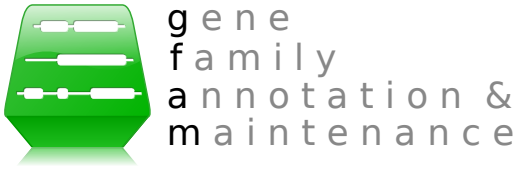
GFam Documentation

Release 1.1

Tamás Nepusz, Rajkumar Sasidharan

July 13, 2012

CONTENTS



This is the documentation of `gfam`, a Python module to aid the automatic annotation of gene families based on consensus domain architecture, written by Tamás Nepusz.

GFam is free and open-source. If you use GFam in your research, please cite the corresponding publication as follows:

> Sasidharan R, Nepusz T, Swarbreck D, Huala E and Paccanaro A:
> **GFam: a platform for automatic annotation of gene families**.
> *Nucleic Acids Research*, Advance Access, 2012.
> doi:10.1093/nar/gks631.

GFam can generally be considered stable. We have used it successfully to annotate the whole genome of *Arabidopsis thaliana* and *Arabidopsis lyrata*. Please email the author if you discover any bugs, or feel free to submit a bug report on GitHub.

If you are interested in the latest and greatest (but maybe unstable) version of GFam, you can get it from GitHub - just click on the **Download source** button on the GitHub page.

CHAPTER

ONE

INTRODUCTION

1.1 What is GFam?

GFam (or `gfam`) is a Python module to aid the automatic annotation of gene families based on consensus domain architectures. `gfam` started out as a collection of loosely coupled Python scripts that process the output of `iprscan` (a tool to obtain domain assignments of individual genes from InterPro) and conduct some analyses using BLAST to detect novel, previously uncharacterised domains. The original domains and the detected novel domain candidates are then used to create a consensus domain assignment for each gene sequence. Genes are then finally assigned to families based on their domain architectures. Finally, the tool derives functional labels for families based on the Gene Ontology and an assignment between InterPro domains and Gene Ontology terms. Optionally, a Gene Ontology overrepresentation analysis can also be conducted on the GO annotations of individual domains in the same sequence to reinforce the functional labels.

1.2 Requirements

You will need the following tools to run `gfam`:

- Python 2.5 or later. Python 3 is not supported yet. `gfam` was also tested with Jython 2.5.1.
- NCBI BLAST; in particular, the `formatdb` and `blastall` tools from the legacy C-based BLAST distribution. You can still use `gfam` wit the newer, C++-based BLAST if you have the `legacy_blast.pl` wrapper script in the BLAST folder.

The latest release of SciPy is recommended, but not necessary. `gfam` uses SciPy for calculating the logarithm of the gamma function in the overrepresentation analysis routines, but it falls back to a (somewhat slower) Python implementation if SciPy is not installed.

1.3 For the impatient

`gfam` is driven by a master configuration file named `gfam.cfg`. A sample configuration file is given in the distribution. The sample file works fine for the gene sequences of *Arabidopsis thaliana*; for other species, you might have to tweak some of the parameters, and you will surely have to modify the paths to the data files. The configuration file is documented and mostly self-explanatory.

If you do not have a configuration file for some reason, or you want to generate a new one from scratch, you can ask `gfam` to do it:

```
$ bin/gfam init
```

This will create a default configuration file named `gfam.cfg` (if it does not exist already) and lists the configuration options you have to set in the file before starting GFam.

If the configuration file is well in order, you can launch `gfam` by typing:

```
$ bin/gfam
```

This will run the whole `gfam` analysis pipeline using the configuration specified in `gfam.cfg`. If your configuration file is named otherwise, you can run it by typing:

```
$ bin/gfam -c my_config.cfg
```

The results will be put into whatever work directory you specified in the configuration file. By default, this is named `work`. See *Output files* for more details on what will be calculated and where you can find them.

1.4 Questions, comments

If you have a question or a comment about `gfam` or you think you have found a bug, feel free to contact me using the email address given in the header of this document.

CHAPTER

TWO

RUNNING GFAM

2.1 Input files

The input files can be grouped into three large groups: *data files*, *mapping files* and the *configuration file*. Data files contain the actual input data that is specific to a given organism. Mapping files usually map between IDs of different data sources (for instance, from InterPro domain IDs to Gene Ontology terms) or IDs to human-readable descriptions. The *configuration file* tells GFam where to find the data files and the mapping files. When one wants to process a new organism with GFam, it is therefore usually enough to replace the paths of the data files in the configuration only, as the mapping files can be re-used for multiple analyses.

GFam requires the following data files:

Sequence file This file must contain all the sequences that are being analysed and annotated by GFam.

Domain assignment file This file is produced by running `iprscan`, the command-line variant of InterProScan and it assigns sections of each segment in the sequence file to known domains in InterPro. The sequence IDs in this file must be identical to the ones in the sequence file; if not, one can specify a regular expression in the *configuration file* to extract the sequence ID from the FASTA defline.

Besides the data files, the following mapping files are also needed:

InterPro – GO mapping This file maps InterPro IDs to their corresponding GO terms, and it can be obtained from <http://www.geneontology.org/external2go/interpro2go>.

Mapping of domain IDs to human-readable names Fairly self-explanatory; a tab-separated flat file with two columns, the first being the domain ID and the second being the corresponding human-readable name. It is advisable to construct a file which contains at least the InterPro, Pfam, SMART and Superfamily IDs as these are the most common (and many Pfam, SMART and Superfamily IDs do not have corresponding InterPro IDs yet). If you want to create such a mapping file easily, please refer to *Updating the mapping of IDs to human-readable names*.

Parent-child relationships of InterPro terms This file contains the parent/child relationships between InterPro accession numbers to indicate family/subfamily relationships. This file is used to map each InterPro subfamily ID to the corresponding family ID, and it can be obtained from EBI.

The Gene Ontology This file contains the Gene Ontology in OBO format, and it is required only for the label assignment and overrepresentation analysis steps. The latest version of the file can be obtained from the homepage of the Gene Ontology project.

GFam accepts uncompressed files or files compressed with `gzip` or `bzip2` for both the data and the mapping files. Compressed files will be decompressed on-the-fly in memory when needed.

2.2 The configuration file

The default configuration file of GFam is called `gfam.cfg`, but you can specify an alternative configuration file name on the command line using the `-c` switch. A sample configuration file is included in the GFam distribution; however, you can always generate a new one by running the following command:

```
$ bin/gfam init
```

This will generate a file named `gfam.cfg` in the current directory and list the configuration keys you have to modify before starting your analyses.

The configuration file consists of sections, led by a `[section]` header and followed by `name=value` entries. Lines beginning with `#` or `;` are ignored and used to provide comments. Lines containing whitespace characters only are also ignored. For more details about the configuration file format, please refer to the ConfigParser module in the documentation of Python.

The full list of supported configuration keys and their default values is as follows:

2.2.1 Section `DEFAULT`

`file.input.iprscan` Raw output file from IPRScan that contains domain assignments for all the sequences we are interested in.

`file.input.sequences` A FASTA file containing all the sequences being analysed.

`file.log.iprscan_exclusions` Log file to store why given sequences have been rejected during the filtering of the input IPRScan file. Feel free to leave it empty.

`file.mapping.gene_ontology` A file containing the Gene Ontology in OBO format.

Default value: `data/gene_ontology.obo`

`file.mapping.interpro2go` File containing the mapping of GO terms to InterPro entries, as downloaded from geneontology.org.

Default value: `data/interpro2go`

`file.mapping.interpro2name` File containing a tab-separated list of InterPro IDs and their corresponding human-readable descriptions. This can be constructed by the following command:

```
$ bin/download_names.py | gzip -9 >data/names.dat.gz.
```

Default value: `data/names.dat.gz`

`file.mapping.interpro_parent_child` File containing the parent-child relationships of InterPro terms.

Default value: `data/ParentChildTreeFile.txt`

`folder.work` The working folder in which to put intermediary files.

Default value: `work`

`folder.output` The output folder in which to put the final results.

Default value: `work`

`sequence_id_regexp` Python regular expression that matches gene IDs from the sequence file. This is necessary if the IPRScan output uses a sequence ID that is only a part of the sequence ID in the input FASTA file. If it is empty, no ID transformation will be done, the sequence IDs in the input FASTA file will be matched intact to the IPRScan output. If it is not empty, it must be a valid Python regular expression with a *named* group "id" that matches the gene ID that is used in the IPRScan output. If you don't know what named groups are, check the documentation of the Python `re` module.

untrusted_sources Which assignment sources NOT to trust from InterPro? (space separated)

Default value: `HAMAP PatternScan FPrintScan Seg Coil`

max_overlap Maximum overlap allowed between assignments originating from the same data source.

Default value: `20`

num_cpu_cores Hint on the number of CPU cores to use during the analysis. Currently the BLAST invocation uses this hint to select the number of threads used by BLAST to speed up calculations.

The default value is 1 since it is not possible to auto-detect the number of CPU cores in a platform independent way. Feel free to raise this value if your computer has multiple CPU cores.

Default value: `1`

2.2.2 Section `analysis:blast_filter`

min_seq_identity Minimum sequence identity between two fragments to consider them as being in the same domain.

Default value: `45`

min_alignment_length Minimum alignment length between two fragments to consider them as being in the same domain. If normalization_method is not off, this must be the normalized alignment length threshold according to the chosen normalization method.

Default value: `0.7`

max_e_value Maximum E-value between two fragments in order to consider them as being in the same domain.

Default value: `1e-3`

normalization_method Normalization method to use for calculating normalized alignment length Must be one of: off, smaller, larger, query, hit.

Default value: `query`

2.2.3 Section `analysis:find_domain_arch`

min_novel_domain_size A novel domain occur in at least this number of sequences.

Default value: `4`

2.2.4 Section `analysis:find_unassigned`

min_seq_length Minimum number of amino acids in a sequence in order to consider it further (i.e. to calculate its unassigned fragments)

Default value: `30`

min_fragment_length Minimum number of amino acids in a sequence fragment in order to consider that fragment as a novel domain candidate.

Default value: `75`

sequences_file Input FASTA file containing all the sequences of the representative gene model being analysed.

Default value: same as `file.input.sequences`

2.2.5 Section analysis:iprscan_filter

e_value_thresholds E-value thresholds to use when processing the initial InterPro file. This entry is a semicolon-separated list of source=threshold pairs, the given threshold will be used for the given data source. If an entry contains only a threshold, this will be considered as a default threshold for sources not explicitly mentioned here.

Default value: `1e-3;superfamily=inf;HMMPanther=inf;Gene3D=inf;HMMPIR=inf`

interpro_parent_child_mapping File containing the parent-child relationships of InterPro terms.

Default value: same as `file.mapping.interpro_parent_child`

stages.1 These configuration keys specify which assignment sources are to be taken into account at each stage of the analysis. For more information about what these stages are, please refer to the documentation, especially the description of Step 2 in section "Steps of the GFam pipeline"

Default value: `ALL-HMMPanther-Gene3D`

2.2.6 Section analysis:jaccard

min_similarity Minimum Jaccard similarity between the neighbour sets of two fragments in order to consider them as being in the same domain.

Default value: `0.66`

assume_loops Whether to assume that a protein is connected to itself or not.

Default value: `1`

only_linked Whether to consider only those protein pairs which are linked in the input file. If this is 1, protein pairs not in the input file will not be returned even if their Jaccard similarity is larger than the given threshold.

Default value: `1`

2.2.7 Section analysis:overrep

confidence The p-value threshold in the hypergeometric test used in the overrepresentation analysis process.

Default value: `0.05`

correction The method used to account for multiple hypothesis testing. Valid choices: bonferroni (Bonferroni correction), Sidak (Sidak correction), fdr (Benjamini-Hochberg method), none (off). The Bonferroni and Sidak correction methods control the family-wise error rate (FWER), while the Benjamini-Hochberg method control the false discovery rate.

Default value: `fdr`

min_term_size The minimum number of annotated domains a GO term must have in order to be considered in the overrepresentation analysis.

Default value: `1`

2.2.8 Section generated

file.unassigned_fragments File in which the unassigned sequence fragments are stored.

Default value: `%(folder.work)s/unassigned_fragments.ffa`

file.valid_gene_ids File containing a list of valid gene IDs (extracted from the input file)

Default value: `%(folder.work)s/gene_ids.txt`

file.domain_architecture_details File containing the detailed final domain architecture for each sequence.

Default value: `%(folder.output)s/domain_architecture_details.txt`

file.domain_architecture_stats File containing genome-level domain architecture statistics.

Default value: `%(folder.output)s/domain_architecture_stats.txt`

2.2.9 Section `utilities`

folder.blast The folder containing the BLAST executables. It does not matter whether you have the old C-based or the newer C++-based tools, GFam can use both if you also have the `legacy_blast.pl` script that adapts the new tools to the command line syntax used by the older ones.

util.formatdb The path to formatdb. You may use the name of the folder containing the BLAST executables or the full path (including the name of the tool). If you have the newer, C++-based BLAST tools (which do not have formatdb), pass the name of the folder containing the BLAST executables here, and if you have the `legacy_blast.pl` script in the same folder (plus a working Perl setup), GFam will detect the situation and run `legacy_blast.pl` accordingly.

Default value: same as `folder.blast`

util.blastall The path to blastall. You may use the name of the folder containing the BLAST executables or the full path (including the name of the tool). If you have the newer, C++-based BLAST tools (which do not have blastall), pass the name of the folder containing the BLAST executables here, and if you have the `legacy_blast.pl` script in the same folder (plus a working Perl setup), GFam will detect the situation and run `legacy_blast.pl` accordingly.

Default value: same as `folder.blast`

2.3 Output files

GFam produces four output files in the output folder specified in the *configuration file*. These files are as follows:

2.3.1 `domain_architectures.tab`

A simple tab-separated flat file that contains the inferred domain architecture for each sequence in a simple, summarised format. The file is sorted in a way such that more frequent domain architectures are placed at the top. Sequences having the same domain architecture are sorted according to their IDs.

The file has six columns. The first column is the ID of the sequence (e.g., `AT1G09650.1`), the second is the sequence length (e.g, `382`). The third column contains a summary of the domain architecture of the sequence, where domains are ordered according to the starting position, and consecutive domain IDs are separated by semicolons (e.g., `IPR022364;IPR017451`). The InterPro domain ID is used whenever possible. Novel domains identified by GFam are denoted by `NOVELxxxxx`, where `xxxxx` is a five-digit identifier. The fourth column contains the frequency of this domain architecture (i.e. the number of sequences that have the same domain architecture). The fifth column is the same as the third, but the exact starting and ending positions of the domain are also added in parentheses after the domain ID (e.g., `IPR022364(9-57);IPR017451(112-357)`). The sixth column contains the concatenated human-readable descriptions of the domains (for instance, `F-box domain, Skp2-like;F-box associated interaction domain`).

2.3.2 domain_architecture_details.txt

This file is the human-readable variant of `domain_architectures.tab` (which is more suitable for machine parsing). It contains blocks separated by two newline characters; each block corresponds to a sequence and has the following format:

```
AT1G09650.1
    Primary assignment source: HMMTigr
    Number of data sources used: 2
    Data sources: superfamily, HMMTigr
    Coverage: 0.772
    Coverage w/o novel domains: 0.772
       9-  57: SSF81383 (superfamily, stage: 2) (InterPro ID: IPR022364)
               F-box domain, Skp2-like
     112- 357: TIGR01640 (HMMTigr, stage: 1) (InterPro ID: IPR017451)
               F-box associated interaction domain
```

The first line of each block is unindentend and contains the sequence ID. The remaining lines are indented by at least four spaces. The second line contains the name of the InterPro data source that was used to come up with the primary assignment in *step 2 of the pipeline* (see more details later in *Steps of the GFam pipeline*), followed by the number of data sources used to construct the final assignment, and of course the data sources themselves. The fifth and sixth lines contain the fraction of positions in the sequence that are covered by at least one domain; the fifth line takes into account novel domains (`NOVELxxxxx`), while the sixth line does not. The remaining lines list the domains themselves along with the data source they came from and the stage in which they were selected. For more details about the stages, see *Steps of the GFam pipeline*.

2.3.3 assigned_labels.txt

TODO

2.3.4 overrepresentation_analysis.txt

This file contains the results of the Gene Ontology overrepresentation analysis for the domain architecture of each sequence. Note that since the results of the overrepresentation analysis depend only on the domain architecture, the results of sequences having the same domain architecture will be completely identical.

The file consists of blocks separated by two newlines, and each block corresponds to one sequence. Each block has the following format:

```
AT1G61040.1
  0.0009: GO:0016570 (histone modification)
  0.0009: GO:0016569 (covalent chromatin modification)
  0.0024: GO:0016568 (chromatin modification)
  0.0036: GO:0006325 (chromatin organization)
  0.0049: GO:0051276 (chromosome organization)
  0.0055: GO:0006352 (transcription initiation)
  0.0095: GO:0006461 (protein complex assembly)
  0.0109: GO:0065003 (macromolecular complex assembly)
  0.0111: GO:0006996 (organelle organization)
  0.0126: GO:0043933 (macromolecular complex subunit organization)
```

In each block, the first number is the p-value obtained from the overrepresentation analysis, the second column is the GO ID. The name corresponding to the GO label is contained in parentheses. Blocks containing a sequence ID only represent sequences with no significant overrepresented GO labels in their domain architecture.

2.4 Command line options

GFam is started by the master script in `bin/` as follows:

```
$ bin/gfam
```

The exact command line syntax is `bin/gfam [options] [command]`, where `command` is one of the following:

`init` Generates a configuration file for GFam from scratch. The name of the configuration file will be `gfam.cfg` by default, but you can change it with the `-c` switch. GFam will refuse to overwrite existing configuration files. Example:

```
$ bin/gfam -c a_lyrata.cfg init
```

`run` Runs the whole GFam pipeline. This is the default command.

`clean` Removes the temporary directory used to store the intermediate results. The name of the temporary directory is determined by the `folder.work` configuration option in the *configuration file*.

> **Warning:** If the output directory is the same as the temporary directory (`folder.work` is equal to `folder.output` in the configuration), the `clean` command will also delete the final results from the output folder!

The default configuration file used is always `gfam.cfg`, but it can be overridden with the `-c` switch. For example, the following command will clean the work directory specified in `a_lyrata.cfg`:

```
$ bin/gfam -c a_lyrata.cfg clean
```

The following extra command line switches are also available:

-h, --help	shows a help message and then exits
-c FILE, --config-file=FILE	specifies the name of the configuration `FILE`
-v, --verbose	enables verbose logging
-d, --debug	shows debug messages as well
-f, --force	forces the recalculation of the results of intermediary steps in the GFam pipeline even when GFam thinks everything is up-to-date.

Besides the master script, there are scripts for re-running individual steps of the GFam pipeline. These scripts are separate Python modules in `gfam/scripts` and they correspond to the *steps of the GFam pipeline*. It is unlikely that you will have to run them by hand, but if you do, you have to supply the necessary input on the standard input stream of the scripts. For instance, if you want to do some custom filtering on a BLAST tabular result file, you can use `gfam/scripts/blast_filter.py` as follows:

```
$ python -m gfam.scripts.blast_filter -e 1e-5 <input.blast
```

This will filter `input.blast` and remove all entries with an E-value larger than 10^{-5}. The result will be written to the standard output.

You can get a summary of the usage of each script in `gfam/scripts` as follows:

```
$ python -m gfam.scripts.blast_filter --help
```

Of course replace `blast_filter` with the name of the script you are interested in. The default values of the command line switches of these scripts come from the *configuration file*, and they also support `-c` to change the name of the configuration file.

In 99.9999% of the cases, you will only have to do `bin/gfam init` to create a new configuration file, `bin/gfam` to run the pipeline and `bin/gfam clean` to clean up the results.

CHAPTER

THREE

STEPS OF THE GFAM PIPELINE

The GFam pipeline consists of multiple steps. In this section, we will describe what input files does the GFam pipeline operate on, how the steps are executed in order one by one and what output files are produced in the end. First, a short overview of the whole process will be given, followed by a more detailed description of each step.

3.1 Overview of a GFam analysis

GFam infers annotations for sequences by first finding a consensus domain architecture for each step, then collecting Gene Ontology terms for each domain in a given domain architecture, and selecting a few more specific ones. Optionally, a Gene Ontology overrepresentation analysis can also be performed on the terms to determine whether some GO terms occur more frequently in a given domain architecture than expected by random chance. Out of these three steps, the calculation of the consensus domain architecture is the most complicated one, as GFam has to account for not only the known domain assignments from InterPro, but also for the possible existence of novel, previously uncharacterised domains. The whole pipeline can be broken to 8+1 steps as follows:

1. Extracting valid gene IDs from the sequence file.
2. Determining a preliminary domain architecture for each sequence by considering known domains from the domain assignment file only.
3. Finding the unassigned regions of each sequence; i.e. the regions that are not assigned to any domain in the preliminary domain architecture.
4. Running an all-against-all BLAST comparison of the unassigned sequence fragments and filtering BLAST results to determine which fragments may correspond to the same novel domain. Such filtering is based primarily on E-values and alignment lengths. At this point, we obtain a graph on the sequence fragments where two fragments are connected if they passed the BLAST filter.
5. Calculating the Jaccard similarity of the sequence fragments based on the connection patterns and removing those connections which have a low Jaccard similarity.
6. Finding the connected components of the remaining graph. Each connected component will correspond to a tentative novel domain.
7. Calculating the consensus domain architecture by merging the preliminary domain architecture with the newly detected novel domains.
8. Selecting a functional label for each of the domain architectures based on a mapping between InterPro domains and Gene Ontology terms.
9. Conducting a Gene Ontology overrepresentation analysis on each of the sequences and their domain architectures to derive the final annotations.

These steps will be described more in detail in the next few subsections.

3.2 Step 1 – Extracting valid gene IDs

In this step, the input sequence file is read once and the gene IDs are extracted from the FASTA deflines. The gene ID is assumed to be the first word of the defline. If the deflines in the original FASTA file follow some other format, one can supply a regular expression in the *configuration file* that can be used to extract the actual ID from the first word of the defline.

3.3 Step 2 – Preliminary domain architecture

This step processes the domain assignment file and tries to determine a preliminary domain architecture for each sequence. A preliminary domain architecture considers known domains from InterPro only. Domain architectures for each sequence are determined in isolation, so the domain architecture of one sequence has no effect on another.

For each sequence, we first collect the set of domain assignments from the domain assignment file. Each assignment has a data source (e.g., HMMPfam, Superfamily, HMMSmart and so on), a domain ID according to the schema of the source, the starting and ending indices of the domain in the amino acid chain, an optional InterPro ID to which the domain ID is mapped, and an optional E-value. First, the list is filtered based on E-values, where one might apply different E-value thresholds for different data sources. This leads to a list of trusted domain assignments that are not likely to be artifacts. After that, GFam performs multiple passes on the list of trusted domain assignments, starting with a subset focused on more reliable data sources. Less reliable data sources join in the later stages, and it is possible that some data sources are not considered at all.

During the first pass, one single data source that is giving the highest coverage of the sequence is selected from the most reliable data sources. This data source will be referred to as the *primary data source*, and the domains of the primary data source will be called the *primary assignment*. After the first pass, the primary assignment will be extended by domains from other data sources in a greedy manner using the following rules:

1. Larger domains from other data sources will be considered first. (In other words, the remaining assignments not included already in the primary assignment are sorted by length in descending order).
2. Domains are considered one by one for addition to the primary assignment.
3. If a domain is the exact duplicate of some other domain already added (in the sense that it starts and ends at the same amino acid index), the domain is excluded from further consideration.
4. If a domain to be added overlaps with an already added domain from another data source, the domain is excluded from further consideration.
5. If a domain to be added is inserted *completely* into another domain from the same data source, it is added to the primary assignment and the process continues with the next domain from step 2. Note that the opposite cannot happen as we consider domains in decreasing order of their sizes.
6. If a domain to be added overlaps partially with an already added domain from the same data source, the size of the overlap decides what to do. Overlaps smaller than a given threshold are allowed, the domain will be added and the process continues from step 2. Otherwise, the domain is excluded from further consideration and the process continues from step 2 until there are no more domains left in the current stage.

We call this five-step procedure the *expansion* of a primary assignment. Remember, GFam works in multiple stages; the first stage creates the primary assignment with a limited set of trusted data sources, the second stage expands the primary assignment with an extended set of data sources, and there might be a third or fourth stage and so on with even more extended sets of data sources. For *Arabidopsis thaliana* and *Arabidopsis lyrata*, we found the following strategy to be successful:

1. Assignments from HAMAP, PatternScan, FPrintScan, Seg and Coil are thrown away completely for the following reasons:

- HAMAP may not be a suitable resource for eukaryotic family annotation as it is geared towards completely sequenced microbial proteome sets and provides manually curated microbial protein families in UniProtKB/Swiss-Prot [1]. For *Arabidopsis thaliana*, there were only 133 domains annotated by HAMAP and all domains had E-values larger than 0.001.
- PatternScan and FPrintScan [2] are resources for identifying motifs in a sequence and are not very helpful in understanding larger evolutionary units or domains. The match size ranges between 3 and 103 amino acids for PatternScan and between 4 and 30 amino acids for FPrintScan.
- Seg and Coil were ignored as these define regions of low compositional complexity and coiled coils, respectively, and are not particularly informative in the context of defining gene families.

2. An E-value threshold of 10^{-3} is applied to the remaining data sources, except for Superfamily, HMMPanther, Gene3D and HMMPIR which are taken into account without any thresholding.

   The threshold of 10^{-3} was chosen based on the following observation. There are 3,816 domain assignments from HMMPfam with a E-value larger than 0.1, 1,625 assignments with an E-value between 0.1 and 0.01 and 1,650 assignments with an E-value between 0.01 and 0.001. We looked at the type of domains that had an E-value between 0.1 and 0.01 and 0.01 and 0.001. We noticed that at least 80% of the domains are some kind of repeat domains (PPR, Kelch, LLR, TPR etc) or short protein motifs (different types of zinc fingers, EF-hand, HLH etc). It is reasonable to believe that at an E-value less than 0.001, the majority of the domains are likely to be spurious matches due to the sequence nature (low-complex and short) of these domains. We decided to consider domains from HMMPfam that had an E-value of 0.001 or smaller. We may miss but only a handful of real domains if we choose 0.001 as our E-value threshold. However, we would like to point out that the threshold is not hard-wired into GFam, rather it is a parameter than can be tuned for each assignment source to suit the users' needs.

3. GFam performs three passes on the list of domain assignments obtained up to now. The first and second passes do not consider HMMPanther and Gene3D assignments as they tend to split the sequence too much. The third stage considers all the data sources.

4. The maximum overlap allowed between two domains of the same source (excluding complete insertions which are always accepted) is 30 amino acids. This was based on the distribution of domain overlap lengths for the different resources.

The stages and the E-value thresholds are configurable in the *configuration file*.

3.4 Step 3 – Finding unassigned sequence fragments

This step begins the exploration for novel, previously uncharacterised domains among the sequence fragments left uncovered by the preliminary assignment that we calculated in *step 2*. We improvised on the method described by Haas *et al* [3] to identify novel domains. The step iterates over each sequence and extract the fragments that are not covered by any of the domains in the preliminary domain assignment. Sequences or fragments that are too short are thrown away, the remaining fragments are written in FASTA format into an intermediary file. The sequence and fragment length thresholds are configurable. For the analysis of *A.thaliana* and *A.lyrata* sequences, the minimum fragment length is set to 75 amino acids.

[1] Lima T, Auchincloss AH, Coudert E, Keller G, Michoud K, Rivoire C, Bulliard V, de Castro E, Lachaize C, Baratin D, Phan I, Bougueleret L and Bairoch A. HAMAP: a database of completely sequenced microbial proteome sets and manually curated microbial protein families in UniProtKB/Swiss-Prot. *Nucl Acids Res* **37**(Database):D471-D478, 2009.

[2] Scordis P, Flower DR and Attwood TK. FingerPRINTScan: intelligent searching of the PRINTS motif database. *Bioinformatics* **15**(10):799-806, 1999.

[3] Haas BJ, Wortman JR, Ronning CM, Hannick LI, Smith RK Jr, Maiti R, Chan AP, Yu C, Farzad M, Wu D, White O, Town CD. Complete reannotation of the *Arabidopsis* genome: methods, tools, protocols and the final release. *BMC Biol* **3**:7, 2005.

3.5 Step 4 – All-against-all BLAST comparison and filtering

This step uses the external `NCBI BLAST` executables (namely `formatdb` and `blastall`) to determine pairwise similarity scores between the unassigned sequence fragments. First, a database is created from all sequence fragments using `formatdb` in a temporary folder, then a BLAST query is run on the database with the same set of unassigned fragments using `blastall -p blastp`. Matches with a sequence percent identity or an alignment length less than a given threshold are thrown away, so are matches with an E-value larger than a given threshold. The user may choose between using unnormalised alignment lengths or normalised alignment lengths with various normalisation methods (normalising with the length of the smaller, the larger, the query or the hit sequence).

For *A.thaliana* and *A.lyrata*, the following settings were used:

- Minimum sequence identity: 45%
- Minimum normalised alignment length: 0.7 (normalisation done by the length of the query sequence)
- Maximum E-value: 10^{-3}

3.6 Step 5 – Calculation of Jaccard similarity

After the fourth step, we have essentially obtained a graph representation of similarity relations between unassigned sequence fragments. In this graph representation, each sequence fragment is a node, and two fragments are connected by an edge if they passed the BLAST filter in *step 4*. We will be looking for tightly connected regions in this graph in order to identify sequence fragments that potentially contain the same novel domain. It is a reasonable assumption that if two sequences contain the same novel domain, their neighbour sets in the similarity graph should be very similar. Jaccard similarity is a way of quantifying similarity between nodes in a graph by looking at their neighbour sets. Let *i* and *j* denote two nodes in a graph and let Γ_i denote the set consisting of *i* itself and *i*'s neighbours in the graph. The Jaccard similarity of *i* and *j* is then defined as follows:

$$\sigma_{ij} = \frac{|\Gamma_i \cap \Gamma_j|}{|\Gamma_i \cup \Gamma_j|}$$

where $|\ldots|$ denotes the size of a set. We calculate the Jaccard similarity of each connected pairs of nodes and keep those which have a Jaccard similarity larger than 0.66. This corresponds to keeping pairs where roughly 2/3 of their neighbours are shared. The Jaccard similarity threshold can be adjusted in the *configuration file*.

3.7 Step 6 – Identification of novel domains

Having obtained the graph filtered by Jaccard similarity in *step 5*, we detect the connected regions of this graph by performing a simple connected component analysis. In other words, sequence fragments corresponding to the same connected component of the filtered graph are assumed to belong to the same novel domain. Note that these novel domains should be treated with care, as some may belong to those that were already characterised in the original input domain assignment file but were filtered in *step 2*.

Novel domains are given temporary IDs consisting of the string `NOVEL` and a five-digit numerical identifier; for instance, `NOVEL00042` is the 42nd novel domain found during this process. Components containing less than four sequence fragments are not considered novel domains. The size threshold of connected components can be adjusted in the *configuration file*.

3.8 Step 7 – Consensus domain architecture

This step determines the final consensus domain architecture for each sequence by starting out from the preliminary domain architecture obtained in *step 2* and extending it with the novel domains found for the given sequence. The consensus domain architectures are written into two files, one containing a simpler flat-file representation of the consensus architectures suitable for further processing, while the other containing a detailed domain architecture description with InterPro IDs and human-readable descriptions for each domain in each sequence. This latter file also lists the primary data source for the sequence, the coverage of the sequence with and without novel domains, and also the number of the stage in which each domain was selected into the consensus assignment.

3.9 Step 8 – Functional label assignment

This step tries to assign a functional label to every sequence by looking at the list of its domains and collecting the corresponding Gene Ontology terms using a mapping file that assigns Gene Ontology terms to InterPro IDs. Such a file can be obtained from the InterPro2GO project. For each sequence, the collected Gene Ontology terms are filtered such that only those terms are kept which are either leaf terms (i.e. they have no descendants in the GO tree) or none of their descendants are included in the set of collected terms. These terms are then written in decreasing order of specificity to an output file, where specificity is assessed by the number of domains a given term is assigned to in the InterPro2GO mapping file; terms assigned to a smaller number of domains are considered more specific.

3.10 Step 9 – Overrepresentation analysis

This optional step conducts a Gene Ontology overrepresentation analysis on the domain architecture of the sequences given in the input file. For each sequence, we find the Gene Ontology terms corresponding to each of the domains in the consensus domain architecture of the sequence, and check each term using a hypergeometric test to determine whether it is overrepresented within the annotations of the sequence domains or not.

During the overrepresentation analysis, *multiple* hypergeometric tests are performed to determine the significantly overrepresented terms for a *single* sequence. GFam lets the user account for the effects of multiple hypothesis testing by correcting the p-values either by controlling the family-wise error rate (FWER) using the Bonferroni or Sidák methods, or by controlling the false discovery rate (FDR) using the Benjamini-Hochberg method.

The result of the overrepresentation analysis is saved into a human-readable text file that lists the overrepresented Gene Ontology terms in increasing order of p-values for each sequence.

CHAPTER

FOUR

SUPPLEMENTARY SCRIPTS

The scripts described in this chapter are not parts of the main GFam pipeline, but they provide useful extra functionality nevertheless. These scripts are found in the `bin` subdirectory of GFam, and they can be run separately from the command line, provided that GFam itself is on the Python path. Since the current directory is always on the Python path, it is best to run these scripts from the root directory of GFam.

4.1 Plotting descriptive statistics of the input file

The GFam pipeline has several parameters (e.g., E-value and domain length thresholds) with sensible default values, but in order to achieve the best results on a given dataset, these parameters can be adapted to the properties of the input data if necessary. Such decisions are made by humans after inspecting the distribution of E-values and domain lengths, and the distribution of overlap sizes between different domains in the InterPro domain assignment file. GFam can readily generate these plots using Matplotlib, a plotting library for Python. Matplotlib is available as a package in all major Linux distributions, and the project provides an installer for Microsoft Windows and Mac OS X.

The script can be invoked as follows (assuming that the configuration file is named `gfam.cfg`):

```
$ bin/plot.py -c gfam.cfg figurename
```

where *figurename* is the name of the figure to be plotted. You may also save figures to an output file:

```
$ bin/plot.py -c gfam.cfg -o *outfile*.pdf figurename1 figurename2 ...
```

The supported output formats include PDF, PNG, JPG and SVG, provided that the corresponding Matplotlib backends are installed. ASCII art representations of the histograms may also be printed if the extension of the output file is `.txt`.

To get a list of the supported figure names, specify `list` in place of the figure name:

```
$ bin/plot.py -c gfam.cfg list
```

The supported figures are as follows:

`evalue_distribution` Plots the count or relative frequency of domains with a given log E-value, sorted by different data sources in the InterPro input file, using a bin for each integer log E-value.

`length_distribution` Plots the count or relative frequency of domains with a given length, sorted by different data sources in the InterPro input file, using 25 bins up to a length of at most 750. The rightmost column contains all domains with length greater than 750.

`overlap_distribution` Plots the count or relative frequency of overlap length between all pairs of domains that overlap by at least one residue and have the same data source. The plots are sorted by data sources and use a bin width of 5.

4.1.1 Command line options

-a FILE, --assignment-file=FILE If you don't have a GFam configuration file or you want to run the script on a different InterPro assignment file (not the one specified in the configuration file), you may specify the name of the InterPro file directly using this switch. In this case, `-c` is not needed.

--cumulative Plot cumulative distributions (if that makes sense for the selected plot).

-o FILE, --output=FILE Specify the name of the file to save the plots to. The desired format of the file is inferred from its extension. Supported formats: PNG, JPG, SVG and PDF (assuming that the required Matplotlib backends are installed). You may also use a `.txt` extension here, which turns on `--text-mode` automatically.

--relative Plot relative frequencies instead of absolute counts on the Y axis (if that makes sense for the selected plot), and use a line chart instead of a bar chart.

--survival Plot survival distributions (if that makes sense for the selected plot), and use a line chart instead of a bar chart.

-t, --text-mode Print an ASCII art representation of each histogram. This option is useful if you are sitting at a non-graphical terminal (e.g., an ssh shell) or if you want to dump the histograms to a text file that you can analyze later. This option is turned on automatically if the extension of the output file is `.txt`.

4.2 Updating the mapping of IDs to human-readable names

GFam relies on an external tab-separated flat file to map domain IDs to human-readable descriptions when producing the final output. Such a file should contain at least the InterPro, Pfam, SMART and Superfamily IDs. The GFam distribution contains a script that can download the mappings automatically from known sources on the Internet. The script can be invoked as follows:

```
$ bin/download_names.py >data/names.dat
```

This will download the InterPro, Pfam, SMART and Superfamily IDs from the Internet and prepare the appropriate name mapping file in `data/names.dat`. If you wish to put it elsewhere, simply specify a different output file name. If you omit the trailing `>data/names.dat` part, the mapping will be written into the standard output. You can also compress the mapping file on-the-fly using `gzip` or `bzip2` and use the compressed file directly in the configuration file as GFam will uncompress it when needed. The following command constructs a compressed name mapping file:

```
$ bin/download_names.py | gzip -9 >data/names.dat.gz
```

Note that the script relies on the following locations to download data:

- <ftp://ftp.ebi.ac.uk/pub/databases/interpro/names.dat> for the InterPro name mapping
- <http://pfam.sanger.ac.uk/families?output=text> for the Pfam name mapping
- <http://smart.embl-heidelberg.de/smart/descriptions.pl> for the SMART name mapping
- <http://scop.mrc-lmb.cam.ac.uk/scop/parse/> for the SCOP description files (named `dir.des.scop.txt_X.XX`, where `X.XX` stands for the SCOP version number). It also relies on the most recent version of the SCOP description file being linked from the above page. The script will simply scan the links of the above page to determine what is the most recent version of SCOP. If the version number cannot be determined, the script will silently skip downloading the SCOP IDs.

CHAPTER

FIVE

API DOCUMENTATION

5.1 gfam – The main module

This is the main module of GFam.

On its own, this module contains nothing, all the functionality is implemented in one of the following submodules:

gfam.assignment Contains routines related to assignments, i.e. the objects describing the fact that a given section of a sequence is assigned to a given domain.

gfam.blast Classes and functions related to handling BLAST output files and external BLAST utilities.

gfam.compat Classes and functions to maintain compatibility with Python 2.5.

gfam.config An extension of Python's built-in `optparse` module to allow supplying default values for command line options from a configuration file.

gfam.enum A simple enumeration class using some metaclass magic. Sadly enough, Python does not have a built-in and flexible enumeration class like Java does.

gfam.fasta A simple parser and emitter for the FASTA sequence format. Yes, we could have used BioPython instead, but this way we saved a dependency.

gfam.go Classes and functions related to the Gene Ontology.

gfam.interpro Classes and functions related to parsing InterPro files, especially the output of `iprscan`.

gfam.modula A simple module that manages tasks and dependencies of the GFam pipeline. It will ensure that a GFam execution does not run long calculations unnecessarily if the same results are available from a previous interrupted GFam run.

gfam.scripts Implementations for the main steps of the GFam pipeline. Each submodule of this module can be executed on its own as a command-line utility.

gfam.sequence A simple replacement for the `Sequence` and `SeqRecord` classes of BioPython.

gfam.utils Various utility routines that did not fit anywhere else.

General comments that apply for the whole GFam API:

- Routines that accept files usually accept either filenames or file-like objects. If the filename ends in `.bz2` or `.gz`, it will be decompressed on-the-fly in memory. If the filename starts with `http://`, `https://` or `ftp://`, it is assumed to be remote object and will be downloaded accordingly. This is achieved by `gfam.utils.open_anything`, which is called whenever a filename or a file-like object is passed into a function.

5.2 gfam.assignment – Routines related to sequence-domain annotations

Classes corresponding to domain assignments (Assignment) and sequences with corresponding domain assignments (SequenceWithAssignments).

class gfam.assignment.**Assignment**
Class representing a record in an InterPro iprscan output.

An InterPro domain assignment has the following fields:

- id: the ID of the sequence
- length: the length of the sequence
- start: the starting position of the region in the sequence that is assigned to some InterPro domain (inclusive)
- end: the ending position (inclusive)
- source: the assignment source as reported by iprscan
- domain: the ID of the domain being assigned, according to the assignment source
- evalue: E-value of the assignment if that makes sense, None otherwise
- interpro_id: the InterPro ID corresponding to domain in source.
- comment: an arbitrary comment

get_assigned_length()
Returns the number of amino acids covered by the assignment within the sequence.

resolve_interpro_ids(*interpro*)
If the assignment has an InterPro ID, this method makes sure that the domain is equal to the highest common ancestor of the InterPro ID in the InterPro tree. If the assignment does not have an InterPro ID yet, this method tries to look it up.

Returns a new tuple which might or might not be equal to this one.

short_repr()
Short representation of this assignment, used in error messages

class gfam.assignment.**AssignmentOverlapChecker**
Static class that contains the central logic of determining whether an assignment can be added to a partially assigned SequenceWithAssignments.

The class has a class variable named max_overlap which stores the maximum allowed overlap size. This is 20 by default.

classmethod check(*sequence*, *assignment*)
Checks whether an assignment can be added to a partially assigned sequence. sequence must be an instance of SequenceWithAssignments, assignment must be an instance of Assignment.

The output is equivalent to the output of the first check_single that returns anything different from OverlapType.NO_OVERLAP, or OverlapType.NO_OVERLAP otherwise.

classmethod check_single(*assignment*, *other_assignment*)
Checks whether the given assignment overlaps with another assignment other_assignment. Returns one of the following:

- OverlapType.NO_OVERLAP: there is no overlap between the two given assignments

- `OverlapType.DUPLICATE`: `assignment` is a duplicate of `other_assignment` (same starting and ending positions)
- `OverlapType.INSERTION`: there is a complete domain insertion in either direction
- `OverlapType.INSERTION_DIFFERENT`: `assignment` is inserted into `other_assignment` or vice versa, but they have different data sources.
- `OverlapType.DIFFERENT`: `other_assignment` overlaps with `assignment` partially, but they have different data sources
- `OverlapType.OVERLAP`: `other_assignment` overlaps with `assignment` partially, they have the same data source, but the size of the overlap is larger than the maximum allowed overlap specified in `AssignmentOverlapChecker.max_overlap`.

classmethod **`get_overlap_size`**(*assignment*, *other_assignment*)
Returns the length of the overlap between the given `assignment` and another (`other_assignment`). It is assumed (and not checked) that the two assignments refer to the same sequence.

`max_overlap` **= 20**
The maximum allowed overlap size.

class `gfam.assignment.`**`OverlapType`**(*key*, *value*, ***kwds*)
Enum describing the different overlap types that can be detected by `AssignmentOverlapChecker`. See the documentation of `AssignmentOverlapChecker.check_single` for more details.

class `gfam.assignment.`**`SequenceWithAssignments`**(*name*, *length*)
Class representing a sequence for which some parts are assigned to InterPro domains.

The class has the following fields:

- `name`: the name of the sequence
- `length`: the number of amino acids in the sequence
- `assignments`: a list of `Assignment` instances that describe the domain architecture of the sequence

`acceptable_overlaps` **= set([OverlapType.INSERTION, OverlapType.NO_OVERLAP])**
Acceptable overlap types which we allow in assignments. By default, we consider complete domain insertions and the absence of overlaps acceptable.

`assign`(*assignment*, *overlap_check=True*)
Assigns a fragment of this sequence using the given assignment. If `overlap_check` is `False`, we will not check for overlaps or conflicts with existing assignments.

Returns `True` if the assignment was added, `False` if it wasn't due to an overlap conflict.

`assign_`(*start*, *end*, *domain*, *source='Novel'*, **args*, ***kwds*)
Assigns a fragment of this sequence to the given domain. `start` and `end` are the starting and ending positions, inclusive. `domain_name` is the name of the domain, `source` is the assignment source (`Novel` by default).

`coverage`(*sources=None*)
Returns the coverage of the sequence, i.e. the fraction of residues covered by at least one assignment.

`sources` specifies the data sources to be included in the coverage calculation. If `None`, all the data sources will be considered; otherwise it must be a set containing the accepted sources.

`data_sources`()
Returns the list of data sources that were used in this assignment.

`domain_architecture`(*sources=None*)
Returns the domain architecture of the assignment.

The domain architecture is a list which contains the IDs of the assigned regions (domains) in ascending order of their starting positions. If `sources` is `None`, all data sources will be considered; otherwise it must be a set or iterable which specifies the data sources to be included in the result.

is_completely_unassigned(*start*, *end*)
Checks whether the given region is completely unassigned. start and end positions are both inclusive

overlap_checker
The overlap checker used by this instance. This points to `AssignmentOverlapChecker` by default.

alias of `AssignmentOverlapChecker`

resolve_interpro_ids(*interpro*)
Calls `Assignment.resolve_interpro_ids` on each assignment of this sequence

unassigned_regions()
Returns a generator that iterates over the unassigned regions of the sequence. Each entry yielded by the generator is a tuple containing the start and end positions

class `gfam.assignment.`**EValueFilter**
Given an `Assignment`, this filter tells whether the assignment's E-value is satisfactory to accept it.

The filter supports different E-values for different data sources. By default, the E-value threshold is infinity for all data sources.

classmethod FromString(*description*)
Constructs an E-value filter from a string description that can be used in command line arguments and configuration files.

The string description is a semicolon-separated list of source-threshold pairs. For instance, the following is a valid description giving an E-value of 0.001 for HMMPfam sources, 0.005 for HMMSmart sources and 0.007 for everything else:

```
HMMPfam=0.001;HMMSmart=0.005;0.007
```

The last entry denotes the default E-value; in particular, if a number is not preceded by a source name, it is assumed to be a default E-value. If no default E-value is given, infinity will be used.

is_acceptable(*assignment*)
Checks whether the given assignment is acceptable.

This method looks up the E-value threshold corresponding to the `source` of the assignment and returns `True` if the E-value of the assignment is less than the threshold, `False` otherwise.

set_threshold(*source*, *evalue*)
Sets the E-value threshold for the given data source.

5.3 `gfam.blast` – Handling BLAST file formats and utilities

Classes and functions related to BLAST files and utilities

class `gfam.blast.`**BlastFilter**
Filters BLAST records, i.e. drops the ones that do not satisfy some criteria.

You can tune the filter with the following instance variables:

- `min_sequence_identity`: the minimum sequence identity required by the filter
- `min_alignment_length`: the minimum alignment length required by the filter
- `max_e_value`: the maximum E-value required by the filter

You can also ask the filter to normalize the alignment length to between zero and one by calling `set_normalize_func`.

accepts (*line*)
Returns `True` if the filter accepts the given line, `False` otherwise.

load_sequences (*seq_generator*)
Loads the sequences yielded by the sequence generator.

This method iterates over the given sequences and stores their names and lengths in an internal dict. The dict will be used later to normalize sequences by their lengths. It is necessary to call this method if you are using any of the normalizing functions.

load_sequences_from_file (*fname*)
Loads the sequences from the given file. The file must be in FASTA format. You are allowed to pass file pointers or names of gzipped/bzipped files here.

set_normalize_func (*name*)
Sets the normalizing function used by the filter when filtering by the length of match. The following normalizing functions are known:

- `off`: returns the unnormalized match length
- `smaller`: divides the match length by the length of the smaller sequence
- `larger`: divides the match length by the length of the larger sequence
- `query`: divides the match length by the length of the query sequence
- `hit`: divides the match length by the length of the hit sequence

Alternatively, you may pass a callable in place of the function name. The callable must accept four arguments, the first is the `BlastFilter` object itself, the second is the length of the query sequence, the third is the length of the hit sequence, the fourth is the unnormalized match length.

5.4 `gfam.compat` – Compatibility classes for Python 2.5

Compatibility classes for Python 2.5 and earlier.

class `gfam.compat.`**Mapping**
An abstract base class similar to `collections.Mapping` in Python 2.6. This will be used in place of `collections.Mapping` if necessary.

`gfam.compat.`**namedtuple** (*typename*, *field_names*, *verbose=False*, *rename=False*)
Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', 'x y')
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                       # fields also accessable by name
33
>>> d = p._asdict()                 # convert to a dictionary
>>> d['x']
11
```

```
>>> Point(**d)                     # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)              # _replace() is like str.replace() but targets named fields
Point(x=100, y=22)
```

5.5 gfam.config – Configuration file handling

Configuration-related classes for GFam.

This module provides ConfigurableOptionParser, an extension of Python's built-in optparse.OptionParser that lets you provide default values for command-line options from a given configuration file.

class gfam.config.**ConfigurableOption**(**args*, ***kwds*)
An extension of the Option class of Python that stores a configuration key which can be used to fetch the value of the option if it is not given in the command line.

get_config_section_and_item()
Returns the section and item in which we have to look for the value of this option.

process(**args*, ***kwds*)
Method invoked by the command line parser when it sees this option on the command line.

This method simply makes note of the fact that the option was seen on the command line, and calls the superclass.

seen
Returns whether this option was seen on the command line.

class gfam.config.**ConfigurableOptionParser**(**args*, ***kwds*)
An extension of the OptionParser class of Python that also uses a config file.

Basically, the scripts of GFam can be driven both from command line arguments and configuration files. This class lets you specify the command line arguments using the same syntax you would use with OptionParser, but it also lets you attach a configuration key to each of the options. It also registers an option -c and a long option --config-file that can be used to specify the input configuration file. If an option is not present on the command line, this option parser will try to look up the corresponding configuration key in the given configuration file.

Configuration keys must be specified in slashed format (i.e. section/item).

This class also exposes an instance attribute named config, which contains the parsed configuration values from the specified configuration file. config will be an instance of ConfigParser or None if parse_args() was not called so far.

parse_args(**args*, ***kwds*)
Parses the command line and returns a tuple (options, args), where options contains the values of the parsed command line options (after adding the values from the config file for missing options) and args contains the list of positional arguments.

5.6 gfam.enum – A simple enumeration class

Simple enumeration class and metaclass.

class gfam.enum.**Enum**(*key*, *value*, ***kwds*)
An instance of an enumeration value and a class representing a whole enum.

This is mainly used as a superclass for enumerations. There is a clear distinction between using the class itself or using one of its instances. Using the class means that you are referring to the enum as a whole (with all its possible keys and values). Using one of the instances means that you are using a single key-value pair from the enum. Instances should never be created directly, as all the valid instances are accessible as attributes of the class itself.

Usage example:

```
>>> class Spam(Enum):
...     SPAM  = "Spam spam spam"
...     EGGS  = "Eggs"
...     BACON = "Bacon"
```

After you have defined an enum class like the one above, you can make use of it this way:

```
>>> Spam.BACON
Spam.BACON
>>> Spam.BACON.value
'Bacon'
```

Think about enums as Python dictionaries that map symbolic names to values. Enums even provide methods similar to the non-mutating methods of Python dictionaries:

```
>>> sorted(Spam.keys())
['BACON', 'EGGS', 'SPAM']
```

You can also get an instance of the enum from its symbolic name or value:

```
>>> Spam.from_name("BACON")
Spam.BACON
>>> Spam.from_value("Spam spam spam")
Spam.SPAM
```

5.7 `gfam.fasta` – FASTA parser and emitter

This module contains routines for parsing and writing FASTA files.

It is not meant to be a full-fledged FASTA parser (check BioPython if you need one), but it works well in most cases, at least for neatly formatted FASTA files.

class `gfam.fasta.`**`Parser`**(*handle*)

Parser for FASTA files.

Usage example:

```
parser = Parser(open("test.ffa"))
for sequence in parser:
    print sequence.seq
```

It also works with remote FASTA files if you use the `urllib2` module:

```
from urllib2 import urlopen
parser = Parser(urlopen("ftp://whatever.org/remote_fasta_file.ffa"))
for sequence in parser:
    print sequence.seq
```

`sequences`()

Returns a generator that iterates over all the sequences in the FASTA file. The generator will yield `SeqRecord` objects.

classmethod to_dict (**args*, ***kwds*)
Creates a dictionary out of a FASTA file.

The resulting dictionary will map sequence IDs to their corresponding `SeqRecord` objects. The arguments are passed on intact to the constructor of `Parser`.

Usage example:

```
seq_dict = Parser.to_dict("test.ffa")
```

`gfam.fasta.`**regexp_remapper** (*iterable*, *regexp=None*, *replacement='\\g<id>'*)
Regexp-based sequence ID remapper.

This class takes a sequence iterator as an input and remaps each ID in the sequence using a call to `re.sub`. `iterable` must yield `SeqRecord` objects; `regexp` is the regular expression matching the part to be replaced, `replacement` is the replacement string that may contain backreferences to `regexp`.

If `regexp` is `None` or an empty string, returns the original iterable.

class `gfam.fasta.`**Writer** (*handle*)
Writes `SeqRecord` objects in FASTA format to a given file handle.

write (*seq_record*)
Writes the given sequence record to the file handle passed at construction time.

5.8 `gfam.go` – Handling the Gene Ontology

A higher level Gene Ontology representation in Python

class `gfam.go.`**Annotation** (**args*, ***kwds*)
Class representing a GO annotation (possibly parsed from an annotation file).

The class has the following attributes (corresponding to the columns of a GO annotation file):

- `db`: refers to the database from which the identifier in the next column (`db_object_id`) is drawn
- `db_object_id`: a unique identifier in `db` for the item being annotated.
- `db_object_symbol`: a unique and valid symbol to which `db_object_id` is matched. Usually a symbol that means something to a biologist.
- `qualifiers`: a list of flags that modify the interpretation of the annotation (e.g., `NOT`). Note that this is always a list, even when no qualifier exists.
- `go_id`: the GO identifier for the term attributed to `db_object_id`.
- `db_references`: a list of unique identifiers for a single source cited as an authority for the attribution of the `go_id` to the `db_object_id`. May be a literature or a database record.
- `evidence_code`: the GO evidence code
- `with`: required for some evidence codes. Holds an additional identifier for certain evidence codes.
- `aspect`: one of `P` (biological process), `F` (molecular function) or `C` (cellular compartment).
- `db_object_name`: name of gene or gene product
- `db_object_synonyms`: a gene symbol or some other human-readable text
- `db_object_type`: what kind of thing is being annotated. This is either gene (`SO:0000704`), transcript (`SO:0000673`), protein (`SO:0000358`), `protein_structure` or `complex`.
- `taxons`: taxonomic identifiers (at most two, at least 1). This is always a list

- `date`: date on which the annotation was made
- `assigned_by`: the database which made the annotation.

Constructs an annotation. Use keyword arguments to specify the values of the different attributes. If you use positional arguments, the order of the arguments must be the same as they are in the GO annotation file. No syntax checking is done on the values entered, but attributes with a maximum cardinality more than one are converted to lists automatically. (If you specify a string with vertical bar separators as they are in the input file, the string will be splitted appropriately).

class `gfam.go.`**`AnnotationFile`**(*file_handle*)
A parser class that processes GO annotation files.

Creates an annotation file parser that reads the given file-like object. You can also specify filenames. If the filename ends in `.gz`, the file is assumed to contain gzipped data and it will be unzipped on the fly. Example:

```
>>> import gfam.go as go
>>> parser = go.AnnotationFile("gene_association.sgd.gz")
```

To read the annotations in the file, you must iterate over the parser as if it were a list. The iterator yields `Annotation` objects.

`annotations`()
Iterates over the annotations in this annotation file, yielding an `Annotation` object for each annotation.

class `gfam.go.`**`Tree`**
Class representing the GO tree. A GO tree contains many GO terms represented by `Term` objects.

`add`(*term*)
Adds a `Term` to this GO tree

`add_alias`(*canonical*, *alias*)
Adds an alias to the given canonical term in the GO tree

`ancestors`(**args*)
Returns all the ancestors of a given `Term` (or multiple terms) in this tree. The result is a list of `Term` instances.

`ensure_term`(*term_or_id*)
Given a `Term` or a GO term ID, returns an object that is surely a `Term`

classmethod `from_obo`(*fp*)
Constructs a GO tree from an OBO file. `fp` is a file pointer to the OBO file we want to use

`lookup`(*identifier*)
Looks up a `Term` in this tree by ID. Also cares about alternative IDs

`parents`(*term_or_id*)
Returns the direct parents of a `Term` in this tree. `term_or_id` can be a GO term ID or a `Term`. The result is a list of `Term` instances.

`paths_to_root`(**args*)
Finds all the paths from a term (or multiple terms if multiple arguments are used) to the root.

`to_igraph`(*rel='is_a'*)
Returns an `igraph` graph representing this GO tree. This is handy if you happen to use igraph.

class `gfam.go.`**`Term`**(*id*, *name=''*, *tags=None*)
Class representing a single GO term

Constructs a GO term with the given ID, the given human- readable name and the given tags.

classmethod from_stanza(*stanza*)
Constructs a GO term from a stanza coming from an OBO file. `stanza` must be an instance of `gfam.go.obo.Stanza`.

5.8.1 gfam.go.obo – Parsing OBO ontology files

A very simple and not 100% compliant parser for the OBO file format.

This parser is supplied "as is"; it is not an official parser, it might puke on perfectly valid OBO files, it might parse perfectly invalid OBO files, it might steal your kitten or set your garden shed on fire. Apart from that, it should be working, or at least it should be in a suitable condition to parse the Gene Ontology, which is my only test case anyway.

Usage example:

```
import gfam.go.obo
parser = gfam.go.obo.Parser(open("gene_ontology.1_2.obo"))
gene_ontology = {}
for stanza in parser:
    gene_ontology[stanza.tags["id"][0]] = stanza.tags
```

exception gfam.go.obo.**ParseError**(*msg*, *lineno=1*)
Exception thrown when a parsing error occurred

class gfam.go.obo.**Stanza**(*name*, *tags=None*)
Class representing an OBO stanza.

An OBO stanza looks like this:

```
[name]
tag: value
tag: value
tag: value
```

Values may optionally have modifiers, see the OBO specification for more details. This class stores the stanza name in the `name` member variable and the tags and values in a Python dict called `tags`. Given a valid stanza, you can do stuff like this:

```
>>> stanza.name
"Term"
>>> print stanza.tags["id"]
['GO:0015036']
>>> print stanza.tags["name"]
['disulfide oxidoreductase activity']
```

Note that the `tags` dict contains lists associated to each tag name. This is because theoretically there could be more than a single value associated to a tag in the OBO file format.

class gfam.go.obo.**Parser**(*file_handle*)
The main attraction, the OBO parser.

Creates an OBO parser that reads the given file-like object. If you want to create a parser that reads an OBO file, do this:

```
>>> import gfam.go.obo
>>> parser = gfam.go.obo.Parser(open("gene_ontology.1_2.obo"))
```

Only the headers are read when creating the parser. You can access these right after construction as follows:

```
>>> parser.headers["format-version"]
['1.2']
```

To read the stanzas in the file, you must iterate over the parser as if it were a list. The iterator yields `Stanza` objects.

stanzas()
Iterates over the stanzas in this OBO file, yielding a `Stanza` object for each stanza.

class gfam.go.obo.**Value**(*value*, *modifiers=()*)
Class representing a value and its modifiers in the OBO file

This class has two member variables. `value` is the value itself, `modifiers` are the corresponding modifiers in a tuple. Currently the modifiers are not parsed in any way, but this might change in the future.

5.8.2 gfam.go.overrepresentation – Overrepresentation analysis

Overrepresentation analysis of Gene Ontology terms

class gfam.go.overrepresentation.**OverrepresentationAnalyser**(*tree*, *mapping*, *confidence=0.05*, *min_count=5*, *correction='fdr'*)
Performs overrepresentation analysis of Gene Ontology terms on sets of entities that are annotated by some GO terms.

Initializes the overrepresentation analysis algorithm by associating it to a given Gene Ontology tree and a given mapping from entities to their respective GO terms.

`tree` must be an instance of `gfam.go.Tree`. `mapping` must be a bidirectional dictionary object (`gfam.utils.bidict`) that maps entities to GO terms and vice versa. For `mapping`, if an entity is annotated by a GO term, it is not necessary to list all the ancestors of that GO term for that entity, this will be taken care of by the class itself which always works on the copy of the given mapping.

`confidence` is the confidence level of the test.

`min_count` specifies which GO terms are to be excluded from the overrepresentation analysis; if a GO term occurs less than `min_count` times in `mapping`, it will not be considered.

`correction` specifies the multiple hypothesis testing correction to be used and it must be one of the following:

- `None`, `"none"` or `"off"`: no correction
- `"fdr"`: controlling the false discovery rate according to the method of Benjamini and Hochberg
- `"bonferroni"`: Bonferroni correction of the family-wise error rate
- `"sidak"`: Sidak correction of the family-wise error rate

enrichment_p(*term_or_id*, *count*, *group_size*)
Calculates the enrichment p-score of the given GO term or ID (`term_or_id`) if it occurs `count` times in a group of size `group_size`.

test_counts(*counts*, *group_size*)
Given a dict that maps Gene Ontology terms to their occurrence counts and the number of entities in the group from which these term counts originate, calculates a list of overrepresented Gene Ontology terms.

test_group(*group*)
Overrepresentation analysis of the given group of objects. `group` must be an iterable yielding objects that are in `self.mapping.left`.

5.9 gfam.interpro – Handling InterPro-related files

Classes related to handling InterPro-related files in HyFam

class gfam.interpro.**AssignmentReader**(*filename*)
Iterates over assignments in an InterPro domain assignment file.

This reader parses the output of iprscan and yields appropriate Assignment instances for each line.

assignments()
A generator that yields the assignments in the InterPro domain assignment file one by one. Each object yielded by this generator will be an instance of Assignment.

assignments_and_lines()
A generator that yields the assignments in the InterPro domain assignment file and the corresponding raw lines one by one. Each object yielded by this generator will be a tuple containing an instance of Assignment and the corresponding line.

parse_line(*line*)
Parses a single line from an InterPro domain assignment file and returns a corresponding Assignment instance.

class gfam.interpro.**InterPro**
Class that encapsulates the InterPro parent-child tree (an instance of InterProTree) and the InterPro ID mapper (an instance of InterProIDMapper) under the same hood. This makes it easier to pass both of them around in the code.

When parsing the parent-child tree, subfamiy IDs are removed from aliases starting with PTHR; i.e. PTHR10829:SF4 will be stored as PTHR10829 (and mapped to IPR015503 at the time of writing).

For usage examples, see InterProTree and InterProIDMapper.

classmethod FromFile(*filename*)
Constructs this object from an InterPro parent-child mapping file, pointed to by the given filename. Both the tree and the ID-name mapping will be built from the same file.

class gfam.interpro.**InterProIDMapper**
Dict-like object that maps domain IDs from various data sources to their corresponding InterPro IDs.

This class is usually not constructed directly; an instance of this class is a member of every instance of InterPro.

The class can generally be used like a dictionary:

```
>>> interpro = InterPro.FromFile("data/ParentChildTreeFile.txt")
>>> mapper = interpro.mapping
>>> mapper["PF00031"]          # an alias for IPR000010
'IPR000010'
>>> mapper["IPR9999"]          # no such ID
Traceback (most recent call last):
    ...
KeyError: 'IPR9999'
>>> mapper.get("IPR9999")      # no such ID
'IPR9999'
>>> mapper["IPR9999"] = "Fake ID for testing"
>>> mapper["IPR9999"]
'Fake ID for testing'
>>> del mapper["IPR9999"]
>>> "IPR9999" in mapper
False
```

get(*key*, *default=None*)
Returns the InterPro ID corresponding to the given key or the given default value if no InterPro ID exists for that key. If the default value is None, returns the key itself if there is no InterPro ID for the key

class gfam.interpro.**InterProNames**
Dict-like object mapping IDs to human-readable names, the only difference being that unknown IDs are handled gracefully instead of raising a `KeyError`.

The name of this class is a bit of a misnomer as it works for *any* type of IDs, not only for InterPro IDs.

Usage example:

```
>>> names = InterProNames.FromFile("data/names.dat.gz")
>>> "IPR015503" in names
True
>>> "no-such-name" in names
False
>>> names["IPR015503"]
'Cortactin'
>>> names["no-such-name"]
'no-such-name'
```

classmethod FromFile(*filename*)
Shortcut method that does exactly what the following snippet does:

```
>>> names = InterProNames()
>>> names.load(filename)
```

load(*filename*)
Loads ID-name assignments from a simple tab-separated flat file.

Lines not containins any tab characters are silently ignored.

class gfam.interpro.**InterProTree**
Dict-like object that tells the parent ID corresponding to every InterPro domain ID.

This class is usually not constructed directly; an instance of this class is a member of every instance of `InterPro`.

The class behaves much like a dictionary that returns the parent ID for every InterPro domain ID. For unknown domain IDs, the domain ID itself is returned. For sub-subfamilies, the returned value is the ID of the subfamily, not the family; use `get_most_remote_ancestor` if you always need the family ID no matter what.

Usage example:

```
>>> interpro = InterPro.FromFile("data/ParentChildTreeFile.txt")
>>> tree = interpro.tree
>>> tree["IPR000010"]      # this is a family ID
'IPR000010'
>>> tree["IPR001713"]      # this is a subfamily of IPR000010
'IPR000010'
>>> tree["IPR001321"]      # a sub-subfamily ID
'IPR013655'
>>> tree["IPR9999"]        # this is an unknown ID
'IPR9999'
>>> "IPR9999" in tree
False
>>> tree.get_most_remote_ancestor("IPR001321")
'IPR000014'
```

get_most_remote_ancestor(*item*)
Returns the most remote ancestor of the given item.

For family IDs, this returns the ID itself. For subfamily IDs and below, this returns the corresponding family ID.

class gfam.interpro.**InterPro2GOMapping**
Bidirectional dictionary (bidict) that tells the corresponding Gene Ontology terms of every InterPro ID and vice versa.

This object encapsulates two dictionaries: self.terms gives the Gene Ontology terms of a given InterPro domain, and self.domains gives the InterPro domains annotated by a given GO term.

add_annotation(*interpro_id*, *go_id*)
Adds a single Gene Ontology ID to the list of Gene Ontology IDs for a given InterPro ID

classmethod from_file(*filename*, *tree*)
Constructs a mapping from a mapping file. The format of this file should be identical to the official interpro2go file provided by the Gene Ontology project. tree is a Gene Ontology tree object (see gfam.go.Tree) that will be used to look up terms from IDs.

5.10 gfam.modula – Modular calculation framework

Modula is a modular calculation framework for Python that allows you to define tasks that depend on input files and on each others. Modula will figure out which tasks have to be executed in which order in order to calculate the final results – this is done by a simple depth first search on the task dependency graph.

Modula started out as a separate project, and you don't have to know its internals in order to use the GFam API. The only reason why it has been placed as a submodule of GFam is to avoid foricng users to install Modula separately. The Modula API is not documented here as it is not an internal part of GFam. Modula is used only by the GFam master script (see gfam.scripts.master) to execute the calculation steps in the proper order.

5.11 gfam.sequence – Simple sequence and sequence record classes

This module contains a simple Sequence and SeqRecord (sequence record) class. Sequence instances are just strings for the time being, while a SeqRecord encapsulates a Sequence with some associated metadata.

class gfam.sequence.**SeqRecord**(*seq*, *id='<unknown id>'*, *name='<unknown name>'*, *description='<no description>'*)
A sequence with its associated metadata.

This class is very close to BioPython's SeqRecord class; however, there is no guarantee about API compatibility.

The class has the following fields:

- seq: the sequence itself, an instance of Sequence.
- id: a short, unique ID for the sequence
- name: the human-readable name of the sequence
- description: the whole description line of the sequence as parsed from the original data source (such as a FASTA file). The FASTA writer (gfam.fasta.Writer) uses this when writing the sequence record to a file (unless if there is no description, in which case the ID is used).

class gfam.sequence.**Sequence**
String representing a sequence.

This class tries to be compatible with BioPython's `Sequence` class; however, there is no guarantee that it will always stay so.

For the time being, this class is essentially equivalent to a Python string.

5.12 `gfam.scripts` – Command line scripts

Classes and utilities commonly used in GFam command line scripts.

The scripts implementing the individual steps of the GFam pipeline are designed in a way that they can either be invoked in standalone mode (like any other command line script) or they can be instantiated and used from other Python modules. Each command line script is derived from the `CommandLineApp` class, which takes care of implementing functionality common for all the scripts, such as:

- providing a command line parser (an instance of `ConfigurableOptionParser`)
- providing a logger instance
- defining methods for extending the default option parser and for signaling fatal errors to the caller

class `gfam.scripts.`**`CommandLineApp`**(*logger=None*)

Generic command line application class that provides common functionality for all GFam command line scripts.

Creates a command line script instance that will use the given logger to log messages. If `logger` is `None`, it will create a logger exclusively for the script.

The following fields will be set up:

- `self.log`: the logger where messages should be logged to
- `self.parser`: an instance of `ConfigurableOptionParser` to parse the command line options. This is set up in `create_parser()`, not in the constructor.
- `self.options`: the parsed command line options. Parsing occurs when `run()` is called, so this field contains `None` when `run()` has not been called.
- `self.args`: the positional arguments from the parsed command line.

`create_logger`()

Creates a logger for the application and returns it.

`create_parser`()

Creates a command line parser for the application.

By default, the parser will use the class docstring as help string and add two options to the parser: `-v` specifies verbose logging mode, while `-d` specifies debug mode. In verbose mode, log messages having a level above or equal to `logging.INFO` are printed. In debug mode, all log messages (including debug messages) are printed. The default is to print warnings and errors only.

The created command line parser will be returned to the caller.

`error`(*message*)

Signals a fatal error and shuts down the application.

`run`(*args=None*)

Runs the application. This method processes the command line using the command line parser and as such, it should not be overridden in child classes unless you know what you are doing. If you want to implement the actual logic of your application, override `run_real` instead.

`args` contains the command line arguments that should be parsed. If `args` is `None`, the arguments will be obtained from `sys.argv[1:]`.

Each step in the GFam pipeline is implemented in a separate submodule of `gfam.scripts`. These submodules contain only a single class per submodule, derived from `gfam.scripts.CommandLineApp`. The submodules are invoked automatically in the right order by a master script in `gfam.scripts.master`. In general, you only have to invoke the master script and it will do the rest for you, but some of the steps might be useful on their own, so they can be invoked independently from the command line as:

```
$ python -m gfam.scripts.modulename
```

where *modulename* is the name of the submodule to be executed. You can get usage information for each submodule by typing:

```
$ python -m gfam.scripts.modulename --help
```

5.13 `gfam.utils` – Utility classes and functions

Common routines and utility classes for GFam that fit nowhere else.

class `gfam.utils.`**`bidict`**(*items=None*)

Bidirectional many-to-many mapping.

This class models many-to-many mappings that are used in some places in GFam. For instance, the mapping of GO term identifiers to InterPro domain identifiers is typically a many-to-many mapping (domains are annotated by multiple GO terms, and GO terms may belong to multiple domains).

Being a general many-to-many mapping class, instances contain two member dictionaries: `left` (which maps items from one side of the relationship to *sets* of items from the other side of the relationship) and `right` (which contains the exact opposite). You should only query these dictionaries directly, manipulation should be done by the methods provided by the `bidict` class to ensure that the two dicts are kept in sync.

Example:

```
>>> bd = bidict()
>>> bd.add_left("foo", "bar")
>>> bd.add_left("foo", "baz")
>>> bd.get_left("foo") == set(['bar', 'baz'])
True
>>> bd.add_right("baz", "frob")
>>> bd.get_right("bar")
set(['foo'])
>>> bd.get_right("baz") == set(['foo', 'frob'])
True
>>> bd.len_left()
2
>>> bd.len_right()
2
```

`add_left`(*v1*, *v2*)

Adds a pair of items `v1` and `v2` to the mapping s.t. `v1` as a left item is mapped to `v2` as a right item.

`add_left_multi`(*v1*, *v2s*)

Associates multiple items in `v2s` to `v1` when `v1` is interpreted as a left item

`add_right`(*v1*, *v2*)

Adds a pair of items `v1` and `v2` to the mapping s.t. `v1` as a right item is mapped to `v2` as a left item.

`add_right_multi`(*v2*, *v1s*)

Associates multiple items in `v1s` to `v2` when `v2` is interpreted as a right item

get_left(*v1*, *default=None*)
Returns the items associated to `v1` when `v1` is looked up from the left dictionary. `default` will be returned if `v1` is not in the left dictionary.

get_right(*v1*, *default=None*)
Returns the items associated to `v1` when `v1` is looked up from the right dictionary. `default` will be returned if `v2` is not in the right dictionary.

iteritems_left()
Iterates over the left dictionary

iteritems_right()
Iterates over the right dictionary

len_left()
Returns the number of unique left items

len_right()
Returns the number of unique right items

class gfam.utils.**complementerset**(*iterable=()*)
This object behaves more or less like a set, with one exception, the membership checking. For a `complementerset` object, you can define the elements which are *not* in the set, everything else is contained in it. The semantics of the operators are the same as for sets.

Usage example:

```
>>> s = complementerset()
>>> "abc" in s
True
>>> s in s
True
>>> s -= set(["abc"])
>>> s
complementerset(['abc'])
>>> "abc" in s
False
```

Constructs a complementer set that contains everything except the members of the given iterable.

difference_update(**args*)
Removes all elements of another set from this set.

Example:

```
>>> s = complementerset([1,2])
>>> s.difference_update([4,5])
>>> print s
complementerset([1, 2, 4, 5])
>>> s.difference_update([2], [1,6], [7,5,"spam"])
>>> print any(item in s for item in [2,1,6,7,5,"spam",4])
False
```

discard(*member*)
Removes an element from the complementer set if it is a member.

Example:

```
>>> s = complementerset()
>>> s.discard(2)
>>> print s
complementerset([2])
```

```
>>> s.discard(2)
>>> print s
complementerset([2])
```

iterexcluded()
Iterates over the items excluded from the complementer set.

Example:

```
>>> s = complementerset([5, 7, 4])
>>> print sorted(list(s.iterexcluded()))
[4, 5, 7]
```

remove(*member*)
Removes an element from the complementer set; it must be a member. If the element is not a member, raises a `KeyError`.

Example:

```
>>> s = complementerset()
>>> s.remove(2)
>>> print s
complementerset([2])
>>> s.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 4, in ?
KeyError: 2
```

class gfam.utils.**Histogram**(*bin_width=1*, *data=None*)
Generic histogram class for real numbers

Example:

```
>>> h = Histogram(5)      # Initializing, bin width = 5
>>> h << [2,3,2,7,8,5,5,0,7,9]      # Adding more items
>>> print h
N = 10, mean +- sd: 4.8000 +- 2.9740
[ 0,  5): **** (4)
[ 5, 10): ****** (6)
```

Initializes the histogram with the given data set.

Parameters

- **bin_width** – the bin width of the histogram.
- **data** – the data set to be used. Must contain real numbers.

add(*num*, *repeat=1*)
Adds a single number to the histogram.

Parameters

- **num** – the number to be added
- **repeat** – number of repeated additions

add_many(*data*)
Adds a single number or the elements of an iterable to the histogram.

Parameters **data** – an iterable containing the data to be added

bin_width
Returns the bin width of the histogram

bins()
Generator returning the bins of the histogram in increasing order

Returns a tuple with the following elements: left bound, right bound, number of elements in the bin

clear()
Clears the collected data

mean
Returns the mean of the elements in the histogram

n
Returns the number of elements in the histogram

sd
Returns the standard deviation of the elements in the histogram

to_string(*max_width=78*, *show_bars=True*, *show_counts=True*)
Returns the string representation of the histogram.

`max_width` is the maximal width of each line of the string representation. `show_bars` specify whether the histogram bars should be shown, `show_counts` specify whether the histogram counts should be shown. If both are `False`, only a general descriptive statistics (number of elements, mean and standard deviation) is shown.

`max_width` may not be obeyed if it is too small.

var
Returns the variance of the elements in the histogram

gfam.utils.**open_anything**(*fname*, **args*, ***kwds*)
Opens the given file. The file may be given as a file object or a filename. If the filename ends in `.bz2` or `.gz`, it will automatically be decompressed on the fly. If the filename starts with `http://`, `https://` or `ftp://` and there is no other argument given, the remote URL will be opened for reading. A single dash in place of the filename means the standard input.

gfam.utils.**redirected**(**args*, ***kwds*)
Context manager that temporarily redirects some of the input/output streams.

`stdin` is the new standard input, `stdout` is the new standard output, `stderr` is the new standard error. `None` means to leave the corresponding stream unchanged.

Example:

```
with redirected(stdout=open("test.txt", "w")):
    print "Yay, redirected to a file!"
```

class gfam.utils.**RunningMean**(*n=0.0*, *mean=0.0*, *sd=0.0*)
Running mean calculator.

This class can be used to calculate the mean of elements from a list, tuple, iterable or any other data source. The mean is calculated on the fly without explicitly summing the values, so it can be used for data sets with arbitrary item count. Also capable of returning the standard deviation (also calculated on the fly)

RunningMean(n=0.0, mean=0.0, sd=0.0)

Initializes the running mean calculator. Optionally the number of already processed elements and an initial mean can be supplied if we want to continue an interrupted calculation.

@param n: the initial number of elements already processed @param mean: the initial mean @param sd: the initial standard deviation

add(*value*, *repeat=1*)
Adds the given value to the elements from which we calculate the mean and the standard deviation.

@param value: the element to be added @param repeat: number of repeated additions @return: the new mean and standard deviation as a tuple

add_many(*values*)
RunningMean.add(values)

Adds the values in the given iterable to the elements from which we calculate the mean. Can also accept a single number. The left shift (C{<<}) operator is aliased to this function, so you can use it to add elements as well:

```
>>> rm=RunningMean()
>>> rm << [1,2,3,4]
(2.5, 1.290994...)
```

@param values: the element(s) to be added @type values: iterable @return: the new mean

mean
Returns the current mean

result
Returns the current mean and standard deviation as a tuple

sd
Returns the current standard deviation

var
Returns the current variation

gfam.utils.**search_file**(*filename*, *search_path=None*, *executable=True*)
Finds the given `filename` in the given search path. If `executable` and we are on Windows, `.exe` will be appended to the filename. Returns the full path of the file or `None` if it is not found on the path.

gfam.utils.**temporary_dir**(**args*, ***kwds*)
Context manager that creates a temporary directory when entering the context and removes it when exiting.

Every argument is passed on to `tempfile.mkdtemp` except a keyword argument named `change` which tells whether we should change to the newly created temporary directory or not. The current directory will be restored when exiting the context manager.

class gfam.utils.**UniqueIdGenerator**(*id_generator=None*)
A dictionary-like class that can be used to assign unique integer IDs to names.

Usage:

```
>>> gen = UniqueIdGenerator()
>>> gen["A"]
0
>>> gen["B"]
1
>>> gen["C"]
2
>>> gen["A"]       # Retrieving already existing ID
0
>>> len(gen)       # Number of already used IDs
3
```

Creates a new unique ID generator. `id_generator` specifies how do we assign new IDs to elements that do not have an ID yet. If it is `None`, elements will be assigned integer identifiers starting from 0. If it is an integer, elements will be assigned identifiers starting from the given integer. If it is an iterator or generator, its `next` method will be called every time a new ID is needed.

reverse_dict()
Returns the reversed mapping, i.e., the one that maps generated IDs to their corresponding items

values()
Returns the list of items added so far. Items are ordered according to the standard sorting order of their keys, so the values will be exactly in the same order they were added if the ID generator generates IDs in ascending order. This hold, for instance, to numeric ID generators that assign integers starting from a given number.

CHAPTER

SIX

INDICES AND TABLES

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

PYTHON MODULE INDEX

g

INDEX

T

U

V

W