SC Tools Documentation

Release 0.4.0

Ambrose J. Carr

Aug 30, 2021

OVERVIEW

Single Cell Tools provides utilities for manipulating sequence data formats suitable for use in distributed systems analyzing large biological datasets.

CHAPTER

ONE

DOWNLOAD AND INSTALLATION

CHAPTER

TWO

SCTOOLS PACKAGE

The sctools package provides both command line utilities and classes designed for use in python programs.

CHAPTER

THREE

COMMAND LINE UTILITIES

1. Attach10XBarcodes: Attached barcodes stored in fastq files to reads in an unaligned bam file
2. SplitBam: Split a bam file into chunks, guaranteeing that cells are contained in 1 chunk
3. CalculateGeneMetrics: Calculate information about genes in an experiment or chunk
4. CalculateCellMetrics: Calculate information about cells in an experiment or chunk
5. MergeGeneMetrics: Merge gene metrics calculated from different chunks of an experiment
6. MergeCellMetrics Merge cell metrics calculated from different chunks of an experiment

CHAPTER

FOUR

MAIN PACKAGE CLASSES

1. **Platform**: an abstract class that defines a common data structure for different 3' sequencing formats. All algorithms and methods in this package that are designed to work on 3' sequencing data speak to this common data structure. Currently 10X_v2 is defined.
2. **Reader**: a general iterator over arbitrarily zipped file(s) that is extended to work with common sequence formats like fastq (fastq.Reader) and gtf (gtf.Reader). We recommend using the pysam package for reading sam and bam files.
3. **TwoBit & ThreeBit** DNA encoders that store DNA in 2- and 3-bit form. 2-bit is smaller but randomizes "N" nucleotides. Both classes support fastq operations over common sequence tasks such as the calculation of GC content.
4. **ObservedBarcodeSet & PriorBarcodeSet**: classes for analysis and comparison of sets of barcodes such as the cell barcodes used by 10X genomics. Supports operations like summarizing hamming distances and comparing observed sequence diversity to expected (normally uniform) diversity.
5. **gtf.Reader & gtf.Record** GTF iterator and GTF record class that exposes the gtf fields as a lightweight, lazy-parsed python object.
6. **fastq.Reader & fastq.Record** fastq reader and fastq record class that exposes the fastq fields as a lightweight, lazy-parsed python object.
7. **Metrics** calculate information about the genes and cells of an experiment
8. **Bam** Split bam files into chunks and attach barcodes as tags

CHAPTER

FIVE

VIEWING TEST RESULTS AND COVERAGE

To calculate and view test coverage cd to the `sctools` directory and type the following two commands to generate the report and open it in your web browser:

```
pytest --cov-report html:cov_html --cov=sctools
open cov_html/index.html
```

CHAPTER

SIX

DEFINITIONS

Several definitions are helpful to understand how sequence data is analyzed.

1. **Cell**: an individual cell, the target of single-cell RNA-seq experiments and the entity that we with to characterize

2. **Capture Primer**: A DNA oligonucleotide containing amplification machinery, a fixed cell barcode, a random molecule barcode, and an oligo-dT tail to capture poly-adenylated RNA

3. **Molecule**: A molecule refers to a single mRNA molecule that is captured by an oligo-dT capture primer in a single-cell sequencing experiment

4. **Molecule Barcode**: A molecule barcode (alias: UMI, RMT) is a short, random DNA barcode attached to the capture primer that has adequate length to be probabilistically unique across the experiment. Therefore, when multiple molecules of the same gene are captured in the same cell, they can be differentiated through having different molecule barcodes. The proposed GA4GH standard tag for a molecule barcode is UB and molecule barcode qualities is UY

5. **Cell Barcode**: A short DNA barcode that is typically selected from a whitelist of barcodes that will be used in an experiment. All capture primers for a given cell will contain the same cell barcode. The proposed GA4GH standard tag for a cell barcode is CB and cell barcode qualities is CY

6. **Fragment**: During library construction, mRNA molecules captured on capture primers are amplified, and the resulting amplified oligonucleotides are fragmented. In 3' experiments, only the fragment that contains the 3' end is retained, but the break point will be random, which means fragments often have different lengths. Once sequenced, different fragments can be identified as unique combinations of cell barcode, molecule barcode, the chromosome the sequence aligns to, and the position it aligns to on that chromosome, after correcting for clipping that the aligner may add

7. **Bam/Sam file**: The GA4GH standard file type for the storage of aligned sequencing reads. Unless specified, our Single Cell Tools will operate over bam files containing either aligned or unaligned reads

CHAPTER

SEVEN

DEVELOPMENT

7.1 Code Style

The sctools code base is complying with the PEP-8 and using Black to format our code, in order to avoid "nitpicky" comments during the code review process so we spend more time discussing about the logic, not code styles.

In order to enable the auto-formatting in the development process, you have to spend a few seconds setting up the `pre-commit` the first time you clone the repo:

1. Install `pre-commit` by running: `pip install pre-commit` (or simply run `pip install -r requirements.txt`).
2. Run *pre-commit install* to install the git hook.

Once you successfully install the `pre-commit` hook to this repo, the Black linter/formatter will be automatically triggered and run on this repo. Please make sure you followed the above steps, otherwise your commits might fail at the linting test!

If you really want to manually trigger the linters and formatters on your code, make sure `Black` and `flake8` are installed in your Python environment and run `flake8 DIR1 DIR2` and `black DIR1 DIR2 --skip-string-normalization` respectively.

CHAPTER

EIGHT

SCTOOLS PACKAGE

8.1 Submodules

8.1.1 sctools.bam module

Tools for Manipulating SAM/BAM format files

This module provides functions and classes to subsample reads from bam files that correspond to specific chromosomes, split bam files into chunks, assign tags to bam files from paired fastq records, and iterate over sorted bam files by one or more tags

This module makes heavy use of the pysam wrapper for HTSlib, a high-performance c-library designed to manipulate sam files

iter_tag_groups	function to iterate over reads by an arbitrary tag
iter_cell_barcodes	wrapper for iter_tag_groups that iterates over cell barcode tags
iter_genes	wrapper for iter_tag_groups that iterates over gene tags
iter_molecules	wrapper for iter_tag_groups that iterates over molecule tags
sort_by_tags_and_queryname followed by query name	sort bam by given list of zero or more tags,
verify_sort then query name	verifies whether bam is correctly sorted by given list of tags,

sctools.Classes()

SubsetAlignments	class to extract reads specific to requested chromosome(s)
Tagger bam records from paired fastq records	class to add tags to sam/
AlignmentSortOrder	abstract class to represent alignment sort orders
QueryNameSortOrder	alignment sort order by query name
TagSortableRecord AlignedSegments	class to facilitate sorting of pysam.
SortError	error raised when sorting is incorrect

References

htslib : https://github.com/samtools/htslib

class sctools.bam.**AlignmentSortOrder**
Bases: object

The base class of alignment sort orders.

abstract property key_generator: Callable[pysam.libcalignedsegment.AlignedSegment, Any]
Returns a callable function that calculates a sort key from given pysam.AlignedSegment.

class sctools.bam.**QueryNameSortOrder**
Bases: *sctools.bam.AlignmentSortOrder*

Alignment record sort order by query name.

static get_sort_key(*alignment: pysam.libcalignedsegment.AlignedSegment*) → str

property key_generator
Returns a callable function that calculates a sort key from given pysam.AlignedSegment.

exception sctools.bam.**SortError**
Bases: Exception

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class sctools.bam.**SubsetAlignments**(*alignment_file: str*, *open_mode: Optional[str] = None*)
Bases: object

Wrapper for pysam/htslib that extracts reads corresponding to requested chromosome(s)

Parameters

- **alignment_file** (*str*) – sam or bam file
- **open_mode** (*{'r', 'rb', None}, optional*) – open mode for pysam.AlignmentFile. 'r' indicates a sam file, 'rb' indicates a bam file, and None attempts to autodetect based on the file suffix (Default = None)

indices_by_chromosome()
returns indices to line numbers containing the requested number of reads for a specified chromosome

Notes

samtools is a good general-purpose tool for that is capable of most subsampling tasks. It is a good idea to check the samtools documentation when approaching these types of tasks.

References

samtools documentation : http://www.htslib.org/doc/samtools.html

indices_by_chromosome(*n_specific: int*, *chromosome: str*, *include_other: int = 0*) → Union[List[int], Tuple[List[int], List[int]]]

Return the list of first *n_specific* indices of reads aligned to *chromosome*.

Parameters

- **n_specific** (*int*) – Number of aligned reads to return indices for
- **chromosome** (*str*) – Only reads from this chromosome are considered valid
- **include_other** (*int, optional*) – The number of reads to include that are NOT aligned to chromosome. These can be aligned or unaligned reads (default = 0).

Returns

- **chromosome_indices** (*List[int]*) – list of indices to reads aligning to *chromosome*
- **other_indices** (*List[int], optional*) – list of indices to reads NOT aligning to chromosome, only returned if include_other is not 0.

class sctools.bam.**TagSortableRecord**(*tag_keys: Iterable[str]*, *tag_values: Iterable[str]*, *query_name: str*, *record: Optional[pysam.libcalignedsegment.AlignedSegment] = None*)

Bases: object

Wrapper for pysam.AlignedSegment that facilitates sorting by tags and query name.

classmethod from_aligned_segment(*record: pysam.libcalignedsegment.AlignedSegment*, *tag_keys: Iterable[str]*) → *sctools.bam.TagSortableRecord*

Create a TagSortableRecord from a pysam.AlignedSegment and list of tag keys

class sctools.bam.**Tagger**(*bam_file: str*)

Bases: object

Add tags to a bam file from tag generators.

Parameters **bam_file** (*str*) – Bam file that tags are to be added to.

tag()

tag bam records given tag_generators (often generated from paired bam or fastq files) # todo this should probably be wrapped up in __init__ to make this more function-like

tag(*output_bam_name: str*, *tag_generators*) → None

Add tags to bam_file.

Given a bam file and tag generators derived from files sharing the same sort order, adds tags to the .bam file, and writes the resulting file to output_bam_name.

Parameters

- **output_bam_name** (*str*) – Name of output tagged bam.
- **tag_generators** (*List[fastq.TagGenerator]*) – list of generators that yield fastq.Tag objects

sctools.bam.**get_barcode_for_alignment**(*alignment: pysam.libcalignedsegment.AlignedSegment*, *tags: List[str]*, *raise_missing: bool*) → str

Get the barcode for an Alignment

Parameters

- **alignment** – pysam.AlignedSegment An Alignment from pysam.
- **tags** – List[str] Tags in the bam that might contain barcodes. If multiple Tags are passed, will return the contents of the first tag that contains a barcode.
- **raise_missing** – bool Raise an error if no barcodes can be found.

Returns str A barcode for the alignment, or None if one is not found and raise_missing is False.

sctools.bam.**get_barcodes_from_bam**(*in_bam: str*, *tags: List[str]*, *raise_missing: bool*) → Set[str]
Get all the distinct barcodes from a bam

Parameters

- **in_bam** – str Input bam file.
- **tags** – List[str] Tags in the bam that might contain barcodes.
- **raise_missing** – bool Raise an error if no barcodes can be found.

Returns set A set of barcodes found in the bam This set will not contain a None value

sctools.bam.**get_tag_or_default**(*alignment: pysam.libcalignedsegment.AlignedSegment*, *tag_key: str*, *default: Optional[str] = None*) → Optional[str]
Extracts the value associated to *tag_key* from *alignment*, and returns a default value if the tag is not present.

sctools.bam.**iter_cell_barcodes**(*bam_iterator: Iterator[pysam.libcalignedsegment.AlignedSegment]*) → Generator
Iterate over all the cells of a bam file sorted by cell.

Parameters **bam_iterator** (*Iterator[pysam.AlignedSegment]*) – open bam file that can be iterated over

Yields

- **grouped_by_tag** (*Iterator[pysam.AlignedSegment]*) – reads sharing a unique cell barcode tag
- **current_tag** (*str*) – the cell barcode that reads in the group all share

sctools.bam.**iter_genes**(*bam_iterator: Iterator[pysam.libcalignedsegment.AlignedSegment]*) → Generator
Iterate over all the cells of a bam file sorted by gene.

Parameters **bam_iterator** (*Iterator[pysam.AlignedSegment]*) – open bam file that can be iterated over

Yields

- **grouped_by_tag** (*Iterator[pysam.AlignedSegment]*) – reads sharing a unique gene name tag
- **current_tag** (*str*) – the gene id that reads in the group all share

sctools.bam.**iter_molecule_barcodes**(*bam_iterator: Iterator[pysam.libcalignedsegment.AlignedSegment]*) → Generator
Iterate over all the molecules of a bam file sorted by molecule.

Parameters **bam_iterator** (*Iterator[pysam.AlignedSegment]*) – open bam file that can be iterated over

Yields

- **grouped_by_tag** (*Iterator[pysam.AlignedSegment]*) – reads sharing a unique molecule barcode tag
- **current_tag** (*str*) – the molecule barcode that records in the group all share

sctools.bam.**iter_tag_groups**(*tag: str*, *bam_iterator: Iterator[pysam.libcalignedsegment.AlignedSegment]*, *filter_null: bool = False*) → Generator

Iterates over reads and yields them grouped by the provided tag value

Parameters

- **tag** (*str*) – BAM tag to group over
- **bam_iterator** (*Iterator[pysam.AlignedSegment]*) – open bam file that can be iterated over
- **filter_null** (*bool, optional*) – If False, all reads that lack the requested tag are yielded together. Else, all reads that lack the tag will be discarded (default = False).

Yields

- **grouped_by_tag** (*Iterator[pysam.AlignedSegment]*) – reads sharing a unique value of tag
- **current_tag** (*str*) – the tag that reads in the group all share

sctools.bam.**merge_bams**(*bams: List[str]*) → str

Merge input bams using samtools.

This cannot be a local function within *split* because then Python "cannot pickle a local object". :param bams: Name of the final bam + bams to merge.

Because of how its called using multiprocessing, the bam basename is the first element of the list.

Returns The output bam name.

sctools.bam.**sort_by_tags_and_queryname**(*records: Iterable[pysam.libcalignedsegment.AlignedSegment]*, *tag_keys: Iterable[str]*) → Iterable[pysam.libcalignedsegment.AlignedSegment]

Sorts the given bam records by the given tags, followed by query name. If no tags are given, just sorts by query name.

sctools.bam.**split**(*in_bams: List[str]*, *out_prefix: str*, *tags: List[str]*, *approx_mb_per_split: float = 1000*, *raise_missing: bool = True*, *num_processes: Optional[int] = None*) → List[str]

split *in_bam* by tag into files of *approx_mb_per_split*

Parameters

- **in_bams** (*str*) – Input bam files.
- **out_prefix** (*str*) – Prefix for all output files; output will be named as prefix_n where n is an integer equal to the chunk number.
- **tags** (*List[str]*) – The bam tags to split on. The tags are checked in order, and sorting is done based on the first identified tag. Further tags are only checked if the first tag is missing. This is useful in cases where sorting is executed over a corrected barcode, but some records only have a raw barcode.
- **approx_mb_per_split** (*float*) – The target file size for each chunk in mb
- **raise_missing** (*bool, optional*) – if True, raise a RuntimeError if a record is encountered without a tag. Else silently discard the record (default = True)
- **num_processes** (*int, optional*) – The number of processes to parallelize over. If not set, will use all available processes.

Returns **output_filenames** – list of filenames of bam chunks

Return type List[str]

Raises

- **ValueError** – when *tags* is empty
- **RuntimeError** – when *raise_missing* is true and any passed read contains no *tags*

sctools.bam.**verify_sort**(*records: Iterable[*sctools.bam.TagSortableRecord*]*, *tag_keys: Iterable[str]*) → None
Raise AssertionError if the given records are not correctly sorted by the given tags and query name

sctools.bam.**write_barcodes_to_bins**(*in_bam: str*, *tags: List[str]*, *barcodes_to_bins: Dict[str, int]*, *raise_missing: bool*) → List[str]
Write barcodes to appropriate bins as defined by barcodes_to_bins

Parameters

- **in_bam** – str The bam file to read.
- **tags** – List[str] Tags in the bam that might contain barcodes.
- **barcodes_to_bins** – Dict[str, int] A Dict from barcode to bin. All barcodes of the same type need to be written to the same bin. These numbered bins are merged after parallelization so that all alignments with the same barcode are in the same bam.
- **raise_missing** – bool Raise an error if no barcodes can be found.

Returns A list of paths to the written bins.

8.1.2 sctools.barcode module

Nucleotide Barcode Manipulation Tools

This module contains tools to characterize oligonucleotide barcodes and a simple hamming-base error-correction approach which corrects barcodes within a specified distance of a "whitelist" of expected barcodes.

Classes

Barcodes Class to characterize a set of barcodes ErrorsToCorrectBarcodesMap Class to carry out error correction routines

class sctools.barcode.**Barcodes**(*barcodes: Mapping[str, int]*, *barcode_length: int*)
Bases: object

Container for a set of nucleotide barcodes.

Contained barcodes are encoded in 2bit representation for fast operations. Instances of this class can optionally be constructed from an iterable where barcodes can be present multiple times. In these cases, barcodes are analyzed based on their observed frequencies.

Parameters

- **barcodes** (*Mapping[str, int]*) – dictionary-like mapping barcodes to the number of times they were observed
- **barcode_length** (*int*) – the length of all barcodes in the set. Different-length barcodes are not supported.

See also:

sctools.encodings.TwoBit

base_frequency(*weighted=False*) → numpy.ndarray
return the frequency of each base at each position in the barcode set

Notes

weighting is currently not supported, and must be set to False or base_frequency will raise NotImplementedError # todo fix

Parameters **weighted** (*bool, optional*) – if True, each barcode is counted once for each time it was observed (default = False)

Returns **frequencies** – barcode_length x 4 2d numpy array

Return type np.array

Raises **NotImplementedError** – if weighted is True

effective_diversity(*weighted=False*) → numpy.ndarray
Returns the effective base diversity of the barcode set by position.

maximum diversity for each position is 1, and represents a perfect split of 25% per base at a given position.

Parameters **weighted** (*bool, optional*) – if True, each barcode is counted once for each time it was observed (default = False)

Returns **effective_diversity** – 1-d array of size barcode_length containing floats in [0, 1]

Return type np.array[float]

classmethod from_iterable_bytes(*iterable: Iterable[bytes]*, *barcode_length: int*)
Construct an ObservedBarcodeSet from an iterable of bytes barcodes.

Parameters

- **iterable** (*Iterable[bytes]*) – iterable of barcodes in bytes representation
- **barcode_length** (*int*) – the length of the barcodes in *iterable*

Returns **barcodes** – class object containing barcodes from a whitelist file

Return type *Barcodes*

classmethod from_iterable_encoded(*iterable: Iterable[int]*, *barcode_length: int*)
Construct an ObservedBarcodeSet from an iterable of encoded barcodes.

Parameters

- **iterable** (*Iterable[int]*) – iterable of barcodes encoded in TwoBit representation
- **barcode_length** (*int*) – the length of the barcodes in *iterable*

Returns **barcodes** – class object containing barcodes from a whitelist file

Return type *Barcodes*

classmethod from_iterable_strings(*iterable: Iterable[str]*, *barcode_length: int*)
Construct an ObservedBarcodeSet from an iterable of string barcodes.

Parameters

- **iterable** (*Iterable[str]*) – iterable of barcodes encoded in TwoBit representation
- **barcode_length** (*int*) – the length of the barcodes in *iterable*

Returns **barcodes** – class object containing barcodes from a whitelist file

Return type *Barcodes*

classmethod from_whitelist(*file_: str*, *barcode_length: int*)
Creates a barcode set from a whitelist file.

Parameters

- **file** (*str*) – location of the whitelist file. Should be formatted one barcode per line. Barcodes should be encoded in plain text (UTF-8, ASCII), not bit-encoded. Each barcode will be assigned a count of 1.
- **barcode_length** (*int*) – Length of the barcodes in the file.

Returns **barcodes** – class object containing barcodes from a whitelist file

Return type *Barcodes*

summarize_hamming_distances() → Mapping[str, float]
Returns descriptive statistics on hamming distances between pairs of barcodes.

Returns **descriptive_statistics** – minimum, 25th percentile, median, 75th percentile, maximum, and average hamming distance between all pairs of barcodes

Return type Mapping[str, float]

References

https://en.wikipedia.org/wiki/Hamming_distance

class sctools.barcode.**ErrorsToCorrectBarcodesMap**(*errors_to_barcodes: Mapping[str, str]*)
Bases: object

Correct any barcode that is within one hamming distance of a whitelisted barcode

Parameters **errors_to_barcodes** (*Mapping[str, str]*) – dict-like mapping 1-base errors to the whitelist barcode that they could be generated from

get_corrected_barcode(*barcode: str*)
Return a barcode if it is whitelist, or the corrected version if within edit distance 1

correct_bam(*bam_file: str*, *output_bam_file: str*)
correct barcodes in a bam file, given a whitelist

References

https://en.wikipedia.org/wiki/Hamming_distance

correct_bam(*bam_file: str*, *output_bam_file: str*) → None
Correct barcodes in a (potentially unaligned) bamfile, given a whitelist.

Parameters

- **bam_file** (*str*) – BAM format file in same order as the fastq files
- **output_bam_file** (*str*) – BAM format file containing cell, umi, and sample tags.

get_corrected_barcode(*barcode: str*) → str
Return a barcode if it is whitelist, or the corrected version if within edit distance 1

Parameters **barcode** (*str*) – the barcode to return the corrected version of. If the barcode is in the whitelist, the input barcode is returned unchanged.

Returns **corrected_barcode** – corrected version of the barcode

Return type str

Raises `KeyError` – if the passed barcode is not within 1 hamming distance of any whitelist barcode

References

https://en.wikipedia.org/wiki/Hamming_distance

`classmethod single_hamming_errors_from_whitelist(whitelist_file: str)`
Factory method to generate instance of class from a file containing "correct" barcodes.

Parameters `whitelist_file` (*str*) – Text file containing barcode per line.

Returns **errors_to_barcodes_map** – instance of cls, built from whitelist

Return type *ErrorsToCorrectBarcodesMap*

8.1.3 sctools.encodings module

Compressed Barcode Encoding Methods

This module defines several classes to encode DNA sequences in memory-efficient forms, using 2 bits to encode bases of a 4-letter DNA alphabet (ACGT) or 3 bits to encode a 5-letter DNA alphabet that includes the ambiguous call often included by Illumina base calling software (ACGTN). The classes also contain several methods useful for efficient querying and manipulation of the encoded sequence.

Classes

Encoding Encoder base class ThreeBit Three bit DNA encoder / decoder TwoBit Two bit DNA encoder / decoder

`class sctools.encodings.Encoding`
Bases: `object`

`encoding_map`
Class that mimics a Mapping[bytes, str] where bytes must be a single byte encoded character (encoder)

Type *TwoBitEncodingMap*

`decoding_map`
Dictionary that maps integers to bytes human-readable representations (decoder)

Type Mapping[int, bytes]

`bits_per_base`
number of bits used to encode each base

Type int

`encode(bytes_encoded: bytes)`
encode a DNA string in a compressed representation

`decode(integer_encoded: int)`
decode a compressed DNA string into a human readable bytes format

`gc_content(integer_encoded: int)`
calculate the GC content of an encoded DNA string

`hamming_distance(a: int, b: int)`
calculate the hamming distance between two encoded DNA strings

bits_per_base: int = NotImplemented

decode(*integer_encoded: int*) → bytes
Decode a DNA bytes string.

Parameters **integer_encoded** (*bytes*) – Integer encoded DNA string

Returns **decoded** – Bytes decoded DNA sequence

Return type bytes

decoding_map: Mapping[int, AnyStr] = NotImplemented

classmethod encode(*bytes_encoded: bytes*) → int
Encode a DNA bytes string.

Parameters **bytes_encoded** (*bytes*) – bytes DNA string

Returns **encoded** – Encoded DNA sequence

Return type int

encoding_map: Mapping[AnyStr, int] = NotImplemented

gc_content(*integer_encoded: int*) → int
Return the number of G or C nucleotides in *integer_encoded*

Parameters **integer_encoded** (*int*) – Integer encoded DNA string

Returns number of bases in *integer_encoded* input that are G or C.

Return type gc_content, int

static hamming_distance(*a*, *b*) → int
Calculate the hamming distance between two DNA sequences

The hamming distance counts the number of bases that are not the same nucleotide

Parameters

- **a** (*int*) – integer encoded
- **b** (*int*) – integer encoded

Returns **d** – hamming distance between a and b

Return type int

class sctools.encodings.**ThreeBit**(**args*, ***kwargs*)
Bases: *sctools.encodings.Encoding*

Encode a DNA sequence using a 3-bit encoding.

Since no bases are encoded as 0, an empty triplet is interpreted as the end of the encoded string; Three-bit encoding can be used to encode and decode strings without knowledge of their length.

encoding_map
Class that mimics a Mapping[bytes, str] where bytes must be a single byte encoded character (encoder)

Type *TwoBitEncodingMap*

decoding_map
Dictionary that maps integers to bytes human-readable representations (decoder)

Type Mapping[int, bytes]

bits_per_base
number of bits used to encode each base

Type int

encode(*bytes_encoded: bytes*)
encode a DNA string in a compressed representation

decode(*integer_encoded: int*)
decode a compressed DNA string into a human readable bytes format

gc_content(*integer_encoded: int*)
calculate the GC content of an encoded DNA string

hamming_distance(*a: int*, *b: int*)
calculate the hamming distance between two encoded DNA strings

class ThreeBitEncodingMap
Bases: `object`

Dict-like class that maps bytes to 3-bit integer representations

All IUPAC ambiguous codes are treated as "N"

map_ = {65: 2, 67: 1, 71: 3, 78: 6, 84: 4, 97: 2, 99: 1, 103: 3, 110: 6, 116: 4}

bits_per_base: int = 3

classmethod decode(*integer_encoded: int*) → bytes
Decode a DNA bytes string.

Parameters **integer_encoded** (*bytes*) – Integer encoded DNA string

Returns **decoded** – Bytes decoded DNA sequence

Return type bytes

decoding_map: Mapping[int, bytes] = {1: b'C', 2: b'A', 3: b'G', 4: b'T', 6: b'N'}

classmethod encode(*bytes_encoded: bytes*) → int
Encode a DNA bytes string.

Parameters **bytes_encoded** (*bytes*) – bytes DNA string

Returns **encoded** – Encoded DNA sequence

Return type int

encoding_map: *sctools.encodings.ThreeBit.ThreeBitEncodingMap* = <sctools.encodings.ThreeBit.ThreeBitEncodingMap object>

classmethod gc_content(*integer_encoded: int*) → int
Return the number of G or C nucleotides in *integer_encoded*

Parameters **integer_encoded** (*int*) – Integer encoded DNA string

Returns number of bases in *integer_encoded* input that are G or C.

Return type gc_content, int

static hamming_distance(*a: int*, *b: int*) → int
Calculate the hamming distance between two DNA sequences

The hamming distance counts the number of bases that are not the same nucleotide

Parameters

- **a** (*int*) – integer encoded

- **b** (*int*) – integer encoded

Returns **d** – hamming distance between a and b

Return type int

class sctools.encodings.**TwoBit**(*sequence_length: int*)

Bases: *sctools.encodings.Encoding*

Encode a DNA sequence using a 2-bit encoding.

Two-bit encoding uses 0 for an encoded nucleotide. As such, it cannot distinguish between the end of sequence and trailing A nucleotides, and thus decoding these strings requires knowledge of their length. Therefore, it is only appropriate for encoding fixed sequence lengths

In addition, in order to encode in 2-bit, N-nucleotides must be randomized to one of A, C, G, and T.

Parameters **sequence_length** (*int*) – number of nucleotides that are being encoded

encoding_map

Class that mimics a Mapping[bytes, str] where bytes must be a single byte encoded character (encoder)

Type *TwoBitEncodingMap*

decoding_map

Dictionary that maps integers to bytes human-readable representations (decoder)

Type Mapping[int, bytes]

bits_per_base

number of bits used to encode each base

Type int

encode(*bytes_encoded: bytes*)

encode a DNA string in a compressed representation

decode(*integer_encoded: int*)

decode a compressed DNA string into a human readable bytes format

gc_content(*integer_encoded: int*)

calculate the GC content of an encoded DNA string

hamming_distance(*a: int*, *b: int*)

calculate the hamming distance between two encoded DNA strings

class TwoBitEncodingMap

Bases: object

Dict-like class that maps bytes to 2-bit integer representations

Generates random nucleotides for ambiguous nucleotides e.g. N

iupac_ambiguous: Set[int] = {66, 68, 72, 75, 77, 78, 82, 83, 86, 87, 89, 98, 100, 104, 107, 109, 110, 114, 115, 118, 119, 121}

map_ = {65: 0, 67: 1, 71: 3, 84: 2, 97: 0, 99: 1, 103: 3, 116: 2}

bits_per_base: int = 2

decode(*integer_encoded: int*) → bytes

Decode a DNA bytes string.

Parameters **integer_encoded** (*bytes*) – Integer encoded DNA string

Returns **decoded** – Bytes decoded DNA sequence

Return type bytes

decoding_map: Mapping[int, bytes] = {0: b'A', 1: b'C', 2: b'T', 3: b'G'}

classmethod encode(*bytes_encoded: bytes*) → int
Encode a DNA bytes string.

Parameters **bytes_encoded** (*bytes*) – bytes DNA string

Returns **encoded** – Encoded DNA sequence

Return type int

encoding_map: *sctools.encodings.TwoBit.TwoBitEncodingMap* = <sctools.encodings.TwoBit.TwoBitEncodingMap object>

gc_content(*integer_encoded: int*) → int
Return the number of G or C nucleotides in *integer_encoded*

Parameters **integer_encoded** (*int*) – Integer encoded DNA string

Returns number of bases in *integer_encoded* input that are G or C.

Return type gc_content, int

static hamming_distance(*a: int*, *b: int*) → int
Calculate the hamming distance between two DNA sequences

The hamming distance counts the number of bases that are not the same nucleotide

Parameters

- **a** (*int*) – integer encoded
- **b** (*int*) – integer encoded

Returns **d** – hamming distance between a and b

Return type int

8.1.4 sctools.fastq module

Efficient Fastq Iterators and Representations

This module implements classes for representing fastq records, reading and writing them, and extracting parts of fastq sequence for transformation into bam format tags

sctools.**extract_barcode**(*record*, *embedded_barcode*)
extract a barcode, defined by *embedded_barcode* from *record*

sctools.**Classes**()

Record	**Represents fastq records (input as bytes)**
StrRecord	**Represents fastq records (input as str)**
Reader	**Opens and iterates over fastq files**
EmbeddedBarcodeGenerator	**Generates barcodes from a fastq file**
BarcodeGeneratorWithCorrectedCellBarcodes	**Generates (corrected) barcodes from a fastq file**

References

https://en.wikipedia.org/wiki/FASTQ_format

class sctools.fastq.**BarcodeGeneratorWithCorrectedCellBarcodes**(*fastq_files: Union[str, Iterable[str]]*, *embedded_cell_barcode: sctools.fastq.Tag*, *whitelist: str*, *other_embedded_barcodes: Iterable[sctools.fastq.Tag] = ()*, **args*, ***kwargs*)

Bases: *sctools.fastq.Reader*

Generate barcodes from FASTQ file(s) from positions defined by EmbeddedBarcode(s)

Extracted barcode objects are produced in a form that is consumable by pysam's bam and sam set_tag methods. In this class, one EmbeddedBarcode must be defined as an *embedded_cell_barcode*, which is checked against a whitelist and error corrected during generation

Parameters

- **fastq_files** (*str | List, optional*) – FASTQ file or files to be read. (default = sys.stdin)
- **mode** (*{'r', 'rb'}, optional*) – open mode for fastq files. If 'r', return string. If 'rb', return bytes (default = 'r')
- **whitelist** (*str*) – whitelist file containing "correct" cell barcodes for an experiment
- **embedded_cell_barcodes** (*EmbeddedBarcode*) – EmbeddedBarcode containing information about the position and names of cell barcode tags
- **other_embedded_barcodes** (*Iterable[EmbeddedBarcode], optional*) – tag objects defining start and end of the sequence containing the tag, and the tag identifiers for sequence and quality tags (default = None)

extract_cell_barcode(*record:* Record, *cb: str*)

extract_cell_barcode(*record: Tuple[str]*, *cb: sctools.fastq.Tag*)

Extract a cell barcode from a fastq record

Parameters

- **record** (*Tuple[str]*) – fastq record comprised of four strings: name, sequence, name2, and quality
- **cb** (*EmbeddedBarcode*) – defines the position and tag identifier for a call barcode

Returns

- **sequence_tag** (*Tuple[str, str, 'Z']*) – raw sequence tag identifier, sequence, SAM tag type ('Z' implies a string tag)
- **quality_tag** (*Tuple[str, str, 'Z']*) – quality tag identifier, quality, SAM tag type ('Z' implies a string tag)
- **corrected_tag** (*Optional[Tuple[str, str, 'Z']]*) – Whitelist verified sequence tag. Only present if the raw sequence tag is in the whitelist or within 1 hamming distance of one of its barcodes

property filenames: List[str]

select_record_indices(*indices: Set*) → Generator

Iterate over provided indices only, skipping other records.

Parameters **indices** (*Set[int]*) – indices to include in the output

Yields *record, str* – records from file corresponding to indices

property size: int
return the collective size of all files being read in bytes

sctools.fastq.**EmbeddedBarcode**
alias of sctools.fastq.Tag

class sctools.fastq.**EmbeddedBarcodeGenerator**(*fastq_files*, *embedded_barcodes*, **args*, ***kwargs*)
Bases: *sctools.fastq.Reader*

Generate barcodes from a FASTQ file(s) from positions defined by EmbeddedBarcode(s)

Extracted barcode objects are produced in a form that is consumable by pysam's bam and sam set_tag methods.

Parameters

- **embedded_barcodes** (*Iterable[EmbeddedBarcode]*) – tag objects defining start and end of the sequence containing the tag, and the tag identifiers for sequence and quality tags
- **fastq_files** (*str | List, optional*) – FASTQ file or files to be read. (default = sys.stdin)
- **mode** (*{'r', 'rb'}, optional*) – open mode for FASTQ files. If 'r', return string. If 'rb', return bytes (default = 'r')

property filenames: List[str]

select_record_indices(*indices: Set*) → Generator
Iterate over provided indices only, skipping other records.

Parameters **indices** (*Set[int]*) – indices to include in the output

Yields *record, str* – records from file corresponding to indices

property size: int
return the collective size of all files being read in bytes

class sctools.fastq.**Reader**(*files='-'*, *mode='r'*, *header_comment_char=None*)
Bases: *sctools.reader.Reader*

Fastq Reader that defines some special methods for reading and summarizing FASTQ data.

Simple reader class that exposes an __iter__ and __len__ method

Examples

#todo add examples

See also:

sctools.reader.Reader

References

https://en.wikipedia.org/wiki/FASTQ_format

property filenames: List[str]

select_record_indices(*indices: Set*) → Generator
Iterate over provided indices only, skipping other records.

Parameters **indices** (*Set[int]*) – indices to include in the output

Yields *record, str* – records from file corresponding to indices

property size: int
return the collective size of all files being read in bytes

class sctools.fastq.**Record**(*record: Iterable[AnyStr]*)
Bases: object

Fastq Record.

Parameters **record** (*Iterable[bytes]*) – Iterable of 4 bytes strings that comprise a fastq record

name
fastq record name

Type bytes

sequence
fastq nucleotide sequence

Type bytes

name2
second fastq record name field (rarely used)

Type bytes

quality
base call quality for each nucleotide in sequence

Type bytes

average_quality()
The average quality of the fastq record

average_quality() → float
return the average quality of this record

property name: AnyStr

property name2: AnyStr

property quality: AnyStr

property sequence: AnyStr

class sctools.fastq.**StrRecord**(*record: Iterable[AnyStr]*)
Bases: *sctools.fastq.Record*

Fastq Record.

Parameters **record** (*Iterable[str]*) – Iterable of 4 bytes strings that comprise a FASTQ record

name
FASTQ record name

Type str

sequence
FASTQ nucleotide sequence

Type str

name2
second FASTQ record name field (rarely used)

Type str

quality
base call quality for each nucleotide in sequence

Type str

average_quality()
The average quality of the FASTQ record

average_quality() → float
return the average quality of this record

property name: str

property name2: AnyStr

property quality: AnyStr

property sequence: AnyStr

sctools.fastq.**extract_barcode**(*record*, *embedded_barcode*) → Tuple[Tuple[str, str, str], Tuple[str, str, str]]
Extracts barcodes from a FASTQ record at positions defined by an EmbeddedBarcode object.

Parameters

- **record** (*FastqRecord*) – Record to extract from
- **embedded_barcode** (*EmbeddedBarcode*) – Defines the barcode start and end positions and the tag name for the sequence and quality tags

Returns

- **sequence_tag** (*Tuple[str, str, 'Z']*) – sequence tag identifier, sequence, SAM tag type ('Z' implies a string tag)
- **quality_tag** (*Tuple[str, str, 'Z']*) – quality tag identifier, quality, SAM tag type ('Z' implies a string tag)

8.1.5 sctools.gtf module

GTF Records and Iterators

This module defines a GTF record class and a Reader class to iterate over GTF-format files

Classes

Record Data class that exposes GTF record fields by name Reader GTF file reader that yields GTF Records

References

https://useast.ensembl.org/info/website/upload/gff.html

class sctools.gtf.**GTFRecord**(*record: str*)

Bases: object

Data class for storing and interacting with GTF records

Subclassed to produce exon, transcript, and gene-specific record types. A GTF record has 8 fixed fields which are followed by optional fields separated by ; , which are stored by this class in the attributes field and accessible by get_attribute. Fixed fields are accessible by name.

Parameters **record** (*str*) – an unparsed GTF record

seqname

The name of the sequence (often chromosome) this record is found on.

Type str

chromosome

Synonym for seqname.

Type str

source

The group responsible for generating this annotation.

Type str

feature

The type of record (e.g. gene, exon, ...).

Type str

start

The start position of this feature relative to the beginning of seqname.

Type str

end

The end position of this feature relative to the beginning of seqname....

Type str

score

The annotation score. Rarely used.

Type str

strand

The strand of seqname that this annotation is found on

Type {'+', '-'}

frame

'0' indicates that the first base of the feature is the first base of a codon, '1' that the second base is the first base of a codon, and so on

Type {'0', '1', '2'}

size
the number of nucleotides spanned by this feature

Type int

get_attribute(*key: str*)
attempt to retrieve a variable field with name equal to *key*

set_attribute(*key: str*, *value: str*)
set variable field *key* equal to *value*. Overwrites *key* if already present.

property chromosome: str

property end: int

property feature: str

property frame: str

get_attribute(*key*) → str
access an item from the attribute field of a GTF file.

Parameters **key** (*str*) – Item to retrieve

Returns **value** – Contents of variable attribute *key*

Return type str

Raises **KeyError** – if there is no variable attribute *key* associated with this record

property score: str

property seqname: str

set_attribute(*key*, *value*) → None
Set variable attribute *key* equal to *value*

If attribute *key* is already set for this record, its contents are overwritten by *value*

Parameters

- **key** (*str*) – attribute name
- **value** (*str*) – attribute content

property size: int

property source: str

property start: int

property strand: str

class sctools.gtf.**Reader**(*files='-'*, *mode='r'*, *header_comment_char='#'*)
Bases: *sctools.reader.Reader*

GTF file iterator

Parameters

- **files** (*Union[str, List], optional*) – File(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – Open mode. If 'r', read strings. If 'rb', read bytes (default = 'r').
- **header_comment_char** (*str, optional*) – lines beginning with this character are skipped (default = '#')

filter(*retain_types: Iterable[str]*)
Iterate over a GTF file, only yielding records in *retain_types*.

__iter__()
iterate over GTF records in file, yielding *Record* objects

See also:

sctools.reader.Reader

property filenames: List[str]

filter(*retain_types: Iterable[str]*) → Generator
Iterate over a GTF file, returning only record whose feature type is in retain_types.

Features are stored in GTF field 2.

Parameters **retain_types** (*Iterable[str]*) – Record feature types to retain.

Yields **gtf_record** (*Record*) – gtf *Record* object

select_record_indices(*indices: Set*) → Generator
Iterate over provided indices only, skipping other records.

Parameters **indices** (*Set[int]*) – indices to include in the output

Yields *record, str* – records from file corresponding to indices

property size: int
return the collective size of all files being read in bytes

sctools.gtf.**extract_extended_gene_names**(*files: Union[str, List[str]] = '-'*, *mode: str = 'r'*, *header_comment_char: str = '#'*) → Dict[str, List[tuple]]
Extract extended gene names from GTF file(s) and returns a map from gene names to their corresponding occurrence locations the given file(s).

Parameters

- **files** (*Union[str, List], optional*) – File(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – Open mode. If 'r', read strings. If 'rb', read bytes (default = 'r').
- **header_comment_char** (*str, optional*) – lines beginning with this character are skipped (default = '#')

Returns A dictionary of chromosome names mapping to a List of tuples, each containing a range as the the first element and a gene name as the second. Dict[str, List(Tuple((start,end), gene)))

Return type Dict[str, List[tuple]]

sctools.gtf.**extract_gene_exons**(*files: Union[str, List[str]] = '-'*, *mode: str = 'r'*, *header_comment_char: str = '#'*) → Dict[str, List[tuple]]
Extract extended gene names from GTF file(s) and returns a map from gene names to the the list of exons in the ascending order of the start positions file(s).

Parameters

- **files** (*Union[str, List], optional*) – File(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – Open mode. If 'r', read strings. If 'rb', read bytes (default = 'r').

- **header_comment_char** (*str, optional*) – lines beginning with this character are skipped (default = '#')

Returns A dictionary of chromosome names mapping to a List of tuples, each containing a the exons in the ascending order of the start positions. Dict[str, List(Tuple((start,end), gene)))

Return type Dict[str, List[tuple]]

sctools.gtf.**extract_gene_names**(*files: Union[str, List[str]] = '-', mode: str = 'r', header_comment_char: str = '#'*) → Dict[str, int]

Extract gene names from GTF file(s) and returns a map from gene names to their corresponding occurrence orders in the given file(s).

Parameters

- **files** (*Union[str, List], optional*) – File(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – Open mode. If 'r', read strings. If 'rb', read bytes (default = 'r').
- **header_comment_char** (*str, optional*) – lines beginning with this character are skipped (default = '#')

Returns A map from gene names to their linear index

Return type Dict[str, int]

sctools.gtf.**get_mitochondrial_gene_names**(*files: Union[str, List[str]] = '-', mode: str = 'r', header_comment_char: str = '#'*) → Set[str]

Extract mitocholdrial gene names from GTF file(s) and returns a set of mitochondrial gene id occurrence in the given file(s).

Parameters

- **files** (*Union[str, List], optional*) – File(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – Open mode. If 'r', read strings. If 'rb', read bytes (default = 'r').
- **header_comment_char** (*str, optional*) – lines beginning with this character are skipped (default = '#')

Returns A set of the mitochondrial gene ids

Return type Set(str)

8.1.6 sctools.platform module

Command Line Interface for SC Tools:

This module defines the command line interface for SC Tools. Tools are separated into those that are specific to particular chemistries (e.g. Smart-seq 2) or experimental platforms (e.g. 10x Genomics v2) and those that are general across any sequencing experiment.

Currently, only general modules and those used for 10x v2 are implemented

Classes

GenericPlatform Class containing all general command line utilities TenXV2 Class containing 10x v2 specific command line utilities

class sctools.platform.**BarcodePlatform**

Bases: *sctools.platform.GenericPlatform*

Command Line Interface for extracting and attaching barcodes with specified positions generalizing TenXV2 attach barcodes

Sample, cell and/or molecule barcodes can be extracted and attached to an unmapped bam when the corresponding barcode's start position and and length are provided. The sample barcode is extracted from the index i7 fastq file and the cell and molecule barcode are extracted from the r1 fastq file

This class defines several methods that are created as CLI tools when sctools is installed (see setup.py)

cell_barcode

A data class that defines the start and end position of the cell barcode and the tags to assign the sequence and quality of the cell barcode

Type fastq.EmbeddedBarcode

molecule_barcode

A data class that defines the start and end position of the molecule barcode and the tags to assign the sequence and quality of the molecule barcode

Type fastq.EmbeddedBarcode

sample_barcode

A data class that defines the start and end position of the sample barcode and the tags to assign the sequence and quality of the sample barcode

Type fastq.EmbeddedBarcode

attach_barcodes()

Attach barcodes from the forward (r1) and optionally index (i1) fastq files to the reverse (r2) bam file

classmethod attach_barcodes(*args=None*)

Command line entrypoint for attaching barcodes to a bamfile.

Parameters **args** (*Iterable[str], optional*) – arguments list, The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod bam_to_count_matrix(*args: Optional[Iterable[str]] = None*) → int

Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod calculate_cell_metrics(*args: Optional[Iterable[str]] = None*) → int

Command line entrypoint for calculating cell metrics from a sorted bamfile.

Writes metrics to .csv

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod calculate_gene_metrics(*args: Optional[Iterable[str]] = None*) → int

Command line entrypoint for calculating gene metrics from a sorted bamfile.

Writes metrics to .csv

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

cell_barcode = None

classmethod get_tags(*raw_tags: Optional[Sequence[str]]*) → Iterable[str]

classmethod group_qc_outputs(*args: Optional[Iterable[str]] = None*) → int

Commandline entrypoint for parsing picard metrics files, hisat2 and rsem statistics log files. :param args: file_names: array of files

output_name: prefix of output file name. metrics_type: Picard, PicardTable, HISAT2, RSEM and Core.

Returns **return** – return if the program completes successfully.

Return type 0

classmethod merge_cell_metrics(*args: Optional[Iterable[str]] = None*) → int

Command line entrypoint for merging multiple cell metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_count_matrices(*args: Optional[Iterable[str]] = None*) → int

Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_gene_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for merging multiple gene metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

molecule_barcode = None

sample_barcode = None

classmethod split_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for splitting a bamfile into subfiles of equal size.

prints filenames of chunks to stdout

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod tag_sort_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for sorting a bam file by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod verify_bam_sort(*args: Optional[Iterable] = None*) → int
Command line entrypoint for verifying bam is properly sorted by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

class sctools.platform.**GenericPlatform**
Bases: object

Platform-agnostic command line functions available in SC Tools.

tag_sort_bam(): sort a bam file by zero or more tags and then by queryname

verify_bam_sort(): verifies whether bam file is correctly sorted by given list of zero or more tags, then queryname

split_bam() split a bam file into subfiles of equal size

calculate_gene_metrics() calculate information about genes captured by a sequencing experiment

calculate_cell_metrics() calculate information about cells captured by a sequencing experiment

merge_gene_metrics() merge multiple gene metrics files into a single output

merge_cell_metrics() merge multiple cell metrics files into a single output

bam_to_count() construct a compressed sparse row count file from a tagged, aligned bam file

merge_count_matrices() merge multiple csr-format count matrices into a single csr matrix

group_qc_outputs() aggregate Picard, HISAT2 and RSME QC statisitics

`classmethod bam_to_count_matrix`(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters `args` (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

`classmethod calculate_cell_metrics`(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for calculating cell metrics from a sorted bamfile.

Writes metrics to .csv

Parameters `args` (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

`classmethod calculate_gene_metrics`(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for calculating gene metrics from a sorted bamfile.

Writes metrics to .csv

Parameters `args` (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

`classmethod get_tags`(*raw_tags: Optional[Sequence[str]]*) → Iterable[str]

`classmethod group_qc_outputs`(*args: Optional[Iterable[str]] = None*) → int
Commandline entrypoint for parsing picard metrics files, hisat2 and rsem statistics log files. :param args: file_names: array of files

output_name: prefix of output file name. metrics_type: Picard, PicardTable, HISAT2, RSEM and Core.

Returns **return** – return if the program completes successfully.

Return type 0

classmethod merge_cell_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for merging multiple cell metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_count_matrices(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_gene_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for merging multiple gene metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod split_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for splitting a bamfile into subfiles of equal size.

prints filenames of chunks to stdout

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod tag_sort_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for sorting a bam file by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod verify_bam_sort(*args: Optional[Iterable] = None*) → int
Command line entrypoint for verifying bam is properly sorted by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

class sctools.platform.**TenXV2**
Bases: *sctools.platform.GenericPlatform*

Command Line Interface for 10x Genomics v2 RNA-sequencing programs

This class defines several methods that are created as CLI tools when sctools is installed (see setup.py)

cell_barcode
A data class that defines the start and end position of the cell barcode and the tags to assign the sequence and quality of the cell barcode

Type fastq.EmbeddedBarcode

molecule_barcode
A data class that defines the start and end position of the molecule barcode and the tags to assign the sequence and quality of the molecule barcode

Type fastq.EmbeddedBarcode

sample_barcode
A data class that defines the start and end position of the sample barcode and the tags to assign the sequence and quality of the sample barcode

Type fastq.EmbeddedBarcode

attach_barcodes()
Attach barcodes from the forward (r1) and optionally index (i1) fastq files to the reverse (r2) bam file

classmethod attach_barcodes(*args=None*)
Command line entrypoint for attaching barcodes to a bamfile.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod bam_to_count_matrix(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod calculate_cell_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for calculating cell metrics from a sorted bamfile.

Writes metrics to .csv

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod calculate_gene_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for calculating gene metrics from a sorted bamfile.

Writes metrics to .csv

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

cell_barcode = Tag(start=0, end=16, sequence_tag='CR', quality_tag='CY')

classmethod get_tags(*raw_tags: Optional[Sequence[str]]*) → Iterable[str]

classmethod group_qc_outputs(*args: Optional[Iterable[str]] = None*) → int
Commandline entrypoint for parsing picard metrics files, hisat2 and rsem statistics log files. :param args: file_names: array of files

output_name: prefix of output file name. metrics_type: Picard, PicardTable, HISAT2, RSEM and Core.

Returns **return** – return if the program completes successfully.

Return type 0

classmethod merge_cell_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for merging multiple cell metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_count_matrices(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for constructing a count matrix from a tagged bam file.

Constructs a count matrix from an aligned bam file sorted by cell barcode, molecule barcode, and gene id.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod merge_gene_metrics(*args: Optional[Iterable[str]] = None*) → int
Command line entrypoint for merging multiple gene metrics files.

Merges multiple metrics inputs into a single metrics file that matches the shape and order of the generated count matrix.

Parameters **args** (*Iterable[str], optional*) – Arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

molecule_barcode = Tag(start=16, end=26, sequence_tag='UR', quality_tag='UY')

sample_barcode = Tag(start=0, end=8, sequence_tag='SR', quality_tag='SY')

classmethod split_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for splitting a bamfile into subfiles of equal size.

prints filenames of chunks to stdout

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod tag_sort_bam(*args: Optional[Iterable] = None*) → int
Command line entrypoint for sorting a bam file by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

classmethod verify_bam_sort(*args: Optional[Iterable] = None*) → int
Command line entrypoint for verifying bam is properly sorted by zero or more tags, followed by queryname.

Parameters **args** (*Iterable[str], optional*) – arguments list, for testing (see test/test_entrypoints.py for example). The default value of None, when passed to *parser.parse_args* causes the parser to read *sys.argv*

Returns **return_call** – return call if the program completes successfully

Return type 0

8.1.7 sctools.reader module

Sequence File Iterators

This module defines a general iterator and some helper functions for iterating over files that contain sequencing data

sctools.**infer_open**(*file_: str*, *mode: str*)
helper function that determines the compression type of a file without relying on its extension

sctools.**zip_readers**(**readers*, *indices=None*)
helper function that iterates over one or more readers, optionally extracting only the records that correspond to indices

sctools.**Classes**()

Reader **Basic reader that loops over one or more input files.**

See also:

sctools.gtf.Reader, *sctools.fastq.Reader*

class sctools.reader.**Reader**(*files='-'*, *mode='r'*, *header_comment_char=None*)
Bases: object

Basic reader object that seamlessly loops over multiple input files.

Is subclassed to create readers for specific file types (e.g. fastq, gtf, etc.)

Parameters

- **files** (*Union[str, List], optional*) – The file(s) to read. If '-', read sys.stdin (default = '-')
- **mode** (*{'r', 'rb'}, optional*) – The open mode for files. If 'r', yield string data, if 'rb', yield bytes data (default = 'r').
- **header_comment_char** (*str, optional*) – If not None, skip lines beginning with this character (default = None).

property filenames: List[str]

select_record_indices(*indices: Set*) → Generator
Iterate over provided indices only, skipping other records.

Parameters **indices** (*Set[int]*) – indices to include in the output

Yields *record, str* – records from file corresponding to indices

property size: int
return the collective size of all files being read in bytes

sctools.reader.**infer_open**(*file_: str*, *mode: str*) → Callable
Helper function to infer the correct compression type of an input file

Identifies files that are .gz or .bz2 compressed without requiring file extensions

Parameters

- **file** (*str*) – the file to open
- **mode** (*{'r', 'rb'}*) – the mode to open the file in. 'r' returns strings, 'rb' returns bytes

Returns **open_function** – the correct open function for the file's compression with mode pre-set through functools partial

Return type Callable

sctools.reader.**zip_readers**(**readers*, *indices=None*) → Generator
Zip together multiple reader objects, yielding records simultaneously.

If indices is passed, only return lines in file that correspond to indices

Parameters

- ***readers** (*List[Reader]*) – Reader objects to simultaneously iterate over
- **indices** (*Set[int], optional*) – indices to include in the output

Yields **records** (*Tuple[str]*) – one record per reader passed

8.1.8 sctools.stats module

Statistics Functions for Sequence Data Analysis

This module implements statistical modules for sequence analysis

sctools.**base4_entropy**(*x: np.array*, *axis: int = 1*)
calculate the entropy of a 4 x sequence length base frequency matrix

sctools.**Classes**()

OnlineGaussianSuficientStatistic Empirical (online) calculation of mean and variance

class sctools.stats.**OnlineGaussianSufficientStatistic**
Bases: object

Implementation of Welford's online mean and variance algorithm

update(*new_value: float*)
incorporate new_value into the online estimate of mean and variance

mean()
return the mean value

calculate_variance()
calculate and return the variance

mean_and_variance()
return both mean and variance

calculate_variance()
calculate and return the variance

property mean: float
return the mean value

mean_and_variance() → Tuple[float, float]
calculate and return the mean and variance

update(*new_value: float*) → None

sctools.stats.**base4_entropy**(*x*, *axis=1*)
Calculate entropy in base four of a data matrix x

Useful for measuring DNA entropy (with 4 nucleotides) as the output is restricted to [0, 1]

Parameters

- **x** (*np.ndarray*) – array of dimension one or more containing numeric types
- **axis** (*int, optional*) – axis to calculate entropy across. Values in this axis are treated as observation frequencies

Returns **entropy** – array of input dimension - 1 containin entropy values bounded in [0, 1]

Return type np.ndarray

CHAPTER

NINE

SCTOOLS.METRICS PACKAGE

9.1 Submodules

9.1.1 sctools.metrics.aggregator module

Sequence Metric Aggregators

This module provides classes useful for aggregating metric information for individual cells or genes. These classes consume BAM files that have been pre-sorted such that all sequencing reads that correspond to the molecules of a cell (CellMetrics) or the molecules of a gene (GeneMetrics) are yielded sequentially.

Classes

—

Notes

This module can be rewritten with dataclass when python 3.7 stabilizes, see https://www.python.org/dev/peps/pep-0557/

See also:

sctools.metrics.gatherer, *sctools.metrics.merge*, *sctools.metrics.writer*

class sctools.metrics.aggregator.**CellMetrics**

Bases: *sctools.metrics.aggregator.MetricAggregator*

Cell Metric Aggregator

Aggregator that captures metric information about a cell by parsing all of the molecules in an experiment that were annotated with a specific cell barcode, as recorded in the CB tag.

perfect_cell_barcodes

The number of reads whose cell barcodes contain no errors (tag CB == CR)

Type int

reads_mapped_intergenic

The number of reads mapped to an intergenic region for this cell

Type int

reads_mapped_too_many_loci
: The number of reads that were mapped to too many loci across the genome and as a consequence, are reported unmapped by the aligner

  Type int

cell_barcode_fraction_bases_above_30_variance
: The variance of the fraction of Illumina base calls for the cell barcode sequence that are greater than 30, across molecules

  Type float

cell_barcode_fraction_bases_above_30_mean
: The average fraction of Illumina base calls for the cell barcode sequence that are greater than 30, across molecules

  Type float

n_genes
: The number of genes detected by this cell

  Type int

genes_detected_multiple_observations
: The number of genes that are observed by more than one read in this cell

  Type int

n_mitochondrial_genes
: The number of mitochondrial genes detected by this cell

  Type int

n_mitochondrial_molecules
: The number of molecules from mitochondrial genes detected for this cell

  Type int

pct_mitochondrial_molecules
: The percentage of molecules from mitochondrial genes detected for this cell

  Type int

Metric Aggregator Base Class

The ``MetricAggregator`` class defines a set of metrics that can be extracted from an

aligned bam file.
It defines all the metrics that are general across genes and cells. This

class is subclassed by ``GeneMetrics`` and ``CellMetrics``,
which define data-specific metrics

in the ``parse_extra_fields`` method.
An instance of ``GeneMetrics`` or ``CellMetrics`` is

instantiated for each gene or molecule in a bam file, respectively.

n_reads
: The number of reads associated with this entity

  Type int

noise_reads
: Number of reads that are categorized by 10x genomics cellranger as “noise”. Refers to long polymers, or reads with high numbers of N (ambiguous) nucleotides

Type int, NotImplemented

perfect_molecule_barcodes
The number of reads with molecule barcodes that have no errors (cell barcode tag == raw barcode tag)

Type int

reads_mapped_exonic
The number of reads for this entity that are mapped to exons

Type int

reads_mapped_intronic
The number of reads for this entity that are mapped to introns

Type int

reads_mapped_utr
The number of reads for this entity that are mapped to 3' untranslated regions (UTRs)

Type int

reads_mapped_uniquely
The number of reads mapped to a single unambiguous location in the genome

Type int

reads_mapped_multiple
The number of reads mapped to multiple genomic positions with equal confidence # todo make sure equal confidence is accurate

Type int

duplicate_reads
The number of reads that are duplicates (see README.md for defition of a duplicate)

Type int

spliced_reads
The number of reads that overlap splicing junctions

Type int

antisense_reads
The number of reads that are mapped to the antisense strand instead of the transcribed strand

Type int

molecule_barcode_fraction_bases_above_30_mean
The average fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

Type float

molecule_barcode_fraction_bases_above_30_variance
The variance in the fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

Type float

genomic_reads_fraction_bases_quality_above_30_mean
The average fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

Type float

genomic_reads_fraction_bases_quality_above_30_variance
: The variance in the fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

  Type float

genomic_read_quality_mean
: Average quality of Illumina base calls in the genomic reads corresponding to this entity

  Type float

genomic_read_quality_variance
: Variance in quality of Illumina base calls in the genomic reads corresponding to this entity

  Type float

n_molecules
: Number of molecules corresponding to this entity. See README.md for the definition of a Molecule

  Type float

n_fragments
: Number of fragments corresponding to this entity. See README.md for the definition of a Fragment

  Type float

reads_per_molecule
: The average number of reads associated with each molecule in this entity

  Type float

reads_per_fragment
: The average number of reads associated with each fragment in this entity

  Type float

fragments_per_molecule
: The average number of fragments associated with each molecule in this entity

  Type float

fragments_with_single_read_evidence
: The number of fragments associated with this entity that are observed by only one read

  Type int

molecules_with_single_read_evidence
: The number of molecules associated with this entity that are observed by only one read

  Type int

parse_extra_fields(tags, record), NotImplemented
: Abstract method that must be implemented by subclasses. Called by `parse_molecule()` to gather information for subclass-specific metrics

parse_molecule(*tags*, *record*)
: Extract information from a set of sequencing reads that correspond to a molecule and store the data in the MetricAggregator class.

finalize()
: Some metrics cannot be calculated until all the information for an entity has been aggregated, for example, the number of *fragments_per_molecule*. Finalize calculates all such higher-order metrics

Examples

todo implement me

See also:

GeneMetrics

extra_docs = '\n Examples\n --------\n # todo implement me\n\n See Also\n --------\n GeneMetrics\n\n '

finalize(*mitochondrial_genes={}*)
Calculate metrics that require information from all molecules of an entity

`finalize()` replaces attributes in-place that were initialized by the constructor as `None` with a value calculated across all molecule data that has been aggregated.

parse_extra_fields(*tags: Sequence[str]*, *record: pysam.libcalignedsegment.AlignedSegment*) → None
Parses a record to extract gene-specific information

Gene-specific metric data is stored in-place in the MetricAggregator

Parameters

- **tags** (*Sequence[str]*) – The GE, UB and CB tags that define this molecule
- **record** (*pysam.AlignedSegment*) – SAM record to be parsed

parse_molecule(*tags: Sequence[str]*, *records: Iterable[pysam.libcalignedsegment.AlignedSegment]*) → None
Parse information from all records of a molecule.

The parsed information is stored in the MetricAggregator in-place.

Parameters

- **tags** (*Sequence[str]*) – all the tags that define this molecule. one of {[CB, GE, UB], [GE, CB, UB]}
- **records** (*Iterable[pysam.AlignedSegment]*) – the sam records associated with the molecule

class sctools.metrics.aggregator.**GeneMetrics**
Bases: *sctools.metrics.aggregator.MetricAggregator*

Gene Metric Aggregator

Aggregator that captures metric information about a gene by parsing all of the molecules in an experiment that were annotated with a specific gene ID, as recorded in the `GE` tag.

number_cells_detected_multiple
The number of cells which observe more than one read of this gene

Type int

number_cells_expressing
The number of cells that detect this gene

Type int

Metric Aggregator Base Class

The ``MetricAggregator`` class defines a set of metrics that can be extracted from an aligned bam file.
It defines all the metrics that are general across genes and cells. This

class is subclassed by ``GeneMetrics`` and ``CellMetrics``,
which define data-specific metrics

in the ``parse_extra_fields`` method.
An instance of ``GeneMetrics`` or ``CellMetrics`` is

instantiated for each gene or molecule in a bam file, respectively.

n_reads
: The number of reads associated with this entity

  Type int

noise_reads
: Number of reads that are categorized by 10x genomics cellranger as "noise". Refers to long polymers, or reads with high numbers of N (ambiguous) nucleotides

  Type int, NotImplemented

perfect_molecule_barcodes
: The number of reads with molecule barcodes that have no errors (cell barcode tag == raw barcode tag)

  Type int

reads_mapped_exonic
: The number of reads for this entity that are mapped to exons

  Type int

reads_mapped_intronic
: The number of reads for this entity that are mapped to introns

  Type int

reads_mapped_utr
: The number of reads for this entity that are mapped to 3' untranslated regions (UTRs)

  Type int

reads_mapped_uniquely
: The number of reads mapped to a single unambiguous location in the genome

  Type int

reads_mapped_multiple
: The number of reads mapped to multiple genomic positions with equal confidence # todo make sure equal confidence is accurate

  Type int

duplicate_reads
: The number of reads that are duplicates (see README.md for defition of a duplicate)

  Type int

spliced_reads
: The number of reads that overlap splicing junctions

  Type int

antisense_reads
: The number of reads that are mapped to the antisense strand instead of the transcribed strand

  Type int

molecule_barcode_fraction_bases_above_30_mean
: The average fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

  Type float

molecule_barcode_fraction_bases_above_30_variance
: The variance in the fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

  Type float

genomic_reads_fraction_bases_quality_above_30_mean
: The average fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

  Type float

genomic_reads_fraction_bases_quality_above_30_variance
: The variance in the fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

  Type float

genomic_read_quality_mean
: Average quality of Illumina base calls in the genomic reads corresponding to this entity

  Type float

genomic_read_quality_variance
: Variance in quality of Illumina base calls in the genomic reads corresponding to this entity

  Type float

n_molecules
: Number of molecules corresponding to this entity. See README.md for the definition of a Molecule

  Type float

n_fragments
: Number of fragments corresponding to this entity. See README.md for the definition of a Fragment

  Type float

reads_per_molecule
: The average number of reads associated with each molecule in this entity

  Type float

reads_per_fragment
: The average number of reads associated with each fragment in this entity

  Type float

fragments_per_molecule
: The average number of fragments associated with each molecule in this entity

  Type float

fragments_with_single_read_evidence
: The number of fragments associated with this entity that are observed by only one read

  Type int

molecules_with_single_read_evidence
: The number of molecules associated with this entity that are observed by only one read

Type int

parse_extra_fields(tags, record), NotImplemented
Abstract method that must be implemented by subclasses. Called by `parse_molecule()` to gather information for subclass-specific metrics

parse_molecule(*tags*, *record*)
Extract information from a set of sequencing reads that correspond to a molecule and store the data in the MetricAggregator class.

finalize()
Some metrics cannot be calculated until all the information for an entity has been aggregated, for example, the number of *fragments_per_molecule*. Finalize calculates all such higher-order metrics

Examples

todo implement me

See also:

CellMetrics

extra_docs = '\n Examples\n --------\n # todo implement me\n\n See Also\n --------\n CellMetrics\n\n '

finalize()
Calculate metrics that require information from all molecules of an entity

`finalize()` replaces attributes in-place that were initialized by the constructor as `None` with a value calculated across all molecule data that has been aggregated.

parse_extra_fields(*tags: Sequence[str]*, *record: pysam.libcalignedsegment.AlignedSegment*) → None
Parses a record to extract cell-specific information

Cell-specific metric data is stored in-place in the MetricAggregator

Parameters

- **tags** (*Sequence[str]*) – The CB, UB and GE tags that define this molecule
- **record** (*pysam.AlignedSegment*) – SAM record to be parsed

parse_molecule(*tags: Sequence[str]*, *records: Iterable[pysam.libcalignedsegment.AlignedSegment]*) → None
Parse information from all records of a molecule.

The parsed information is stored in the MetricAggregator in-place.

Parameters

- **tags** (*Sequence[str]*) – all the tags that define this molecule. one of {[CB, GE, UB], [GE, CB, UB]}
- **records** (*Iterable[pysam.AlignedSegment]*) – the sam records associated with the molecule

class sctools.metrics.aggregator.**MetricAggregator**
Bases: `object`

Metric Aggregator Base Class

The `MetricAggregator` class defines a set of metrics that can be extracted from an aligned bam file. It defines all the metrics that are general across genes and cells. This class is subclassed by `GeneMetrics` and `CellMetrics`,

which define data-specific metrics in the `parse_extra_fields` method. An instance of `GeneMetrics` or `CellMetrics` is instantiated for each gene or molecule in a bam file, respectively.

n_reads
The number of reads associated with this entity

Type int

noise_reads
Number of reads that are categorized by 10x genomics cellranger as "noise". Refers to long polymers, or reads with high numbers of N (ambiguous) nucleotides

Type int, NotImplemented

perfect_molecule_barcodes
The number of reads with molecule barcodes that have no errors (cell barcode tag == raw barcode tag)

Type int

reads_mapped_exonic
The number of reads for this entity that are mapped to exons

Type int

reads_mapped_intronic
The number of reads for this entity that are mapped to introns

Type int

reads_mapped_utr
The number of reads for this entity that are mapped to 3' untranslated regions (UTRs)

Type int

reads_mapped_uniquely
The number of reads mapped to a single unambiguous location in the genome

Type int

reads_mapped_multiple
The number of reads mapped to multiple genomic positions with equal confidence # todo make sure equal confidence is accurate

Type int

duplicate_reads
The number of reads that are duplicates (see README.md for defition of a duplicate)

Type int

spliced_reads
The number of reads that overlap splicing junctions

Type int

antisense_reads
The number of reads that are mapped to the antisense strand instead of the transcribed strand

Type int

molecule_barcode_fraction_bases_above_30_mean
The average fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

Type float

molecule_barcode_fraction_bases_above_30_variance
The variance in the fraction of bases in molecule barcodes that receive quality scores greater than 30 across the reads of this entity

Type float

genomic_reads_fraction_bases_quality_above_30_mean
The average fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

Type float

genomic_reads_fraction_bases_quality_above_30_variance
The variance in the fraction of bases in the genomic read that receive quality scores greater than 30 across the reads of this entity (included for 10x cell ranger count comparison)

Type float

genomic_read_quality_mean
Average quality of Illumina base calls in the genomic reads corresponding to this entity

Type float

genomic_read_quality_variance
Variance in quality of Illumina base calls in the genomic reads corresponding to this entity

Type float

n_molecules
Number of molecules corresponding to this entity. See README.md for the definition of a Molecule

Type float

n_fragments
Number of fragments corresponding to this entity. See README.md for the definition of a Fragment

Type float

reads_per_molecule
The average number of reads associated with each molecule in this entity

Type float

reads_per_fragment
The average number of reads associated with each fragment in this entity

Type float

fragments_per_molecule
The average number of fragments associated with each molecule in this entity

Type float

fragments_with_single_read_evidence
The number of fragments associated with this entity that are observed by only one read

Type int

molecules_with_single_read_evidence
The number of molecules associated with this entity that are observed by only one read

Type int

parse_extra_fields(tags, record), NotImplemented
Abstract method that must be implemented by subclasses. Called by `parse_molecule()` to gather information for subclass-specific metrics

parse_molecule(*tags*, *record*)
Extract information from a set of sequencing reads that correspond to a molecule and store the data in the MetricAggregator class.

finalize()
Some metrics cannot be calculated until all the information for an entity has been aggregated, for example, the number of *fragments_per_molecule*. Finalize calculates all such higher-order metrics

finalize() → None
Calculate metrics that require information from all molecules of an entity

`finalize()` replaces attributes in-place that were initialized by the constructor as `None` with a value calculated across all molecule data that has been aggregated.

parse_extra_fields(*tags: Sequence[str]*, *record: pysam.libcalignedsegment.AlignedSegment*) → None
Defined by subclasses to extract class-specific information from molecules

parse_molecule(*tags: Sequence[str]*, *records: Iterable[pysam.libcalignedsegment.AlignedSegment]*) → None
Parse information from all records of a molecule.

The parsed information is stored in the MetricAggregator in-place.

Parameters

- **tags** (*Sequence[str]*) – all the tags that define this molecule. one of {[CB, GE, UB], [GE, CB, UB]}
- **records** (*Iterable[pysam.AlignedSegment]*) – the sam records associated with the molecule

9.1.2 sctools.metrics.gatherer module

Sequence Metric Gatherers

..currentmodule:: sctools.metrics

This module defines classes to gather metrics across the cells or genes of an experiment and write them to gzip-compressed csv files

Classes

MetricGatherer(bam_file, output_stem, ...)	Gathers Metrics from an experiment
GatherCellMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers
GatherGeneMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers

sctools.metrics.gatherer.MetricGatherer

class sctools.metrics.gatherer.**MetricGatherer**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Gathers Metrics from an experiment

Because molecules tend to have relatively small numbers of reads, the memory footprint of this method is typically small (tens of megabytes).

Parameters

- **bam_file** (*str*) – the bam file containing the reads that metrics should be calculated from. Can be a chunk of cells or an entire experiment
- **output_stem** (*str*) – the file stem for the gzipped csv output

extract_metrics()
extracts metrics from `bam_file` and writes them to output_stem.csv.gz

__init__(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Methods

__init__(bam_file, output_stem[, ...])	

extract_metrics([mode])	extract metrics from the provided bam file and write the results to csv.

Attributes

bam_file	the bam file that metrics are generated from

sctools.metrics.gatherer.GatherCellMetrics

class sctools.metrics.gatherer.**GatherCellMetrics**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

..currentmodule:: sctools.metrics

This module defines classes to gather metrics across the cells or genes of an experiment and write them to gzip-compressed csv files

MetricGatherer(bam_file, output_stem, ...)	Gathers Metrics from an experiment
GatherCellMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers
GatherGeneMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers

See also:

sctools.metrics.aggregator, *sctools.metrics.merge*, *sctools.metrics.writer*

`bam_file` must be sorted by gene (`GE`), molecule (`UB`), and cell (`CB`), where gene varies fastest.

```
>>> from sctools.metrics.gatherer import GatherCellMetrics
>>> import os, tempfile
```

```
>>> # example data
>>> bam_file = os.path.abspath(__file__) + '../test/data/test.bam'
>>> temp_dir = tempfile.mkdtemp()
>>> g = GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test',␣
↪compress=True)
>>> g.extract_metrics()
```

GatherGeneMetrics

__init__(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Methods

__init__(bam_file, output_stem[, ...])	
extract_metrics([mode])	Extract cell metrics from self.bam_file

Attributes

bam_file	the bam file that metrics are generated from
extra_docs	

sctools.metrics.gatherer.GatherGeneMetrics

class sctools.metrics.gatherer.**GatherGeneMetrics**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

..currentmodule:: sctools.metrics

This module defines classes to gather metrics across the cells or genes of an experiment and write them to gzip-compressed csv files

MetricGatherer(bam_file, output_stem, ...)	Gathers Metrics from an experiment
GatherCellMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers
GatherGeneMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers

See also:

sctools.metrics.aggregator, *sctools.metrics.merge*, *sctools.metrics.writer*

`bam_file` must be sorted by molecule (`UB`), cell (`CB`), and gene (`GE`), where molecule varies fastest.

```
>>> from sctools.metrics.gatherer import GatherCellMetrics
>>> import os, tempfile
```

```
>>> # example data
>>> bam_file = os.path.abspath(__file__) + '../test/data/test.bam'
>>> temp_dir = tempfile.mkdtemp()
>>> g = GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test',␣
↪compress=True)
>>> g.extract_metrics()
```

GatherGeneMetrics

__init__(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Methods

__init__(bam_file, output_stem[, ...])	

extract_metrics([mode])	Extract gene metrics from self.bam_file

Attributes

bam_file	the bam file that metrics are generated from
extra_docs	

See also:

sctools.metrics.aggregator, *sctools.metrics.merge*, *sctools.metrics.writer*

class sctools.metrics.gatherer.**GatherCellMetrics**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Bases: *sctools.metrics.gatherer.MetricGatherer*

..currentmodule:: sctools.metrics

This module defines classes to gather metrics across the cells or genes of an experiment and write them to gzip-compressed csv files

MetricGatherer(bam_file, output_stem, ...)	Gathers Metrics from an experiment
GatherCellMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers
GatherGeneMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers

See also:

sctools.metrics.aggregator, *sctools.metrics.merge*, *sctools.metrics.writer*

`bam_file` must be sorted by gene (`GE`), molecule (`UB`), and cell (`CB`), where gene varies fastest.

```
>>> from sctools.metrics.gatherer import GatherCellMetrics
>>> import os, tempfile
```

```
>>> # example data
>>> bam_file = os.path.abspath(__file__) + '../test/data/test.bam'
```

(continues on next page)

(continued from previous page)

```
>>> temp_dir = tempfile.mkdtemp()
>>> g = GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test',␣
↪compress=True)
>>> g.extract_metrics()
```

GatherGeneMetrics

property bam_file: str
the bam file that metrics are generated from

extra_docs = "\n Notes\n -----\n ``bam_file`` must be sorted by gene (``GE``), molecule (``UB``), and cell (``CB``), where gene\n varies fastest.\n\n Examples\n --------\n >>> from sctools.metrics.gatherer import GatherCellMetrics\n >>> import os, tempfile\n\n >>> # example data\n >>> bam_file = os.path.abspath(__file__) + '../test/data/test.bam'\n >>> temp_dir = tempfile.mkdtemp()\n >>> g = GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test', compress=True)\n >>> g.extract_metrics()\n\n See Also\n --------\n GatherGeneMetrics\n\n "

extract_metrics(*mode: str = 'rb'*) → None
Extract cell metrics from self.bam_file

Parameters **mode** (*str, optional*) – Open mode for self.bam. 'r' -> sam, 'rb' -> bam (default = 'rb').

class sctools.metrics.gatherer.**GatherGeneMetrics**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Bases: *sctools.metrics.gatherer.MetricGatherer*

..currentmodule:: sctools.metrics

This module defines classes to gather metrics across the cells or genes of an experiment and write them to gzip-compressed csv files

MetricGatherer(bam_file, output_stem, ...)	Gathers Metrics from an experiment
GatherCellMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers
GatherGeneMetrics(bam_file, output_stem, ...)	Sequence Metric Gatherers

See also:

sctools.metrics.aggregator, *sctools.metrics.merge*, *sctools.metrics.writer*

`bam_file` must be sorted by molecule (`UB`), cell (`CB`), and gene (`GE`), where molecule varies fastest.

```
>>> from sctools.metrics.gatherer import GatherCellMetrics
>>> import os, tempfile
```

```
>>> # example data
>>> bam_file = os.path.abspath(__file__) + '../test/data/test.bam'
>>> temp_dir = tempfile.mkdtemp()
>>> g = GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test',␣
↪compress=True)
>>> g.extract_metrics()
```

GatherGeneMetrics

property bam_file: str

the bam file that metrics are generated from

```
extra_docs = "\n Notes\n -----\n ``bam_file`` must be sorted by molecule (``UB``),
cell (``CB``), and gene (``GE``), where\n molecule varies fastest.\n\n Examples\n
--------\n >>> from sctools.metrics.gatherer import GatherCellMetrics\n >>> import
os, tempfile\n\n >>> # example data\n >>> bam_file = os.path.abspath(__file__) +
'../test/data/test.bam'\n >>> temp_dir = tempfile.mkdtemp()\n >>> g =
GatherCellMetrics(bam_file=bam_file, output_stem=temp_dir + 'test', compress=True)\n
>>> g.extract_metrics()\n\n See Also\n --------\n GatherGeneMetrics\n\n "
```

extract_metrics(*mode: str = 'rb'*) → None
Extract gene metrics from self.bam_file

Parameters **mode** (*str, optional*) – Open mode for self.bam. ‘r’ -> sam, ‘rb’ -> bam (default = ‘rb’).

class sctools.metrics.gatherer.**MetricGatherer**(*bam_file: str*, *output_stem: str*, *mitochondrial_gene_ids: Set[str] = {}*, *compress: bool = True*)

Bases: object

Gathers Metrics from an experiment

Because molecules tend to have relatively small numbers of reads, the memory footprint of this method is typically small (tens of megabytes).

Parameters

- **bam_file** (*str*) – the bam file containing the reads that metrics should be calculated from. Can be a chunk of cells or an entire experiment
- **output_stem** (*str*) – the file stem for the gzipped csv output

extract_metrics()
extracts metrics from bam_file and writes them to output_stem.csv.gz

property bam_file: str
the bam file that metrics are generated from

extract_metrics(*mode='rb'*) → None
extract metrics from the provided bam file and write the results to csv.

Parameters **mode** (*{'r', 'rb'}, default 'rb'*) – the open mode for pysam.AlignmentFile. ‘r’ indicates the input is a sam file, and ‘rb’ indicates a bam file.

9.1.3 sctools.metrics.merge module

Merge Sequence Metrics

..currentmodule:: sctools.metrics

This module defines classes to merge multiple metrics files that have been gathered from bam files containing disjoint sets of cells. This is a common use pattern, as sequencing datasets are often chunked to enable horizontal scaling using scatter-gather patterns.

Classes

MergeMetrics Merge Metrics base class MergeCellMetrics Class to merge cell metrics MergeGeneMetrics Class to merge gene metrics

See also:

sctools.metrics.gatherer, *sctools.metrics.aggregator*, *sctools.metrics.writer*

class sctools.metrics.merge.**MergeCellMetrics**(*metric_files: Sequence[str]*, *output_file: str*)
Bases: *sctools.metrics.merge.MergeMetrics*

execute() → None
Concatenate input cell metric files

Since bam files that metrics are calculated from contain disjoint sets of cells, cell metrics can simply be concatenated together.

class sctools.metrics.merge.**MergeGeneMetrics**(*metric_files: Sequence[str]*, *output_file: str*)
Bases: *sctools.metrics.merge.MergeMetrics*

execute() → None
Merge input gene metric files

The bam files that metrics are calculated from contain disjoint sets of cells, each of which can measure the same genes. As a result, the metric values must be summed (count based metrics) averaged over (fractional, averge, or variance metrics) or recalculated (metrics that depend on other metrics).

class sctools.metrics.merge.**MergeMetrics**(*metric_files: Sequence[str]*, *output_file: str*)
Bases: `object`

Merges multiple metrics files into a single gzip compressed csv file

Parameters

- **metric_files** (*Sequence[str]*) – metrics files to merge
- **output_file** (*str*) – file name for the merged output

execute()
merge metrics files # todo this should probably be wrapped into __init__ to make this more like a function

execute() → None

9.1.4 sctools.metrics.writer module

Metric Writers

..currentmodule:: sctools.metrics

This module defines a class to write metrics to csv as the data is generated, cell by cell or gene by gene. This strategy keeps memory usage low, as no more than a single molecule's worth of sam records and one cell or gene's worth of metric data are in-memory at a time.

Classes

MetricCSVWriter Class to write metrics to file

See also:

sctools.metrics.gatherer, *sctools.metrics.aggregator*, *sctools.metrics.merge*

class sctools.metrics.writer.**MetricCSVWriter**(*output_stem: str*, *compress=True*)
Bases: object

Writes metric information iteratively to (optionally compressed) csv.

Parameters

- **output_stem** (*str*) – File stem for the output file.
- **compress** (*bool, optional*) – Whether or not to compress the output file (default = True).

write_header()
Write the metric header to file.

write()
Write an array of cell or gene metrics to file.

close()
Close the metric file.

close() → None
Close the metrics file.

property filename: str
filename with correct suffix added

write(*index: str*, *record: Mapping[str, numbers.Number]*) → None
Write the array of metric values for a cell or gene to file.

Parameters

- **index** (*str*) – The name of the cell or gene that these metrics summarize
- **record** (*Mapping[str, Number]*) – Output of vars() called on an sctools.metrics.aggregator.MetricAggregator instance, producing a dictionary of keys to metric values.

write_header(*record: Mapping[str, Any]*) → None
Write the metric keys to file, producing the header line of the csv file.

Parameters **record** (*Mapping[str, Any]*) – Output of vars() called on an sctools.metrics.aggregator.MetricAggregator instance, producing a dictionary of keys to metric values.

CHAPTER

TEN

SCTOOLS.TEST PACKAGE

10.1 Submodules

10.1.1 sctools.test.test_bam module

sctools.test.test_bam.**bamfile**(*request*)

sctools.test.test_bam.**indices**(*request*)
fixture returns indices from a SubsetAlignments objects for testing

sctools.test.test_bam.**make_records_from_values**(*tag_keys*, *tags_and_query_name*)

sctools.test.test_bam.**n_nonspecific**()
the number of non-specific records to extract

sctools.test.test_bam.**n_specific**()
the number of specific records to extract

sctools.test.test_bam.**sa_object**(*request*)
fixture returns SubsetAlignments objects for testing

sctools.test.test_bam.**tagged_bam**()

sctools.test.test_bam.**test_chromosome_19_comes_before_21**(*indices*)
chromosome 19 comes before 21 in the test file, this should be replicated in the output

sctools.test.test_bam.**test_correct_number_of_indices_are_extracted**(*sa_object*, *n_specific*, *n_nonspecific*)

sctools.test.test_bam.**test_get_barcode_for_alignment**(*tagged_bam*)

sctools.test.test_bam.**test_get_barcode_for_alignment_raises_error_for_missing_tag**(*tagged_bam*)

sctools.test.test_bam.**test_get_barcodes_from_bam**(*tagged_bam*)

sctools.test.test_bam.**test_get_barcodes_from_bam_with_raise_missing_true_raises_warning_without_cr_barc**

sctools.test.test_bam.**test_incorrect_extension_does_not_raise_when_open_mode_is_specified**()

sctools.test.test_bam.**test_incorrect_extension_without_open_mode_raises_value_error**()

sctools.test.test_bam.**test_indices_are_all_greater_than_zero**(*sa_object*, *n_specific*, *n_nonspecific*)

sctools.test.test_bam.**test_sort_by_tags_and_queryname_sorts_correctly_from_file**()

sctools.test.test_bam.**test_sort_by_tags_and_queryname_sorts_correctly_from_file_no_tag_keys**()

sctools.test.test_bam.**test_sort_by_tags_and_queryname_sorts_correctly_no_tag_keys**()

sctools.test.test_bam.**test_split_bam_raises_value_error_when_passed_bam_without_barcodes**(*bamfile*)

sctools.test.test_bam.**test_split_on_tagged_bam**(*tagged_bam*)

sctools.test.test_bam.**test_split_succeeds_with_raise_missing_false_and_no_cr_barcode_passed**(*tagged_bam*)

sctools.test.test_bam.**test_split_with_large_chunk_size_generates_one_file**(*tagged_bam*)

sctools.test.test_bam.**test_split_with_raise_missing_true_raises_warning_without_cr_barcode_passed**(*tagged*

sctools.test.test_bam.**test_str_and_int_chromosomes_both_function**(*sa_object*)

sctools.test.test_bam.**test_tag_sortable_record_eq_is_false_when_any_difference_exists**()

sctools.test.test_bam.**test_tag_sortable_record_eq_is_true_for_identical_records**()

sctools.test.test_bam.**test_tag_sortable_record_lt_empty_query_name_is_smaller**()

sctools.test.test_bam.**test_tag_sortable_record_lt_empty_tag_is_smaller**()

sctools.test.test_bam.**test_tag_sortable_record_lt_is_false_for_equal_records**()

sctools.test.test_bam.**test_tag_sortable_record_lt_is_true_for_smaller_query_name**()

sctools.test.test_bam.**test_tag_sortable_record_lt_is_true_for_smaller_tag**()

sctools.test.test_bam.**test_tag_sortable_record_lt_is_true_for_smaller_tag_regardless_of_query_name**()

sctools.test.test_bam.**test_tag_sortable_record_missing_tag_value_is_empty_string**()

sctools.test.test_bam.**test_tag_sortable_records_compare_correctly**()

sctools.test.test_bam.**test_tag_sortable_records_raises_error_on_different_tag_lists**()

sctools.test.test_bam.**test_tag_sortable_records_sort_correctly**()

sctools.test.test_bam.**test_tag_sortable_records_sort_correctly_when_already_sorted**()

sctools.test.test_bam.**test_tag_sortable_records_str**()

sctools.test.test_bam.**test_verify_sort_on_unsorted_records_raises_error**()

sctools.test.test_bam.**test_verify_sort_raises_no_error_on_sorted_records**()

sctools.test.test_bam.**test_write_barcodes_to_bins**(*tagged_bam*)

10.1.2 sctools.test.test_barcode module

sctools.test.test_barcode.**barcode_set**()

sctools.test.test_barcode.**short_barcode_set_from_encoded**()

sctools.test.test_barcode.**short_barcode_set_from_iterable**(*request*)

sctools.test.test_barcode.**tagged_bamfile**()

sctools.test.test_barcode.**test_barcode_diversity_is_in_range**(*barcode_set*)

sctools.test.test_barcode.**test_base_frequency_sums_are_all_equal_to_barcode_set_length**(*barcode_set*)

sctools.test.test_barcode.**test_correct_bam_produces_cb_tags**(*tagged_bamfile*, *truncated_whitelist_from_10x*)

sctools.test.test_barcode.**test_correct_barcode_finds_and_corrects_1_base_errors**(*trivial_whitelist*)

sctools.test.test_barcode.**test_correct_barcode_raises_keyerror_when_barcode_has_more_than_one_error**(*trivi*

sctools.test.test_barcode.**test_correct_barcode_raises_keyerror_when_barcode_not_correct_length**(*trivial_whi*

sctools.test.test_barcode.**test_incorrect_input_raises_errors**(*trivial_whitelist*)

sctools.test.test_barcode.**test_iterable_produces_correct_barcodes**(*short_barcode_set_from_encoded*)

sctools.test.test_barcode.**test_reads_barcodes_from_file**(*barcode_set*)

sctools.test.test_barcode.**test_summarize_hamming_distances_gives_reasonable_results**(*short_barcode_set_from_*

sctools.test.test_barcode.**trivial_whitelist**()

sctools.test.test_barcode.**truncated_whitelist_from_10x**()

10.1.3 sctools.test.test_encodings module

sctools.test.test_encodings.**encoder**(*request*)

sctools.test.test_encodings.**encoder_2bit**(*sequence*)

sctools.test.test_encodings.**encoder_3bit**()

sctools.test.test_encodings.**sequence**()

sctools.test.test_encodings.**simple_barcodes**()
simple barcode set with min_hamming = 1, max_hamming = 2

sctools.test.test_encodings.**simple_hamming_distances**(*simple_barcodes*)

sctools.test.test_encodings.**test_encoded_hamming_distance_is_accurate**(*simple_hamming_distances*, *simple_barcodes*, *encoder*)

sctools.test.test_encodings.**test_three_bit_encode_decode_produces_same_string**(*sequence*, *encoder_3bit*)

sctools.test.test_encodings.**test_three_bit_encoder_gets_correct_gc_content**(*sequence*, *encoder_3bit*)

sctools.test.test_encodings.**test_three_bit_encodes_unknown_nucleotides_as_N**(*encoder_3bit*)

sctools.test.test_encodings.**test_two_bit_encode_decode_produces_same_string_except_for_N**(*sequence*, *encoder_2bit*)

sctools.test.test_encodings.**test_two_bit_encoder_gets_correct_gc_content**(*encoder_2bit*)

sctools.test.test_encodings.**test_two_bit_throws_errors_when_asked_to_encode_unknown_nucleotide**(*encoder_2*

10.1.4 sctools.test.test_entrypoints module

sctools.test.test_entrypoints.**test_Attach10XBarcodes_entrypoint**()

sctools.test.test_entrypoints.**test_Attach10XBarcodes_entrypoint_with_whitelist**()

sctools.test.test_entrypoints.**test_AttachBarcodes_entrypoint_with_whitelist**()

sctools.test.test_entrypoints.**test_count_merge**()

sctools.test.test_entrypoints.**test_split_bam**()

sctools.test.test_entrypoints.**test_tag_sort_bam**()

sctools.test.test_entrypoints.**test_tag_sort_bam_dash_t_specified_multiple_times**()

sctools.test.test_entrypoints.**test_tag_sort_bam_no_tags**()

sctools.test.test_entrypoints.**test_verify_bam_sort**()

sctools.test.test_entrypoints.**test_verify_bam_sort_raises_error_on_unsorted**()

10.1.5 sctools.test.test_fastq module

sctools.test.test_fastq.**barcode_generator_with_corrected_cell_barcodes**()

sctools.test.test_fastq.**bytes_fastq_record**()

sctools.test.test_fastq.**embedded_barcode_generator**()

sctools.test.test_fastq.**i7_files_compressions_and_modes**(*request*)
generates different compression types and modes for testing

sctools.test.test_fastq.**reader_all_compressions**(*request*)
generates open fastq reader files for each compression and read mode

sctools.test.test_fastq.**string_fastq_record**()

sctools.test.test_fastq.**test_bytes_fastq_record_quality_score_parsing**(*bytes_fastq_record*)

sctools.test.test_fastq.**test_corrects_barcodes**(*barcode_generator_with_corrected_cell_barcodes*)

sctools.test.test_fastq.**test_embedded_barcode_generator_produces_outputs_of_expected_size**(*embedded_barco*

sctools.test.test_fastq.**test_fastq_returns_correct_filesize_for_single_and_multiple_files**()

sctools.test.test_fastq.**test_fields_populate_properly**(*reader_all_compressions*)

sctools.test.test_fastq.**test_invalid_open_mode_raises_valueerror**()

sctools.test.test_fastq.**test_mixed_filetype_read_gets_correct_record_number**()

sctools.test.test_fastq.**test_non_string_filename_in_iterable_raises_typeerror**()

sctools.test.test_fastq.**test_non_string_filename_raises_typeerror**()

sctools.test.test_fastq.**test_printing_bytes_record_generates_valid_fastq_record**(*bytes_fastq_record*)

sctools.test.test_fastq.**test_printing_string_record_generates_valid_fastq_record**(*string_fastq_record*)

sctools.test.test_fastq.**test_reader_properly_subsets_based_on_indices**()

sctools.test.test_fastq.**test_reader_reads_correct_number_of_records_across_multiple_files**(*reader_all_compr*

sctools.test.test_fastq.**test_reader_reads_first_record**(*reader_all_compressions*)

sctools.test.test_fastq.**test_reader_skips_header_character_raises_value_error**(*i7_files_compressions_and_mode*

test should skip the first name line, shifting each record up 1. As a result, the first sequence should be found in the name field

sctools.test.test_fastq.**test_reader_stores_filenames**()

sctools.test.test_fastq.**test_string_fastq_record_quality_score_parsing**(*string_fastq_record*)

sctools.test.test_fastq.**test_zipping_readers_generates_expected_output**()

sctools.test.test_fastq.**test_zipping_readers_with_indices_generates_expected_output**()

10.1.6 sctools.test.test_gtf module

sctools.test.test_gtf.**files**(*request*)
returns a filename

sctools.test.test_gtf.**test_opens_file_parses_size**(*files*)

sctools.test.test_gtf.**test_opens_file_populates_fields_properly**(*files*)

sctools.test.test_gtf.**test_opens_file_reads_first_line**(*files*)

sctools.test.test_gtf.**test_set_attribute_verify_included_in_output_string**(*files*)

10.1.7 sctools.test.test_metrics module

sctools.test.test_metrics.**mergeable_cell_metrics**()

sctools.test.test_metrics.**mergeable_gene_metrics**()

sctools.test.test_metrics.**split_metrics_file**(*metrics_file*)
produces two mergeable on-disk metric files from a single file that contain the first 3/4 of the file in the first output and the last 3/4 of the file in the second output, such that 1/2 of the metrics in the two files overlap

sctools.test.test_metrics.**test_calculate_cell_metrics_cli**()
test the sctools cell metrics CLI invocation

sctools.test.test_metrics.**test_calculate_gene_metrics_cli**()
test the sctools gene metrics CLI invocation

sctools.test.test_metrics.**test_cell_metrics_mean_n_genes_observed**()
test that the GatherCellMetrics method identifies the correct number of genes per cell, on average.

sctools.test.test_metrics.**test_duplicate_records**(*metrics*, *expected_value*)
Duplicate records are identified by the 1024 bit being set in the sam flag

sctools.test.test_metrics.**test_fragments_number_is_greater_than_molecule_number**(*metrics*)
There should always be more fragments than molecules, as the minimum definition of a molecule is a fragment covered by a single read

sctools.test.test_metrics.**test_gene_metrics_n_genes**()
Test that GatherGeneMetrics identifies the total number of genes in the test file

sctools.test.test_metrics.**test_gzip_compression**(*bam: str*, *gatherer: Callable*)
gzip compression should produce a .gz file which is identical when uncompressed to the uncompressed version

sctools.test.test_metrics.**test_higher_order_metrics_by_gene**(*metrics*, *key*, *expected_value*)
Test metrics that depend on other metrics

This class tests a very large number of higher-order metrics that examine the functionality of the test suite across all measured instances of the metric class. E.g. for cell metrics (class), each test will verify the value for each cell (instance).

Parameters

- **metrics** (*pd.DataFrame*) – Output from subclass of sctools.metrics.MetricAggregator
- **key** (*str*) – The column of metrics to interrogate in the parametrized test
- **expected_value** (*np.ndarray*) – An array of expected values

sctools.test.test_metrics.**test_merge_cell_metrics_cli**(*mergeable_cell_metrics*)
test the sctools merge cell metrics CLI invocation

sctools.test.test_metrics.**test_merge_cell_metrics_does_not_correct_duplicates**(*mergeable_cell_metrics*)
test takes offset cell metrics outputs and merges them. Cell metrics does not check for duplication, so should return a 2x length file.

sctools.test.test_metrics.**test_merge_gene_metrics_averages_over_multiply_detected_genes**(*mergeable_gene_me*

sctools.test.test_metrics.**test_merge_gene_metrics_cli**(*mergeable_gene_metrics*)
test the sctools merge gene metrics CLI invocation

sctools.test.test_metrics.**test_metrics_highest_expression_class**(*metrics*, *expected_value*)
for gene metrics, this is the highest expression gene. For cell metrics, this is the highest expression cell.

sctools.test.test_metrics.**test_metrics_highest_read_count**(*metrics*, *expected_value*)
Test that each metric identifies the what the highest read count associated with any single entity

sctools.test.test_metrics.**test_metrics_n_fragments**(*metrics*, *expected_value*)
Test that each metric identifies the total number of fragments in the test file.

Fragments are defined as a unique combination of {cell barcode, molecule barcode, strand, position, chromosome}

sctools.test.test_metrics.**test_metrics_n_molecules**(*metrics*, *expected_value*)
Test that each metric identifies the total number of molecules in the test file

Molecules are defined as a unique combination of {cell barcode, molecule barcode, gene}

sctools.test.test_metrics.**test_metrics_n_reads**(*metrics*, *expected_value*)
test that the metrics identify the correct read number

sctools.test.test_metrics.**test_metrics_number_perfect_cell_barcodes**(*metrics*, *expected_value*)
Test that each metric correctly identifies the number of perfect cell barcodes where CB == CR

sctools.test.test_metrics.**test_metrics_number_perfect_molecule_barcodes**(*metrics*, *expected_value*)
Test that each metric correctly identifies the number of perfect molecule barcodes where UB == UR

sctools.test.test_metrics.**test_reads_mapped_exonic**(*metrics*, *expected_value*)
Test that each metric identifies the number of reads mapped to an exon (XF==’CODING’)

sctools.test.test_metrics.**test_reads_mapped_intronic**(*metrics*, *expected_value*)
Test that each metric identifies the number of reads mapped to an intron (XF==’INTRONIC’)

sctools.test.test_metrics.**test_reads_mapped_uniquely**(*metrics*, *expected_value*)
Uniquely mapping reads will be tagged with NH==1

sctools.test.test_metrics.**test_reads_mapped_utr**(*metrics*, *expected_value*)
Test that each metric identifies the number of reads mapped to a UTR (XF==’UTR’)

sctools.test.test_metrics.**test_single_read_evidence**(*metrics*, *key*, *expected_value*)
We want to determine how many molecules and fragments are covered by only one read, as reads covered by multiple reads have much lower probabilities of being the result of error processes.

sctools.test.test_metrics.**test_spliced_reads**(*metrics*, *expected_value*)
This pipeline defines spliced reads as containing an N segment of any length in the cigar string

10.1.8 sctools.test.test_stats module

sctools.test.test_stats.**test_balanced_data_produces_entropy_1**()

sctools.test.test_stats.**test_balanced_unnormalized_data_produces_entropy_1**()

sctools.test.test_stats.**test_concentrated_data_produces_entropy_0**()

sctools.test.test_stats.**test_concentrated_unnormalized_data_produces_entropy_0**()

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

N

S

T

U

V

W

Z