
vcfpy Documentation

Release 0.3.0

Manuel Holtgrewe

September 21, 2016

1	Quick Example	3
2	Frequently Asked Questions	5

VCFPy is a Python 3 module for reading and writing VCF (Variant Call Format) files. The documentation is split into three parts (accessible through the navigation on the left):

Installation & Getting Started Instructions for the installation of the module and some examples to get you started.

API Documentation This section contains the API documentation for the module

Project Info More information on the project, including the changelog, list of contributing authors, and contribution instructions.

Quick Example

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import vcfpy

# Open input, add FILTER header, and open output file
reader = vcfpy.Reader.from_path('input.vcf')
reader.header.add_filter_line(vcfpy.OrderedDict([
    ('ID', 'DP10'), ('Description', 'total DP < 10')]))
writer = vcfpy.Writer.from_path(
    '/dev/stdout', reader.header, reader.samples)

# Add "DP10" filter to records having less than 10 reads
for record in reader:
    ad = sum(c.data.get('DP', 0) for c in record.calls)
    if ad < 10:
        record.add_filter('DP10')
    writer.write_record(record)
```

Frequently Asked Questions

Why another Python library for VCF? I've been using PyVCF with quite some success in the past. However, the main bottleneck of PyVCF is when you want to modify the per-sample genotype information. There are some issues in the tracker of PyVCF but none of them can really be considered solved. I tried several hours to solve these problems within PyVCF but this never got far or towards a complete rewrite...

For this reason, VCFPy was born and here it is!

Why Python 3 only? As I'm only using Python 3 code, I see no advantage in carrying around support for legacy Python 2 and maintaining it. At a later point when VCFPy is known to be stable, Python 2 support might be added if someone contributes a pull request.

What's the state? VCFPy is the result of two full days of development plus some maintenance work later now (right now). I'm using it in several projects but it is not as battle-tested as PyVCF.

What's the difference to PyVCF? The main difference is technical. Instead of using `collections.namedtuple` for storing the call annotation, VCFPy uses `collections.OrderedDict`. This has the advantage that (1) access to optional settings is much more pythonic using `.get(KEY, DEFAULT)` instead of `getattr()`. Further, (2) adding call annotations (FORMAT) fields is able without any performance penalty where for PyVCF, `copy.deepcopy` has to be used at some point which is very slow. There has not been any movement in supporting modifying FORMAT fields in PyVCF and here is a library that does this well.

What's the aim? The aim of the project is to provide simple yet efficient read and write access to VCF files. Eventually, PySAM will probably be a better choice once it has a Python wrapper for the VCF part of `htslib`. However, as this is still missing, VCFPy is a good solution for the time being.

2.1 Installation

2.1.1 Stable release

To install `vcfpy`, run this command in your terminal:

```
$ pip install vcfpy
```

This is the preferred method to install VCFPy, as it will always install the most recent stable release.

If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

2.1.2 From sources

The sources for vcfpy can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/bihealth/vcfpy
```

Or download the [tarball](#):

```
$ curl -OL https://github.com/bihealth/vcfpy/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

2.2 Getting Started

After installation, you can use VCFPy in your project simply by importing the module.

```
import vcfpy
```

That's all, continue and look at the list of examples.

2.3 Examples

This chapter contains several examples for the most important use cases of VCFPy.

2.3.1 Reading VCF Files

The following is an example for reading VCF files and writing out a TSV file with the genotype calls of all SNVs. You can find the example Python and VCF file in the sources below the directory `examples/vcf_to_tsv`.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import vcfpy

# Open file, this will read in the header
reader = vcfpy.Reader.from_path('input.vcf')

# Build and print header
header = ['#CHROM', 'POS', 'REF', 'ALT'] + reader.header.samples.names
print('\t'.join(header))

for record in reader:
    if not record.is_snv():
        continue
    line = [record.CHROM, record.POS, record.REF]
    line += [alt.value for alt in record.ALT]
    line += [call.data.get('GT') or './.' for call in record.calls]
    print('\t'.join(map(str, line)))
```

The program call looks as follows.

```
$ ./vcf_to_tsv.py
#CHROM      POS      REF      ALT      BLANK      NA12878      NA12891      NA12892
chr22      42522392      G      A      0/0      0/1      0/1      0/0      0/0
chr22      42522597      C      T      0/1      0/0      0/0      0/0      0/0
chr22      42522613      G      C      0/1      0/1      0/0      0/1      0/1
chr22      42523003      A      G      0/1      1/1      0/1      0/1      0/1
chr22      42523209      T      C      0/1      1/1      0/1      0/1      0/1
chr22      42523211      T      C      0/0      0/1      0/1      0/0      0/0
chr22      42523409      G      T      0/1      0/1      0/0      0/1      0/1
chr22      42523491      C      T      0/1      0/0      0/0      0/0      0/0
chr22      42523507      A      G      0/1      0/0      0/0      0/0      0/0
chr22      42523805      C      T      0/0      0/0      0/1      0/0      0/0
chr22      42523943      A      G      0/1      1/1      0/1      0/1      0/1
chr22      42524435      T      A      0/1      0/0      0/0      0/0      0/0
[...]
```

2.3.2 Writing VCF Files

The following shows how to add values to the FILTER column to records of an existing VCF file. Adding to existing records is simpler than constructing them from scratch, of course.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import vcfpy

# Open input, add FILTER header, and open output file
reader = vcfpy.Reader.from_path('input.vcf')
reader.header.add_filter_line(vcfpy.OrderedDict([
    ('ID', 'DP10'), ('Description', 'total DP < 10')]))
writer = vcfpy.Writer.from_path(
    '/dev/stdout', reader.header, reader.samples)

# Add "DP10" filter to records having less than 10 reads
for record in reader:
    ad = sum(c.data.get('DP', 0) for c in record.calls)
    if ad < 10:
        record.add_filter('DP10')
    writer.write_record(record)
```

The program call looks as follows.

```
##fileformat=VCFv4.3
##contig=<ID=20,length=62435964>
##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=A,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">
##FILTER=<ID=DP10,Description="total DP < 10">
```

#CHROM	POS	ID	REF	ALT	QUAL	FILTER	INFO
--------	-----	----	-----	-----	------	--------	------

FORMA

20	14370	rs6054257	G	A	29	PASS	NS=3;DP=14;AF=0.5;DB=
20	17330	.	T	A	3	q10	NS=3;DP=11;AF=0.017
20	1110696	rs6040355	A	G,T	67	PASS	NS=2;DP=10;AF=0.1
20	1230237	.	T	.	47	PASS	NS=3;DP=13;AA=T
20	1234567	microsat1	GTC	G,GTCT	50	PASS;DP10	NS=3;DP=10

2.3.3 Jumping in Tabix-indexed Files

The following shows a small program that extracts a genomic region from the input VCF file and writes it to stdout.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import vcfpy

# Open input, add FILTER header, and open output file
reader = vcfpy.Reader.from_path('input.vcf.gz')
writer = vcfpy.Writer.from_path(
    '/dev/stdout', reader.header, reader.samples)

# Fetch region 20:1,110,694-1,230,237. Note that the coordinates
# in the API call are zero-based and describe half-open intervals.
for record in reader.fetch('20', 1110695, 1230237):
    writer.write_record(record)
```

The program call looks as follows.

```
##fileformat=VCFv4.3
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=1,length=249250621>
##contig=<ID=2,length=243199373>
##contig=<ID=20,length=62435964>
##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=A,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">
#CHROM      POS      ID      REF      ALT      QUAL      FILTER      INFO      FORMAT
20          1110696  rs6040355  A      G,T      67        PASS        NS=2;DP=10;AF=0.1
20          1230237  .      T      .      47        PASS        NS=3;DP=13;AA=T      GT:
```

2.4 Best Practice

While not strictly part of the documentation of VCFPy, we include some notes on hints that we consider best practice when building VCF processing applications.

2.4.1 Keep Input Verbatim Where Possible

Try to keep the input verbatim if there is no strong reason for adjusting it. Strong reasons include fixing `Type` or `Number` in header lines describing arrays of strings, for example.

Whenever possible, keep the header order intact. VCFPy does this automatically for you (in contrast to PyVCF).

2.4.2 Prefer Soft-Filters over Hard-Filters

Soft-filters mean annotating your VCF records in the `FILTER` column whereas **Hard**-filters mean removing records from VCF file. In many situations, it is useful to keep around all VCF records and just annotate why they are to be dropped. Then, in the last step, only the interesting ones are kept.

This makes tracing back easier when and why a record was removed.

2.5 Header

Contents

- *Header*
 - *vcfpy.OrderedDict*
 - *vcfpy.Header*
 - *vcfpy.HeaderLine*
 - *vcfpy.header_without_lines*
 - *vcfpy.SimpleHeaderFile*
 - *vcfpy.AltAlleleHeaderLine*
 - *vcfpy.MetaHeaderLine*
 - *vcfpy.PedigreeHeaderLine*
 - *vcfpy.SampleHeaderLine*
 - *vcfpy.ContigHeaderLine*
 - *vcfpy.FilterHeaderLine*
 - *vcfpy.CompoundHeaderLine*
 - *vcfpy.InfoHeaderLine*
 - *vcfpy.FormatHeaderLine*
 - *vcfpy.FieldInfo*
 - *vcfpy.SamplesInfos*

2.5.1 vcfpy.OrderedDict

Convenience export of `OrderedDict`. When available, the `cyordereddict`, a Cython-reimplementation of `OrderedDict` is used for Python before 3.5 (from 3.5, Python ships with a fast, C implementation of `OrderedDict`).

class `vcfpy.OrderedDict`

Dictionary that remembers insertion order

clear () → None. Remove all items from od.

copy () → a shallow copy of od

fromkeys (*S*[, *v*]) → New ordered dictionary with keys from *S*.
If not specified, the value defaults to None.

items () → a set-like object providing a view on D's items

keys () → a set-like object providing a view on D's keys

move_to_end ()
Move an existing element to the end (or beginning if last==False). Raises `KeyError` if the element does not exist. When last=True, acts like a fast version of `self[key]=self.pop(key)`.

pop (*k* [, *d*]) → *v*, remove specified key and return the corresponding value. If key is not found, *d* is returned if given, otherwise `KeyError` is raised.

popitem () → (*k*, *v*), return and remove a (key, value) pair.
Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault (*k* [, *d*]) → *od.get(k,d)*, also set *od[k]=d* if *k* not in *od*

update ([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a `.keys()` method, does: for *k* in E: *D[k] = E[k]* If E present and lacks `.keys()` method, does: for (*k*, *v*) in E: *D[k] = v* In either case, this is followed by: for *k*, *v* in *F.items()*: *D[k] = v*

values () → an object providing a view on D's values

2.5.2 vcfpy.Header

class `vcfpy.Header` (*lines*=[], *samples*=None)
Represent header of VCF file

While this class allows mutating records, it should not be changed once it has been assigned to

This class provides function for adding lines to a header and updating the supporting index data structures. There is no explicit API for removing header lines, the best way is to reconstruct a new `Header` instance with a filtered list of header lines.

add_contig_line (*mapping*)
Add "contig" header line constructed from the given mapping

add_filter_line (*mapping*)
Add FILTER header line constructed from the given mapping

add_format_line (*mapping*)
Add FORMAT header line constructed from the given mapping

add_info_line (*mapping*)
Add INFO header line constructed from the given mapping

add_line (*header_line*)
Add header line, updating any necessary support indices

filter_ids ()
Return list of all filter IDs

format_ids ()
Return list of all format IDs

get_format_field_info (*key*)
Return `FieldInfo` for the given INFO field

get_info_field_info (*key*)
Return `FieldInfo` for the given INFO field

get_lines (*key*)
Return header lines having the given *key* as their type

info_ids()
Return list of all info IDs

lines = None
list of :py:HeaderLine objects

samples = None
SamplesInfo object

2.5.3 vcfpy.HeaderLine

class vcfpy.HeaderLine (*key, value*)
Base class for VCF header lines

key = None
str with key of header line

serialize()
Return VCF-serialized version of this header line

2.5.4 vcfpy.header_without_lines

vcfpy.header_without_lines (*header, remove*)
Return Header without lines given in *remove*

remove is an iterable of pairs key/ID with the VCF header key and ID of entry to remove. In the case that a line does not have a mapping entry, you can give the full value to remove.

2.5.5 vcfpy.SimpleHeaderFile

2.5.6 vcfpy.AltAlleleHeaderLine

class vcfpy.AltAlleleHeaderLine (*key, value, mapping*)
Alternative allele header line

Mostly used for defining symbolic alleles for structural variants and IUPAC ambiguity codes

classmethod from_mapping (*klass, mapping*)
Construct from mapping, not requiring the string value

id = None
name of the alternative allele

2.5.7 vcfpy.MetaHeaderLine

class vcfpy.MetaHeaderLine (*key, value, mapping*)
Alternative allele header line

Used for defining set of valid values for samples keys

classmethod from_mapping (*klass, mapping*)
Construct from mapping, not requiring the string value

id = None
name of the alternative allele

2.5.8 vcfpy.PedigreeHeaderLine

```
class vcfpy.PedigreeHeaderLine(key, value, mapping)
    Header line for defining a pedigree entry

    classmethod from_mapping(klass, mapping)
        Construct from mapping, not requiring the string value

    id = None
        name of the alternative allele
```

2.5.9 vcfpy.SampleHeaderLine

```
class vcfpy.SampleHeaderLine(key, value, mapping)
    Header line for defining a SAMPLE entry

    classmethod from_mapping(klass, mapping)
        Construct from mapping, not requiring the string value

    id = None
        name of the alternative allele
```

2.5.10 vcfpy.ContigHeaderLine

```
class vcfpy.ContigHeaderLine(key, value, mapping)
    Contig header line

    Most importantly, parses the 'length' key into an integer

    classmethod from_mapping(klass, mapping)
        Construct from mapping, not requiring the string value

    id = None
        name of the contig

    length = None
        length of the contig, None if missing
```

2.5.11 vcfpy.FilterHeaderLine

```
class vcfpy.FilterHeaderLine(key, value, mapping)
    FILTER header line

    description = None
        description for the filter, None if missing

    classmethod from_mapping(klass, mapping)
        Construct from mapping, not requiring the string value

    id = None
        token for the filter
```


2.5.12 vcfpy.CompoundHeaderLine

class vcfpy.**CompoundHeaderLine** (*key, value, mapping*)
 Base class for compound header lines, currently format and header lines
 Compound header lines describe fields that can have more than one entry.
mapping = None
 OrderedDict with key/value mapping

2.5.13 vcfpy.InfoHeaderLine

class vcfpy.**InfoHeaderLine** (*key, value, mapping*)
 Header line for INFO fields
 Note that the `Number` field will be parsed into an `int` if possible. Otherwise, the constants `HEADER_NUMBER_*` will be used.
description = None
 description, should be given, None if not given
classmethod **from_mapping** (*klass, mapping*)
 Construct from mapping, not requiring the string value
id = None
 key in the INFO field
source = None
 source of INFO field, None if not given
type = None
 value type
version = None
 version of INFO field, None if not given

2.5.14 vcfpy.FormatHeaderLine

class vcfpy.**FormatHeaderLine** (*key, value, mapping*)
 Header line for FORMAT fields
description = None
 description, should be given, None if not given
classmethod **from_mapping** (*klass, mapping*)
 Construct from mapping, not requiring the string value
id = None
 key in the INFO field
source = None
 source of INFO field, None if not given
type = None
 value type
version = None
 version of INFO field, None if not given

2.5.15 vcfpy.FieldInfo

```
class vcfpy.FieldInfo (type_, number)  
    Core information for describing field type and number  
  
    number = None  
        Number description, either an int or constant  
  
    type = None  
        The type, one of INFO_TYPES or FORMAT_TYPES
```

2.5.16 vcfpy.SamplesInfos

```
class vcfpy.SamplesInfos (sample_names)  
    Helper class for handling and mapping of sample names to numeric indices  
  
    name_to_idx = None  
        mapping from sample name to index  
  
    names = None  
        list of sample names
```

2.6 Input/Output

Contents

- *Input/Output*
 - *vcfpy.Reader*
 - *vcfpy.Writer*

2.6.1 vcfpy.Reader

2.6.2 vcfpy.Writer

2.7 Exceptions

Contents

- *Exceptions*
 - *vcfpy.VCFPyException*
 - *vcfpy.InvalidHeaderException*
 - *vcfpy.InvalidRecordException*
 - *vcfpy.IncorrectVCFFormat*
 - *vcfpy.HeaderNotFound*

2.7.1 vcfpy.VCFPyException

exception `vcfpy.VCFPyException`
Base class for module's exception

2.7.2 vcfpy.InvalidHeaderException

exception `vcfpy.InvalidHeaderException`
Raised in the case of invalid header formatting

2.7.3 vcfpy.InvalidRecordException

exception `vcfpy.InvalidRecordException`
Raised in the case of invalid record formatting

2.7.4 vcfpy.IncorrectVCFFormat

exception `vcfpy.IncorrectVCFFormat`
Raised on problems parsing VCF

2.7.5 vcfpy.HeaderNotFound

exception `vcfpy.HeaderNotFound`
Raised when a VCF header could not be found

2.8 Records

Contents

- *Records*
 - *vcfpy.Record*
 - *vcfpy.Call*
 - *vcfpy.AltRecord*
 - *vcfpy.Substitution*
 - *vcfpy.SV*
 - *vcfpy.BreakEnd*
 - *vcfpy.SingleBreakEnd*
 - *vcfpy.SymbolicAllele*

2.8.1 vcfpy.Record

class `vcfpy.Record` (*CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, FORMAT, calls*)
Represent one record from the VCF file

Record objects are iterators of their calls

ALT = None

A list of alternative allele records of type `AltRecord`

CHROM = None

A `str` with the chromosome name

FILTER = None

A list of strings for the FILTER column

FORMAT = None

A list of strings for the FORMAT column

ID = None

A list of the semicolon-separated values of the ID column

INFO = None

An `OrderedDict` giving the values of the INFO column, flags are mapped to `True`

POS = None

An `int` with a 1-based begin position

QUAL = None

The quality value, can be `None`

REF = None

A `str` with the REF value

add_filter (*label*)

Add label to FILTER if not set yet

add_format (*key*, *value=None*)

Add an entry to format

The record's calls `data[key]` will be set to `value` if not yet set and `value` is not `None`. If `key` is already in `FORMAT` then nothing is done.

affected_end

Return affected start position in 0-based coordinates

For SNVs, MNVs, and deletions, the behaviour is based on the start position and the length of the REF. In the case of insertions, the position behind the insert position is returned, yielding a 0-length interval together with **:py:method:'affected_start'**

affected_start

Return affected start position in 0-based coordinates

For SNVs, MNVs, and deletions, the behaviour is the start position. In the case of insertions, the position behind the insert position is returned, yielding a 0-length interval together with **:py:method:'affected_end'**

begin = None

An `int` with a 0-based begin position

call_for_sample = None

A mapping from sample name to entry in `self.calls`

calls = None

A list of genotype `Call` objects

end = None

An `int` with a 0-based end position

is_snv()

Return `True` if it is a SNV

2.8.2 vcfpy.Call

class `vcfpy.Call` (*sample, data, site=None*)

The information for a genotype callable

By VCF, this should always include the genotype information and can contain an arbitrary number of further annotation, e.g., the coverage at the variant position.

called = None

whether or not the variant is fully called

data = None

an OrderedDict with the key/value pair information from the call's data

gt_alleles = None

the allele numbers (0, 1, ...) in this calls or None for no-call

gt_bases

Return the actual genotype alleles, e.g. if VCF genotype is 0/1, could return A/T

gt_phase_char ()

Return character to use for phasing

gt_type

The type of genotype, mapping is

- hom_ref = 0
- het = 1
- hom_alt = 2 (which alt is untracked)
- uncalled = None

is_filtered

Return True for filtered calls

is_het

Return True for filtered calls

is_phased

Return True for phased calls

is_variant

Return True for filtered calls

phased

Return boolean indicating whether this call is phased

plodity = None

the number of alleles in this sample's call

sample = None

the name of the sample for which the call was made

site = None

the Record of this Call

2.8.3 vcfpy.AltRecord

class `vcfpy.AltRecord` (*type=None*)

An alternative allele Record

Currently, can be a substitution, an SV placeholder, or breakend

type = None

String describing the type of the variant, could be one of SNV, MNV, could be any of the types described in the ALT header lines, such as DUP, DEL, INS, ...

2.8.4 vcfpy.Substitution

class vcfpy.Substitution(*type_*, *value*)

A basic alternative allele record describing a REF->AltRecord substitution

Note that this subsumes MNVs, insertions, and deletions.

value = None

The alternative base sequence to use in the substitution

2.8.5 vcfpy.SV

class vcfpy.SV(*type_*, *value*)

Code for structural variant allele

value = None

The alternative base sequence to use in the substitution

2.8.6 vcfpy.BreakEnd

class vcfpy.BreakEnd(*type_*, *value*)

A placeholder for a breakend

value = None

The alternative base sequence to use in the substitution

2.8.7 vcfpy.SingleBreakEnd

class vcfpy.SingleBreakEnd(*type_*, *value*)

A placeholder for a single breakend

value = None

The alternative base sequence to use in the substitution

2.8.8 vcfpy.SymbolicAllele

class vcfpy.SymbolicAllele(*type_*, *value*)

A placeholder for a symbolic allele

value = None

The alternative base sequence to use in the substitution

2.9 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.9.1 Types of Contributions

Report Bugs

Report bugs at <https://github.com/bihealth/vcfpy/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

vcfpy could always use more documentation, whether as part of the official vcfpy docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/bihealth/vcfpy/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.9.2 Get Started!

Ready to contribute? Here’s how to set up *vcfpy* for local development.

1. Fork the *vcfpy* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/vcfpy.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv vcfpy
$ cd vcfpy/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 vcfpy tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.9.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.3, 3.4 and 3.5. Check https://travis-ci.org/bihealth/vcfpy/pull_requests and make sure that the tests pass for all supported Python versions.

2.9.4 Tips

To run a subset of tests:

```
$ py.test tests.test_vcfpy
```

2.10 Credits

2.10.1 Development Lead

- Manuel Holtgrewe <manuel.holtgrewe@bihealth.de>

2.10.2 Contributors

None yet. Why not be the first?

2.11 History

2.11.1 HEAD

- Reader and Writer can now be used as context manager (with with)
- Including license in documentation, including Biopython license
- Adding support for writing bgzf files (taken from Biopython)
- Adding support for parsing arrays in header lines
- Removing example-4.1-bnd.vcf example file because v4.1 tumor derival lacks ID field
- Adding AltAlleleHeaderLine, MetaHeaderLine, PedigreeHeaderLine, and SampleHeaderLine
- Renaming SimpleHeaderFile to SimpleHeaderLine
- Warn on missing FILTER entries on parsing
- Reordered parameters in from_stream and from_file (#18)
- Renamed from_file to from_stream (#18)
- Renamed Reader.jump_to to Reader.fetch
- Adding header_without_lines function
- Generally extending API to make it esier to use
- Upgrading dependencies, enabling pyup-bot
- Greatly extending documentation

2.11.2 0.2.1 (2016-09-19)

- First release on PyPI

2.12 License

2.12.1 VCFPy License

You can find the License of VCFPy below.

MIT License

Copyright (c) 2016, Berlin Institute of Health

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies

of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2.12.2 Biopython License Agreement

The bgzf writing code is taken from the Biopython project. You can find a copy of the license below.

Biopython License Agreement

Permission to use, copy, modify, and distribute this software and its documentation with or without modifications and for any purpose and without fee is hereby granted, provided that any copyright notices appear in all copies and that both those copyright notices and this permission notice appear in supporting documentation, and that the names of the contributors or copyright holders not be used in advertising or publicity pertaining to distribution of the software without specific prior permission.

THE CONTRIBUTORS AND COPYRIGHT HOLDERS OF THIS SOFTWARE DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

A

add_contig_line() (vcfpy.Header method), 10
 add_filter() (vcfpy.Record method), 16
 add_filter_line() (vcfpy.Header method), 10
 add_format() (vcfpy.Record method), 16
 add_format_line() (vcfpy.Header method), 10
 add_info_line() (vcfpy.Header method), 10
 add_line() (vcfpy.Header method), 10
 affected_end (vcfpy.Record attribute), 16
 affected_start (vcfpy.Record attribute), 16
 ALT (vcfpy.Record attribute), 15
 AltAlleleHeaderLine (class in vcfpy), 11
 AltRecord (class in vcfpy), 17

B

begin (vcfpy.Record attribute), 16
 BreakEnd (class in vcfpy), 18

C

Call (class in vcfpy), 17
 call_for_sample (vcfpy.Record attribute), 16
 called (vcfpy.Call attribute), 17
 calls (vcfpy.Record attribute), 16
 CHROM (vcfpy.Record attribute), 16
 clear() (vcfpy.OrderedDict method), 9
 CompoundHeaderLine (class in vcfpy), 13
 ContigHeaderLine (class in vcfpy), 12
 copy() (vcfpy.OrderedDict method), 9

D

data (vcfpy.Call attribute), 17
 description (vcfpy.FilterHeaderLine attribute), 12
 description (vcfpy.FormatHeaderLine attribute), 13
 description (vcfpy.InfoHeaderLine attribute), 13

E

end (vcfpy.Record attribute), 16

F

FieldInfo (class in vcfpy), 14

FILTER (vcfpy.Record attribute), 16
 filter_ids() (vcfpy.Header method), 10
 FilterHeaderLine (class in vcfpy), 12
 FORMAT (vcfpy.Record attribute), 16
 format_ids() (vcfpy.Header method), 10
 FormatHeaderLine (class in vcfpy), 13
 from_mapping() (vcfpy.AltAlleleHeaderLine class method), 11
 from_mapping() (vcfpy.ContigHeaderLine class method), 12
 from_mapping() (vcfpy.FilterHeaderLine class method), 12
 from_mapping() (vcfpy.FormatHeaderLine class method), 13
 from_mapping() (vcfpy.InfoHeaderLine class method), 13
 from_mapping() (vcfpy.MetaHeaderLine class method), 11
 from_mapping() (vcfpy.PedigreeHeaderLine class method), 12
 from_mapping() (vcfpy.SampleHeaderLine class method), 12
 fromkeys() (vcfpy.OrderedDict method), 9

G

get_format_field_info() (vcfpy.Header method), 10
 get_info_field_info() (vcfpy.Header method), 10
 get_lines() (vcfpy.Header method), 10
 gt_alleles (vcfpy.Call attribute), 17
 gt_bases (vcfpy.Call attribute), 17
 gt_phase_char() (vcfpy.Call method), 17
 gt_type (vcfpy.Call attribute), 17

H

Header (class in vcfpy), 10
 header_without_lines() (in module vcfpy), 11
 HeaderLine (class in vcfpy), 11
 HeaderNotFound, 15

I

id (vcfpy.AltAlleleHeaderLine attribute), 11

id (vcfpy.ContigHeaderLine attribute), 12
id (vcfpy.FilterHeaderLine attribute), 12
id (vcfpy.FormatHeaderLine attribute), 13
id (vcfpy.InfoHeaderLine attribute), 13
id (vcfpy.MetaHeaderLine attribute), 11
id (vcfpy.PedigreeHeaderLine attribute), 12
ID (vcfpy.Record attribute), 16
id (vcfpy.SampleHeaderLine attribute), 12
IncorrectVCFFormat, 15
INFO (vcfpy.Record attribute), 16
info_ids() (vcfpy.Header method), 10
InfoHeaderLine (class in vcfpy), 13
InvalidHeaderException, 15
InvalidRecordException, 15
is_filtered (vcfpy.Call attribute), 17
is_het (vcfpy.Call attribute), 17
is_phased (vcfpy.Call attribute), 17
is_snv() (vcfpy.Record method), 16
is_variant (vcfpy.Call attribute), 17
items() (vcfpy.OrderedDict method), 9

K

key (vcfpy.HeaderLine attribute), 11
keys() (vcfpy.OrderedDict method), 10

L

length (vcfpy.ContigHeaderLine attribute), 12
lines (vcfpy.Header attribute), 11

M

mapping (vcfpy.CompoundHeaderLine attribute), 13
MetaHeaderLine (class in vcfpy), 11
move_to_end() (vcfpy.OrderedDict method), 10

N

name_to_idx (vcfpy.SamplesInfos attribute), 14
names (vcfpy.SamplesInfos attribute), 14
number (vcfpy.FieldInfo attribute), 14

O

OrderedDict (class in vcfpy), 9

P

PedigreeHeaderLine (class in vcfpy), 12
phased (vcfpy.Call attribute), 17
plodity (vcfpy.Call attribute), 17
pop() (vcfpy.OrderedDict method), 10
popitem() (vcfpy.OrderedDict method), 10
POS (vcfpy.Record attribute), 16

Q

QUAL (vcfpy.Record attribute), 16

R

Record (class in vcfpy), 15
REF (vcfpy.Record attribute), 16

S

sample (vcfpy.Call attribute), 17
SampleHeaderLine (class in vcfpy), 12
samples (vcfpy.Header attribute), 11
SamplesInfos (class in vcfpy), 14
serialize() (vcfpy.HeaderLine method), 11
setdefault() (vcfpy.OrderedDict method), 10
SingleBreakEnd (class in vcfpy), 18
site (vcfpy.Call attribute), 17
source (vcfpy.FormatHeaderLine attribute), 13
source (vcfpy.InfoHeaderLine attribute), 13
Substitution (class in vcfpy), 18
SV (class in vcfpy), 18
SymbolicAllele (class in vcfpy), 18

T

type (vcfpy.AltRecord attribute), 18
type (vcfpy.FieldInfo attribute), 14
type (vcfpy.FormatHeaderLine attribute), 13
type (vcfpy.InfoHeaderLine attribute), 13

U

update() (vcfpy.OrderedDict method), 10

V

value (vcfpy.BreakEnd attribute), 18
value (vcfpy.SingleBreakEnd attribute), 18
value (vcfpy.Substitution attribute), 18
value (vcfpy.SV attribute), 18
value (vcfpy.SymbolicAllele attribute), 18
values() (vcfpy.OrderedDict method), 10
VCFPyException, 15
version (vcfpy.FormatHeaderLine attribute), 13
version (vcfpy.InfoHeaderLine attribute), 13