vcfpy Documentation

Release 0.8.1+0.gea51ef6.dirty

Manuel Holtgrewe

February 08, 2017

VCFPy is a Python 3 library with good support for both reading and writing VCF 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

```python
#!/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)
```

Features

- Support for reading and writing VCF v4.3
- Interface to `INFO` and `FORMAT` fields is based on `OrderedDict` allows for easier modification than PyVCF (also I find this more pythonic)
- Read (and jump in) and write BGZF files just using `vcfpy`

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 misssing, `VCFPy` is a good solution for the time being.

3.1 Installation

3.1.1 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.

3.1.2 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
```

3.2 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.

3.3 Examples

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

3.3.1 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`.

```python
#!/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
[...]
```

3.3.2 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.

```python
#!/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      FORMAT
```

```
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          GT:C
20          1110696        rs6040355       A          G,T          67          PASS         NS=2;DP=10;AF=0.3
20          1230237        .        T          .          47          PASS         NS=3;DP=13;AA=T          GT:C
20          1234567        microsat1       GTC          G,GTCT          50          PASS;DP10          NS=3;DP
```

3.3.3 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.

```python
#!/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        FORMA
20          1110696        rs6040355       A          G,T          67          PASS         NS=2;DP=10;AF=0.3
20          1230237        .        T          .          47          PASS         NS=3;DP=13;AA=T          GT:C
```

3.4 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.

3.4.1 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).

3.4.2 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.

3.5 Header

Contents

3.5.1 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

3.5.2 vcfpy.Header

class vcfpy.**Header**(*lines=None*, *samples=None*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
> 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

> > > **Parameters mapping** – OrderedDict with mapping to add. It is recommended to use OrderedDict over dict as this makes the result reproducible

> > > **Returns** False on conflicting line and True otherwise

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

> > > **Parameters mapping** – OrderedDict with mapping to add. It is recommended to use OrderedDict over dict as this makes the result reproducible

> > > **Returns** False on conflicting line and True otherwise

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

> > > **Parameters mapping** – OrderedDict with mapping to add. It is recommended to use OrderedDict over dict as this makes the result reproducible

> > > **Returns** False on conflicting line and True otherwise

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

> > > **Parameters mapping** – OrderedDict with mapping to add. It is recommended to use OrderedDict over dict as this makes the result reproducible

> > **Returns** `False` on conflicting line and `True` otherwise

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

> > **Returns** `False` on conflicting line and `True` otherwise

> **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

> **has_header_line**(*key*, *id_*)
> > Return whether there is a header line with the given ID of the type given by `key`

> > **Parameters**
> >
> > - **key** – The VCF header key/line type.
> >
> > - **id** – The ID value to compare fore
> >
> > **Returns** `True` if there is a header line starting with `##${key}=` in the VCF file having the mapping entry ID set to id_.

> **info_ids**()
> > Return list of all info IDs

> **lines = None**
> > `list` of :py:HeaderLine objects

> **samples = None**
> > `SamplesInfo` object

3.5.3 vcfpy.HeaderLine

class vcfpy.**HeaderLine**(*key*, *value*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
> Base class for VCF header lines

> **key = None**
> > `str` with key of header line

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

> **warning_helper = None**
> > Helper for printing warnings

3.5.4 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.

```python
# header is a vcfpy.Header, e.g., as read earlier from file
new_header = vcfpy.without_header_lines(
    header, [('assembly', None), ('FILTER', 'PASS')])
# now, the header lines starting with "##assembly=" and the "PASS"
# filter line will be missing from new_header
```

3.5.5 vcfpy.SimpleHeaderLine

class `vcfpy.`**`SimpleHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)

Base class for simple header lines, currently contig and filter header lines

Don't use this class directly but rather the sub classes.

> **Raises** `vcfpy.exceptions.InvalidHeaderException` in the case of missing key `"ID"`

`mapping` = **None**
> `collections.OrderedDict` with key/value mapping of the attributes

3.5.6 vcfpy.AltAlleleHeaderLine

class `vcfpy.`**`AltAlleleHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)

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

3.5.7 vcfpy.MetaHeaderLine

class `vcfpy.`**`MetaHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)

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

3.5.8 vcfpy.PedigreeHeaderLine

class `vcfpy.`**`PedigreeHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)

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

3.5.9 vcfpy.SampleHeaderLine

class vcfpy.**SampleHeaderLine**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
: 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

3.5.10 vcfpy.ContigHeaderLine

class vcfpy.**ContigHeaderLine**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
: 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

3.5.11 vcfpy.FilterHeaderLine

class vcfpy.**FilterHeaderLine**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
: 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

3.5.12 vcfpy.CompoundHeaderLine

class vcfpy.**CompoundHeaderLine**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
: Base class for compound header lines, currently format and header lines

Compound header lines describe fields that can have more than one entry.

Don't use this class directly but rather the sub classes.

mapping = None
> OrderedDict with key/value mapping

3.5.13 vcfpy.InfoHeaderLine

class `vcfpy.`**`InfoHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
> 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

3.5.14 vcfpy.FormatHeaderLine

class `vcfpy.`**`FormatHeaderLine`**(*key*, *value*, *mapping*, *warning_helper=<vcfpy.warn_utils.WarningHelper object>*)
> 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

3.5.15 vcfpy.FieldInfo

class `vcfpy.`**`FieldInfo`**(*type_*, *number*, *description=None*)
    Core information for describing field type and number

    `description` = **None**
        Description for the header field, optional

    `number` = **None**
        Number description, either an int or constant

    `type` = **None**
        The type, one of INFO_TYPES or FORMAT_TYPES

3.5.16 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

3.6 Input/Output

Contents

3.6.1 vcfpy.Reader

class `vcfpy.`**`Reader`**(*stream*, *path=None*, *tabix_path=None*, *record_checks=None*)
    Class for parsing of files from `file`-like objects

    Instead of using the constructor, use the class methods `from_stream()` and `from_path()`.

    On construction, the header will be read from the file which can cause problems. After construction, `Reader` can be used as an iterable of `Record`.

        Raises `InvalidHeaderException` in the case of problems reading the header

    `close`()
        Close underlying stream

    `fetch`(*chrom_or_region*, *begin=None*, *end=None*)
        Jump to the start position of the given chromosomal position and limit iteration to the end position

        Parameters

            • **chrom_or_region** (`str`) – name of the chromosome to jump to if begin and end are given and a samtools region string otherwise (e.g. "chr1:123,456-123,900").

- **begin** (*int*) – 0-based begin position (inclusive)

- **end** (*int*) – 0-based end position (exclusive)

classmethod **from_path** (*klass*, *path*, *tabix_path=None*, *record_checks=None*)
> Create new `Reader` from path

>> **Parameters**

>>> - **path** – the path to load from (converted to `str` for compatibility with `path.py`)

>>> - **tabix_path** – optional string with path to TBI index, automatic inferral from `path` will be tried on the fly if not given

>>> - **record_checks** (*list*) – record checks to perform, can contain 'INFO' and 'FOR-MAT'

classmethod **from_stream** (*klass*, *stream*, *path=None*, *tabix_path=None*, *record_checks=None*)
> Create new `Reader` from file

>> **Parameters**

>>> - **stream** – `file`-like object to read from

>>> - **path** – optional string with path to store (for display only)

>>> - **record_checks** (*list*) – record checks to perform, can contain 'INFO' and 'FOR-MAT'

header = None
> the Header

parser = None
> the parser to use

path = None
> optional `str` with the path to the stream

record_checks = None
> checks to perform on records, can contain 'FORMAT' and 'INFO'

samples = None
> the `vcfpy.header.SamplesInfos` object with the sample name information

stream = None
> stream (`file`-like object) to read from

tabix_file = None
> the `pysam.TabixFile` used for reading from index bgzip-ed VCF; constructed on the fly

tabix_path = None
> optional `str` with path to tabix file

3.6.2 vcfpy.Writer

class vcfpy.**Writer** (*stream*, *header*, *samples*, *path=None*)
> Class for writing VCF files to `file`-like objects

> Instead of using the constructor, use the class methods `from_stream()` and `from_path()`.

> The writer has to be constructed with a `Header` object and the full VCF header will be written immediately on construction. This, of course, implies that modifying the header after construction is illegal.

close()
> Close underlying stream

classmethod **from_path**(*klass*, *path*, *header*)
> Create new `Writer` from path

> > **Parameters**

> > - **path** – the path to load from (converted to `str` for compatibility with `path.py`)

> > - **header** – VCF header to use

> > - **samples** – SamplesInfos to use

classmethod **from_stream**(*klass*, *stream*, *header*, *path=None*, *use_bgzf=None*)
> Create new `Writer` from file

> Note that for getting bgzf support, you have to pass in a stream opened in binary mode. Further, you either have to provide a `path` ending in `".gz"` or set `use_bgzf=True`. Otherwise, you will get the notorious "TypeError: 'str' does not support the buffer interface".

> > **Parameters**

> > - **stream** – `file`-like object to write to

> > - **header** – VCF header to use

> > - **path** – optional string with path to store (for display only)

> > - **use_bgzf** – indicator whether to write bgzf to `stream` if `True`, prevent if `False`, interpret `path` if `None`

header = None
> the :py:class:~vcfpy.header.Header' written out

path = None
> optional `str` with the path to the stream

stream = None
> stream (`file`-like object) to read from

write_record(*record*)
> Write out the given `vcfpy.record.Record` to this Writer

3.7 Exceptions

> **Contents**
>
> - *Exceptions*

3.7.1 vcfpy.VCFPyException

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

3.7.2 vcfpy.InvalidHeaderException

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

3.7.3 vcfpy.InvalidRecordException

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

3.7.4 vcfpy.IncorrectVCFFormat

exception `vcfpy.`**`IncorrectVCFFormat`**
    Raised on problems parsing VCF

3.7.5 vcfpy.HeaderNotFound

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

3.8 Records

Contents

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

3.8.1 Record-Related Constants

The following constants are also available as `vcfpy.CONSTANT`.

`vcfpy.record.`**`HOM_REF = 0`**
    Code for homozygous reference

vcfpy.record.**HOM_ALT = 2**
    Code for homozygous alternative

vcfpy.record.**FIVE_PRIME = '5'**
    code for five prime orientation `BreakEnd`

vcfpy.record.**THREE_PRIME = '3'**
    code for three prime orientation `BreakEnd`

vcfpy.record.**FORWARD = '+'**
    code for forward orientation

vcfpy.record.**REVERSE = '-'**
    code for reverse orientation

3.8.2 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, removing `PASS` entry if present

    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 `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 `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

3.8.3 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 bases, 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, returns one of `HOM_REF`, `HOM_ALT`, and `HET`.

is_filtered(*require=None*, *ignore=None*)
> Return `True` for filtered calls

> > **Parameters**
> >
> > - **ignore** (*iterable*) – if set, the filters to ignore, make sure to include 'PASS', when setting, default is `['PASS']`
> >
> > - **require** (*iterable*) – if set, the filters to require for returning `True`

is_het
> Return `True` for heterozygous calls

is_phased
>   Return boolean indicating whether this call is phased

is_variant
>   Return `True` for non-hom-ref calls

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`

3.8.4 vcfpy.AltRecord

class `vcfpy.`**AltRecord**(*type_=None*)
>   An alternative allele Record

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

>   **serialize**()
>   >   Return `str` with representation for VCF file

>   **type** = None
>   >   String describing the type of the variant, could be one of SNV, MNV, could be any of teh types described
>   >   in the ALT header lines, such as DUP, DEL, INS, ...

3.8.5 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

3.8.6 vcfpy.SV

`vcfpy.`**SV**

3.8.7 vcfpy.BreakEnd

class `vcfpy.`**BreakEnd**(*mate_chrom*, *mate_pos*, *orientation*, *mate_orientation*, *sequence*, *within_main_assembly*)
>   A placeholder for a breakend

>   **mate_chrom** = None
>   >   chromosome of the mate breakend

>   **mate_orientation** = None
>   >   orientation breakend's mate

>   **mate_pos** = None
>   >   position of the mate breakend

> **orientation** = None
> > orientation of this breakend
>
> **sequence** = None
> > breakpoint's connecting sequence
>
> **serialize**()
> > Return string representation for VCF
>
> **within_main_assembly** = None
> > `bool` specifying if the breakend mate is within the assembly (`True`) or in an ancillary assembly (`False`)

3.8.8 vcfpy.SingleBreakEnd

class `vcfpy.`**`SingleBreakEnd`**(*orientation*, *sequence*)
> A placeholder for a single breakend

3.8.9 vcfpy.SymbolicAllele

class `vcfpy.`**`SymbolicAllele`**(*value*)
> A placeholder for a symbolic allele
>
> The allele symbol must be defined in the header using an `ALT` header before being parsed. Usually, this is used for succinct descriptions of structural variants or IUPAC parameters.
>
> **value** = None
> > The symbolic value, e.g. 'DUP'

3.9 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:

3.9.1 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 :)

3.9.2 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.

3.9.3 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.

3.9.4 Tips

To run a subset of tests:

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

3.10 Credits

3.10.1 Development Lead

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

3.10.2 Contributors

None yet. Why not be the first?

3.11 History

3.11.1 0.8.1 (2017-02-08)

- PEP8 style adjustments
- Using versioneer for versioning
- Using `requirements*.txt` files now from setup.py
- Fixing dependency on cyordereddict to be for Python <3.6 instead of <3.5
- Jumping by samtools coordinate string now also allowed

3.11.2 0.8.0 (2016-10-31)

- Adding `Header.has_header_line` for querying existence of header line
- `Header.add_*_line` return a `bool` no indicating any conflicts
- Construction of Writer uses samples within header and no extra parameter (breaks API)

3.11.3 0.7.0 (2016-09-25)

- Smaller improvements and fixes to documentation
- Adding Codacy coverage and static code analysis results to README
- Various smaller code cleanup triggered by Codacy results
- Adding `__eq__`, `__neq__` and `__hash__` to data types (where applicable)

3.11.4 0.6.0 (2016-09-25

- Refining implementation for breakend and symbolic allele class
- Removing `record.SV_CODES`
- Refactoring parser module a bit to make the code cleaner
- Fixing small typos and problems in documentation

3.11.5 0.5.0 (2016-09-24)

- Deactivating warnings on record parsing by default because of performance
- Adding validation for `INFO` and `FORMAT` fields on reading (#8)
- Adding predefined `INFO` and `FORMAT` fields to `pyvcf.header` (#32)

3.11.6 0.4.1 (2016-09-22)

- Initially enabling codeclimate

3.11.7 0.4.0 (2016-09-22)

- Exporting constants for encoding variant types
- Exporting genotype constants `HOM_REF`, `HOM_ALT`, `HET`
- Implementing `Call.is_phased`, `Call.is_het`, `Call.is_variant`, `Call.is_phased`, `Call.is_hom_ref`, `Call.is_hom_alt`
- Removing `Call.phased` (breaks API, next release is 0.4.0)
- Adding tests, fixing bugs for methods of `Call`

3.11.8 0.3.1 (2016-09-21)

- Work around `FORMAT/FT` being a string; this is done so in the Delly output

3.11.9 0.3.0 (2016-09-21)

- `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

3.11.10 0.2.1 (2016-09-19)

- First release on PyPI

3.12 License

3.12.1 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.
```

3.12.2 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

B

C

D

E

F

G

U

V

W