.index.GroupIndex`

`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.

`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.

classmethod `load`(*filename*)

Create a new FixedTree from the named file.

`store`(*filename*)

Save the FixedTree to the file named by *filename*.

If *filename* already exists, EnvironmentError will be raised.

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*)

classmethod `load`(*buffer*)
Return a new node by reading it from buffer, recursively

`store`(*buffer*)
Copy this node in to *buffer*, and then recursively (but in a fixed order) copy the children.

`fixedtree_test` Module

This module provides unit tests for the `tigerlily.index.fixedtree` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `tigerlily.index.fixedtree_test.`**`FixedTreeTests`**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Test harness for `tigerlily.index.fixedtree.FixedTree` class.

`setUp`()
Create the testing environment

`tearDown`()
Remove the testing environment

`test_create_index`()
fixedtree.py: Test tree creation from NucleicSequence group

`test_store_index`()
fixedtree.py: Test writing FixedTree to disk

`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.

3.2.3 sequences Package

`sequences` Package

tigerlily.sequences - tools for handling polynomial sequences (DNA, RNA, etc)

All sequences descend from two parent classes, `PolymerSequence` and `PolymerSequenceGroup`. Another group of `PolymerSequence` descendents inherit from `FormattedSequence`, which adds support for printing or saving the sequence to some file format.

`fasta` Module

Support for the NCBI FASTA format.

This implementation honors the NCBI FASTA requirement that sequences not contain any comments (only a single identifier line is allowed), and that '>' is the only valid identifier character (not ';').

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

Bases: `tigerlily.sequences.sequence.FormattedSequence`

Container for a single FASTA Sequence.

To parse a list of FASTA sequences out of a FASTA-formatted file, use `parseFASTA()` from this module. To create a FASTA-formatted file from a list of (any type of) sequence, use `writeFASTA()`.

You can also instantiate `FASTASequence` objects directly. There is often little point in doing so, but it is an option available to you. Keep in mind that `FASTASequence` objects *need* both a sequence and an identifier.

`FASTASequence` sequences must use a subset of the NCBI specification. The sequence must only contain uppercase or lowercase variants of the characters in either `FASTASequence.ALLOWED_NUCLEIC_CHARS` or `FASTASequence.ALLOWED_AMINO_CHARS`. You can override the variables or (preferably) subclass `FASTASequence` to allow different characters in the sequence (and you are encouraged to do so), but you then lose the gauruntee that the resulting sequence will always be NCBI-portable. Note that 'degenerate' sequences and some other control or meta sequences are not supported in this implementation even though the NCBI specification does allow them - this is for interoperability with other Tiger Lily sequence classes.=

```
>>> seq1 = FASTASequence('TTAATTCTACTTATTTTATTA',identifier='seq1')
>>> seq1.format()
'>seq1\nTTAATTCTACTTATTTTATTA\n'
```

`ALLOWED_AMINO_CHARS` = 'ACDEFGHIKLMNPQRSTVWYX*'

`ALLOWED_NUCLEIC_CHARS` = 'ATGCUN'

`MAX_LINE_WIDTH` = 79

identifier

The identifier of this FASTASequence.

Note that unlike many other `PolynomialSequence` descendents, this class *requires* a valid identifier to be given at init time.

sequence

The sequence of this FASTASequence.

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 that the** memory footprint should be much smaller.

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

Parse the given file or data and return `FASTASequence` objects.

This function will generate each sequence in *file* or *data*.

You must specify either *file* or *data* and not both. *data* must be a sequence of type `str` (and not `bytes`). *file* can be any file-like object opened for reading in non-binary mode (such that `file.read()` will produce `str` objects).

```
>>> example_data = (">seq1\n"
... "CATTTACGGTACGTGATCTTACGATGCTAGCTTTGTACTAC\n"
... ">seq2\n"
... "TTAGGGACGTAATCGGACTCAGACGTTTTATGCGCGCGGCGCTTGGCGATATTAGGCGT\n"
... )
>>> seqs = [s for s in parseFASTA(data=example_data)]
>>> len(seqs)
2
>>> seqs[1].identifier
'seq2'
>>> seqs[0].sequence
'CATTTACGGTACGTGATCTTACGATGCTAGCTTTGTACTAC'
```

And then, as a file-object:

```
>>> from io import StringIO
>>> filewrap = StringIO(example_data)
>>> file_seqs = [s for s in parseFASTA(file=filewrap)]
>>> len(file_seqs)
2
```

The results are equivalent:

```
>>> for s1,s2 in zip(seqs,file_seqs):
...     s1.sequence == s2.sequence
...     s1.identifier == s2.identifier
True
True
True
True
```

`tigerlily.sequences.fasta.`**writeFASTA** (*file*, **seqs*)

Write an arbitray amount of sequences to an open writable file object.

The sequences in *seqs* will be converted to `FASTASequence` objects first, and then printed using the `write` method.

The result is a valid FASTA-formatted file.

```
>>> import sys
>>> from tigerlily.sequences import NucleicSequence
>>> writeFASTA(sys.stdout, NucleicSequence('ATTTCGAT'))
>Unknown
ATTTCGAT
```

`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'
>>> nucleic[1:3].sequence
'TA'
```

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.`**`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'
```

`raw` Module

Support for sequences that don't fit any other format, or have none.

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

Bases: `tigerlily.sequences.sequence.FormattedSequence`

Container for a 'raw' sequence. This is just a bare sequence without any other syntactic or semantic requirement or information. That makes this class useful for handling sequences that don't fit in any other class.

```
>>> 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*)

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

Parse the given file or data to yield all `RawSequence` objects

Either *file* or *data* must be set. *data* must be a `str` object and *file* must be a file-like object opened in non-binary mode (such that it returns `str` objects when `.read()` is called.)

```
>>> data = ("TGATCGCAGTCAG\n"
...         "ATATCGTA\n"
...         "\n"
...         "TTGATTAGCTAGTCGACGAT\n"
...         "\n"
...         "ACGTTGTTTTAGTCAGTC"
... )
>>> seqs = [s for s in parseRaw(data=data)]
>>> len(seqs)
4
>>> seqs[1].identifier
'RawSeq_Line2'
>>> seqs[2].identifier
'RawSeq_Line4'
```

And the equivalent using file-based objects:

```
>>> from io import StringIO
>>> data_f = StringIO(data)
>>> seqs_f = [s for s in parseRaw(file=data_f)]
>>> for s1,s2 in zip(seqs,seqs_f):
```

```
...     s1.sequence == s2.sequence
...     s1.identifier == s2.identifier
True
True
True
True
True
True
True
True
```

`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.

3.2.4 utility Package

`utility` Package

`archive` Module

Tools for extracting files from common archive formats like .tar.gz and .zip.

Additionally includes helper functions to extract common sequence formats from these archives without actually 'inflating' the archive on the disk.

class `tigerlily.utility.archive.`**`Archive`**(*filepath=None*, *data=None*)
Bases: `builtins.object`

Common interface for extracting file objects from common archive formats.

When initializing, you must specify either *filepath* or *data* but not both. In general, prefer to use *filepath* as it prevents using a layer of abstraction to provide a file-like object to the underlying archive formats, but either will work more or less equivalently.

The format of the archive will be automatically detected without using the file name or extension.

This interface is **read-only**, and no plans to add file writing exist at this time.

Currently supported formats are (again, recall that extensions may be arbitrary): .tar .tar.gz .tar.bz2 .zip

In all cases the archive may contain one file or many files. If the archive has a nested folder structure, this structure will be ignored and all file members will be scanned without regard to their placement in the arhcive's folder structure.

`getfasta`()
Generate every member of the archive that looks like it is a FASTA file as a file-like object.

`getmember`(*name*)
Open the given member as a file-like object.

`getmembers`()
Generate every member of the archive as a file-like object.

`getnames`()
Return a list of the names of every member in the archive.

These names will be suitable for passing to the getmember() function, but their explicit type is not specified.

`getnofasta`()
Generate every member of the archive that does NOT look like it is a FASTA file as a file-like object.

`archive_test` Module

This module provides unit tests for the `tigerlily.utility.archive` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `tigerlily.utility.archive_test.`**`ArchiveTests`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Test harness for `tigerlily.utility.archive.Archive` class.

`setUp`()

archive.py: Create the testing environment

`test_tar`()

archive.py: Test .tar archive support

`test_tarbz2`()

archive.py: Test .tar.bz2 archive support

`test_targz`()

archive.py: Test .tar.gz archive support

`test_zip`()

archive.py: Test .zip archive support

`download` Module

Tools for downloading resources from URLs with a variety of 'fancy' interfaces (such as progressbars and login info for HTTP authentication).

class `tigerlily.utility.download.`**`ConsoleDownloader`**(**args*, ***kwargs*)

Bases: `urllib.request.FancyURLopener`

Class which provides an interface to download URLs in a console environment. This may be interactive and have verbose status messages, or alternately supports a silent non-interactive mode.

`retrieve`(*url*, *filename=None*, *silent=False*, ***kwargs*)

Wrapper for `urllib.request.URLopener.retrieve`.

If *silent* is left as False, a status message will be printed to the console informing the user of the progress on the file download.

***kwargs* will be passed to `urllib.request.URLopener.retrieve`, but please do not specify either `'url'`, `'filename'`, or `'reporthook'` as those values are provided by this function. As of this writing, this leaves only `'data'` as an extra argument to specify in ***kwargs*.

A helper function, `make_filename`, has been provided in this module to assist in creating filenames - see its documentation for further information.

This function returns a tuple `(filename, headers)` as per the documentation given in `urllib.request.URLopener.retrieve`.

`tigerlily.utility.download.`**`make_filename`**(*name=None*, *dir=None*, *makedirs=False*, *overwrite=False*)

Return (or create) a complete file path, optionally creating directories.

name will be the name of the file created, irrespective of (and not including) the directory path that will contain the file. If left as `None`, a randomly generated file name will be chosen in the directory.

dir will be the directory in which *name* is created. If left as `None`, the current working directory is used.

If the directory structure that will contain *name* does not exist, EnvironmentError will be raised. This behavior can be surpressed by enabling *makedirs*, in which case the necessary folders will be created.

If the resulting filepath already exists, EnvironmentError will be raised. This behaior can be surpressed by enabling *overwrite*, which will simpy return the filepath as generated (which would generally cause the file to be overwritten, depending on what the caller uses the path for.)

`download_test` Module

This module provides unit tests for the `tigerlily.utility.download` module.

As with all unit test modules, the tests it contains can be executed in many ways, but most easily by going to the project root dir and executing `python3 setup.py nosetests`.

class `tigerlily.utility.download_test.`**`ConsoleDownloaderTests`**`(`*methodName='runTest'*`)`

Bases: `unittest.case.TestCase`

Test harness for `tigerlily.utility.download.ConsoleDownloader` class.

`setUp``()`

Create the testing environment

`tearDown``()`

Remove the testing environment

`test_local``()`

download.py: Test 'downloading' local resources.

`test_make_filename``()`

download.py: Test directory structure creation from make_filename

`test_remote_open``()`

download.py: Test the Downloader.open() method

`test_remote_retrieve``()`

download.py: Test the Downloader.retrieve() method

`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__, 7
tigerlily.grc, 7
tigerlily.grc.genome, 7
tigerlily.index.fixedtree, 9
tigerlily.index.fixedtree_test, 10
tigerlily.index.index, 10
tigerlily.sequences, 11
tigerlily.sequences.fasta, 11
tigerlily.sequences.genomic, 13
tigerlily.sequences.raw, 16
tigerlily.sequences.sequence, 17
tigerlily.utility, 18
tigerlily.utility.archive, 18
tigerlily.utility.archive_test, 19
tigerlily.utility.download, 19
tigerlily.utility.download_test, 20
tigerlily.utility.string_relations, 20

INDEX

A

add_sequence() (tigerlily.index.fixedtree.FixedTree method), 9
alignments() (tigerlily.index.fixedtree.FixedTree method), 9
alignments() (tigerlily.index.fixedtree.FixedTreeNode method), 10
alignments() (tigerlily.index.index.GroupIndex method), 10
ALLOWED_AMINO_CHARS (tigerlily.sequences.fasta.FASTASequence attribute), 11
ALLOWED_NUCLEIC_CHARS (tigerlily.sequences.fasta.FASTASequence attribute), 11
AminoSequence (class in tigerlily.sequences.genomic), 13
Archive (class in tigerlily.utility.archive), 18
ArchiveTests (class in tigerlily.utility.archive_test), 19

C

complement() (tigerlily.sequences.genomic.NucleicSequence method), 14
ConsoleDownloader (class in tigerlily.utility.download), 19
ConsoleDownloaderTests (class in tigerlily.utility.download_test), 20
convert() (tigerlily.sequences.sequence.PolymerSequence method), 17

D

download() (tigerlily.grc.genome.GRCGenome class method), 7

F

FASTASequence (class in tigerlily.sequences.fasta), 11
FixedTree (class in tigerlily.index.fixedtree), 9
FixedTreeNode (class in tigerlily.index.fixedtree), 9
FixedTreeTests (class in tigerlily.index.fixedtree_test), 10
format() (tigerlily.sequences.sequence.FormattedSequence method), 17
FormattedSequence (class in tigerlily.sequences.sequence), 17

G

getfasta() (tigerlily.utility.archive.Archive method), 18
getmember() (tigerlily.utility.archive.Archive method), 18
getmembers() (tigerlily.utility.archive.Archive method), 18
getnames() (tigerlily.utility.archive.Archive method), 18
getnofasta() (tigerlily.utility.archive.Archive method), 18
GRCGenome (class in tigerlily.grc.genome), 7
greatest_common_prefix() (in module tigerlily.utility.string_relations), 20
GroupIndex (class in tigerlily.index.index), 10

H

hamming_distance() (in module tigerlily.utility.string_relations), 20

I

identifier (tigerlily.sequences.fasta.FASTASequence attribute), 11
identifier (tigerlily.sequences.genomic.AminoSequence attribute), 13
identifier (tigerlily.sequences.genomic.NucleicSequence attribute), 14
identifier (tigerlily.sequences.raw.RawSequence attribute), 16
identifier (tigerlily.sequences.sequence.PolymerSequence attribute), 17
insert() (tigerlily.index.fixedtree.FixedTreeNode method), 10

L

levenshtein_distance() (in module tigerlily.utility.string_relations), 21
load() (tigerlily.grc.genome.GRCGenome class method), 8
load() (tigerlily.index.fixedtree.FixedTree class method), 9
load() (tigerlily.index.fixedtree.FixedTreeNode class method), 10

load_archive() (tigerlily.grc.genome.GRCGenome class method), 8

M

make_filename() (in module tigerlily.utility.download), 19
MAX_LINE_WIDTH (tigerlily.sequences.fasta.FASTASequence attribute), 11

N

NucleicSequence (class in tigerlily.sequences.genomic), 13

P

parseFASTA() (in module tigerlily.sequences.fasta), 12
parseRaw() (in module tigerlily.sequences.raw), 16
PolymerSequence (class in tigerlily.sequences.sequence), 17

R

RawSequence (class in tigerlily.sequences.raw), 16
retrieve() (tigerlily.utility.download.ConsoleDownloader method), 19
reverse() (tigerlily.sequences.genomic.NucleicSequence method), 14
reverse_complement() (in module tigerlily.sequences.genomic), 15
reverse_complement() (tigerlily.sequences.genomic.NucleicSequence method), 14

S

sequence (tigerlily.sequences.fasta.FASTASequence attribute), 12
sequence (tigerlily.sequences.genomic.AminoSequence attribute), 13
sequence (tigerlily.sequences.genomic.NucleicSequence attribute), 15
sequence (tigerlily.sequences.raw.RawSequence attribute), 16
sequence (tigerlily.sequences.sequence.PolymerSequence attribute), 18
sequences() (tigerlily.grc.genome.GRCGenome method), 8
setUp() (tigerlily.index.fixedtree_test.FixedTreeTests method), 10
setUp() (tigerlily.utility.archive_test.ArchiveTests method), 19
setUp() (tigerlily.utility.download_test.ConsoleDownloaderTests method), 20
store() (tigerlily.index.fixedtree.FixedTree method), 9
store() (tigerlily.index.fixedtree.FixedTreeNode method), 10

T

tearDown() (tigerlily.index.fixedtree_test.FixedTreeTests method), 10
tearDown() (tigerlily.utility.download_test.ConsoleDownloaderTests method), 20
test_create_index() (tigerlily.index.fixedtree_test.FixedTreeTests method), 10
test_local() (tigerlily.utility.download_test.ConsoleDownloaderTests method), 20
test_make_filename() (tigerlily.utility.download_test.ConsoleDownloaderTe method), 20
test_remote_open() (tigerlily.utility.download_test.ConsoleDownloaderTest method), 20
test_remote_retrieve() (tigerlily.utility.download_test.ConsoleDownloaderT method), 20
test_store_index() (tigerlily.index.fixedtree_test.FixedTreeTests method), 10
test_tar() (tigerlily.utility.archive_test.ArchiveTests method), 19
test_tarbz2() (tigerlily.utility.archive_test.ArchiveTests method), 19
test_targz() (tigerlily.utility.archive_test.ArchiveTests method), 19
test_zip() (tigerlily.utility.archive_test.ArchiveTests method), 19
tigerlily.__init__ (module), 7
tigerlily.grc (module), 7
tigerlily.grc.genome (module), 7
tigerlily.index.fixedtree (module), 9
tigerlily.index.fixedtree_test (module), 10
tigerlily.index.index (module), 10
tigerlily.sequences (module), 11
tigerlily.sequences.fasta (module), 11
tigerlily.sequences.genomic (module), 13
tigerlily.sequences.raw (module), 16
tigerlily.sequences.sequence (module), 17
tigerlily.utility (module), 18
tigerlily.utility.archive (module), 18
tigerlily.utility.archive_test (module), 19
tigerlily.utility.download (module), 19
tigerlily.utility.download_test (module), 20
tigerlily.utility.string_relations (module), 20
translate() (tigerlily.sequences.genomic.NucleicSequence method), 15
translations() (tigerlily.sequences.genomic.AminoSequence method), 13

W

write() (tigerlily.sequences.fasta.FASTASequence method), 12
write() (tigerlily.sequences.raw.RawSequence method), 16
write() (tigerlily.sequences.sequence.FormattedSequence method), 17

writeFASTA() (in module tigerlily.sequences.fasta), 12