pysam documentation

Release 0.12.0.1

Andreas Heger, Kevin Jacobs et al.

Sep 08, 2017

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 . . . 29
1.3 Using samtools commands within python . . . 31
1.4 Working with tabix-indexed files . . . 32
1.5 Working with VCF/BCF formatted files . . . 32
1.6 Extending pysam . . . 34
1.7 Installing pysam . . . 35
1.8 FAQ . . . 36
1.9 Developer's guide . . . 40
1.10 Release notes . . . 41
1.11 Glossary . . . 52

2 Indices and tables **55**

3 References **57**

Bibliography **59**

Author Andreas Heger, Kevin Jacobs and contributors

Date Sep 08, 2017

Version 0.12.0.1

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-1.3*, *samtools-1.3* and *bcftools-1.3*.

To install the latest release, type:

```
pip install pysam
```

See the *Installation notes* for details.

CHAPTER 1

Contents

pysam - An interface for reading and writing SAM files

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<;    ␣
↪69     CTCAAGGTTGTTGCAAGGGGGTCTATGTGAACAAA     0       192     1
EAS56_57:6:190:289:82       0       99      <<<<<<;<<<<<<<<<<;<<;<<<<;8<6;9;;2;    ␣
↪137    AGGGGTGCAGAGCCGAGTCACGGGGTTGCCAGCAC     73      64      1
EAS51_64:3:190:727:308      0       102     <<<<<<<<<<<<<<<<<<<<<<<<<<<::<<<844    ␣
↪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

API

SAM/BAM 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)

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 *template*.
- **`header`** (*`dict`*) – 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.
- **`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)

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 compatiblity 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=None*, *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 compatiblity 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)
- **`stop`** (*`int`*) – end of the genomic region (0-based exclusive)
- **`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 `AlignmentFile.parse_region()` for more information on genomic regions. *reference* and *end* are also accepted for backward compatiblity 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 *reference* is set, all reads aligned to *reference* will be fetched.

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

`FastaFile` `IteratorRow` `IteratorRow` `IteratorRow` `IteratorRow`

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)

`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_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 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

`header`

two-level dictionay with header information from the file.

This is a read-only attribute.

The first level contains the record (`HD`, `SQ`, etc) and the second level contains the fields (`VN`, `LN`, etc).

The parser is validating and will raise an AssertionError if if encounters any record or field tags that are not part of the SAM specification. Use the `pysam.AlignmentFile.text` attribute to get the unparsed header.

The parsing follows the SAM format specification with the exception of the `CL` field. This option will consume the rest of a header line irrespective of any additional fields. This behaviour has been added to accommodate command line options that contain characters that are not valid field separators.

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

`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.

`nreferences`
"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 compatiblity 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 **`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

`samtools` same filter and read processing as in *csamtools* pileup. This requires a 'fastafile' to be given.

fastafile : *`FastaFile`* object.

This is required for some of the steppers.

max_depth [int] Maximum read depth permitted. The default limit is '8000'.

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 specificied are returned.

Returns

Return type an iterator over genomic positions.

`references`

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

`text`

string with the full contents of the *sam file* header as a string.

This is a read-only attribute.

See *`pysam.AlignmentFile.header`* to get a parsed representation of the header.

`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`**

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.

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

`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_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.

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.

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 aligment 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, query_alignment_qualities instead

qstart
deprecated, use query_alignment_start instead

qual
deprecated, query_qualities instead

query
deprecated, query_alignment_sequence instead

query_alignment_end
end index of the aligned query portion of the sequence (0-based, exclusive)

This 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_squence = 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 *`pysam.AlignmentFile.getrname()`*

`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 (None if no AlignmentFile is associated)

`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.

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

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

`set_tags`(*self*, *tags*)
sets the fields in the optional alignmest 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

`tostring`(*self*, *AlignmentFile_t htsfile*)
returns a string representation of the aligned segment.

The output format is valid SAM format.

Parameters -- `AlignmentFile object to map numerical` (*`htsfile`*) – identifiers to chromosome names.

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.

`nsegments`

number of reads mapping to this column.

`pileups`

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

`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).

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

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.

Note: The header is only available for local files. For remote files an Attribute Error is raised.

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 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* does not end in ”.gz”, it will be automatically compressed. The original file will be removed and only the compressed file will be retained.

If *filename* ends in *gz*, the file is assumed to be already compressed with bgzf.

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

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 bocks
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:

Column	*Name*	*Content*
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:

Name	*Content*
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".

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 faster iteration, but an entry will not persist when the iteration continues or is not in-place modifyable.

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)

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)

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 incurrs 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*)
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

Working with BAM/CRAM/SAM-formatted files

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")
```

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 returns 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.

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))
```

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="AGCTTAGCTAGCTACCTATATCTTGGTCTTGGCCG"
    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
    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)
```

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 changed from `r` to `rb`.

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):
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.

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 we 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)
```

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
```

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

   ...

   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 `pyxbld` 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.

Installing pysam

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

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.

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 explicitely 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-varirable *LD_LIBRARY_PATH*.

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.

FAQ

How should I cite pysam

Pysam has not been published in print. When refering 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>*.

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 implemenented 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 objec - 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 hapeding, use the option `mulitple_iterator=True` when calling a fetch() method. This will return an iterator on a newly opened file.

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()`.

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/calignmentfile.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 `mulitple_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.

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

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

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.

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.

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]
```

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.

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 to 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
```

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)

ImportError: cannot import name csamtools

In version 0.10.0 and onwards, all pysam extension modules contain a `lib`-prefix. This facilates 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
```

Developer's guide

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

`htslib` Source code from htslib shipped with pysam. See `setup.py` about importing.

`samtools` Source code from *csamtools* shipped with pysam. See `setup.py` about importing.

Importing new versions of htslib and samtools

See instructions in `setup.py` to import the latest version of htslib and samtools.

Unit testing

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

```
nosetests -s -v tests
```

Note to use the `-s/--nocapture` option to prevent nosetests from captpuring standard output.

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.

Release notes

Release 0.12.0.1

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

Release 0.12.0

This release wraps htslib/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 AligmentFile, Simplify file detection logic and allow remote index files * Removed attempts to guess data and index file names; this is magic left

    to htslib.

    - Removed file existence check prior to opening files with htslib
    - 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 htslib/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 plannig 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 feed-back. The pull-request is here: https://github.com/pysam-developers/pysam/pull/518 Please comment on github.

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

Release 0.11.2.1

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

Release 0.11.2

This release wraps htslib/samtools/bcfools versions 1.4.1 in response to a security fix in these libraries. Additionaly 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

Release 0.11.1

Bugfix release

- [#440] add deprecated 'always' option to infer_query_length for backwards compatibility.

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

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
```

Release 0.9.1

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

- patch included htslib to work with older libcurl versions, fixes #262.
- do not require cython for python 3 install, fixes #260
- FastaFile 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 htslib-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

Release 0.9.0

Overview

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

Htslib 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 htslib 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 htslib library. Previously, each pysam module contained its own version. This reduces compilation time and code bloat.
  - run configure for the builtin htslib library in order to detect optional libraries such as libcurl. Configure behaviour can be controlled by setting the environmet 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.

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 isses 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()
  - getrname() 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 reconsituteReferenceSequence
- 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 chaching of array attributes in AlignedSegment, closes #121
- add export of Fastafile
- 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

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.

Release 0.8.2.1

- Installation bugfix release.

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.

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 (intstead 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

Release 0.8.0

- **Disabled features**
    - IteratorColumn.setMask() disabled as htslib 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, malformatted interval).

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

Release 0.7.7

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

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

Release 0.7.5

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

Release 0.7.4

- further bugfixes to setup.py and package layout

Release 0.7.3

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

Release 0.7.2

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

Release 0.7.1

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

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

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

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

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

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

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 alter 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 aligend 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 htslib 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 bot 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 htslib http://www.htslib.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

Index

A

add_line() (pysam.VariantHeader method), 27
add_meta() (pysam.VariantHeader method), 27
add_record() (pysam.VariantHeader method), 27
add_sample() (pysam.VariantHeader method), 27
aend (pysam.AlignedSegment attribute), 12
alen (pysam.AlignedSegment attribute), 12
aligned_pairs (pysam.AlignedSegment attribute), 12
AlignedSegment (class in pysam), 11
alignment (pysam.PileupRead attribute), 19
AlignmentFile (class in pysam), 5
alleles (pysam.VariantRecord attribute), 27
alts (pysam.VariantHeader attribute), 27
alts (pysam.VariantRecord attribute), 27
asBed (class in pysam), 22
asGTF (class in pysam), 22
asTuple (class in pysam), 21
asVCF (class in pysam), 22
attrs (pysam.VariantHeaderRecord attribute), 28

B

BAM, **52**
BCF, **52**
bgzip, **52**
bin (pysam.AlignedSegment attribute), 12
blocks (pysam.AlignedSegment attribute), 12
build() (pysam.IndexedReads method), 20

C

check_index() (pysam.AlignmentFile method), 7
chrom (pysam.VariantRecord attribute), 28
cigar, **52**
cigar (pysam.AlignedSegment attribute), 12
cigarstring (pysam.AlignedSegment attribute), 12
cigartuples (pysam.AlignedSegment attribute), 12
close() (pysam.AlignmentFile method), 7
close() (pysam.FastaFile method), 23
close() (pysam.FastxFile method), 24
close() (pysam.TabixFile method), 20
close() (pysam.VariantFile method), 26
closed (pysam.FastaFile attribute), 23
closed (pysam.FastxFile attribute), 24
column, **52**
compare() (pysam.AlignedSegment method), 13
contig, **52**
contig (pysam.VariantRecord attribute), 28
contigs (pysam.TabixFile attribute), 20
contigs (pysam.VariantHeader attribute), 27
copy() (pysam.VariantFile method), 26
copy() (pysam.VariantHeader method), 27
copy() (pysam.VariantRecord method), 28
count() (pysam.AlignmentFile method), 7
count_coverage() (pysam.AlignmentFile method), 8
csamtools, **52**

F

faidx, **53**
FastaFile (class in pysam), 23
FastqProxy (class in pysam), 25
FastxFile (class in pysam), 24
fetch() (pysam.AlignmentFile method), 8
fetch() (pysam.FastaFile method), 23
fetch() (pysam.TabixFile method), 20
fetch() (pysam.VariantFile method), 26
fetching, **53**
filename (pysam.FastaFile attribute), 23
filename (pysam.FastxFile attribute), 24
filter (pysam.VariantRecord attribute), 28
filters (pysam.VariantHeader attribute), 27
find() (pysam.IndexedReads method), 20
find_introns() (pysam.AlignmentFile method), 9
flag (pysam.AlignedSegment attribute), 13
format (pysam.VariantRecord attribute), 28
formats (pysam.VariantHeader attribute), 27

G

get() (pysam.VariantHeaderRecord method), 28
get_aligned_pairs() (pysam.AlignedSegment method), 13
get_blocks() (pysam.AlignedSegment method), 13

get_cigar_stats() (pysam.AlignedSegment method), 13
get_index_statistics() (pysam.AlignmentFile method), 9
get_overlap() (pysam.AlignedSegment method), 13
get_quality_array() (pysam.FastqProxy method), 25
get_reference_length() (pysam.FastaFile method), 23
get_reference_name() (pysam.AlignmentFile method), 9
get_reference_name() (pysam.VariantFile method), 26
get_reference_positions() (pysam.AlignedSegment method), 13
get_reference_sequence() (pysam.AlignedSegment method), 14
get_tag() (pysam.AlignedSegment method), 14
get_tags() (pysam.AlignedSegment method), 14
get_tid() (pysam.AlignmentFile method), 9
get_tid() (pysam.VariantFile method), 26
getrname() (pysam.AlignmentFile method), 9
gettid() (pysam.AlignmentFile method), 9

H

hard clipped, **53**
hard clipping, **53**
has_index() (pysam.AlignmentFile method), 9
has_tag() (pysam.AlignedSegment method), 14
head() (pysam.AlignmentFile method), 9
header (pysam.AlignmentFile attribute), 9
header (pysam.TabixFile attribute), 21

I

id (pysam.VariantRecord attribute), 28
indel (pysam.PileupRead attribute), 19
IndexedReads (class in pysam), 19
infer_query_length() (pysam.AlignedSegment method), 14
infer_read_length() (pysam.AlignedSegment method), 14
inferred_length (pysam.AlignedSegment attribute), 15
info (pysam.VariantHeader attribute), 27
info (pysam.VariantRecord attribute), 28
is_del (pysam.PileupRead attribute), 19
is_duplicate (pysam.AlignedSegment attribute), 15
is_head (pysam.PileupRead attribute), 19
is_open() (pysam.FastaFile method), 23
is_open() (pysam.FastxFile method), 25
is_paired (pysam.AlignedSegment attribute), 15
is_proper_pair (pysam.AlignedSegment attribute), 15
is_qcfail (pysam.AlignedSegment attribute), 15
is_read1 (pysam.AlignedSegment attribute), 15
is_read2 (pysam.AlignedSegment attribute), 15
is_refskip (pysam.PileupRead attribute), 19
is_reverse (pysam.AlignedSegment attribute), 15
is_secondary (pysam.AlignedSegment attribute), 15
is_supplementary (pysam.AlignedSegment attribute), 15
is_tail (pysam.PileupRead attribute), 19
is_unmapped (pysam.AlignedSegment attribute), 15
is_valid_tid() (pysam.AlignmentFile method), 10
is_valid_tid() (pysam.VariantFile method), 26
isize (pysam.AlignedSegment attribute), 15
items() (pysam.VariantHeaderRecord method), 28
iteritems() (pysam.VariantHeaderRecord method), 28
iterkeys() (pysam.VariantHeaderRecord method), 28
itervalues() (pysam.VariantHeaderRecord method), 28

K

key (pysam.VariantHeaderRecord attribute), 29
keys() (pysam.VariantHeaderRecord method), 29

L

lengths (pysam.AlignmentFile attribute), 10
lengths (pysam.FastaFile attribute), 24
level (pysam.PileupRead attribute), 19

M

mapped (pysam.AlignmentFile attribute), 10
mapping_quality (pysam.AlignedSegment attribute), 15
mapq (pysam.AlignedSegment attribute), 15
mate() (pysam.AlignmentFile method), 10
mate_is_reverse (pysam.AlignedSegment attribute), 15
mate_is_unmapped (pysam.AlignedSegment attribute), 15
merge() (pysam.VariantHeader method), 27
mpos (pysam.AlignedSegment attribute), 15
mrnm (pysam.AlignedSegment attribute), 15

N

name (pysam.FastqProxy attribute), 25
new_record() (pysam.VariantFile method), 26
new_record() (pysam.VariantHeader method), 27
next (pysam.AlignmentFile attribute), 10
next (pysam.FastxFile attribute), 25
next (pysam.VariantFile attribute), 26
next_reference_id (pysam.AlignedSegment attribute), 15
next_reference_name (pysam.AlignedSegment attribute), 15
next_reference_start (pysam.AlignedSegment attribute), 15
nocoordinate (pysam.AlignmentFile attribute), 10
nreferences (pysam.AlignmentFile attribute), 10
nreferences (pysam.FastaFile attribute), 24
nsegments (pysam.PileupColumn attribute), 19

O

open() (pysam.VariantFile method), 26
opt() (pysam.AlignedSegment method), 15
overlap() (pysam.AlignedSegment method), 16

P

pileup, **53**
pileup() (pysam.AlignmentFile method), 10

PileupColumn (class in pysam), 18
PileupRead (class in pysam), 19
pileups (pysam.PileupColumn attribute), 19
pnext (pysam.AlignedSegment attribute), 16
pop() (pysam.VariantHeaderRecord method), 29
pos (pysam.AlignedSegment attribute), 16
pos (pysam.VariantRecord attribute), 28
positions (pysam.AlignedSegment attribute), 16

Q

qend (pysam.AlignedSegment attribute), 16
qlen (pysam.AlignedSegment attribute), 16
qname (pysam.AlignedSegment attribute), 16
qqual (pysam.AlignedSegment attribute), 16
qstart (pysam.AlignedSegment attribute), 16
qual (pysam.AlignedSegment attribute), 16
qual (pysam.VariantRecord attribute), 28
quality (pysam.FastqProxy attribute), 25
query (pysam.AlignedSegment attribute), 16
query_alignment_end (pysam.AlignedSegment attribute), 16
query_alignment_length (pysam.AlignedSegment attribute), 16
query_alignment_qualities (pysam.AlignedSegment attribute), 16
query_alignment_sequence (pysam.AlignedSegment attribute), 16
query_alignment_start (pysam.AlignedSegment attribute), 17
query_length (pysam.AlignedSegment attribute), 17
query_name (pysam.AlignedSegment attribute), 17
query_position (pysam.PileupRead attribute), 19
query_position_or_next (pysam.PileupRead attribute), 19
query_qualities (pysam.AlignedSegment attribute), 17
query_sequence (pysam.AlignedSegment attribute), 17

R

records (pysam.VariantHeader attribute), 27
ref (pysam.VariantRecord attribute), 28
Reference, **53**
reference_end (pysam.AlignedSegment attribute), 17
reference_id (pysam.AlignedSegment attribute), 17
reference_id (pysam.PileupColumn attribute), 19
reference_length (pysam.AlignedSegment attribute), 17
reference_name (pysam.AlignedSegment attribute), 18
reference_name (pysam.PileupColumn attribute), 19
reference_pos (pysam.PileupColumn attribute), 19
reference_start (pysam.AlignedSegment attribute), 18
references (pysam.AlignmentFile attribute), 11
references (pysam.FastaFile attribute), 24
region, **53**
remove() (pysam.VariantHeaderRecord method), 29
reset() (pysam.VariantFile method), 26
rid (pysam.VariantRecord attribute), 28
rlen (pysam.AlignedSegment attribute), 18
rlen (pysam.VariantRecord attribute), 28
rname (pysam.AlignedSegment attribute), 18
rnext (pysam.AlignedSegment attribute), 18

S

SAM, **53**
sam file, **53**
samples (pysam.VariantHeader attribute), 27
samples (pysam.VariantRecord attribute), 28
samtools, **53**
seq (pysam.AlignedSegment attribute), 18
sequence (pysam.FastqProxy attribute), 25
set_tag() (pysam.AlignedSegment method), 18
set_tags() (pysam.AlignedSegment method), 18
setTag() (pysam.AlignedSegment method), 18
soft clipped, **53**
soft clipping, **53**
start (pysam.VariantRecord attribute), 28
stop (pysam.VariantRecord attribute), 28
subset_samples() (pysam.VariantFile method), 26

T

tabix, **53**
tabix file, **53**
tabix row, **53**
tabix_compress() (in module pysam), 21
tabix_index() (in module pysam), 21
tabix_iterator() (in module pysam), 21
TabixFile (class in pysam), 20
tags (pysam.AlignedSegment attribute), 18
TAM, **53**
target, **53**
template_length (pysam.AlignedSegment attribute), 18
text (pysam.AlignmentFile attribute), 11
tid, **53**
tid (pysam.AlignedSegment attribute), 18
tlen (pysam.AlignedSegment attribute), 18
tostring() (pysam.AlignedSegment method), 18
translate() (pysam.VariantRecord method), 28
type (pysam.VariantHeaderRecord attribute), 29

U

unmapped (pysam.AlignmentFile attribute), 11
update() (pysam.VariantHeaderRecord method), 29

V

value (pysam.VariantHeaderRecord attribute), 29
values() (pysam.VariantHeaderRecord method), 29
VariantFile (class in pysam), 25
VariantHeader (class in pysam), 27
VariantHeaderRecord (class in pysam), 28
VariantRecord (class in pysam), 27

VCF, **53**
version (pysam.VariantHeader attribute), 27

W

write() (pysam.AlignmentFile method), 11
write() (pysam.VariantFile method), 26