Genepidgin Documentation

Release 1.2.0

Clint Howarth

February 04, 2013

CONTENTS

Genepidgin is a suite of tools that assist in evaluation and assignment gene product names. There are three primary components:

Genepidgin cleaner standardizes gene names per UNIPROT naming guidelines

Genepidgin compare compares two or more sets of gene names

Genepidgin select selects the most appropriate product name from a vareity of homology evidence

`genepidgin` is developed and lightly maintained by engineers and biologists at the Broad Institute.

CHAPTER

ONE

DEVELOPMENT STATUS

Warning: *This code is in maintenance mode only, and there are better ways of doing this.* When we started this project, well-defined ontology sets were uncertain. There are enough around now that this approach is relatively antiquated. Nowadays, you're almost certainly better off with EC lookups, go-terms and similar, more direct methods.

CHAPTER

TWO

CONTENTS

2.1 Installation

Genepidgin requires *Python 2.5+*. Python comes with easy_install.

(If you don't have `pip`, prepend the following instructions with: `easy_install pip`)

```
pip install genepidgin
```

For more instructions on subcommands, visit *Genepidgin cleaner*, *Genepidgin compare*, or *Genepidgin select*.

2.1.1 Testing

Development and testing requires the nose package. To execute the unit tests, `pip install nose` and then execute:

```
nosetests
```

2.2 Genepidgin *cleaner*

Note: this logic in particular is from a different era of computes, and you'd almost certainly be better off with a homology-derived name against a tightly-governed protein library than a loose alignment against a less controlled one. Use GO.

Genepidgin *cleaner* standardizes the format of gene product names derived from diverse databases, including FIGfam, KEGG, Pfam, RefSeq, SwissProt and TIGRFAM. It's the product of many years of production genome annotation.

This software package consists of a large collection of heuristics, formatting rules and regular expressions which are designed to take a name from any of Genepidgin's supported databases and present it in a common style. Though our regexp library is large, it is not infinite; thus, Genepidgin *cleaner* cannot detect every possible name error. However, the vast majority of source names end up better and more informative for having gone through Genepidgin *cleaner*.

2.2.1 Goals

- Names should agree with the prevailing conventions in cases where such conventions can be easily identified and agreed upon.
- Names should be as clear and concise as possible.

- Names should not be descriptive phrases that define function (for example, "protein involved in folding" is not useful, but "chaperonin" is).
- Names should not include programmatic references.
- Names should be derived from high-confidence alignments to homologous proteins. Names generated by Genepidgin, once deposited in public databases, may themselves be used as a basis to name other genes transitively. To prevent the propagation of incorrect product names, only high-confidence alignments should be used for naming.
- Prefer no name or an obviously generic name to an uninformative name.
- Prefer lowercase words in everything but acronyms and proper names.
- Prefer a standardized expression of common protein names.
- Prefer American English spelling.
- Use only 7-bit ASCII characters, so that names render correctly on every computing platform.

2.2.2 Steps in Filtering Process

The following list is a rough description of the steps involved in processing a name. This list is not a literal description of the layout of the code, but rather a high-level overview of how Genepidgin *cleaner* works.

whole name filtering and deletion Sometimes, names are published into the global protein namespace that are obviously the output of a malformed SQL query or accidentally copied Excel spreadsheet. We process these before doing anything else, extracting useful information when possible.

typo correction People misspell (for example) *hypothetical* and *transporter* in many, many ways. Correcting these names early prevents later filters from missing human-obvious corrections.

uninformative clause removal Subclauses that are globally uninformative are removed. For example, documented proteins should not have their functions described within their names, so phrases like "X involved with Y" simply become "X".

clause replacement The largest transformations happen here, where names are changed to become more consistent. For example, the phrase "transport family protein" becomes "transporter".

organism names The vast majority of the time, specific organism names are not informative when copied across species by homology or alignment. We remove them.

id removal Many published genes have obvious database ids. Genepidgin does not transitively assign these to new gene annotations.

punctuation cleanup Removing ids and other phrases often leaves bad punctuation and/or leftover parentheses, which then must be themselves removed.

standardize format The grammatical structures of product names are improved late in the cleaner process. This category assumes that by this point, the name is a keeper, and simply reformats it for consistent presentation.

final sanity check If, after filtering, the entire name is otherwise uninformative such as "CDS" or "small secreted protein", then the name is misleading and will be dropped.

capitalization Finally, Genepidgin tries to establish consistent capitalization: only proper names and acronyms are capitalized.

2.2.3 How to Use Genepidgin *cleaner*

All files used as input and output are in the Simple Name File Format.

via the command-line

- **"cleaner"** takes a name and applies the full list of filters to it. A name can be filtered to an empty string by this function; the output of the command will tell you why. Names that are filtered to nothing are ones Genepidgin considers to be uninformative.

```
$ genepidgin cleaner <inputfile>
```

Setting the -d flag indicates that genepidgin should return a default name (`"hypothetical protein"`) when a name would otherwise be blank.

usage doc

```
$ genepidgin cleaner -h

usage: cmdline.py cleaner [-h] [--silent] [--default] input output

positional arguments:
  input          filename with names to clean
  output         output file

optional arguments:
  -h, --help     show this help message and exit
  --silent, -s   display etymology to stdout during compute
  --default, -d  return default name "hypothetical project" when names filter
                 to nothing, else return emptry string
```

via Python

From inside your python shell, let's set up your first test case.

```
>>> import pidgin.cleaner
>>> bname = pidgin.cleaner.BioName()
>>> name = "BT002689 glycine/betaine/L-proline ABC transport protein, periplasmic-binding protein [De
```

Instatiating `BioName` compiles a couple hundred regular expressions. Instantiating a new `BioName` object for every name to be changed can get expensive. A single `BioName` object can reformat any number of names, so callers need only instantiate the class once.

Under the hood, `cleaner` calls on either `filter` or `cleanup`. When everyone had different default names, this distinction was more meaningful, but now everyone follows UNIPROT's *hypothetical protein* standard.

This name contains a great deal of spurious and unreliable information. A quick `cleaner` of this name...

```
>>> cleaned = bname.filter(name)
>>> print cleaned
"glycine/betaine/L-proline ABC transporter"
```

To see what happened during the filter process, we set `getOutput` to true when we call `filter`. Note the additional returned value.

```
>>> (cleaned, process_string) = bname.filter(name, getOutput=1)
>>> print cleaned
"glycine/betaine/L-proline ABC transporter"
>>> print process_string
filtered name in 5 steps:
0) original: BT002689 glycine/betaine/L-proline ABC transport protein, periplasmic-binding protein [D
```

```
1)   reason: transport protein -> transporter
    pattern: \btransport(er)?\s+protein\b
   filtered: BT002689 glycine/betaine/L-proline ABC transporter, periplasmic-binding protein [Desulfo
2)   reason: id
    pattern: \b[A-Za-z0-9]+\d{4,}(?<!\b(?:DUF|UPF)\d{4})\b(?!\s*(kD(a)?|-like|family|protein\s+family
   filtered:  glycine/betaine/L-proline ABC transporter, periplasmic-binding protein [Desulfovibrio d
3)   reason: delete spaces at beginning of name
    pattern: ^\s+
   filtered: glycine/betaine/L-proline ABC transporter, periplasmic-binding protein [Desulfovibrio de
4)   reason: delete closing brackets at end of name
    pattern: (?:\[[^]]*)\]\s*$
   filtered: glycine/betaine/L-proline ABC transporter, periplasmic-binding protein
5)   reason: delete notes after commas, dashes, semicolon--except when followed by family or superfam
    pattern: [-,;]\s+(?!family)(?!superfamily).*
   filtered: glycine/betaine/L-proline ABC transporter
```

(Note that `process_string` is a single multiline string, which looks good when `print`'ed but bad when simply exported.)

Reference the documentation in the code for more information on parameters. It's fairly well commented, if not clear.

Note: Please see *Credits* for contributor information.

2.3 Genepidgin *compare*

Genepidgin *compare* uses a combination of edit distance and longest-common-substring calculations to estimate the degree of similarity between two or more protein names.

2.3.1 Algorithm

To compare two names, we

1. decompose each name into tokens,
2. remove uninformative tokens,
3. rearrange the tokens in such a way as to... - minimize the edit distance between them, and - maximize the length of common token substrings
4. report a single number between 0 and 1 (inclusive) summarizing the distance between the two names.

In more detail:

1. **decompose each name into tokens**

   First, we split the names up by spaces, remove EC numbers and punctuation and other sorts of extra characters, convert everything to lowercase, etc.

   in: "Ribosomal protein, S23-type" **out:** "ribosomal" · "protein" · "s23-type"

2. **remove uninformative tokens**

   In this step we strike out words that are only useful in a grammatical sense, including *an, and, in, is, of, the,* etc. We also remove weasel words, such as *generic, hypothetical, related,* etc. Finally, we remove glue words, such as *associated, class, component, protein, system,* and *type.* When these words are stripped we are left with a "core" name that identifies the protein; different namers may use different glue words to format the core name and we ignore those.

in: "ribosomal" · "protein" · "s23-type" **out:** "ribosomal" · "s23"

Because we strip out noninformative tokens, we count all of the following strings as equal.

- "predicted protein"
- "putative protein"
- "hypothetical protein"
- "conserved hypothetical protein"

3. **rearrange the tokens in such a way as to ...**

   Finding the best edit distance between two names of, say, 4 tokens each is a bit tricky, because it's possible that the lowest cumulative edit distance will involve one or more sub-optimal individual token matches. In fact there are cases where the lowest distance is composed entirely of sub-optimal token pairings. So we need to try a lot of combinations. To do this we precompute two scores for each pair of tokens, and build two $n \times n$ matrices to hold them. We then score all possible paths with distinct pairwise token pairings via these matrices. For each path we combine two scores: we try to minimize the normalized edit distance between token pairs, and we try to maximize the length of the longest pairwise common substrings between pairs of tokens.

   In one matrix, we store the pairwise token-token edit distance, using the Damerau-Levenshtein distance, leveraging the excellent Python implementation by Michael Homer. We normalize the edit distance by dividing it by the number of characters in the longer token. The other `*n* × *n*` matrix holds the length of the longest common substring between each pair of tokens. Our LCS finder is similar to that published on the Wikipedia.

   In the case where the protein names have different numbers of tokens, we build square matrices from the largest dimension, padding the shorter dimension with empty tokens. There also are heuristics to handle cases where a token in one name is composed of two or more tokens in the other. The special handling for these special cases is too detailed for this document; see the source or contact the authors for details.

   Note that token order has no effect on the distance between two names.

4. **report a single number between 0 and 1 (inclusive) summarizing the distance between the two names.**

   A perfect token-token match is really good. A lot of perfect matches are really, really good. Long common substrings are fairly good. The Damerau-Levenshtein distance can return higher distances than we might like for these three types of token matches. On the other hand, maximizing the length of the longest common substring(s) has its own set of problems. After a great deal of trial and error, we have settled on the following equation, which has worked well on genome-scale scoring studies across a variety of prokaryotes.

```
"Genepidgin" distance =
    SUM(per-token normalized edit distance) *
    (1 - (SUM(per-token LCS length) / LENGTH(longer name))) *
    (1 / COUNT(compared tokens))
```

   The first line of this distance metric weights each pair of tokens equally. Thus a "SecG" · "SecG" match counts just as much as a "phosphoribosylglycinamide" · "phosphoribosylglycinamide" match.

   The second line of the metric weights each character equally, thereby lowering the distances between long tokens that differ only slightly, for example

   2,3,4,5-tetrahydropyridine-2,6-dicarboxylate 2,3,4,5-tetrahydropyridine-2-carboxylate

   The third line of the distance metric above simply normalizes the score from 0 to 1. A distance of 0 indicates the names have identical information content and are essentially equivalent. A distance of 1 indicates the names have nothing in common.

2.3.2 How to use Genepidgin *compare*

Given at least two input files, one reference and one or more queries, score the distance (using `genepidgin.distance.DistanceTool()`) between the names found in the files.

```
genepidgin compare (options) <reference_file> <query_file> [<query_file2> ...]

options:
  --help: this information
```

All input files must be in the Simple Name File Format.

This tool will create one output file per query file. The per-query output file(s) will have name(s) of the form `<query_file>.compared`.

If there are multiple query files, a summary file containing the closest query match for each reference name will also be created. The summary file will be named `<reference_file>.summary`.

Each line in the two-way comparison result will consist of the following tab-separated fields:

```
0.  ID. This is the string from the first field of the entry from the reference file.
1.  Score. The distance between the two names.
2.  Reference name. The reference name used for the comparison.
3.  Query name. The query name used for the comparison.
```

If a summary file is generated, each line in that file will consist of the following tab-separated fields:

```
0.  ID. This is the string from the first field of the entry from the reference file.
1.  Score. The distance between the two names.
2.  Reference name. The reference name used for the comparison.
3.  Best query name. The best matching query name.  In cases where multiple query names scored identi
4.  Best query source. The basename of the file which held the best query name. (ex: query_file1) In
```

Results are presented in the same order as in the input reference file. Names in query files that correspond to an ID not present in the reference file will be ignored. Names in the reference file with no corresponding query are scored as a complete miss (1.0). Input query and reference files may reside in any directory, but no two files may have the same basename.

2.3.3 Genepidgin *compare* Score Range

The distribution in accuracy is not linear between 0.0 and 1.0; that is, after a certain level of dissimilarity it doesn't matter how much more dissimilar two names are.

The following table presents a quick guide to the interpretation of distance scores.

score	likelihood of functional match
=0.0	functionally identical
0.0 - 0.1	excellent match
0.1 - 0.3	good match
0.3 - 0.5	possibly similar, with potentially significant distances
0.5 - 1.0	not generally useful
=1.0	completely different

There is support for using the output of Genepidgin *compare* directly within Python; consult `genepidgin/scorer.py` for details.

Note: Please see *Credits* for contributor information.

2.4 Genepidgin *select*

2.4.1 Goals

Genepidgin *select* generates gene product names from alignments to proteins in curated libraries (currently FIGfam, KEGG, Pfam, RefSeq, SwissProt and TIGRFAM). Blast and hmmer alignments from those libraries are read into Genepidgin via simple data formats (.pidginb and .pidginh, respectively), where they are sifted through to find the best name.

2.4.2 Selection Recipe

Summary

Sort qualifying sources, preferring: hmmer alignments to blast alignments, a lower e-value in hmmer hits, and a higher percent identity in blast hits. Walk through the sorted list until we find a name that remains informative after running through *Genepidgin cleaner*.

Details

Group all evidence by `dest_id` and consider each `dest_id` independently.

Over the course of this search, if a name filters to something uninformative (via *Genepidgin cleaner*), then examine the next relevant source, until either a valid source and name are found, or no sources remain and the name "hypothetical protein" is assigned.

Start by examining the hmmer hits. Remove hits that are neither TIGRFAM equivalogs nor Pfam hits labeled as equivalog-equivalents by JCVI. Next, remove hits whose score is less than its `family_trusted_cutoff` (see .pidginh). Take the name of the hit with the lowest e-value. If multiple hits have equivalent e-values, select the hit with the highest bit score.

If a `dest_id` has no hmmer hits deemed sutable for naming, examine the blast evidence (see .pidginb or .blastm8), calculating the following terms:

```
source_coverage = (source_stop - source_start + 1) / source_len
dest_coverage = (dest_stop - dest_start + 1) / dest_len
min_coverage = min(source_coverage, dest_coverage)

source_pct_identity = num_identities / source_len
dest_pct_identity = num_identities / dest_len
min_pct_identity = min(source_pct_identity, dest_pct_identity)

upper_pct_identity = max(min_pct_identity for all hits whose min_coverage  0.6)
lower_pct_identity = max(0.5, upper_pct_identity - 0.05)
```

Cluster all hits associated with `dest_id` that have `min_coverage` 0.6 and whose `min_pct_identity` is between `upper_pct_identity` and `lower_pct_identity` (inclusive). If `upper_pct_identity` < `lower_pct_identity`, ignore all hits.

If the cluster is not empty, and any of the hits in the cluster has a `source_auth` (see .pidginb) of KEGG, then select the name from the one with the highest `min_pct_identity`. If there are no hits from KEGG, proceed to SwissProt hits, then FIGfam and finally RefSeq, searching in each bin for the hit with the highest `min_pct_identity` within that bin.

2.4.3 Usage

Given a series of data files, use the selection recipe described above to determine product names for the given genes.

```
genepidgin select (options) [inputfiles]

options:
  -o --output    : where to save files, defaults to ./pidgin_names.txt
  -e --etymology : where to save etymology (debug), defaults to ./pidgin_etymology.txt
  -h --help      : this information
  --use_custom_blast_cutoffs : use different cutoffs for different sources
```

The format of Input and Output files are described below.

2.4.4 Input

Any number of input files following the following three formats are permitted. The ordering of the files, and the ordering of the lines within the files, does not matter. No tabs, newlines, or control characters are permitted in any of these fields.

.pidginb

All files with the extension `.pidginb` are assumed to contain BLAST alignments.

Each line in a `.pidginb` file will consist of the following tab-separated fields:

```
0.  dest_id STRING an identifier for a destination protein (i.e., a protein that
    should receive a name)
1.  dest_start INTEGER 1-based index of first aligned amino acid in destination
    protein
2.  dest_stop INTEGER 1-based index of last aligned amino acid in destination
    protein
3.  dest_len INTEGER number of amino acids in destination protein
4.  source_id STRING an identifier for a source protein (i.e., a protein whose
    name should be considered for assignment to the destination protein)
5.  source_start INTEGER 1-based index of first aligned amino acid in source
    protein
6.  source_stop INTEGER 1-based index of last aligned amino acid in source
    protein
7.  source_len INTEGER number of amino acids in source protein
8.  source_auth STRING the source of the data, used for heuristic processing,
    must be one of:
      - "FIGfam"
      - "KEGG"
      - "RefSeq"
      - "SwissProt"
9.  num_identities INTEGER number of exact amino acid matches in alignment
10. num_similarities INTEGER number of similar amino acid matches in alignment
11. raw_name STRING the name of the source protein
12. comment STRING can be used for any purpose
```

A sample line:

```
7000002454063496        134     581     448     7000000120703332        127     596     470     FIGfa
```

.pidginh

All files with the extension `.pidginh` are assumed to contain HMMER alignments.

Note: per-domain scores are ignored; we consider the whole hit only.

Each line in a `.pidginh` file will consist of the following tab-separated fields:

```
0.  dest_id STRING an identifier for a destination protein (i.e., a protein that
    should receive a name)
1.  dest_start INTEGER 1-based index of first aligned amino acid in destination
    protein
2.  dest_stop INTEGER 1-based index of last aligned amino acid in destination
    protein
3.  dest_len INTEGER number of amino acids in destination protein
4.  source_id STRING an identifier for a source family (i.e., a profile whose
    name should be considered for assignment to the destination protein)
    currently should be a TIGRFAM or Pfam id.
5.  source_start INTEGER 1-based index of first aligned position in source family
6.  source_stop INTEGER 1-based index of last aligned position in source family
7.  source_len INTEGER number of positions in source family
8.  score FLOAT score reported by hmmer
9.  family_trusted_cutoff FLOAT
10. e_value FLOAT+INTEGER in the format X.XXeY where X.XX is a positive float and Y is an integer
11. raw_name STRING the name of the source family
12. comment STRING can be used for any purpose
```

A sample line:

```
7000002454071269        3       140     138     TIGRfam 13      155     143     83.519997       80.0
```

.blastm8

Blast `-m8` format is also acceptable, but requires also submitting a name key via `--ref`, as m8 format contains no names. This method also has much slower execution time.

It is assumed that all names derived from `.blastm8` have lower priority than other sources.

2.4.5 Output

The names of these files are governed by the option usage, as described above.

Names

Each line of the name file has four columns:

```
0. dest_id STRING an identifier for a destination protein (i.e., a protein that should receive a name
1. name STRING the best available name for the destination protein
2. source_id STRING the id of the blast or hmmer hit used to name this protein
3. comment STRING the comment field from the line used to name this protein
```

A snippet from a names.txt from a development run:

```
7000002454076078   fructose-1-6-bisphosphatase   FIGfam    run on library updated 2009/10/22
7000002454076081   hypothetical protein          (blank)   (blank)
```

Note that hypothetical proteins don't have the final two fields, as they did not pick up a name from the given sources.

Etymology

The etymology file consists of a sequence of entries. Each entry describes the process by which the resulting name was given, showing tracking information as data is discarded and then summary information of how the name was cleaned up (plugs directly into *Genepidgin cleaner*) before it is presented.

Entries are separated by five equals signs and a newline: `=====`

Each entry begins with the dest_id alone on the first line of the block. Convenient for searching!

A snippet of a local run:

```
7000002454076078
1 hmmer source found.
0 hmmer sources were removed due to not meeting the trusted family score.
One hmmer source had a good name.
Found an acceptable name in the hmmer sources. The one we liked best came from:
./test/Rho_sphaeroides_241_HMMERTRANSCRIPTS_17.pidginh:2013
This source's name was cleaned up by genepidgin:
filtered name in 1 step:
0) original: Fructose-1-6-bisphosphatase
1)   reason: protein names should not start with a capital letter
    pattern: (?:(?<=similar to )|^)([A-Z])(?=[a-z][a-z]+([ /,-]|$))
   filtered: fructose-1-6-bisphosphatase
Final name: fructose-1-6-bisphosphatase
=====
7000002454076081
0 hmmer sources found.
No name was derived from hmmer sources.
2 blast sources were found.
0 blast sources were removed by filtering for low coverage (<0.6).
The highest percent identity of any remaining blast source is 0.992. The lowest is 0.945.
0 blast sources were removed due to not being within the percent identity window (0.992, 0.942).
All 2 blast sources had names that filtered to nothing.
No name was ultimately selected from any of the supplied sources.
Final name: hypothetical protein
```

use_custom_blast_cutoffs

As hardcoded in pidgin.select, widens the cutoff margin for blast hits (currently KEGG-only, check source for details).

Note: Please see *Credits* for contributor information.

2.5 Credits

Genepidgin was written by Clint Howarth and Matthew Pearson, with recent updates made by Janet Gainer-Dewar. Many people have contributed to the project:

2.5.1 cleaner

The design of `genepidgin cleaner` grew out of years of suggestions from many people, including annotators who have worked in Genome Annotation in the Microbial Sequencing Platform at the Broad Institute. It was implemented by Clint Howarth and Matthew Pearson.

Many people have contributed to the name cleaning logic, including: Lucia Alvarado-Balderrama[1], Sinead Chapman[1], Zehua Chen[1], Jonathan Goldberg[1], Sharvari Gujja[1], Clint Howarth[1], Chinnappa Kodira[2], Teena Mehta[1], Matthew Pearson[1], Narmada Shenoy[1], Tom Walk[1], Chandri Yandava[1], Qiandong Zeng[1], and the Autoannotate development team[3].

[1] Broad Institute [2] 454 Life Sciences [3] J. Craig Venter Institute

2.5.2 compare

`genepidgin compare` was designed and implemented by Matthew Pearson.

It includes an open-source implementation of the Damerau-Levenshtein distance written by Michael Homer.

2.5.3 select

`genepidgin select` was designed by Sharvari Gujja, Brian Haas, Clint Howarth, Matthew Pearson, and Qiandong Zeng. Clint Howarth implemented it, and Janet Gainer-Dewar has added features.

2.5.4 Special Thanks

Finally, thanks to the Autoannotate development team at JCVI, who were kind enough to share the source code of their naming utility with us. Seeing how hard their institute worked to reformat names motivated us to release and document our own naming logic.

2.5.5 Project Name History

This project began life as BioName. It turns out that there already is a project named Bioname. Though this BioName addresses a completely different problem, our goal is to help reduce name-related confusion. Thus we decided to change the name of our software toolkit to Pidgin. We retain the term BioName as an internal class name for source compatibility. We are aware that there is an IM chat client called Pidgin, and even though it's completely unrelated to gene naming, some people found this confusing. This project is now `Genepidgin`, and that's that.

We would like to take this opportunity to point out that naming is a challenging problem, on many levels. We apologize for any confusion.

2.6 Changes

1.2 added option for source-specific cutoffs when culling blast hits (thanks, @jdewar)

1.1 revised initial release, better python structure, packaging, and documentation

1.0 initial public release

2.7 Simple Name File Format

We try to use the same input/output format for names as much as possible throughout `genepidgin`.

The simple name file format is a flat text file. It's human-readable and was designed with simple database interactions in mind.

Each line has three columns:

1. A unique identifier, for example a database id, or simply a blank space.
2. A tab character (`\t`)
3. The name, until the first tab character.

Ignored:

- Lines beginning with `#`
- Any information following the second tab in a line

An example of a simple name file:

```
id1    the name can be any length
id2    and have any character but a newline
# this line is ignored
id3    this name is not ignored
id4    name followed by tab                  this information is ignored
```

2.8 License Information

Pidgin is offered under the BSD license.

```
#
# Copyright (c) 2009 The Broad Institute, Inc. All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# Redistributions of source code must retain the above copyright notice,
# this list of conditions and the following disclaimer.
#
# Redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution.
#
# Neither the name of the Broad Institute nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE BROAD INSTITUTE ''AS IS'' AND ANY
# EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
# PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE BROAD INSTITUTE BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
# BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
# WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
# OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
# EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
```