Genomedata Documentation

Release 1.3.2

Michael M. Hoffman

March 26, 2012

CONTENTS

Contents:

CHAPTER

ONE

GENOMEDATA 1.3 DOCUMENTATION

Website http://noble.gs.washington.edu/proj/genomedata

Author Michael M. Hoffman <mmh1 at uw dot edu>

Organization University of Washington

Address Department of Genome Sciences, PO Box 355065, Seattle, WA 98195-5065, United States of America

For a broad overview, see the paper:

Hoffman MM, Buske OJ, Noble WS. (2010). **'The Genomedata format for storing large-scale functional genomics data'__**. *Bioinformatics*, **26**(11):1458-1459; doi:10.1093/bioinformatics/btq164

Please cite this paper if you use Genomedata.

1.1 Installation

A simple, interactive script has been created to install Genomedata (and most dependencies) on any Unix platform. Installation is as simple as downloading and running this script! For instance:

```
wget http://noble.gs.washington.edu/proj/genomedata/install.py
python install.py
```

Note: The following are prerequisites:

- **Linux/Unix** This software has been tested on Linux and Mac OS X systems. We would love to add support for other systems in the future and will gladly accept any contributions toward this end.
- Python 2.5-2.7
- Zlib

Note: For questions, comments, or troubleshooting, please refer to the support section.

1.2 Overview

Genomedata provides a way to store and access large-scale functional genomics data in a format which is both space-efficient and allows efficient random-access. Genomedata archives are currently write-once, although we are working to fix this.

Under the surface, Genomedata is *implemented* as one or more HDF5 files, but Genomedata provides a transparent interface to interact with your underlying data without having to worry about the mess of repeatedly parsing large data files or having to keep them in memory for random access.

The Genomedata hierarchy:

Each `Genome` contains many `Chromosomes`

Each `Chromosome` contains many `Supercontigs`

Each `Supercontig` contains one `continuous` data set Each `continuous` data set is a numpy.array of floating point numbers with a column for each data track and a row for each base in the data set.

Why have `Supercontigs`? Genomic data seldom covers the entire genome but instead tends to be defined in large but scattered regions. In order to avoid storing the undefined data between the regions, chromosomes are divided into separate supercontigs when regions of defined data are far enough apart. They also serve as a convenient chunk since they can usually fit entirely in memory.

1.3 Implementation

Genomedata archives are implemented as one or more HDF5 files. The *API* handles both single-file and directory archives transparently, but the implementation options exist for several performance reasons.

Use a directory with few chromosomes/scaffolds:

- Parallel load/access
- Smaller file sizes

Use a single file with many chromosomes/scaffolds:

- More efficient access with many chromosomes/scaffolds
- Easier archive distribution

Implementing the archive as a directory makes it easier to parallelize access to the data. In particular, it makes it easy to create the archives in parallel with one chromosome on each machine. It also reduces the likelihood of running into the 2 GB file limit applicable to older applications and older versions of 32-bit UNIX. We are currently using an 81-track Genomedata archive for our research which has a total size of 18 GB, but the largest single file (chr1) is only 1.6 GB.

A directory-based Genomedata archive is not ideal for all circumstances, however, such as when working with genomes with many chromosomes, contigs, or scaffolds. In these situations, a single file implementation would be much more efficient. Additionally, having the archive as a single file allows the archive to be distributed much more easily (without tar/zip/etc).

Note: The default behavior is to implement the Genomedata archive as a directory if there are fewer than 100 sequences being loaded and as a single file otherwise.

New in version 1.1: Single-file-based Genomedata archives

1.4 Creation

A Genomedata archive contains sequence and may also contain numerical data associated with that sequence. You can easily load sequence and numerical data into a Genomedata archive with the *genomedata-load* command (see command details additional details):

```
genomedata-load [-t trackname=signalfile]... [-s sequencefile]... GENOMEDATAFILE
```

This command is a user-friendly shortcut to the typical workflow. The underlying commands are still installed and may be used if more fine-grained control is required (for instance, parallel data loading or adding additional tracks later). The commands and required ordering are:

1. *genomedata-load-assembly*
2. *genomedata-open-data*
3. *genomedata-load-data*
4. *genomedata-close-data*

Entire data tracks can later be replaced with the following pipeline:

1. *genomedata-erase-data*
2. *genomedata-load-data*
3. *genomedata-close-data*

New in version 1.1: The ability to replace data tracks. Additional data tracks can be added to an existing archive with the following pipeline:

1. *genomedata-open-data*
2. *genomedata-load-data*
3. *genomedata-close-data*

New in version 1.2: The ability to add data tracks. As of the current version, Genomedata archives must include the underlying genomic sequence and can only be created with *genomedata-load-assembly*. A Genomedata archive can be created without any tracks, however, using the following pipeline:

1. *genomedata-load-assembly*
2. *genomedata-close-data*

New in version 1.2: The ability to create an archive without any data tracks.

Note: A call to **h5repack** after *genomedata-close-data* may be used to transparently compress the data.

1.4.1 Example

The following is a brief example for creating a Genomedata archive from sequence and signal files.

Given the following two sequence files:

1. chr1.fa:

```
>chr1
taaccctaaccctaaccctaaccctaaccctaaccctaaccctaacccta
accctaaccctaaccctaaccctaaccct
```

2. chrY.fa.gz:

```
>chrY
ctaaccctaaccctaaccctaaccctaaccctaaccctCTGaaagtggac
```

and the following two signal files:

1. signal_low.wigFix:

```
fixedStep chrom=chr1 start=5 step=1
0.372
-2.540
0.371
-2.611
0.372
-2.320
```

2. signal_high.bed.gz:

```
chrY    0       12      4.67
chrY    20      23      9.24
chr1    1       3       2.71
chr1    3       6       1.61
chr1    6       24      3.14
```

A Genomedata archive (`genomedata.test`) could then be created with the following command:

```
genomedata-load -s chr1.fa -s chrY.fa.gz -t low=signal_low.wigFix \
    -t high=signal_high.bed.gz genomedata.test
```

or the following pipeline:

```
genomedata-load-assembly genomedata.test chr1.fa chrY.fa.gz
genomedata-open-data genomedata.test low high
genomedata-load-data genomedata.test low < signal_low.wigFix
zcat signal_high.bed.gz | genomedata-load-data genomedata.test high
genomedata-close-data genomedata.test
```

Note: chr1.fa and chrY.fa.gz could also be combined into a single sequence file with two sequences.

Warning: It is important that the sequence names (*chrY*, *chr1*) in the signal files match the sequence identifiers in the sequence files exactly.

1.5 Genomedata usage

1.5.1 Python interface

The data in Genomedata is accessed through the hierarchy described in *Overview*. A full *Python API* is also available. To appreciate the full benefit of Genomedata, it is most easily used as a contextmanager:

```
from genomedata import Genome
[...]
gdfilename = "/path/to/genomedata/archive"
with Genome(gdfilename) as genome:
    [...]
```

Note: If Genome is used as a context manager, it will clean up any opened Chromosomes automatically. If not, the Genome object (and all opened chromosomes) should be closed manually with a call to `Genome.close()`.

Basic usage

Genomedata is designed to make it easy to get to the data you want. Here are a few examples:

Get arbitrary sequence (10-bp sequence starting at chr2:1423):

```
>>> chromosome = genome["chr2"]
>>> seq = chromosome.seq[1423:1433]
>>> seq
array([116,  99,  99,  99,  99, 103, 103, 103, 103, 103], dtype=uint8)
>>> seq.tostring()
'tccccggggg'
```

Get arbitrary data (data from first 3 tracks for region chr8:999-1000):

```
>>> chromosome = genome["chr8"]
>>> chromosome[999:1001, 0:3]  # Note the half-open, zero-based indexing
array([[ NaN,  NaN,  NaN],
       [ 3. ,  5.5,  3.5]], dtype=float32)
```

Get data for a specific track (specified data in first 5-bp of chr1):

```
>>> chromosome = genome["chr1"]
>>> data = chromosome[0:5, "sample_track"]
>>> data
array([ 47.,  NaN,  NaN,  NaN,  NaN], dtype=float32)
```

Only specified data:

```
>>> from numpy import isfinite
>>> data[isfinite(data)]
array([ 47.], dtype=float32)
```

Note: Specify a slice for the track to keep the data in column form:

```
>>> col_index = chromosome.index_continuous("sample_track")
>>> data = chromosome[0:5, col_index:col_index+1]
```

1.5.2 Command-line interface

Genomedata archives can be created and loaded from the command line with the *genomedata-load* command.

genomedata-load

This is a convenience script that will do everything necessary to create a Genomedata archive. This script takes as input:

- assembly files in either FASTA (`.fa` or `.fa.gz`) format (where the sequence identifiers are the names of the chromosomes/scaffolds to create), or assembly files in AGP format (when used with *`--assembly`*). This is **mandatory**, despite having an option interface.

- **trackname, datafile pairs (specified as `trackname=datafile`), where:**
  - trackname is a `string` identifier (e.g. `broad.h3k27me3`)
  - datafile contains signal data for this data track in one of the following formats: WIG, BED, bedGraph, or a gzip'd form of any of the preceding
  - the chromosomes/scaffolds referred to in the datafile MUST be identical to those found in the sequence files
- the name of the Genomedata archive to create

See the *full example* for more details.

Command-line usage information:

```
Usage: genomedata-load [OPTIONS] GENOMEDATAFILE

--track and --sequence may be repeated to specify multiple trackname=trackfile
pairings and sequence files, respectively

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -s SEQFILE, --sequence=SEQFILE
                        Add the sequence data in the specified file
  -t TRACK, --track=TRACK
                        Add data for the given track. TRACK should be
                        specified in the form: NAME=FILE, such as: -t
                        signal=signal.dat
```

Alternately, as described in *Overview*, the underlying Python and C load scripts are also accessible for more finely-grained control. This can be especially useful for parallelizing Genomedata loading over a cluster.

You can use wildcards when specifying sequence files, such as in `genomedata-load-assembly -s 'chr*.fa'`. You must be sure to quote the wildcards so that they are not expanded by your shell. For most shells, this means using single quotes (`'chr*.fa'`) instead of double quotes (`"chr*.fa"`).

If you aren't going to use the sequence later on, loading the assembly from an AGP file will be faster and take less memory during loading, and disk space afterward.

genomedata-load-assembly

This command adds the provided sequence files to the specified Genomedata, archive creating it if it does not already exist. Sequence files should be in FASTA (`.fa` or `.fa.gz`) format. Gaps of >= 100,000 base pairs (specified as *`gap-length`*) in the reference sequence, are used to divide the sequence into supercontigs. The FASTA definition line will be used as the name for the chromosomes/scaffolds created within the Genomedata archive and must be consistent between these sequence files and the data loaded later with *genomedata-load-data*. See *this example* for details.

```
Usage: genomedata-load-assembly [OPTION]... GENOMEDATAFILE SEQFILE...

Options:
  -g, --gap-length  XXX: Implement this.
  --version         show program's version number and exit
  -h, --help        show this help message and exit
```

genomedata-open-data

This command opens the specified tracks in the Genomedata archive, allowing data for those tracks to be loaded with *genomedata-load-data*.

```
Usage: genomedata-open-data [OPTION]... GENOMEDATAFILE TRACKNAME...

Options:
  --version   show program's version number and exit
  -h, --help  show this help message and exit
```

genomedata-load-data

This command loads data from stdin into Genomedata under the given trackname. The input data must be in one of these supported datatypes: WIG, BED, bedGraph. The chromosome/scaffold references in these files must match the sequence identifiers in the sequence files loaded with *genomedata-load-assembly*. See *this example* for details. A `chunk-size` can be specified to control the size of hdf5 chunks (the smallest data read size, like a page size). Larger values of `chunk-size` can increase the level of compression, but they also increase the minimum amount of data that must be read to access a single value.

```
Usage: genomedata-load-data [OPTION...] GENOMEDATAFILE TRACKNAME
Loads data into Genomedata format
Takes track data in on stdin

  -c, --chunk-size=NROWS     Chunk hdf5 data into blocks of NROWS. A higher
                             value increases compression but slows random
                             access. Must always be smaller than the max size
                             for a dataset. [default: 10000]
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

genomedata-close-data

Closes the specified Genomedata arhive.

```
Usage: genomedata-close-data [OPTION]... GENOMEDATAFILE

Options:
  --version   show program's version number and exit
  -h, --help  show this help message and exit
```

genomedata-erase-data

Erases all data associated with the specified tracks, allowing the data to then be replaced. The pipeline for replacing a data track is:

1. *genomedata-erase-data*
2. *genomedata-load-data*
3. *genomedata-close-data*

```
Usage: genomedata-erase-data [OPTION]... GENOMEDATAFILE TRACKNAME...

Erase the specified tracks from the Genomedata archive in such a way that
the track can be replaced (via genomedata-load-data).

Options:
  --version      show program's version number and exit
  -h, --help     show this help message and exit
  -v, --verbose  Print status updates and diagnostic messages
```

1.5.3 Python API

The Genomedata package is designed to be used from a variety of scripting languages, but currently only exports the following Python API.

class genomedata.**Genome** (*filename*, **args*, ***kwargs*)

The root level of the genomedata object hierarchy.

If you use this as a context manager, it will keep track of any open Chromosomes and close them (and the Genome object) for you later when the context is left:

```
with Genome("/path/to/genomedata") as genome:
  chromosome = genome["chr1"]
  [...]
```

If not used as a context manager, you are responsible for closing the Genomedata archive once you are done:

```
>>> genome = Genome("/path/to/genomedata")
>>> chromosome = genome["chr1"]
[...]
>>> genome.close()
```

__init__ (*filename*, **args*, ***kwargs*)

Create a Genome object from a genomdata archive.

Parameters

- **filename** (*string*) – the root of the Genomedata object hierarchy. This can either be a .genomedata file that contains the entire genome or a directory containing multiple chromosome files.
- ***args** – args passed on to openFile if single file or to Chromosome if directory
- ****kwargs** – keyword args passed on to openFile if single file or to Chromosome if directory

Example:

```
>>> genome = Genome("./genomedata.ctcf.pol2b/")
>>> genome
Genome("./genomedata.ctcf.pol2b/")
    [...]
>>> genome.close()
>>> genome = Genome("./cat_chipseq.genomedata", mode="r")
    [...]
>>> genome.close()
```

__iter__()

Return next chromosome, in sorted order, with memoization.

Example:

```
for chromosome in genome:
  print chromosome.name
  for supercontig, continuous in chromosome.itercontinuous():
    [...]
```

__getitem__(*name*)
Return a reference to a chromosome of the given name.

Parameters **name** (*string*) – name of the chromosome (e.g. "chr1" if chr1.genomedata is a file in the Genomedata archive or chr1 is a top-level group in the single-file Genomedata archive)

Returns Chromosome

Example:

```
>>> genome["chrX"]
<Chromosome 'chrX', file='/path/to/genomedata/chrX.genomedata'>
>>> genome["chrZ"]
KeyError: 'Could not find chromosome: chrZ'
```

add_track_continuous(*trackname*)
Add a new track

The Genome object must have been created with :param mode:="r+". Behavior is undefined if this is not the case.

Currently sets the dirty bit, which can only be erased with genomedata-close-data

close()
Close this Genomedata archive and any open chromosomes

If the Genomedata archive is a directory, this closes all open chromosomes. If it is a single file, this closes that file. This should only be used if Genome is not a context manager (see Genome). The behavior is undefined if this is called while Genome is being used as a context manager.

erase_data(*trackname*)
Erase all data for the given track across all chromosomes

The Genome object must have been created with :param mode:="r+". Behavior is undefined if this is not the case.

Currently sets the dirty bit, which can only be erased with genomedata-close-data

format_version
Genomedata format version

None means there are no chromosomes in it already.

index_continuous(*trackname*)
Return the column index of the trackname in the continuous data.

Parameters **trackname** (*string*) – name of data track

Returns integer

This is used for efficient indexing into continuous data:

```
>>> col_index = genome.index_continuous("sample_track")
>>> data = genome["chr3"][100:150, col_index]
```

although for typical use, the track can be indexed directly:

```
>>> data = genome["chr3"][100:150, "sample_track"]
```

isopen
Return a boolean indicating if the Genome is still open

maxs
Return a vector of the maximum value for each track.

Returns numpy.array

means
Return a vector of the mean value of each track.

Returns numpy.array

mins
Return the minimum value for each track.

Returns numpy.array

num_datapoints
Return the number of datapoints in each track.

Returns a numpy.array vector with an entry for each track.

num_tracks_continuous
Returns the number of continuous data tracks.

sums
Return a vector of the sum of the values for each track.

Returns numpy.array

sums_squares
Return a vector of the sum of squared values for each track's data.

Returns numpy.array

tracknames_continuous
Return a list of the names of all data tracks stored.

vars
Return a vector of the variance in the data for each track.

Returns numpy.array

class genomedata.**Chromosome**(*h5file*, *where='/'*, *name=None*)
The Genomedata object corresponding to data for a given chromosome.

Usually created by keying into a Genome object with the name of a chromosome, as in:

```
>>> with Genome("/path/to/genomedata") as genome:
...     chromosome = genome["chrX"]
...     chromosome
...
<Chromosome 'chrX', file='/path/to/genomedata/chrX.genomedata'>
```

__iter__()
Return next supercontig in chromosome. New in version 1.2: Supercontigs are ordered by start index Seldom used in favor of the more direct: `Chromosome.itercontinuous()`

Example:

```
>>> for supercontig in chromosome:
...     supercontig  # calls repr()
...
<Supercontig 'supercontig_0', [0:66115833]>
<Supercontig 'supercontig_1', [66375833:90587544]>
<Supercontig 'supercontig_2', [94987544:199501827]>
```

__getitem__(*key*)

Return the continuous data corresponding to this bp slice

Parameters **key** – base_key must index or slice bases track_key specify data tracks with index, slice, string, list of strings, list of indexes, or array of indexes

but can also index, slice, or directly specify (string or list of strings) the data tracks.

Returns numpy.array

If slice is taken over or outside a supercontig boundary, missing data is filled in with NaN's automatically and a warning is printed.

Typical use:

```
>>> chromosome = genome["chr4"]
>>> chromosome[0:5]  # Get all data for the first five bases of chr4
>>> chromosome[0, 0:2]  # Get data for first two tracks at chr4:0
>>> chromosome[100, "ctcf"]  # Get "ctcf" track value at chr4:100
```

exception ChromosomeDirtyError

Chromosome.**attrs**

Return the attributes for this Chromosome.

This may also include Genome-wide attributes if the archive is implemented as a directory.

Chromosome.**close**()

Close the current chromosome file.

This only needs to be called when Genomedata files are manually opened as Chromosomes. Otherwise, `Genome.close()` should be called to close any open chromosomes or Genomedata files. The behavior is undefined if this is called on a Chromosome accessed through a Genome object. Using Genomedata as a context manager makes life easy by memoizing chromosome access and guaranteeing the proper cleanup. See `Genome`.

Chromosome.**end**

Return the index past the last base in this chromosome.

For `Genome.format_version` > 0, this will be the number of bases of sequence in the chromosome. For == 0, this will be the end of the last supercontig.

This is the end in half-open coordinates, making slicing simple:

```
>>> chromosome.seq[chromosome.start:chromosome.end]
```

Chromosome.**index_continuous**(*trackname*)

Return the column index of the trackname in the continuous data.

Parameters **trackname** (*string*) – name of data track

Returns integer

This is used for efficient indexing into continuous data:

```
>>> chromosome = genome["chr3"]
>>> col_index = chromosome.index_continuous("sample_track")
>>> data = chromosome[100:150, col_index]
```

although for typical use, the track can be indexed directly:

```
>>> data = chromosome[100:150, "sample_track"]
```

Chromosome.**isopen**
Return a boolean indicating if the Chromosome is still open

Chromosome.**itercontinuous**()
Return a generator over all supercontig, continuous pairs. New in version 1.2: Supercontigs are ordered by increasing supercontig.start. This is the best way to efficiently iterate over the data since all specified data is in supercontigs:

```
for supercontig, continuous in chromosome.itercontinuous():
    print supercontig, supercontig.start, supercontig.end
    [...]
```

Chromosome.**maxs**
See Genome.maxs

Chromosome.**mins**
See Genome.mins

Chromosome.**name**
Return the name of this chromosome (same as __str__()).

Chromosome.**num_datapoints**
See Genome.num_datapoints

Chromosome.**num_tracks_continuous**
Return the number of tracks in this chromosome

Chromosome.**seq**
Return the genomic sequence of this chromosome.

If the index or slice spans a non-supercontig range, N's are inserted in place of the missing data and a warning is issued.

Example:

```
>>> chromosome = genome["chr1"]
>>> for supercontig in chromosome:
...     print repr(supercontig)
...
<Supercontig 'supercontig_0', [0:121186957]>
<Supercontig 'supercontig_1', [141476957:143422081]>
<Supercontig 'supercontig_2', [143522081:247249719]>
>>> chromosome.seq[0:10].tostring()  # Inside supercontig
'taaccctaac'
>>> chromosome.seq[121186950:121186970].tostring() # Supercontig boundary
'agAATTCNNNNNNNNNNNNN'
>>> chromosome.seq[121186957:121186960].tostring() # Not in supercontig
UserWarning: slice of chromosome sequence does not overlap any supercontig (filling with 'N'
'NNN'
```

The entire sequence for a chromosome can be retrieved with:

```
>>> chromosome.seq[chromosome.start:chromosome.end]
```

Chromosome.**start**
Return the index of the first base in this chromosome.

For Genome.format_version > 0, this will always be 0. For == 0, this will be the start of the first supercontig.

Chromosome.**sums**
See Genome.sums

Chromosome.**sums_squares**
See Genome.sums_squares

Chromosome.**supercontigs**
Return the supercontig that contains this range if possible.

Returns Supercontig

Indexable with a slice or simple index:

```
>>> chromosome.supercontigs[100]
[<Supercontig 'supercontig_0', [0:66115833]>]
>>> chromosome.supercontigs[1:100000000]
[<Supercontig 'supercontig_0', [0:66115833]>, <Supercontig 'supercontig_1', [66375833:905875
>>> chromosome.supercontigs[66115833:66375833]  # Between two supercontigs
[]
```

Chromosome.**tracknames_continuous**
Return a list of the data track names in this Chromosome.

class genomedata.**Supercontig**(*h5group*)
A container for a segment of data in one chromosome.

Implemented via a HDF5 Group

attrs
Return the attributes of this supercontig.

continuous
Return the underlying continuous data in this supercontig. To read the whole dataset into memory as a *numpy.array*, use continuous.read()

Returns *tables.EArray*

end
Return the index past the last base in this supercontig.

This is the end in half-open coordinates, making slicing simpler:

```
>>> supercontig.seq[supercontig.start:supercontig:end]
```

name
Return the name of this supercontig.

project(*pos*, *bound=False*)
Project chromsomal coordinates to supercontig coordinates.

Parameters

- **pos** (*integer*) – chromosome coordinate
- **bound** (*boolean*) – bound result to valid supercontig coordinates

Returns integer

seq
See `Chromosome.seq`.

start
Return the index of the first base in this supercontig.

The first base is index 0.

1.6 Tips and tricks

If you find yourself creating many Genomedata archives on the same genome, it might be useful to save a copy of an archive after you load sequence, but before you load any data. Obviously, you can only do this if you use the fine-grained workflow of *genomedata-load-assembly*, *genomedata-open-data*, *genomedata-load-data*, and *genomedata-close-data*.

1.7 Technical matters

1.7.1 Chunking and chunk cache overhead

Genomedata uses an HDF5 data store. The data is stored in chunks. The chunk size is 10,000 bp and one data track of 32-bit single-precision floats, which makes the chunk 40 kB. Each chunk is gzip compressed so on disk it will be smaller. To read a single position you have to read its entire chunk off of the disk and then decompress it. There is a tradeoff here between latency and throughput. Larger chunk sizes mean more latency but better throughput and better compression.

The only disk storage overhead is that compression is slightly less efficient than compressing the whole binary data file when you break it into chunks. This is far outweighed by the efficient random access capability. If you have different needs, then it should be possible to change the chunk shape (`genomedata.CONTINUOUS_CHUNK_SHAPE`) or compression method (`genomedata._util.FILTERS_GZIP`).

The memory overhead is dominated by the chunk cache defined by PyTables. On the version of PyTables we use, this is 2 MiB. You can change this by setting **'tables.parameters.CHUNK_CACHE_SIZE'__**.

:: __: http://www.pytables.org/docs/manual/apc.html

1.8 Bugs

There is currently an interaction between Genomedata and PyTables that can result in the emission of Performance-Warnings when a Genomedata file is opened. These can be ignored. We would like to fix these at some point.

1.9 Support

To stay informed of **new releases**, subscribe to the moderated `genomedata-announce` mailing list (mail volume very low):

https://mailman1.u.washington.edu/mailman/listinfo/genomedata-announce

For **discussion and questions** about the use of the Genomedata system, there is a `genomedata-users` mailing list:

https://mailman1.u.washington.edu/mailman/listinfo/genomedata-users

For issues related to the use of Genomedata on **Mac OS X**, please use the above mailing list or contact Jay Hesselberth <jay dot hesselberth at ucdenver dot edu>.

If you want to **report a bug or request a feature**, please do so using our issue tracker:

http://code.google.com/p/genomedata/issues

For other support with Genomedata, or to provide feedback, please write contact the authors directly. We are interested in all comments regarding the package and the ease of use of installation and documentation.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

Symbols

A

C

E

F

G

I

M

N

P

S

T

V