
pysam documentation

Release 0.15.0

Andreas Heger, Kevin Jacobs et al.

Jul 27, 2018

Contents

1	Contents	3
1.1	pysam - An interface for reading and writing SAM files	3
1.2	Working with BAM/CRAM/SAM-formatted files	34
1.3	Using samtools commands within python	36
1.4	Working with tabix-indexed files	37
1.5	Working with VCF/BCF formatted files	37
1.6	Extending pysam	39
1.7	Installing pysam	40
1.8	FAQ	42
1.9	Developer's guide	46
1.10	Release notes	47
1.11	Benchmarking	59
1.12	Glossary	63
2	Indices and tables	65
3	References	67
	Bibliography	69

Author Andreas Heger, Kevin Jacobs and contributors

Date Jul 27, 2018

Version 0.15.0

Pysam is a python module for reading, manipulating and writing genomic data sets.

Pysam is a wrapper of the [htslib](#) C-API and provides facilities to read and write SAM/BAM/VCF/BCF/BED/GFF/GTF/FASTA/FASTQ files as well as access to the command line functionality of the [samtools](#) and [bcftools](#) packages. The module supports compression and random access through indexing.

This module provides a low-level wrapper around the [htslib](#) C-API as using cython and a high-level, pythonic API for convenient access to the data within genomic file formats.

The current version wraps *htslib-1.7*, *samtools-1.7* and *bcftools-1.6*.

To install the latest release, type:

```
pip install pysam
```

See the [Installation notes](#) for details.

1.1 pysam - An interface for reading and writing SAM files

1.1.1 Introduction

Pysam is a python module that makes it easy to read and manipulate mapped short read sequence data stored in SAM/BAM files. It is a lightweight wrapper of the [htslib](#) C-API.

This page provides a quick introduction in using pysam followed by the API. See *Working with BAM/CRAM/SAM-formatted files* for more detailed usage instructions.

To use the module to read a file in BAM format, create a *AlignmentFile* object:

```
import pysam
samfile = pysam.AlignmentFile("ex1.bam", "rb")
```

Once a file is opened you can iterate over all of the read mapping to a specified region using *fetch()*. Each iteration returns a *AlignedSegment* object which represents a single read along with its fields and optional tags:

```
for read in samfile.fetch('chr1', 100, 120):
    print read

samfile.close()
```

To give:

```
EAS56_57:6:190:289:82      0      99      <<<7<<<;<<<<<<<8;;<7;4<;<::;94<;      1
→69      CTCAAGGTTGTTGCAAGGGGTTCTATGTGAACAAA      0      192      1
EAS56_57:6:190:289:82      0      99      <<<<<;<<<<<<<<<;<<;<<<<;8<6;9;;2;      1
→137     AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC      73      64      1
EAS51_64:3:190:727:308      0      102     <<<<<<<<<<<<<<<<<<<<<<<<<::<<<844      1
→99      GGTGCAGAGCCGAGTCACGGGGTTGCCAGCACAGG      99      18      1
...
```

You can also write to a *AlignmentFile*:

```
import pysam
samfile = pysam.AlignmentFile("ex1.bam", "rb")
pairedreads = pysam.AlignmentFile("allpaired.bam", "wb", template=samfile)
for read in samfile.fetch():
    if read.is_paired:
        pairedreads.write(read)

pairedreads.close()
samfile.close()
```

An alternative way of accessing the data in a SAM file is by iterating over each base of a specified region using the `pileup()` method. Each iteration returns a `PileupColumn` which represents all the reads in the SAM file that map to a single base in the reference sequence. The list of reads are represented as `PileupRead` objects in the `PileupColumn.pileups` property:

```
import pysam
samfile = pysam.AlignmentFile("ex1.bam", "rb" )
for pileupcolumn in samfile.pileup("chr1", 100, 120):
    print ("\ncoverage at base %s = %s" %
          (pileupcolumn.pos, pileupcolumn.n))
    for pileupread in pileupcolumn.pileups:
        if not pileupread.is_del and not pileupread.is_refskip:
            # query position is None if is_del or is_refskip is set.
            print ('\tbase in read %s = %s' %
                  (pileupread.alignment.query_name,
                   pileupread.alignment.query_sequence[pileupread.query_position]))

samfile.close()
```

The above code outputs:

```
coverage at base 99 = 1
    base in read EAS56_57:6:190:289:82 = A

coverage at base 100 = 1
    base in read EAS56_57:6:190:289:82 = G

coverage at base 101 = 1
    base in read EAS56_57:6:190:289:82 = G

coverage at base 102 = 2
    base in read EAS56_57:6:190:289:82 = G
    base in read EAS51_64:3:190:727:308 = G
...
```

Commands available in `csamtools` are available as simple function calls. For example:

```
pysam.sort("-o", "output.bam", "ex1.bam")
```

corresponds to the command line:

```
samtools sort -o output.bam ex1.bam
```

Analogous to `AlignmentFile`, a `TabixFile` allows fast random access to compressed and tabix indexed tab-separated file formats with genomic data:


```
import pysam
tabixfile = pysam.TabixFile("example.gtf.gz")

for gtf in tabixfile.fetch("chr1", 1000, 2000):
    print (gtf.contig, gtf.start, gtf.end, gtf.gene_id)
```

TabixFile implements lazy parsing in order to iterate over large tables efficiently.

More detailed usage instructions is at [Working with BAM/CRAM/SAM-formatted files](#).

Note: Coordinates in pysam are always 0-based (following the python convention). SAM text files use 1-based coordinates.

Note:

The above examples can be run in the `tests` directory of the installation directory. Type ‘make’ before running them.

See also:

<https://github.com/pysam-developers/pysam>

The pysam code repository, containing source code and download instructions

<http://pysam.readthedocs.org/en/latest/>

The pysam website containing documentation

1.1.2 API

SAM/BAM/CRAM files

Objects of type *AlignmentFile* allow working with BAM/SAM formatted files.

class `pysam.AlignmentFile`

`AlignmentFile(filepath_or_object, mode=None, template=None, reference_names=None, reference_lengths=None, text=NULL, header=None, add_sq_text=False, check_header=True, check_sq=True, reference_filename=None, filename=None, index_filename=None, filepath_index=None, require_index=False, duplicate_filehandle=True, ignore_truncation=False, threads=1)`

A *SAM/BAM/CRAM* formatted file.

If *filepath_or_object* is a string, the file is automatically opened. If *filepath_or_object* is a python File object, the already opened file will be used.

If the file is opened for reading and an index exists (if file is BAM, a .bai file or if CRAM a .crai file), it will be opened automatically. *index_filename* may be specified explicitly. If the index is not named in the standard manner, not located in the same directory as the BAM/CRAM file, or is remote. Without an index, random access via *fetch()* and *pileup()* is disabled.

For writing, the header of a *SAM* file/*BAM* file can be constituted from several sources (see also the samtools format specification):

1. If *template* is given, the header is copied from a another *AlignmentFile* (*template* must be a *AlignmentFile*).
2. If *header* is given, the header is built from a multi-level dictionary.

3. If *text* is given, new header text is copied from raw text.
4. The names (*reference_names*) and lengths (*reference_lengths*) are supplied directly as lists.

When reading or writing a CRAM file, the filename of a FASTA-formatted reference can be specified with *reference_filename*.

By default, if a file is opened in mode 'r', it is checked for a valid header (*check_header* = True) and a definition of chromosome names (*check_sq* = True).

Parameters

- **mode** (*string*) – *mode* should be *r* for reading or *w* for writing. The default is text mode (*SAM*). For binary (*BAM*) I/O you should append *b* for compressed or *u* for uncompressed *BAM* output. Use *h* to output header information in text (*TAM*) mode. Use *c* for CRAM formatted files.

If *b* is present, it must immediately follow *r* or *w*. Valid modes are *r*, *w*, *wh*, *rb*, *wb*, *wbu*, *wb0*, *rc* and *wc*. For instance, to open a *BAM* formatted file for reading, type:

```
f = pysam.AlignmentFile('ex1.bam', 'rb')
```

If *mode* is not specified, the method will try to auto-detect in the order 'rb', 'r', thus both the following should work:

```
f1 = pysam.AlignmentFile('ex1.bam')
f2 = pysam.AlignmentFile('ex1.sam')
```

- **template** (*AlignmentFile*) – when writing, copy header from file *template*.
- **header** (*dict* or *AlignmentHeader*) – when writing, build header from a multi-level dictionary. The first level are the four types ('HD', 'SQ', ...). The second level are a list of lines, with each line being a list of tag-value pairs. The header is constructed first from all the defined fields, followed by user tags in alphabetical order. Alternatively, an *AlignmentHeader* object can be passed directly.
- **text** (*string*) – when writing, use the string provided as the header
- **reference_names** (*list*) – see *reference_lengths*
- **reference_lengths** (*list*) – when writing or opening a SAM file without header build header from list of chromosome names and lengths. By default, 'SQ' and 'LN' tags will be added to the header text. This option can be changed by unsetting the flag *add_sq_text*.
- **add_sq_text** (*bool*) – do not add 'SQ' and 'LN' tags to header. This option permits construction *SAM* formatted files without a header.
- **add_sam_header** (*bool*) – when outputting SAM the default is to output a header. This is equivalent to opening the file in 'wh' mode. If this option is set to False, no header will be output. To read such a file, set *check_header=False*.
- **check_header** (*bool*) – obsolete: when reading a SAM file, check if header is present (default=True)
- **check_sq** (*bool*) – when reading, check if SQ entries are present in header (default=True)
- **reference_filename** (*string*) – Path to a FASTA-formatted reference file. Valid only for CRAM files. When reading a CRAM file, this overrides both \$REF_PATH and the URL specified in the header (UR tag), which are normally used to find the reference.

- **index_filename** (*string*) – Explicit path to the index file. Only needed if the index is not named in the standard manner, not located in the same directory as the BAM/CRAM file, or is remote. An `IOError` is raised if the index cannot be found or is invalid.
- **filepath_index** (*string*) – Alias for *index_filename*.
- **require_index** (*bool*) – When reading, require that an index file is present and is valid or raise an `IOError`. (default=False)
- **filename** (*string*) – Alternative to *filepath_or_object*. Filename of the file to be opened.
- **duplicate_filehandle** (*bool*) – By default, file handles passed either directly or through File-like objects will be duplicated before passing them to `htslib`. The duplication prevents issues where the same stream will be closed by `htslib` and through destruction of the high-level python object. Set to `False` to turn off duplication.
- **ignore_truncation** (*bool*) – Issue a warning, instead of raising an error if the current file appears to be truncated due to a missing EOF marker. Only applies to bgzipped formats. (Default=False)
- **format_options** (*list*) – A list of key=value strings, as accepted by `-input-fmt-option` and `-output-fmt-option` in `samtools`.
- **threads** (*integer*) – Number of threads to use for compressing/decompressing BAM/CRAM files. Setting threads to `> 1` cannot be combined with *ignore_truncation*. (Default=1)

check_index (*self*)
return True if index is present.

Raises

- `AttributeError` – if `htsfile` is *SAM* formatted and thus has no index.
- `ValueError` – if `htsfile` is closed or index could not be opened.

close (*self*)
closes the *pysam.AlignmentFile*.

count (*self*, *contig=None*, *start=None*, *stop=None*, *region=None*, *until_eof=False*, *read_callback='nofilter'*, *reference=None*, *end=None*)
count the number of reads in *region*

The region is specified by *contig*, *start* and *stop*. *reference* and *end* are also accepted for backward compatibility as synonyms for *contig* and *stop*, respectively. Alternatively, a *samtools region* string can be supplied.

A *SAM* file does not allow random access and if *region* or *contig* are given, an exception is raised.

Parameters

- **contig** (*string*) – reference_name of the genomic region (chromosome)
- **start** (*int*) – start of the genomic region (0-based inclusive)
- **stop** (*int*) – end of the genomic region (0-based exclusive)
- **region** (*string*) – a region string in `samtools` format.
- **until_eof** (*bool*) – count until the end of the file, possibly including unmapped reads as well.
- **read_callback** (*string or function*) – select a call-back to ignore reads when counting. It can be either a string with the following values:

all skip reads in which any of the following flags are set: BAM_FUNMAP, BAM_FSECONDARY, BAM_FQCFAIL, BAM_FDUP

nofilter uses every single read

Alternatively, *read_callback* can be a function `check_read(read)` that should return True only for those reads that shall be included in the counting.

- **reference** (*string*) – backward compatible synonym for *contig*
- **end** (*int*) – backward compatible synonym for *stop*

Raises `ValueError` – if the genomic coordinates are out of range or invalid.

count_coverage (*self*, *contig*, *start=None*, *stop=None*, *region=None*, *quality_threshold=15*, *read_callback='all'*, *reference=None*, *end=None*)

count the coverage of genomic positions by reads in *region*.

The region is specified by *contig*, *start* and *stop*. *reference* and *end* are also accepted for backward compatibility as synonyms for *contig* and *stop*, respectively. Alternatively, a *samtools region* string can be supplied. The coverage is computed per-base [ACGT].

Parameters

- **contig** (*string*) – reference_name of the genomic region (chromosome)
- **start** (*int*) – start of the genomic region (0-based inclusive). If not given, count from the start of the chromosome.
- **stop** (*int*) – end of the genomic region (0-based exclusive). If not given, count to the end of the chromosome.
- **region** (*int*) – a region string.
- **quality_threshold** (*int*) – quality_threshold is the minimum quality score (in phred) a base has to reach to be counted.
- **read_callback** (*string or function*) – select a call-back to ignore reads when counting. It can be either a string with the following values:

all skip reads in which any of the following flags are set: BAM_FUNMAP, BAM_FSECONDARY, BAM_FQCFAIL, BAM_FDUP

nofilter uses every single read

Alternatively, *read_callback* can be a function `check_read(read)` that should return True only for those reads that shall be included in the counting.

- **reference** (*string*) – backward compatible synonym for *contig*
- **end** (*int*) – backward compatible synonym for *stop*

Raises `ValueError` – if the genomic coordinates are out of range or invalid.

Returns four array.arrays of the same length in order A C G T

Return type `tuple`

fetch (*self*, *contig=None*, *start=None*, *stop=None*, *region=None*, *tid=None*, *until_eof=False*, *multiple_iterators=False*, *reference=None*, *end=None*)

fetch reads aligned in a *region*.

See `parse_region()` for more information on how genomic regions can be specified. *reference* and *end* are also accepted for backward compatibility as synonyms for *contig* and *stop*, respectively.

Without a *contig* or *region* all mapped reads in the file will be fetched. The reads will be returned ordered by reference sequence, which will not necessarily be the order within the file. This mode of iteration still requires an index. If there is no index, use `until_eof=True`.

If only *contig* is set, all reads aligned to *contig* will be fetched.

A *SAM* file does not allow random access. If *region* or *contig* are given, an exception is raised.

Parameters

- **until_eof** (*bool*) – If *until_eof* is True, all reads from the current file position will be returned in order as they are within the file. Using this option will also fetch unmapped reads.
- **multiple_iterators** (*bool*) – If *multiple_iterators* is True, multiple iterators on the same file can be used at the same time. The iterator returned will receive its own copy of a filehandle to the file effectively re-opening the file. Re-opening a file creates some overhead, so beware.

Returns

Return type An iterator over a collection of reads.

Raises `ValueError` – if the genomic coordinates are out of range or invalid or the file does not permit random access to genomic coordinates.

find_introns (*self*, *read_iterator*)

Return a dictionary {(start, stop): count} Listing the intronic sites in the reads (identified by ‘N’ in the cigar strings), and their support (= number of reads).

read_iterator can be the result of a `.fetch(...)` call. Or it can be a generator filtering such reads. Example `samfile.find_introns((read for read in samfile.fetch(...) if read.is_reverse)`

find_introns_slow (*self*, *read_iterator*)

Return a dictionary {(start, stop): count} Listing the intronic sites in the reads (identified by ‘N’ in the cigar strings), and their support (= number of reads).

read_iterator can be the result of a `.fetch(...)` call. Or it can be a generator filtering such reads. Example `samfile.find_introns((read for read in samfile.fetch(...) if read.is_reverse)`

get_index_statistics (*self*)

return statistics about mapped/unmapped reads per chromosome as they are stored in the index.

Returns list – ‘mapped’, ‘unmapped’ and ‘total’.

Return type a list of records for each chromosome. Each record has the attributes ‘contig’,

get_reference_length (*self*, *reference*)

return *reference* name corresponding to numerical *tid*

get_reference_name (*self*, *tid*)

return *reference* name corresponding to numerical *tid*

get_tid (*self*, *reference*)

return the numerical *tid* corresponding to *reference*

returns -1 if reference is not known.

getrname (*self*, *tid*)

deprecated, use `get_reference_name()` instead

gettid (*self*, *reference*)

deprecated, use `get_tid()` instead

has_index (*self*)
return true if htsfile has an existing (and opened) index.

head (*self*, *n*, *multiple_iterators=True*)
return an iterator over the first *n* alignments.

This iterator is useful for inspecting the bam-file.

Parameters **multiple_iterators** (*bool*) – is set to `True` by default in order to avoid changing the current file position.

Returns

Return type an iterator over a collection of reads

is_valid_tid (*self*, *int tid*)
return `True` if the numerical *tid* is valid; `False` otherwise.

Note that the unmapped tid code (-1) counts as an invalid.

lengths
tuple of the lengths of the *reference* sequences. This is a read-only attribute. The lengths are in the same order as `pysam.AlignmentFile.references`

mapped
int with total number of mapped alignments according to the statistics recorded in the index. This is a read-only attribute.

mate (*self*, *AlignedSegment read*)
return the mate of *AlignedSegment read*.

Note: Calling this method will change the file position. This might interfere with any iterators that have not re-opened the file.

Note: This method is too slow for high-throughput processing. If a read needs to be processed with its mate, work from a read name sorted file or, better, cache reads.

Returns :class:`~pysam.AlignedSegment`

Return type the mate

Raises `ValueError` – if the read is unpaired or the mate is unmapped

next

nocoordinate
int with total number of reads without coordinates according to the statistics recorded in the index. This is a read-only attribute.

references
“int with the number of *reference* sequences in the file. This is a read-only attribute.

pileup (*self*, *contig=None*, *start=None*, *stop=None*, *region=None*, *reference=None*, *end=None*,
 ***kwargs*)
perform a *pileup* within a *region*. The region is specified by *contig*, *start* and *stop* (using 0-based indexing). *reference* and *end* are also accepted for backward compatibility as synonyms for *contig* and *stop*, respectively. Alternatively, a samtools ‘region’ string can be supplied.

Without ‘contig’ or ‘region’ all reads will be used for the pileup. The reads will be returned ordered by *contig* sequence, which will not necessarily be the order within the file.

Note that *SAM* formatted files do not allow random access. In these files, if a ‘region’ or ‘contig’ are given an exception is raised.

Note: ‘all’ reads which overlap the region are returned. The first base returned will be the first base of the first read ‘not’ necessarily the first base of the region used in the query.

Parameters

- **truncate** (*bool*) – By default, the samtools pileup engine outputs all reads overlapping a region. If truncate is True and a region is given, only columns in the exact region specified are returned.
- **max_depth** (*int*) – Maximum read depth permitted. The default limit is ‘8000’.
- **stepper** (*string*) – The stepper controls how the iterator advances. Possible options for the stepper are
 - all** skip reads in which any of the following flags are set: BAM_FUNMAP, BAM_FSECONDARY, BAM_FQCFAIL, BAM_FDUP
 - nofilter** uses every single read turning off any filtering.
 - samtools** same filter and read processing as in *csamtools* pileup. For full compatibility, this requires a ‘fastfile’ to be given. The following options all pertain to filtering of the samtools stepper.
- **fastfile** (*FastaFile* object.) – This is required for some of the steppers.
- **ignore_overlaps** (*bool*) – If set to True, detect if read pairs overlap and only take the higher quality base. This is the default.
- **flag_filter** (*int*) – ignore reads where any of the bits in the flag are set. The default is BAM_FUNMAP | BAM_FSECONDARY | BAM_FQCFAIL | BAM_FDUP.
- **flag_require** (*int*) – only use reads where certain flags are set. The default is 0.
- **ignore_orphans** (*bool*) – ignore orphans (paired reads that are not in a proper pair). The default is to ignore orphans.
- **min_base_quality** (*int*) – Minimum base quality. Bases below the minimum quality will not be output.
- **adjust_capq_threshold** (*int*) – adjust mapping quality. The default is 0 for no adjustment. The recommended value for adjustment is 50.
- **min_mapping_quality** (*int*) – only use reads above a minimum mapping quality. The default is 0.
- **compute_baq** (*bool*) – re-alignment computing per-Base Alignment Qualities (BAQ). The default is to do re-alignment. Realignment requires a reference sequence. If none is present, no realignment will be performed.
- **redo_baq** (*bool*) – recompute per-Base Alignment Quality on the fly ignoring existing base qualities. The default is False (use existing base qualities).
- **adjust_capq_threshold** – adjust mapping quality. The default is 0 for no adjustment. The recommended value for adjustment is 50.

Returns

Return type an iterator over genomic positions.

references

tuple with the names of *reference* sequences. This is a read-only attribute

text

deprecated, use `.header` directly

unmapped

int with total number of unmapped reads according to the statistics recorded in the index. This number of reads includes the number of reads without coordinates. This is a read-only attribute.

write (*self*, *AlignedSegment read*) → int

write a single *pysam.AlignedSegment* to disk.

Raises `ValueError` – if the writing failed

Returns `int` – this will be 0.

Return type the number of bytes written. If the file is closed,

An *AlignedSegment* represents an aligned segment within a SAM/BAM file.

class `pysam.AlignedSegment` (*AlignmentHeader header=None*)

Class representing an aligned segment.

This class stores a handle to the samtools C-structure representing an aligned read. Member read access is forwarded to the C-structure and converted into python objects. This implementation should be fast, as only the data needed is converted.

For write access, the C-structure is updated in-place. This is not the most efficient way to build BAM entries, as the variable length data is concatenated and thus needs to be resized if a field is updated. Furthermore, the BAM entry might be in an inconsistent state.

One issue to look out for is that the sequence should always be set *before* the quality scores. Setting the sequence will also erase any quality scores that were set previously.

:param header – AlignmentHeader object to map numerical: identifiers to chromosome names. If not given, an empty header is created.

aend

deprecated, `reference_end` instead

alen

deprecated, `reference_length` instead

aligned_pairs

deprecated, use `get_aligned_pairs()` instead

bin

properties bin

blocks

deprecated, use `get_blocks()` instead

cigar

deprecated, use `cigartuples` instead

cigarstring

the *cigar* alignment as a string.

The cigar string is a string of alternating integers and characters denoting the length and the type of an operation.

Note: The order length,operation is specified in the SAM format. It is different from the order of the *cigar* property.

Returns None if not present.

To unset the cigarstring, assign None or the empty string.

cigartuples

the *cigar* alignment. The alignment is returned as a list of tuples of (operation, length).

If the alignment is not present, None is returned.

The operations are:

M	BAM_CMATCH	0
I	BAM_CINS	1
D	BAM_CDEL	2
N	BAM_CREF_SKIP	3
S	BAM_CSOFT_CLIP	4
H	BAM_CHARD_CLIP	5
P	BAM_CPAD	6
=	BAM_CEQUAL	7
X	BAM_CDIFF	8
B	BAM_CBACK	9

Note: The output is a list of (operation, length) tuples, such as [(0, 30)]. This is different from the SAM specification and the *cigarstring* property, which uses a (length, operation) order, for example: 30M.

To unset the cigar property, assign an empty list or None.

compare (*self*, *AlignedSegment other*)

return -1,0,1, if contents in this are binary <,<=,> to *other*

flag

properties flag

from_dict (*type cls*, *sam_dict*, *AlignmentHeader header*)

parses a dictionary representation of the aligned segment.

Parameters -- dictionary of alignment values, keys corresponding to output from (*sam_dict*)-todict().

fromstring (*type cls*, *sam*, *AlignmentHeader header*)

parses a string representation of the aligned segment.

The input format should be valid SAM format.

:param sam – *SAM* formatted string:

get_aligned_pairs (*self*, *matches_only=False*, *with_seq=False*)

a list of aligned read (query) and reference positions.

For inserts, deletions, skipping either query or reference position may be None.

Padding is currently not supported and leads to an exception.

Parameters

- **matches_only** (*bool*) – If True, only matched bases are returned - no None on either side.
- **with_seq** (*bool*) – If True, return a third element in the tuple containing the reference sequence. Substitutions are lower-case. This option requires an MD tag to be present.

Returns *aligned_pairs*

Return type list of tuples

get_blocks (*self*)

a list of start and end positions of aligned gapless blocks.

The start and end positions are in genomic coordinates.

Blocks are not normalized, i.e. two blocks might be directly adjacent. This happens if the two blocks are separated by an insertion in the read.

get_cigar_stats (*self*)

summary of operations in cigar string.

The output order in the array is “MIDNSHP=X” followed by a field for the NM tag. If the NM tag is not present, this field will always be 0.

M	BAM_CMATCH	0
I	BAM_CINS	1
D	BAM_CDEL	2
N	BAM_CREF_SKIP	3
S	BAM_CSOFT_CLIP	4
H	BAM_CHARD_CLIP	5
P	BAM_CPAD	6
=	BAM_CEQUAL	7
X	BAM_CDIFF	8
B	BAM_CBACK	9
NM	NM tag	10

If no cigar string is present, empty arrays will be returned.

Returns *arrays* – each cigar operation, the second contains the number of blocks for each cigar operation.

Return type two arrays. The first contains the nucleotide counts within

get_forward_qualities (*self*)

return the original read sequence.

Reads mapping to the reverse strand will be reverse complemented.

get_forward_sequence (*self*)

return the original read sequence.

Reads mapping to the reverse strand will be reverse complemented.

get_overlap (*self*, *uint32_t start*, *uint32_t end*)

return number of aligned bases of read overlapping the interval *start* and *end* on the reference sequence.

Return None if cigar alignment is not available.

get_reference_positions (*self*, *full_length=False*)

a list of reference positions that this read aligns to.

By default, this method only returns positions in the reference that are within the alignment. If *full_length* is set, None values will be included for any soft-clipped or unaligned positions within the read. The returned list will thus be of the same length as the read.

get_reference_sequence (*self*)

return the reference sequence in the region that is covered by the alignment of the read to the reference.

This method requires the MD tag to be set.

get_tag (*self*, *tag*, *with_value_type=False*)

retrieves data from the optional alignment section given a two-letter *tag* denoting the field.

The returned value is cast into an appropriate python type.

This method is the fastest way to access the optional alignment section if only few tags need to be retrieved.

Possible value types are “AcCsSlfZHB” (see BAM format specification) as well as additional value type ‘d’ as implemented in htlib.

Parameters

- **tag** – data tag.
- **with_value_type** (*Optional[bool]*) – if set to True, the return value is a tuple of (tag value, type code). (default False)

Returns

- A python object with the value of the *tag*. The type of the
- *object depends on the data type in the data record.*

Raises `KeyError` – If *tag* is not present, a `KeyError` is raised.

get_tags (*self*, *with_value_type=False*)

the fields in the optional alignment section.

Returns a list of all fields in the optional alignment section. Values are converted to appropriate python values. For example:

```
[(NM, 2), (RG, "GJP00TM04")]
```

If *with_value_type* is set, the value type as encode in the `AlignedSegment` record will be returned as well:

```
[(NM, 2, "i"), (RG, "GJP00TM04", "Z")]
```

This method will convert all values in the optional alignment section. When getting only one or few tags, please see `get_tag()` for a quicker way to achieve this.

has_tag (*self*, *tag*)

returns true if the optional alignment section contains a given *tag*.

infer_query_length (*self*, *always=False*)

infer query length from CIGAR alignment.

This method deduces the query length from the CIGAR alignment but does not include hard-clipped bases.

Returns None if CIGAR alignment is not present.

If *always* is set to True, *infer_read_length* is used instead. This is deprecated and only present for backward compatibility.

infer_read_length (*self*)

infer read length from CIGAR alignment.

This method deduces the read length from the CIGAR alignment including hard-clipped bases.

Returns None if CIGAR alignment is not present.

inferred_length

deprecated, use `infer_query_length()` instead

is_duplicate

true if optical or PCR duplicate

is_paired

true if read is paired in sequencing

is_proper_pair

true if read is mapped in a proper pair

is_qcfail

true if QC failure

is_read1

true if this is read1

is_read2

true if this is read2

is_reverse

true if read is mapped to reverse strand

is_secondary

true if not primary alignment

is_supplementary

true if this is a supplementary alignment

is_unmapped

true if read itself is unmapped

isize

deprecated, use `template_length` instead

mapping_quality

mapping quality

mapq

deprecated, use `mapping_quality` instead

mate_is_reverse

true is read is mapped to reverse strand

mate_is_unmapped

true if the mate is unmapped

mpos

deprecated, use `next_reference_start` instead

mrnm

deprecated, use `next_reference_id` instead

next_reference_id

the *reference* id of the mate/next read.

next_reference_name

reference name of the mate/next read (None if no AlignmentFile is associated)

next_reference_start

the position of the mate/next read.

opt (*self*, *tag*)
 deprecated, use `get_tag()` instead

overlap (*self*)
 deprecated, use `get_overlap()` instead

pnext
 deprecated, use `next_reference_start` instead

pos
 deprecated, use `reference_start` instead

positions
 deprecated, use `get_reference_positions()` instead

qend
 deprecated, use `query_alignment_end` instead

qlen
 deprecated, use `query_alignment_length` instead

qname
 deprecated, use `query_name` instead

qqual
 deprecated, use `query_alignment_qualities` instead

qstart
 deprecated, use `query_alignment_start` instead

qual
 deprecated, use `query_qualities` instead

query
 deprecated, use `query_alignment_sequence` instead

query_alignment_end
 end index of the aligned query portion of the sequence (0-based, exclusive)
 This is the index just past the last base in *seq* that is not soft-clipped.

query_alignment_length
 length of the aligned query sequence.
 This is equal to `qend - qstart`

query_alignment_qualities
 aligned query sequence quality values (None if not present). These are the quality values that correspond to *query*, that is, they exclude qualities of *soft clipped* bases. This is equal to `qual[qstart:qend]`.
 Quality scores are returned as a python array of unsigned chars. Note that this is not the ASCII-encoded value typically seen in FASTQ or SAM formatted files. Thus, no offset of 33 needs to be subtracted.
 This property is read-only.

query_alignment_sequence
 aligned portion of the read.
 This is a substring of *seq* that excludes flanking bases that were *soft clipped* (None if not present). It is equal to `seq[qstart:qend]`.
 SAM/BAM files may include extra flanking bases that are not part of the alignment. These bases may be the result of the Smith-Waterman or other algorithms, which may not require alignments that begin at the

first residue or end at the last. In addition, extra sequencing adapters, multiplex identifiers, and low-quality bases that were not considered for alignment may have been retained.

query_alignment_start

start index of the aligned query portion of the sequence (0-based, inclusive).

This the index of the first base in *seq* that is not soft-clipped.

query_length

the length of the query/read.

This value corresponds to the length of the sequence supplied in the BAM/SAM file. The length of a query is 0 if there is no sequence in the BAM/SAM file. In those cases, the read length can be inferred from the CIGAR alignment, see `pysam.AlignedSegment.infer_query_length()`.

The length includes soft-clipped bases and is equal to `len(query_sequence)`.

This property is read-only but can be set by providing a sequence.

Returns 0 if not available.

query_name

the query template name (None if not present)

query_qualities

******read sequence base qualities, including* – term* – *soft clipped* bases (None if not present).

Quality scores are returned as a python array of unsigned chars. Note that this is not the ASCII-encoded value typically seen in FASTQ or SAM formatted files. Thus, no offset of 33 needs to be subtracted.

Note that to set quality scores the sequence has to be set beforehand as this will determine the expected length of the quality score array.

This method raises a ValueError if the length of the quality scores and the sequence are not the same.

query_sequence

read sequence bases, including *soft clipped* bases (None if not present).

Note that assigning to `seq` will invalidate any quality scores. Thus, to in-place edit the sequence and quality scores, copies of the quality scores need to be taken. Consider trimming for example:

```
q = read.query_qualities
read.query_sequence = read.query_sequence[5:10]
read.query_qualities = q[5:10]
```

The sequence is returned as it is stored in the BAM file. Some mappers might have stored a reverse complement of the original read sequence.

reference_end

aligned reference position of the read on the reference genome.

`reference_end` points to one past the last aligned residue. Returns None if not available (read is unmapped or no cigar alignment present).

reference_id

reference ID

Note: This field contains the index of the reference sequence in the sequence dictionary. To obtain the name of the reference sequence, use `get_reference_name()`

reference_length

aligned length of the read on the reference genome.

This is equal to *aend - pos*. Returns None if not available.

reference_name

reference name

reference_start

0-based leftmost coordinate

rlen

deprecated, query_length instead

rname

deprecated, use reference_id instead

rnext

deprecated, use next_reference_id instead

seq

deprecated, use query_sequence instead

setTag (*self*, *tag*, *value*, *value_type=None*, *replace=True*)

deprecated, use set_tag() instead

set_tag (*self*, *tag*, *value*, *value_type=None*, *replace=True*)

sets a particular field *tag* to *value* in the optional alignment section.

value_type describes the type of *value* that is to entered into the alignment record. It can be set explicitly to one of the valid one-letter type codes. If unset, an appropriate type will be chosen automatically based on the python type of *value*.

An existing value of the same *tag* will be overwritten unless *replace* is set to False. This is usually not recommended as a tag may only appear once in the optional alignment section.

If *value* is None, the tag will be deleted.

This method accepts valid SAM specification value types, which are:

```
A: printable char
i: signed int
f: float
Z: printable string
H: Byte array in hex format
B: Integer or numeric array
```

Additionally, it will accept the integer BAM types ('cCsSI')

For htslib compatibility, 'a' is synonymous with 'A' and the method accepts a 'd' type code for a double precision float.

When deducing the type code by the python type of *value*, the following mapping is applied:

```
i: python int
f: python float
Z: python str or bytes
B: python array.array, list or tuple
```

Note that a single character string will be output as 'Z' and not 'A' as the former is the more general type.

set_tags (*self*, *tags*)

sets the fields in the optional alignment section with a list of (tag, value) tuples.

The value type of the values is determined from the python type. Optionally, a type may be given explicitly as a third value in the tuple. For example:

```
x.set_tags([(NM, 2, "i"), (RG, "GJP00TM04", "Z")])
```

This method will not enforce the rule that the same tag may appear only once in the optional alignment section.

tags

deprecated, use `get_tags()` instead

template_length

the observed query template length

tid

deprecated, use `reference_id` instead

tlen

deprecated, use `template_length` instead

to_dict (*self*)

returns a json representation of the aligned segment.

Field names are abbreviated versions of the class attributes.

to_string (*self*)

returns a string representation of the aligned segment.

The output format is valid SAM format if a header is associated with the `AlignedSegment`.

tostring (*self*, *htsfile=None*)

deprecated, use `to_string()` instead.

Parameters -- (deprecated) AlignmentFile object to map numerical
(*htsfile*) – identifiers to chromosome names. This parameter is present for backwards compatibility and ignored.

class pysam.PileupColumn

A pileup of reads at a particular reference sequence position (*column*). A pileup column contains all the reads that map to a certain target base.

This class is a proxy for results returned by the samtools pileup engine. If the underlying engine iterator advances, the results of this column will change.

get_mapping_qualities (*self*)

query mapping quality scores at pileup column position.

Returns list

Return type a list of quality scores

get_num_aligned (*self*)

return number of aligned bases at pileup column position.

This method applies a base quality filter and the number is equal to the size of `get_query_sequences()`, `get_mapping_qualities()`, etc.

get_query_names (*self*)

query/read names aligned at pileup column position.

Returns list

Return type a list of query names at pileup column position.

get_query_positions (*self*)

positions in read at pileup column position.

Returns list

Return type a list of read positions

get_query_qualities (*self*)

query base quality scores at pileup column position.

Returns list

Return type a list of quality scores

get_query_sequences (*self*, *bool* *mark_matches=False*, *bool* *mark_ends=False*, *bool* *add_indels=False*)

query bases/sequences at pileup column position.

Optionally, the bases/sequences can be annotated according to the samtools mpileup format. This is the format description from the samtools mpileup tool:

Information on match, mismatch, indel, strand, mapping quality and start and end of a read are all encoded at the read base column. At this column, a dot stands for a match to the reference base on the forward strand, a comma for a match on the reverse strand, a '>' or '<' for a reference skip, 'ACGTN' for a mismatch on the forward strand and 'acgtn' for a mismatch on the reverse strand. A pattern '[0-9]+[ACGTNacgtn]+' indicates there is an insertion between this reference position and the next reference position. The length of the insertion is given by the integer in the pattern, followed by the inserted sequence. Similarly, a pattern '-[0-9]+[ACGTNacgtn]+' represents a deletion from the reference. The deleted bases will be presented as '*' in the following lines. Also at the read base column, a symbol '^' marks the start of a read. The ASCII of the character following '^' minus 33 gives the mapping quality. A symbol '\$' marks the end of a read segment

To reproduce samtools mpileup format, set all of *mark_matches*, *mark_ends* and *add_indels* to True.

Parameters

- **mark_matches** (*bool*) – If True, output bases matching the reference as “,” or “.” for forward and reverse strand, respectively. This mark requires the reference sequence. If no reference is present, this option is ignored.
- **mark_ends** (*bool*) – If True, add markers “^” and “\$” for read start and end, respectively.
- **add_indels** (*bool*) – If True, add bases for bases inserted into the reference and ‘N’s for base skipped from the reference. If a reference sequence is given, add the actual bases.

Returns list

Return type a list of bases/sequences per read at pileup column position.

n

deprecated – use *nsegments*

nsegments

number of reads mapping to this column.

Note that this number ignores the base quality filter.

pileups

list of reads (*pysam.PileupRead*) aligned to this column

pos
deprecated – use `reference_pos`

reference_id
the reference sequence number as defined in the header

reference_name
reference name (None if no `AlignmentFile` is associated)

reference_pos
the position in the reference sequence (0-based).

set_min_base_quality (*self*, *min_base_quality*)
set the minimum base quality for this pileup column.

tid
deprecated – use `reference_id`

class `pysam.PileupRead`

Representation of a read aligned to a particular position in the reference sequence.

alignment
a `pysam.AlignedSegment` object of the aligned read

indel
indel length for the position following the current pileup site.

This quantity peeks ahead to the next cigar operation in this alignment. If the next operation is an insertion, indel will be positive. If the next operation is a deletion, it will be negation. 0 if the next operation is not an indel.

is_del
1 iff the base on the padded read is a deletion

is_head
1 iff the base on the padded read is the left-most base.

is_refskip
1 iff the base on the padded read is part of CIGAR N op.

is_tail
1 iff the base on the padded read is the right-most base.

level
the level of the read in the “viewer” mode. Note that this value is currently not computed.

query_position
position of the read base at the pileup site, 0-based. None if `is_del` or `is_refskip` is set.

query_position_or_next
position of the read base at the pileup site, 0-based.
If the current position is a deletion, returns the next aligned base.

class `pysam.IndexedReads` (*AlignmentFile samfile*, *int multiple_iterators=True*)
**(AlignmentFile samfile, multiple_iterators=True)*

Index a Sam/BAM-file by query name while keeping the original sort order intact.

The index is kept in memory and can be substantial.

By default, the file is re-openend to avoid conflicts if multiple operators work on the same file. Set *multiple_iterators* = False to not re-open *samfile*.

Parameters

- **samfile** (*AlignmentFile*) – File to be indexed.
- **multiple_iterators** (*bool*) – Flag indicating whether the file should be reopened. Reopening prevents existing iterators being affected by the indexing.

build (*self*)
build the index.

find (*self*, *query_name*)
find *query_name* in index.

Returns Returns an iterator over all reads with *query_name*.

Return type *IteratorRowSelection*

Raises *KeyError* – if the *query_name* is not in the index.

Tabix files

TabixFile opens tabular files that have been indexed with *tabix*.

class *pysam.TabixFile*

Random access to bgzf formatted files that have been indexed by *tabix*.

The file is automatically opened. The index file of file <filename> is expected to be called <filename>.*tbi* by default (see parameter *index*).

Parameters

- **filename** (*string*) – Filename of bgzf file to be opened.
- **index** (*string*) – The filename of the index. If not set, the default is to assume that the index is called “filename.tbi”
- **mode** (*char*) – The file opening mode. Currently, only *r* is permitted.
- **parser** (*pysam.Parser*) – sets the default parser for this tabix file. If *parser* is *None*, the results are returned as an unparsed string. Otherwise, *parser* is assumed to be a functor that will return parsed data (see for example *asTuple* and *asGTF*).
- **encoding** (*string*) – The encoding passed to the parser
- **threads** (*integer*) – Number of threads to use for decompressing Tabix files. (Default=1)

Raises

- *ValueError* – if index file is missing.
- *IOError* – if file could not be opened

close (*self*)
closes the *pysam.TabixFile*.

contigs
list of chromosome names

fetch (*self*, *reference=None*, *start=None*, *end=None*, *region=None*, *parser=None*, *multiple_iterators=False*)
fetch one or more rows in a *region* using 0-based indexing. The region is specified by *reference*, *start* and *end*. Alternatively, a samtools *region* string can be supplied.

Without *reference* or *region* all entries will be fetched.

If only *reference* is set, all reads matching on *reference* will be fetched.

If *parser* is None, the default parser will be used for parsing.

Set *multiple_iterators* to true if you will be using multiple iterators on the same file at the same time. The iterator returned will receive its own copy of a filehandle to the file effectively re-opening the file. Re-opening a file creates some overhead, so beware.

header

the file header.

The file header consists of the lines at the beginning of a file that are prefixed by the comment character #.

Note: The header is returned as an iterator presenting lines without the newline character.

To iterate over tabix files, use `tabix_iterator()`:

`pysam.tabix_iterator(infile, parser)`
return an iterator over all entries in a file.

Results are returned parsed as specified by the *parser*. If *parser* is None, the results are returned as an unparsed string. Otherwise, *parser* is assumed to be a functor that will return parsed data (see for example `asTuple` and `asGTF`).

`pysam.tabix_compress(filename_in, filename_out, force=False)`
compress *filename_in* writing the output to *filename_out*.

Raise an IOError if *filename_out* already exists, unless *force* is set.

`pysam.tabix_index(filename, force=False, seq_col=None, start_col=None, end_col=None, preset=None, meta_char='#', int_line_skip=0, zerobased=False, int_min_shift=-1, index=None, keep_original=False, csi=False)`
index tab-separated *filename* using tabix.

An existing index will not be overwritten unless *force* is set.

The index will be built from coordinates in columns *seq_col*, *start_col* and *end_col*.

The contents of *filename* have to be sorted by contig and position - the method does not check if the file is sorted.

Column indices are 0-based. Note that this is different from the tabix command line utility where column indices start at 1.

Coordinates in the file are assumed to be 1-based unless *zerobased* is set.

If *preset* is provided, the column coordinates are taken from a preset. Valid values for preset are “gff”, “bed”, “sam”, “vcf”, “psltbl”, “pileup”.

Lines beginning with *meta_char* and the first *line_skip* lines will be skipped.

If *filename* is not detected as a gzip file it will be automatically compressed. The original file will be removed and only the compressed file will be retained.

min-shift sets the minimal interval size to $1 << \text{INT}$; 0 for the old tabix index. The default of -1 is changed inside htslib to the old tabix default of 0.

index controls the filename which should be used for creating the index. If not set, the default is to append `.tbi` to *filename*.

If *csi* is set, create a CSI index, the default is to create a TBI index.

When automatically compressing files, if *keep_original* is set the uncompressed file will not be deleted.

returns the filename of the compressed data

class pysam.asTuple

converts a *tabix row* into a python tuple.

A field in a row is accessed by numeric index.

class pysam.asVCF

converts a *tabix row* into a VCF record with the following fields:

Column	Field	Contents
1	contig	chromosome
2	pos	chromosomal position, zero-based
3	id	id
4	ref	reference allele
5	alt	alternate alleles
6	qual	quality
7	filter	filter
8	info	info
9	format	format specifier.

Access to genotypes is via index:

```
contig = vcf.contig
first_sample_genotype = vcf[0]
second_sample_genotype = vcf[1]
```

class pysam.asBed

converts a *tabix row* into a bed record with the following fields:

Column	Field	Contents
1	contig	contig
2	start	genomic start coordinate (zero-based)
3	end	genomic end coordinate plus one (zero-based)
4	name	name of feature.
5	score	score of feature
6	strand	strand of feature
7	thickStart	thickStart
8	thickEnd	thickEnd
9	itemRGB	itemRGB
10	blockCount	number of blocks
11	blockSizes	',' separated string of block sizes
12	blockStarts	',' separated string of block genomic start positions

Only the first three fields are required. Additional fields are optional, but if one is defined, all the preceding need to be defined as well.

class pysam.asGTF

converts a *tabix row* into a GTF record with the following fields:

<i>Column</i>	<i>Name</i>	<i>Content</i>
1	contig	the chromosome name
2	feature	The feature type
3	source	The feature source
4	start	genomic start coordinate (0-based)
5	end	genomic end coordinate (0-based)
6	score	feature score
7	strand	strand
8	frame	frame
9	attributes	the attribute field

GTF formatted entries also define the following fields that are derived from the attributes field:

<i>Name</i>	<i>Content</i>
gene_id	the gene identifier
transcript_id	the transcript identifier

Fasta files

`class pysam.FastaFile`

Random access to fasta formatted files that have been indexed by *faidx*.

The file is automatically opened. The index file of file <filename> is expected to be called <filename>.fai.

Parameters

- **filename** (*string*) – Filename of fasta file to be opened.
- **filepath_index** (*string*) – Optional, filename of the index. By default this is the filename + “.fai”.
- **filepath_index_compressed** (*string*) – Optional, filename of the index if fasta file is. By default this is the filename + “.gzi”.

Raises

- `ValueError` – if index file is missing
- `IOError` – if file could not be opened

close (*self*)
close the file.

closed
“bool indicating the current state of the file object. This is a read-only attribute; the close() method changes the value.

fetch (*self*, *reference=None*, *start=None*, *end=None*, *region=None*)
fetch sequences in a *region*.

A region can either be specified by *reference*, *start* and *end*. *start* and *end* denote 0-based, half-open intervals.

Alternatively, a samtools *region* string can be supplied.

If any of the coordinates are missing they will be replaced by the minimum (*start*) or maximum (*end*) coordinate.

Note that region strings are 1-based, while *start* and *end* denote an interval in python coordinates. The region is specified by *reference*, *start* and *end*.

Returns string

Return type a string with the sequence specified by the region.

Raises

- `IndexError` – if the coordinates are out of range
- `ValueError` – if the region is invalid

filename

filename associated with this object. This is a read-only attribute.

get_reference_length (*self*, *reference*)

return the length of reference.

is_open (*self*)

return true if samfile has been opened.

lengths

tuple with the lengths of *reference* sequences.

nreferences

“int with the number of *reference* sequences in the file. This is a read-only attribute.

references

tuple with the names of *reference* sequences.

Fastq files

class `pysam.FastxFile`

Stream access to fasta or fastq formatted files.

The file is automatically opened.

Entries in the file can be both fastq or fasta formatted or even a mixture of the two.

This file object permits iterating over all entries in the file. Random access is not implemented. The iteration returns objects of type *FastqProxy*

Parameters

- **filename** (*string*) – Filename of fasta/fastq file to be opened.
- **persist** (*bool*) – If True (default) make a copy of the entry in the file during iteration. If set to False, no copy will be made. This will permit much faster iteration, but an entry will not persist when the iteration continues and an entry is read-only.

Notes

Prior to version 0.8.2, this class was called FastqFile.

Raises `IOError` – if file could not be opened

Examples

```
>>> with pysam.FastxFile(filename) as fh:
...     for entry in fh:
...         print(entry.name)
...         print(entry.sequence)
...         print(entry.comment)
...         print(entry.quality)
>>> with pysam.FastxFile(filename) as fin, open(out_filename, mode='w') as fout:
...     for entry in fin:
...         fout.write(str(entry))
```

close (*self*)
close the file.

closed
“bool indicating the current state of the file object. This is a read-only attribute; the close() method changes the value.

filename
string with the filename associated with this object.

is_open (*self*)
return true if samfile has been opened.

next

class pysam.FastqProxy
A single entry in a fastq file.

get_quality_array (*self*, *int offset=33*) → array
return quality values as integer array after subtracting offset.

name
The name of each entry in the fastq file.

quality
The quality score of each entry in the fastq file, represented as a string.

sequence
The sequence of each entry in the fastq file.

VCF files

class pysam.VariantFile (*args, **kwargs)
(filename, mode=None, index_filename=None, header=None, drop_samples=False, duplicate_filehandle=True, ignore_truncation=False, threads=1)

A *VCF/BCF* formatted file. The file is automatically opened.

If an index for a variant file exists (.csi or .tbi), it will be opened automatically. Without an index random access to records via *fetch()* is disabled.

For writing, a *VariantHeader* object must be provided, typically obtained from another *VCF* file/*BCF* file.

Parameters

- **mode** (*string*) – *mode* should be *r* for reading or *w* for writing. The default is text mode (*VCF*). For binary (*BCF*) I/O you should append *b* for compressed or *u* for uncompressed *BCF* output.

If `b` is present, it must immediately follow `r` or `w`. Valid modes are `r`, `w`, `wh`, `rb`, `wb`, `wbu` and `wb0`. For instance, to open a *BCF* formatted file for reading, type:

```
f = pysam.VariantFile('ex1.bcf', 'r')
```

If mode is not specified, we will try to auto-detect the file type. All of the following should work:

```
f1 = pysam.VariantFile('ex1.bcf')
f2 = pysam.VariantFile('ex1.vcf')
f3 = pysam.VariantFile('ex1.vcf.gz')
```

- **index_filename** (*string*) – Explicit path to an index file.
- **header** (*VariantHeader*) – *VariantHeader* object required for writing.
- **drop_samples** (*bool*) – Ignore sample information when reading.
- **duplicate_filehandle** (*bool*) – By default, file handles passed either directly or through File-like objects will be duplicated before passing them to `htslib`. The duplication prevents issues where the same stream will be closed by `htslib` and through destruction of the high-level python object. Set to `False` to turn off duplication.
- **ignore_truncation** (*bool*) – Issue a warning, instead of raising an error if the current file appears to be truncated due to a missing EOF marker. Only applies to bgzipped formats. (Default=`False`)
- **threads** (*integer*) – Number of threads to use for compressing/decompressing VCF/BCF files. Setting threads to `> 1` cannot be combined with *ignore_truncation*. (Default=`1`)

close (*self*)
closes the *pysam.VariantFile*.

copy (*self*)

fetch (*self*, *contig=None*, *start=None*, *stop=None*, *region=None*, *reopen=False*, *end=None*, *reference=None*)
fetch records in a *region* using 0-based indexing. The region is specified by *contig*, *start* and *end*. Alternatively, a samtools *region* string can be supplied.

Without *contig* or *region* all mapped records will be fetched. The records will be returned ordered by *contig*, which will not necessarily be the order within the file.

Set *reopen* to true if you will be using multiple iterators on the same file at the same time. The iterator returned will receive its own copy of a filehandle to the file effectively re-opening the file. Re-opening a file incurs some overhead, so use with care.

If only *contig* is set, all records on *contig* will be fetched. If both *region* and *contig* are given, an exception is raised.

Note that a bgzipped *VCF.gz* file without a tabix/CSI index (*.tbi/.csi*) or a *BCF* file without a CSI index can only be read sequentially.

get_reference_name (*self*, *tid*)
return *reference* name corresponding to numerical *tid*

get_tid (*self*, *reference*)
return the numerical *tid* corresponding to *reference*
returns -1 if reference is not known.

is_valid_tid (*self*, *tid*)
return True if the numerical *tid* is valid; False otherwise.
returns -1 if reference is not known.

new_record (*self*, **args*, ***kwargs*)
Create a new empty *VariantRecord*.
See *VariantHeader.new_record()*

next

open (*self*, *filename*, *mode*='r', *index_filename*=None, *VariantHeader header*=None, *drop_samples*=False, *duplicate_filehandle*=True, *ignore_truncation*=False, *threads*=1)
open a vcf/bcf file.

If open is called on an existing *VariantFile*, the current file will be closed and a new file will be opened.

reset (*self*)
reset file position to beginning of file just after the header.

subset_samples (*self*, *include_samples*)
Read only a subset of samples to reduce processing time and memory. Must be called prior to retrieving records.

write (*self*, *VariantRecord record*) → int
write a single *pysam.VariantRecord* to disk.
returns the number of bytes written.

class *pysam.VariantHeader*
header information for a *VariantFile* object

add_line (*self*, *line*)
Add a metadata line to this header

add_meta (*self*, *key*, *value*=None, *items*=None)
Add metadata to this header

add_record (*self*, *VariantHeaderRecord record*)
Add an existing *VariantHeaderRecord* to this header

add_sample (*self*, *name*)
Add a new sample to this header

alts
alt metadata (dict ID->record).

The data returned just a snapshot of alt records, is created every time the property is requested, and modifications will not be reflected in the header metadata and vice versa.

i.e. it is just a dict that reflects the state of alt records at the time it is created.

contigs
contig information (*VariantHeaderContigs*)

copy (*self*)

filters
filter metadata (*VariantHeaderMetadata*)

formats
format metadata (*VariantHeaderMetadata*)

info
info metadata (*VariantHeaderMetadata*)

merge (*self*, *VariantHeader header*)

new_record (*self*, *contig=None*, *start=0*, *stop=0*, *alleles=None*, *id=None*, *qual=None*, *filter=None*,
info=None, *samples=None*, ***kwargs*)

Create a new empty VariantRecord.

Arguments are currently experimental. Use with caution and expect changes in upcoming releases.

records

header records (*VariantHeaderRecords*)

samples

version

VCF version

class *pysam.VariantRecord* (**args*, ***kwargs*)

Variant record

alleles

tuple of reference allele followed by alt alleles

alts

tuple of alt alleles

chrom

chromosome/contig name

contig

chromosome/contig name

copy (*self*)

return a copy of this VariantRecord object

filter

filter information (see *VariantRecordFilter*)

format

sample format metadata (see *VariantRecordFormat*)

id

record identifier or None if not available

info

info data (see *VariantRecordInfo*)

pos

record start position on chrom/contig (1-based inclusive)

qual

phred scaled quality score or None if not available

ref

reference allele

rid

internal reference id number

rlen

record length on chrom/contig (aka *rec.stop* - *rec.start*)

samples

sample data (see *VariantRecordSamples*)

start
record start position on chrom/contig (0-based inclusive)

stop
record stop position on chrom/contig (0-based exclusive)

translate (*self*, *VariantHeader dst_header*)

class `pysam.VariantHeaderRecord` (**args*, ***kwargs*)
header record from a *VariantHeader* object

attrs
sequence of additional header attributes

get (*self*, *key*, *default=None*)
D.get(k,d) -> D[k] if k in D, else d. d defaults to None.

items (*self*)
D.items() -> list of D's (key, value) pairs, as 2-tuples

iteritems (*self*)
D.iteritems() -> an iterator over the (key, value) items of D

iterkeys (*self*)
D.iterkeys() -> an iterator over the keys of D

itervalues (*self*)
D.itervalues() -> an iterator over the values of D

key
header key (the part before '=', in FILTER/INFO/FORMAT/contig/fileformat etc.)

keys (*self*)
D.keys() -> list of D's keys

pop (*self*, *key*, *default=_nothing*)

remove (*self*)

type
header type – FILTER, INFO, FORMAT, CONTIG, STRUCTURED, or GENERIC

update (*self*, *items=None*, ***kwargs*)
D.update([E,]**F) -> None.
Update D from dict/iterable E and F.

value
header value. Set only for generic lines, None for FILTER/INFO, etc.

values (*self*)
D.values() -> list of D's values

HTSFile

HTSFile is the base class for *pysam.AlignmentFile* and *pysam.VariantFile*.

class `pysam.HTSFile`
Base class for HTS file types

add_hts_options (*self*, *format_options=None*)
Given a list of key=value format option strings, add them to an open htsFile

category

General file format category. One of UNKNOWN, ALIGNMENTS, VARIANTS, INDEX, REGIONS

check_truncation (*self*, *ignore_truncation=False*)

Check if file is truncated.

close (*self*)**closed**

return True if HTSFile is closed.

compression

File compression.

One of NONE, GZIP, BGZF, CUSTOM.

description

Vaguely human readable description of the file format

format

File format.

One of UNKNOWN, BINARY_FORMAT, TEXT_FORMAT, SAM, BAM, BAI, CRAM, CRAI, VCF, BCF, CSI, GZI, TBI, BED.

get_reference_name (*self*, *tid*)

return *contig* name corresponding to numerical *tid*

get_tid (*self*, *contig*)

return the numerical *tid* corresponding to *contig*

returns -1 if contig is not known.

is_bam

return True if HTSFile is reading or writing a BAM alignment file

is_bcf

return True if HTSFile is reading or writing a BCF variant file

is_closed

return True if HTSFile is closed.

is_cram

return True if HTSFile is reading or writing a BAM alignment file

is_open

return True if HTSFile is open and in a valid state.

is_read

return True if HTSFile is open for reading

is_sam

return True if HTSFile is reading or writing a SAM alignment file

is_valid_reference_name (*self*, *contig*)

return True if the contig name *contig* is valid; False otherwise.

is_valid_tid (*self*, *tid*)

return True if the numerical *tid* is valid; False otherwise.

returns -1 if contig is not known.

is_vcf

return True if HTSFile is reading or writing a VCF variant file

is_write

return True if HTSFile is open for writing

parse_region (*self*, *contig=None*, *start=None*, *stop=None*, *region=None*, *tid=None*, *reference=None*, *end=None*)

parse alternative ways to specify a genomic region. A region can either be specified by *contig*, *start* and *stop*. *start* and *stop* denote 0-based, half-open intervals. *reference* and *end* are also accepted for backward compatibility as synonyms for *contig* and *stop*, respectively.

Alternatively, a samtools *region* string can be supplied.

If any of the coordinates are missing they will be replaced by the minimum (*start*) or maximum (*stop*) coordinate.

Note that region strings are 1-based inclusive, while *start* and *stop* denote an interval in 0-based, half-open coordinates (like BED files and Python slices).

If *contig* or *region* or are *, unmapped reads at the end of a BAM file will be returned. Setting either to . will iterate from the beginning of the file.

Returns

- **tuple** (a tuple of *flag*, *tid*, *start* and
- *stop*. The flag indicates whether no coordinates were
- *supplied and the genomic region is the complete genomic space*.

Raises `ValueError` – for invalid or out of bounds regions.

reset (*self*)

reset file position to beginning of file just after the header.

Returns

Return type The file position after moving the file pointer.

seek (*self*, *uint64_t offset*)

move file pointer to position *offset*, see `pysam.HTSFile.tell()`.

tell (*self*)

return current file position, see `pysam.HTSFile.seek()`.

version

Tuple of file format version numbers (major, minor)

1.2 Working with BAM/CRAM/SAM-formatted files

1.2.1 Opening a file

To begin with, import the pysam module and open a `pysam.AlignmentFile`:

```
import pysam
samfile = pysam.AlignmentFile("ex1.bam", "rb")
```

The above command opens the file `ex1.bam` for reading. The `b` qualifier indicates that this is a *BAM* file. To open a *SAM* file, type:

```
import pysam
samfile = pysam.AlignmentFile("ex1.sam", "r")
```

CRAM files are identified by a `c` qualifier:

```
import pysam
samfile = pysam.AlignmentFile("ex1.cram", "rc")
```

1.2.2 Fetching reads mapped to a region

Reads are obtained through a call to the `pysam.AlignmentFile.fetch()` method which returns an iterator. Each call to the iterator will return a `pysam.AlignedSegment` object:

```
iter = samfile.fetch("seq1", 10, 20)
for x in iter:
    print (str(x))
```

`pysam.AlignmentFile.fetch()` returns all reads overlapping a region sorted by the first aligned base in the *reference* sequence. Note that it will also return reads that are only partially overlapping with the *region*. Thus the reads returned might span a region that is larger than the one queried.

1.2.3 Using the pileup-engine

In contrast to *fetching*, the *pileup* engine returns for each base in the *reference* sequence the reads that map to that particular position. In the typical view of reads stacking vertically on top of the reference sequence similar to a multiple alignment, *fetching* iterates over the rows of this implied multiple alignment while a *pileup* iterates over the columns.

Calling `pileup()` will return an iterator over each *column* (reference base) of a specified *region*. Each call to the iterator returns an object of the type `pysam.PileupColumn` that provides access to all the reads aligned to that particular reference position as well as some additional information:

```
iter = samfile.pileup('seq1', 10, 20)
for x in iter:
    print (str(x))
```

1.2.4 Creating BAM/CRAM/SAM files from scratch

The following example shows how a new *BAM* file is constructed from scratch. The important part here is that the `pysam.AlignmentFile` class needs to receive the sequence identifiers. These can be given either as a dictionary in a header structure, as lists of names and sizes, or from a template file. Here, we use a header dictionary:

```
header = { 'HD': {'VN': '1.0'},
           'SQ': [{'LN': 1575, 'SN': 'chr1'},
                  {'LN': 1584, 'SN': 'chr2'}] }

with pysam.AlignmentFile(tmpfilename, "wb", header=header) as outf:
    a = pysam.AlignedSegment()
    a.query_name = "read_28833_29006_6945"
    a.query_sequence="AGCTTAGCTAGCTACCTATATCTTGGTCTTGCCG"
    a.flag = 99
    a.reference_id = 0
    a.reference_start = 32
    a.mapping_quality = 20
    a.cigar = ((0,10), (2,1), (0,25))
    a.next_reference_id = 0
```

(continues on next page)

(continued from previous page)

```
a.next_reference_start=199
a.template_length=167
a.query_qualities = pysam.qualitystring_to_array("<<<<<<<<<<<<<<<<<<<<<<<:<9/,&,22;;
↪<<<<")
a.tags = (("NM", 1),
          ("RG", "L1"))
outf.write(a)
```

1.2.5 Using streams

Pysam does not support reading and writing from true python file objects, but it does support reading and writing from stdin and stdout. The following example reads from stdin and writes to stdout:

```
infile = pysam.AlignmentFile("-", "r")
outfile = pysam.AlignmentFile("-", "w", template=infile)
for s in infile:
    outfile.write(s)
```

It will also work with *BAM* files. The following script converts a *BAM* formatted file on stdin to a *SAM* formatted file on stdout:

```
infile = pysam.AlignmentFile("-", "rb")
outfile = pysam.AlignmentFile("-", "w", template=infile)
for s in infile:
    outfile.write(s)
```

Note that the file open mode needs to be changed from `r` to `rb`.

1.3 Using samtools commands within python

Commands available in *csamtools* are available as simple function calls. Command line options are provided as arguments. For example:

```
pysam.sort("-o", "output.bam", "ex1.bam")
```

corresponds to the command line:

```
samtools sort -o output.bam ex1.bam
```

Or for example:

```
pysam.sort("-m", "1000000", "-o", "output.bam", "ex1.bam")
```

In order to get usage information, try:

```
print(pysam.sort.usage())
```

Argument errors raise a `pysam.SamtoolsError`:

```
pysam.sort()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
File "x.py", line 12, in <module>
    pysam.sort()
File "/build/lib.linux-x86_64-2.6/pysam/__init__.py", line 37, in __call__
    if retval: raise SamtoolsError( "\n".join( stderr ) )
pysam.SamtoolsError: 'Usage: samtools sort [-n] [-m <maxMem>] <in.bam> <out.prefix>\n'
```

Messages from *csamtools* on stderr are captured and are available using the `getMessages()` method:

```
pysam.sort.getMessage()
```

Note that only the output from the last invocation of a command is stored.

In order for pysam to make the output of samtools commands accessible the stdout stream needs to be redirected. This is the default behaviour, but can cause problems in environments such as the ipython notebook. A solution is to pass the `catch_stdout` keyword argument:

```
pysam.sort(catch_stdout=False)
```

Note that this means that output from commands which produce output on stdout will not be available. The only solution is to run samtools commands through subprocess.

1.4 Working with tabix-indexed files

To open a tabular file that has been indexed with *tabix*, use *TabixFile*:

```
import pysam
tbx = pysam.TabixFile("example.bed.gz")
```

Similar to *fetch*, intervals within a region can be retrieved by calling *fetch()*:

```
for row in tbx.fetch("chr1", 1000, 2000):
    print (str(row))
```

This will return a tuple-like data structure in which columns can be retrieved by numeric index:

```
for row in tbx.fetch("chr1", 1000, 2000): print ("chromosome is", row[0])
```

By providing a parser to *fetch* or *TabixFile*, the data will be presented in parsed form:

```
for row in tbx.fetch("chr1", 1000, 2000, parser=pysam.asTuple()):
    print ("chromosome is", row.contig)
    print ("first field (chrom)=", row[0])
```

Pre-built parsers are available for bed (*asBed*) formatted files and gtf (*asGTF*) formatted files. Thus, additional fields become available through named access, for example:

```
for row in tbx.fetch("chr1", 1000, 2000, parser=pysam.asBed()):
    print ("name is", row.name)
```

1.5 Working with VCF/BCF formatted files

To iterate through a VCF/BCF formatted file use *VariantFile*:

```

from pysam import VariantFile

bcf_in = VariantFile("test.bcf") # auto-detect input format
bcf_out = VariantFile('-', 'w', header=bcf_in.header)

for rec in bcf_in.fetch('chr1', 100000, 200000):
    bcf_out.write(rec)

```

`_pysam.VariantFile.fetch()` iterates over *VariantRecord* objects which provides access to simple variant attributes such as *contig*, *pos*, *ref*:

```

for rec in bcf_in.fetch():
    print (rec.pos)

```

but also to complex attributes such as the contents to the info, format and genotype columns. These complex attributes are views on the underlying htslib data structures and provide dictionary-like access to the data:

```

for rec in bcf_in.fetch():
    print (rec.info)
    print (rec.info.keys())
    print (rec.info["DP"])

```

The header attribute (*VariantHeader*) provides access information stored in the *vcf* header. The complete header can be printed:

```

>>> print (bcf_in.header)
##fileformat=VCFv4.2
##FILTER=<ID=PASS,Description="All filters passed">
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=1000GenomesPilot-NCBI36
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=.,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
##contig=<ID=M>
##contig=<ID=17>
##contig=<ID=20>
##bcftools_viewVersion=1.3+htslib-1.3
##bcftools_viewCommand=view -O b -o example_vcf42.bcf
example_vcf42.vcf.gz
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA00001_
->NA00002 NA0000

```

Individual contents such as contigs, info fields, samples, formats can be retrieved as attributes from header:

```
>>> print (bcf_in.header.contigs)
<pysam.cbcf.VariantHeaderContigs object at 0xf250f8>
```

To convert these views to native python types, iterate through the views:

```
>>> print list((bcf_in.header.contigs))
['M', '17', '20']
>>> print list((bcf_in.header.filters))
['PASS', 'q10', 's50']
>>> print list((bcf_in.header.info))
['NS', 'DP', 'AF', 'AA', 'DB', 'H2']
>>> print list((bcf_in.header.samples))
['NA00001', 'NA00002', 'NA00003']
```

Alternatively, it is possible to iterate through all records in the header returning objects of type *VariantHeaderRecord*:

```
>>> for x in bcf_in.header.records:
>>>     print (x)
>>>     print (x.type, x.key)
GENERIC fileformat
FILTER FILTER
GENERIC fileDate
GENERIC source
GENERIC reference
GENERIC phasing
INFO INFO
INFO INFO
INFO INFO
INFO INFO
INFO INFO
INFO INFO
FILTER FILTER
FILTER FILTER
FORMAT FORMAT
FORMAT FORMAT
FORMAT FORMAT
FORMAT FORMAT
CONTIG contig
CONTIG contig
CONTIG contig
GENERIC bcftools_viewVersion
GENERIC bcftools_viewCommand
```

1.6 Extending pysam

Using `pyximport`, it is (relatively) straight-forward to access pysam internals and the underlying samtools library. An example is provided in the `tests` directory. The example emulates the samtools `flagstat` command and consists of three files:

1. The main script `pysam_flagstat.py`. The important lines in this script are:

```
import pyximport
pyximport.install()
import _pysam_flagstat
```

(continues on next page)

(continued from previous page)

```
...  
flag_counts = _pysam_flagstat.count(pysam_in)
```

The first part imports, sets up `pyximport` and imports the cython module `_pysam_flagstat`. The second part calls the `count` method in `_pysam_flagstat`.

2. The cython implementation `_pysam_flagstat.pyx`. This script imports the pysam API via:

```
from pysam.calignmentfile cimport AlignmentFile, AlignedSegment
```

This statement imports, amongst others, `AlignedSegment` into the namespace. Speed can be gained from declaring variables. For example, to efficiently iterate over a file, an `AlignedSegment` object is declared:

```
# loop over samfile  
cdef AlignedSegment read  
for read in samfile:  
    ...
```

3. A `pyxbuild` providing `pyximport` with build information. Required are the locations of the samtools and pysam header libraries of a source installation of pysam plus the `csamtools.so` shared library. For example:

```
def make_ext(modname, pyxfilename):  
    from distutils.extension import Extension  
    import pysam  
    return Extension(name=modname,  
                     sources=[pyxfilename],  
                     extra_link_args=pysam.get_libraries(),  
                     include_dirs=pysam.get_include(),  
                     define_macros=pysam.get_defines())
```

If the script `pysam_flagstat.py` is called the first time, `pyximport` will compile the cython extension `_pysam_flagstat.pyx` and make it available to the script. Compilation requires a working compiler and cython installation. Each time `_pysam_flagstat.pyx` is modified, a new compilation will take place.

`pyximport` comes with `cython`.

1.7 Installing pysam

Pysam can be installed through `conda`, `pypi` and from the repository. The recommended way to install pysam is through `conda/bioconda`.

1.7.1 Conda installation

To install pysam in your current `conda` environment, type:

```
conda config --add channels r  
conda config --add channels bioconda  
conda install pysam
```

This will install pysam from the `bioconda` channel and automatically makes sure that dependencies are installed. Also, compilation flags will be set automatically, which will potentially save a lot of trouble on OS X.

1.7.2 Pypi installation

Pysam provides a python interface to the functionality contained within the [htslib](#) C library. There are two ways that these two can be combined, `builtin` and `external`.

Builtin

The typical installation will be through `pypi`:

```
pip install pysam
```

This will compile the `builtin` `htslib` source code within `pysam`.

`htslib` can be configured at compilation to turn on additional features such support using encrypted configurations, enable plugins, and more. See the [htslib](#) project for more information on these.

Pysam will attempt to configure `htslib` to turn on some advanced features. If these fail, for example due to missing library dependencies (*libcurl*, *libcrypto*), it will fall back to conservative defaults.

Options can be passed to the configure script explicitly by setting the environment variable `HTSLIB_CONFIGURE_OPTIONS`. For example:

```
export HTSLIB_CONFIGURE_OPTIONS=--enable-plugins
pip install pysam
```

External

`pysam` can be combined with an externally installed `htslib` library. This is a good way to avoid duplication of libraries. To link against an externally installed library, set the environment variables `HTSLIB_LIBRARY_DIR` and `HTSLIB_INCLUDE_DIR` before installing:

```
export HTSLIB_LIBRARY_DIR=/usr/local/lib
export HTSLIB_INCLUDE_DIR=/usr/local/include
pip install pysam
```

Note that the location of the file `libhts.so` needs to be known to the linker once you run `pysam`, for example by setting the environment-variable `LD_LIBRARY_PATH`.

Note that generally the `pysam` and `htslib` version need to be compatible. See the release notes for more information.

1.7.3 Installation from repository

`pysam` depends on `cython` to provide the connectivity to the `htslib` C library. The installation of the source tarball (`.tar.gz`) contains pre-built C-files and `cython` needs not be present during installation. However, when installing from the repository, `cython` needs to be installed beforehand.

To install from repository, type:

```
python setup.py install
```

For compilation options, see the section on Pypi installation above.

1.7.4 Requirements

Depending on the installation method, requirements for building pysam differ.

When installing through `conda`, dependencies will be resolved by the package manager. The `pip` installation and installation from source require a C compiler and its standard libraries as well as all requirements for building `htslib`. `Htslib` requirements are listed in the `htslib/INSTALL` file.

Installing from the repository will require `cython` to be installed.

1.8 FAQ

1.8.1 How should I cite pysam

Pysam has not been published in print. When referring pysam, please use the github URL: <https://github.com/pysam-developers/pysam>. As pysam is a wrapper around `htslib` and the `samtools` package, I suggest cite *Li et al (2009)* <<http://www.ncbi.nlm.nih.gov/pubmed/19505943>>.

1.8.2 Is pysam thread-safe?

Pysam is a mix of python and C code. Instructions within python are generally made thread-safe through python's `global interpreter lock (GIL)`. This ensures that python data structures will always be in a consistent state.

If an external function outside python is called, the programmer has a choice to keep the GIL in place or to release it. Keeping the GIL in place will make sure that all python threads wait until the external function has completed. This is a safe option and ensures thread-safety.

Alternatively, the GIL can be released while the external function is called. This will allow other threads to run concurrently. This can be beneficial if the external function is expected to halt, for example when waiting for data to read or write. However, to achieve thread-safety, the external function needs to be implemented with thread-safety in mind. This means that there can be no shared state between threads, or if there is shared, it needs to be controlled to prevent any access conflicts.

Pysam generally uses the latter option and aims to release the GIL for I/O intensive tasks. This is generally fine, but thread-safety of all parts have not been fully tested.

A related issue is when different threads read from the same file object - or the same thread uses two iterators over a file. There is only a single file-position for each opened file. To prevent this from happening, use the option `multiple_iterator=True` when calling a `fetch()` method. This will return an iterator on a newly opened file.

1.8.3 pysam coordinates are wrong

pysam uses 0-based coordinates and the half-open notation for ranges as does python. Coordinates and intervals reported from pysam always follow that convention.

Confusion might arise as different file formats might have different conventions. For example, the SAM format is 1-based while the BAM format is 0-based. It is important to remember that pysam will always conform to the python convention and translate to/from the file format automatically.

The only exception is the `region` string in the `fetch()` and `pileup()` methods. This string follows the convention of the `samtools` command line utilities. The same is true for any coordinates passed to the `samtools` command utilities directly, such as `pysam.mpileup()`.

1.8.4 Calling `pysam.fetch()` confuses existing iterators

The following code will cause unexpected behaviour:

```
samfile = pysam.AlignmentFile("pysam_ex1.bam", "rb")

iter1 = samfile.fetch("chr1")
print (iter1.next().reference_id)
iter2 = samfile.fetch("chr2")
print (iter2.next().reference_id)
print (iter1.next().reference_id)
```

This will give the following output:

```
0
1
Traceback (most recent call last):
  File "xx.py", line 8, in <module>
    print iter1.next().reference_id
  File "calignmentfile.pyx", line 1408, in
    pysam.calignmentfile.IteratorRowRegion.__next__
    (pysam/alignmentfile.c:16461)
StopIteration
```

Note how the second iterator stops as the file pointer has moved to chr2. The correct way to work with multiple iterators is:

```
samfile = pysam.AlignmentFile("pysam_ex1.bam", "rb")

iter1 = samfile.fetch("chr1", all)
print (iter1.next().reference_id)
iter2 = samfile.fetch("chr2")
print (iter2.next().reference_id)
print (iter1.next().reference_id)
```

Here, the output is:

```
0
1
0
```

The reason for this behaviour is that every iterator needs to keep track of its current position in the file. Within pysam, each opened file can only keep track of one file position and hence there can only be one iterator per file. Using the option `multiple_iterators=True` will return an iterator within a newly opened file. This iterator will not interfere with existing iterators as it has its own file handle associated with it.

Note that re-opening files incurs a performance penalty which can become severe when calling `fetch()` often. Thus, `multiple_iterators` is set to `False` by default.

1.8.5 `AlignmentFile.fetch` does not show unmapped reads

`fetch()` will only iterate over alignments in the SAM/BAM file. The following thus always works:

```
bf = pysam.AlignmentFile(fname, "rb")
for r in bf.fetch():
    assert not r.is_unmapped
```

If the SAM/BAM file contains unaligned reads, they can be included in the iteration by adding the `until_eof=True` flag:

```
bf = pysam.AlignmentFile(fname, "rb")
for r in bf.fetch(until_eof=True):
    if r.is_unmapped:
        print("read is unmapped")
```

1.8.6 I can't call `AlignmentFile.fetch` on a file without index

`fetch()` requires an index when iterating over a SAM/BAM file. To iterate over a file without index, use the `“until_eof=True”`:

```
bf = pysam.AlignmentFile(fname, "rb")
for r in bf.fetch(until_eof=True):
    print(r)
```

1.8.7 BAM files with a large number of reference sequences are slow

If you have many reference sequences in a bam file, the following might be slow:

```
track = pysam.AlignmentFile(fname, "rb")
for aln in track.fetch():
    pass
```

The reason is that `track.fetch()` will iterate through the bam file for each reference sequence in the order as it is defined in the header. This might require a lot of jumping around in the file. To avoid this, use:

```
track = pysam.AlignmentFile(fname, "rb")
for aln in track.fetch(until_eof=True):
    pass
```

This will iterate through reads as they appear in the file.

1.8.8 Weirdness with spliced reads in `samfile.pileup(chr,start,end)` given spliced alignments from an RNA-seq bam file

Spliced reads are reported within `samfile.pileup`. To ignore these in your analysis, test the flags `is_del == True` and `indel=0` in the `PileupRead` object.

1.8.9 I can't edit quality scores in place

Editing reads in-place generally works, though there is some quirk to be aware of. Assigning to `AlignedRead.seq` will invalidate any quality scores in `AlignedRead.qual`. The reason is that samtools manages the memory of the sequence and quality scores together and thus requires them to always be of the same length or 0.

Thus, to in-place edit the sequence and quality scores, copies of the quality scores need to be taken. Consider trimming for example:

```
q = read.qual
read.seq = read.seq[5:10]
read.qual = q[5:10]
```


1.8.10 Why is there no SNPCaller class anymore?

SNP calling is highly complex and heavily parameterized. There was a danger that the pysam implementations might show different behaviour from the samtools implementation, which would have caused a lot of confusion.

The best way to use samtools SNP calling from python is to use the `pysam.mpileup()` command and parse the output directly.

1.8.11 I get an error ‘PileupProxy accessed after iterator finished’

Pysam works by providing proxy objects to objects defined within the C-samtools package. Thus, some attention must be paid at the lifetime of objects. The following code snippets will cause an error:

```
s = AlignmentFile('ex1.bam')
for p in s.pileup('chr1', 1000, 1010):
    pass

for pp in p.pileups:
    print pp
```

The iteration has finished, thus the contents of `p` are invalid. A variation of this:

```
p = next(AlignmentFile('ex1.bam').pileup('chr1', 1000, 1010))
for pp in p.pileups:
    print pp
```

Again, the iteration finishes as the temporary iterator created by `pileup` goes out of scope. The solution is to keep a handle to the iterator that remains alive:

```
i = AlignmentFile('ex1.bam').pileup('chr1', 1000, 1010)
p = next(i)
for pp in p.pileups:
    print pp
```

1.8.12 Pysam won't compile

Compiling pysam can be tricky as there are numerous variables that differ between build environments such as OS, version, python version, and compiler. It is difficult to build software that build cleanly on all systems and the process might fail. Please see the [pysam user group](#) for common issues.

If there is a build issue, read the generated output carefully - generally the cause of the problem is among the first errors to be reported. For example, you will need to have the development version of python installed that includes the header files such as `Python.h`. If that file is missing, the compiler will report this at the very top of its error messages but will follow it with any unknown function or variable definition it encounters later on.

A general advice is to always use the latest version on [python](#) and [cython](#) when building pysam. There are some known incompatibilities:

- Python 3.4 requires cython 0.20.2 or later (see [here](#))

1.8.13 ImportError: cannot import name csamtools

In version 0.10.0 and onwards, all pysam extension modules contain a `lib-` prefix. This facilitates linking against pysam extension modules with compilers that require to start with `lib`. As a consequence, all code using pysam extension modules directly will need to be adapted. For example, for example:

```
cimport pysam.csamtools
```

will become:

```
cimport pysam.libcamtools
```

1.9 Developer's guide

1.9.1 Code organization

The top level directory is organized in the following directories:

pysam Code specific to pysam

doc The documentation. To build the latest documention type:

```
make -C doc html
```

tests Code and data for testing and benchmarking

htslib Source code from [htslib](#) shipped with pysam. See `import.py` about importing.

samtools Source code from [samtools](#) shipped with pysam. See `import.py` about importing.

bcftools Source code from [bcftools](#) shipped with pysam. See `import.py` about importing.

1.9.2 Importing new versions of htslib and samtools

See instructions in `import.py` to import the latest version of [htslib](#), [samtools](#) and [bcftools](#).

1.9.3 Unit testing

Unit tests are in the `tests` directory. To run all unit tests, run:

```
pytest tests
```

1.9.4 Benchmarking

To run the benchmarking suite, make sure that [pytest-benchmark](#) is installed. To run all benchmarks, type:

```
pytest tests/*_bench.py
```

See [Benchmarking](#) for more on this topic.

1.9.5 Contributors

Please see [github](#) for a list of all contributors:

<https://github.com/pysam-developers/pysam/graphs/contributors>

Many thanks to all contributors for helping in making pysam useful.

1.10 Release notes

1.10.1 Release 0.14.1

This is mostly a bugfix release, though bcftools has now also been upgraded to 1.7.0.

- [#621] Add a warning to count_coverage when an alignment has an empty QUAL field
- [#635] Speed-up of AlignedSegment.find_intro()
- treat border case of all bases in pileup column below quality score
- [#634] Fix access to pileup reference_sequence

1.10.2 Release 0.14.0

This release wraps htlib/samtools versions 1.7.0.

- SAM/BAM/CRAM headers are now managed by a separate AlignmentHeader class.
- AlignmentFile.header.as_dict() returns an ordered dictionary.
- Use “stop” instead of “end” to ensure consistency to VariantFile. The end designations have been kept for backwards compatibility.
- [#611] and [#293] CRAM repeated fetch now works, each iterator reloads index if multiple_iterators=True
- [#608] pysam now wraps htlib 1.7 and samtools 1.7.
- [#580] reference_name and next_reference_name can now be set to “*” (will be converted to None to indicate an unmapped location)
- [#302] providing no coordinate to count_coverage will not count from start/end of contig.
- [#325] @SQ records will be automatically added to header if they are absent from text section of header.
- [#529] add get_forward_sequence() and get_forward_qualities() methods
- [#577] add from_string() and to_dict()/from_dict() methods to AlignedSegment. Rename tostring() to to_string() throughout for consistency
- [#589] return None from build_alignment_sequence if no MD tag is set
- [#528] add PileupColumn.__len__ method

Backwards incompatible changes:

- AlignmentFile.header now returns an AlignmentHeader object. Use AlignmentFile.header.to_dict() to get the dictionary as previously. Most dictionary accessor methods (keys(), values(), __getitem__, ...) have been implemented to ensure some level of backwards compatibility when only reading.

The rationale for this change is to have consistency between AlignmentFile and VariantFile.

- AlignmentFile and FastFile now raise IOError instead of OSError

Medium term we plan to have a 1.0 release. The pysam interface has grown over the years and the API is cluttered with deprecated names (Samfile, getname(), gettid(), ...). To work towards this, the next release (0.15.0) will yield DeprecationWarnings for any parts of the API that are considered obsolete and will not be in 1.0. Once 1.0 has been reached, we will use semantic versioning.

1.10.3 Release 0.13.0

This release wraps htlib/samtools/bcftools versions 1.6.0 and contains a series of bugfixes.

- [#544] reading header from remote TabixFiles now works.
- [#531] add missing tag types H and A. A python float will now be added as 'f' type instead of 'd' type.
- [#543] use FastaFile instead of Fastafile in pileup.
- [#546] set is_modified flag in setAttribute so updated attributes are output.
- [#537] allow tabix index files to be created in a custom location.
- [#530] add get_index_statistics() method

1.10.4 Release 0.12.0.1

Bugfix release to solve compilation issue due to missing bcftools/config.h file.

1.10.5 Release 0.12.0

This release wraps htlib/samtools/bcftools versions 1.5.0 and contains a series of bugfixes.

- [#473] A new FastxRecord class that can be instantiated from class and modified in-place. Replaces PersistentFastqProxy.
- [#521] In AlignmentFile, Simplify file detection logic and allow remote index files * Removed attempts to guess data and index file names; this is magic left
 - to htlib.
 - Removed file existence check prior to opening files with htlib
 - Better error checking after opening files that raise the appropriate error (IOError for when errno is set, ValueError otherwise for backward compatibility).
 - Report IO errors when loading an index by name.
 - Allow remote indices (tested using S3 signed URLs).
 - Document filepath_index and make it an alias for index_filename.
 - Added a require_index parameter to AlignmentFile
- [#526] handle unset ref when creating new records
- [#513] fix bcf_translate to skip deleted FORMAT fields to avoid segfaults
- [#516] expose IO errors via IOError exceptions
- [#487] add tabix line_skip, remove 'pileup' preset
- add FastxRecord, replaces PersistentFastqProxy (still present for backwards compatibility)
- [#496] upgrade to htlib/samtools/bcftools versions 1.5
- add start/stop to AlignmentFile.fetch() to be consistent with VariantFile.fetch(). "end" is kept for backwards compatibility.
- [#512] add get_index_statistics() method to AlignmentFile.

Upcoming changes:

In the next release we are planning to separate the header information from `AlignmentFile` into a separate class `AlignmentHeader`. This layout is similar to `VariantFile/VariantHeader`. With this change we will ensure that an `AlignedSegment` record will be linked to a header so that chromosome names can be automatically translated from the numeric representation. As a consequence, the way new `AlignedSegment` records are created will need to change as the constructor requires a header:

```
header = pysam.AlignmentHeader(
    reference_names=["chr1", "chr2"],
    reference_lengths=[1000, 1000])

read = pysam.AlignedSegment(header)
```

This will affect all code that instantiates `AlignedSegment` objects directly. We have not yet merged to allow users to provide feedback. The pull-request is here: <https://github.com/pysam-developers/pysam/pull/518> Please comment on github.

1.10.6 Release 0.11.2.2

Bugfix release to address two issues:

- Changes in 0.11.2.1 broke the GTF/GFF3 parser. Corrected and more tests have been added.
- [#479] Correct `VariantRecord` edge cases described in issue

1.10.7 Release 0.11.2.1

Release to fix release tar-ball containing 0.11.1 pre-compiled C-files.

1.10.8 Release 0.11.2

This release wraps `htslib/samtools/bcftools` versions 1.4.1 in response to a security fix in these libraries. Additionally the following issues have been fixed:

- [#452] add GFF3 support for tabix parsers
- [#461] Multiple fixes related to `VariantRecordInfo` and handling of INFO/END
- [#447] limit query name to 251 characters (only partially addresses issue)

`VariantFile` and related object fixes

- Restore `VariantFile.__dealloc__`
- Correct handling of `bcf_str_missing` in `bcf_array_to_object` and `bcf_object_to_array`
- Added `update()` and `pop()` methods to some dict-like proxy objects
- scalar INFO entries could not be set again after being deleted
- `VariantRecordInfo.__delitem__` now allows unset flags to be deleted without raising a `KeyError`
- Multiple other fixes for `VariantRecordInfo` methods
- INFO/END is now accessible only via `VariantRecord.stop` and `VariantRecord.rlen`. Even if present behind the scenes, it is no longer accessible via `VariantRecordInfo`.
- Add argument to issue a warning instead of an exception if input appears to be truncated

Other features and fixes:

- Make AlignmentFile `__dealloc__` and close more stringent
- Add argument AlignmentFile to issue a warning instead of an exception if input appears to be truncated

1.10.9 Release 0.11.1

Bugfix release

- [#440] add deprecated ‘always’ option to `infer_query_length` for backwards compatibility.

1.10.10 Release 0.11.0

This release wraps the latest versions of htslib/samtools/bcftools and implements a few bugfixes.

- [#413] Wrap HTSlib/Samtools/BCFtools 1.4
- [#422] Fix missing `pysam.sort.usage()` message
- [#411] Fix BGZfile initialization bug
- [#412] Add seek support for BGZFile
- [#395] Make BGZfile iterable
- [#433] Correct `getQueryEnd`
- [#419] Export SAM enums such as `pysam.CMATCH`
- [#415] Fix access by tid in `AlignmentFile.fetch()`
- [#405] Writing SAM now outputs a header by default.
- [#332] split `infer_query_length(always)` into `infer_query_length` and `infer_read_length`

1.10.11 Release 0.10.0

This release implements further functionality in the VariantFile API and includes several bugfixes:

- treat special case `-c` option in samtools view outputs to stdout even if `-o` given, fixes #315
- permit reading BAM files with CSI index, closes #370
- raise Error if query name exceeds maximum length, fixes #373
- new method to compute hash value for `AlignedSegment`
- `AlignmentFile`, `VariantFile` and `TabixFile` all inherit from `HTSFile`
- Avoid segfault by detecting out of range `reference_id` and `next_reference` in `AlignedSegment.tostring`
- Issue #355: Implement streams using file descriptors for `VariantFile`
- upgrade to htslib 1.3.2
- fix compilation with musl libc
- Issue #316, #360: Rename all Cython modules to have lib as a prefix
- Issue #332, hardclipped bases in cigar included by `pysam.AlignedSegment.infer_query_length()`
- Added support for Python 3.6 filename encoding protocol

- Issue #371, fix incorrect parsing of scalar INFO and FORMAT fields in VariantRecord
- Issue #331, fix failure in VariantFile.reset() method
- Issue #314, add VariantHeader.new_record(), VariantFile.new_record() and VariantRecord.copy() methods to create new VariantRecord objects
- Added VariantRecordFilter.add() method to allow setting new VariantRecord filters
- Preliminary (potentially unsafe) support for removing and altering header metadata
- Many minor fixes and improvements to VariantFile and related objects

Please note that all internal cython extensions now have a lib prefix to facilitate linking against pysam extension modules. Any user cython extensions using cimport to import pysam definitions will need changes, for example:

```
cimport pysam.csamtools
```

will become:

```
cimport pysam.libcamtools
```

1.10.12 Release 0.9.1

This is a bugfix release addressing some installation problems in pysam 0.9.0, in particular:

- patch included htlib to work with older libcurl versions, fixes #262.
- do not require cython for python 3 install, fixes #260
- FastFile does not accept filepath_index any more, see #270
- add AlignedSegment.get_cigar_stats method.
- py3 bugfix in VariantFile.subset_samples, fixes #272
- add missing sysconfig import, fixes #278
- do not redirect stdout, but instead write to a separately created file. This should resolve issues when pysam is used in notebooks or other environments that redirect stdout.
- wrap htlib-1.3.1, samtools-1.3.1 and bcftools-1.3.1
- use bgzf throughout instead of gzip
- allow specifying a fasta reference for CRAM file when opening for both read and write, fixes #280

1.10.13 Release 0.9.0

Overview

The 0.9.0 release upgrades htlib to htlib 1.3 and numerous other enhancements and bugfixes. See below for a detailed list.

Htlib 1.3 comes with additional capabilities for remote file access which depend on the presence of optional system libraries. As a consequence, the installation script `setup.py` has become more complex. For an overview, see [Installing pysam](#). We have tested installation on linux and OS X, but could not capture all variations. It is possible that a 0.9.1 release might follow soon addressing installation issues.

The *VariantFile* class provides access to *vcf* and *bcf* formatted files. The class is certainly usable and interface is reaching completion, but the API and the functionality is subject to change.

Detailed release notes

- upgrade to htlib 1.3
- python 3 compatibility tested throughout.
- added a first set of bcftools commands in the pysam.bcftools submodule.
- samtools commands are now in the pysam.samtools module. For backwards compatibility they are still imported into the pysam namespace.
- samtools/bcftools return stdout as a single (byte) string. As output can be binary (VCF.gz, BAM) this is necessary to ensure py2/py3 compatibility. To replicate the previous behaviour in py2.7, use:

```
pysam.samtools.view(self.filename).splitlines(True)
```

- get_tags() returns the tag type as a character, not an integer (#214)
- TabixFile now raises ValueError on indices created by tabix <1.0 (#206)
- improve OSX installation and develop mode
- FastxIterator now handles empty sequences (#204)
- TabixFile.isremote is not TabixFile.is_remote in line with AlignmentFile
- AlignmentFile.count() has extra optional argument read_callback
- **setup.py has been changed to:**
 - install a single builtin htlib library. Previously, each pysam module contained its own version. This reduces compilation time and code bloat.
 - run configure for the builtin htlib library in order to detect optional libraries such as libcurl. Configure behaviour can be controlled by setting the environment variable HTSLIB_CONFIGURE_OPTIONS.
- get_reference_sequence() now returns the reference sequence and not something looking like it. This bug had effects on get_aligned_pairs(with_seq=True), see #225. If you have relied on on get_aligned_pairs(with_seq=True) in pysam-0.8.4, please check your results.
- improved autodetection of file formats in AlignmentFile and VariantFile.

1.10.14 Release 0.8.4

This release contains numerous bugfixes and a first implementation of a pythonic interface to VCF/BCF files. Note that this code is still incomplete and preliminary, but does offer a nearly complete immutable Pythonic interface to VCF/BCF metadata and data with reading and writing capability.

Potential issues when upgrading from v0.8.3:

- binary tags are now returned as python arrays
- renamed several methods for pep8 compatibility, old names still retained for backwards compatibility, but should be considered deprecated.
 - gettid() is now get_tid()
 - getname() is now get_reference_name()
 - parseRegion() is now parse_region()
- some methods have changed for pep8 compatibility without the old names being present:
 - fromQualityString() is now qualitystring_to_array()

- `toQualityString()` is now `qualities_to_qualitystring()`
- `faidx` now returns strings and not binary strings in py3.
- The cython components have been broken up into smaller files with more specific content. This will affect users using the cython interfaces.

Edited list of commit log changes:

- fixes `AlignmentFile.check_index` to return `True`
- add RG/PM header tag - closes #179
- add `with_seq` option to `get_aligned_pairs`
- use `char *` inside `reconstituteReferenceSequence`
- add soft clipping for `get_reference_sequence`
- add `get_reference_sequence`
- `queryEnd` now computes length from cigar string if no sequence present, closes #176
- tolerate missing space at end of gtf files, closes #162
- do not raise `Error` when receiving output on `stderr`
- add docu about fetching without index, closes #170
- `FastaFile` and `FastxFile` now return strings in python3, closes #173
- py3 compat: relative -> absolute imports.
- add `reference_name` and `next_reference_name` attributes to `AlignedSegment`
- add function signatures to `cvcf` cython. Added note about other VCF code.
- add context manager functions to `FastaFile`
- add `reference_name` and `next_reference_name` attributes to `AlignedSegment`
- `PileupColumn` also gets a `reference_name` attribute.
- add context manager functions to `FastaFile`
- `TabixFile.header` for remote files raises `AttributeError`, fixes #157
- add context manager interface to `TabixFile`, closes #165
- change `ctypedef enum` to `typedef enum` for cython 0.23
- add function signatures to `cvcf` cython, also added note about other VCF code
- remove exception for custom upper-case header record tags.
- rename `VALID_HEADER_FIELDS` to `KNOWN_HEADER_FIELDS`
- fix header record tag parsing for custom tags.
- use `cython.str` in `count_coverage`, fixes #141
- avoid `maketrans` (issues with python3)
- refactoring: `AlignedSegment` now in separate module
- do not execute remote tests if URL not available
- fix the unmapped count, incl reads with no SQ group
- add raw output to tags

- added write access for binary tags
- bugfix in call to resize
- implemented writing of binary tags from arrays
- implemented `convert_binary_tag` to use arrays
- add special cases for reads that are unmapped or whose mates are unmapped.
- rename `TabProxies` to `ctabixproxies`
- remove underscores from utility functions
- move utility methods into `cutils`
- remove callback argument to `fetch` - closes #128
- avoid calling `close` in `dealloc`
- add unit tests for `File` object opening
- change `AlignmentFile.open` to `filepath_or_object`
- implement `copy.copy`, close #65
- add caching of array attributes in `AlignedSegment`, closes #121
- add export of `FastqFile`
- remove superfluous `pysam_dispatch`
- use `persist` option in `FastqFile`
- `get_tag`: expose tag type if requested with *with_value_type*
- fix to allow reading vcf record info via tabix-based vcf reader
- add `pFastqProxy` and `pFastqFile` objects to make it possible to work with multiple fastq records per file handle, unlike `FastqProxy/FastqFile`.
- release GIL around `htslib` IO operations
- More work on read/write support, API improvements
- add *phased* property on `VariantRecordSample`
- add mutable properties to `VariantRecord`
- BCF fixes and start of read/write support
- `VariantHeaderRecord` objects now act like mappings for attributes.
- add `VariantHeader.alts` dict from alt ID->Record.
- Bug fix to strong representation of structured header records.
- `VariantHeader` is now mutable

1.10.15 Release 0.8.3

- `samtools` command now accept the “`catch_stdout`” option.
- `get_aligned_pairs` now works for soft-clipped reads.
- `query_position` is now `None` when a `PileupRead` is not aligned to a particular position.
- `AlignedSegments` are now comparable and hashable.

1.10.16 Release 0.8.2.1

- Installation bugfix release.

1.10.17 Release 0.8.2

- Pysam now wraps htslib 1.2.1 and samtools version 1.2.
- Added CRAM file support to pysam.
- **New alignment info interface.**
 - `opt()` and `setTag` are deprecated, use `get_tag()` and `set_tag()` instead.
 - added `has_tag()`
 - `tags` is deprecated, use `get_tags()` and `set_tags()` instead.
- `FastqFile` is now `FastxFile` to reflect that the latter permits iteration over both fastq- and fasta-formatted files.
- A Cython wrapper for htslib VCF/BCF reader/writer. The wrapper provides a nearly complete Pythonic interface to VCF/BCF metadata with reading and writing capability. However, the interface is still incomplete and preliminary and lacks capability to mutate the resulting data.

1.10.18 Release 0.8.1

- Pysam now wraps htslib and samtools versions 1.1.
- Bugfixes, most notable: * issue #43: uncompressed BAM output * issue #42: skip tests requiring network if none available * issue #19: multiple iterators can now be made to work on the same tabix file * issue #24: All strings returned from/passed to the pysam API are now unicode in python 3 * issue #5: type guessing for lists of integers fixed
- API changes for consistency. The old API is still present, but deprecated. In particular:
 - `Tabixfile` -> `TabixFile`
 - `Fastafile` -> `FastaFile`
 - `Fastqfile` -> `FastqFile`
 - `Samfile` -> `AlignmentFile`
 - **`AlignedRead` -> `AlignedSegment`**
 - * `qname` -> `query_name`
 - * `tid` -> `reference_id`
 - * `pos` -> `reference_start`
 - * `mapq` -> `mapping_quality`
 - * `rnext` -> `next_reference_id`
 - * `pnext` -> `next_reference_start`
 - * `cigar` -> `cigartuples`
 - * `cigarstring` -> `cigarstring`
 - * `tlen` -> `template_length`
 - * `seq` -> `query_sequence`

- * qual -> query_qualities, now returns array
- * qqual -> query_alignment_qualities, now returns array
- * tags -> tags
- * alen -> reference_length, reference is always “alignment”, so removed
- * aend -> reference_end
- * rlen -> query_length
- * query -> query_alignment_sequence
- * qstart -> query_alignment_start
- * qend -> query_alignment_end
- * qlen -> query_alignment_length
- * mrnm -> next_reference_id
- * mpos -> next_reference_start
- * rname -> reference_id
- * isize -> template_length
- * blocks -> get_blocks()
- * aligned_pairs -> get_aligned_pairs()
- * inferred_length -> infer_query_length()
- * positions -> get_reference_positions()
- * overlap() -> get_overlap()

- All strings are now passed to or received from the pysam API as strings, no more bytes.

Other changes:

- AlignmentFile.fetch(reopen) option is now multiple_iterators. The default changed to not reopen a file unless requested by the user.
- FastaFile.getReferenceLength is now FastaFile.get_reference_length

Backwards incompatible changes

- Empty cigarstring now returns None (instead of '')
- Empty cigar now returns None (instead of [])
- When using the extension classes in cython modules, AlignedRead needs to be substituted with AlignedSegment.
- fancy_str() has been removed
- qual, qqual now return arrays

1.10.19 Release 0.8.0

- **Disabled features**
 - IteratorColumn.setMask() disabled as htlib does not implement this functionality?
- **Not implemented yet:**
 - reading SAM files without header

Tabix files between version 0.7.8 and 0.8.0 are not compatible and need to be re-indexed.

While version 0.7.8 and 0.8.0 should be mostly compatible, there are some notable exceptions:

- tabix iterators will fail if there are comments in the middle or the end of a file.
- tabix raises always `ValueError` for invalid intervals. Previously, different types of errors were raised (`KeyError`, `IndexError`, `ValueError`) depending on the type of invalid intervals (missing chromosome, out-of-range, malformed interval).

1.10.20 Release 0.7.8

- added `AlignedRead.setTag` method
- added `AlignedRead.blocks`
- unsetting CIGAR strings is now possible
- empty CIGAR string returns empty list
- added reopen flag to `Samfile.fetch()`
- various bugfixes

1.10.21 Release 0.7.7

- added `Fastfile.references`, `.nreferences` and `.lengths`
- `tabix_iterator` now uses `kseq.h` for python 2.7

1.10.22 Release 0.7.6

- added `inferred_length` property
- issue 122: MACOSX `getline` missing, now it works?
- `seq` and `qual` can be set `None`
- added `Fastqfile`

1.10.23 Release 0.7.5

- switch to samtools 0.1.19
- issue 122: MACOSX `getline` missing
- issue 130: clean up tempfiles
- various other bugfixes

1.10.24 Release 0.7.4

- further bugfixes to `setup.py` and package layout

1.10.25 Release 0.7.3

- further bugfixes to setup.py
- upgraded distribute_setup.py to 0.6.34

1.10.26 Release 0.7.2

- bugfix in installer - failed when cython not present
- changed installation locations of shared libraries

1.10.27 Release 0.7.1

- bugfix: missing PP tag PG records in header
- added pre-built .c files to distribution

1.10.28 Release 0.7

- switch to tabix 0.2.6
- added cigarstring field
- python3 compatibility
- added B tag handling
- added check_sq and check_header options to Samfile.__init__
- added lazy GTF parsing to tabix
- reworked support for VCF format parsing
- bugfixes

1.10.29 Release 0.6

- switch to samtools 0.1.18
- various bugfixes
- removed references to deprecated 'samtools pileup' functionality
- AlignedRead.tags now returns an empty list if there are no tags.
- added pnext, rnext and tlen

1.10.30 Release 0.5

- switch to samtools 0.1.16 and tabix 0.2.5
- improved tabix parsing, added vcf support
- re-organized code to permit linking against pysam
- various bugfixes
- added Samfile.positions and Samfile.overlap

1.10.31 Release 0.4

- switch to samtools 0.1.12a and tabix 0.2.3
- added snp and indel calling.
- switch from pyrex to cython
- changed handling of samtools stderr
- various bugfixes
- added Samfile.count and Samfile.mate
- deprecated AlignedRead.rname, added AlignedRead.tid

1.10.32 Release 0.3

- switch to samtools 0.1.8
- added support for tabix files
- numerous bugfixes including
- permit simultaneous iterators on the same file
- working access to remote files

1.11 Benchmarking

Latest benchmarking results:

----- benchmark: 57 tests -----							

Name (time in us)						Min	
	Max		Mean		StdDev		
Median		IQR	Outliers		OPS	Rounds	
Iterations							

test_set_binary_tag						93.8382 (1.	
→0)	199.8786 (1.0)		96.1554 (1.0)		2.6241 (1.0)		
→95.7036 (1.0)		1.2442 (1.0)	276;303	10,399.8372 (1.0)		3170	
→1							
test_fasta_iteration_long_sequences_without_persistence						145.3292 (1.	
→55)	296.4940 (1.48)		176.5367 (1.84)		30.0896 (11.47)		
→167.2544 (1.75)		25.1532 (20.22)	840;579	5,664.5438 (0.54)		4972	
→1							
test_fasta_iteration_long_sequences_as_file						149.0638 (1.	
→59)	312.6319 (1.56)		177.2990 (1.84)		30.5218 (11.63)		
→168.6383 (1.76)		27.7516 (22.30)	669;510	5,640.1883 (0.54)		4493	
→1							
test_fasta_iteration_long_sequences						150.3211 (1.	
→60)	429.9153 (2.15)		176.7384 (1.84)		30.7021 (11.70)		
→167.5002 (1.75)		21.7482 (17.48)	682;647	5,658.0803 (0.54)		4845	
→1							

(continues on next page)

(continued from previous page)

```

test_fasta_iteration_short_sequences_as_file                190.6604 (2.
→03)          395.8736 (1.98)          216.6984 (2.25)          32.8710 (12.53)
→208.9385 (2.18)          16.8057 (13.51)          291;326          4,614.7086 (0.44)          3205
→
→1
test_read_python_uncompressed                                214.1297 (2.
→28)          450.2051 (2.25)          234.6956 (2.44)          33.3642 (12.71)
→223.4913 (2.34)          15.0129 (12.07)          140;221          4,260.8375 (0.41)          2043
→
→1
test_fastq_iteration_short_sequences_as_file                217.5961 (2.
→32)          420.1643 (2.10)          238.0264 (2.48)          30.4469 (11.60)
→228.8874 (2.39)          15.5573 (12.50)          209;235          4,201.2146 (0.40)          2877
→
→1
test_iterate_file_uncompressed                              225.5719 (2.
→40)          481.0989 (2.41)          249.7603 (2.60)          40.7266 (15.52)
→239.5548 (2.50)          15.9908 (12.85)          209;327          4,003.8387 (0.38)          3627
→
→1
test_iterate_file_compressed                                257.5517 (2.
→74)          576.6843 (2.89)          287.2536 (2.99)          49.8803 (19.01)
→277.4149 (2.90)          14.3824 (11.56)          126;248          3,481.2446 (0.33)          3023
→
→1
test_iterate_generic_uncompressed                          344.8855 (3.
→68)          712.5959 (3.57)          370.9369 (3.86)          46.2477 (17.62)
→365.0384 (3.81)          15.7915 (12.69)          54;83          2,695.8763 (0.26)          2240
→
→1
test_iterate_parsed_uncompressed                            349.3819 (3.
→72)          765.3460 (3.83)          378.1446 (3.93)          54.2199 (20.66)
→369.5488 (3.86)          15.4320 (12.40)          48;69          2,644.4910 (0.25)          1638
→
→1
test_read_python_compressed                                462.0645 (4.
→92)          836.2234 (4.18)          493.6158 (5.13)          51.2946 (19.55)
→485.5469 (5.07)          11.4385 (9.19)          28;46          2,025.8670 (0.19)          906
→
→1
test_iterate_parsed_compressed                              586.5730 (6.
→25)          1,050.1631 (5.25)          632.5464 (6.58)          82.6438 (31.49)
→608.1080 (6.35)          20.1799 (16.22)          115;143          1,580.9117 (0.15)          1474
→
→1
test_iterate_generic_compressed                            587.7707 (6.
→26)          1,093.4286 (5.47)          639.6830 (6.65)          85.2593 (32.49)
→612.6408 (6.40)          19.9433 (16.03)          105;144          1,563.2743 (0.15)          1260
→
→1
test_fasta_iteration_short_sequences_without_persistence    725.2563 (7.
→73)          1,091.6069 (5.46)          774.6549 (8.06)          53.0901 (20.23)
→751.8455 (7.86)          45.0788 (36.23)          172;110          1,290.8974 (0.12)          1276
→
→1
test_read_binary_tag                                         817.2598 (8.
→71)          1,232.0559 (6.16)          902.7002 (9.39)          87.7807 (33.45)
→871.5261 (9.11)          17.4227 (14.00)          39;55          1,107.7875 (0.11)          331
→
→1
test_fastq_iteration_short_sequences_without_persistence    840.3640 (8.
→96)          1,201.5756 (6.01)          870.9679 (9.06)          35.8709 (13.67)
→866.1682 (9.05)          15.9768 (12.84)          53;85          1,148.1480 (0.11)          1124
→
→1
test_count_number_lines_from_sam_with_pysam                2,755.6140
→(29.37)          9,656.9806 (48.31)          2,963.2206 (30.82)          729.0077 (277.81)
→2,814.3115 (29.41)          106.2388 (85.38)          8;14          337.4707 (0.03)          281
→
→1
test_fasta_iteration_short_sequences                        2,905.5942
→(30.96)          3,674.1439 (18.38)          2,982.3892 (31.02)          90.9585 (34.95)
→2,941.2108 (30.73)          96.1348 (77.26)          27;7          335.3016 (0.03)          302
→
→1

```

(continues on next page)

(continued from previous page)

```

test_fastq_iteration_short_sequences                                3,601.3145_
→(38.38)      4,065.5769 (20.34)      3,671.8361 (38.19)      71.5348 (27.26)      _
→3,635.1625 (37.98)      81.5415 (65.53)      43;10      272.3433 (0.03)      243_
→
→1
test_count_number_lines_from_bam_with_pysam                        4,178.8872_
→(44.53)      12,063.5536 (60.35)      4,395.9713 (45.72)      794.3991 (302.73)      _
→4,240.1757 (44.31)      67.0198 (53.86)      6;19      227.4810 (0.02)      205_
→
→1
test_build_depth_from_bam_with_pysam                              6,290.2980_
→(67.03)      6,788.0806 (33.96)      6,420.3862 (66.77)      126.5447 (48.22)      _
→6,357.6270 (66.43)      94.3104 (75.80)      35;29      155.7539 (0.01)      145_
→
→1
test_build_depth_with_filter_from_bam_with_pysam                  6,935.8926_
→(73.91)      8,309.5767 (41.57)      7,085.6801 (73.69)      221.5069 (84.41)      _
→7,005.6170 (73.20)      74.4388 (59.83)      14;27      141.1297 (0.01)      139_
→
→1
test_build_query_positions_from_bam_with_pysam                    8,729.2437_
→(93.02)      16,836.6525 (84.23)      9,173.2902 (95.40)      1,057.4321 (402.97)      _
→8,885.4395 (92.84)      264.7634 (212.79)      3;9      109.0121 (0.01)      103_
→
→1
test_build_mapping_qualities_from_bam_with_pysam                  8,805.0570_
→(93.83)      15,829.8910 (79.20)      9,274.7475 (96.46)      964.6925 (367.63)      _
→9,004.7438 (94.09)      354.9103 (285.24)      4;9      107.8196 (0.01)      106_
→
→1
test_build_query_qualities_from_bam_with_pysam                    8,947.6798_
→(95.35)      16,756.8158 (83.83)      9,272.9117 (96.44)      1,008.7759 (384.43)      _
→9,017.0493 (94.22)      217.2068 (174.57)      3;5      107.8410 (0.01)      94_
→
→1
test_build_query_bases_from_bam_with_samtoolspysam                9,149.0448_
→(97.50)      9,776.0092 (48.91)      9,208.3405 (95.77)      86.1889 (32.85)      _
→9,190.2809 (96.03)      34.1963 (27.48)      5;7      108.5972 (0.01)      84_
→
→1
test_count_number_lines_from_bam_with_samtoolspipe                10,431.4052_
→(111.16)      13,431.7447 (67.20)      11,052.2360 (114.94)      445.8241 (169.90)      _
→11,014.8042 (115.09)      492.6044 (395.91)      17;3      90.4794 (0.01)      _
→90
→1
test_count_number_lines_from_bam_with_samtools                    12,057.8520_
→(128.50)      13,537.5429 (67.73)      12,595.0708 (130.99)      381.1799 (145.26)      _
→12,499.5783 (130.61)      457.8107 (367.94)      25;1      79.3961 (0.01)      _
→73
→1
test_count_number_lines_from_sam_with_samtoolspipe                12,434.8756_
→(132.51)      14,234.1983 (71.21)      13,034.1032 (135.55)      371.0373 (141.40)      _
→13,049.9089 (136.36)      464.8147 (373.57)      21;3      76.7218 (0.01)      _
→72
→1
test_build_query_bases_from_bam_with_pysam                        12,811.6887_
→(136.53)      19,972.7099 (99.92)      13,268.7908 (137.99)      1,103.9459 (420.70)      _
→13,023.1632 (136.08)      202.3596 (162.64)      4;6      75.3648 (0.01)      _
→76
→1
test_build_pileup_from_bam_with_samtoolspipe                      13,480.7490_
→(143.66)      15,413.4836 (77.11)      13,968.3817 (145.27)      362.1282 (138.00)      _
→13,917.4843 (145.42)      447.9736 (360.04)      12;3      71.5903 (0.01)      _
→66
→1
test_count_number_lines_from_sam_with_samtools                    13,963.2300_
→(148.80)      15,936.2238 (79.73)      14,442.5627 (150.20)      340.4846 (129.75)      _
→14,395.4996 (150.42)      339.3036 (272.70)      12;2      69.2398 (0.01)      _
→59
→1
test_build_pileup_from_bam_with_samtoolsshell                      15,140.2969_
→(161.34)      16,469.8567 (82.40)      15,525.9034 (161.47)      256.0741 (continues on next page)
→15,526.7641 (162.24)      318.5044 (255.98)      13;2      64.4085 (0.01)      _
→51
→1

```

(continued from previous page)

test_build_depth_from_bam_with_samtoolsshell	15,291.4841	
→ (162.96) 16,387.9916 (81.99) 15,656.5365 (162.83) 217.2420 (82.79)		
→ 15,609.9945 (163.11) 258.9412 (208.11) 16;2 63.8711 (0.01)		
→ 57 1		
test_build_query_bases_from_bam_with_samtoolspipe	15,478.0652	
→ (164.94) 17,068.8462 (85.40) 16,072.5749 (167.15) 327.6668 (124.87)		
→ 16,040.5049 (167.61) 428.1597 (344.11) 18;2 62.2178 (0.01)		
→ 60 1		
test_build_query_bases_from_bam_with_samtoolsshell	15,765.5794	
→ (168.01) 17,102.8059 (85.57) 16,305.4955 (169.57) 367.6865 (140.12)		
→ 16,256.2486 (169.86) 593.9230 (477.34) 20;0 61.3290 (0.01)		
→ 57 1		
test_build_depth_from_bam_with_samtoolspipe	16,024.7944	
→ (170.77) 25,804.2309 (129.10) 17,041.3497 (177.23) 1,522.0046 (580.01)		
→ 16,726.8887 (174.78) 767.1015 (616.52) 1;1 58.6808 (0.01)		
→ 40 1		
test_build_query_qualities_from_bam_with_samtoolspipe	16,275.7467	
→ (173.44) 18,501.0433 (92.56) 17,262.6517 (179.53) 642.2656 (244.76)		
→ 17,255.3696 (180.30) 1,060.3429 (852.20) 20;0 57.9285 (0.01)		
→ 51 1		
test_build_pileup_from_bam_with_pysam	16,632.5681	
→ (177.25) 19,306.6560 (96.59) 17,037.7162 (177.19) 531.0751 (202.38)		
→ 16,870.8330 (176.28) 279.1677 (224.37) 3;4 58.6933 (0.01)		
→ 32 1		
test_build_mapping_qualities_from_bam_with_samtoolspipe	17,398.3518	
→ (185.41) 18,481.6569 (92.46) 17,832.1261 (185.45) 294.9103 (112.39)		
→ 17,837.8206 (186.39) 511.8400 (411.37) 21;0 56.0786 (0.01)		
→ 52 1		
test_build_query_names_from_bam_with_pysam	31,534.0199	
→ (336.05) 38,316.2647 (191.70) 32,677.2804 (339.84) 1,371.6101 (522.70)		
→ 32,179.0325 (336.24) 1,392.1391 (>1000.0) 3;1 30.6023 (0.00)		
→ 30 1		
test_fetch_plain	43,747.1233	
→ (466.20) 48,690.7829 (243.60) 44,573.8085 (463.56) 1,170.8588 (446.20)		
→ 44,067.5411 (460.46) 1,436.5837 (>1000.0) 3;1 22.4347 (0.00)		
→ 22 1		
test_build_query_positions_from_bam_with_samtoolspipe	46,316.5548	
→ (493.58) 53,700.3577 (268.66) 48,235.6640 (501.64) 1,861.0317 (709.21)		
→ 47,625.2194 (497.63) 2,603.7074 (>1000.0) 6;1 20.7315 (0.00)		
→ 21 1		
test_build_query_bases_with_reference_from_bam_with_samtoolspysam	51,565.1479	
→ (549.51) 53,523.1121 (267.78) 51,778.6650 (538.49) 424.4621 (161.76)		
→ 51,656.8925 (539.76) 110.5051 (88.81) 1;4 19.3130 (0.00)		
→ 20 1		
test_build_query_bases_with_reference_from_bam_with_pysam	58,850.6740	
→ (627.15) 62,164.5451 (311.01) 60,161.6779 (625.67) 1,120.6101 (427.05)		
→ 59,595.2785 (622.71) 1,995.0196 (>1000.0) 7;0 16.6219 (0.00)		
→ 16 1		
test_iterate_file_large_uncompressed	59,419.9076	
→ (633.22) 69,053.0874 (345.48) 62,825.7126 (653.38) 3,805.1150 (>1000.0)		
→ 60,805.1391 (635.35) 6,407.6949 (>1000.0) 3;0 15.9170 (0.00)		
→ 14 1		
test_iterate_file_large_compressed	63,986.9571	
→ (681.89) 74,156.9120 (371.01) 68,370.8835 (711.05) 3,254.6200 (>1000.0)		
→ 67,221.2075 (702.39) 4,770.3525 (>1000.0) 5;0 14.6261 (0.00)		
→ 16 1		
test_read_python_large_uncompressed	67,611.8080	
→ (720.51) 89,112.1533 (445.83) 73,631.8916 (765.76) 5,978.7052 (continues on next page)		
→ 71,702.4822 (749.21) 7,856.0165 (>1000.0) 2;1 13.5811 (0.00)		
→ 13 1		

(continued from previous page)

```

test_fetch_parsed                                72,237.5344
→ (769.81) 81,976.9856 (410.13) 74,540.4099 (775.21) 2,326.3668 (886.54)
→ 74,016.3419 (773.39) 1,579.0667 (>1000.0) 1;1 13.4155 (0.00)
→ 14 1
test_build_query_bases_from_bam_with_pysam_pileups 75,841.3766
→ (808.21) 87,975.1425 (440.14) 79,393.9784 (825.68) 3,516.2458 (>1000.0)
→ 79,346.7313 (829.09) 5,217.5033 (>1000.0) 2;0 12.5954 (0.00)
→ 13 1
test_build_query_bases_with_reference_from_bam_with_samtoolspipe 131,162.9396 (>
→ 1000.0) 132,858.8147 (664.70) 131,838.0199 (>1000.0) 595.6090 (226.98) 131,
→ 801.5633 (>1000.0) 956.9665 (769.11) 2;0 7.5851 (0.00) 8
→ 1
test_iterate_parsed_large_uncompressed 132,991.8914 (>
→ 1000.0) 140,153.4099 (701.19) 134,101.5892 (>1000.0) 2,452.3161 (934.54) 133,
→ 218.1590 (>1000.0) 396.3616 (318.56) 1;1 7.4570 (0.00) 8
→ 1
test_iterate_generic_large_uncompressed 134,743.7277 (>
→ 1000.0) 142,329.4526 (712.08) 138,167.4954 (>1000.0) 3,346.7773 (>1000.0) 136,
→ 839.1849 (>1000.0) 6,541.1879 (>1000.0) 4;0 7.2376 (0.00) 8
→ 1
test_read_python_large_compressed 175,127.6311 (>
→ 1000.0) 190,855.1529 (954.86) 181,702.9339 (>1000.0) 5,756.2207 (>1000.0) 181,
→ 221.0185 (>1000.0) 8,577.6616 (>1000.0) 2;0 5.5035 (0.00) 6
→ 1
test_iterate_parsed_large_compressed 231,405.7611 (>
→ 1000.0) 243,728.8519 (>1000.0) 239,037.8296 (>1000.0) 5,212.7778 (>1000.0) 241,
→ 544.3324 (>1000.0) 8,062.6113 (>1000.0) 1;0 4.1834 (0.00) 5
→ 1
test_iterate_generic_large_compressed 235,042.3876 (>
→ 1000.0) 256,518.8371 (>1000.0) 242,535.5468 (>1000.0) 8,272.5197 (>1000.0) 240,
→ 163.0748 (>1000.0) 8,360.1568 (>1000.0) 1;0 4.1231 (0.00) 5
→ 1
-----
→ -----
→ -----
→ -----

```

1.12 Glossary

BAM Binary SAM format. BAM files are binary formatted, indexed and allow random access.

BCF Binary *VCF*

bgzip Utility in the *htslib* package to block compress genomic data files.

cigar Stands for Compact Idiosyncratic Gapped Alignment Report and represents a compressed (run-length encoded) pairwise alignment format. It was first defined by the Exonerate Aligner, but was later adapted and adopted as part of the *SAM* standard and many other aligners. In the Python API, the cigar alignment is presented as a list of tuples (operation, length). For example, the tuple [(0, 3), (1, 5), (0, 2)] refers to an alignment with 3 matches, 5 insertions and another 2 matches.

column Reads that are aligned to a base in the *reference* sequence.

contig The sequence that a *tid* refers to. For example chr1, contig123.

csamtools The *samtools* C-API.

faidx Utility in the samtools package to index fasta formatted files.

fetching Retrieving all mapped reads mapped to a *region*.

hard clipping

hard clipped In hard clipped reads, part of the sequence has been removed prior to alignment. That only a subsequence is aligned might be recorded in the *cigar* alignment, but the removed sequence will not be part of the alignment record, in contrast to *soft clipped* reads.

pileup Pileup

Reference Synonym for *contig*

region A genomic region, stated relative to a reference sequence. A region consists of reference name ('chr1'), start (10000), and end (20000). Start and end can be omitted for regions spanning a whole chromosome. If end is missing, the region will span from start to the end of the chromosome. Within pysam, coordinates are 0-based, half-open intervals, i.e., the position 10,000 is part of the interval, but 20,000 is not. An exception are *samtools* compatible region strings such as 'chr1:10000:20000', which are closed, i.e., both positions 10,000 and 20,000 are part of the interval.

SAM A textual format for storing genomic alignment information.

sam file A file containing aligned reads. The *sam file* can either be a *BAM* file or a *TAM* file.

samtools The *samtools* package.

soft clipping

soft clipped In alignments with soft clipping part of the query sequence are not aligned. The unaligned query sequence is still part of the alignment record. This is in difference to *hard clipped* reads.

tabix Utility in the htstlib package to index *bgzip* compressed files.

tabix file A sorted, compressed and indexed tab-separated file created by the command line tool *tabix* or the commands *tabix_compress()* and *tabix_index()*. The file is indexed by chromosomal coordinates.

tabix row A row in a *tabix file*. Fields within a row are tab-separated.

TAM Text SAM file. TAM files are human readable files of tab-separated fields. TAM files do not allow random access.

target The sequence that a read has been aligned to. Target sequences have both a numerical identifier (*tid*) and an alphanumeric name (*Reference*).

tid The *target* id. The target id is 0 or a positive integer mapping to entries within the sequence dictionary in the header section of a *TAM* file or *BAM* file.

VCF Variant call format

CHAPTER 2

Indices and tables

Contents:

- [genindex](#)
- [modindex](#)
- [search](#)

CHAPTER 3

References

See also:

Information about htlib <http://www.htlib.org>

The samtools homepage <http://samtools.sourceforge.net>

The cython C-extensions for python <http://cython.org/>

The python language <http://www.python.org>

Bibliography

[Li2009] The Sequence Alignment/Map format and SAMtools. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R; 1000 Genome Project Data Processing Subgroup. *Bioinformatics*. 2009 Aug 15;25(16):2078-9. Epub 2009 Jun 8. [PMID: 19505943](#)

A

`add_hts_options()` (pysam.HTSFile method), 32
`add_line()` (pysam.VariantHeader method), 30
`add_meta()` (pysam.VariantHeader method), 30
`add_record()` (pysam.VariantHeader method), 30
`add_sample()` (pysam.VariantHeader method), 30
`aend` (pysam.AlignedSegment attribute), 12
`alen` (pysam.AlignedSegment attribute), 12
`aligned_pairs` (pysam.AlignedSegment attribute), 12
`AlignedSegment` (class in pysam), 12
`alignment` (pysam.PileupRead attribute), 22
`AlignmentFile` (class in pysam), 5
`alleles` (pysam.VariantRecord attribute), 31
`alts` (pysam.VariantHeader attribute), 30
`alts` (pysam.VariantRecord attribute), 31
`asBed` (class in pysam), 25
`asGTF` (class in pysam), 25
`asTuple` (class in pysam), 24
`asVCF` (class in pysam), 25
`attrs` (pysam.VariantHeaderRecord attribute), 32

B

`BAM`, 63
`BCF`, 63
`bgzip`, 63
`bin` (pysam.AlignedSegment attribute), 12
`blocks` (pysam.AlignedSegment attribute), 12
`build()` (pysam.IndexedReads method), 23

C

`category` (pysam.HTSFile attribute), 32
`check_index()` (pysam.AlignmentFile method), 7
`check_truncation()` (pysam.HTSFile method), 33
`chrom` (pysam.VariantRecord attribute), 31
`cigar`, 63
`cigar` (pysam.AlignedSegment attribute), 12
`cigarstring` (pysam.AlignedSegment attribute), 12
`cigartuples` (pysam.AlignedSegment attribute), 13
`close()` (pysam.AlignmentFile method), 7

`close()` (pysam.FastaFile method), 26
`close()` (pysam.FastxFile method), 28
`close()` (pysam.HTSFile method), 33
`close()` (pysam.TabixFile method), 23
`close()` (pysam.VariantFile method), 29
`closed` (pysam.FastaFile attribute), 26
`closed` (pysam.FastxFile attribute), 28
`closed` (pysam.HTSFile attribute), 33
`column`, 63
`compare()` (pysam.AlignedSegment method), 13
`compression` (pysam.HTSFile attribute), 33
`contig`, 63
`contig` (pysam.VariantRecord attribute), 31
`contigs` (pysam.TabixFile attribute), 23
`contigs` (pysam.VariantHeader attribute), 30
`copy()` (pysam.VariantFile method), 29
`copy()` (pysam.VariantHeader method), 30
`copy()` (pysam.VariantRecord method), 31
`count()` (pysam.AlignmentFile method), 7
`count_coverage()` (pysam.AlignmentFile method), 8
`csamtools`, 63

D

`description` (pysam.HTSFile attribute), 33

F

`faidx`, 64
`FastaFile` (class in pysam), 26
`FastqProxy` (class in pysam), 28
`FastxFile` (class in pysam), 27
`fetch()` (pysam.AlignmentFile method), 8
`fetch()` (pysam.FastaFile method), 26
`fetch()` (pysam.TabixFile method), 23
`fetch()` (pysam.VariantFile method), 29
`fetching`, 64
`filename` (pysam.FastaFile attribute), 27
`filename` (pysam.FastxFile attribute), 28
`filter` (pysam.VariantRecord attribute), 31
`filters` (pysam.VariantHeader attribute), 30
`find()` (pysam.IndexedReads method), 23

find_introns() (pysam.AlignmentFile method), 9
find_introns_slow() (pysam.AlignmentFile method), 9
flag (pysam.AlignedSegment attribute), 13
format (pysam.HTSFile attribute), 33
format (pysam.VariantRecord attribute), 31
formats (pysam.VariantHeader attribute), 30
from_dict() (pysam.AlignedSegment method), 13
fromstring() (pysam.AlignedSegment method), 13

G

get() (pysam.VariantHeaderRecord method), 32
get_aligned_pairs() (pysam.AlignedSegment method), 13
get_blocks() (pysam.AlignedSegment method), 14
get_cigar_stats() (pysam.AlignedSegment method), 14
get_forward_qualities() (pysam.AlignedSegment method), 14
get_forward_sequence() (pysam.AlignedSegment method), 14
get_index_statistics() (pysam.AlignmentFile method), 9
get_mapping_qualities() (pysam.PileupColumn method), 20
get_num_aligned() (pysam.PileupColumn method), 20
get_overlap() (pysam.AlignedSegment method), 14
get_quality_array() (pysam.FastqProxy method), 28
get_query_names() (pysam.PileupColumn method), 20
get_query_positions() (pysam.PileupColumn method), 20
get_query_qualities() (pysam.PileupColumn method), 21
get_query_sequences() (pysam.PileupColumn method), 21
get_reference_length() (pysam.AlignmentFile method), 9
get_reference_length() (pysam.FastaFile method), 27
get_reference_name() (pysam.AlignmentFile method), 9
get_reference_name() (pysam.HTSFile method), 33
get_reference_name() (pysam.VariantFile method), 29
get_reference_positions() (pysam.AlignedSegment method), 14
get_reference_sequence() (pysam.AlignedSegment method), 15
get_tag() (pysam.AlignedSegment method), 15
get_tags() (pysam.AlignedSegment method), 15
get_tid() (pysam.AlignmentFile method), 9
get_tid() (pysam.HTSFile method), 33
get_tid() (pysam.VariantFile method), 29
getname() (pysam.AlignmentFile method), 9
gettid() (pysam.AlignmentFile method), 9

H

hard clipped, 64
hard clipping, 64
has_index() (pysam.AlignmentFile method), 9
has_tag() (pysam.AlignedSegment method), 15
head() (pysam.AlignmentFile method), 10
header (pysam.TabixFile attribute), 24
HTSFile (class in pysam), 32

I

id (pysam.VariantRecord attribute), 31
indel (pysam.PileupRead attribute), 22
IndexedReads (class in pysam), 22
infer_query_length() (pysam.AlignedSegment method), 15
infer_read_length() (pysam.AlignedSegment method), 15
inferred_length (pysam.AlignedSegment attribute), 16
info (pysam.VariantHeader attribute), 30
info (pysam.VariantRecord attribute), 31
is_bam (pysam.HTSFile attribute), 33
is_bcf (pysam.HTSFile attribute), 33
is_closed (pysam.HTSFile attribute), 33
is_cram (pysam.HTSFile attribute), 33
is_del (pysam.PileupRead attribute), 22
is_duplicate (pysam.AlignedSegment attribute), 16
is_head (pysam.PileupRead attribute), 22
is_open (pysam.HTSFile attribute), 33
is_open() (pysam.FastaFile method), 27
is_open() (pysam.FastxFile method), 28
is_paired (pysam.AlignedSegment attribute), 16
is_proper_pair (pysam.AlignedSegment attribute), 16
is_qcfail (pysam.AlignedSegment attribute), 16
is_read (pysam.HTSFile attribute), 33
is_read1 (pysam.AlignedSegment attribute), 16
is_read2 (pysam.AlignedSegment attribute), 16
is_refskip (pysam.PileupRead attribute), 22
is_reverse (pysam.AlignedSegment attribute), 16
is_sam (pysam.HTSFile attribute), 33
is_secondary (pysam.AlignedSegment attribute), 16
is_supplementary (pysam.AlignedSegment attribute), 16
is_tail (pysam.PileupRead attribute), 22
is_unmapped (pysam.AlignedSegment attribute), 16
is_valid_reference_name() (pysam.HTSFile method), 33
is_valid_tid() (pysam.AlignmentFile method), 10
is_valid_tid() (pysam.HTSFile method), 33
is_valid_tid() (pysam.VariantFile method), 29
is_vcf (pysam.HTSFile attribute), 33
is_write (pysam.HTSFile attribute), 33
isize (pysam.AlignedSegment attribute), 16
items() (pysam.VariantHeaderRecord method), 32
iteritems() (pysam.VariantHeaderRecord method), 32
iterkeys() (pysam.VariantHeaderRecord method), 32
itervalues() (pysam.VariantHeaderRecord method), 32

K

key (pysam.VariantHeaderRecord attribute), 32
keys() (pysam.VariantHeaderRecord method), 32

L

lengths (pysam.AlignmentFile attribute), 10
lengths (pysam.FastaFile attribute), 27
level (pysam.PileupRead attribute), 22

M

mapped (pysam.AlignmentFile attribute), 10
 mapping_quality (pysam.AlignedSegment attribute), 16
 mapq (pysam.AlignedSegment attribute), 16
 mate() (pysam.AlignmentFile method), 10
 mate_is_reverse (pysam.AlignedSegment attribute), 16
 mate_is_unmapped (pysam.AlignedSegment attribute), 16
 merge() (pysam.VariantHeader method), 31
 mpos (pysam.AlignedSegment attribute), 16
 mrnm (pysam.AlignedSegment attribute), 16

N

n (pysam.PileupColumn attribute), 21
 name (pysam.FastqProxy attribute), 28
 new_record() (pysam.VariantFile method), 30
 new_record() (pysam.VariantHeader method), 31
 next (pysam.AlignmentFile attribute), 10
 next (pysam.FastxFile attribute), 28
 next (pysam.VariantFile attribute), 30
 next_reference_id (pysam.AlignedSegment attribute), 16
 next_reference_name (pysam.AlignedSegment attribute), 16
 next_reference_start (pysam.AlignedSegment attribute), 16
 nocoordinate (pysam.AlignmentFile attribute), 10
 nreferences (pysam.AlignmentFile attribute), 10
 nreferences (pysam.FastaFile attribute), 27
 nsegments (pysam.PileupColumn attribute), 21

O

open() (pysam.VariantFile method), 30
 opt() (pysam.AlignedSegment method), 16
 overlap() (pysam.AlignedSegment method), 17

P

parse_region() (pysam.HTSFile method), 34
 pileup, 64
 pileup() (pysam.AlignmentFile method), 10
 PileupColumn (class in pysam), 20
 PileupRead (class in pysam), 22
 pileups (pysam.PileupColumn attribute), 21
 pnext (pysam.AlignedSegment attribute), 17
 pop() (pysam.VariantHeaderRecord method), 32
 pos (pysam.AlignedSegment attribute), 17
 pos (pysam.PileupColumn attribute), 21
 pos (pysam.VariantRecord attribute), 31
 positions (pysam.AlignedSegment attribute), 17

Q

qend (pysam.AlignedSegment attribute), 17
 qlen (pysam.AlignedSegment attribute), 17
 qname (pysam.AlignedSegment attribute), 17

qqual (pysam.AlignedSegment attribute), 17
 qstart (pysam.AlignedSegment attribute), 17
 qual (pysam.AlignedSegment attribute), 17
 qual (pysam.VariantRecord attribute), 31
 quality (pysam.FastqProxy attribute), 28
 query (pysam.AlignedSegment attribute), 17
 query_alignment_end (pysam.AlignedSegment attribute), 17
 query_alignment_length (pysam.AlignedSegment attribute), 17
 query_alignment_qualities (pysam.AlignedSegment attribute), 17
 query_alignment_sequence (pysam.AlignedSegment attribute), 17
 query_alignment_start (pysam.AlignedSegment attribute), 18
 query_length (pysam.AlignedSegment attribute), 18
 query_name (pysam.AlignedSegment attribute), 18
 query_position (pysam.PileupRead attribute), 22
 query_position_or_next (pysam.PileupRead attribute), 22
 query_qualities (pysam.AlignedSegment attribute), 18
 query_sequence (pysam.AlignedSegment attribute), 18

R

records (pysam.VariantHeader attribute), 31
 ref (pysam.VariantRecord attribute), 31
 Reference, 64
 reference_end (pysam.AlignedSegment attribute), 18
 reference_id (pysam.AlignedSegment attribute), 18
 reference_id (pysam.PileupColumn attribute), 22
 reference_length (pysam.AlignedSegment attribute), 18
 reference_name (pysam.AlignedSegment attribute), 19
 reference_name (pysam.PileupColumn attribute), 22
 reference_pos (pysam.PileupColumn attribute), 22
 reference_start (pysam.AlignedSegment attribute), 19
 references (pysam.AlignmentFile attribute), 12
 references (pysam.FastaFile attribute), 27
 region, 64
 remove() (pysam.VariantHeaderRecord method), 32
 reset() (pysam.HTSFile method), 34
 reset() (pysam.VariantFile method), 30
 rid (pysam.VariantRecord attribute), 31
 rlen (pysam.AlignedSegment attribute), 19
 rlen (pysam.VariantRecord attribute), 31
 rname (pysam.AlignedSegment attribute), 19
 rnext (pysam.AlignedSegment attribute), 19

S

SAM, 64
 sam file, 64
 samples (pysam.VariantHeader attribute), 31
 samples (pysam.VariantRecord attribute), 31
 samtools, 64
 seek() (pysam.HTSFile method), 34

seq (pysam.AlignedSegment attribute), 19
sequence (pysam.FastqProxy attribute), 28
set_min_base_quality() (pysam.PileupColumn method),
22
set_tag() (pysam.AlignedSegment method), 19
set_tags() (pysam.AlignedSegment method), 19
setTag() (pysam.AlignedSegment method), 19
soft clipped, 64
soft clipping, 64
start (pysam.VariantRecord attribute), 31
stop (pysam.VariantRecord attribute), 32
subset_samples() (pysam.VariantFile method), 30

T

tabix, 64
tabix file, 64
tabix row, 64
tabix_compress() (in module pysam), 24
tabix_index() (in module pysam), 24
tabix_iterator() (in module pysam), 24
TabixFile (class in pysam), 23
tags (pysam.AlignedSegment attribute), 20
TAM, 64
target, 64
tell() (pysam.HTSFile method), 34
template_length (pysam.AlignedSegment attribute), 20
text (pysam.AlignmentFile attribute), 12
tid, 64
tid (pysam.AlignedSegment attribute), 20
tid (pysam.PileupColumn attribute), 22
tlen (pysam.AlignedSegment attribute), 20
to_dict() (pysam.AlignedSegment method), 20
to_string() (pysam.AlignedSegment method), 20
tostring() (pysam.AlignedSegment method), 20
translate() (pysam.VariantRecord method), 32
type (pysam.VariantHeaderRecord attribute), 32

U

unmapped (pysam.AlignmentFile attribute), 12
update() (pysam.VariantHeaderRecord method), 32

V

value (pysam.VariantHeaderRecord attribute), 32
values() (pysam.VariantHeaderRecord method), 32
VariantFile (class in pysam), 28
VariantHeader (class in pysam), 30
VariantHeaderRecord (class in pysam), 32
VariantRecord (class in pysam), 31
VCF, 64
version (pysam.HTSFile attribute), 34
version (pysam.VariantHeader attribute), 31

W

write() (pysam.AlignmentFile method), 12