

---

# PyVCF Documentation

*Release 0.6.8*

**James Casbon, @jdoughertyii**

February 15, 2017



<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>API</b>	<b>7</b>
2.1	vcf.Reader . . . . .	7
2.2	vcf.Writer . . . . .	8
2.3	vcf.model._Record . . . . .	8
2.4	vcf.model._Call . . . . .	10
2.5	vcf.model._AltRecord . . . . .	11
2.6	vcf.model._Substitution . . . . .	11
2.7	vcf.model._SV . . . . .	11
2.8	vcf.model._SingleBreakend . . . . .	11
2.9	vcf.model._Breakend . . . . .	12
<b>3</b>	<b>Filtering VCF files</b>	<b>13</b>
3.1	The filter script: vcf_filter.py . . . . .	13
3.2	Existing Filters . . . . .	13
3.3	Adding a filter . . . . .	14
3.4	The filter base class: vcf.filters.Base . . . . .	15
<b>4</b>	<b>Utilities</b>	<b>17</b>
4.1	Simultaneously iterate two or more files . . . . .	17
4.2	Trim common suffix . . . . .	17
4.3	vcf_melt . . . . .	18
<b>5</b>	<b>Development</b>	<b>19</b>
5.1	Running tests . . . . .	19
<b>6</b>	<b>Changes</b>	<b>21</b>
6.1	0.6.7 Release . . . . .	21
6.2	0.6.6 Release . . . . .	21
6.3	0.6.5 Release . . . . .	21
6.4	0.6.4 Release . . . . .	21
6.5	0.6.3 Release . . . . .	22
6.6	0.6.2 Release . . . . .	22
6.7	0.6.1 Release . . . . .	22
6.8	0.6.0 Release . . . . .	22
6.9	0.5.0 Release . . . . .	22
6.10	0.4.6 Release . . . . .	23
6.11	0.4.5 Release . . . . .	23

6.12	0.4.4 Release	23
6.13	0.4.3 Release	23
6.14	0.4.2 Release	23
6.15	0.4.1 Release	23
6.16	0.4.0 Release	23
6.17	0.3.0 Release	24
6.18	0.2.2 Release	24
6.19	0.2.1 Release	24
6.20	0.2 Release	24
6.21	0.1 Release	24
<b>7</b>	<b>Contributions</b>	<b>27</b>
<b>8</b>	<b>Indices and tables</b>	<b>29</b>
	<b>Python Module Index</b>	<b>31</b>

Contents:



---

## Introduction

---

A VCFv4.0 and 4.1 parser for Python.

Online version of PyVCF documentation is available at <http://pyvcf.rtfld.org/>

The intent of this module is to mimic the `csv` module in the Python stdlib, as opposed to more flexible serialization formats like JSON or YAML. `vcf` will attempt to parse the content of each record based on the data types specified in the meta-information lines – specifically the `##INFO` and `##FORMAT` lines. If these lines are missing or incomplete, it will check against the reserved types mentioned in the spec. Failing that, it will just return strings.

There main interface is the class: `Reader`. It takes a file-like object and acts as a reader:

```
>>> import vcf
>>> vcf_reader = vcf.Reader(open('vcf/test/example-4.0.vcf', 'r'))
>>> for record in vcf_reader:
...     print record
Record(CHROM=20, POS=14370, REF=G, ALT=[A])
Record(CHROM=20, POS=17330, REF=T, ALT=[A])
Record(CHROM=20, POS=1110696, REF=A, ALT=[G, T])
Record(CHROM=20, POS=1230237, REF=T, ALT=[None])
Record(CHROM=20, POS=1234567, REF=GTCT, ALT=[G, GTACT])
```

This produces a great deal of information, but it is conveniently accessed. The attributes of a `Record` are the 8 fixed fields from the VCF spec:

```
* ``Record.CHROM``
* ``Record.POS``
* ``Record.ID``
* ``Record.REF``
* ``Record.ALT``
* ``Record.QUAL``
* ``Record.FILTER``
* ``Record.INFO``
```

plus attributes to handle genotype information:

- `Record.FORMAT`
- `Record.samples`
- `Record.genotype`

`samples` and `genotype`, not being the title of any column, are left lowercase. The format of the fixed fields is from the spec. Comma-separated lists in the VCF are converted to lists. In particular, one-entry VCF lists are converted to one-entry Python lists (see, e.g., `Record.ALT`). Semicolon-delimited lists of key=value pairs are converted to Python dictionaries, with flags being given a `True` value. Integers and floats are handled exactly as you'd expect:

```
>>> vcf_reader = vcf.Reader(open('vcf/test/example-4.0.vcf', 'r'))
>>> record = next(vcf_reader)
>>> print record.POS
14370
>>> print record.ALT
[A]
>>> print record.INFO['AF']
[0.5]
```

There are a number of convenience methods and properties for each Record allowing you to examine properties of interest:

```
>>> print record.num_called, record.call_rate, record.num_unknown
3 1.0 0
>>> print record.num_hom_ref, record.num_het, record.num_hom_alt
1 1 1
>>> print record.nucl_diversity, record.aaf, record.heterozygosity
0.6 [0.5] 0.5
>>> print record.get_hets()
[Call(sample=NA00002, CallData(GT=1|0, GQ=48, DP=8, HQ=[51, 51]))]
>>> print record.is_snp, record.is_indel, record.is_transition, record.is_deletion
True False True False
>>> print record.var_type, record.var_subtype
snp ts
>>> print record.is_monomorphic
False
```

record.FORMAT will be a string specifying the format of the genotype fields. In case the FORMAT column does not exist, record.FORMAT is None. Finally, record.samples is a list of dictionaries containing the parsed sample column and record.genotype is a way of looking up genotypes by sample name:

```
>>> record = next(vcf_reader)
>>> for sample in record.samples:
...     print sample['GT']
0|0
0|1
0/0
>>> print record.genotype('NA00001')['GT']
0|0
```

The genotypes are represented by Call objects, which have three attributes: the corresponding Record site, the sample name in sample and a dictionary of call data in data:

```
>>> call = record.genotype('NA00001')
>>> print call.site
Record(CHROM=20, POS=17330, REF=T, ALT=[A])
>>> print call.sample
NA00001
>>> print call.data
CallData(GT=0|0, GQ=49, DP=3, HQ=[58, 50])
```

Please note that as of release 0.4.0, attributes known to have single values (such as DP and GQ above) are returned as values. Other attributes are returned as lists (such as HQ above).

There are also a number of methods:

```
>>> print call.called, call.gt_type, call.gt_bases, call.phased
True 0 T|T True
```

Metadata regarding the VCF file itself can be investigated through the following attributes:



- Reader.metadata
- Reader.infos
- Reader.filters
- Reader.formats
- Reader.samples

For example:

```
>>> vcf_reader.metadata['fileDate']
'20090805'
>>> vcf_reader.samples
['NA00001', 'NA00002', 'NA00003']
>>> vcf_reader.filters
OrderedDict([('q10', Filter(id='q10', desc='Quality below 10')), ('s50', Filter(id='s50', desc='Less
>>> vcf_reader.infos['AA'].desc
'Ancestral Allele'
```

ALT records are actually classes, so that you can interrogate them:

```
>>> reader = vcf.Reader(open('vcf/test/example-4.1-bnd.vcf'))
>>> _ = next(reader); row = next(reader)
>>> print row
Record(CHROM=1, POS=2, REF=T, ALT=[T[2:3[])
>>> bnd = row.ALT[0]
>>> print bnd.withinMainAssembly, bnd.orientation, bnd.remoteOrientation, bnd.connectingSequence
True False True T
```

The Reader supports retrieval of records within designated regions for files with tabix indexes via the fetch method. This requires the pysam module as a dependency. Pass in a chromosome, and, optionally, start and end coordinates, for the regions of interest:

```
>>> vcf_reader = vcf.Reader(filename='vcf/test/tb.vcf.gz')
>>> # fetch all records on chromosome 20 from base 1110696 through 1230237
>>> for record in vcf_reader.fetch('20', 1110695, 1230237):
...     print record
Record(CHROM=20, POS=1110696, REF=A, ALT=[G, T])
Record(CHROM=20, POS=1230237, REF=T, ALT=[None])
```

Note that the start and end coordinates are in the zero-based, half-open coordinate system, similar to `_Record.start` and `_Record.end`. The very first base of a chromosome is index 0, and the the region includes bases up to, but not including the base at the end coordinate. For example:

```
>>> # fetch all records on chromosome 4 from base 11 through 20
>>> vcf_reader.fetch('4', 10, 20)
```

would include all records overlapping a 10 base pair region from the 11th base of through the 20th base (which is at index 19) of chromosome 4. It would not include the 21st base (at index 20). (See [http://genomewiki.ucsc.edu/index.php/Coordinate\\_Transforms](http://genomewiki.ucsc.edu/index.php/Coordinate_Transforms) for more information on the zero-based, half-open coordinate system.)

The Writer class provides a way of writing a VCF file. Currently, you must specify a template Reader which provides the metadata:

```
>>> vcf_reader = vcf.Reader(filename='vcf/test/tb.vcf.gz')
>>> vcf_writer = vcf.Writer(open('/dev/null', 'w'), vcf_reader)
>>> for record in vcf_reader:
...     vcf_writer.write_record(record)
```

An extensible script is available to filter vcf files in `vcf_filter.py`. VCF filters declared by other packages will be available for use in this script. Please see [Filtering VCF files](#) for full description.

## 2.1 vcf.Reader

**class** `vcf.Reader` (*fsock=None, filename=None, compressed=None, prepend\_chr=False, strict\_whitespace=False, encoding='ascii'*)

Reader for a VCF v 4.0 file, an iterator returning `_Record` objects

**alts = None**

ALT fields from header

**contigs = None**

contig fields from header

**fetch** (*chrom, start=None, end=None*)

Fetches records from a tabix-indexed VCF file and returns an iterable of `_Record` instances

`chrom` must be specified.

The start and end coordinates are in the zero-based, half-open coordinate system, similar to `_Record.start` and `_Record.end`. The very first base of a chromosome is index 0, and the region includes bases up to, but not including the base at the end coordinate. For example `fetch('4', 10, 20)` would include all variants overlapping a 10 base pair region from the 11th base of through the 20th base (which is at index 19) of chromosome 4. It would not include the 21st base (at index 20). See [http://genomewiki.ucsc.edu/index.php/Coordinate\\_Transforms](http://genomewiki.ucsc.edu/index.php/Coordinate_Transforms) for more information on the zero-based, half-open coordinate system.

If end is omitted, all variants from start until the end of the chromosome `chrom` will be included.

If start and end are omitted, all variants on `chrom` will be returned.

requires `pysam`

**filters = None**

FILTER fields from header

**formats = None**

FORMAT fields from header

**infos = None**

INFO fields from header

**metadata = None**

metadata fields from header (string or hash, depending)

**next** ()

Return the next record in the file.

## 2.2 vcf.Writer

```
class vcf.Writer (stream, template, lineterminator='n')
    VCF Writer. On Windows Python 2, open stream with 'wb'.

    close ()
        Close the writer

    flush ()
        Flush the writer

    write_record (record)
        write a record to the file
```

## 2.3 vcf.model.\_Record

```
class vcf.model._Record (CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, FORMAT, sam-
                        ple_indexes, samples=None)
    A set of calls at a site. Equivalent to a row in a VCF file.
```

The standard VCF fields CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO and FORMAT are available as properties.

The list of genotype calls is in the `samples` property.

Regarding the coordinates associated with each instance:

- POS, per VCF specification, is the one-based index (the first base of the contig has an index of 1) of the first base of the REF sequence.
- The `start` and `end` denote the coordinates of the entire REF sequence in the zero-based, half-open coordinate system (see [http://genomewiki.ucsc.edu/index.php/Coordinate\\_Transforms](http://genomewiki.ucsc.edu/index.php/Coordinate_Transforms)), where the first base of the contig has an index of 0, and the interval runs up to, but does not include, the base at the `end` index. This indexing scheme is analagous to Python slice notation.
- The `affected_start` and `affected_end` coordinates are also in the zero-based, half-open coordinate system. These coordinates indicate the precise region of the reference genome actually affected by the events denoted in ALT (i.e., the minimum `affected_start` and maximum `affected_end`).
  - For SNPs and structural variants, the affected region includes all bases of REF, including the first base (i.e., `affected_start = start = POS - 1`).
  - For deletions, the region includes all bases of REF except the first base, which flanks upstream the actual deletion event, per VCF specification.
  - For insertions, the `affected_start` and `affected_end` coordinates represent a 0 bp-length region between the two flanking bases (i.e., `affected_start = affected_end`). This is analagous to Python slice notation (see <http://stackoverflow.com/a/2947881/38140>). Neither the upstream nor downstream flanking bases are included in the region.

**POS = None**

the one-based coordinate of the first nucleotide in REF

**aaf**

A list of allele frequencies of alternate alleles. NOTE: Denominator calc'ed from `_called_` genotypes.

**affected\_end = None**

zero-based, half-open end coordinate of affected region of reference genome (not included in the region)

**affected\_start = None**

zero-based, half-open start coordinate of affected region of reference genome

**alleles = None**

list of alleles. [0] = REF, [1:] = ALTS

**call\_rate**

The fraction of genotypes that were actually called.

**end = None**

zero-based, half-open end coordinate of REF

**genotype (name)**

Lookup a `_Call` for the sample given in `name`

**get\_hets ()**

The list of het genotypes

**get\_hom\_alts ()**

The list of hom alt genotypes

**get\_hom\_refs ()**

The list of hom ref genotypes

**get\_unknowns ()**

The list of unknown genotypes

**heterozygosity**

Heterozygosity of a site. Heterozygosity gives the probability that two randomly chosen chromosomes from the population have different alleles, giving a measure of the degree of polymorphism in a population.

If there are  $i$  alleles with frequency  $p_i$ ,  $H=1-\sum_i(p_i^2)$

**is\_deletion**

Return whether or not the INDEL is a deletion

**is\_filtered**

Return True if a variant has been filtered

**is\_indel**

Return whether or not the variant is an INDEL

**is\_monomorphic**

Return True for reference calls

**is\_snp**

Return whether or not the variant is a SNP

**is\_sv**

Return whether or not the variant is a structural variant

**is\_sv\_precise**

Return whether the SV coordinates are mapped to 1 b.p. resolution.

**is\_transition**

Return whether or not the SNP is a transition

**nucl\_diversity**

$\pi_{\text{hat}}$  (estimation of nucleotide diversity) for the site. This metric can be summed across multiple sites to compute regional nucleotide diversity estimates. For example,  $\pi_{\text{hat}}$  for all variants in a given gene.

Derived from: "Population Genetics: A Concise Guide, 2nd ed., p.45" John Gillespie.

**num\_called**

The number of called samples

**num\_het**

The number of heterozygous genotypes

**num\_hom\_alt**

The number of homozygous for alt allele genotypes

**num\_hom\_ref**

The number of homozygous for ref allele genotypes

**num\_unknown**

The number of unknown genotypes

**samples = None**

list of `_Calls` for each sample ordered as in source VCF

**start = None**

zero-based, half-open start coordinate of REF

**sv\_end**

Return the end position for the SV

**var\_subtype**

Return the subtype of variant.

- For SNPs and INDELS, yeild one of: [ts, tv, ins, del]
- For SVs yield either “complex” or the SV type defined in the ALT fields (removing the brackets). E.g.:

```
<DEL>      -> DEL
<INS:ME:L1> -> INS:ME:L1
<DUP>      -> DUP
```

The logic is meant to follow the rules outlined in the following paragraph at:

<http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-41>

“For precisely known variants, the REF and ALT fields should contain the full sequences for the alleles, following the usual VCF conventions. For imprecise variants, the REF field may contain a single base and the ALT fields should contain symbolic alleles (e.g. <ID>), described in more detail below. Imprecise variants should also be marked by the presence of an IMPRECISE flag in the INFO field.”

**var\_type**

Return the type of variant [snp, indel, unknown] TO DO: support SVs

## 2.4 vcf.model.\_Call

**class** `vcf.model._Call` (*site, sample, data*)

A genotype call, a cell entry in a VCF file

**data**

Namedtuple of data from the VCF file

**gt\_bases**

The actual genotype alleles. E.g. if VCF genotype is 0/1, return A/G

**gt\_type**

The type of genotype. hom\_ref = 0 het = 1 hom\_alt = 2 (we don;t track \_which+ ALT) uncalled = None

**is\_filtered**

Return True for filtered calls

**is\_het**

Return True for heterozygous calls

**is\_variant**

Return True if not a reference call

**phased**

A boolean indicating whether or not the genotype is phased for this sample

**sample**

The sample name

**site**

The `_Record` for this `_Call`

## 2.5 vcf.model.\_AltRecord

**class** `vcf.model._AltRecord` (*type*, *\*\*kwargs*)

An alternative allele record: either replacement string, SV placeholder, or breakend

**type = None**

String to describe the type of variant, by default “SNV” or “MNV”, but can be extended to any of the types described in the ALT lines of the header (e.g. “DUP”, “DEL”, “INS”...)

## 2.6 vcf.model.\_Substitution

**class** `vcf.model._Substitution` (*nucleotides*, *\*\*kwargs*)

A basic ALT record, where a REF sequence is replaced by an ALT sequence

**sequence = None**

Alternate sequence

## 2.7 vcf.model.\_SV

**class** `vcf.model._SV` (*type*, *\*\*kwargs*)

An SV placeholder

## 2.8 vcf.model.\_SingleBreakend

**class** `vcf.model._SingleBreakend` (*orientation*, *connectingSequence*, *\*\*kwargs*)

A single breakend

## 2.9 vcf.model.\_Breakend

**class** `vcf.parser._Breakend` (*chr, pos, orientation, remoteOrientation, connectingSequence, withinMainAssembly, \*\*kwargs*)

A breakend which is paired to a remote location on or off the genome

**connectingSequence = None**

The breakpoint's connecting sequence.

**orientation = None**

The orientation of breakend. If the sequence 3' of the breakend is connected, True, else if the sequence 5' of the breakend is connected, False.

**remoteOrientation = None**

The orientation of breakend's mate. If the sequence 3' of the breakend's mate is connected, True, else if the sequence 5' of the breakend's mate is connected, False.

**withinMainAssembly = None**

If the breakend mate is within the assembly, True, else False if the breakend mate is on a contig in an ancillary assembly file.



---

## Filtering VCF files

---

### 3.1 The filter script: `vcf_filter.py`

Filtering a VCF file based on some properties of interest is a common enough operation that PyVCF offers an extensible script. `vcf_filter.py` does the work of reading input, updating the metadata and filtering the records.

### 3.2 Existing Filters

**class** `vcf.filters.SiteQuality` (*args*)  
Filter low quality sites

**class** `vcf.filters.VariantGenotypeQuality` (*args*)  
Filters sites with only low quality variants.

It is possible to have a high site quality with many low quality calls. This filter demands at least one call be above a threshold quality.

**class** `vcf.filters.ErrorBiasFilter` (*args*)  
Filter sites that look like correlated sequencing errors.

Some sequencing technologies, notably pyrosequencing, produce mutation hotspots where there is a constant level of noise, producing some reference and some heterozygote calls.

This filter computes a Bayes Factor for each site by comparing the binomial likelihood of the observed allelic depths under:

- A model with constant error equal to the MAF.
- A model where each sample is the ploidy reported by the caller.

The test value is the log of the bayes factor. Higher values are more likely to be errors.

Note: this filter requires `rpy2`

**class** `vcf.filters.DepthPerSample` (*args*)  
Threshold read depth per sample

**class** `vcf.filters.AvgDepthPerSample` (*args*)  
Threshold average read depth per sample (`read_depth / sample_count`)

**class** `vcf.filters.SnpOnly` (*args*)  
Choose only SNP variants

### 3.3 Adding a filter

You can reuse this work by providing a filter class, rather than writing your own filter. For example, lets say I want to filter each site based on the quality of the site. I can create a class like this:

```
import vcf.filters
class SiteQuality(vcf.filters.Base):
    'Filter sites by quality'

    name = 'sq'

    @classmethod
    def customize_parser(self, parser):
        parser.add_argument('--site-quality', type=int, default=30,
                            help='Filter sites below this quality')

    def __init__(self, args):
        self.threshold = args.site_quality

    def __call__(self, record):
        if record.QUAL < self.threshold:
            return record.QUAL
```

This class subclasses `vcf.filters.Base` which provides the interface for VCF filters. The docstring and name are metadata about the parser. The docstring provides the help for the script, and the first line is included in the FILTER metadata when applied to a file.

The `customize_parser` method allows you to add arguments to the script. We use the `__init__` method to grab the argument of interest from the parser. Finally, the `__call__` method processes each record and returns a value if the filter failed. The base class uses the name and threshold to create the filter ID in the VCF file.

To make `vcf_filter.py` aware of the filter, you can either use the local script option or declare an entry point. To use a local script, simply call `vcf_filter`:

```
$ vcf_filter.py --local-script my_filters.py ...
```

To use an entry point, you need to declare a `vcf.filters` entry point in your setup:

```
setup(
    ...
    entry_points = {
        'vcf.filters': [
            'site_quality = module.path:SiteQuality',
        ]
    }
)
```

Either way, when you call `vcf_filter.py`, you should see your filter in the list of available filters:

```
usage: vcf_filter.py [-h] [--no-short-circuit] [--no-filtered]
                  [--output OUTPUT] [--local-script LOCAL_SCRIPT]
                  input filter [filter_args] [filter [filter_args]] ...

Filter a VCF file

positional arguments:
  input                File to process (use - for STDIN) (default: None)
```

```

optional arguments:
  -h, --help                Show this help message and exit. (default: False)
  --no-short-circuit        Do not stop filter processing on a site if any filter
                           is triggered (default: False)
  --output OUTPUT          Filename to output [STDOUT] (default: <open file
                           '<stdout>', mode 'w' at 0x1002841e0>)
  --no-filtered            Output only sites passing the filters (default: False)
  --local-script LOCAL_SCRIPT
                           Python file in current working directory with the
                           filter classes (default: None)

sq:
  Filter sites by quality

  --site-quality SITE_QUALITY
                           Filter sites below this quality (default: 30)

```

### 3.4 The filter base class: `vcf.filters.Base`

**class** `vcf.filters.Base` (*args*)

Base class for `vcf_filter.py` filters.

Use the class docstring to provide the filter description as it appears in `vcf_filter.py`

**classmethod** `customize_parser` (*parser*)

hook to extend argparse parser with custom arguments

**filter\_name** ()

return the name to put in the VCF header, default is name + threshold

**name = 'f'**

name used to activate filter and in VCF headers



Utilities for VCF files.

## 4.1 Simultaneously iterate two or more files

`vcf.utils.walk_together(*readers, **kwargs)`

Simultaneously iterate over two or more VCF readers. For each genomic position with a variant, return a list of size equal to the number of VCF readers. This list contains the VCF record from readers that have this variant, and `None` for readers that don't have it. The caller must make sure that inputs are sorted in the same way and use the same reference otherwise behaviour is undefined.

**Args:**

**`vcf_record_sort_key`:** function that takes a VCF record and returns a tuple that can be used as a key for comparing and sorting VCF records across all readers. This tuple defines what it means for two variants to be equal (eg. whether it's only their position or also their allele values), and implicitly determines the chromosome ordering since the tuple's 1st element is typically the chromosome name (or calculated from it).

## 4.2 Trim common suffix

`vcf.utils.trim_common_suffix(*sequences)`

Trim a list of sequences by removing the longest common suffix while leaving all of them at least one character in length.

Standard convention with VCF is to place an indel at the left-most position, but some tools add additional context to the right of the sequences (e.g. samtools). These common suffixes are undesirable when comparing variants, for example in variant databases.

```
>>> trim_common_suffix('TATATATA', 'TATATA')
['TAT', 'T']
```

```
>>> trim_common_suffix('ACCCCC', 'ACCCCCCC', 'ACCCCCC', 'ACCCCCCCCC')
['A', 'ACCC', 'ACC', 'ACCCC']
```

### 4.3 vcf\_melt

This script converts a VCF file from wide format (many calls per row) to a long format (one call per row). This is useful if you want to grep per sample or for really quick import into, say, a spreadsheet:

```
$ vcf_melt < vcf/test/gatk.vcf
```

SAMPLE	AD	DP	GQ	GT	PL	FILTER	CHROM	POS	REF	ALT	ID
BLANK	6,0	6	18.04	0/0	0,18,211	.	chr22	42522392			G
NA12878	138,107	250	99.0	0/1	1961,0,3049	.	chr22	42522392			G
NA12891	169,77	250	99.0	0/1	1038,0,3533	.	chr22	42522392			G
NA12892	249,0	250	99.0	0/0	0,600,5732	.	chr22	42522392			G
NA19238	248,1	250	99.0	0/0	0,627,6191	.	chr22	42522392			G
NA19239	250,0	250	99.0	0/0	0,615,5899	.	chr22	42522392			G
NA19240	250,0	250	99.0	0/0	0,579,5674	.	chr22	42522392			G
BLANK	13,4	17	62.64	0/1	63,0,296	.	chr22	42522613			G
NA12878	118,127	246	99.0	0/1	2396,0,1719	.	chr22	42522613			G
NA12891	241,0	244	99.0	0/0	0,459,4476	.	chr22	42522613			G
NA12892	161,85	246	99.0	0/1	1489,0,2353	.	chr22	42522613			G
NA19238	110,132	242	99.0	0/1	2561,0,1488	.	chr22	42522613			G
NA19239	106,135	242	99.0	0/1	2613,0,1389	.	chr22	42522613			G
NA19240	116,126	243	99.0	0/1	2489,0,1537	.	chr22	42522613			G

---

## Development

---

Please use the [PyVCF repository](#). Pull requests gladly accepted. Issues should be reported at the [github issue tracker](#).

### 5.1 Running tests

Please check the tests by running them with:

```
python setup.py test
```

New features should have test code sent with them.





## 6.1 0.6.7 Release

- Include missing .pyx files

## 6.2 0.6.6 Release

- better walk together record ordering (Thanks @datagram, #141)

## 6.3 0.6.5 Release

- Better contig handling (#115, #116, #119 thanks Martijn)
- INFO lines with type character (#120, #121 thanks @AndrewUzilov, Martijn)
- Single breakends fix (#126 thanks @pkrushe)
- Speedup by losing ordering of INFO (#128 thanks Martijn)
- HOMSEQ and other missing fields in INFO (#130 thanks Martijn)
- Add aaf property, (thanks @mgymrek #131)
- Custom equality for walk\_together, thanks bow #132
- Change default line encoding to 'n'
- Improved \_\_eq\_\_ (#134, thanks bow)

## 6.4 0.6.4 Release

- Handle INFO fields with multiple values, thanks
- Support writing records without GT data #88, thanks @bow
- Pickleable call data #112, thanks @superbobry
- Write files without FORMAT #95 thanks Martijn
- Strict whitespace mode, thanks Martijn, Lee Lichtenstein and Manawsi Gupta

- Add support for contigs in header, thanks @gcnh and Martijn
- Fix GATK header parsing, thanks @alimanfoo

## 6.5 0.6.3 Release

- cython port of #79
- correct writing of meta lines #84

## 6.6 0.6.2 Release

- issues #78, #79 (thanks Sean, Brad)

## 6.7 0.6.1 Release

- Add strict whitespace mode for well formed VCFs with spaces in sample names (thanks Marco)
- Ignore blank lines in files (thanks Martijn)
- Tweaks for handling missing data (thanks Sean)
- bcftools tests (thanks Martijn)
- record.FILTER is always a list

## 6.8 0.6.0 Release

- Backwards incompatible change: `_Call.data` is now a namedtuple (previously it was a dict)
- Optional cython version, much improved performance.
- Improvements to writer (thanks @cmclean)
- Improvements to inheritance of classes (thanks @lennax)

## 6.9 0.5.0 Release

- VCF 4.1 support: - support missing genotype #28 (thanks @martijnvermaat) - parseALT for svcs #42, #48 (thanks @dzerbino)
- `trim_common_suffix` method #22 (thanks @martijnvermaat)
- Multiple metadata with the same key is stored (#52)
- Writer improvements: - A/G in Number INFO fields #53 (thanks @lennax) - Better output #55 (thanks @cmclean)
- Allow malformed INFO fields #49 (thanks @ilyaminkin)
- Added bayes factor error bias VCF filter
- Added docs on `vcf_melt`

- filters from @libor-m (SNP only, depth per sample, avg depth per sample)
- change to the filter API, use docstring for filter description

## 6.10 0.4.6 Release

- Performance improvements (#47)
- Preserve order of INFO column (#46)

## 6.11 0.4.5 Release

- Support exponent syntax qual values (#43, #44) (thanks @martijnvermaat)
- Preserve order of header lines (#45)

## 6.12 0.4.4 Release

- Support whitespace in sample names
- SV work (thanks @arq5x)
- Python 3 support via 2to3 (thanks @marcelm)
- Improved filtering script, capable of importing local files

## 6.13 0.4.3 Release

- Single floats in Reader.\_sample\_parser not being converted to float #35
- Handle String INFO values when Number=1 in header #34

## 6.14 0.4.2 Release

- Installation problems

## 6.15 0.4.1 Release

- Installation problems

## 6.16 0.4.0 Release

- Package structure
- add `vcf.utils` module with `walk_together` method
- samtools tests

- support Freebayes' non standard '.' for no call
- fix `vcf_melt`
- support monomorphic sites, add `is_monomorphic` method, handle null QUALs
- filter support for files with monomorphic calls
- Values declared as single are no-longer returned in lists
- several performance improvements

## 6.17 0.3.0 Release

- Fix `setup.py` for python < 2.7
- Add `__eq__` to `_Record` and `_Call`
- Add `is_het` and `is_variant` to `_Call`
- Drop aggressive parse mode: we're always aggressive.
- Add `tabix fetch` for single calls, fix one->zero based indexing
- add `prepend_chr` mode for `Reader` to add `chr` to CHROM attributes

## 6.18 0.2.2 Release

Documentation release

## 6.19 0.2.1 Release

- Add shebang to `vcf_filter.py`

## 6.20 0.2 Release

- Replace genotype dictionary with a `Call` object
- Methods on `Record` and `Call` (thanks @arq5x)
- Shortcut `parse_sample` when genotype is `None`

## 6.21 0.1 Release

- Added test code
- Added `Writer` class
- Allow negative number in `INFO` and `FORMAT` fields (thanks @martijnvermaat)
- Prefer `vcf.Reader` to `vcf.VCFReader`
- Support compressed files with guessing where filename is available on `fsock`

- Allow opening by filename as well as filesocket
- Support fetching rows for tabixed indexed files
- Performance improvements (see `test/prof.py`)
- Added extensible filter script (see `FILTERS.md`), `vcf_filter.py`



---

## Contributions

---

Project started by @jdoughertyii and taken over by @jamescasbon on 12th January 2011. Contributions from @arq5x, @brentp, @martijnvermaat, @ian1roberts, @marcelm.

This project was supported by [Population Genetics](#).





---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



**V**

`vcf.utils`, 17



## Symbols

\_AltRecord (class in vcf.model), 11  
 \_Breakend (class in vcf.parser), 12  
 \_Call (class in vcf.model), 10  
 \_Record (class in vcf.model), 8  
 \_SV (class in vcf.model), 11  
 \_SingleBreakend (class in vcf.model), 11  
 \_Substitution (class in vcf.model), 11

## A

aaf (vcf.model.\_Record attribute), 8  
 affected\_end (vcf.model.\_Record attribute), 8  
 affected\_start (vcf.model.\_Record attribute), 8  
 alleles (vcf.model.\_Record attribute), 9  
 alts (vcf.Reader attribute), 7  
 AvgDepthPerSample (class in vcf.filters), 13

## B

Base (class in vcf.filters), 15

## C

call\_rate (vcf.model.\_Record attribute), 9  
 close() (vcf.Writer method), 8  
 connectingSequence (vcf.parser.\_Breakend attribute), 12  
 contigs (vcf.Reader attribute), 7  
 customize\_parser() (vcf.filters.Base class method), 15

## D

data (vcf.model.\_Call attribute), 10  
 DepthPerSample (class in vcf.filters), 13

## E

end (vcf.model.\_Record attribute), 9  
 ErrorBiasFilter (class in vcf.filters), 13

## F

fetch() (vcf.Reader method), 7  
 filter\_name() (vcf.filters.Base method), 15  
 filters (vcf.Reader attribute), 7  
 flush() (vcf.Writer method), 8

formats (vcf.Reader attribute), 7

## G

genotype() (vcf.model.\_Record method), 9  
 get\_hets() (vcf.model.\_Record method), 9  
 get\_hom\_alts() (vcf.model.\_Record method), 9  
 get\_hom\_refs() (vcf.model.\_Record method), 9  
 get\_unknowns() (vcf.model.\_Record method), 9  
 gt\_bases (vcf.model.\_Call attribute), 10  
 gt\_type (vcf.model.\_Call attribute), 10

## H

heterozygosity (vcf.model.\_Record attribute), 9

## I

infos (vcf.Reader attribute), 7  
 is\_deletion (vcf.model.\_Record attribute), 9  
 is\_filtered (vcf.model.\_Call attribute), 11  
 is\_filtered (vcf.model.\_Record attribute), 9  
 is\_het (vcf.model.\_Call attribute), 11  
 is\_indel (vcf.model.\_Record attribute), 9  
 is\_monomorphic (vcf.model.\_Record attribute), 9  
 is\_snp (vcf.model.\_Record attribute), 9  
 is\_sv (vcf.model.\_Record attribute), 9  
 is\_sv\_precise (vcf.model.\_Record attribute), 9  
 is\_transition (vcf.model.\_Record attribute), 9  
 is\_variant (vcf.model.\_Call attribute), 11

## M

metadata (vcf.Reader attribute), 7

## N

name (vcf.filters.Base attribute), 15  
 next() (vcf.Reader method), 7  
 nucl\_diversity (vcf.model.\_Record attribute), 9  
 num\_called (vcf.model.\_Record attribute), 9  
 num\_het (vcf.model.\_Record attribute), 10  
 num\_hom\_alt (vcf.model.\_Record attribute), 10  
 num\_hom\_ref (vcf.model.\_Record attribute), 10  
 num\_unknown (vcf.model.\_Record attribute), 10

## O

orientation (vcf.parser.\_Breakend attribute), 12

## P

phased (vcf.model.\_Call attribute), 11

POS (vcf.model.\_Record attribute), 8

## R

Reader (class in vcf), 7

remoteOrientation (vcf.parser.\_Breakend attribute), 12

## S

sample (vcf.model.\_Call attribute), 11

samples (vcf.model.\_Record attribute), 10

sequence (vcf.model.\_Substitution attribute), 11

site (vcf.model.\_Call attribute), 11

SiteQuality (class in vcf.filters), 13

Snponly (class in vcf.filters), 13

start (vcf.model.\_Record attribute), 10

sv\_end (vcf.model.\_Record attribute), 10

## T

trim\_common\_suffix() (in module vcf.utils), 17

type (vcf.model.\_AltRecord attribute), 11

## V

var\_subtype (vcf.model.\_Record attribute), 10

var\_type (vcf.model.\_Record attribute), 10

VariantGenotypeQuality (class in vcf.filters), 13

vcf.utils (module), 17

## W

walk\_together() (in module vcf.utils), 17

withinMainAssembly (vcf.parser.\_Breakend attribute), 12

write\_record() (vcf.Writer method), 8

Writer (class in vcf), 8