GAMBIT

Release 0.6.0

Jared Lumpe

Feb 16, 2023

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Installation and Setup

1.1.1 Install from Bioconda

The recommended way to install the tool is through the conda package manager (available here):

```
conda install -c bioconda gambit
```

1.1.2 Install from source

Installing from source requires the `cython` package as well as a C compiler be installed on your system. Clone the repository and navigate to the directory, and then run:

```
pip install .
```

Or do an editable development install with:

```
pip install -e .
```

1.1.3 Database files

Download files for the latest database release from the *Reference database releases* page and place them in a directory of your choice. The directory should not contain any other files with the same extension.

1.2 Tutorial

Before starting, make sure you have followed the instructions in *Installation and Setup*.

Also see the *Command Line Interface* page for complete documentation of all GAMBIT subcommands and options.

1.2.1 Example data set

The examples in this page make use of the following genome assembly files:

- 16AC1611138-CAP.fasta.gz
- 17AC0001410A.fasta.gz
- 17AC0006310.fasta.gz
- 17AC0006313-1.fasta.gz
- 19AC0011210.fasta.gz

From here on we'll pretend they've been downloaded to a directory called `genomes/`.

These are from genome set 3 in the initial GAMBIT publication, derived from clinical samples.

1.2.2 Telling GAMBIT where the database files are

The *query* command requires the GAMBIT reference database files. You can let GAMBIT know which directory contains these files in one of two ways:

Using a command line option

The first is to explicitly pass it via the command line using the `--db` option, like so:

```
gambit --db /path/to/database/ COMMAND ...
```

Note that this option must appear immediately after `gambit` and before the command name.

Using an environment variable

The second is to use the `GAMBIT_DB_PATH` environment variable. This can be done by running the following command at the beginning of your shell session:

```
export GAMBIT_DB_PATH="/path/to/database/"
```

Alternatively, you can add this line to your `.bashrc` to have it apply to all future sessions (make sure to restart your current session after doing so).

1.2.3 Genome input

Genome assemblies used as input must be in FASTA format, optionally compressed with gzip.

Most commands accept a list of genome files as positional arguments, e.g.:

```
gambit COMMAND [OPTIONS] genomes/16AC1611138-CAP.fasta.gz genomes/17AC0001410A.fasta.gz .
↪..
```

or making use of shell expansion:

```
gambit COMMAND [OPTIONS] genomes/*.fasta.gz
```

Alternatively you can use the `-l` option to provide a text file containing the genome file names/paths, one per line. The paths in this file are considered relative to the directory given by the `--ldir` option if given.

So for example, you can create the file `genomes.txt` containing the following:

```
16AC1611138-CAP.fasta.gz
17AC0001410A.fasta.gz
17AC0006310.fasta.gz
17AC0006313-1.fasta.gz
19AC0011210.fasta.gz
```

The command would then be:

```
gambit COMMAND [OPTIONS] -l genomes.txt --ldir genomes/
```

This method makes more sense when you have a lot of files to include. Note that the `gambit dist` command has different names for these options because there are two lists of genomes to specify. See the *Command Line Interface* page for more complete information.

1.2.4 Predicting taxonomy of unknown genomes

The *query* command compares a set of query genomes against the reference database and attempts to predict their taxonomy. The following runs a query with our five FASTA files and writes the results to `out.csv`:

```
gambit query -o out.csv genomes/*.fasta.gz
```

Table 1: Contents of out.csv

query	predicted.name	predicted.rank	predicted.node_id	predicted.threshold	closest.distance	closest.description	next.name	next.rank	next.node_id	next.threshold
16AC1611138-CAP	Escherichia coli	species	562	0.0	0.1586	[GCF_000351725.1] Escherichia coli KTE77 (E. coli)				
17AC0001410A	Enterococcus faecalis	species	1351	0.4697	0.1264	[GCF_000148005.1] Enterococcus faecalis DAPTO 512 (firmicutes)				
17AC0006310	Bacillus cereus	species	1396	0.0	0.1068	[GCF_001619385.1] Bacillus cereus (firmicutes)				
17AC0006313-1	Veillonella	genus	29465	0.9438	0.9126	[GCF_000024945.1] Veillonella parvula DSM 2008 (firmicutes)	Veillonella parvula	species	29466	0.6693
19AC0011210					0.9916	[GCF_000169595.1] Ureaplasma urealyticum serovar 9 str. ATCC 33175 (mycoplasmas)	Ureaplasma	genus	2129	0.8897

The `predicted` columns describe the predicted taxonomic classification of each query genome. `closest.description` is the database reference genome closest to the query, `closest.distance` is the distance between them. The `next` columns have the same format as `predicted` but describe the next most specific taxon for which the classification threshold was not met.

In this example GAMBIT was able to make a species-level prediction for the first three genomes but stopped at the genus level for the fourth and made no prediction for the fifth. This is because GAMBIT attempts to be conservative and error on the side of making a less specific prediction or no prediction rather than giving false positives. The `next` columns can give you a clue as to what a more specific classifiction might be, however.

See the *cli documentation* for a complete description of the output columns. Generally the CSV output format should be sufficient, but there is also a JSON-based format which contains more detailed information and may be useful in pipelines. Use `-f json` to use this format.

Todo: Explain why `predicted.threshold` is sometimes zero for certain taxa.

1.2.5 Pre-computing k-mer signatures

TODO

1.2.6 Calculating GAMBIT distances

TODO

1.2.7 Creating relatedness trees

TODO

1.3 Command Line Interface

1.3.1 Root command group

```
gambit [OPTIONS] COMMAND [ARGS]...
```

Some top-level options are set at the root command group, and should be specified *before* the name of the subcommand to run.

Options

-d, --db DIR

Path to the directory containing reference database files. Required by most subcommands. As an alternative you can specify the database location with the *GAMBIT_DB_PATH* environment variable.

Environment variables

GAMBIT_DB_PATH

Alternative to *-d* for specifying path to database.

1.3.2 Querying the database

query

```
gambit query [OPTIONS] (-s SIGFILE | -l LIST | GENOMES...)
```

Predict taxonomy of microbial samples from genome sequences.

GENOMES are one or more FASTA files containing assembled query genomes. Alternatively a file containing pre-calculated signatures may be used with the `--sigfile` option. The reference database must be specified from the root command group.

Options

-s, --sigfile FILE

Path to file containing query signatures.

-o, --output FILE

File to write output to. If omitted will write to stdout.

-f, --outfmt {csv|json|archive}

Results format (see next section).

Result Formats

CSV

A .csv file with one row per query. Contains the following columns:

- **query.* - Query genome.**
  - query.name - Name of query.
  - query.path - Path to query file, if any.
- **predicted.* - Predicted taxon.**
  - predicted.name
  - predicted.rank
  - predicted.ncbi_id - Numeric ID in NCBI taxonomy database.
  - predicted.threshold
- **closest.* - Reference genome closest to query.**
  - closest.distance - Distance to closest genome.
  - closest.decription - Text description.
- **next.* - Next most specific taxon for which the classification threshold was not met.**

- next.name
- next.rank
- next.ncbi_id
- next.threshold

JSON

A machine-readable format meant to be used in pipelines.

Todo: Document schema

Archive

A more verbose JSON-based format used for testing and development.

1.3.3 Generating and inspecting k-mer signatures

signatures info

```
gambit signatures info [OPTIONS] FILE
```

Print information about a GAMBIT signatures file. Defaults to a basic human-readable format.

Options

-j, --json
Print information in JSON format. Includes more information than standard output.

-p, --pretty
Prettify JSON output to make it more human-readable.

-i, --ids
Print IDs of all signatures in file.

signatures create

```
gambit signatures create [OPTIONS] GENOMES
```

Calculate GAMBIT signatures of GENOMES and write to file.

The -k and --prefix options may be omitted if a reference database is specified through the root command group, in which case the parameters of the database will be used.

Options

-o, --output FILE
Path to write file to (required).

-k INTEGER
Length of k-mers to find (does not include length of prefix).

-p, --prefix STRING
K-mer prefix to match, a non-empty string of DNA nucleotide codes.

-i, --ids FILE
File containing IDs to assign to signatures in file metadata. Should contain one ID per line.

-m, --meta-json FILE
JSON file containing metadata to attach to file.

Todo: Document metadata schema

1.4 Reference database releases

1.4.1 1.0b2

File	Size
gambit-genomes-1.0b2-rev2-211116.gdb	15.8 MB
gambit-signatures-1.0b1-210719.gs	1.3 GB

Note - the signatures file is identical to 1.0b1.

Todo: Describe differences to 1.0b1

1.4.2 1.0b1

File	Size
gambit-genomes-1.0b1-210719.gdb	15.8 MB
gambit-signatures-1.0b1-210719.gs	1.3 GB

1.5 Python API

1.5.1 K-mer signatures

gambit.seq

Generic code for working with sequence data.

Note that all code in this package operates on DNA sequences as sequences of bytes containing ascii-encoded nucleotide codes.

gambit.seq.**NUCLEOTIDES**

bytes corresponding to the four DNA nucleotides. Ascii-encoded upper case letters ACGT. Note that the order, while arbitrary, is important in this variable as it defines how unique indices are assigned to k-mer sequences.

gambit.seq.**revcomp**(*seq: bytes*) → bytes

Get the reverse complement of a nucleotide sequence.

Parameters
seq (*bytes*) – ASCII-encoded nucleotide sequence. Case does not matter.

Returns
Reverse complement sequence. All characters in the input which are not valid nucleotide codes will appear unchanged in the corresponding reverse position.

Return type
bytes

class gambit.seq.**SequenceFile**

Bases: PathLike

A reference to a DNA sequence file stored in the file system.

Contains all the information needed to read and parse the file. Implements the os.PathLike interface, so it can be substituted for a str or pathlib.Path in most function arguments that take a file path to open.

Parameters

- **path** (*Union[os.PathLike, str]*) – Value of *path* attribute. May be string or path-like object.
- **format** (*str*) – Value of *format* attribute.
- **compression** (*Optional[str]*) – Value of *compression* attribute.

path

Path to the file.

Type
pathlib.Path

format

String describing the file format as interpreted by Bio.SeqIO.parse(), e.g. 'fasta'.

Type
str

compression

String describing compression method of the file, e.g. 'gzip'. None means no compression. See *gambit.util.io.open_compressed()*.

Type
Optional[str]

__init__(*path*, *format*, *compression=None*)

Method generated by attrs for class SequenceFile.

Parameters

- **format** (*str*) –
- **compression** (*Optional[str]*) –

Return type
None

absolute()

Make a copy of the instance with an absolute path.

Return type
SequenceFile

classmethod from_paths(*paths*, *format*, *compression=None*)

Create many instances at once from a collection of paths and a single format and compression type.

Parameters

- **paths** (*Iterable[Union[str, PathLike]]*) – Collection of paths as strings or path-like objects.
- **format** (*str*) – Sequence file format of files.
- **compression** (*Optional[str]*) – Compression method of files.

Return type
List[SequenceFile]

open(*mode='r'*, ***kwargs*)

Open a stream to the file, with compression/decompression applied transparently.

Parameters

- **mode** (*str*) – Same as equivalent argument to the built-in :func:open`. Some modes may not be supported by all compression types.
- ****kwargs** – Additional text mode specific keyword arguments to pass to opener. Equivalent to the following arguments of the built-in *open()*: `encoding`, `errors`, and `newlines`. May not be supported by all compression types.

Returns
Stream to file in given mode.

Return type
IO

parse(***kwargs*)

Open the file and lazily parse its contents.

Returns iterator over sequence data in file. File is parsed lazily, and so must be kept open. The returned iterator is of type *gambit.util.io.ClosingIterator* so it will close the file stream automatically when it finishes. It may also be used as a context manager that closes the stream on exit. You may also close the stream explicitly using the iterator's `close` method.

Parameters
****kwargs** – Keyword arguments to *open()*.

Returns
Iterator yielding `Bio.SeqIO.SeqRecord` instances for each sequence in the file.

Return type
gambit.util.io.ClosingIterator

`gambit.seq.`**`seq_to_bytes`**(*seq*)

Convert generic DNA sequence to byte string representation.

This is for passing sequence data to Cython functions.

Parameters
seq (*Union[str, bytes, bytearray, Seq]*) –

Return type
Union[bytes, bytearray]

`gambit.seq.`**`validate_dna_seq_bytes`**(*seq*)

Check that a sequence contains only valid nucleotide codes (upper case).

Parameters
seq (*bytes*) – ASCII-encoded nucleotide sequence.

Raises
ValueError – If the sequence contains an invalid nucleotide.

`gambit.seq.`**`DNASeq`**

Union of DNA sequence types accepted for k-mer search / signature calculation.

alias of `Union`[`str`, `bytes`, `bytearray`, `Seq`]

`gambit.seq.`**`DNASeqBytes`**

Sequence types accepted directly by native (Cython) code.

alias of `Union`[`bytes`, `bytearray`]

gambit.kmers

Core functions for searching for and working with k-mers.

`gambit.kmers.`**`index_to_kmer`**(*index: int*, *kmer: int*) → bytes

Convert k-mer index to sequence.

class `gambit.kmers.`**`KmerMatch`**

Bases: `object`

Represents a

kmerspec

K-mer spec used for search.

Type
gambit.kmers.KmerSpec

seq

The sequence searched within.

Type
Union[str, bytes, bytearray, Bio.Seq.Seq]

pos

Index of first nucleotide of prefix in `seq`.

Type
int

reverse

If the match is on the reverse strand.

Type
bool

__init__(*kmerspec*, *seq*, *pos*, *reverse*)

Method generated by attrs for class KmerMatch.

Parameters

- **kmerspec** (KmerSpec) –
- **seq** (*Union[str, bytes, bytearray, Seq]*) –
- **pos** (*int*) –
- **reverse** (*bool*) –

Return type
None

full_indices()

Index range for prefix plus k-mer in sequence.

Return type
slice

kmer()

Get matched k-mer sequence.

Return type
bytes

kmer_index()

Get index of matched k-mer.

Raises
ValueError – If the k-mer contains invalid nucleotides.

Return type
int

kmer_indices()

Index range for k-mer in sequence (without prefix).

Return type
slice

class gambit.kmers.**KmerSpec**

Bases: *Jsonable*

Specifications for a k-mer search operation.

k

Number of nucleotides in k-mer *after* prefix.

Type
int

prefix

Constant prefix of k-mers to search for, upper-case nucleotide codes as ascii-encoded `bytes`.

Type
bytes

prefix_str

Prefix as string.

Type
str

prefix_len

Number of nucleotides in prefix.

Type
int

total_len

Sum of `prefix_len` and `k`.

Type
int

idx_len

Maximum value (plus one) of integer needed to index one of the found k-mers. Also the number of possible k-mers fitting the spec. Equal to `4 ** k`.

index_dtype

Smallest unsigned integer dtype that can store k-mer indices.

Type
numpy.dtype

__init__(*k*, *prefix*)

Parameters

- **k** (*int*) – Value of *k* attribute.
- **prefix** (*Union[str, bytes, bytearray, Seq]*) – Value of *prefix* attribute. Will be converted to `bytes`.

gambit.kmers.**find_kmers**(*kmerspec*, *seq*)

Locate k-mers with the given prefix in a DNA sequence.

Searches sequence both backwards and forwards (reverse complement). The sequence may contain invalid characters (not one of the four nucleotide codes) which will simply not be matched.

Parameters

- **kmerspec** (KmerSpec) – K-mer spec to use for search.
- **seq** (*Union[str, bytes, bytearray, Seq]*) – Sequence to search within. Lowercase characters are OK and will be matched as uppercase.

Returns
Iterator of *KmerMatch* objects.

Return type
Iterator[*KmerMatch*]

gambit.kmers.**index_dtype**(*k*)

Get the smallest unsigned integer dtype that can store k-mer indices for the given k.

Parameters
k (*int*) –

Return type
dtype

gambit.kmers.**kmer_to_index**(*kmer*)

Convert a k-mer to its integer index.

Raises
ValueError – If an invalid nucleotide code is encountered.

Parameters
kmer (*Union[str, bytes, bytearray, Seq]*) –

Return type
int

gambit.kmers.**kmer_to_index_rc**(*kmer*)

Get the integer index of a k-mer's reverse complement.

Raises
ValueError – If an invalid nucleotide code is encountered.

Parameters
kmer (*Union[str, bytes, bytearray, Seq]*) –

Return type
int

gambit.kmers.**nkmers**(*k*)

Get the number of possible distinct k-mers for a given value of k.

Parameters
k (*int*) –

Return type
int

gambit.kmers.**DEFAULT_KMERSPEC = KmerSpec(11, 'ATGAC')**

Default settings for k-mer search

gambit.sigs

Calculate and store collections of k-mer signatures.

gambit.sigs.base

class gambit.sigs.base.**AbstractSignatureArray**

Bases: Sequence[*KmerSignature*]

Abstract base class for types which behave as a (non-mutable) sequence of k-mer signatures (k-mer sets in sparse coordinate format).

The signature data itself may already be present in memory or may be loaded lazily from the file system when the object is indexed.

Elements should be Numpy arrays with integer data type. Should implement numpy-style advanced indexing, see *gambit.util.indexing.AdvancedIndexingMixin*. Slicing and advanced indexing should return another instance of AbstractSignatureArray.

kmerspec

K-mer spec used to calculate signatures.

Type
Optional[*gambit.kmers.KmerSpec*]

dtype

Numpy data type of signatures.

Type
numpy.dtype

__eq__(*other*)

Compare two AbstractSignatureArray instances for equality.

Two instances are considered equal if they are equivalent as sequences (see *sigarray_eq()*) and have the same *kmerspec*.

sizeof(*index*)

Get the size/length of the signature at the given index.

Should be the case that

sigarray.size_of(i) == len(sigarray[i])

Parameters
index (*int*) – Index of signature in array.

Return type
int

sizes()

Get the sizes of all signatures in the array.

Return type
Sequence[int]

class gambit.sigs.base.**AbstractSignatureArray**

Bases: Sequence[*KmerSignature*]

Abstract base class for types which behave as a (non-mutable) sequence of k-mer signatures (k-mer sets in sparse coordinate format).

The signature data itself may already be present in memory or may be loaded lazily from the file system when the object is indexed.

Elements should be Numpy arrays with integer data type. Should implement numpy-style advanced indexing, see *gambit.util.indexing.AdvancedIndexingMixin*. Slicing and advanced indexing should return another instance of AbstractSignatureArray.

kmerspec

K-mer spec used to calculate signatures.

Type
Optional[*gambit.kmers.KmerSpec*]

dtype

Numpy data type of signatures.

Type
numpy.dtype

sizeof(*index*)

Get the size/length of the signature at the given index.

Should be the case that

sigarray.size_of(i) == len(sigarray[i])

Parameters
index (*int*) – Index of signature in array.

Return type
int

sizes()

Get the sizes of all signatures in the array.

Return type
Sequence[int]

class gambit.sigs.base.**AnnotatedSignatures**

Bases: *ReferenceSignatures*

Wrapper around a signature array which adds id and meta attributes.

__init__(*signatures*, *ids=None*, *meta=None*)

Parameters

- **signatures** (AbstractSignatureArray) – Signature array to wrap.
- **ids** (*Optional[Sequence]*) – Unique IDs for signatures. Defaults to consecutive integers starting from zero.
- **meta** (*Optional[*SignaturesMeta*]*) – Additional metadata describing signatures.

class gambit.sigs.base.**ConcatenatedSignatureArray**

Bases: *AdvancedIndexingMixin*, *AbstractSignatureArray*

Base class for signature arrays which store signatures in a single data array.

values

K-mer signatures concatenated into single numpy-like array.

bounds

Numpy-like array storing indices bounding each individual k-mer signature in `values`. The `i`th signature is at `values[bounds[i]:bounds[i + 1]]`.

sizeof(*index*)

Get the size/length of the signature at the given index.

Should be the case that

sigarray.size_of(i) == len(sigarray[i])

Parameters
index – Index of signature in array.

sizes()

Get the sizes of all signatures in the array.

class gambit.sigs.base.**ReferenceSignatures**

Bases: *AbstractSignatureArray*

Base class for an array of reference genome signatures plus metadata.

This contains the extra data needed for the signatures to be used for running queries.

ids

Array of unique string or integer IDs for each signature. Length should be equal to length of `ReferenceSignatures` object.

Type
Sequence

meta

Other metadata describing signatures.

Type
gambit.sigs.base.SignaturesMeta

class gambit.sigs.base.**SignatureArray**

Bases: *ConcatenatedSignatureArray*

Stores a collection of k-mer signatures in a single contiguous Numpy array.

This format enables the calculation of many Jaccard scores in parallel, see *gambit.metric.jaccarddist_array()*.

Numpy-style indexing with an array of integers or bools is supported and will return another `SignatureArray`. If indexed with a contiguous slice the *values* of the returned array will be a view of the original instead of a copy.

values

K-mer signatures concatenated into single Numpy array.

Type
numpy.ndarray

bounds

Array storing indices bounding each individual k-mer signature in *values*. The `i`th signature is at `values[bounds[i]:bounds[i + 1]]`.

Type
numpy.ndarray

__init__(*signatures*, *kmerspec=None*, *dtype=None*)

Parameters

- **signatures** (*Sequence[KmerSignature]*) – Sequence of k-mer signatures.
- **kmerspec** (*Optional[KmerSpec]*) – K-mer spec used to calculate signatures. If None will take from `signatures` if it is an *AbstractSignatureArray* instance.
- **dtype** (*Optional[dtype]*) – Numpy dtype of *values* array. If None will use dtype of first element of `signatures`.

classmethod from_arrays(*values*, *bounds*, *kmerspec*)

Create directly from values and bounds arrays.

Parameters

- **values** (*ndarray*) –
- **bounds** (*ndarray*) –
- **kmerspec** (*Optional[KmerSpec]*) –

Return type
SignatureArray

classmethod uninitialized(*lengths*, *kmerspec*, *dtype=None*)

Create with an uninitialized values array.

Parameters

- **lengths** (*Sequence[int]*) – Sequence of lengths for each sub-array/signature.
- **kmerspec** (*Optional[KmerSpec]*) –
- **dtype** (*Optional[dtype]*) – Numpy dtype of shared coordinates array.

Return type
SignatureArray

class gambit.sigs.base.**SignatureList**

Bases: *AdvancedIndexingMixin*, *AbstractSignatureArray*, MutableSequence[*KmerSignature*]

Stores a collection of k-mer signatures in a standard Python list.

Compared to *SignatureArray* this isn't as efficient to calculate Jaccard scores with, but supports mutation and won't have to copy signatures to a new array on creation.

__init__(*signatures*, *kmerspec=None*, *dtype=None*)

Parameters

- **signatures** (*Iterable[KmerSignature]*) – Iterable of k-mer signatures.
- **kmerspec** (*Optional[KmerSpec]*) – K-mer spec used to calculate signatures. If None will take from `signatures` if it is an *AbstractSignatureArray* instance.

- **dtype** (*Optional[dtype]*) – Numpy dtype of signatures. If None will use dtype of first element of `signatures`.

insert(*i*, *sig*)

S.insert(index, value) – insert value before index

Parameters

- **i** (*int*) –
- **sig** (*KmerSignature*) –

class gambit.sigs.base.**SignaturesMeta**

Bases: `object`

Metadata describing a set of k-mer signatures.

All attributes are optional.

id

Any kind of string ID that can be used to uniquely identify the signature set.

Type
Optional[str]

version

Version string (ideally PEP 440-compliant).

Type
Optional[str]

name

Short human-readable name.

Type
Optional[str]

id_attr

Name of `Genome` attribute the IDs correspond to (see `gambit.db.models.GENOME_ID_ATTRS`). Optional, but signature set cannot be used as a reference for queries without it.

Type
Optional[str]

description

Human-readable description.

Type
Optional[str]

extra

Extra arbitrary metadata. Should be a `dict` or other mapping which can be converted to JSON.

Type
Mapping[str, Any]

__init__(*, *id=None*, *name=None*, *version=None*, *id_attr=None*, *description=None*, *extra=_Nothing.NOTHING*)

Method generated by attrs for class SignaturesMeta.

Parameters

- **id** (*Optional[str]*) –

- **name** (*Optional[str]*) –
- **version** (*Optional[str]*) –
- **id_attr** (*Optional[str]*) –
- **description** (*Optional[str]*) –
- **extra** (*Mapping[str, Any]*) –

Return type
None

gambit.sigs.base.**dump_signatures**(*path*, *signatures*, *format='hdf5'*, ***kw*)

Write k-mer signatures and associated metadata to a file.

Parameters

- **path** (*Union[str, PathLike]*) – File to write to.
- **signatures** (AbstractSignatureArray) – Array of signatures to store.
- **format** (*str*) – Format to use. Currently the only valid value is *'hdf5'*.
- ****kw** – Additional keyword arguments depending on format.

gambit.sigs.base.**load_signatures**(*path*, ***kw*)

Load signatures from file.

Currently the only format used to store signatures is the one in *gambit.sigs.hdf5*, but there may be more in the future. The format should be determined automatically.

Parameters

- **path** (*Union[str, PathLike]*) – File to open.
- ****kw** – Additional keyword arguments to `h5py.File()`.

Return type
AbstractSignatureArray

gambit.sigs.base.**sigarray_eq**(*a1*, *a2*)

Check two sequences of sparse k-mer signatures for equality.

Unlike *AbstractSignatureArray.__eq__()* this works on any sequence type containing signatures and does not use the *AbstractSignatureArray.kmerspec* attribute.

Parameters

- **a1** (*Sequence[KmerSignature]*) –
- **a2** (*Sequence[KmerSignature]*) –

Return type
bool

gambit.sigs.base.**KmerSignature**

Type for k-mer signatures (k-mer sets in sparse coordinate format)

alias of `ndarray`

gambit.sigs.calc

Calculate k-mer signatures from sequence data.

class gambit.sigs.calc.**ArrayAccumulator**

Bases: *KmerAccumulator*

K-mer accumulator implemented as a dense boolean array.

This is pretty efficient for smaller values of **k**, but time and space requirements increase exponentially with larger values.

__init__(*k*)

Parameters
k (*int*) –

add(*i*)

Add an element.

Parameters
i (*int*) –

clear()

This is slow (creates N new iterators!) but effective.

discard(*i*)

Remove an element. Do not raise an exception if absent.

Parameters
i (*int*) –

signature()

Get signature for accumulated k-mers.

Return type
KmerSignature

class gambit.sigs.calc.**KmerAccumulator**

Bases: `MutableSet[int]`

Base class for data structures which track k-mers as they are found in sequences.

Implements the `MutableSet` interface for k-mer indices. Indices are added via `add()` or *add_kmer()* methods, when finished a sparse k-mer signature can be obtained from *signature()*.

add_kmer(*kmer*)

Add a k-mer by its sequence rather than its index.

Argument may contain invalid (non-nucleotide) bytes, in which case it is ignored.

Parameters
kmer (*bytes*) –

abstract signature()

Get signature for accumulated k-mers.

Return type
KmerSignature

class gambit.sigs.calc.**SetAccumulator**

Bases: *KmerAccumulator*

Accumulator which uses the builtin Python `set` class.

This has more overhead than the array version for smaller values of `k` but behaves much better asymptotically.

__init__(*k*)

Parameters
k (*int*) –

add(*index*)

Add an element.

Parameters
index (*int*) –

clear()

This is slow (creates N new iterators!) but effective.

discard(*index*)

Remove an element. Do not raise an exception if absent.

Parameters
index (*int*) –

signature()

Get signature for accumulated k-mers.

Return type
KmerSignature

gambit.sigs.calc.**accumulate_kmers**(*accumulator*, *kmerspec*, *seq*)

Find k-mer matches in sequence and add their indices to an accumulator.

Parameters

- **accumulator** (*KmerAccumulator*) –
- **kmerspec** (*KmerSpec*) –
- **seq** (*Union[str, bytes, bytearray, Seq]*) –

gambit.sigs.calc.**calc_file_signature**(*kspec*, *seqfile*, *, *accumulator=None*)

Open a sequence file on disk and calculate its k-mer signature.

This works identically to `calc_signature_parse()` but takes a *SequenceFile* as input instead of a data stream.

Parameters

- **kspec** (*KmerSpec*) – Spec for k-mer search.
- **seqfile** (*SequenceFile*) – File to read.
- **accumulator** (*Optional[KmerAccumulator]*) – TODO

Returns
K-mer signature in sparse coordinate format (dtype will match `gambit.kmers.dense_to_sparse()`).

Return type
numpy.ndarray

See also:

calc_signature, *calc_file_signatures*

gambit.sigs.calc.**calc_file_signatures**(*kspec*, *files*, *progress=None*, *concurrency='processes'*, *max_workers=None*, *executor=None*)

Parse and calculate k-mer signatures for multiple sequence files.

Parameters

- **kspec** (KmerSpec) – Spec for k-mer search.
- **seqfile** – Files to read.
- **progress** – Display a progress meter. See *gambit.util.progress.get_progress()* for allowed values.
- **concurrency** (*Optional[str]*) – Process files concurrently. `"processes"` for process-based (default), `"threads"` for threads-based, `None` for no concurrency.
- **max_workers** (*Optional[int]*) – Number of worker threads/processes to use if `concurrency` is not None.
- **executor** (*Optional[Executor]*) – Instance of class:*concurrent.futures.Executor* to use for concurrency. Overrides the `concurrency` and `max_workers` arguments.
- **files** (*Sequence[SequenceFile]*) –

Return type
SignatureList

See also:

calc_file_signature

gambit.sigs.calc.**calc_signature**(*kmerspec*, *seqs*, ***, *accumulator=None*)

Calculate the k-mer signature of a DNA sequence or set of sequences.

Searches sequences both backwards and forwards (reverse complement). Sequences may contain invalid characters (not one of the four nucleotide codes) which will simply not be matched.

Parameters

- **kmerspec** (KmerSpec) – K-mer spec to use for search.
- **seqs** (*Union[str, bytes, bytearray, Seq, Iterable[Union[str, bytes, bytearray, Seq]]]*) – Sequence or sequences to search within. Lowercase characters are OK.
- **accumulator** (*Optional[KmerAccumulator]*) – TODO

Returns
K-mer signature in sparse coordinate format. Data type will be `kspec.index_dtype`.

Return type
numpy.ndarray

See also:

calc_file_signature

gambit.sigs.calc.**default_accumulator**(*k*)

Get a default k-mer accumulator instance for the given value of `k`.

Returns a *ArrayAccumulator* for `k <= 11` and a *SetAccumulator* for `k > 11`.

Parameters
k (*int*) –

Return type
KmerAccumulator

gambit.sigs.convert

Convert signatures between representations or from one `KmerSpec` to another.

`gambit.sigs.convert.`**`can_convert`**(*from_kspec*, *to_kspec*)

Check if signatures from one KmerSpec can be converted to another.

Conversion is possible if `to_kspec.prefix` is equal to or starts with `from_kspec.prefix` and `to_kspec.total_len <= from_kspec.total_len`.

Parameters

- **from_kspec** (KmerSpec) –
- **to_kspec** (KmerSpec) –

Return type
bool

`gambit.sigs.convert.`**`check_can_convert`**(*from_kspec*, *to_kspec*)

Check that signatures can be converted from one KmerSpec to another or raise an error with an informative message.

Raises
ValueError – If conversion is not possible.

Parameters

- **from_kspec** (KmerSpec) –
- **to_kspec** (KmerSpec) –

`gambit.sigs.convert.`**`convert_dense`**(*from_kspec*, *to_kspec*, *vec*)

Convert a k-mer signature in dense format from one `KmerSpec` to another.

In the ideal case, if `vec` is the result of `calc_signature(from_kspec, seq, sparse=False)` the output of this function should be identical to `calc_signature(to_kspec, seq, sparse=False)`. In reality this may not hold if any potential matches of `from_kspec` in `seq` are discarded due to an invalid nucleotide which is not included in the corresponding `to_kspec` match.

Parameters

- **from_kspec** (KmerSpec) –
- **to_kspec** (KmerSpec) –
- **vec** (*ndarray*) –

Return type
ndarray

`gambit.sigs.convert.`**`convert_sparse`**(*from_kspec*, *to_kspec*, *sig*)

Convert a k-mer signature in sparse format from one `KmerSpec` to another.

In the ideal case, if `sig` is the result of `calc_signature(from_kspec, seq)` the output of this function should be identical to `calc_signature(to_kspec, seq)`. In reality this may not hold if any potential matches

of `from_kspec` in `seq` are discarded due to an invalid nucleotide which is not included in the corresponding `to_kspec` match.

Parameters

- **from_kspec** (KmerSpec) –
- **to_kspec** (KmerSpec) –
- **sig** (*KmerSignature*) –

Return type
KmerSignature

gambit.sigs.convert.**dense_to_sparse**(*vec*)

Convert k-mer set from dense bit vector to sparse coordinate representation.

Parameters
vec (*Sequence[bool]*) – Boolean vector indicating which k-mers are present.

Returns
Sorted array of coordinates of k-mers present in vector. Data type will be `numpy.intp`.

Return type
numpy.ndarray

See also:

sparse_to_dense

gambit.sigs.convert.**sparse_to_dense**(*k_or_kspec*, *coords*)

Convert k-mer set from sparse coordinate representation back to dense bit vector.

Parameters

- **k_or_kspec** (*Union[int, KmerSpec]*) – Value of k or a *KmerSpec* instance.
- **coords** (*KmerSignature*) – Sparse coordinate array.

Returns
Dense k-mer bit vector.

Return type
numpy.ndarray

See also:

dense_to_sparse

gambit.sigs.hdf5

Store k-mer signature sets in HDF5 format.

class gambit.sigs.hdf5.**HDF5Signatures**

Bases: *ConcatenatedSignatureArray*, *ReferenceSignatures*

Stores a set of k-mer signatures and associated metadata in an HDF5 group.

Inherits from *gambit.sigs.base.AbstractSignatureArray*, so behaves as a sequence of k-mer signatures supporting Numpy-style advanced indexing.

Behaves as a context manager which yields itself on enter and closes the underlying HDF5 file object on exit. The *__bool__()* method can be used to check whether the file is currently open and valid.

group

HDF5 group object data is read from.

Type
h5py._hl.group.Group

Parameters
group (*h5py._hl.group.Group*) – Open, readable `h5py.Group` or `h5py.File` object.

__bool__()

Check whether the underlying HDF5 file object is open.

__init__(*group*)

Parameters
group (*Group*) –

close()

Close the underlying HDF5 file.

classmethod create(*group*, *signatures*, *, *compression=None*, *compression_opts=None*)

Store k-mer signatures and associated metadata in an HDF5 group.

Parameters

- **group** (*Group*) – HDF5 group to store data in.
- **signatures** (AbstractSignatureArray) – Array of signatures to store. If an instance of *gambit.sigs.base.ReferenceSignatures* its metadata will be stored as well, otherwise default/empty values will be used.
- **compression** (*Optional[str]*) – Compression type for values array. One of `['gzip', 'lzf', 'szip']`. See the section on compression filters in `h5py`'s documentation.
- **compression_opts** – Sets compression level (0-9) for gzip compression, no effect for other types.

Return type
HDF5Signatures

class gambit.sigs.hdf5.**HDF5Signatures**

Bases: *ConcatenatedSignatureArray*, *ReferenceSignatures*

Stores a set of k-mer signatures and associated metadata in an HDF5 group.

Inherits from *gambit.sigs.base.AbstractSignatureArray*, so behaves as a sequence of k-mer signatures supporting Numpy-style advanced indexing.

Behaves as a context manager which yields itself on enter and closes the underlying HDF5 file object on exit. The *__bool__()* method can be used to check whether the file is currently open and valid.

group

HDF5 group object data is read from.

Type
h5py._hl.group.Group

Parameters
group (*h5py._hl.group.Group*) – Open, readable `h5py.Group` or `h5py.File` object.

__init__(*group*)

Parameters
group (*Group*) –

close()

Close the underlying HDF5 file.

classmethod create(*group*, *signatures*, *, *compression=None*, *compression_opts=None*)

Store k-mer signatures and associated metadata in an HDF5 group.

Parameters

- **group** (*Group*) – HDF5 group to store data in.
- **signatures** (AbstractSignatureArray) – Array of signatures to store. If an instance of *gambit.sigs.base.ReferenceSignatures* its metadata will be stored as well, otherwise default/empty values will be used.
- **compression** (*Optional[str]*) – Compression type for values array. One of `['gzip', 'lzf', 'szip']`. See the section on compression filters in h5py's documentation.
- **compression_opts** – Sets compression level (0-9) for gzip compression, no effect for other types.

Return type
HDF5Signatures

gambit.sigs.hdf5.**dump_signatures_hdf5**(*path*, *signatures*, ***kw*)

Write k-mer signatures and associated metadata to an HDF5 file.

Parameters

- **path** (*Union[str, PathLike]*) – File to write to.
- **signatures** (AbstractSignatureArray) – Array of signatures to store.
- ****kw** – Additional keyword arguments to *HDF5Signatures.create()*.

gambit.sigs.hdf5.**empty_to_none**(*value*)

Convert `h5py.Empty` instances to None, passing other types through.

gambit.sigs.hdf5.**load_signatures_hdf5**(*path*, ***kw*)

Open HDF5 signature file.

Parameters

- **path** (*Union[str, PathLike]*) – File to open.
- ****kw** – Additional keyword arguments to `h5py.File()`.

Return type
HDF5Signatures

gambit.sigs.hdf5.**none_to_empty**(*value*, *dtype*)

Convert None values to `h5py.Empty`, passing other types through.

Parameters
dtype (*dtype*) –

gambit.sigs.hdf5.**read_metadata**(*group*)

Read signature set metadata from HDF5 group attributes.

Parameters
group (*Group*) –

Return type
SignaturesMeta

gambit.sigs.hdf5.**write_metadata**(*group*, *meta*)

Write signature set metadata to HDF5 group attributes.

Parameters

- **group** (*Group*) –
- **meta** (SignaturesMeta) –

gambit.sigs.hdf5.**CURRENT_FMT_VERSION = 1**

Current version of the data format. Integer which should be incremented each time the format changes.

gambit.sigs.hdf5.**FMT_VERSION_ATTR = 'gambit_signatures_version'**

Name of HDF5 group attribute which both stores the format version and also identifies the group as containing signature data.

1.5.2 Distance Metric

gambit.metric

Calculate the Jaccard index/distance between sets.

gambit.metric.**jaccard**(*coords1*, *coords2*)

Compute the Jaccard index between two k-mer sets in sparse coordinate format.

Arguments are Numpy arrays containing k-mer indices in sorted order. Data types must be 16, 32, or 64-bit signed or unsigned integers, but do not need to match.

This is by far the most efficient way to calculate the metric (this is a native function) and should be used wherever possible.

Parameters

- **coords1** (*numpy.ndarray*) – K-mer set in sparse coordinate format.
- **coords2** (*numpy.ndarray*) – K-mer set in sparse coordinate format.

Returns
Jaccard index between the two sets, a real number between 0 and 1.

Return type
numpy.float32

See also:

jaccarddist

gambit.metric.**jaccarddist**(*coords1*, *coords2*)

Compute the Jaccard distance between two k-mer sets in sparse coordinate format.

The Jaccard distance is equal to one minus the Jaccard index.

Arguments are Numpy arrays containing k-mer indices in sorted order. Data types must be 16, 32, or 64-bit signed or unsigned integers, but do not need to match.

This is by far the most efficient way to calculate the metric (this is a native function) and should be used wherever possible.

Parameters

- **coords1** (*numpy.ndarray*) – K-mer set in sparse coordinate format.
- **coords2** (*numpy.ndarray*) – K-mer set in sparse coordinate format.

Returns
Jaccard distance between the two sets, a real number between 0 and 1.

Return type
numpy.float32

See also:

jaccard

gambit.metric.**jaccard_bits**(*bits1*, *bits2*)

Calculate the Jaccard index between two sets represented as bit arrays ("dense" format for k-mer sets).

See also:

jaccard

Parameters

- **bits1** (*ndarray*) –
- **bits2** (*ndarray*) –

Return type
float

gambit.metric.**jaccard_generic**(*set1*, *set2*)

Get the Jaccard index of of two arbitrary sets.

This is primarily used as a slow, pure-Python alternative to *jaccard()* to be used for testing, but can also be used as a generic way to calculate the Jaccard index which works with any collection or element type.

See also:

jaccard, *jaccard_bits*

Parameters

- **set1** (*Iterable*) –
- **set2** (*Iterable*) –

Return type
float

gambit.metric.**jaccarddist_array**(*query*, *refs*, *out=None*)

Calculate Jaccard distances between a query k-mer signature and a list of reference signatures.

For enhanced performance `refs` should be an instance of *gambit.sigs.base.SignatureArray*. This allows use of optimized Cython code that runs in parallel over all signatures in `refs`. In that case, because of Cython limitations `refs.bounds.dtype` must be `np.intp`, which is usually a 64-bit signed integer. If it is not it will be converted automatically.

Parameters

- **query** (*KmerSignature*) – Query k-mer signature in sparse coordinate format (sorted array of k-mer indices).
- **refs** (*Sequence[KmerSignature]*) – List of reference signatures.
- **out** (*Optional[ndarray]*) – Optional pre-allocated array to write results to. Should be the same length as `refs` with dtype `np.float32`.

Returns
Jaccard distance for `query` against each element of `refs`.

Return type
numpy.ndarray

See also:

jaccarddist, *jaccarddist_matrix*

gambit.metric.**jaccarddist_matrix**(*queries*, *refs*, *ref_indices=None*, *out=None*, *chunksize=None*, *progress=None*)

Calculate a Jaccard distance matrix between a list of query signatures and a list of reference signatures.

This function improves querying performance when the reference signatures are stored in a file (e.g. using *gambit.sigs.hdf5.HDF5Signatures*) by loading them in chunks (via the `chunksize` parameter) instead of all in one go.

Performance is greatly improved if `refs` is a type that yields instances of `SignatureArray` when indexed with a slice object (`SignatureArray` or `HDF5Signatures`), see `jaccarddist_array()`. There is no such dependence on the type of `queries`, which can be a simple list.

Parameters

- **queries** (*Sequence[KmerSignature]*) – Query signatures in sparse coordinate format.
- **refs** (*Sequence[KmerSignature]*) – Reference signatures in sparse coordinate format.
- **ref_indices** (*Optional[Sequence[int]]*) – Optional, indices of `refs` to use.
- **out** (*Optional[ndarray]*) – (Optional) pre-allocated array to write output to.
- **chunksize** (*Optional[int]*) – Divide `refs` into chunks of this size.
- **progress** – Display a progress meter of the number of elements of the output array calculated so far. See *gambit.util.progress.get_progress()* for a description of allowed values.

Returns
Matrix of distances between query signatures in rows and reference signatures in columns.

Return type
np.ndarray

See also:

jaccarddist, *jaccarddist_array*, *jaccarddist_pairwise*

gambit.metric.**jaccarddist_pairwise**(*sigs*, *indices=None*, *flat=False*, *out=None*, *progress=None*)

Calculate all pairwise Jaccard distances for a list of signatures.

This should be roughly twice as fast as calling `jaccarddist_flat()` with the same array for the first and second arguments, because each pairwise distance is computed once instead of twice.

For optimal performance the type of `sigs` is subject to the same requirements as *jaccarddist_array()* and *jaccarddist_matrix()*.

Parameters

- **sigs** (*Sequence[KmerSignature]*) – List of signatures in sparse coordinate format.
- **indices** (*Optional[Sequence[int]]*) – Optional, indices of `sigs` to use.
- **flat** (*bool*) – If True the output is a non-redundant flat (1D) array with exactly one element per pair of signatures. This format can be converted to/from the equivalent full distance matrix with `scipy.spatial.distance.squareform()`.
- **out** (*Optional[ndarray]*) – (Optional) pre-allocated array to write output to.
- **progress** – Display a progress meter of the number of elements of the output array calculated so far. See *gambit.util.progress.get_progress()* for a description of allowed values.

Returns
Pairwise distances in matrix (if `flat=False`) or condensed (`flat=True`) format.

Return type
np.ndarray

See also:

jaccarddist_matrix

gambit.metric.**num_pairs**(*n*)

Get the number of distinct (unordered) pairs of `n` objects.

Parameters
n (*int*) –

Return type
int

1.5.3 Database

gambit.db

gambit.db.refdb

class gambit.db.refdb.**ReferenceDatabase**

Bases: `object`

Object containing reference genomes, their k-mer signatures, and associated data.

This is all that is needed at runtime to run queries.

genomeset

Genome set containing reference genomes.

Type
gambit.db.models.ReferenceGenomeSet

genomes

List of reference genomes.

Type
Sequence[*gambit.db.models.AnnotatedGenome*]

signatures

K-mer signatures for each genome. A subtype of `ReferenceSignatures`, so contains metadata on signatures as well as the signatures themselves. Type may represent signatures stored on disk (e.g. `HDF5Signatures`) instead of in memory. OK to contain additional signatures not corresponding to any genome in `genomes`.

Type
gambit.sigs.base.ReferenceSignatures

sig_indices

Index of signature in `signatures` corresponding to each genome in `genomes`. In sorted order to improve performance when iterating over them (improve locality if in memory and avoid seeking if in file).

Type
Sequence[int]

session

The SQLAlchemy session `genomeset` and the elements of `genomes` belong to. It is important to keep a reference to this, just having references to the ORM objects themselves is not enough to keep the session from being garbage collected.

Parameters

- **genomeset** (gambit.db.models.ReferenceGenomeSet) –
- **signatures** (gambit.sigs.base.ReferenceSignatures) –

__init__(*genomeset*, *signatures*)

Parameters

- **genomeset** (ReferenceGenomeSet) –
- **signatures** (ReferenceSignatures) –

classmethod load(*genomes_file*, *signatures_file*)

Load complete database given paths to SQLite genomes database file and HDF5 signatures file.

Parameters

- **genomes_file** (*Union[str, PathLike]*) –
- **signatures_file** (*Union[str, PathLike]*) –

Return type
ReferenceDatabase

classmethod load_from_dir(*path*)

Load complete database given directory containing SQLite genomes database file and HDF5 signatures file.

See `locate_db_files()` for how these files are located within the directory.

Raises
RuntimeError – If files cannot be located in directory.

Parameters
path (*Union[str, PathLike]*) –

Return type
ReferenceDatabase

classmethod locate_files(*path*)

Locate an SQLite genome database file and HDF5 signatures file in a directory.

Files are located by extension, `.gdb` or `.db` for SQLite file and `.gs` or `.h5` for signatures file. Does not look in subdirectories.

Parameters
path (*Union[str, PathLike]*) – Path to directory to look within.

Return type
Paths to genomes database file and signatures file.

Raises
FileNotFoundError – If files could not be located or if multiple files with the same extension exist in the directory.

gambit.db.refdb.**genomes_by_id**(*genomeset*, *id_attr*, *ids*, *strict=True*)

Match a `ReferenceGenomeSet`'s genomes to a set of ID values.

This is primarily used to match genomes to signatures based on the ID values stored in a signature file. It is expected that the signature file may contain signatures for more genomes than are present in the genome set, see also *genomes_by_id_subset()* for that condition.

Parameters

- **genomeset** (ReferenceGenomeSet) –
- **id_attr** (*Union[str, InstrumentedAttribute]*) – ID attribute of *gambit.db.models.Genome* to use for lookup. Can be used as the attribute itself (e.g. `Genome.refseq_acc`) or just the name (`'refsec_acc'`). See `GENOME_IDS` for the set of allowed values.
- **ids** (*Sequence*) – Sequence of ID values (strings or integers, matching type of attribute).
- **strict** (*bool*) – Raise an exception if a matching genome cannot be found for any ID value.

Returns
List of genomes of same length as `ids`. If `strict=False` and a genome cannot be found for a given ID the list will contain `None` at the corresponding position.

Return type
List[Optional[*AnnotatedGenome*]]

Raises
KeyError – If `strict=True` and any ID value cannot be found.

gambit.db.refdb.**genomes_by_id_subset**(*genomeset*, *id_attr*, *ids*)

Match a `ReferenceGenomeSet`'s genomes to a set of ID values, allowing missing genomes.

This calls *genomes_by_id()* with `strict=False` and filters any `None` values from the output. The filtered list is returned along with the indices of all values in `ids` which were not filtered out. The indices can be used to load only those signatures which have a matched genome from a signature file.

Note that it is not checked that every genome in `genomeset` is matched by an ID. Check the size of the returned lists for this.

Parameters

- **genomeset** (ReferenceGenomeSet) –
- **id_attr** (*Union[str, InstrumentedAttribute]*) – ID attribute of *gambit.db.models.Genome* to use for lookup. Can be used as the attribute itself (e.g. `Genome.`

refseq_acc) or just the name ('refsec_acc'). See GENOME_IDS for the set of allowed values.

- **ids** (*Sequence*) – Sequence of ID values (strings or integers, matching type of attribute).

Return type
Tuple[List[*AnnotatedGenome*], List[int]]

gambit.db.refdb.**load_genomeset**(*db_file*)

Get the only *gambit.db.models.ReferenceGenomeSet* from a genomes database file.

Parameters
db_file (*Union[str, PathLike]*) –

Return type
Tuple[*Session*, ReferenceGenomeSet]

gambit.db.models

SQLAlchemy models for storing reference genomes and taxonomy information.

class gambit.db.models.**AnnotatedGenome**

Bases: Base

A genome with additional annotations as part of a genome set.

This object serves to attach a genome to a *ReferenceGenomeSet*, and to assign a taxonomy classification to that genome. Hybrid attributes mirroring the attributes of the connected genome effectively make this behave as an extended Genome object.

genome_id

Integer column, part of composite primary key. ID of *Genome* the annotations are for.

Type
int

genome_set_id

Integer column, part of composite primary key. ID of the *ReferenceGenomeSet* the annotations are under.

Type
int

organism

String column. Single string describing the organism. May be "Genus species [strain]" but could contain more specific information. Intended to be human-readable and shouldn't have any semantic meaning for the application (in contrast to the taxa relationship).

Type
str

taxon_id

Integer column. ID of the *Taxon* this genome is classified as.

Type
int

genome

Many-to-one relationship to *Genome*.

Type
.Genome

genome_set

Many-to-one relationship to *ReferenceGenomeSet*.

Type
.ReferenceGenomeSet

taxon

Many-to-one relationship to *Taxon*. The primary taxon this genome is classified as under the associated `ReferenceGenomeSet`. Should be the most specific and "regular" (ideally defined on NCBI) taxon this genome belongs to.

Type
.Taxon

key

Hybrid property connected to attribute on *genome*.

Type
str

description

Hybrid property connected to attribute on *genome*.

Type
Optional[str]

ncbi_db

Hybrid property connected to attribute on *genome*.

Type
Optional[str]

ncbi_id

Hybrid property connected to attribute on *genome*.

Type
Optional[int]

genbank_acc

Hybrid property connected to attribute on *genome*.

Type
Optional[str]

refseq_acc

Hybrid property connected to attribute on *genome*.

Type
Optional[str]

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

class gambit.db.models.**Genome**

Bases: Base

Base model for a reference genome that can be compared to query.

Corresponds to a single assembly (one or more contigs, but at least partially assembled) from what should be a single sequencing run. The same organism or strain may have several genome entries for it. Typically this will correspond directly to a record in Genbank (assembly database).

The data on this model should primarily pertain to the sample and sequencing run itself. It would be updated if for example a better assembly was produced from the original raw data, however more advanced interpretation such as taxonomy assignments belong on an attached *AnnotatedGenome* object.

id

Integer column (primary key).

Type
int

key

String column (unique). Unique "external id" used to reference the genome from outside the SQL database, e.g. from a file containing K-mer signatures.

Type
str

description

String column (optional). Short one-line description. Recommended to be unique but this is not enforced.

Type
Optional[str]

ncbi_db

String column (optional). If the genome corresponds to a record downloaded from an NCBI database this column should be the database name (e.g. `'assembly'`) and `ncbi_id` should be the entry's UID. Unique along with `ncbi_id`.

Type
Optional[str]

ncbi_id

Integer column (optional). See previous.

Type
Optional[int]

genbank_acc

String column (optional, unique). GenBank accession number for this genome, if any.

Type
Optional[str]

refseq_acc

String column (optional, unique). RefSeq accession number for this genome, if any.

Type
Optional[str]

extra

JSON column (optional). Additional arbitrary metadata.

Type
Optional[dict]

annotations

One-to-many relationship to *AnnotatedGenome*.

Type
Collection[.AnnotatedGenome]

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

ID_ATTRS = ('key', 'genbank_acc', 'refseq_acc', 'ncbi_id')

Attributes which serve as unique IDs.

class gambit.db.models.**ReferenceGenomeSet**

Bases: `Base`

A collection of reference genomes along with additional annotations and data. A full GAMBIT database which can be used for queries consists of a genome set plus a set of k-mer signatures for those genomes (stored separately).

Membership of `Genome`s in the set is determined by the presence of an associated :class:.AnnotatedGenomes` object, which also holds additional annotation data for the genome. The genome set also includes a set of associated *Taxon* entries, which form a taxonomy tree under which all its genomes are categorized.

This schema technically allows for multiple genome sets within the same database (which can share *Genome*s but with different annotations), but the GAMBIT application generally expects that genome sets are stored in their own SQLite files.

id

Integer primary key.

Type
int

key

String column. An "external id" used to uniquely identify this genome set. Unique along with `version`.

Type
str

version

Optional version string, an updated version of a previous genome set should have the same key with a later version number. Should be in the format defined by PEP 440.

Type
str

name

String column. Unique name.

Type
str

description

Text column. Optional description.

Type
Optional[str]

extra

JSON column. Additional arbitrary data.

Type
Optional[dict]

genomes

Many-to-many relationship with *AnnotatedGenome*, annotated versions of genomes in this set.

Type
Collection[.AnnotatedGenome]

base_genomes

Unannotated *Genome*s in this set. Association proxy to the genome relationship of members of genome.

Type
Collection[.Genome]

taxa

One-to-many relationship to *Taxon*. The taxa that form the classification system for this genome set.

Type
Collection[.Taxon]

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

root_taxa()

Query for root taxa belonging to the set.

Return type
sqlalchemy.orm.query.Query

class gambit.db.models.**Taxon**

Bases: Base

A taxon used for classifying genomes.

Taxa are specific to a *ReferenceGenomeSet* and form a tree/forest structure through the *parent* and *children* relationships.

id

Integer column (primary key).

Type
int

key

String column (unique). An "external id" used to uniquely identify this taxon.

Type
str

name

String column. Human-readable name for the taxon, typically the standard scientific name.

Type
str

rank

String column (optional). Taxonomic rank, if any. Species, genus, family, etc.

Type
Optional[str]

description

String column (optional). Optional description of taxon.

Type
Optional[str]

distance_threshold

Float column (optional). Query genomes within this distance of one of the taxon's reference genomes will be classified as that taxon. If NULL the taxon is just used establish the tree structure and is not used directly in classification.

Type
Optional[float]

report

Boolean column. Whether to report this taxon directly as a match when producing a human-readable query result. Some custom taxa might need to be "hidden" from the user, in which case the value should be false. The application should then ascend the taxon's lineage and choose the first ancestor where this field is true. Defaults to true.

Type
Bool

extra

JSON column (optional). Additional arbitrary data.

Type
Optional[dict]

genome_set_id

Integer column. ID of *ReferenceGenomeSet* the taxon belongs to.

Type
int

parent_id

Integer column. ID of Taxon that is the direct parent of this one.

Type
Optional[int]

ncbi_id

Integer column (optional). ID of the entry in the NCBI taxonomy database this taxon corresponds to, if any.

Type
Optional[int]

parent

Many-to-one relationship with *Taxon*, the parent of this taxon (if any).

Type
Optional[.Taxon]

children

One-to-many relationship with *Taxon*, the children of this taxon.

Type
Collection[.Taxon]

genome_set

Many-to-one relationship to *ReferenceGenomeSet*.

Type
.ReferenceGenomeSet

genomes

One-to-many relationship with *AnnotatedGenome*, genomes which are assigned to this taxon.

Type
Collection[.AnnotatedGenome]

__init__(***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

ancestor_of_rank(*rank*)

Get the taxon's ancestor with the given rank, if it exists.

Parameters
rank (*str*) –

Return type
Optional[Taxon]

ancestors(*incself=False*)

Iterate through the taxon's ancestors from bottom to top.

Parameters
incself (*bool*) – If True start with self, otherwise start with parent.

Return type
Iterable[Taxon]

classmethod common_ancestors(*taxa*)

Get list of common ancestors of a set of taxa.

Returns
Common ancestors from top to bottom (same order as *lineage()*. Will be empty if

Return type
List[.Taxon]

Parameters
taxa (*Iterable[Taxon]*) –

depth()

The number of ancestors the taxon has.

Return type
int

descendants(*postorder=False*)

Iterate through taxa all of the taxon's descendants.

This is the same as *traverse()* except the taxon itself is not included.

Parameters
postorder (*bool*) – Iterate in postorder (parents after children) instead of the default pre-order (parents before children).

Return type
Iterable[Taxon]

has_genome(*genome*)

Check whether the given genome is assigned to this taxon or any of its descendants.

Parameters
genome (AnnotatedGenome) –

Return type
bool

isleaf()

Check if the taxon is a leaf (has no children).

Return type
bool

isroot()

Check if the taxon is a root (has no parent).

Return type
bool

classmethod lca(*taxa*)

Find the Least Common Ancestor of a set of taxa.

Returns None if *taxa* is empty or its members do not all lie in the same tree.

Parameters
taxa (*Iterable[Taxon]*) –

Return type
List[Taxon]

leaves()

Iterate through all leaves in the taxon's subtree.

For leaf taxa this will just yield the taxon itself.

Return type
Iterable[Taxon]

lineage(*ranks=None*)

Get a last of this taxon's ancestors.

With an argument, gets ancestors with the given ranks. Without, gets a sorted list of the taxon's ancestors from top to bottom (including itself)

Parameters
ranks (*Optional[Iterable[str]]*) –

Return type
List[*Optional*[Taxon]]

print_tree(*f=None*, ***, *indent=' '*, *sort_key=None*)

Print the taxon's subtree for debugging.

Parameters

- **f** (*Optional[Callable[[Taxon], str]]*) – A function which takes a taxon and returns the string representation to print for it. Defaults to *short_repr()*.
- **indent** (*str*) – String used to indent each level of descendants.
- **sort_key** (*Optional[Callable[[Taxon], Any]]*) – A function which takes a taxon and returns a sort key, to determine what order a taxon's children are printed in. Defaults to the taxon's name.

root()

Get the root taxon of this taxon's tree.

The set of taxa in a *ReferenceGenomeSet* will generally form a forest instead of a single tree, so there can be multiple root taxa.

Returns self if the taxon has no parent.

Return type
Taxon

short_repr()

Get a short string representation of the Taxon for logging and warning/error messages.

subtree_genomes()

Iterate through all genomes assigned to this taxon or its descendants.

Return type
Iterable[AnnotatedGenome]

traverse(*postorder=False*)

Iterate through all nodes in this taxon's subtree.

Parameters
postorder (*bool*) – Iterate in postorder (parents after children) instead of the default preorder (parents before children).

Return type
Iterable[Taxon]

gambit.db.models.**only_genomeset**(*session*)

Get the only ReferenceGenomeSet in a database.

The format which is used to distribute GAMBIT databases and is expected by the CLI is an sqlite file containing a single genome set.

Parameters
session (*Session*) – ORM session connected to database.

Raises
RuntimeError – If the database does not contain a single genome set.

Return type
ReferenceGenomeSet

gambit.db.models.**reportable_taxon**(*taxon*)

Find the first reportable taxon in a linage.

Parameters
taxon (*Optional[Taxon]*) – Taxon to start looking from. None values are passed through.

Returns
Most specific taxon in ancestry with `report=True`, or `None` if none found.

Return type
Optional[*Taxon*]

gambit.db.sqla

Custom types and other utilities for SQLAlchemy.

class gambit.db.sqla.**JsonString**

Bases: `TypeDecorator`

SQLA column type for JSON data which is stored in the database as a standard string column.

Data is automatically serialized/unserialized when saved/loaded. Important: mutation tracking is not enabled for this type. If the value is a list or dict and you modify it in place these changes will not be detected. Instead, re-assign the attribute.

class gambit.db.sqla.**ReadOnlySession**

Bases: `Session`

Session class that doesn't allow flushing/committing.

gambit.db.migrate

Perform genome database migrations with Alembic.

This package contains all Alembic configuration and data files. Revision files are located in `./alembic/versions`.

Note on alembic configuration - seems like normal usage of Alembic involves getting the database URL from `alembic.ini`. Since this application has no fixed location for the database we can't use this method. Instead we are following the Sharing a Connection with a Series of Migration Commands and Environments recipe in Alembic's documentation, where the connectable object is generated programmatically somehow and then attached to the Alembic configuration object's `attributes` dict. The `run_migrations_offline` and `run_migrations_online` functions in `alembic/env.py` are modified from the version generated by `alembic init` to get their connectable object from this dict instead of creating it based on the contents of `alembic.ini`. Note that this means we can't do (online) migration stuff from the standard alembic CLI command, which gets its connection information only from `alembic.ini`.

The way to use this setup is instead to create an `alembic.config.Config` instance with *get_alembic_config()* and use the functions in `alembic.command`.

gambit.db.migrate.**current_head**()

Get the current head revision number.

Return type
str

gambit.db.migrate.**current_revision**(*connectable*)

Get the current revision number of a genome database.

Parameters
connectable (*Connectable*) –

Return type
str

gambit.db.migrate.**get_alembic_config**(*connectable=None*, ***kwargs*)

Get an alembic config object to perform migrations.

Parameters

- **connectable** (*Optional[Connectable]*) – SQLAlchemy connectable specifying database connection info (optional). Assigned to `'connectable'` key of `alembic.config.Config.attributes`.
- ****kwargs** – Keyword arguments to pass to `alembic.config.Config.__init__()`.

Return type
Alembic config object.

gambit.db.migrate.**init_db**(*connectable*)

Initialize the genome database schema by creating all tables and stamping with the latest Alembic revision.

Expects a fresh database that does not already contain any tables for the *gambit.db.models* models and has not had any migrations run on it yet.

Parameters
connectable (*Connectable*) – SQLAlchemy connectable specifying database connection info.

Raises

- **RuntimeError** – If the database is already stamped with an Alembic revision.
- **sqlalchemy.exc.OperationalError** – If any of the database tables to be created already exist.

gambit.db.migrate.**is_current_revision**(*connectable*)

Check if the current revision of a genome database is the most recent (head) revision.

Parameters
connectable (*Connectable*) –

gambit.db.migrate.**upgrade**(*connectable*, *revision='head'*, *tag=None*, ***kwargs*)

Run the alembic upgrade command.

See `alembic.command.upgrade()` for more information on how this works.

Parameters

- **connectable** (*Connectable*) – SQLAlchemy connectable specifying genome database connection info.
- **revision** (*str*) – Revision to upgrade to. Passed to `alembic.command.upgrade()`.
- **tag** – Passed to `alembic.command.upgrade()`.
- ****kwargs** – Passed to *get_alembic_config()*.

1.5.4 Taxonomic Classification and Database Queries

gambit.classify

Classify queries based on distance to reference sequences.

class gambit.classify.**ClassifierResult**

Bases: object

Result of applying the classifier to a single query genome.

success

Whether the classification process ran successfully with no fatal errors. If True it is still possible no prediction was made.

Type
bool

predicted_taxon

Taxon predicted by classifier.

Type
Optional[*gambit.db.models.Taxon*]

primary_match

Match to closest reference genome which produced a predicted taxon equal to or a descendant of `predicted_taxon`. None if no prediction was made.

Type
Optional[*gambit.classify.GenomeMatch*]

closest_match

Match to closest reference genome overall. This should almost always be identical to `primary_match`.

Type
gambit.classify.GenomeMatch

next_taxon

Next most specific taxon for which the threshold was not met. Currently this just taken from the ancestry of `closest_match.genome.taxon`.

Type
Optional[*gambit.db.models.Taxon*]

warnings

List of non-fatal warning messages to report.

Type
List[str]

error

Message describing a fatal error which occurred, if any.

Type
Optional[str]

__init__(*success*, *predicted_taxon*, *primary_match*, *closest_match*, *next_taxon=_Nothing.NOTHING*, *warnings=_Nothing.NOTHING*, *error=None*)

Method generated by attrs for class ClassifierResult.

Parameters

- **success** (*bool*) –
- **predicted_taxon** (*Optional[Taxon]*) –
- **primary_match** (*Optional[GenomeMatch]*) –
- **closest_match** (GenomeMatch) –
- **next_taxon** (*Optional[Taxon]*) –
- **warnings** (*List[str]*) –
- **error** (*Optional[str]*) –

Return type
None

class gambit.classify.**GenomeMatch**

Bases: object

Match between a query and a single reference genome.

This is just used to report the distance from a query to some significant reference genome, it does not imply that this distance was close enough to actually make a taxonomy prediction or that the prediction was the primary prediction overall.

genome

Reference genome matched to.

Type
gambit.db.models.AnnotatedGenome

distance

Distance between query and reference genome.

Type
float

matching_taxon

Taxon prediction based off of this match alone. Will always be `genome.taxon` or one of its ancestors.

__init__(*genome*, *distance*, *matched_taxon=_Nothing.NOTHING*)

Method generated by attrs for class GenomeMatch.

Parameters

- **genome** (AnnotatedGenome) –
- **distance** (*float*) –
- **matched_taxon** (*Optional[Taxon]*) –

Return type
None

next_taxon()

Get next most specific taxon in lineage of `genome` for which the threshold was not met.

Return type
Optional[Taxon]

gambit.classify.**classify**(*ref_genomes*, *dists*, *, *strict=False*)

Predict the taxonomy of a query genome based on its distances to a set of reference genomes.

Parameters

- **ref_genomes** (*Sequence[AnnotatedGenome]*) – List of reference genomes from database.
- **dists** (*ndarray*) – Array of distances to each reference genome.
- **strict** (*bool*) – If true find all significant matches to reference genomes and attempt to reconcile them if they result in different taxa. If False just consider the top (closest) match. Defaults to False.

Return type
ClassifierResult

gambit.classify.**compare_classifier_results**(*result1*, *result2*)

Compare two `ClassifierResult` instances for equality.

Parameters

- **result1** (`ClassifierResult`) –
- **result2** (`ClassifierResult`) –

Return type
bool

gambit.classify.**compare_genome_matches**(*match1*, *match2*)

Compare two `GenomeMatch` instances for equality.

The values for the `distance` attribute are only checked for approximate equality, to support instances where one was loaded from a results archive (saving and loading a float in JSON is lossy).

Also allows one or both values to be None.

Parameters

- **match1** (*Optional[GenomeMatch]*) –
- **match2** (*Optional[GenomeMatch]*) –

Return type
bool

gambit.classify.**consensus_taxon**(*taxa*)

Take a set of taxa matching a query and find a single consensus taxon for classification.

If a query matches a given taxon, it is expected that there may be matches to some of that taxon's ancestors as well. In this case all matched taxa lie in a single lineage and the most specific will be the consensus.

It may also be possible for a query to match multiple taxa which are "inconsistent" with each other in the sense that one is not a descendant of another. In that case the consensus will be the lowest taxon which is either a descendant or ancestor of all taxa in the argument. It's also possible in pathological cases (depending on reference database design) that the taxa may be within entirely different trees, in which case the consensus will be `None`. The second element of the returned tuple is the set of taxa in the argument which are strict descendants of the consensus. This set will contain at least two taxa in the case of such an inconsistency and be empty otherwise.

Parameters
taxa (*Iterable[Taxon]*) –

Returns
Consensus taxon along with the set of any taxa in the argument which are descended from it.

Return type
Tuple[Optional[*Taxon*], Set[*Taxon*]]

gambit.classify.**find_matches**(*itr*)

Find taxonomy matches given distances from a query to a set of reference genomes.

Parameters
itr (*Iterable[Tuple[AnnotatedGenome, float]]*) – Iterable over (genome, distance) pairs.

Returns
Mapping from taxa to indices of genomes matched to them.

Return type
Dict[*Taxon*, List[Int]]

gambit.classify.**matching_taxon**(*taxon*, *d*)

Find first taxon in linage for which distance d is within its classification threshold.

Parameters

- **taxon** (Taxon) – Taxon to start searching from.
- **d** (*float*) – Distance value.

Returns
Most specific taxon in ancestry with threshold_distance >= d.

Return type
Optional[*Taxon*]

gambit.query

Run queries against a GAMBIT database to predict taxonomy of genome sequences.

class gambit.query.**QueryInput**

Bases: object

Information on a query genome.

label

Some unique label for the input, probably the file name.

Type
str

file

Source file (optional).

Type
Optional[*gambit.seq.SequenceFile*]

__init__(*label*, *file=None*)

Method generated by attrs for class QueryInput.

Parameters

- **label** (*str*) –
- **file** (*Optional[SequenceFile]*) –

Return type
None

classmethod convert(*x*)

Convenience function to convert flexible argument types into QueryInput.

Accepts single string label, `SequenceFile` (uses file path for label), or existing `QueryInput` instance (returned unchanged).

Parameters
x (*Union[QueryInput, SequenceFile, str]*) –

Return type
QueryInput

class gambit.query.**QueryParams**

Bases: `object`

Parameters for running a query.

classify_strict

`strict` parameter to *gambit.classify.classify()*. Defaults to False.

Type
bool

chunksize

Number of reference signatures to process at a time. `None` means no chunking is performed. Defaults to 1000.

Type
int

report_closest

Number of closest genomes to report in results. Does not affect classification.

Type
int

__init__(*classify_strict=False*, *chunksize=1000*, *report_closest=10*)

Method generated by attrs for class QueryParams.

Parameters

- **classify_strict** (*bool*) –
- **chunksize** (*int*) –
- **report_closest** (*int*) –

Return type
None

class gambit.query.**QueryResultItem**

Bases: `object`

Result for a single query sequence.

input

Information on input genome.

Type
gambit.query.QueryInput

classifier_result

Result of running classifier.

Type
gambit.classify.ClassifierResult

report_taxon

Final taxonomy prediction to be reported to the user.

Type
Optional[*gambit.db.models.Taxon*]

closest_genomes

List of closest reference genomes to query. Length determined by `QueryParams.report_closest`.

Type
List[*gambit.classify.GenomeMatch*]

__init__(*input*, *classifier_result*, *report_taxon=None*, *closest_genomes=_Nothing.NOTHING*)

Method generated by attrs for class QueryResultItem.

Parameters

- **input** (QueryInput) –
- **classifier_result** (ClassifierResult) –
- **report_taxon** (*Optional[Taxon]*) –
- **closest_genomes** (*List[GenomeMatch]*) –

Return type
None

class gambit.query.**QueryResults**

Bases: object

Results for a set of queries, as well as information on database and parameters used.

items

Results for each query sequence.

Type
List[*gambit.query.QueryResultItem*]

params

Parameters used to run query.

Type
Optional[*gambit.query.QueryParams*]

genomeset

Genome set used.

Type
Optional[*gambit.db.models.ReferenceGenomeSet*]

signaturesmeta

Metadata for signatures set used.

Type
Optional[*gambit.sigs.base.SignaturesMeta*]

gambit_version

Version of GAMBIT command/library used to generate the results.

Type
str

timestamp

Time query was completed.

Type
datetime.datetime

extra

JSON-able dict containing additional arbitrary metadata.

Type
Dict[str, Any]

__init__(*items*, *params=None*, *genomeset=None*, *signaturesmeta=None*, *gambit_version='1.0.0'*, *timestamp=_Nothing.NOTHING*, *extra=_Nothing.NOTHING*)

Method generated by attrs for class QueryResults.

Parameters

- **items** (*List[QueryResultItem]*) –
- **params** (*Optional[QueryParams]*) –
- **genomeset** (*Optional[ReferenceGenomeSet]*) –
- **signaturesmeta** (*Optional[SignaturesMeta]*) –
- **gambit_version** (*str*) –
- **timestamp** (*datetime*) –
- **extra** (*Dict[str, Any]*) –

Return type
None

gambit.query.**compare_result_items**(*item1*, *item2*)

Compare two `QueryResultItem` instances for equality.

Does not compare the value of the `input` attributes.

Parameters

- **item1** (QueryResultItem) –
- **item2** (QueryResultItem) –

Return type
bool

gambit.query.**get_result_item**(*db*, *params*, *dists*, *input*)

Perform classification and create result item object for single query input.

Parameters

- **db** (ReferenceDatabase) –
- **params** (QueryParams) –
- **dists** (*ndarray*) – Distances from query to reference genomes.

- **input** (QueryInput) –

Return type
QueryResultItem

gambit.query.**query**(*db*, *queries*, *params=None*, ***, *inputs=None*, *progress=None*, ***kw*)

Predict the taxonomy of one or more query genomes using a GAMBIT reference database.

Parameters

- **db** (ReferenceDatabase) – Database to query.
- **queries** (*Sequence[KmerSignature]*) – Sequence of k-mer signatures for query genomes.
- **params** (*Optional[QueryParams]*) – QueryParams instance defining parameter values. If None take values from additional keyword arguments or use defaults.
- **inputs** (*Optional[Sequence[Union[QueryInput, SequenceFile, str]]]*) – Description for each input, converted to gambit.query.result.QueryInput in results object. Only used for reporting, does not any other aspect of results. Items can be QueryInput, SequenceFile or str.
- **progress** – Report progress for distance matrix calculation and classification. See *gambit.util.progress.get_progress()* for description of allowed values.
- ****kw** – Passed to QueryParams.

Return type
QueryResults

gambit.query.**query_parse**(*db*, *files*, *params=None*, ***, *file_labels=None*, *parse_kw=None*, ***kw*)

Query a database with signatures derived by parsing a set of genome sequence files.

Parameters

- **db** (ReferenceDatabase) – Database to query.
- **files** (*Sequence[SequenceFile]*) – Sequence files containing query files.
- **params** (*Optional[QueryParams]*) – QueryParams instance defining parameter values. If None take values from additional keyword arguments or use defaults.
- **file_labels** (*Optional[Sequence[str]]*) – Custom labels to use for each file in returned results object. If None use file names.
- **parse_kw** (*Optional[Dict[str, Any]]*) – Keyword parameters to pass to *gambit.sigs.calc.calc_file_signatures()*.
- ****kw** – Additional keyword arguments passed to *query()*.

Return type
QueryResults

1.5.5 Results export

gambit.results

Export query results in various formats.

gambit.results.base

class gambit.results.base.**AbstractResultsExporter**

Bases: ABC

Base for classes that export formatted query results.

Subclasses must implement *export()*.

abstract export(*file_or_path*, *results*)

Write query results to file.

Parameters

- **file_or_path** (*Union[str, PathLike, IO]*) – Open file-like object or file path to write to.
- **results** (QueryResults) – Results to export.

class gambit.results.base.**BaseJSONResultsExporter**

Bases: *AbstractResultsExporter*

Base class for JSON exporters.

Subclasses need to implement the `to_json` method.

pretty

Write in more human-readable but less compact format. Defaults to False.

Type
bool

__init__(*pretty=False*)

Method generated by attrs for class BaseJSONResultsExporter.

Parameters
pretty (*bool*) –

Return type
None

export(*file_or_path*, *results*)

Write query results to file.

Parameters

- **file_or_path** (*Union[str, PathLike, TextIO]*) – Open file-like object or file path to write to.
- **results** (QueryResults) – Results to export.

to_json(*obj*)

Convert object to JSON-compatible format (need not work recursively).

gambit.results.base.**asdict_method**(*recurse=False*, ***kw*)

Create a `to_json` method which calls `attrs.asdict()` with the given options.

gambit.results.base.**export_to_buffer**(*results*, *exporter*)

Export query results to a *StringIO* buffer.

Parameters
results (QueryResults) –

Return type
StringIO

gambit.results.json

Export results to JSON.

class gambit.results.json.**JSONResultsExporter**

Bases: *BaseJSONResultsExporter*

Exports query results in basic JSON format.

Currently it assumes that the query was run with `classify_strict=False`, so the only relevant information from `ClassifierResult` is the closest genome match.

__init__(*pretty=False*)

Method generated by attrs for class JSONResultsExporter.

Parameters
pretty (*bool*) –

Return type
None

to_json(*obj*)

Convert object to JSON-compatible format (need not work recursively).

gambit.results.csv

Export query results to CSV.

class gambit.results.csv.**CSVResultsExporter**

Bases: *AbstractResultsExporter*

Exports query results in CSV format.

format_opts

Dialect and other formatting arguments passed to `csv.write()`.

Type
Dict[str, Any]

__init__(***format_opts*)

export(*file_or_path*, *results*)

Write query results to file.

Parameters

- **file_or_path** (*Union[str, PathLike, TextIO]*) – Open file-like object or file path to write to.

- **results** (QueryResults) – Results to export.

get_header()

Get values for header row.

Return type
List[str]

get_row(*item*)

Get row values for single result item.

Parameters
item (QueryResultItem) –

Return type
List

gambit.results.archive

Export results to JSON.

class gambit.results.archive.**ResultsArchiveReader**

Bases: object

Loads query results from file created by *ResultsArchiveWriter*.

session

SQLAlchemy session used to load database objects.

Type
sqlalchemy.orm.session.Session

__init__(*session*)

read(*file_or_path*)

Read query results from JSON file.

Parameters
file_or_path (*Union[str, PathLike, IO]*) – Readable file object or file path.

Return type
QueryResults

class gambit.results.archive.**ResultsArchiveWriter**

Bases: *BaseJSONResultsExporter*

Exports query results to "archive" format which captures all stored data.

This format is not intended to be read by users of the application. The exported data can be read and converted back into an identical QueryResults object using *ResultsArchiveReader*.

install_info

Add results of *gambit.util.dev.install_info()* to the QueryResults.extra dict.

Type
bool

__init__(*pretty=False*, *install_info=False*)

Method generated by attrs for class ResultsArchiveWriter.

Parameters

- **pretty** (*bool*) –
- **install_info** (*bool*) –

Return type
None

to_json(*obj*)

Convert object to JSON-compatible format (need not work recursively).

1.5.6 Command Line Interface

gambit.cli

GAMBIT command line interface.

gambit.cli.common

class gambit.cli.common.**CLIContext**

Bases: object

Click context object for GAMBIT CLI.

Loads reference database data lazily the first time it is requested.

Currently a single option (or environment variable) is used to specify the location of the database files, in the future options may be added to specify the reference genomes SQLite file and genome signatures file separately. Class methods treat them as being independent.

root_context

Click context object from root command group.

Type
click.core.Context

db_path

Path to directory containing database files, specified in root command group.

Type
Optional[pathlib.Path]

has_genomes

Whether reference genome metadata is available.

Type
bool

has_signatures

Whether reference signatures are available.

Type
bool

has_database

Whether reference genome metadata and reference signatures are both available.

Type
bool

engine

SQLAlchemy engine connecting to genomes database.

Type
Optional[sqlalchemy.engine.base.Engine]

Session

SQLAlchemy session maker for genomes database.

Type
Optional[sqlalchemy.orm.session.sessionmaker]

signatures

Reference genome signatures.

Type
Optional[*gambit.sigs.base.ReferenceSignatures*]

__init__(*root_context*)

Parameters
root_context (*Context*) – Click context object from root command group.

get_database()

Get reference database object.

Return type
ReferenceDatabase

require_database()

Raise an exception if genome metadata and signatures are not available.

require_genomes()

Raise an exception if genome metadata is not available.

require_signatures()

Raise an exception if signatures are not available.

gambit.cli.common.**check_params_group**(*ctx*, *names*, *exclusive*, *required*)

Check for the presence of the given parameter values and raise an informative error if needed.

Parameters

- **ctx** (*Context*) –
- **names** (*Iterable[str]*) – Parameter names.
- **exclusive** (*bool*) – No more than one of the parameters may be present.
- **required** (*bool*) – At least one of the parameters must be present.

Raises
click.ClickException –

gambit.cli.common.**cores_param**()

Click parameter for number of CPU cores.

gambit.cli.common.**dirpath**(***kw*)

Click Path argument type accepting directories only.

Return type
Path

gambit.cli.common.**filepath**(***kw*)

Click Path argument type accepting files only.

Return type
Path

gambit.cli.common.**genome_files_arg**()

Click positional argument for genome files.

gambit.cli.common.**get_file_id**(*path*, *strip_dir=True*, *strip_ext=True*)

Get sequence file ID derived from file path.

Parameters

- **strip_dir** (*bool*) – Strip leading path components.
- **strip_ext** (*bool*) – Strip file extension(s).
- **path** (*Union[str, PathLike]*) –

Return type
str

gambit.cli.common.**get_sequence_files**(*explicit=None*, *listfile=None*, *listfile_dir=None*, *strip_dir=True*, *strip_ext=True*)

Get list of sequence file paths and IDs from several types of CLI arguments.

Does not check for conflict between `explicit` and `listfile`.

Parameters

- **explicit** (*Optional[Iterable[Union[str, PathLike]]]*) – List of paths given explicitly, such as with a positional argument.
- **listfile** (*Union[None, str, PathLike, TextIO]*) – File listing sequence files, one per line.
- **listfile_dir** (*Optional[str]*) – Parent directory for files in `listfile`.
- **strip_dir** (*bool*) – Strip leading path components from file paths to derive IDs.
- **strip_ext** (*bool*) – Strip file extension from file names to derive IDs.

Returns
`(ids, files)` tuple. `ids` is a list of string IDs that can be used to label output. If the `explicit` and `listfile` arguments are None/empty both components of the tuple will be None as well.

Return type
Tuple[Optional[List[str]], Optional[List[*SequenceFile*]]]

gambit.cli.common.**kspec_from_params**(*k*, *prefix*, *default=False*)

Get KmerSpec from CLI arguments and validate.

Parameters

- **k** (*Optional[int]*) –
- **prefix** (*Optional[str]*) –
- **default** (*bool*) – Return default KmerSpec if arguments are None.

Return type
Optional[KmerSpec]

gambit.cli.common.**kspec_params**(*default=False*)

Returns a decorator to add k and prefix options to command.

Parameters
default (*bool*) – Whether to add default values.

gambit.cli.common.**listfile_dir_param**(**param*, *file_metavar=None*, ***kw*)

Returns decorator to add param for parent directory of paths in list file.

Parameters
param (*str*) –

gambit.cli.common.**listfile_param**(**param*, ***kw*)

Returns decorator to add param for file listing input paths.

Parameters
param (*str*) –

gambit.cli.common.**param_name_human**(*param*)

Get the name/metavar of the given parameter as it appears in the auto-generated help output.

Parameters
param (*Parameter*) –

Return type
str

gambit.cli.common.**params_by_name**(*cmd*, *names=None*)

Get parameters of click command by name.

Parameters

- **cmd** (*Command*) –
- **names** (*Optional[Iterable[str]]*) – Names of specific parameters to get.

Returns
Parameters with given in `names` argument if not None, otherwise a dictionary containing all of the command's parameters keyed by name.

Return type
Union[Dict[str, click.Parameter], List[click.Parameter]]

gambit.cli.common.**print_table**(*rows*, *colsep=' '*, *left=''*, *right=''*)

Print a basic table.

Parameters

- **rows** (*Sequence[Sequence]*) –
- **colsep** (*str*) –
- **left** (*str*) –
- **right** (*str*) –

gambit.cli.common.**progress_param**()

Click argument to show progress meter.

gambit.cli.common.**strip_seq_file_ext**(*filename*)

Strip FASTA and/or gzip extensions from sequence file name.

Parameters
filename (*str*) –

Return type
str

gambit.cli.common.**warn_duplicate_file_ids**(*ids*, *template*)

Print a warning message if duplicate file IDs are present.

Parameters

- **ids** (*List[str]*) – List of file ID strings, such as from *get_sequence_files()*.
- **template** (*str*) – Message template. May contain formatting placeholders for `ids` (comma-delimited string of duplicated IDs), `id` (first duplicated ID), and `n` (number of duplicated IDs).

gambit.cli.root

Define the root CLI command group.

gambit.cli.query

gambit.cli.signatures

gambit.cli.debug

gambit.cli.debug.**make_shell_ns**(*ctx*)

Make the user namespace for the shell command.

Return type
Dict[str, *Any*]

gambit.cli.debug.**SHELL_MODULES = {'kmers': 'gambit.kmers', 'metric': 'gambit.metric'}**

Modules to import in interactive shell.

1.5.7 Miscellaneous and Utility Code

gambit.util

gambit.util.misc

Utility code that doesn't fit anywhere else.

gambit.util.misc.**chunk_slices**(*n*, *size*)

Iterate over slice objects which split a sequence of length `n` into chunks of size `size`.

Parameters

- **n** (*int*) – Length of sequence.
- **size** (*int*) – Size of chunks (apart from last).

Return type
Iterator[slice]

gambit.util.misc.**is_importable**(*module*)

Check if the specified module is importable, without actually importing it.

Parameters
module (*str*) –

Return type
bool

gambit.util.misc.**join_list_human**(*strings*, *conj='and'*)

Join items into a single human-readable string with commas and the given conjunction.

Parameters

- **strings** (*Iterable[str]*) –
- **conj** (*str*) –

Return type
str

gambit.util.misc.**type_singledispatchmethod**(*func*)

Similar to `singledispatchmethod`, but the first (non-self) argument is expected to be a type and dispatch occurs on the argument's *value*.

Parameters
func (*Callable*) – Default implementation. Signature should start with `(self, cls: type, ...)`.

Returns
Function with `register` and `dispatch` attributes similar to `singledispatchmethod`.

Return type
Callable

gambit.util.misc.**zip_strict**(**iterables*)

Like the builtin `zip` function but raises an error if any argument is exhausted before the others.

Parameters
iterables (*Iterator*) – Any number of iterable objects.

Raises
ValueError –

Return type
Iterator[*Tuple*]

gambit.util.typing

Utilities based on the built-in `typing` module.

gambit.util.typing.**is_optional**(*T*)

Check if a parametrized union type is equivalent to one returned by `typing.Optional`.

Return type
bool

gambit.util.typing.**is_union**(*T*)

Check if a type annotation is a *parameterized* `typing.Union`.

Parameters
T – Result of `Union[A, B, ...]`.

Return type
bool

`gambit.util.typing.`**`union_types`**(*T*)

Get the types from a parameterized `typing.Union`.

Parameters
T – Result of `Union[A, B, ...]`.

Return type
tuple

`gambit.util.typing.`**`unwrap_optional`**(*u*)

Get `T` from `typing.Optional[T]`.

gambit.util.io

Utility code for reading/writing data files.

class `gambit.util.io.`**`ClosingIterator`**

Bases: `Iterable[T]`

Wraps an iterator which reads from a stream, closes the stream when finished.

Used to wrap return values from functions which do some sort of lazy IO operation (specifically `Bio.SeqIO.parse()`) and return an iterator which reads from a stream every time `next()` is called on it. The object is an iterator itself, but will close the stream automatically when it finishes. May also be used as a context manager which closes the stream on exit.

`fobj`

The underlying file-like object or stream which the instance is responsible for closing

`iterator`

The iterator which the instance wraps.

`closed`

Read-only boolean property, mirrors the same attribute of *`fobj`*.

Parameters

- **`iterable`** – Iterable to iterate over. The *`iterator`* attribute will be obtained from calling `iter()` on this.
- **`fobj`** – File-like object to close when the iterator finishes, context is exited or the *`close()`* method is called.

`__init__`(*iterable*, *fobj*)

`close`()

Close the stream.

Just calls the `close` method on *`fobj`*.

gambit.util.io.**guess_compression**(*fobj*)

Guess the compression mode of an readable file-like object in binary mode.

Assumes the current position is at the beginning of the file.

Parameters
fobj (*BinaryIO*) –

Return type
Optional[str]

gambit.util.io.**maybe_open**(*file_or_path*, *mode='r'*, ***open_kw*)

Open a file given a file path as an argument, but pass existing file objects though.

Intended to be used by API functions that take either type as an argument. If a file path is passed the function will need to call `open` to get the file object to use, and will need to close that object after it is done. If an existing file object is passed, it should be left to the caller of the function to close it afterwards. This function returns a context manager which performs the correct action for both opening and closing.

Parameters

- **file_or_path** (*Union[str, PathLike, IO]*) – A path-like object or open file object.
- **mode** (*str*) – Mode to open file in.
- ****open_kw** – Keyword arguments to `open()`.

Returns
Context manager which gives an open file object on enter and closes it on exit only if it was opened by this function.

Return type
ContextManager[IO]

gambit.util.io.**open_compressed**(*compression*, *path*, *mode='rt'*, ***kwargs*)

Open a file with compression method specified by a string.

Parameters

- **compression** (*str*) – Compression method. None is no compression. Keys of `COMPRESSED_OPENERS` are the allowed values.
- **path** (*Union[str, PathLike]*) – Path of file to open. May be string or path-like object.
- **mode** (*str*) – Mode to open file in - similar to `open()`. Must be exactly two characters, the first in `rwax` and the second in``tb``.
- ****kwargs** – Additional text-specific keyword arguments identical to the following `open()` arguments: `encoding`, `errors`, and `newlines`.

Returns
Open file object.

Return type
IO

gambit.util.io.**read_lines**(*file_or_path*, *strip=True*, *skip_empty=False*)

Iterate over lines in text file.

Parameters

- **file_or_path** (*Union[str, PathLike, IO]*) – A path-like object or open file object.
- **strip** (*bool*) – Strip whitespace from lines.

- **skip_empty** (*bool*) – Omit empty lines.

Returns
Iterator over lines with trailing newlines removed.

Return type
Iterable[str]

gambit.util.io.**write_lines**(*lines*, *file_or_path*)

Write strings to text file, one per line.

Parameters

- **lines** (*Iterable*) – Iterable of lines to write.
- **file_or_path** (*Union[str, PathLike, IO]*) – A path-like object or open file object.

gambit.util.io.**FilePath**

Alias for types which can represent a file system path

alias of Union[str, PathLike]

gambit.util.json

Convert data to/from JSON format.

The *dump()* and *load()* functions in this module work just like their built-in equivalents in the `json` module, but support additional types such as `attrs`-defined classes.

class gambit.util.json.**Jsonable**

Bases: `object`

Mixin class that provides custom JSON conversion methods.

Either of the special methods `__to_json__` and `__from_json__` may be set to `None` to indicate that the default conversion process should be used.

abstract __to_json__()

Convert the instance to JSON-writable data (anything that can be passed to `json.dump()`).

abstract classmethod __from_json__(*data*)

Create an instance from parsed JSON data.

gambit.util.json.**dump**(*obj*, *f*, ***kw*)

Write the JSON representation of an object to a file.

Parameters

- **obj** – Object to write.
- **f** (*TextIO*) – Writeable file object in text mode.
- ****kw** – Keyword arguments to `json.dump()`.

gambit.util.json.**dumps**(*obj*, ***kw*)

Get the JSON representation of an object as a string.

Parameters

- **obj** – Object to write.
- ****kw** – Keyword arguments to `json.dumps()`.

Return type
str

gambit.util.json.**from_json**(*data*, *cls=typing.Any*)

Load object from parsed JSON data.

Parameters

- **data** – Data parsed from JSON format.
- **cls** – Type to load.

Return type
Instance of `cls`

gambit.util.json.**load**(*f*, *cls=typing.Any*)

Load an object from a JSON file.

Parameters

- **f** (*TextIO*) – Readable file object in text mode.
- **cls** – Type to load.

Return type
Instance of `cls`

gambit.util.json.**loads**(*s*, *cls=typing.Any*)

Load an object from a JSON string.

Parameters

- **s** (*str*) – String containing JSON-encoded data.
- **cls** – Type to load.

Return type
Instance of `cls`

gambit.util.json.**register_hooks**(*cls*, *unstructure*, *structure*, *withtype=False*)

Register converter unstructure and structure hooks in the same call.

gambit.util.json.**register_structure_hook_notype**(*cls*, *f*)

Register converter structure hook taking no 2nd type argument.

gambit.util.json.**to_json**(*obj*)

Convert object to JSON-writable data (anything that can be passed to `json.dump()`).

Parameters
obj – Object to convert.

gambit.util.indexing

Tools to implement and test Numpy-style advanced indexing for Sequence types.

class gambit.util.indexing.**AdvancedIndexingMixin**

Bases: `object`

Mixin class for sequence types which enables Numpy-style advanced indexing.

Implements a `__getitem__` which takes care of identifying the type of index, performing bounds checking, and converting negative indices.

The following methods must be implemented by subtypes: * `_getitem_int()` * `_getitem_int_array()`

The following methods may optionally be overridden, but default to calling `_getitem_int_array()`: * `_getitem_range()` * `_getitem_bool_array()`

gambit.util.progress

Abstract interface for progress meters.

class gambit.util.progress.**AbstractProgressMeter**

Bases: ABC

Abstract base class for an object which displays progress to the user.

Instances can be used as context managers, on exit the *close()* method is called.

n

Number of completed iterations. Do not modify directly, use the *increment()* and *moveto()* methods instead.

Type
int

total

Expected total number of iterations.

Type
int

closed

Whether the meter has been closed/completed.

Type
int

close()

Stop displaying progress and perform whatever cleanup is necessary.

classmethod config(***kw*)

Create a factory function which creates instances with the given default settings.

Keyword arguments are passed on to *create()*.

Return type
ProgressConfig

abstract classmethod create(*total*, ***, *initial=0*, *desc=None*, *file=None*, ***kw*)

Factory function with standardized signature to create instances.

Parameters

- **total** (*int*) – Total number of iterations to completion.
- **initial** (*int*) – Initial value of *n*.
- **desc** (*Optional[str]*) – Description to display to the user.
- **file** (*Optional[TextIO]*) – File-like object to write to. Defaults to `sys.stderr`.
- ****kw** – Additional options depending on the subclass.

Return type
AbstractProgressMeter

abstract increment(*delta=1*)

Increment the position of the meter by the given value.

Parameters
delta (*int*) –

abstract moveto(*n*)

Set the meter's position to the given value.

Parameters
n (*int*) –

class gambit.util.progress.**ClickProgressMeter**

Bases: *AbstractProgressMeter*

Wrapper around a progress bar from the `click` library.

__init__(*pbar*)

close()

Stop displaying progress and perform whatever cleanup is necessary.

classmethod create(*total*, *, *initial=0*, *desc=None*, *file=None*, ***kw*)

Factory function with standardized signature to create instances.

Parameters

- **total** (*int*) – Total number of iterations to completion.
- **initial** (*int*) – Initial value of `n`.
- **desc** (*Optional[str]*) – Description to display to the user.
- **file** (*Optional[TextIO]*) – File-like object to write to. Defaults to `sys.stderr`.
- ****kw** – Additional options depending on the subclass.

increment(*delta=1*)

Increment the position of the meter by the given value.

Parameters
delta (*int*) –

moveto(*n*)

Set the meter's position to the given value.

Parameters
n (*int*) –

class gambit.util.progress.**NullProgressMeter**

Bases: *AbstractProgressMeter*

Progress meter which does nothing.

close()

Stop displaying progress and perform whatever cleanup is necessary.

classmethod create(*total*, *initial=0*, ***kw*)

Factory function with standardized signature to create instances.

Parameters

- **total** (*int*) – Total number of iterations to completion.

- **initial** (*int*) – Initial value of n.
- **desc** – Description to display to the user.
- **file** – File-like object to write to. Defaults to sys.stderr.
- ****kw** – Additional options depending on the subclass.

increment(*delta=1*)

Increment the position of the meter by the given value.

Parameters
delta (*int*) –

moveto(*n*)

Set the meter's position to the given value.

Parameters
n (*int*) –

class gambit.util.progress.**ProgressConfig**

Bases: object

Configuration settings used to create new progress meter instances.

This allows callers to pass the desired progress meter type and other settings to API functions which can then create the instance themselves within the function body by specifying the total length and other final options.

callable

The *AbstractProgressMeter.create()* method of a concrete progress meter type, or another callable with the same signature which returns a progress meter instance.

Type
Callable[[int], *gambit.util.progress.AbstractProgressMeter*]

kw

Keyword arguments to pass to callable.

Type
Dict[str, Any]

__init__(*callable*, *kw*)

Parameters

- **callable** (*Callable[[int], AbstractProgressMeter]*) –
- **kw** (*Dict[str, Any]*) –

create(*total*, ***kw*)

Call the factory function with the stored keyword arguments to create a progress meter instance.

The signature of this function is identical to *AbstractProgressMeter.create()*.

Parameters
total (*int*) –

Return type
AbstractProgressMeter

update(**args*, ***kw*)

Update keyword arguments and return a new instance.

Parameters
args (*Mapping[str, Any]*) –

class gambit.util.progress.**ProgressIterator**

Bases: Iterator

__init__(*iterable*, *meter*)

Parameters

- **iterable** (*Iterable*) –
- **meter** (AbstractProgressMeter) –

class gambit.util.progress.**TestProgressMeter**

Bases: *AbstractProgressMeter*

Progress meter which displays no user information but does track progress information.

To be used for testing.

__init__(*total*, *initial=0*, *allow_decrement=True*, ***kw*)

Parameters

- **total** (*int*) –
- **initial** (*int*) –
- **allow_decrement** (*bool*) –

close()

Stop displaying progress and perform whatever cleanup is necessary.

classmethod create(*total*, *initial=0*, ***kw*)

Factory function with standardized signature to create instances.

Parameters

- **total** (*int*) – Total number of iterations to completion.
- **initial** (*int*) – Initial value of n.
- **desc** – Description to display to the user.
- **file** – File-like object to write to. Defaults to sys.stderr.
- ****kw** – Additional options depending on the subclass.

increment(*delta=1*)

Increment the position of the meter by the given value.

Parameters
delta (*int*) –

moveto(*n*)

Set the meter's position to the given value.

Parameters
n (*int*) –

class gambit.util.progress.**TqdmProgressMeter**

Bases: *AbstractProgressMeter*

Wrapper around a progress meter from the tqdm library.

__init__(*pbar*)

Parameters
pbar (*tqdm.std.tqdm*) –

close()

Stop displaying progress and perform whatever cleanup is necessary.

classmethod create(*total*, *, *initial=0*, *desc=None*, *file=None*, ***kw*)

Factory function with standardized signature to create instances.

Parameters

- **total** (*int*) – Total number of iterations to completion.
- **initial** (*int*) – Initial value of n.
- **desc** (*Optional[str]*) – Description to display to the user.
- **file** (*Optional[TextIO]*) – File-like object to write to. Defaults to sys.stderr.
- ****kw** – Additional options depending on the subclass.

increment(*delta=1*)

Increment the position of the meter by the given value.

Parameters
delta (*int*) –

moveto(*n*)

Set the meter's position to the given value.

Parameters
n (*int*) –

gambit.util.progress.**capture_progress**(*config*)

Creates a ProgressConfig which captures references to the progress meter instances created with it.

This is intended to be used for testing other API functions which create progress meter instances internally that normally would not be accessible by the caller. The captured instance can be checked to ensure it has the correct attributes and went through the full range of iterations, for example.

Returns
The first item is a modified ProgessConfig instance which can be passed to the function to be tested. The second is a list which is initially empty, and is populated with progress meter instances as they are created by it.

Return type
Tuple[*ProgressConfig*, List[*AbstractProgressMeter*]]

Parameters
config (ProgressConfig) –

gambit.util.progress.**check_progress**(*, *total=None*, *allow_decrement=False*, *check_closed=True*)

Context manager which checks a progress meter is advanced to completion.

Returned context manager yields a ProgressConfig instance on enter, tests are run when context is exited. Expects that the config will be used to instantiate exactly one progress meter. Tests are performed with assert statements.

Parameters

- **total** (*Optional[int]*) – Check that the progress meter is created with this total length.

- **allow_decrement** (*bool*) – If false, raise an error if the created progress meter is moved backwards.
- **check_closed** (*bool*) – Check that the progress meter was closed.

Raises
AssertionError – If any checks fail.

Return type
AbstractContextManager[ProgressConfig]

gambit.util.progress.**default_progress_cls**()

Get the default *AbstractProgressMeter* subclass to use.

Currently returns *TqdmProgressMeter* if `tqdm` is importable, otherwise prints a warning and returns *NullProgressMeter*.

Return type
type

gambit.util.progress.**get_progress**(*arg*, *total*, *initial=0*, ***kw*)

Get a progress meter instance.

Meant to be used within other API funcs in which the caller wants to specify the type and parameters of the progress meter but cannot pass an actual instance because the total number of iterations is determined within the body of the function. Instead the API function can take a single `progress` argument which specifies this information, then create the instance by by calling this function with than information along with the total length.

Accepts the following types/values for the argument:

- *ProgressConfig*
- `None` - uses `NullProgressBar`.
- `True` - uses class returned by *default_progress_cls()*.
- `False` - same as `None`.
- `str` key - Looks up progress bar class/factory function in *REGISTRY*.
- *AbstractProgressMeter* subclass
- `callable` - factory function. Must have same signature as *AbstractProgressMeter.create()*.

Parameters

- **arg** (*Optional[Union[*ProgressConfig*, str, bool, type, Callable[[int],* AbstractProgressMeter*]]]*) – See above.
- **total** (*int*) – Length of progress meter to create.
- **initial** (*int*) – Initial position of progress meter.
- ****kw** – Additional keyword arguments to pass to progress meter class or factory function defined by `arg`..

Return type
AbstractProgressMeter

gambit.util.progress.**iter_progress**(*iterable*, *progress=True*, *total=None*, ***kw*)

Display a progress meter while iterating over an object.

The returned iterator object can also be used as a context manager to ensure that the progress meter is closed properly even if iteration does not finish.

Parameters

- **itarable** – Iterable object.
- **progress** (*Optional[Union[ProgressConfig, str, bool, type, Callable[[int], AbstractProgressMeter]]]*) – Passed to *get_progress()*.
- **total** (*Optional[int]*) – Total number of expected iterations. Defaults to `len(iterable)`.
- ****kw** – Additional keyword arguments to pass to progress meter factory.
- **iterable** (*Iterable*) –

Returns
Iterator over values in `iterable` which advances a progress meter.

Return type
.ProgressIterator

gambit.util.progress.**progress_config**(*arg*, ***kw*)

Get a `ProgressConfig` instance from flexible argument types.

See *get_progress()* for description of allowed argument types/values.

Parameters
arg (*Optional[Union[ProgressConfig, str, bool, type, Callable[[int], AbstractProgressMeter]]]*) –

Return type
ProgressConfig

gambit.util.progress.**register**(*key*)

Decorator to register progress meter class or factory function under the given key.

Parameters
key (*str*) –

gambit.util.progress.**ProgressFactoryFunc**

Type alias for a callable which takes `total` and keyword arguments and returns an AbstractProgressMeter

alias of `Callable[[int], AbstractProgressMeter]`

gambit.util.progress.**REGISTRY = {'click': <bound method ClickProgressMeter.create of <class 'gambit.util.progress.ClickProgressMeter'>>, 'tqdm': <bound method TqdmProgressMeter.create of <class 'gambit.util.progress.TqdmProgressMeter'>>}**

TODO

gambit.util.dev

Development tools.

gambit.util.dev.**get_commit_info**(*repo_path*, *commit='HEAD'*)

Get metadata on a git commit.

This calls the `git` command, so it must be installed and available.

Parameters

- **repo_path** (*Union[str, PathLike]*) – Path to git repo.
- **commit** (*str*) – Commit to get information on.

Return type
Dict[str, str]

gambit.util.dev.**install_info**()

Get information on the GAMBIT installation if it is installed in development mode.

If gambit is installed via the setuptools development install method (`pip install -e`), this checks if the source directory is a valid git repo and tries to get information on the current commit. This is used to mark exported results from development versions of the software which do not correspond to an official release.

Return type
Dict[str, *Any*]

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

S

T

U

V

W

Z