HTSeq Documentation

Release 0.8.0

Simon Anders

Jul 04, 2017

Contents

CHAPTER 1

HTSeq: Analysing high-throughput sequencing data with Python

HTSeq is a Python package that provides infrastructure to process data from high-throughput sequencing assays.

- Please see the chapter *A tour through HTSeq* first for an overview on the kind of analysis you can do with HTSeq and the design of the package, and then look at the reference documentation.
- While the main purpose of HTSeq is to allow you to write your own analysis scripts, customized to your needs, there are also a couple of stand-alone scripts for common tasks that can be used without any Python knowledge. See the *Scripts* section in the overview below for what is available.
- For downloads and installation instructions, see *Prequisites and installation*.

Paper

HTSeq is described in the following publication:

Simon Anders, Paul Theodor Pyl, Wolfgang Huber
HTSeq — A Python framework to work with high-throughput sequencing data
Bioinformatics (2014), in print, online at doi:10.1093/bioinformatics/btu638

If you use HTSeq in research, please cite this paper in your publication.

Documentation overview

- *Prequisites and installation*

  Download links and installation instructions can be found here

- *A tour through HTSeq*

  The Tour shows you how to get started. It explains how to install HTSeq, and then demonstrates typical analysis steps with explicit examples. Read this first, and then see the Reference for details.

- *A detailed use case: TSS plots*

  This chapter explains typical usage patterns for HTSeq by explaining in detail three different solutions to the same programming task.

- *Counting reads*

  This chapter explorer in detail the use case of counting the overlap of reads with annotation features and explains how to implement custom logic by writing on's own customized counting scripts

- Reference documentation

  The various classes of *HTSeq* are described here.

  – *Reference overview*

    A brief overview over all classes.

  – *Sequences and FASTA/FASTQ files*

    In order to represent sequences and reads (i.e., sequences with base-call quality information), the classes `Sequence` and `SequenceWithQualities` are used. The classes `FastaReader` and `FastqReader` allow to parse FASTA and FASTQ files.

  – *Genomic intervals and genomic arrays*

    The classes `GenomicInterval` and `GenomicPosition` represent intervals and positions in a genome. The class `GenomicArray` is an all-purpose container with easy access via a genomic interval or position, and `GenomicArrayOfSets` is a special case useful to deal with genomic features (such as genes, exons, etc.)

  – *Read alignments*

    To process the output from short read aligners in various formats (e.g., SAM), the classes described here are used, to represent output files and alignments, i.e., reads with their alignment information.

  – *Features*

    The classes `GenomicFeature` and `GFF_Reader` help to deal with genomic annotation data.

  – *Other parsers*

    This page describes classes to parse VCF, Wiggle and BED files.

  – *Miscellaneous*

- Scripts

  The following scripts can be used without any Python knowledge.

  – *Quality Assessment with htseq-qa*

    Given a FASTQ or SAM file, this script produces a PDF file with plots depicting the base calls and base-call qualities by position in the read. This is useful to assess the technical quality of a sequencing run.

  – *Counting reads in features with htseq-count*

    Given a SAM file with alignments and a GFF file with genomic features, this script counts how many reads map to each feature.

- Appendices
- *Version history*
- *Notes for Contributors*
- *Table of Contents*

- genindex

Author

HTSeq is developed by Simon Anders at EMBL Heidelberg (Genome Biology Unit). Please do not hesitate to contact me (anders *at* embl *dot* de) if you have any comments or questions.

License

CHAPTER 2

Prequisites and installation

HTSeq is available from the Python Package Index (PyPI):

To use HTSeq, you need Python 2.7 or 3.4 or above (3.0-3.3 are not supported), together with:

- NumPy, a commonly used Python package for numerical calculations
- Pysam, a Python interface to samtools.
- To make plots you will need matplotlib, a plotting library.

At the moment, HTSeq supports Linux and OSX but not Windows operating systems, because one of the key dependencies, Pysam, lacks automatic support and none of the HTSeq authors have access to such a machine. However, it *might* work with some work, if you need support for this open an issue on our Github page.

HTSeq follows install conventions of many Python packages. In the best case, it should install from PyPI like this:

```
pip install HTSeq
```

If this does not work, please open an issue on Github and also try the instructions below.

Installation on Linux

You can choose to install HTSeq via your distribution packages or via *pip*. The former is generally recommended but might be updated less often than the *pip* version.

Distribution package manager

- Ubuntu (e.g. for Python 2.7):

```
sudo apt-get install build-essential python2.7-dev python-numpy python-matplotlib␣
↪python-pysam python-htseq
```

- Arch (e.g. using `aura`, you can grab the AUR packages otherwise):

```
sudo pacman -S python python-numpy python-matplotlib
sudo aura -A python-pysam python-htseq
```

PIP

PIP should take care of the requirements for you:

```
pip install HTSeq
```

Installing from GIT

If you want to install a development version, just clone the git repository, switch to the branch/commit you wish, and use `setuptools`:

```
python setup.py build
python setup.py install
```

Typical setuptools options are available (e.g. `--prefix`, `--user`).

To test the installation, change to another director than the build directory, start Python (by typing `python` or `python2.7`) and then try whether typing `import HTSeq` causes an error meesage.

Installation on MacOS X

Mac users should install NumPy as explained here in the NumPy/SciPy documentation. Note that you need to install Xcode to be able to compile NumPy. Due to the mess that Apple recently made out of Xcode, the whole process may be a slight bit more cumbersome than necessary, especially if you work with OSX Lion, so read the instructions carefully.

If you want to produce plots or use htseq-qa, you will also need matplotlib. (For htseq-count, it is not required.) There seems to be a binary package (a "Python egg") available on the matplotlib SourceForge page.

To install HTSeq itself, download the *source* package from the HTSeq PyPI page, unpack the tarball, go into the directory with the unpacked files and type there:

```
python setup.py build
```

to compile HTSeq. If you get an error regarding the availability of a C compiler, you may need to set environment variables to point Python to the . The NumPy/SciPy installation instructions above cover this topic well and apply here, too, so simply do the same as you did to install NumPy.

Once building has been successful, use:

```
python setup.py --user
```

to install HTSeq for the current users. To make HTSeq available to all users, use instead:

```
python setup.py build
sudo python setup.py install
```

To test the installation, change to another director than the build directory, start Python (by typing `python`) and then try whether typing `import HTSeq` causes an error meesage.

MS Windows

If you have not yet installed Python, do so first. You can find an automatic installer for Windows on the Python download page. Make sure to use Python 2.7, not Python 3.3.

Then install the newest version of NumPy. Look on NumPy's PyPI page for the automatic installer.

If you want to produce plots or use htseq-qa, you will also need matplotlib. (For htseq-count, it is not required.) Follow the installation instructions on their web page.

To install HTSeq itself, simply download the Windows installer from the HTSeq download page and run it.

To test your installation, start Python and then try whether typing `import HTSeq` causes an error meesage.

If you get the error message "ImportError: DLL load failed", you are most likely missing the file MSVCR110.DLL on your system, which you can get by downloading and installing the file "vcredist_x86.exe" from this page.

CHAPTER 3

A tour through HTSeq

In the analysis of high-throughput sequencing data, it is often necessary to write custom scripts to form the "glue" between tools or to perform specific analysis tasks. HTSeq is a Python package to facilitate this.

This tour demonstrates the functionality of HTSeq by performing a number of common analysis tasks:

- Getting statistical summaries about the base-call quality scores to study the data quality.
- Calculating a coverage vector and exporting it for visualization in a genome browser.
- Reading in annotation data from a GFF file.
- Assigning aligned reads from an RNA-Seq experiments to exons and genes.

The following description assumes that the reader is familiar with Python and with HTS data. (For a good and not too lengthy introduction to Python, read the *Python Tutorial* on the Python web site.)

If you want to try out the examples on your own system, you can download the example files used from here: HTSeq_example_data.tgz

Reading in reads

In the example data, a FASTQ file is provided with example reads from a yeast RNA-Seq experiment. The file `yeast_RNASeq_excerpt_sequence.txt` is an excerpt of the `_sequence.txt` file produced by the SolexaPipeline software. We can access it from HTSeq with

```
>>> import HTSeq
>>> fastq_file = HTSeq.FastqReader( "yeast_RNASeq_excerpt_sequence.txt", "solexa" )
```

The first argument is the file name. The optional second argument indicates the encoding for the quality string. If you omit, the default ("phred") is used. The example data, however, is from an older experiment, and hence encoded in the offset-64 format that the Solexa/Illumina software pipeline used before version 1.8. (A third option is "solexa_old", for data from the Solexa pipeline prior to version 1.3.)

The variable `fastq_file` is now an object of class *`FastqReader`*, which refers to the file:

```
>>> fastq_file
<FastqReader object, connected to file name 'yeast_RNASeq_excerpt_sequence.txt'>
```

When used in a `for` loop, it generates an iterator of objects representing the reads. Here, we use the `islice` function from `itertools` to cut after 10 reads.

```
>>> import itertools
>>> for read in itertools.islice( fastq_file, 10 ):
...    print(read)
CTTACGTTTTCTGTATCAATACTCGATTTATCATCT
AATTGGTTTCCCCGCCGAGACCGTACACTACCAGCC
TTTGGACTTGATTGTTGACGCTATCAAGGCTGCTGG
ATCTCATATACAATGTCTATCCCAGAAACTCAAAAA
AAAGTTCGAATTAGGCCGTCAACCAGCCAACACCAA
GGAGCAAATTGCCAACAAGGAAAGGCAATATAACGA
AGACAAGCTGCTGCTTCTGTTGTTCCATCTGCTTCC
AAGAGGTTTGAGATCTTTGACCACCGTCTGGGCTGA
GTCATCACTATCAGAGAAGGTAGAACATTGGAAGAT
ACTTTTAAAGATTGGCCAAGAATTGGGGATTGAAGA
```

Of course, there is more to a read than its sequence. The variable `read` still contains the tenth read, and we can examine it:

```
>>> read
<SequenceWithQualities object 'HWI-EAS225:1:10:1284:142#0/1'>
```

A *Sequence* object has two slots, called *seq* and *name*. This object is a *SequenceWithQualities*, and it also has a slot `qual`:

```
>>> read.name
'HWI-EAS225:1:10:1284:142#0/1'
>>> read.seq
b'ACTTTTAAAGATTGGCCAAGAATTGGGGATTGAAGA'
>>> read.qual
array([33, 33, 33, 33, 33, 33, 29, 27, 29, 32, 29, 30, 30, 21, 22, 25, 25,
       25, 23, 28, 24, 24, 29, 29, 29, 25, 28, 24, 24, 26, 25, 25, 24, 24,
       24, 24], dtype=uint8)
```

The values in the quality array are, for each base in the sequence, the Phred score for the correctness of the base.

As a first simple example for the use of HTSeq, we now calculate the average quality score for each position in the reads by adding up the `qual` arrays from all reads and the dividing by the number of reads. We sum everything up in the variable `qualsum`, a `numpy` array of integers:

```
>>> import numpy
>>> len( read )
36
>>> qualsum = numpy.zeros( len(read), numpy.int )
```

Then we loop through the fastq file, adding up the quality scores and counting the reads:

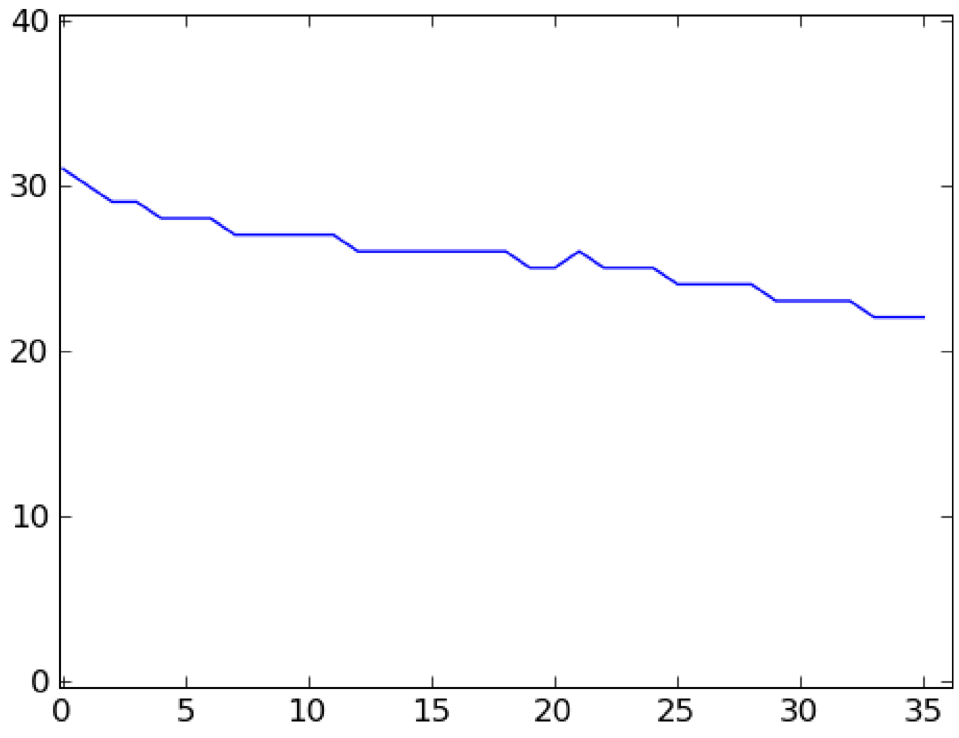
```
>>> nreads = 0
>>> for read in fastq_file:
...    qualsum += read.qual
...    nreads += 1
```

The average qualities are hence:

```
>>> qualsum / float(nreads)
array([ 31.56838274,  30.08288332,  29.4375375 ,  29.00432017,
        28.55290212,  28.26825073,  28.46681867,  27.59082363,
        27.34097364,  27.57330293,  27.11784471,  27.19432777,
        26.84023361,  26.76267051,  26.44885795,  26.79135165,
        26.42901716,  26.49849994,  26.13604544,  25.95823833,
        25.54922197,  26.20460818,  25.42333693,  25.72298892,
        25.04164167,  24.75151006,  24.48561942,  24.27061082,
        24.10720429,  23.68026721,  23.52034081,  23.49437978,
        23.11076443,  22.5576223 ,  22.43549742,  22.62354494])
```

If you have matplotlib installed, you can plot these numbers.

```
>>> from matplotlib import pyplot
>>> pyplot.plot( qualsum / nreads )
>>> pyplot.show()
```



This is a very simple way of looking at the quality scores. For more sophisticated quality-control techniques, see the Chapter *Quality Assessment with htseq-qa*.

Instead of a FASTQ file, you might have a SAM file, with the reads already aligned. The SAM_Reader class can read such data.

```
>>> alignment_file = HTSeq.SAM_Reader( "yeast_RNASeq_excerpt.sam" )
```

If we are only interested in the qualities, we can rewrite the commands from above to use the `alignment_file`:

```
>>> nreads = 0
>>> for aln in alignment_file:
```

```
...     qualsum += aln.read.qual
...     nreads += 1
```

We have simple replaced the *FastqReader* with a *SolexaExportReader*, which iterates, when used in a for loop, over *SolexaExportAlignment* objects. Each of these contain a field *read* that contains the *SequenceWithQualities* object, as before. There are more parses, for example the *SAM_Reader* that can read SAM files, and generates *SAM_Alignment* objects. As all *Alignment* objects contain a *read* slot with the *SequenceWithQualities*, we can use the same code with any alignment file for which a parser has been provided, and all we have to change is the name of the reader class in the first line.

The other fields that all *Alignment* objects contain, is a Boolean called *aligned* that tells us whether the read has been aligned at all, and a field called *iv* (for "interval") that shows where the read was aligned to. We use this information in the next section.

Reading and writing BAM files

HTSeq exposes the samtools API trough pysam, enabling you to read and write BAM files. A simple example of the usage is given here:

```
>>> bam_reader = HTSeq.BAM_Reader( "SRR001432_head_sorted.bam" )
>>> for a in itertools.islice( bam_reader, 5 ):  # printing first 5 reads
...     print(a)
<SAM_Alignment object: Read 'SRR001432.165255 USI-EAS21_0008_3445:8:4:718:439␣
↪length=25' aligned to 1:[29267,29292)/->
<SAM_Alignment object: Read 'SRR001432.238475 USI-EAS21_0008_3445:8:6:888:446␣
↪length=25' aligned to 1:[62943,62968)/->
<SAM_Alignment object: Read 'SRR001432.116075 USI-EAS21_0008_3445:8:3:657:64 length=25
↪' aligned to 1:[86980,87005)/->
<SAM_Alignment object: Read 'SRR001432.159692 USI-EAS21_0008_3445:8:4:618:821␣
↪length=25' aligned to 1:[91360,91385)/->
<SAM_Alignment object: Read 'SRR001432.249247 USI-EAS21_0008_3445:8:6:144:741␣
↪length=25' aligned to 1:[97059,97084)/->
```

```
>>> bam_writer = HTSeq.BAM_Writer.from_BAM_Reader( "region.bam", bam_reader ) #set-up␣
↪BAM_Writer with same header as reader
>>> for a in bam_reader.fetch( region = "1:249000000-249200000" ): #fetching reads in␣
↪a region
...     print("Writing Alignment", a, "to file", bam_writer.filename)
...     bam_writer.write( a )
Writing Alignment <SAM_Alignment object: Read 'SRR001432.104735 USI-EAS21_0008_
↪3445:8:3:934:653 length=25' aligned to 1:[249085369,249085394)/-> to file region.bam
Writing Alignment <SAM_Alignment object: Read 'SRR001432.280764 USI-EAS21_0008_
↪3445:8:7:479:581 length=25' aligned to 1:[249105864,249105889)/-> to file region.bam
...
Writing Alignment <SAM_Alignment object: Read 'SRR001432.248967 USI-EAS21_0008_
↪3445:8:6:862:756 length=25' aligned to 1:[249167916,249167941)/-> to file region.bam
>>> bam_writer.close()
```

Genomic intervals and genomic arrays

Genomic intervals

At the end of the previous section, we looped through a SAM file. In the for loop, the `SAM_Reader` object yields for each alignment line in the SAM file an object of class `SAM_Alignment`. Let's have closer look at such an object, still found in the variable `aln`:

```
>>> aln
<SAM_Alignment object: Read 'HWI-EAS225:1:11:76:63#0/1' aligned to IV:[246048,246084)/
↪+>
```

Every alignment object has a slot `read`, that contains a `SequenceWithQualities` object as described above

```
>>> aln.read
<SequenceWithQualities object 'HWI-EAS225:1:11:76:63#0/1'>
>>> aln.read.name
'HWI-EAS225:1:11:76:63#0/1'
>>> aln.read.seq
b'ACTGTAAATACTTTTCAGAAGAGATTTGTAGAATCC'
>>> aln.read.qualstr
b'BBBB@B?AB?>BAAA@A@>=?=?9=?=;9>988<::'
>>> aln.read.qual
array([33, 33, 33, 33, 31, 33, 30, 32, 33, 30, 29, 33, 32, 32, 32, 31, 32,
       31, 29, 28, 30, 28, 30, 24, 28, 30, 28, 26, 24, 29, 24, 23, 23, 27,
       25, 25], dtype=uint8)
```

Furthermore, every alignment object has a slot `iv` (for "interval") that describes where the read was aligned to (if it was aligned). To hold this information, an object of class `GenomicInterval` is used that has slots as follows:

```
>>> aln.iv
<GenomicInterval object 'IV', [246048,246084), strand '+'>
>>> aln.iv.chrom
'IV'
>>> aln.iv.start
246048
>>> aln.iv.end
246084
>>> aln.iv.strand
'+'
```

Note that all coordinates in HTSeq are zero-based (following Python convention), i.e. the first base of a chromosome has index 0. Also, all intervals are half-open, i.e., the `end` position is not included. The strand can be one of `'+'`, `'-'`, and `'.'`, where the latter indicates that the strand is not defined or not of interest.

Apart from these slots, a `GenomicInterval` object has a number of convenience functions, see the reference.

Note that a SAM file may contain reads that could not be aligned. For these, the *iv* slot contains *None*. To test whether an alignment is present, you can also query the slot *aligned*, which is a Boolean.

Genomic Arrays

The `GenomicArray` data structure is a convenient way to store and retrieve information associated with a genomic position or genomic interval. In a GenomicArray, data (either simple scalar data like a number) or can be stored at a place identified by a GenomicInterval. We demonstrate with a toy example.

Assume you have a genome with three chromosomes with the following lengths (in bp):

```
>>> chromlens = { 'chr1': 3000, 'chr2': 2000, 'chr1': 1000 }
```

We wish to store integer data (typecode "i")

```
>>> ga = HTSeq.GenomicArray( chromlens, stranded=False, typecode="i" )
```

Now, we can assign the value 5 to an interval:

```
>>> iv = HTSeq.GenomicInterval( "chr1", 100, 120, "." )
>>> ga[iv] = 5
```

We may want to add the value 3 to an interval overlapping with the previous one:

```
>>> iv = HTSeq.GenomicInterval( "chr1", 110, 135, "." )
>>> ga[iv] += 3
```

To see the effect of this, we read out an interval encompassing the region that we changed. To display the data, we convert to a list:

```
>>> iv = HTSeq.GenomicInterval( "chr1", 90, 140, "." )
>>> list( ga[iv] )
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 8, 8,
 8, 8, 8, 8, 8, 8, 8, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0,
 0, 0, 0, 0]
```

It would be wasteful to store all these repeats of the same value as it is displayed here. Hence, GenomicArray objects use by default so-called StepVectors that store the data internally in "steps" of constant value. Often, reading out the data that way is useful, too:

```
>>> for iv2, value in ga[iv].steps():
...    print(iv2, value)
...
chr1:[90,100)/. 0
chr1:[100,110)/. 5
chr1:[110,120)/. 8
chr1:[120,135)/. 3
chr1:[135,140)/. 0
```

If the steps become very small, storing them instead of just the unrolled data may become inefficient. In this case, GenomicArrays should be instantiated with storage mode `ndarray` to get a normal numpy array as backend, or with storage mode `memmap` to use a file/memory-mapped numpy array (see reference for details).

In the following section, we demonstrate how a GenomicArray can be used to calculate a coverage vector. In the section after that, we see how a GenomicArray with type code 'O' (which stands for 'object', i.e., any kind of data, not just numbers) is useful to organize metadata.

Calculating coverage vectors

By a "coverage vector", we mean a vector (one-dimensional array) of the length of a chromosome, where each element counts how many reads cover the corresponding base pair in their alignment. A GenomicArray can conveniently bundle the coverage vectors for all the chromosomes in a genome.

Hence, we start by defining a *GenomicArray*:

```
>>> cvg = HTSeq.GenomicArray( "auto", stranded=True, typecode="i" )
```

Instead of listing all chromosomes, we instruct the GenomicArray to add chromosome vectors as needed, by specifiyng `"auto"`. As we set `stranded=True`, there are now two chromosome vectors for each chromosome, all holding integer values (`typecode="i"`). They all have an "infinte" length as we did not specify the actual lengths of the chromosomes.

To build the coverage vectors, we now simply iterate through all the reads and add the value 1 at the interval to which each read was aligned to:

```
>>> alignment_file = HTSeq.SAM_Reader( "yeast_RNASeq_excerpt.sam" )
>>> cvg = HTSeq.GenomicArray( "auto", stranded=True, typecode='i' )
>>> for alngt in alignment_file:
...    if alngt.aligned:
...       cvg[ alngt.iv ] += 1
```

We can plot an excerpt of this with:

```
>>> pyplot.plot( list( cvg[ HTSeq.GenomicInterval( "III", 200000, 500000, "+" ) ] ) )
```

However, a proper genome browser gives a better impression of the data. The following commands write two Bed-Graph (Wiggle) files, one for the plus and one for the minus strands:

```
>>> cvg.write_bedgraph_file( "plus.wig", "+" )
>>> cvg.write_bedgraph_file( "minus.wig", "-" )
```

These two files can then be viewed in a genome browser (e.g. IGB or IGV), alongside the annotation from a GFF file (see below).

GenomicArrayOfSets

Another use of genomic arrays is to store annotation data. In the next section, we will use this to store the position of all exons of the yeast genome in a genomic array and then go through all our reads, querying the array for each read to report the exons overlapped by this read.

In principle, we could use a genomic array with type code 'O' (for object), which can store arbitrary Python objects. However, there might be positions in the genome that are covered by more than one gene, and hence, we better use a data structure that can accommodate overlapping features.

The class:*GenomicArrayOfSets* is meant for this purpose. For each step, it stores a `set` of objects. To illustrate this, we initialize a GenomicArrayOfSets and then store three features in it:

```
>>> gas = HTSeq.GenomicArrayOfSets( "auto", stranded=False )
>>> gas[ HTSeq.GenomicInterval( "chr1", 100, 250 ) ] += "A"
>>> gas[ HTSeq.GenomicInterval( "chr1", 360, 640 ) ] += "A"
>>> gas[ HTSeq.GenomicInterval( "chr1", 510, 950 ) ] += "B"
```

These three features represent three exons of two genes, arranged as shown in this figure:

Note that we used +=, not just =, above when adding the features. With a GenomicArrayOfSets, you need to always use the += operator (rather than =), so that the values gets *added* to the step's set.

Now consider a read that aligns to the following interval (represented in the figure above by the light blue line):

```
>>> read_iv = HTSeq.GenomicInterval( "chr1", 450, 800 )
```

We can query the GenomicArrayOfSets, as before:

```
>>> for iv, val in gas[ read_iv ].steps():
...     print(iv, sorted(val))
chr1:[450,510)/. ['A']
chr1:[510,640)/. ['A', 'B']
chr1:[640,800)/. ['B']
```

The interval has been subdivided into three pieces, corresponding to the three different sets that it overlaps, namely first only A, then A and B, and finally only B.

You might be only interested in the set of all features that the read interval overlaps. To this end, just form the set union of the three reported sets, using Python's set union operator (`|`):

```
>>> fset = set()
>>> for iv, val in gas[ read_iv ].steps():
...     fset |= val
>>> print(sorted(fset))
['A', 'B']
```

Experienced Python developers will recognize that the `for` loop can be replaced with a single line using a generator comprehension and the `reduce` function:

```
>>> sorted(set.union(*[val for iv, val in gas[ read_iv ].steps()]))
['A', 'B']
```

We will come back to the constructs in the next section, after a brief detour on how to read GTF files.

Counting reads by genes

As the example data is from an RNA-Seq experiment, we want to know how many reads fall into the exonic regions of each gene. For this purpose we first need to read in information about the positions of the exons. A convenient source of such information are the GTF files from Ensembl (to be found here).

These file are in the GTF format, a tightening of the GFF format. HTSeq offers the *GFF_Reader* class to read in a GFF file:

```
>>> gtf_file = HTSeq.GFF_Reader( "Saccharomyces_cerevisiae.SGD1.01.56.gtf.gz",
...     end_included=True )
```

The GFF format is, unfortunately, a not very well specified file format. Several standard documents exist, from different groups, which contradict each other in some points. Most importantly, it is unclear whether a range specified in a GFF line is supposed to include the base under the "end" position or not. Here, we specied the this file does include the end. Actually, this is the default for GFF_Reader, so it would not have been necessary to specify it. (Hint, if you are unsure about your GFF file: The length of most coding exons is divisible by 3. If start-end is divisible by 3, too, end is not included, if the division leaves a remainder of two, end is included.)

We iterate through this file as follows:

```
>>> for feature in itertools.islice( gtf_file, 10 ):
...     print(feature)
...
<GenomicFeature: exon 'R0010W' at 2-micron: 251 -> 1523 (strand '+')>
<GenomicFeature: CDS 'R0010W' at 2-micron: 251 -> 1520 (strand '+')>
```

```
<GenomicFeature: start_codon 'R0010W' at 2-micron: 251 -> 254 (strand '+')>
<GenomicFeature: stop_codon 'R0010W' at 2-micron: 1520 -> 1523 (strand '+')>
<GenomicFeature: exon 'R0020C' at 2-micron: 3007 -> 1885 (strand '-')>
<GenomicFeature: CDS 'R0020C' at 2-micron: 3007 -> 1888 (strand '-')>
<GenomicFeature: start_codon 'R0020C' at 2-micron: 3007 -> 3004 (strand '-')>
<GenomicFeature: stop_codon 'R0020C' at 2-micron: 1888 -> 1885 (strand '-')>
<GenomicFeature: exon 'R0030W' at 2-micron: 3270 -> 3816 (strand '+')>
<GenomicFeature: CDS 'R0030W' at 2-micron: 3270 -> 3813 (strand '+')>
```

The `feature` variable is filled with objects of class *GenomicFeature*. If you compare the coordinated with the original file, you will notice that the GFF_Reader has subtracted one from all starts. This is because all file parsers in HTSeq adjust coordinates as necessary to fit the Python convention, which is that indexing starts with zero and the end is not included. Hence, you can immediately compare coordinates from different data formats without having to worry about subtleties like the fact that GFF is one-based and SAM is zero-based.

As with all Python objects, the `dir` function shows us the slots and functions of our loop variable `feature` and so allow us to inspect what data it provides:

```
>>> dir( feature )
['__class__', ..., '__weakref__', 'attr', 'frame', 'get_gff_line',
'iv', 'name', 'score', 'source', 'type']
```

Ignoring the attributes starting with an underscore, we can see now how to access the information stored in the GFF file. The information from the columns of the GFF table is accessible as follows:

```
>>> feature.iv
<GenomicInterval object '2-micron', [3270,3813), strand '+'>
>>> feature.source
'protein_coding'
>>> feature.type
'CDS'
>>> feature.score
'.'
```

The last column (the attributes) is parsed and presented as a dict:

```
>>> sorted(feature.attr.items())
[('exon_number', '1'),
 ('gene_id', 'R0030W'),
 ('gene_name', 'RAF1'),
 ('protein_id', 'R0030W'),
 ('transcript_id', 'R0030W'),
 ('transcript_name', 'RAF1')]
```

The very first attribute in this column is usually some kind of ID, hence it is stored in the slot *name*:

```
>>> feature.name
'R0030W'
```

To deal with this data, we will use the *GenomicArrayOfSets* introuced in the previous section.

```
>>> exons = HTSeq.GenomicArrayOfSets( "auto", stranded=False )
```

However, our RNA-Seq experiment was not strand-specific, i.e., we do not know whether the reads came from the plus or the minus strand. This is why we defined the GenomicArrayOfSet as non-stranded (`stranded=False` in the instantiation of `exons` above), instructing it to ignore all strand information. Teherfore, we now have many overlapping genes, but the GenomicArrayOfSets will handle this.

```
>>> for feature in gtf_file:
...    if feature.type == "exon":
...       exons[ feature.iv ] += feature.name
```

Nate that, we only store the gene name this time, as this will be more convenient later.

Assume we have a read covering this interval:

```
>>> iv = HTSeq.GenomicInterval( "III", 23850, 23950, "." )
```

Its left half covers two genes (YCL058C, YCL058W-A), but its right half only YCL058C because YCL058W-A end in the middle of the read:

```
>>> [(st[0], sorted(st[1])) for st in exons[iv].steps()]
[(<GenomicInterval object 'III', [23850,23925), strand '.'>,
     ['YCL058C', 'YCL058W-A']),
 (<GenomicInterval object 'III', [23925,23950), strand '.'>,
     ['YCL058C'])]
```

Assuming the transcription boundaries in our GTF file to be correct, we may conclude that this read is from the gene that appears in both steps and not from the one that appears in only one of the steps. More generally, whenever a read overlaps multiple steps (a new step starts wherever a feature starts or ends), we get a set of feature names for each step, and we have to find the intersection of all these. This can be coded as follows:

```
>>> iset = None
>>> for iv2, step_set in exons[iv].steps():
...    if iset is None:
...       iset = step_set.copy()
...    else:
...       iset.intersection_update( step_set )
...
>>> print(iset)
{'YCL058C'}
```

When we look at the first step, we make a copy of the steps (in order to not disturb the values stored in `exons`.) For the following steps, we use the `intersection_update` method Python's standard `set` class, which performs a set intersection in place. Afterwards, we have a set with precisely one element. Getting this one element is a tiny bit cumbersome; to access it, one needs to write:

```
>>> list(iset)[0]
'YCL058C'
```

In this way, we can go through all our aligned reads, calculate the intersection set, and, if it contains a single gene name, add a count for this gene. For the counters, we use a dict, which we initialize with a zero for each gene name:

```
>>> counts = {}
>>> for feature in gtf_file:
...    if feature.type == "exon":
...       counts[ feature.name ] = 0
```

Now, we can finally count:

```
>>> sam_file = HTSeq.SAM_Reader( "yeast_RNASeq_excerpt.sam" )
>>> for alnmt in sam_file:
...    if alnmt.aligned:
...       iset = None
...       for iv2, step_set in exons[ alnmt.iv ].steps():
```

```
...             if iset is None:
...                iset = step_set.copy()
...             else:
...                iset.intersection_update( step_set )
...         if len( iset ) == 1:
...            counts[ list(iset)[0] ] += 1
```

We can now conveniently print the result with:

```
>>> for name in sorted( counts.keys() ):
...    print(name, counts[name])
15S_rRNA 0
21S_rRNA 0
HRA1 0
...
YPR048W 2
YPR049C 3
YPR050C 0
YPR051W 1
YPR052C 1
YPR053C 5
YPR054W 0
...
tY(GUA)M2 0
tY(GUA)O 0
tY(GUA)Q 0
```

Some aligners can output gapped or spliced alignments. In a SAM file, this in encoded in the CIGAR string. HTSeq has facilities to handle this conveniently, too, with the class `CigarOperation`. Chapter *Counting reads in features with htseq-count* describes a script which offers some further counting schemes.

And much more

This tour is only meant to give an overview. There are many more tasks that can be solved with HTSeq. Have a look at the reference documentation in the following pages to see what else is there.

CHAPTER 4

A detailed use case: TSS plots

A common task in ChIP-Seq analysis is to get profiles of marks with respect to nearby features. For example, when analysing histone marks, one is often interested in the position and extend of such marks in the vicinity of transcription start sites (TSSs). To this end, one commonly calculates the coverage of reads or fragments across the whole genome, then marks out fixed-size windows centered around the TSSs of all genes, takes the coverages in these windows and adds them up to a "profile" that has the size of the window. This is a simple operation, which, however, can become challenging, when working with large genomes and very many reads.

Here, we demonstrate various ways of how data flow can be organized in HTSeq by means of different solutions to this task.

As example data, we use a short excerpt from the data set by Barski et al. (Cell, 2007, 129:823). We downloaded the FASTQ file for one of the H3K4me3 samples (Short Read Archive accession number SRR001432), aligned it against the Homo sapiens genome build GRCh37 with BWA, and provide the start of this file (actually only containing reads aligned to chromosome 1) as file `SRR001432_head.bam` with the HTSeq example files. As annotation, we use the file `Homo_sapiens.GRCh37.56_chrom1.gtf`, which is the part of the Ensembl GTF file for Homo sapiens for chromosome 1.

Using the full coverage

We start with the straight-forward way of calculating the full coverage first and then summing up the profile. This can be done as described in the *Tour*:

```
>>> import HTSeq
>>> bamfile = HTSeq.BAM_Reader( "SRR001432_head.bam" )
>>> gtffile = HTSeq.GFF_Reader( "Homo_sapiens.GRCh37.56_chrom1.gtf" )
>>> coverage = HTSeq.GenomicArray( "auto", stranded=False, typecode="i" )
>>> for almnt in bamfile:
...    if almnt.aligned:
...       coverage[ almnt.iv ] += 1
```

To find the location of all transcription start sites, we can look in the GTF file for exons with exon number 1 (as indicated by the `exon_number` attribute in Ensembl GTF files) and ask for their directional start (`start_d`). The following loop extracts and prints this information (using `itertools.islice` to go through only the first 100 features in the GTF file):

```
>>> import itertools
>>> for feature in itertools.islice( gtffile, 100):
...    if feature.type == "exon" and feature.attr["exon_number"] == "1":
...       print(feature.attr["gene_id"], feature.attr["transcript_id"], feature.iv.
↪start_d_as_pos)
ENSG00000223972 ENST00000456328 1:11873/+
ENSG00000223972 ENST00000450305 1:12009/+
ENSG00000227232 ENST00000423562 1:29369/-
ENSG00000227232 ENST00000438504 1:29369/-
ENSG00000227232 ENST00000488147 1:29569/-
ENSG00000227232 ENST00000430492 1:29342/-
ENSG00000243485 ENST00000473358 1:29553/+
ENSG00000243485 ENST00000469289 1:30266/+
ENSG00000221311 ENST00000408384 1:30365/+
ENSG00000237613 ENST00000417324 1:36080/-
ENSG00000237613 ENST00000461467 1:36072/-
ENSG00000233004 ENST00000421949 1:53048/+
ENSG00000240361 ENST00000492842 1:62947/+
ENSG00000177693 ENST00000326183 1:69054/+
```

As the GTF file contains several transcripts for each gene, one TSS may appear multiple times, giving undue weight to it. Hence, we collect them in a `set` as this data type enforces uniqueness.

```
>>> tsspos = set()
>>> for feature in gtffile:
...    if feature.type == "exon" and feature.attr["exon_number"] == "1":
...       tsspos.add( feature.iv.start_d_as_pos )
```

Let's take one of these starting positions. To get a nice one, we manually chose this one here, just for demonstration purposes:

```
>>> p = HTSeq.GenomicPosition( "1", 145439814, "+" )
```

This is really one of the TSSs in the set:

```
>>> p in tsspos
True
```

We can get a window centered on this TSS by just subtracting and adding a fixed value (half of the desired window size, let's use 3 kb):

```
>>> halfwinwidth = 3000
>>> window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
>>> window
<GenomicInterval object '1', [145436814,145442814), strand '.'>
```

We can check the coverage in this window by subsetting and transforming to a list:

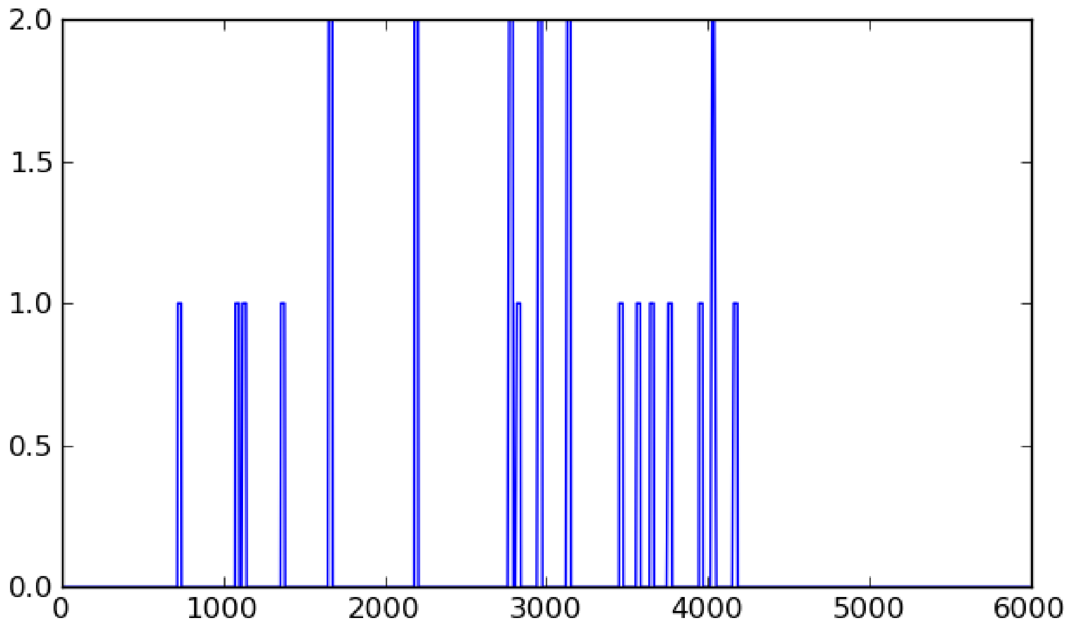
```
>>> list( coverage[window] )
[0, 0, 0, ..., 0, 0]
```

As we will work with numpy from now on, it may be better to get this as numpy array:

```
>>> import numpy
>>> wincvg = numpy.fromiter( coverage[window], dtype='i', count=2*halfwinwidth )
>>> wincvg
array([0, 0, 0, ..., 0, 0, 0], dtype=int32)
```

With matplotlib, we can see that this vector is, in effect, not all zero:

```
>>> from matplotlib import pyplot
>>> pyplot.plot( wincvg )
>>> pyplot.show()
```



To sum up the profile, we initialize a numpy vector of the size of our window with zeroes:

```
>>> profile = numpy.zeros( 2*halfwinwidth, dtype='i' )
```

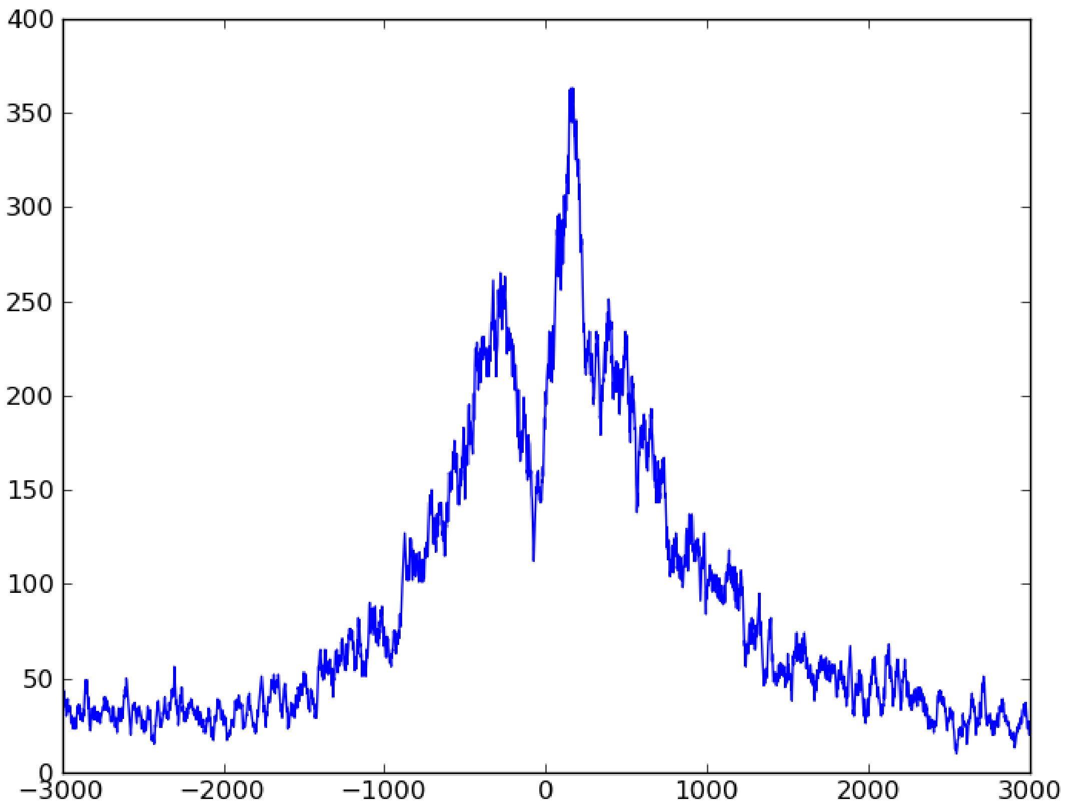
Now, we can go through the TSS positions and add the coverage in their windows to get the profile:

```
>>> for p in tsspos:
...    window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
...    wincvg = numpy.fromiter( coverage[window], dtype='i', count=2*halfwinwidth )
...    if p.strand == "+":
...       profile += wincvg
...    else:
...       profile += wincvg[::-1]
```

Note that we add the window coverage reversed ("`[::-1]`") if the gene was on the minus strand.

Using matplotlib, we can plot this:

```
>>> pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profile )
>>> pyplot.show()
```



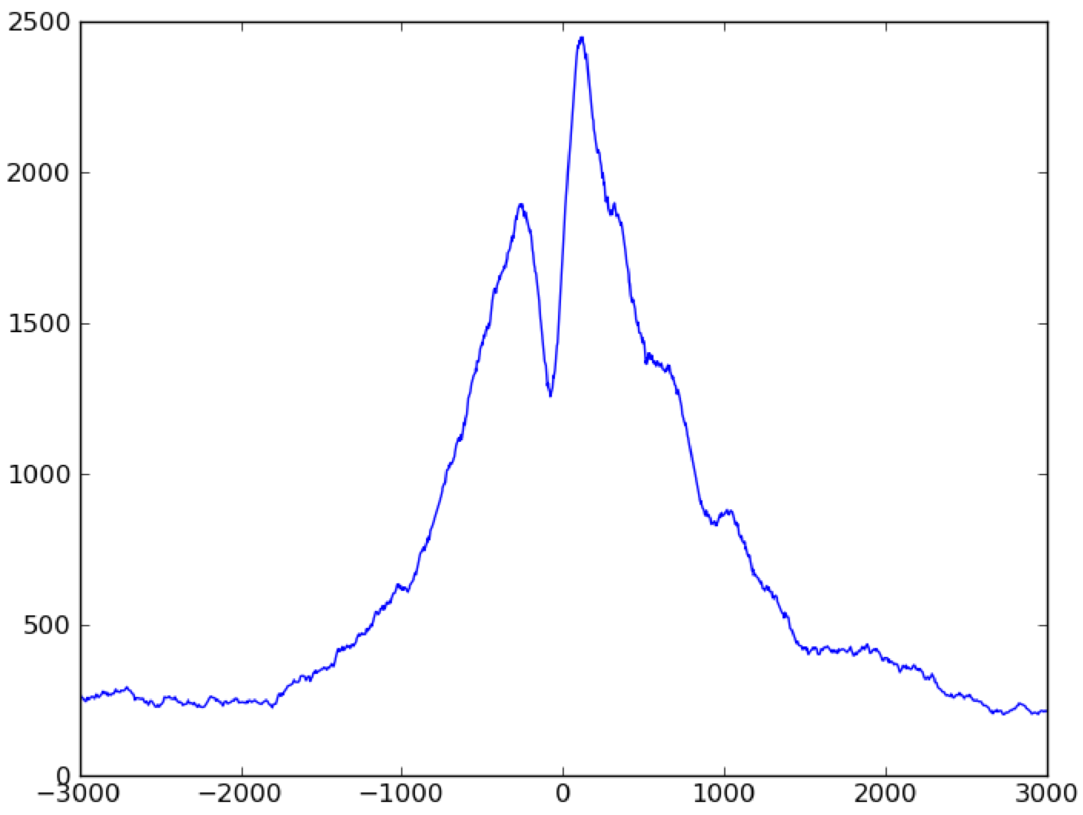
We can see clearly that the reads concentrate around the TSS, with a prominent peak a bit downstream (if you use matplotlib's interactive zoom, you can easily see that the peak is at 153 bp) and a dip upstream, at -79 bp.

Going back to the beginning, there is one possible improvement: When calculating the coverage, we just added one to all the base pairs that the read covered. However, the fragment extends beyond the read, to a length of about 200 bp (the fragment size for which Barski et al. selected). Maybe we get a better picture by calculating the coverage not from the reads but from the *fragments*, i.e., the reads extended to fragment size. For this, we just one line, to extend the read to 200 bp. Using this, we now put the whole script together:

```
import HTSeq
import numpy
from matplotlib import pyplot

bamfile = HTSeq.BAM_Reader( "SRR001432_head.bam" )
gtffile = HTSeq.GFF_Reader( "Homo_sapiens.GRCh37.56_chrom1.gtf" )
halfwinwidth = 3000
fragmentsize = 200

coverage = HTSeq.GenomicArray( "auto", stranded=False, typecode="i" )
for almnt in bamfile:
   if almnt.aligned:
      almnt.iv.length = fragmentsize
      coverage[ almnt.iv ] += 1

tsspos = set()
for feature in gtffile:
   if feature.type == "exon" and feature.attr["exon_number"] == "1":
      tsspos.add( feature.iv.start_d_as_pos )

profile = numpy.zeros( 2*halfwinwidth, dtype='i' )
for p in tsspos:
   window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
   wincvg = numpy.fromiter( coverage[window], dtype='i', count=2*halfwinwidth )
   if p.strand == "+":
      profile += wincvg
   else:
      profile += wincvg[::-1]
```

The script produces a `profile` variable whhich we can plot by adding these lines to it:

```
pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profile )
pyplot.show()
```



The plot looks much smoother with the extended fragments.

The coverage vector can be held in memory, even for a very large genome, because large parts of it are zero and even where there are reads, the values tend to stay constant for stretches of several bases. Hence, GenomicArray's step storage mode is useful. If, however, extremely large amounts of reads are processed, the coverage vector can become "rough" and change value at every position. Then, the step storage mode becomes inefficient and we might be better off with an ordinary dense vector such as provided by numpy. As this numpy vector becomes very large, it may not fit in memory, and the 'memmap' storage (using numpy's memmap facility) then uses temporary files on disk. We mention these possibilities as they may be useful when working with the full coverage vector is required. Here, however, we can do otherwise.

Using indexed BAM files

We do not need the coverage everythere. We only need it close to the TSSs. We can sort our BAM file by position (`samtools sort`) and index it (`samtools index`) and then use random access, as HTSeq exposes this functionality of SAMtools.

Let's say we use the same window as above as example:

```
>>> p = HTSeq.GenomicPosition( "1", 145439814, "+" )
>>> window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
>>> window
<GenomicInterval object '1', [145436814,145442814), strand '.'>
```

Then, we can simply get a list of all reads within this interval as follows:

```
>>> sortedbamfile = HTSeq.BAM_Reader( "SRR001432_head_sorted.bam" )
>>> for almnt in sortedbamfile[ window ]:
...     print(almnt)
<SAM_Alignment object: Read 'SRR001432.90270 USI-EAS21_0008_3445:8:3:245:279 length=25
↪' aligned to 1:[145437532,145437557)/->
 ...
<SAM_Alignment object: Read 'SRR001432.205754 USI-EAS21_0008_3445:8:5:217:355␣
↪length=25' aligned to 1:[145440975,145441000)/->
```

Let's have a closer look at the last alignment. As before, we first extent the read to fragment size:

```
>>> fragmentsize = 200
>>> almnt.iv.length = fragmentsize
>>> almnt
<SAM_Alignment object: Read 'SRR001432.205754 USI-EAS21_0008_3445:8:5:217:355␣
↪length=25' aligned to 1:[145440800,145441000)/->
```

The read has been aligned to the "-" strand, and hence, we should look at its distance to the *end* of the window (i.e., `p.pos`, the position of the TSS, plus half the window width) to see where it should be added to the `profile` vector:

```
>>> start_in_window = p.pos + halfwinwidth - almnt.iv.end
>>> end_in_window   = p.pos + halfwinwidth - almnt.iv.start
>>> print(start_in_window, end_in_window)
1814 2014
```

To account for this read, we should add ones in the profile vector along the indicated interval.

Using this, we can go through the set of all TSS positions (in the `tsspos` set variable that we created above) and for each TSS position, loop through all aligned reads close to it. Here is this double loop:

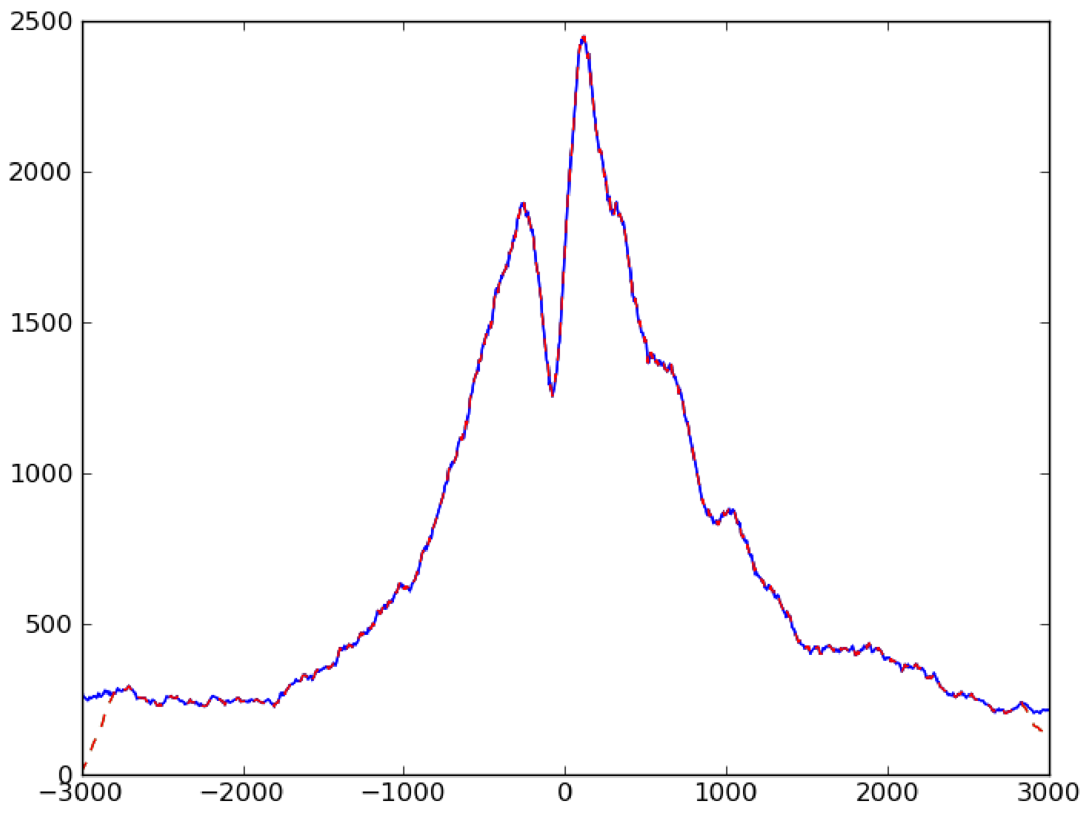
```
>>> profileB = numpy.zeros( 2*halfwinwidth, dtype='i' )
>>> for p in tsspos:
...     window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
...     for almnt in sortedbamfile[ window ]:
```

```
...         almnt.iv.length = fragmentsize
...         if p.strand == "+":
...             start_in_window = almnt.iv.start - p.pos + halfwinwidth
...             end_in_window   = almnt.iv.end   - p.pos + halfwinwidth
...         else:
...             start_in_window = p.pos + halfwinwidth - almnt.iv.end
...             end_in_window   = p.pos + halfwinwidth - almnt.iv.start
...         profileB[ start_in_window : end_in_window ] += 1
```

This loop now runs a good deal faster than our first attempt, and has a much smaller memory footprint.

We can plot the profiles obtained from our two methods on top of each other:

```
>>> pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profile, ls="-", color=
↪"blue" )
>>> pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profileB, ls="--",␣
↪color="red" )
>>> pyplot.show()
```



We notice that they are equal, except for the boundaries. This artifact arose because we extend reads to fragment length: A read which is just outside the `window` will not be found by our new loop even though if may reach into our profile window after extension to fragment length. Therefore, we should make the window used to subset the BAM file a bit wider than before to get even reads that are once the fragment length away. However, with this, we may also get reads that get extended into the wrong direction, such that `start_in_windows` and `end_in_windows` extend beyond the size of the fragment vector. Four extra lines need to be added to deal with these cases, and then, our new script gives the same result as the previous one.

Here is the complete code:

```
import HTSeq
import numpy
from matplotlib import pyplot

sortedbamfile = HTSeq.BAM_Reader( "SRR001432_head_sorted.bam" )
gtffile = HTSeq.GFF_Reader( "Homo_sapiens.GRCh37.56_chrom1.gtf" )
halfwinwidth = 3000
fragmentsize = 200

tsspos = set()
for feature in gtffile:
   if feature.type == "exon" and feature.attr["exon_number"] == "1":
      tsspos.add( feature.iv.start_d_as_pos )

profile = numpy.zeros( 2*halfwinwidth, dtype='i' )
for p in tsspos:
   window = HTSeq.GenomicInterval( p.chrom,
       p.pos - halfwinwidth - fragmentsize, p.pos + halfwinwidth + fragmentsize, "." )
   for almnt in sortedbamfile[ window ]:
      almnt.iv.length = fragmentsize
      if p.strand == "+":
         start_in_window = almnt.iv.start - p.pos + halfwinwidth
         end_in_window   = almnt.iv.end   - p.pos + halfwinwidth
      else:
         start_in_window = p.pos + halfwinwidth - almnt.iv.end
         end_in_window   = p.pos + halfwinwidth - almnt.iv.start
      start_in_window = max( start_in_window, 0 )
      end_in_window = min( end_in_window, 2*halfwinwidth )
      if start_in_window >= 2*halfwinwidth or end_in_window < 0:
         continue
      profile[ start_in_window : end_in_window ] += 1
```

As before, to get a plot, add:

```
pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profile )
pyplot.show()
```

You will now get the same plot as we got with the first method.

Streaming through all reads

The previous solution requires sorting and indexing the BAM file. For large amounts of data, this may be a burden, and hence, we show a third solution that does not require random access to reads. The idea is to go through all reads in arbitrary order, check for each read whether it falls into one or more windows around TSSs, and, if so, adds ones to the profile vector at the appriate places. In essence, it is the same tactic as before, but nesting the two *for* loops the other way round.

In order to be able to check quickly whether a read overlaps with a window, we can use a `GenomicArrayOfSets`, in which we mark off all windows. For easy access, we denote each winow with an `GenomicPosition` object giving its midpoint, i.e., the actual TSS position, as follows:

```
>>> tssarray = HTSeq.GenomicArrayOfSets( "auto", stranded=False )
>>> for feature in gtffile:
...    if feature.type == "exon" and feature.attr["exon_number"] == "1":
...       p = feature.iv.start_d_as_pos
...       window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
...       tssarray[ window ] += p

>>> len( list( tssarray.chrom_vectors["1"]["."].steps() ) )
30085
```

As before, `p` is the position of the TSS, and `window` is the interval around it.

To demonstrate how this data structure can be used, we take a specific read that we selected as a good example:

```
>>> for almnt in bamfile:
...     if almnt.read.name.startswith( "SRR001432.700 " ):
...         break
>>> almnt
<SAM_Alignment object: Read 'SRR001432.700 USI-EAS21_0008_3445:8:1:35:294 length=25'␣
↪aligned to 1:[169677855,169677880)/->
```

Again, we extent the read to fragment size:

```
>>> almnt.iv.length = fragmentsize
>>> almnt
<SAM_Alignment object: Read 'SRR001432.700 USI-EAS21_0008_3445:8:1:35:294 length=25'␣
↪aligned to 1:[169677680,169677880)/->
```

To see which windows the read covers, we subset the `tssarray` and ask for steps that the fragment in `almnt` covers:

```
>>> for step_iv, step_set in tssarray[ almnt.iv ].steps():
...    print("Step", step_iv, ", contained by these windows:")
...    out = set()
...    for p in step_set:
...        out.add("   Window around TSS at "+str(p))
...    print('\n'.join(sorted(out)))
Step 1:[169677680,169677838)/. , contained by these windows:
   Window around TSS at 1:169677780/-
   Window around TSS at 1:169679672/-
Step 1:[169677838,169677880)/. , contained by these windows:
   Window around TSS at 1:169677780/-
   Window around TSS at 1:169679672/-
   Window around TSS at 1:169680838/-
```

As is typical for GenomicArrayOfSets, some TSSs appear in more than one step. To make sure that we don't count them twice, we take the union of all the step sets (with the operator `|=`, which means in-place union when used for Python sets):

```
>>> s = set()
>>> for step_iv, step_set in tssarray[ almnt.iv ].steps():
...    s |= {x.__repr__() for x in step_set}
>>> sorted(s)   #
["<GenomicPosition object '1':169677780, strand '-'>",
 "<GenomicPosition object '1':169679672, strand '-'>",
 "<GenomicPosition object '1':169680838, strand '-'>"]
```

For each of the values for `p` in `s`, we calculate values for `start_in_window` and `stop_in_window`, as before, and then add ones in the `profile` vector at the appropriate places.

Putting all this together leads to this script:

```
import HTSeq
import numpy
from matplotlib import pyplot

bamfile = HTSeq.BAM_Reader( "SRR001432_head.bam" )
gtffile = HTSeq.GFF_Reader( "Homo_sapiens.GRCh37.56_chrom1.gtf" )
halfwinwidth = 3000
fragmentsize = 200

tsspos = HTSeq.GenomicArrayOfSets( "auto", stranded=False )
for feature in gtffile:
   if feature.type == "exon" and feature.attr["exon_number"] == "1":
      p = feature.iv.start_d_as_pos
      window = HTSeq.GenomicInterval( p.chrom, p.pos - halfwinwidth, p.pos +␣
↪halfwinwidth, "." )
      tsspos[ window ] += p

profile = numpy.zeros( 2*halfwinwidth, dtype="i" )
for almnt in bamfile:
   if almnt.aligned:
      almnt.iv.length = fragmentsize
      s = set()
      for step_iv, step_set in tsspos[ almnt.iv ].steps():
         s |= step_set
      for p in s:
         if p.strand == "+":
            start_in_window = almnt.iv.start - p.pos + halfwinwidth
            end_in_window   = almnt.iv.end   - p.pos + halfwinwidth
         else:
            start_in_window = p.pos + halfwinwidth - almnt.iv.end
            end_in_window   = p.pos + halfwinwidth - almnt.iv.start
         start_in_window = max( start_in_window, 0 )
         end_in_window = min( end_in_window, 2*halfwinwidth )
         profile[ start_in_window : end_in_window ] += 1
```

Again, to get a plot (which will look the same as before), add:

```
pyplot.plot( numpy.arange( -halfwinwidth, halfwinwidth ), profile )
pyplot.show()
```

CHAPTER 5

Counting reads

A very typical use case for the HTSeq library is to for a given list of genomic features (such as genes, exons, ChIP-Seq peaks, or the like), how many sequencing reads overlap each of the features. As a more complex example for using HTSeq, we supply the script `htseq-count`, which takes a GTF file with gene models and a SAM file and counts for each gene how many reads map to it; see Section *Counting reads in features with htseq-count*.

The `htseq-count` script, however, has implementation details which were chosen with a specific use case in mind, namely to quantify gene expression for subsequent testing for differential expression, which is why, for example, the script does not count reads that map to multiple genes. For other applications, different resolutions of such ambiguities might be desirable, and then, a bioinformatician may want to create her own counting script. In the following, we expand on the coverage of this topic in the *Tour* (*A tour through HTSeq*) and give building blocks which should make it possible to write such scripts also for bioinformaticians with only modest knowledge of Python.

Preparing the feature array

Our general approach is to define a *`GenomicArrayOfSets`* and fill it with all the features we would like to get counts for.

Similar to the code shown in the *Tour*, we prepare such an object from the GTF file for yeast as follows:

```
import HTSeq

gtf_file = HTSeq.GFF_Reader( "Saccharomyces_cerevisiae.SGD1.01.56.gtf.gz" )
exons = HTSeq.GenomicArrayOfSets( "auto", stranded=True )

for feature in gtf_file:
    if feature.type == "exon":
       exons[ feature.iv ] += feature.attr["gene_id"]
```

A few things might be noteworthy here: For each exon, we just store the gene ID in the genomic array. Hence, all exons from the same gene are represented with the same string. This is deliberate, as we want to count on the level of genes, not exons, but could be done differently: storing the whole `feature` object in the GenomicArrayOfSets uses up noticeably more memory but allows to access more information in downstream processing.

Also note that in a GTF file, an exon that appears in several transcripts appear once for each transcripts. Because all these exons are represented by the same name, they will be collapsed to a single value in the GenomicArrayOfSets.

GTF files are not the only source of feature annotations. One could, as well, read a BED file or other text file with genomic coordinates of, say, ChIP-Seq peaks, putative enhancers, or any other kind of data. For example, if we have a tab-separated text file with feature coordinates in four columns – feature ID, chromosome, start, end – we might use:

```
features =  HTSeq.GenomicArrayOfSets( "auto", stranded=False )
for line in open( "myfeatures.txt" ):
   fields = line.split( "\t" )
   iv = HTSeq.GenomicInterval( fields[1], int(fields[2]), int(fields[3]) )
   features[ iv ] += fields[0]
```

Here, we have assumed that the coordinates follow Python conventions: The first base of a chromosome is numbered 0, not 1, and the end position is not included in the interval. Remember to subtract or add 1 as necessary if this is not the case with your input data.

Counting ungapped single-end reads

We start with the easiest case, that of ungapped single-end reads. We first recapitulate points already shown in the *Tour* and then add further refinements in the following.

If we have a SAM file with unmapped reads, we might use the following code:

```
import collections
counts = collections.Counter( )

almnt_file = HTSeq.SAM_Reader( "my_alignments.sam" )
for almnt in almnt_file:
   if not almnt.aligned:
      count[ "_unmapped" ] += 1
      continue
   gene_ids = set()
   for iv, val in features[ almnt.iv ].steps():
      gene_ids |= val
   if len(gene_ids) == 1:
      gene_id = list(gene_ids)[0]
      counts[ gene_id ] += 1
   elif len(gene_ids) == 0:
      counts[ "_no_feature" ] += 1
   else:
      counts[ "_ambiguous" ] += 1

for gene_id in counts:
   print gene_id, counts[ gene_id ]
```

For the benefit of readers with only limited Python knowledge, we go through this code chunk step by step:

The variable `counts` contains a dictionary, which will associate gene IDs with read counts. We use a variant of Python's usual `dict` type, namely the `Counter` class from the `collections` module in the standard library (from Python 2.7 onwards), which initialized any new key with the value zero. (Users of Python 2.6 can use `collections.defaultdict(int)` instead.)

We then instantiate a `SAM_Reader`` object (If you have a BAM file, use `BAM_Reader` instead) and run through all its record in a `for` loop. As described in the Tour, each record in the SAM file is vprovided to the loop body in the variable `almnt`.

We first check whether the read might be unaligned, and if so, increment a special counter that we call `_unmapped` (with an underscore as prefix to distinguish it from gene IDs).

For the aligned reads, the alignment's genomic interval, `almnt.iv`, shows us the interval covered by the read. Using this as an index to `feature` gives us a view on this stretch of the `feature` container, in which we had stored the exons. The iterator `features[ almnt.iv ].steps()` returns pairs of the form `(iv, val)`, where `iv` is a genomic interval and `val` is the set of gene IDs associated with the exons overlapping this step. Using the `|=` operator, we get the union of the sets of all the steps in the initially empty set `gene_ids`, which, at the end, contains the gene IDs of all genes that the read overlaps. Remember that a `set` can contain each element at most once. Hence, even if we see the same gene in several steps (for example because the read overlaps with several exons), we still get it only once in `gene_ids`.

We then treat three possible cases, namely that the set `gene_ids` contains exactly one element, that it is empty, or that it contains mroe than one element. The first case is the desired one: The read overlaps with precisely one gene, and we hence increase the count for this gene by one. Note that we need the idiom `list(gene_ids)[0]` to extract the name of this single gene from the set. If the read did not overlap with a gene (`len(gene_ids) == 0`), we increase a special counter that we call `_no_feature`.

What should we do if the read overlaps more than one gene? Here, one might now come up with sophisticated logic to decide which gene to count the read for. To keep things simple, however, we simply count the read for none of the overlapped genes and instead increase the special counter `_ambiguous` .

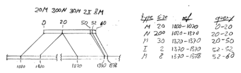
In the final two lines, we loop through the counter to print out the counts.

Counting gapped single-end reads

CIGAR Operations

The above code can be used as is e.g. for ChIP-Seq data, but for RNA-Seq data, we need an additional ingredient: When sequencing RNA, many reads will pass over an exon-exon junction and hence align to two (or more) disjunct intervals on the genome, tyically with an intron in between. If the reads have been aligned with a splice-aware alignment tool, such gapped alignment is indicated in the SAM file by the CIGAR string.

HTSeq parses the CIGAR string and presents it in the `cigar` slot of a class:*SAM_Alignment* object as a list of class:*CigarOperation* objects. As an example, consider a SAM alignment record describing a read that has been aligned to position 1000 on the '+" strand of chromosome `chr1`, with CIGAR string `20M300N30M2I8M`. Following the SAM specification (please read it first if you are unfamiliar with CIGAR strings), this means an alignment as depicted here:



type	size	ref	query
M	20	1000-1020	0-20
N	200	1020-1320	20-20
M	30	1320-1370	20-50
I	2	1370-1370	52-52
M	8	1370-1378	52-60

[[TO DO: Nicer image, add "chr1:"]]

The `SAM_Alignment` object will hence have in its `cigar` slot a list of 5 objects, each giving the information of one row of the table. Note how some operations are associated with zero-length intervals on either the reference (i.e., chromosome) or the query (i.e., read). For example, the intron (`N200`) spans 200 bp on the chromosome (1020-1320) but a zero-length interval on the read (20-20). In this manner, the `CigarOperation` class conveniently shows which intervals are affected by which operation.

Counting with gapped reads

In the code above, we used this for loop

```
gene_ids = set()
for iv, val in features[ almnt.iv ].steps():
    gene_ids |= val
```

to collect the gene IDs of all exons overlapped by the reads interval. For loop runs over the whole interval covered by the aligned read, i.e., in the figure above, it would run from position 1000 to position 1378 on chromosome 1, including the alignment gap from 1020 to 1320. By looking at each cigar operation separately we can correctly skip the gaps. We only need to replace the for loop with the following double loop

```
gene_ids = set()
for cigop in almnt.cigar:
   if cigop.type != "M":
      continue
   for iv, val in features[ cigop.ref_iv ].steps():
      gene_ids |= val
```

The outer loop goes through the CIGAR operation, skipping all but the *match* operations, and the inner loop inspects the steps covered by the match operations and collects the gene_ids in the `gene_ids` set variable. The rest of the code stays as above.

Of course, custom logic can be implemented her to infer useful information from other cigar operation types, but for the simple counting task at hand here, we do not need this.

Dealing with multiple alignments

If the aligner finds multiple alignments for a read, these are typically reported in multiple SAM records. If the SAM file unsorted or sorted by alignment position, it is hard to look at all the possible alignments of a given read together, because the records with the alignments for a given read are spread throughout the file. If the purpose of the counting is subsequent testing for differential expression, it is often safest, anyway, to skip all multiply aligned reads (because a read that is counted for several genes may cause spurious calls of differential expression), and then, we merely need to recognize that a read has multiple alignments. In the *htseq-count* script (see ref:*count*), this is done by two means: First, many (but not all aligners) use the optional field "NH", which indicates the number of reported alignments. Testing for `almnt.optional_field("NH") > 1` allows to find these read. Also, if there are multiple good alignments, without one of them being considered by the aligner to be more likely than the others, then the alignment quality (also denoted mapping quality, MAPQ, in the SAM specification) should be 3 or less. Hence, if one skips all reads with an alignment quality below, say, 10 (`almnt.aQual < 10`), one will skip over all multiply aligned reads (provided the aligner indicates the mapping quality correctly, which is not always the case).

For more advanced use cases, it may be desirable to inspect all reported alignment, for example, to the chose one using some custom logic, or to aggregate information over all of them. If the SAM or BAM file has been sorted by *read name* then alternative alignments for the same read will be in adjacent lines or records. To facilitate handling this case, HTSeq offers the function function:*bundle_multiple_alignments*. It takes an iterator over *`SAM_Alignment`* objects (e.g., a *`SAM_Reader`* or *`BAM_Reader`* object) and returns an iterator over lists of *`SAM_Alignment`* objects. Each list contains only records describing alignments for the same read. For this to work, the SAM file has to be sorted by read name to ensure that mutiple alignments for the same read appear in adjacent records.

Handling paired-end reads

In the case of paired-end alignments, we will typically want to count read pairs, not reads. After all, the fragment (and not the reads from either of its ends) are the actual evidence for a gene's expression that we want to count. Therefore, we want to process the alignment information for the two mated ends together.

First a quick review of how alignments for paired-end data are presented in a SAM file: The two "mated" reads referring to the two end of a DNA fragment are reported in two separate records. The fact that the records describe the same fragment can be seen from the fact that they have the same read name (given by the `read.name` slot). That they refer to opposite ends can be seen from the respective bits in the FLAG field, which are exposed by the *`SAM_Alignment.pe_which`* slot of the *`SAM_Alignment`* class, which takes the values `first` or `second` (or `unknown` if not specified in the flag field, or `not_paired_end` if an alignment of a single-end read is represented) . If the read pair has multiple alignments, each alignment is reported by a pair of SAM records. As corresponding records are not necessarily in adjacent lines, they are "linked" by the mate position fields (called RNEXT and PNEXT in the SAM specification), which are exposed by the slot `SAM_Alignment.mate_pos`, which contains a *`GenomicPosition`* object. The two records describing the two halves of a given alignment can be recognized as being correspondent from the fact that each record's `mate_pos` attribute is equal to the starting position (given by``iv.start_as_pos``).

Note that all the SAM records for a given read pair may be spread throughout the file. Only if the file is sorted by read name can we expect them to be at adjacent places, and even then, the records for multiple alignments can be intermixed.

To facilitate handling paired-end alignments, HTSeq offers the function *`pair_SAM_alignments()`*. This function expects an iterator over SAM records (typically, a *`SAM_Reader`* or *`BAM_Reader`* object) and returns an iterator over pairs (i.e., tuples of length 2) of *`SAM_Alignment`* records, with the first element being the alignment of the read from the first sequencing pass (i.e., from the 5' end of the DNA fragment) and the second element the corresponding alignment from the second pass (i.e., the 3' read). The function expects the SAM file to be sorted by read name. It proceeds by reading in consecutive records with the same read name and storing them in a list. Once it finds a record with a differing read name, the function goes through the list, sorts its content into pairs of corresponding alignment records and yields these pairs. If the function's option `bundle`` [TODO: add description of "bundle" in alignment.rst, too] is set to ``True`, the function does not yield the pairs separately but instead yields a list of all alignment pairs for the same read.

Using these features, we can modify our counting loop as follows for paired-end data:

```
almnt_file = HTSeq.SAM_Reader( "my_paired_alignments.sam" )
counts = collections.Counter( )
for bundle in HTSeq.pair_SAM_alignments( almnt_file, bundle=True ):
   if len(bundle) != 1
      continue  # Skip multiple alignments
   first_almnt, second_almnt = bundle[0]  # extract pair
   if not first_almnt.aligned and second_almnt.aligned:
      count[ "_unmapped" ] += 1
      continue
   gene_ids = set()
   for iv, val in features[ left_almnt.iv ].steps():
      gene_ids |= val
   for iv, val in features[ right_almnt.iv ].steps():
      gene_ids |= val
   if len(gene_ids) == 1:
      gene_id = list(gene_ids)[0]
      counts[ gene_id ] += 1
   elif len(gene_ids) == 0:
      counts[ "_no_feature" ] += 1
   else:
      counts[ "_ambiguous" ] += 1

for gene_id in counts:
   print gene_id, counts[ gene_id ]
```

Note that here, we skip reads if only one of two mates are aligned. Of course, one could choose as well to count such a pair for the gene to which the aligned mate has been mapped.

The need to sort paired-end SAM files by read name can be an inconvenience, because many aligners output the SAM file sorted by position. In many use case, we can expect that the two ends of the same read will align to positions close to each other on the genome. Then, an alternative strategy to pair up alignment records is go through the SAM file, which has been sorted by position, and keep a dictionary of alignment records whose partner record has not been found yet. For each record, we check the dictionary for its partner (i.e., for a record with the same read name and matching position information). If we find the partner, we remove it from the dictionary and yield both together as a pair. If the partner is not in the dictionary, the record is stored in the dictionary to wait for its partner to come along. As long as mated records are not too far from each other in the file, the dictionary will only contain a manageable number of records. Only if reads are often very far from each other (e.g., because the file was not sorted by position), the dictionary might become too large to fit into memory. HTSeq offers this manner of pairing up alignment records in the function `pair_SAM_alignment_with_buffer()`, which can be used in the same manner as `pair_SAM_alignment()`, but takes one optional additional argument, the maximum size of the buffer (by default, 3 million).

CHAPTER 6

Reference overview

This page offers a brief overview over all classes and functions offered by HTSeq.

Parser and record classes

For all supported file formats, parser classes (called `Reader`) are provided. These classes all instatiated by giving a file name or an open file or stream and the function as iterator generators, i.e., the parser objects can be used, e.g., in a `for` loop to yield a sequence of objects, each desribing one record. The table gives the parse class and the record class yielded. For details, see the linked documentation

For most formats, functionality for writing files of the format is provided. See the detailed documentation as these methods and classes have varying semantics.

File format	typical content	Parser class for reading	Record class yielded by parser	Method/class method for writing
FASTA	DNA sequences	*FastaReader*	*Sequence*	Sequence.write_to_fasta_file()
FASTQ	sequenced reads	*FastqReader*	*SequenceWithQualities*	*SequenceWithQuality.write_to_fastq_file()*
GFF (incl. GFF3 and GTF)	genomic annotation	*GFF_Reader*	*GenomicFeature*	*GenomicFeature.get_gff_line()*
BED	score data or annotation	*BED_Reader*	*GenomicFeature*	
Wiggle (incl. BedGraph)	score data on a genome	*WiggleReader*	pair: (*GenomicInterval*, float)	*GenomicArray.write_bedgraph_file()*
SAM	aligned reads	*SAM_Reader*	*SAM_Alignment*	*SAM_Alignment.get_sam_line()*
BAM	aligned reads	*BAM_Reader*	*SAM_Alignment*	BAM_Writer
VCF	variant calls	*VCF_Reader*	*VariantCall*	
Bowtie (legacy format)	aligned reads	*BowtieReader*	*BowtieAlignment*	
SolexaExport (legacy format)	aligned reads	*SolexaExportReader*	*SolexaExportAlignment*	

Most parser classes are sub-classes of class *FileOrSequence*, which users will, however, rarely use directly.

Specifying genomic positions and intervals

The class *GenomicInterval* specifies an interval on a chromosome (or contig or the like). It is defined by specifying the chromosome (or contig) name, the start and the end and the strand. Convenience methods are offered for different ways of accessing this information, and for tetsing for overlap between intervals. A *GenomicPosition*, technically a GenomicInterval of length 1, refers to a single nucleotide or base-pair position.

Objects of these classes are used internally wherever intervals or positions are represented, especially in record classes and as index keys to genomic array.

See page *Genomic intervals and genomic arrays* for details.

Genomic arrays

The classes *GenomicArray* and *GenomicArrayOfSets* are container classes to store data associated with genomic positions or intervals.

See page *Genomic intervals and genomic arrays* for details.

Special features for SAM/BAM files

The class *CigarOperation* offers a convenient way to handle the information encoded in the CIGAR field of SAM files.

The functions *pair_SAM_alignments()* and *pair_SAM_alignments_with_buffer()* help to `pair up` the records in a SAM file that describe a pair of alignments for mated reads from the same DNA fragment.

Similarly, the function `bundle_multiple_alignments()` bundles multiple alignment record pertaining to the same read or read pair.

See page *Read alignments* for details.

CHAPTER 7

Sequences and FASTA/FASTQ files

Sequence

A **Sequence** object holds a DNA sequence. Besides the actual sequence, an object may also hold a name.

Instantiation

class `HTSeq.`**`Sequence`**(*seq*, *name="unnamed"*)

Pass the DNA sequence and, optionally, a name or ID to the constructor:

```
>>> myseq = HTSeq.Sequence( b"ACCGTTAC", "my_sequence" )
```

(If the name is omitted, the default `"unnamed"` is used.)

Attributes

`Sequence.`**`seq`**
`Sequence.`**`name`**
`Sequence.`**`descr`**

The information can be accessed via the attributes **seq** and **name**, which are strings:

```
>>> myseq.seq
b'ACCGTTAC'
>>> myseq.name
'my_sequence'
```

There is a third attribute, called **descr**, which is by default `None` but may contain a "description". See class *`FastaReader`* for more information.

Representation and string conversion

The **__repr__** method gives name and length:

```
>>> myseq
<Sequence object 'my_sequence' (length 8)>
```

The **__str__** method returns just the sequence:

```
>>> print(myseq)
ACCGTTAC
```

Note that the length of a sequence is the number of bases:

```
>>> len( myseq )
8
```

Subsetting Subsetting works as with strings:

```
>>> myseq2 = myseq[3:5]
>>> myseq2.name
'my_sequence[part]'
>>> myseq2.seq
b'GT'
```

(Note that "`[part]`" is appended to the name of the subsetted copy.)

Reverse complement

`Sequence.`**`get_reverse_complement`**`()`

```
>>> print(myseq.get_reverse_complement())
GTAACGGT
>>> rc = myseq.get_reverse_complement()
>>> rc.name
'revcomp_of_my_sequence'
>>> rc.seq
b'GTAACGGT'
```

Writing to FASTA file

To write **Sequence** objects into a FASTA file, open a text file for writing, then call **write_to_fasta_file** for each sequence, providing the open file handle as only argument, and close the file:

```
>>> my_fasta_file = open( "test.fa", "w" )
>>> myseq.write_to_fasta_file( my_fasta_file )
>>> my_fasta_file.close()
```

To read from a FASTA file, see class *`FastaReader`*.

Extended UIPAC letters These are not (yet) supported. A sequence should only contain A, C, G, T and N.

Counting bases

For read quality assessment, it is often helpful to count the proportions of called bases, stratified by position in the read. To obtain such counts, the following idiom is helpful:

```
>>> import numpy
>>> reads = HTSeq.FastqReader( "yeast_RNASeq_excerpt_sequence.txt" )
>>> counts = numpy.zeros( ( 36, 5 ), numpy.int )
>>> for read in reads:
...     read.add_bases_to_count_array( counts )
>>> counts
array([[16194,  2048,  4017,  2683,    57],
       [10716,  3321,  4933,  6029,     0],
       [ 7816,  5024,  5946,  6213,     0],
       ...
       [ 8526,  4812,  5460,  6197,     4],
       [ 8088,  4915,  5531,  6464,     1]])
```

Here, a two-dimensional numpy array of integer zeroes is defined and then passed to the **add_bases_to_count_array** method of each Sequence object obtained from the Fastq file. The method *add_bases_to_count_array* adds, for each base, a one to one of the array elements such that, in the end, the 36 rows of the array correspond to the positions in the reads (all of length 36 bp in this example), and the 5 columns correspond to the base letters 'A', 'C', 'G', 'T', and 'N', as given by the constant **base_to_columns**

`base_to_column = { 'A': 0, 'C': 1, 'G': 2, 'T': 3, 'N': 4 }`

Hence, we can get the proportion of 'C's in each position as follows:

```
>>> counts = numpy.array( counts, numpy.float )
>>> #counts[ : , HTSeq.base_to_column['C'] ] / counts.sum(1)
array([ 0.08192328,  0.13284531,  0.20096804,  0.16872675,  0.21200848,
        ...
        0.18560742,  0.19236769,  0.19088764,  0.17872715,  0.1924877 ,
        0.19660786])
```

(Here, we first convert the count array to type `float` to allow to proper division, and then divide the second column (`HTSeq.base_to_column['C']`) by the row-wise sums (`counts.sum(1)`; the `1` requests summing along rows).)

Trimming reads

`Sequence.`**`trim_left_end`**(*pattern*, *mismatch_prop = 0.*)
`Sequence.`**`trim_right_end`**(*pattern*, *mismatch_prop = 0.*)

In high-throughput sequencing, reads are sometimes contaminated with adapters or sequencing primers. These function take a pattern and attempt to match either the right end of the pattern to the left end of the sequence (`trim_left_end`) or the left end of the pattern to the right end of the sequence (`trim_right_end`). The match is the trimmed off.

Here is an example:

```
>>> seq2 = HTSeq.Sequence( b"ACGTAAAGCGGTACGGGGGG" )
>>> left_seq = HTSeq.Sequence( b"CCCACG" )
>>> print(seq2.trim_left_end( left_seq ))
TAAAGCGGTACGGGGGG
```

The right end of the pattern ("ACG") matched the left end of the sequence, and has hence been trimmed off.

The optional argument `mismatch_prop` is the number of allowed mismatches as proportion of the length of the match:

```
>>> right_seq = HTSeq.Sequence( b"GGGTGGG" )
>>> print(seq2.trim_right_end( right_seq ))
ACGTAAAGCGGTACGGG
>>> print(seq2.trim_right_end( right_seq, 1/6. ))
ACGTAAAGCGGTAC
>>> print(seq2.trim_right_end( right_seq, 1/7. ))
ACGTAAAGCGGTACGGG
```

Here, if we allow at least one mismatch per six bases, the whole pattern gets cut off.

If you have quality information, you can use this, too, to specify the allowed amount of mismatch. See *`SequenceWithQualities.trim_left_end_with_quals()`* and

SequenceWithQualities.trim_left_end_with_quals().

SequenceWithQualities

The sequences obtained from high-throughput sequencing devices (in the following also referred to as "reads") typically come with *base-call quality scores*, which indicate how sure the software was that the right base was called. The class `SequenceWithQualities` represents such reads.

`SequenceWithQualities` is a daughter class of *Sequence* and inherits all its features.

Instantiation

class `HTSeq.`**`SequenceWithQualities`**(*seq*, *name qualstr*, *qualscale="phred"*)

A `SequenceWithQualities` can be instantiated as a `Sequence`, but now with a third argument, the quality string:

```
>>> myread = HTSeq.SequenceWithQualities( b"ACGACTGACC", "my_read", b"IICGAB#
↪#(!" )
```

The quality string is interpreted as Sanger-encoded string of Phred values, as defined in the FASTQ format specification, i.e., each letter in the quality string corresponds to one base in the sequence and if the value 33 is subtracted from the quality characters ASCII value, the Phred score is obtained.

The Phred scores can then be found in the slot `qual`:

```
>>> myread.qualstr
b'IICGAB##(!'
>>> myread.qual
array([40, 40, 34, 38, 32, 33,  2,  2,  7,  0], dtype=uint8)
```

If the quality string follows the *Solexa FASTQ* specification, the value to be subtracted is not 33 but 64. If you pass a quality string in this format, set `qualscale="solexa"`.

Prior to version 1.3, the SolexaPipeline software used a yet another style of encoding quality string. If you want to use this one, specify `qualscale="solexa-old"`

Attributes

As for `Sequence` objects, there are attributes `name`, `seq`, and `descr`.

Furthermore, we now have the attributes `qual` and `qualstr`, already mentioned above.

`SequenceWithQuality.`**`qual`**

`qual` is a `numpy` array of data type *integer*, with as many elements as there are bases. Each element is a *Phred score*. A Phred score *S* is defined to mean that the base caller estimates the probability *p* of the base call being wrong as *p* = -log10 (*S*/10).

Note that `qual` is always the probability, even if the `solexa-old` quality string format has been used, which encodes the odds *p* (1 - *p*), i.e., in that case, the odds are converted to probabilities.

`SequenceWithQuality.`**`qualstr`**

The quality string according to Sanger Phred encoding. In case the quality was originally given in `solexa` or `solexa-old` format, it is converted:

```
>>> read2 = HTSeq.SequenceWithQualities( b"ACGACTGACC", "my_read", b
↪"hhgddaZVFF", "solexa" )
>>> read2.qual
array([40, 40, 39, 36, 36, 33, 26, 22,  6,  6], dtype=uint8)
```

```
>>> read2.qualstr
b"IIHEEB;7''"
```

Writing to FASTQ file

`SequenceWithQuality.`**`write_to_fastq_file`**`(`*fasta_file*`)`

To write `SequenceWithQualities` objects into a FASTQ file, open a text file for writing, then call `write_to_fastq_file` for each sequence, providing the open file handle as only argument, and close the file:

```
>>> my_fastq_file = open( "test.fq", "w" )
>>> myread.write_to_fastq_file( my_fastq_file )
>>> my_fastq_file.close()
```

Note that the reads will always be written with quality strings in Sanger encoding.

To read from a FASTQ file, see class *`FastqReader`*.

Counting quality values

Similar to `Sequence.add_bases_to_count_array()`, this method counts the occuring quality values stratified by position. This then allows to calculate average qualities as well as histograms.

Here is a usage example:

```
>>> import numpy
>>> reads = HTSeq.FastqReader( "yeast_RNASeq_excerpt_sequence.txt", "solexa"␣
↪)
>>> counts = numpy.zeros( ( 36, 41 ), numpy.int )
>>> for read in reads:
...     read.add_qual_to_count_array( counts )
>>> #counts
array([[   0,     0,   64, ...,     0,     0,     0],
       [   0,     0,   93, ...,     0,     0,     0],
       ...,
       [   0,     0, 2445, ...,     0,     0,     0],
       [   0,     0, 2920, ...,     0,     0,     0]])
```

The value `counts[i,j]` is then the number of reads for which the base at position `i` hat the quality scores `j`. According to the Fastq standard, quality scores range from 0 to 40; hence, the array is initialized to have 41 columns.

Trimming reads

`SequenceWithQualities.`**`trim_left_end_with_quals`**`(`*pattern*, *max_mm_qual = 5*`)`
`SequenceWithQualities.`**`trim_right_end_with_quals`**`(`*pattern*, *max_mm_qual = 5*`)`

These methods work as *`Sequence.trim_left_end()`* and *`Sequence.trim_right_end()`* (which are, of course, avilable for `SequenceWithQualities` objects, too). The difference is, that for the `_with_quals` trimming methods, the maximum amount of allowed mismatch is specified as the maximum value that the sum of the quality scores of the mismatched bases may take.

TODO: Add example

FastaReader

The FastaReader class allows to read and parse a FASTA file. It can generates an iterator of `Sequence` objects.

class `HTSeq.`**`FastaReader`**(*filename_or_sequence*)

As daughter class of `FileOrSequence`, `FastaReader` can be instantiated with either a filename, or with a sequence. See *`FileOrSequence`* for details.

Example 1 The typical use for FastaReader is to go through a FASTA file and do something with each sequence, e.g.:

```
>>> for s in HTSeq.FastaReader( "fastaEx.fa" ):
...     print("Sequence '%s' has length %d." % ( s.name, len(s) ))
Sequence 'sequence1' has length 72.
Sequence 'sequence2' has length 70.
```

Example 2 Often, one might to read a whole Fasta file into memory to access it as a dict. This is a good idiom for this purpose:

```
>>> sequences = dict( (s.name, s) for s in HTSeq.FastaReader("fastaEx.fa") )
>>> sequences["sequence1"].seq
b'AGTACGTAGTCGCTGCTGCTACGGGCGCTAGCTAGTACGTCACGACGTAGATGCTAGCTGACTAAACGATGC'
```

FastqReader

The **FastqReader** class works similar to *`FastaReader`*. It reads a Fastq file and generates an iterator over *`SequenceWithQualities`* objects.

class `HTSeq.`**`FastqReader`**(*filename_or_sequence*, *qual_scale="phred"*)

As daughter class of `FileOrSequence`, `FastaReader` can be instantiated with either a filename, or with a sequence. See *`FileOrSequence`* for details.

By default, the quality strings are assumed to be encoded according to the Sanger/Phred standard. You may alternatively specify `"solexa"` or `"solexa-old"` (see `SequenceWithQuality`).

CHAPTER 8

Genomic intervals and genomic arrays

GenomicInterval

A genomic interval is a consecutive stretch on a genomic sequence such as a chromosome. It is represented by a `GenomicInterval` object.

Instantiation

class `HTSeq.`**`GenomicInterval`**(*chrom*, *start*, *end*, *strand*)

`chrom` **(string)** The name of a sequence (i.e., chromosome, contig, or the like).

`start` **(int)** The start of the interval. Even on the reverse strand, this is always the smaller of the two values 'start' and 'end'. Note that all positions should be given and interpreted as 0-based value!

`end` **(int)** The end of the interval. Following Python convention for ranges, this in one more than the coordinate of the last base that is considered part of the sequence.

`strand` **(string)** The strand, as a single character, `'+'`, `'-'`, or `'.'`. `'.'` indicates that the strand is irrelevant.

Representation and string conversion The class's `__str__` method gives a spcae-saving description of the interval, the `__repr__` method is a bit more verbose:

```
>>> iv = HTSeq.GenomicInterval( "chr3", 123203, 127245, "+" )
>>> print(iv)
chr3:[123203,127245)/+
>>> iv
<GenomicInterval object 'chr3', [123203,127245), strand '+'>
```

Attributes

`GenomicInterval.`**`chrom`**
`GenomicInterval.`**`start`**
`GenomicInterval.`**`end`**
`GenomicInterval.`**`strand`**
as above

`GenomicInterval.`**`start_d`**

The “directional start” position. This is the position of the first base of the interval, taking the strand into account. Hence, this is the same as `start` except when `strand == '-'`, in which case it is `end-1`.

Note that if you set `start_d`, both `start` and `end` are changed, such that the interval gets the requested new directional start and its length stays unchanged.

`GenomicInterval.`**`end_d`**

The “directional end”: The same as `end`, unless `strand=='-'`, in which case it is `start-1`. This convention allows to go from `start_d` to `end_d` (not including, as usual in Python, the last value) and get all bases in “reading” direction.

`end_d` is not writable.

`GenomicInterval.`**`length`**

The length is calculated as end - start. If you set the length, `start_d` will be preserved, i.e., `end` is changed, unless the strand is `-`, in which case `start` is changed.

`GenomicInterval.`**`start_as_pos`**
`GenomicInterval.`**`end_as_pos`**
`GenomicInterval.`**`start_d_as_pos`**
`GenomicInterval.`**`end_d_as_pos`**

These attributes return *`GenomicPosition`* objects referring to the respective positions.

Directional instantiation

`HTSeq.`**`GenomicInterval_from_directional`**(*chrom*, *start_d*, *length*, *strand=”.”*)

This function allows to create a new `GenomicInterval` object specifying directional start and length instead of start and end.

Methods

`GenomicInterval.`**`is_contained_in`**(*iv*)
`GenomicInterval.`**`contains`**(*iv*)
`GenomicInterval.`**`overlaps`**(*iv*)

These methods test whether the object is contained in, contains, or overlaps the second `GenomicInterval` object `iv`.

For any of of these conditions to be true, the `start` and `end` values have to be appropriate, and furthermore, the `chrom` values have to be equal and the `strand` values consistent. The latter means that the strands have to be the same if both intervals have strand information. However, if at least one of the objects has `strand == '.'`, the strand information of the other object is disregarded.

Note that all three methods return `True` for identical intervals.

`GenomicInterval.`**`xrange`**(*step = 1*)
`GenomicInterval.`**`xrange_d`**(*step = 1*)

These methods yield iterators of :class:GenomicPosition objects from `start` to `end` (or, for `xrange_d` from `start_d` to `end_d`).

`GenomicInterval.`**`extend_to_include`**(*iv*)

Change the object’s `start` end `end` values such that `iv` becomes contained.

Special methods

`GenomicInterval` implements the methods necessary for

- obtaining a copy of the object (the `copy` method)
- pickling the object
- representing the object and converting it to a string (see above)

- comparing two GenomicIntervals for equality and inequality
- hashing the object

GenomicPosition

A `GenomicPosition` represents the position of a single base or base pair, i.e., it is an interval of length 1, and hence, the class is a subclass of :class:GenomicInterval.

class `HTSeq.`**`GenomicPosition`**`(`*chrom*, *pos*, *strand='.'*`)`
The initialisation is as for a :class:GenomicInterval object, but no `length` argument is passed.

Attributes

`HTSeq.`**`pos`**
pos is an alias for *`GenomicInterval.start_d`*.

All other attributes of *`GenomicInterval`* are still exposed. Refrain from using them, unless you want to use the object as an interval, not as a position. Some of them are now read-only to prevent the length to be changed.

GenomicArray

A `GenomicArray` is a collection of `ChromVector` objects, either one or two for each chromosome of a genome. It allows to access the data in these transparently via *`GenomicInterval`* objects.

Note: ‘‘GenomicArray‘‘’s interface changed significantly in version 0.5.0. Please see the Version History page.

Instantiation

class `HTSeq.`**`GenomicArray`**`(`*chroms*, *stranded=True*, *typecode='d'*, *storage='step'*, *memmap_dir=''*`)`

Creates a `GenomicArray`.

If `chroms` is a list of chromosome names, two (or one, see below) `ChromVector` objects for each chromosome are created, with start index 0 and indefinite length. If `chroms` is a `dict`, the keys are used for the chromosome names and the values should be the lengths of the chromosome, i.e., the ChromVectors index ranges are then from 0 to these lengths. (Note that the term chromosome is used only for convenience. Of course, you can as well specify contig IDs or the like.) Finally, if `chroms` is the string `"auto"`, the GenomicArray is created without any chromosomes but whenever the user attempts to assign a value to a yet unknown chromosome, a new one is automatically created with *`GenomicArray.add_chrom()`*.

If `stranded` is `True`, two `StepVector` objects are created for each chromosome, one for the '+' and one for the '-' strand. For `stranded == False`, only one `StepVector` per chromosome is used. In that case, the strand argument of all `GenomicInterval` objects that are used later to specify regions in the `GenomicArray` are ignored.

The `typecode` determines the data type and is as in `numpy`, i.e.:

- `'d'` for float values (C type 'double'),
- `'i'` for int values,
- `'b'` for Boolean values,
- `'O'` for arbitrary Python objects as value.

The storage mode determines how the ChromVectors store the data internally:

- mode `'step'`: A step vector is used. This is the default and useful for large amounts of data which may stay constant along a range of indices. Each such step is stored internally as a node in a red-black tree.
- mode `'ndarray'`: A 1D numpy array is used internally. This is useful if the data changes a lot, and steps are hence inefficient. Using this mode requires that chromosome lengths are specified.
- mode `memmap`: This is useful for large amounts of data with very short steps, where `step` is inefficient, but a numpy vectors would not fit into memory. A numpy memmap is used that stores the whole vector in a file on disk and transparently maps into memory windows of the data. This mode requires chromosome lengths, and specification of a directory, via the `memmap_dir` argument, to store the temporary files in. It is not suitable for type code `O`.

Attributes

`GenomicArray.`**`stranded`**
`GenomicArray.`**`typecode`**
see above

`GenomicArray.`**`chrom_vectors`**
a dict of dicts of `ChromVector` objects, using the chromosome names, and the strand as keys:

```
.. doctest::
```

```
>>> ga = HTSeq.GenomicArray( [ "chr1", "chr2" ], stranded=False )
>>> sorted(ga.chrom_vectors.items())
[('chr1', {'.': <ChromVector object, chr1:[0,Inf)/., step>}),
 ('chr2', {'.': <ChromVector object, chr2:[0,Inf)/., step>})]
>>> ga = HTSeq.GenomicArray( [ "chr1", "chr2" ], stranded=True )
>>> sorted([(st[0], sorted(st[1].items())) for st in ga.chrom_vectors.
↪items()])
[('chr1', [('+', <ChromVector object, chr1:[0,Inf)/+, step>),
           ('-', <ChromVector object, chr1:[0,Inf)/-, step>)]),
 ('chr2', [('+', <ChromVector object, chr2:[0,Inf)/+, step>),
           ('-', <ChromVector object, chr2:[0,Inf)/-, step>)])]
```

`GenomicArray.`**`auto_add_chroms`**
A boolean. This attribute is set to True if the GenomicArray was created with the `"auto"` arguments for the `chroms` parameter. If it is true, an new chromosome will be added whenever needed.

Data access To access the data, use :class:GenomicInterval objects.

To set an single position or an interval, use:

```
>>> ga[ HTSeq.GenomicPosition( "chr1", 100, "+" ) ] = 7
>>> ga[ HTSeq.GenomicInterval( "chr1", 250, 400, "+" ) ] = 20
```

To read a single position:

```
>>> ga[ HTSeq.GenomicPosition( "chr1", 300, "+" ) ]
20.0
```

`ChromVector.`**`steps`**(*self*)

To read an interval, use a `GenomicInterval` object as index, and obtain a `ChromVector` with a sub-view:

```
>>> iv = HTSeq.GenomicInterval( "chr1", 250, 450, "+" )
>>> v = ga[ iv ]
>>> v
<ChromVector object, chr1:[250,450)/+, step>
>>> list( v.steps() )
```

```
[(<GenomicInterval object 'chr1', [250,400), strand '+'>, 20.0),
 (<GenomicInterval object 'chr1', [400,450), strand '+'>, 0.0)]
```

Note that you get (interval, value) pairs , i.e., you can conveniently cycle through them with:

```
>>> for iv, value in ga[ iv ].steps():
...     print(iv, value)
chr1:[250,400)/+ 20.0
chr1:[400,450)/+ 0.0
```

`GenomicArray.`**`steps`**`(`*self*`)`

You can get all steps from all chromosomes by calling the arrays own `steps` method.

Modifying values

ChromVector implements the `__iadd__` method. Hence you can use +=:

```
>>> ga[ HTSeq.GenomicInterval( "chr1", 290, 310, "+" ) ] += 1000
>>> list( ga[ HTSeq.GenomicInterval( "chr1", 250, 450, "+" ) ].steps() )
[(<GenomicInterval object 'chr1', [250,290), strand '+'>, 20.0),
 (<GenomicInterval object 'chr1', [290,310), strand '+'>, 1020.0),
 (<GenomicInterval object 'chr1', [310,400), strand '+'>, 20.0),
 (<GenomicInterval object 'chr1', [400,450), strand '+'>, 0.0)]
```

To do manipulations other than additions, use Chromvector's `apply` method:

`ChromVector.`**`apply`**`(`*func*`)`

```
>>> ga[ HTSeq.GenomicInterval( "chr1", 290, 300, "+" ) ].apply( lambda␣
↪x: x * 2 )
>>> list( ga[ HTSeq.GenomicInterval( "chr1", 250, 450, "+" ) ].steps() )
[(<GenomicInterval object 'chr1', [250,290), strand '+'>, 20.0),
 (<GenomicInterval object 'chr1', [290,300), strand '+'>, 2040.0),
 (<GenomicInterval object 'chr1', [300,310), strand '+'>, 1020.0),
 (<GenomicInterval object 'chr1', [310,400), strand '+'>, 20.0),
 (<GenomicInterval object 'chr1', [400,450), strand '+'>, 0.0)]
```

Writing to a file

`GenomicArray.`**`write_bedgraph_file`**`(`*file_or_filename*, *strand=”.”*, *track_options=””*`)`

Write out the data in the GenomicArray as a BedGraph track. This is a subtype of the Wiggle format (i.e., the file extension is usually ”.wig”) and such files can be conveniently viewed in a genome browser, e.g., with IGB.

This works only for numerical data, i.e., `datatype 'i'` or `'d'`. As a bedgraph track cannot store strand information, you have to specify either `'+'` or `'-'` as the strand argument if your `GenomicArray` is stranded (`stranded==True`). Typically, you will write two wiggle files, one for each strand, and display them together.

Adding a chromosome

`GenomicArray.`**`add_chrom`**`(`*chrom*, *length=sys.maxint*, *start_index=0*`)`

Adds step vector(s) for a further chromosome. This is useful if you do not have a full list of chromosome names yet when instantiating the `GenomicArray`.

GenomicArrayOfSets

A `GenomicArrayOfSets` is a sub-class of *`GenomicArray`* that deal with the common special case of overlapping features. This is best explained by an example: Let's say, we have two features, `"geneA"` and `"geneB"`, that are at overlapping positions:

```
>>> ivA = HTSeq.GenomicInterval( "chr1", 100, 300, "." )
>>> ivB = HTSeq.GenomicInterval( "chr1", 200, 400, "." )
```

In a `GenomicArrayOfSets`, the value of each step is a `set` and so can hold more than one object. The __iadd__ method is overloaded to add elements to the sets:

```
>>> gas = HTSeq.GenomicArrayOfSets( ["chr1", "chr2"], stranded=False )
>>> gas[ivA] += "gene A"
>>> gas[ivB] += "gene B"
>>> [(st[0], sorted(st[1])) for st in gas[ HTSeq.GenomicInterval( "chr1", 0, 500, "."␣
↪) ].steps()]
[(<GenomicInterval object 'chr1', [0,100), strand '.'>,   []),
 (<GenomicInterval object 'chr1', [100,200), strand '.'>, ['gene A']),
 (<GenomicInterval object 'chr1', [200,300), strand '.'>, ['gene A', 'gene B']),
 (<GenomicInterval object 'chr1', [300,400), strand '.'>, ['gene B']),
 (<GenomicInterval object 'chr1', [400,500), strand '.'>, [])]
```

class `HTSeq.`**`GenomicArrayOfSets`**(*chroms*, *stranded = True*)

Instantiation is as in *`GenomicArray`*, only that `datatype` is always `'O'`.

CHAPTER 9

Read alignments

Concepts

There are a large number of different tools to align short reads to a reference. Most of them use their own output format, even though the SAM format seems to become the common standard now that many of the newer tools use it.

HTSeq aims to offer a uniform way to analyse alignments from different tools. To this end, for all supported alignment formats a parse class is offered that reads an alignment file and generates an iterator over the individual alignment records. These are represented as objects of a sub-class of *`Alignment`* and hence all offer a common interface.

So, you can easily write code that should work for all aligner formats. As a simple example, consider this function that counts the number of reads falling on each chromosome:

```
>>> import collections
>>> def count_in_chroms( alignments ):
...     counts = collections.defaultdict( lambda: 0 )
...     for almnt in alignments:
...         if almnt.aligned:
...             counts[ almnt.iv.chrom ] += 1
...     return counts
```

If you have a SAM file (e.g., from BWA or BowTie), you can call it with:

```
>>> sorted(count_in_chroms( HTSeq.SAM_Reader( "yeast_RNASeq_excerpt.sam" ) ).items())
[('2-micron', 46), ('I', 362), ('II', 1724), ('III', 365), ('IV', 3015),
 ('IX', 648), ('V', 999), ('VI', 332), ('VII', 2316), ('VIII', 932),
 ('X', 1129), ('XI', 1170), ('XII', 4215), ('XIII', 1471), ('XIV', 1297),
 ('XV', 2133), ('XVI', 1509)]
```

If, however, you have done your alignment with Eland from the SolexaPipeline, which uses the "Solexa export" format, you can use the same function, only using *`SolexaExportReader`* instead of *`SAM_Reader`*:

```
>>> count_in_chroms( HTSeq.SolexaExportReader( "mydata_export.txt" ) )
```

Both class generate iterators of similar objects. On the other hand, some formats contain more information and then the `Alignment` objects from these contain additional fields.

Parser classes

Depending on the format of your alignment file, choose from the following parsers:

class `HTSeq.`**`BowtieReader`**(*filename_or_sequence*)
class `HTSeq.`**`SAM_Reader`**(*filename_or_sequence*)
class `HTSeq.`**`BAM_Reader`**(*filename_or_sequence*)
class `HTSeq.`**`SolexaExportReader`**(*filename_or_sequence*)

All of these are derived from *`FileOrSequence`*. When asked for an iterator, they yield `Alignment` objects of types *`BowtieAlignment`*, *`SAM_Alignment`*, or *`SolexaExportAlignment`*. See below for their properties.

Adding support for a new format is very easy. Ask me if you need something and I can probably add it right-away. Alternatively, you can convert your format to the SAM format. The SAMtools contain Perl skripts to convert nearly all common formats.

`SAM_Reader.peek( num = 1 ):`
Peek into a SAM file or connection, reporting the first `num` records. If you then call an iterator on the `SAM_Reader`, the record will be yielded again.

`Alignment` and `AlignmentWithSequenceReversal`

class `HTSeq.`**`Alignment`**(*read*, *iv*)
This is the base class of all Alignment classes. Any class derived from `Alignment` has at least the following attributes:

`read`
The read. An object of type `SequenceWithQuality`. See there for the sub-attributes.

Note that some aligners store the reverse complement of the read if it was aligned to the '-' strand. In this case, the parser revers-complements the read again, so that you can be sure that the read is always presented as it was sequenced (see also *`AlignmentWithSequenceReversal`*).

`aligned`
A boolean. Some formats (e.g., those of Maq and Bowtie) contain only aligned reads (and the aligner collects the unaligned reads in a seperate FASTQ file if requested). For these formats, `aligned` is always `True`. Other formats (e.g., SAM and Solexa Export) list all reads, including those which could not be aligned. In that case, check `aligned` to see whether the read has an alignment.

`iv`
An object of class *`GenomicInterval`* or `None`.

The genomic interval to which the read was aligned (or `None` if `aligned=False`). See *`GenomicInterval`* for the sub-attributes. Note that different formats have different conventions for genomic coordinates. The parser class takes care of normalizing this, so that `iv` always adheres to the conventions outlined in :class:GenomicInterval. Especially, all coordinates are counted from zero, not one.

`paired_end`
A boolean. True if the read stems from a paired-end sequencing run. (Note: At the moment paired-end data is only supported for the SAM format.)

class `HTSeq.`**`AlignmentWithSequenceReversal`**(*read_as_aligned*, *iv*)

Some aligners store the reverse complement of the read if it was aligned to the '-' strand. For these aligners, the Alignment class is derived from `AlignmentWithSequenceReversal`, which undoes the reverse-complement if necessary to ensure that the `read` attribute always presents the read in the ordder in which it was sequenced.

To get better performance, this is done via lazy evaluation, i.e., the reverse complement is only calculated when the `read` attribute is accessed for the first time. The original read as read from the file is stored as well. You can access both with these attributes:

`read_as_aligned`

A *`SequenceWithQualities`* object. The read as it was found in the file.

`read_as_sequenced`

A *`SequenceWithQualities`* object. The read as it was sequenced, i.e., an alias for *`Alignment.read`*.

Format-specific Alignment classes

Note: All format-specific Alignment classes take a string as argument for their constructor. This is a line from the alignment file describing the alignment and is passed in by the corresponding `Reader` object. As you do not create `Alignment` objects yourself but get them from the `Reader` object you typically never call the constructor yourself.

class `HTSeq.`**`BowtieAlignment`**(*bowtie_line*)

`BowtieAlignment` objects contain all the attributes from *`Alignment`* and *`AlignmentWithSequenceReversal`*, and, in addition, these:

`reserved`

A string. The `reserved` field from the Bowtie output file. See the Bowtie manual for its meaning.

`substitutions`

A string. The substitutions string that describes mismatches in the format `22:A>C, 25:C>T` to indicate a change from A to C in position 22 and from C to T in position 25. No further parsing for this is offered yet.

class `HTSeq.`**`SAM_Alignment`**(*line*)

`BowtieAlignment` objects contain all the attributes from *`Alignment`* and *`AlignmentWithSequenceReversal`*, and, in addition, these:

`aQual`

An int. The alignment quality score in Phread style encoding.

`cigar`

A list of *`CigarOperation`* objects, as parsed from the extended CIGAR string. See *`CigarOperation`* for details.

`not_primary_alignment`

A boolean. Whether the alignment is secondary. (See SAM format reference, flag 0x0100. See also supplementary alignments, flag 0x0800.)

`failed_platform_qc`

A boolean. Whether the read failed a platform quality check. (See SAM format reference, flag 0x0200.)

`pcr_or_optical_duplicate`

A boolean. Whether the read is a PCR or optical duplicate. (See SAM format reference, flag 0x0400.)

`supplementary`

A boolean. Whether the alignment is supplementary. (See SAM format reference, flag 0x0800.)

These methods access the optional fields:

optional_field(*tag*)
Returns the optional field `tag`. See SAM format reference for the defined tags (which are two-letter strings).

optional_fields()
Returns a dict with all optional fields, using their tags as keys.

This method is useful to write out a SAM file:

get_sam_line()
Constructs a SAM line to describe the alignment, which is returned as a string.

Paired-end support

SAM_Alignment objects can represent paired-end data. If *Alignment.paired_end* is True, the following fields may be used:

mate_aligned
A boolean. Whether the mate was aligned

pe_which
A string. Takes one of the values "first", "second", "unknown" and "not_paired_end", to indicate whether the read stems from the first or second pass of the paired-end sequencing.

proper_pair
Boolean. Whether the mates form a proper pair. (See SAM format reference, flag 0x0002.)

mate_start
A *GenomicPosition* object. The start (i.e., left-most position) of the mate's alignment. Note that mate_start.strand is opposite to iv.strand for proper pairs.

inferred_insert_size
An int. The inferred size of the insert between the reads.

HTSeq.**pair_SAM_alignments**(*alnmt_seq*)
This function takes a generator of *SAM_Alignment* objects (e.g., a *SAM_Reader* object) and yields a sequence of pairs of alignments. A typical use may be:

```
for first, second in HTSeq.SAM_Reader( "some_paired_end_data.sam" ):
    print("Pair, consisting of")
    print("    ", first)
    print("    ", second)
```

Here, `first` and `second` are *SAM_Alignment* objects, representing two reads of the same cluster. For this to work, the SAM file has to be arranged such that paired reads are always in adjacent lines. As the SAM format requires that the query names (first column of the SAM file) is the same for mate pairs, this arrangement can easily be achieved by sorting the SAM file lines lexicographically.

Special care is taken to properly pair up multiple alignment lines for the same read.

In the SAM format, alignments for paired-end reads must be reported in paired alignment records. If the mate of an alignment record is missing, this fact is counted and, at the end, a warning stating the number of such violating reads is issued. The singleton alignments are yielded as pairs, with the alignment in the first or second element of the pair (depending on the sequencing pass it originates from) and the other element is set to `None`.

HTSeq.**pair_SAM_alignments_with_buffer**(*alignments*, *max_buffer_size=3000000*)
This function pairs up reads in a SAM file, in the same manner as *pair_SAM_alignments()* but does not require that mated alignments appear in adjacent records, i.e., the SAM file does not need to be sorted by read name beforehand. Rather, once the first alignment of a pair is encountered, it is stored in a buffer until its mated alignment is encountered, and then both are yielded together as pair. It is recommended that the data should

be sorted by position, because then, mated alignments will typicalle not be too distant from each other in the file and hence only a limited number of alignments have to be held concurrently in the buffer, thereby reducing memory needs. To avoid overflowing the system's memory, the function stops and raises an exception once the number of alignment records held in the buffer exceeds `max_buffer_size`.

class `HTSeq.`**`SolexaExportAlignment`**(*line*)

`SolexaExportAlignment` objects contain all the attributes from *`Alignment`* and *`AlignmentWithSequenceReversal`*, and, in addition, these:

`passed_filter`

A boolean. Whether the read passed the chastity filter. If `passed_filter==False`, then `aligned==False`.

`nomatch_code`

A string. For `aligned==False`, a code indicating why no match could be found. See the description of the 11th column of the Solexa Export format in the SolexaPipeline manual for the meaning of the codes. For `aligned==True`, `nomatch_code==None`.

Multiple alignments

`HTSeq.`**`bundle_multiple_alignments`**(*sequence_of_alignments*)

Some alignment programs, e.g., Bowtie, can output multiple alignments, i.e., the same read is reported consecutively with different alignments. This function takes an iterator over alignments (as provided by one of the alignment Reader classes) and bundles consecutive alignments regarding the same read to a list of Alignment objects and returns an iterator over these.

CIGAR strings

When reading in SAM files, the CIGAR string is parsed and stored as a list of `CigarOperation` objects. For example, assume, a 36 bp read has been aligned to the '+' strand of chromosome 'chr3', extending to the right from position 1000, with the CIGAR string `"20M6I10M"`. The function :function:parse_cigar spells out what this means:

```
>>> HTSeq.parse_cigar( "20M6I10M", 1000, "chr2", "+" )
[< CigarOperation: 20 base(s) matched on ref iv chr2:[1000,1020)/+, query iv [0,20) >,
 < CigarOperation: 6 base(s) inserted on ref iv chr2:[1020,1020)/+, query iv [20,26) >
↪,
 < CigarOperation: 10 base(s) matched on ref iv chr2:[1020,1030)/+, query iv [26,36) >
↪]
```

We can see that the map includes an insert. Hence, the affected coordinates run from 1000 to 1030 on the reference (i.e., the chromosome) but only from 0 to 36 on the query (i.e., the read).

We can convenient access to the parsed data by looking at the attributes of the three `CigarOperation` objects in the list.

class `HTSeq.`**`CigarOperation`**(...)

The available attributes are:

`type`

The type of the operation. One of the letters M, I, D, N, S, H, or P. Use the dict **cigar_operation_names** to transform this to names:

```
>>> sorted(HTSeq.cigar_operation_names.items())
[('D', 'deleted'),
 ('H', 'hard-clipped'),
 ('I', 'inserted'),
 ('M', 'matched'),
 ('N', 'skipped'),
 ('P', 'padded'),
 ('S', 'soft-clipped')]
```

`size`
The number of affected bases, an int.

`ref_iv`
A *`GenomicInterval`* specifying the affected bases on the reference. In case of an insertion, this is a zero-length interval.

`query_from`
`query_to`
Two ints, specifying the affected bases on the query (the read). In case of a deletion, `query_from == query_to`.

`check``()`
Checks the `CigarOperation` object for consitency. Returns a boolean.

CHAPTER 10

Features

The easiest way to work with annotation is to use *GenomicArray* with `typecode=='O'` or *GenomicArrayOfSets*. If you have your annotation in a flat file, with each line describing a feature and giving its coordinates, you can read in the file line for line, parse it (see the standard Python module `csv`), use the information on chromosome, start, end and strand to create a *GenomicInterval* object and then store the data from the line in the genomic array at the place indicated by the genomic interval.

For example, if you have data in a tab-separated file as follows:

```
>>> for line in open( "feature_list.txt" ):
...     print(line)
chr2  100    300     +       "gene A"
chr2 200     400     -       "gene B"
chr3 150     270     +       "gene C"
```

Then, you could load this information as follows:

```
>>> import csv
>>> genes = HTSeq.GenomicArray( [ "chr1", "chr2", "chr3" ], typecode='O' )
>>> for (chrom, start, end, strand, name) in \
...        csv.reader( open("feature_list.txt"), delimiter="\t" ):
...     iv = HTSeq.GenomicInterval( chrom, int(start), int(end), strand )
...     genes[ iv ] = name
```

Now, to see whether there is a feature at a given *GenomicPosition*, you just query the genomic array:

```
>>> print(genes[ HTSeq.GenomicPosition( "chr3", 100, "+" ) ])
None
>>> print(genes[ HTSeq.GenomicPosition( "chr3", 200, "+" ) ])
gene C
```

See *GenomicArray* and *GenomicArrayOfSets* for more sophisticated use.

`GFF_Reader` and `GenomicFeature`

One of the most common format for annotation data is GFF (which includes GTF as a sub-type). Hence, a parse for GFF files is included in HTSeq.

As usual, there is a parser class, called **GFF_Reader**, that can generate an iterator of objects describing the features. These objects are of type :class'GenomicFeature' and each describes one line of a GFF file. See Section *A tour through HTSeq* for an example.

class `HTSeq.`**`GFF_Reader`** (*filename_or_sequence*, *end_included=True*)

As a subclass of *`FileOrSequence`*, GFF_Reader can be initialized either with a file name or with an open file or another sequence of lines.

When requesting an iterator, it generates objects of type *`GenomicFeature`*.

The GFF specification is unclear on whether the end coordinate marks the last base-pair of the feature (closed intervals, `end_included=True`) or the one after (half-open intervals, `end_included=False`). The default, True, is correct for Ensembl GTF files. If in doubt, look for a CDS or stop_codon feature in you GFF file. Its length should be divisible by 3. If "end-start" is divisible by 3, you need `end_included=False`. If "end-start+1" is divisible by 3, you need `end_included=True`.

GFF_Reader will convert the coordinates from GFF standard (1-based, end maybe included) to HTSeq standard (0-base, end not included) by subtracting 1 from the start position, and, for `end_included=True`, also subtract 1 from the end position.

`metadata`

GFF_Reader skips all lines starting with a single '#' as this marks a comment. However, lines starying with '##' contain meta data (at least accoring to the Sanger Institute's version of the GFF standard.) Such meta data has the format `##key value`. When a metadata line is encountered, it is added to the `metadata` dictionary.

class `HTSeq.`**`GenomicFeature`** (*name*, *type_*, *interval*)

A GenomicFeature object always contains the following attributes:

`name`

A name of ID for the feature. As the GFF format does not have a dedicated field for this, the value of the first attribute in the *attributes* column is assumed to be the name of ID.

`type`

The type of the feature, i.e., a string like `"exon"` or `"gene"`. For GFF files, the 3rd column (*feature*) is taken as the type.

`interval`

The interval that the feature covers on the genome. For GFF files, this information is taken from the first (*seqname*), the forth (*start*), the fifth (*end*), and the seventh (*strand*) column.

When created by a *`GFF_Reader`* object, the following attributes are also present, with the information from the remaining GFF columns:

`source`

The 2nd column, denoted *source* in the specification, and intended to specify the data source.

`frame`

The 8th column (*frame*), giving the reading frame in case of a coding feature. Its value is an integer (0, 1, or 2), or the string `'.'` in case that a frame is not specified or would not make sense.

`score`

The 6th column (*score*), giving some numerical score for the feature. Its value is a float, or `'.'` in case that a score is not specified or would not make sense

attr
The last (9th) column of a GFF file contains *attributes*, i.e. a list of name/value pairs. These are transformed into a dict, such that, e.g., `gf.attr['gene_id']` gives the value of the attribute `gene_id` in the feature described by `GenomicFeature` object `gf`. The parser for the attribute field is reasonably flexible to deal with format variations (it was never clearly established whetehr name and value should be sperarated by a colon or an equal sign, and whether quotes need to be used) and also does a URL style decoding, as is often required.

In order to write a GFF file from a sequence of features, this method is provided:

get_gff_line(*with_equal_sign=False*)
Returns a line to describe the feature in the GFF format. This works even if the optional attributes given above are missing. Call this for each of your `GenomicFeature` objects and write the lines into a file to get a GFF file.

`HTSeq.`**parse_GFF_attribute_string**(*attrStr*, *extra_return_first_value=False*)
This is the function that *`GFF_Reader`* uses to parse the attribute column. (See *`GenomicFeature.attr`*.) It returns a dict, or, if requested, a pair of the dict and the first value.

CHAPTER 11

Other parsers

`VCF_Reader` and `VariantCall`

VCF is a text file format (most likely stored in a compressed manner). It contains meta-information lines, a header line, and then data lines each containing information about a position in the genome.

There is an option whether to contain genotype information on samples for each position or not.

See the definitions at

As usual, there is a parser class, called **VCF_Reader**, that can generate an iterator of objects describing the structural variant calls. These objects are of type `VariantCall` and each describes one line of a VCF file. See below for an example.

class `HTSeq.`**`VCF_Reader`**(*filename_or_sequence*)

As a subclass of `FileOrSequence`, VCF_Reader can be initialized either with a file name or with an open file or another sequence of lines.

When requesting an iterator, it generates objects of type `VariantCall`.

`metadata`

VCF_Reader skips all lines starting with a single '#' as this marks a comment. However, lines starying with '##' contain meta data (Information about filters, and the fields in the 'info'-column).

`parse_meta`(*header_filename = None*)

The VCF_Reader normally does not parse the meta-information and also the `VariantCall` does not contain unpacked metainformation. The function parse_meta reads the header information either from the attached `FileOrSequence` or from a file connection being opened to a provided 'header-filename'. This is important if you want to access sample-specific information for the :class'VariantCall's in your .vcf-file.

`make_info_dict`()

This function will parse the info string and create the attribute `infodict` which contains a dict with key:value-pairs containig the type-information for each entry of the `VariantCall`'s info field.

class `HTSeq.`**`VariantCall`**(*line*, *nsamples = 0*, *sampleids=[]*)

A VariantCall object always contains the following attributes:

`alt`
The alternative base(s) of the *`VariantCall`*. This is a list containing all called alternatives.

`chrom`
The Chromosome on which the *`VariantCall`* was called.

`filter`
This specifies if the *`VariantCall`* passed all the filters given in the .vcf-header (value=PASS) or contains a list of filters that failed (the filter-id's are specified in the header also).

`format`
Contains the format string specifying which per-sample information is stored in *`VariantCall.samples`*.

`id`
The id of the *`VariantCall`*, if it has been found in any database, for unknown variants this will be ".".

`info`
This will contain either the string version of the info field for this *`VariantCall`* or a dict with the parsed and processed info-string.

`pos`
A *`HTSeq.GenomicPosition`* that specifies the position of the *`VariantCall`*.

`qual`
The quality of the *`VariantCall`*.

`ref`
The reference base(s) of the *`VariantCall`*.

`samples`
A dict mapping sample-id's to subdicts which use the *`VariantCall.format`* as keys to store the per-sample information.

`unpack_info`(*infodict*)
This function parses the info-string and replaces it with a dict rperesentation if the infodict of the originating VCF_Reader is provided.

Example Workflow for reading the dbSNP in VCF-format (obtained from *dbSNP <ftp://ftp.ncbi.nih.gov/snp/organisms/human_9606/VCF/v4.0/00-All.vcf.gz>_*):

```
>>> vcfr = HTSeq.VCF_Reader( "00-All.vcf.gz" )
>>> vcfr.parse_meta()
>>> vcfr.make_info_dict()
>>> for vc in vcfr:
...     print vc,
1:10327:'T'->'C'
1:10433:'A'->'AC'
1:10439:'AC'->'A'
1:10440:'C'->'A'
```

FIXME The example above is not run, as the example file is still missing!

Wiggle Reader

The Wiggle format (file extension often `.wig`) is a format to describe numeric scores assigned to base-pair positions on a genome. The class *`WiggleReader`* is parser for such files.

class `HTSeq.`**`WiggleReader`**(*filename_or_sequence*, *verbose=True*)

The class is instatiated with the file name of a Wiggle file, or a sequence of lines in Wiggle format. A `WiggleReader` object generates an iterator, which yields pairs of the form `(iv, score)`, where `iv` is a *`GenomicInterval`* object and `score` is a `float` with the score that the file assigns to the specified interval. If `verbose` is set to True, the user is alerted to skipped lines (comments or `browser` lines) by a message printed to the standard output.

BED Reader

The BED format is a format originally used to describe gene models but is also commonly used to describe other genomic features.

class `HTSeq.`**`BED_Reader`**(*filename_or_sequence*)

The class is instatiated with the file name of a BED file, or a sequence of lines in BED format. A `BED_Reader` object generates an iterator, which yields a *`GenomicFeature`* object for each line in the BED file (except for lines starting with `track`, whcih are skipped).

The attributes of the yielded `GenomicFeature` objects are as follows:

`iv` a *`GenomicInterval`* object with the coordinates as given by the 1st, 2nd, 3rd, and 6th column of the BED file. If the BED file has less than 6 columns, the strand is set to "`.`".

`name` the name of feature as given in the 4th column, or `unnamed`, if the file has only three columns

`type` always the string `BED line`

`score` a float with the score as given by the 5th column (or `None` if the BED file has less 5 columns).

`thick` a *`GenomicInterval`* object containg the "thick" part of the feature, as specified by the 6th and 7th column, with chromosome and strand copied from `iv` (or `None` if the BED file has less 7 columns).

`itemRgb` a list of three `int` values, taken from the 8th column (`None` if the BED file has less 8 columns). In a BED file, this triple is meant to specify the colour in which the feature should be drawn in a browser.

CHAPTER 12

Miscellaneous

`FileOrSequence`

class `HTSeq.`**`FileOrSequence`**(*filename_or_sequence*)

This class is a a canvenience wrapper around a file.

The construcutor takes one argument, which may either be a string, which is interpreted as a file name (possibly with path), or a connection, by which we mean a text file opened for reading, or any other object that can provide an iterator over strings (lines of the file).

The advantage of passing a file name instead of an already opened file is that if an iterator is requested several times, the file will be re-opened each time. If the file is already open, its lines can be read only once, and then, the iterator stays exhausted.

Furthermore, if a file name is passed that end in ".gz" or ".gzip" (case insensitive), it is transparently gunzipped.

`fos`

The argument passed to the constructor, i.e., a filename or a sequence

`line_no`

The line number (1-based) of the most recently read line. Initially None.

`get_line_number_string``()`

Returns a string describing the position in the file. Useful for error messages.

Version

`HTSeq.`**`__version__`**

a string containing the current version

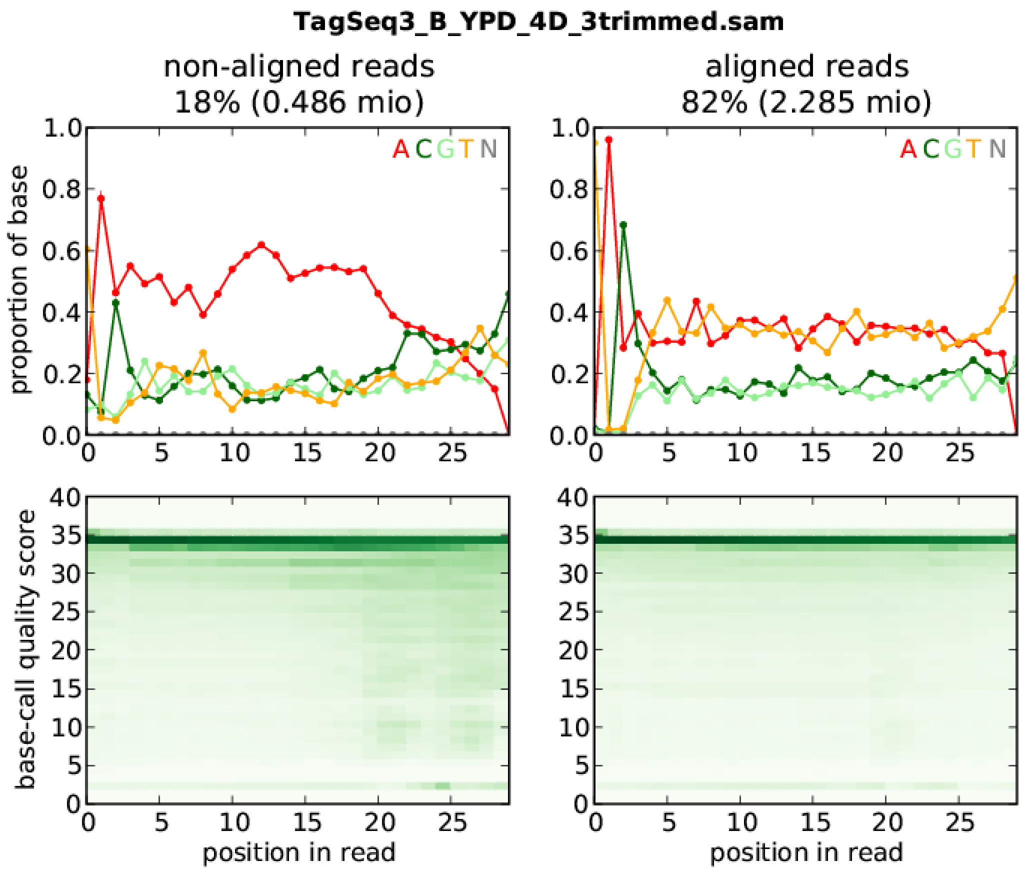
CHAPTER 13

Quality Assessment with `htseq-qa`

The Python script `htseq-qa` takes a file with sequencing reads (either raw or aligned reads) and produces a PDF file with useful plots to assess the technical quality of a run.

Plot

Here is a typical plot:



The plot is made from a SAM file, which contained aligned and unalignable reads. The left column is made from the non-aligned, the right column from the aligned reads. The header informs you about the name of the SAM file, and the number of reads.

The upper row shows how often which base was called for each position in the read. In this sample, the non-alignable reads have a clear excess in A. The aligned reads have a balance between complementing reads: A and C (reddish colours) have equal levels, and so do C and G (greenish colours). The sequences seem to be AT rich. Furthermore, nearly all aligned reads start with a T, followed by an A, and then, a C in 70% and an A in 30% of the reads. Such an imbalance would be reason for concern if it has no good explanation. Here, the reason is that the fragmentation of the sample was done by enzyme digestion.

The lower half shows the abundance of base-call quality scores at the different positions in the read. Nearly all aligned reads have a quality of 34 over their whole length, while for the non-aligned reads, some reads have lower quality scores towards their ends.

Usage

Note that `htseq-qa` needs matplotlib to produce the plot, so you need to install this module, as described here on the matplotlib web site.

After you have installed HTSeq (see *Prequisites and installation*) and matplotlib, you can run `htseq-qa` from the command line:

```
htseq-qa [options] read_file
```

If the file `htseq-qa` is not in your path, you can, alternatively, call the script with

```
python -m HTSeq.scripts.qa [options] read_file
```

The *read_file* is either a FASTQ file or a SAM file. For a SAM file, a plot with two columns is produced as above, for a FASTQ file, you get only one column.

The output is written into a file with the same name as *read_file*, with the suffix `.pdf` added. View it with a PDF viewer such as the Acrobat Reader.

Options

-t `<type>`, **--type**=`<type>`
The file type of the *read_file*. Supported values for <*type*> are:

- `sam`: a SAM file (Note that the SAMtools contain Perl scripts to convert most alignment formats to SAM)
- `solexa-export`: an `_export.txt` file as produced by the SolexaPipeline software after aligning with Eland (`htseq-qa` expects the new Solexa quality encoding as produced by version 1.3 or newer of the SolexaPipeline)
- `fastq`: a FASTQ file with standard (Sanger or Phred) quality encoding
- `solexa-fastq`: a FASTQ file with Solexa quality encoding, as produced by the SolexaPipeline after base-calling with Bustard (`htseq-qa` expects the new Solexa quality encoding as produced by version 1.3 or newer of the SolexaPipeline)

-o `<outfile>`, **--outfile**=`<outfile>`
output filename (default is <*read_file*>‘‘.pdf‘‘)

-r `<readlen>`, **--readlength**=`<readlen>`
the maximum read length (when not specified, the script guesses from the file

-g `<gamma>`, **--gamma**=`<gamma>`
the gamma factor for the contrast adjustment of the quality score plot

-n, --nosplit
do not split reads in unaligned and aligned ones, i.e., produce a one-column plot

-m, --maxqual
the maximum quality score that appears in the data (default: 40)

-h, --help
Show a usage summary and exit

CHAPTER 14

Counting reads in features with `htseq-count`

Given a file with aligned sequencing reads and a list of genomic features, a common task is to count how many reads map to each feature.

A feature is here an interval (i.e., a range of positions) on a chromosome or a union of such intervals.

In the case of RNA-Seq, the features are typically genes, where each gene is considered here as the union of all its exons. One may also consider each exon as a feature, e.g., in order to check for alternative splicing. For comparative ChIP-Seq, the features might be binding region from a pre-determined list.

Special care must be taken to decide how to deal with reads that align to or overlap with more than one feature. The `htseq-count` script allows to choose between three modes. Of course, if none of these fits your needs, you can write your own script with HTSeq. See the chapter *A tour through HTSeq* for a step-by-step guide on how to do so. See also the FAQ at the end, if the following explanation seems too technical.

The three overlap resolution modes of `htseq-count` work as follows. For each position *i* in the read, a set *S(i)* is defined as the set of all features overlapping position *i*. Then, consider the set *S*, which is (with *i* running through all position within the read or a read pair)

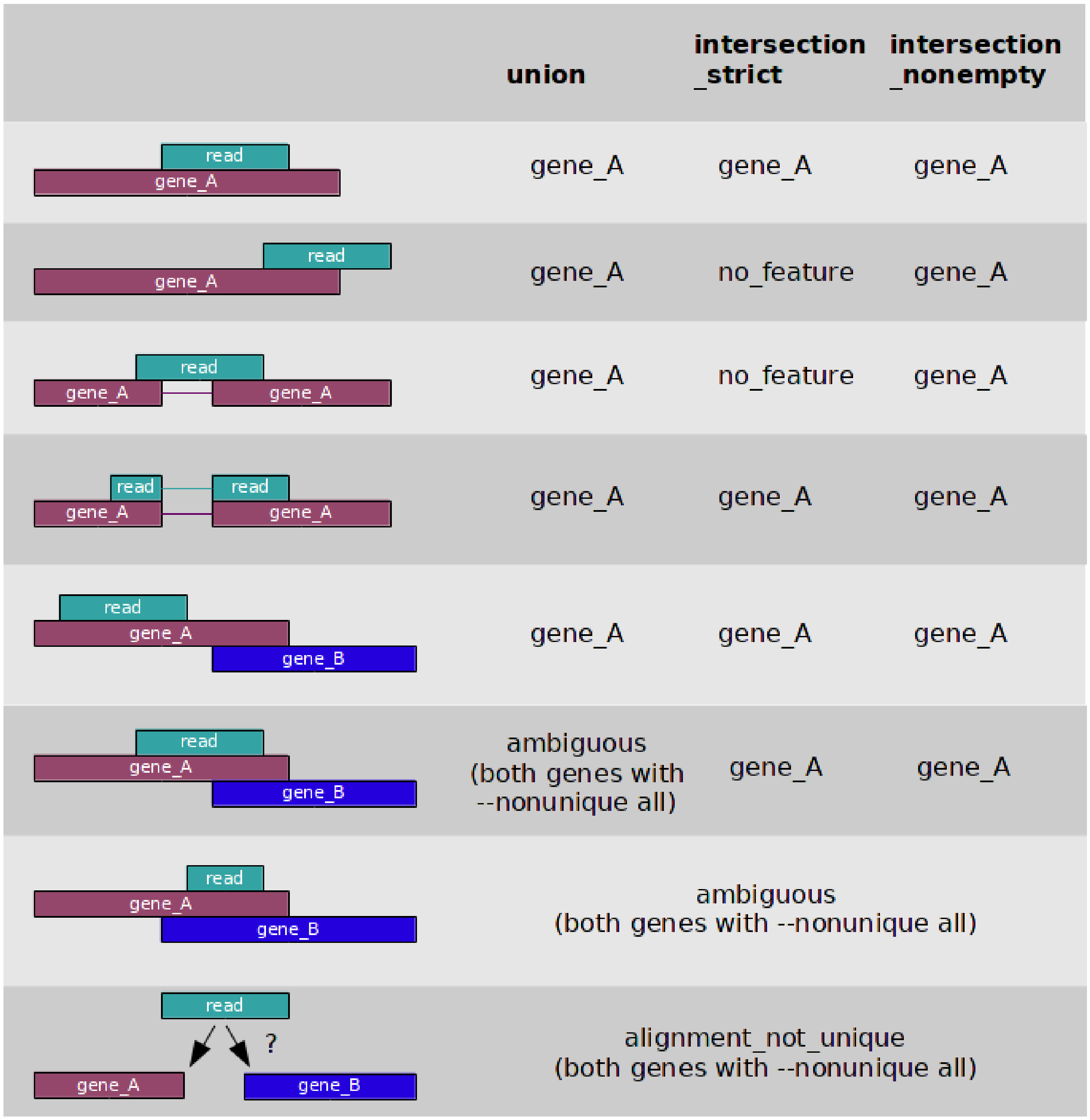
- the union of all the sets *S(i)* for mode `union`. This mode is recommended for most use cases.
- the intersection of all the sets *S(i)* for mode `intersection-strict`.
- the intersection of all non-empty sets *S(i)* for mode `intersection-nonempty`.

If *S* contains precisely one feature, the read (or read pair) is counted for this feature. If *S* is empty, the read (or read pair) is counted as `no_feature`. If *S* contains more than one feature, `htseq-count` behaves differently based on the `--nonunique` option:

- `--nonunique none` (default): the read (or read pair) is counted as `ambiguous` and not counted for any features. Also, if the read (or read pair) aligns to more than one location in the reference, it is scored as `alignment_not_unique`.
- `--nonunique all`: the read (or read pair) is counted as `ambiguous` and is also counted in all features to which it was assigned. Also, if the read (or read pair) aligns to more than one location in the reference, it is scored as `alignment_not_unique` and also separately for each location.

Notice that when using `--nonunique all` the sum of all counts will not be equal to the number of reads (or read pairs), because those with multiple alignments or overlaps get scored multiple times.

The following figure illustrates the effect of these three modes and the `--nonunique` option:



	union	intersection _strict	intersection _nonempty
	gene_A	gene_A	gene_A
	gene_A	no_feature	gene_A
	gene_A	no_feature	gene_A
	gene_A	gene_A	gene_A
	gene_A	gene_A	gene_A
	ambiguous (both genes with --nonunique all)	gene_A	gene_A
	ambiguous (both genes with --nonunique all)		
	alignment_not_unique (both genes with --nonunique all)		

Usage

After you have installed HTSeq (see *Prequisites and installation*), you can run `htseq-count` from the command line:

```
htseq-count [options] <alignment_files> <gff_file>
```

If the file `htseq-count` is not in your path, you can, alternatively, call the script with

```
python -m HTSeq.scripts.count [options] <alignment_files> <gff_file>
```

The `<alignment_files>` are one or more files containing the aligned reads in SAM format. (SAMtools contain Perl scripts to convert most alignment formats to SAM.) Make sure to use a splicing-aware aligner such as STAR. HTSeq-count makes full use of the information in the CIGAR field.

To read from standard input, use `-` as `<alignment_files>`.

If you have paired-end data, pay attention to the `-r` option described below.

The `<gff_file>` contains the features in the GFF format.

The script outputs a table with counts for each feature, followed by the special counters, which count reads that were not counted for any feature for various reasons. The names of the special counters all start with a double underscore, to facilitate filtering. (Note: The double unscore was absent up to version 0.5.4). The special counters are:

- `__no_feature`: reads (or read pairs) which could not be assigned to any feature (set *S* as described above was empty).
- `__ambiguous`: reads (or read pairs) which could have been assigned to more than one feature and hence were not counted for any of these, unless the `--nonunique all` option was used (set *S* had more than one element).
- `__too_low_aQual`: reads (or read pairs) which were skipped due to the `-a` option, see below
- `__not_aligned`: reads (or read pairs) in the SAM file without alignment
- `__alignment_not_unique`: reads (or read pairs) with more than one reported alignment. These reads are recognized from the `NH` optional SAM field tag. (If the aligner does not set this field, multiply aligned reads will be counted multiple times, unless they getv filtered out by due to the `-a` option.) Note that if the `--nonunique all` option was used, these reads (or read pairs) are still assigned to features.

Important: The default for strandedness is *yes*. If your RNA-Seq data has not been made with a strand-specific protocol, this causes half of the reads to be lost. Hence, make sure to set the option `--stranded=no` unless you have strand-specific data!

Options

-f `<format>`, **--format**=`<format>`

Format of the input data. Possible values are `sam` (for text SAM files) and `bam` (for binary BAM files). Default is `sam`.

-r `<order>`, **--order**=`<order>`

For paired-end data, the alignment have to be sorted either by read name or by alignment position. If your data is not sorted, use the `samtools sort` function of `samtools` to sort it. Use this option, with `name` or `pos` for `<order>` to indicate how the input data has been sorted. The default is `name`.

If `name` is indicated, `htseq-count` expects all the alignments for the reads of a given read pair to appear in adjacent records in the input data. For `pos`, this is not expected; rather, read alignments whose mate alignment have not yet been seen are kept in a buffer in memory until the mate is found. While, strictly speaking, the latter will also work with unsorted data, sorting ensures that most alignment mates appear close to each other in the data and hence the buffer is much less likely to overflow.

--max-reads-in-buffer=`<number>`

When <alignment_file> is paired end sorted by position, allow only so many reads to stay in memory until the mates are found (raising this number will use more memory). Has no effect for single end or paired end sorted by name. (default: `30000000`)

-s `<yes/no/reverse>`, **--stranded**=`<yes/no/reverse>`
whether the data is from a strand-specific assay (default: `yes`)

For `stranded=no`, a read is considered overlapping with a feature regardless of whether it is mapped to the same or the opposite strand as the feature. For `stranded=yes` and single-end reads, the read has to be mapped to the same strand as the feature. For paired-end reads, the first read has to be on the same strand and the second read on the opposite strand. For `stranded=reverse`, these rules are reversed.

-a `<minaqual>`, **--a**=`<minaqual>`
skip all reads with alignment quality lower than the given minimum value (default: 10 — Note: the default used to be 0 until version 0.5.4.)

-t `<feature type>`, **--type**=`<feature type>`
feature type (3rd column in GFF file) to be used, all features of other type are ignored (default, suitable for RNA-Seq analysis using an Ensembl GTF file: `exon`)

-i `<id attribute>`, **--idattr**=`<id attribute>`
GFF attribute to be used as feature ID. Several GFF lines with the same feature ID will be considered as parts of the same feature. The feature ID is used to identity the counts in the output table. The default, suitable for RNA-Seq analysis using an Ensembl GTF file, is `gene_id`.

--additional-attr=`<id attributes>`
Additional feature attributes, which will be printed as an additional column after the primary attribute column but before the counts column(s). The default is none, a suitable value to get gene names using an Ensembl GTF file is `gene_name`.

-m `<mode>`, **--mode**=`<mode>`
Mode to handle reads overlapping more than one feature. Possible values for <*mode*> are `union`, `intersection-strict` and `intersection-nonempty` (default: `union`)

--nonunique=`<nonunique mode>`
Mode to handle reads that align to or are assigned to more than one feature in the overlap <*mode*> of choice (see -m option). <*nonunique mode*> are `none` and `all` (default: `none`)

-o `<samout>`, **--samout**=`<samout>`
write out all SAM alignment records into an output SAM file called <samout>, annotating each line with its assignment to a feature or a special counter (as an optional field with tag 'XF')

-q, --quiet
suppress progress report and warnings

-h, --help
Show a usage summary and exit

Frequenctly asked questions

My shell reports "command not found" when I try to run "htseq-count". How can I launch the script? The file "htseq-count" has to be in the system's search path. By default, Python places it in its script directory, which you have to add to your search path. A maybe easier alternative is to write `python -m HTSeq.scripts.count` instead of `htseq-count`, followed by the options and arguments, which will launch the htseq-count script as well.

Why are multi-mapping reads and reads overlapping multiple features discarded rather than counted for each feature?
The primary intended use case for `htseq-count` is *differential* expression analysis, where one compares the expression of the same gene across samples and not the expression of different genes within a sample. Now, consider two genes, which share a stretch of common sequence such that for a read mapping to this stretch, the aligner cannot decide which of the two genes the read originated from and hence reports a multiple alignment. If we discard all such reads, we undercount the total output of the genes, but the *ratio* of expression strength (the "fold change") between samples or experimental condition will still be correct, because we discard the

same fratcion of reads in all samples. On the other hand, if we counted these reads for both genes, a subsequent diffential-expression analysis might find false positives: Even if only one of the gene changes increases its expression in reaction to treatment, the additional read caused by this would be counted for both genes, giving the wrong appearance that both genes reacted to the treatment.

I have used a GTF file generated by the Table Browser function of the UCSC Genome Browser, and most reads are counted as ambi In these files, the `gene_id` attribute incorrectly contains the same value as the `transcript_id` attribute and hence a different value for each transcript of the same gene. Hence, if a read maps to an exon shared by several transcripts of the same gene, this will appear to `htseq-count` as and overlap with several genes. Therefore, these GTF files cannot be used as is. Either correct the incorrect `gene_id` attributes with a suitable script, or use a GTF file from a different source.

Can I use htseq-count to count reads mapping to transcripts rather than genes? In principle, you could instruct htseq-count to count for each of a gene's transcript individually, by specifying `--idattr transcript_id`. However, all reads mapping to exons shared by several transcripts will then be considered ambiguous. (See second question.) Counting them for each transcript that contains the exons would be possible but makes little sense for typical use cases. (See first question.) If you want to perform differential expression analysis on the level of individual transcripts, maybe ahve a look at our paper on DEXSeq for a discussion on why we prefer performing such analyses on the level of exons instead.

For paired-end data, does htseq-count count reads or read pairs? Read pairs. The script is designed to count "units of evidence" for gene expression. If both mates map to the same gene, this still only shows that one cDNA fragment originated from that gene. Hence, it should be counted only once.

What happens if the two reads in a pair overlap two different features? The same as if one read overlaps two features: The read or read pair is counted as ambiguous.

What happend if the mate of an aligned read is not aligned? For the default mode "union", only the aligned read determines how the read pair is counted. For the other modes, see their description.

Most of my RNA-Seq reads are counted as ``__no_feature``. What could have gone wrong? Common causes include: - The `--stranded` option was set wrongly. Use a genome browser (e.g., IGV) to check. - The GTF file uses coordinates from another reference assembly as the SAM file. - The chromosome names differ between GTF and SAM file (e.g., `chr1` in one file and jsut `1` in the other).

Which overlap mode should I use? When I wrote `htseq-count`, I was not sure which option is best and included three possibilities. Now, several years later, I have seen very few cases where the default `union` would not be appropriate and hence tend to recommend to just stick to `union`.

I have a GTF file? How do I convert it to GFF? No need to do that, because GTF is a tightening of the GFF format. Hence, all GTF files are GFF files, too. By default, htseq-count expects a GTF file.

I have a GFF file, not a GTF file. How can I use it to count RNA-Seq reads? The GTF format specifies, inter alia, that exons are marked by the word `exon` in the third column and that the gene ID is given in an attribute named `gene_id`, and htseq-count expects these words to be used by default. If you GFF file uses a word other than `exon` in its third column to mark lines describing exons, notify `htseq-count` using the `--type` option. If the name of the attribute containing the gene ID for exon lines is not `gene_id`, use the `--idattr`. Often, its is, for example, `Parent`, `GeneID` or `ID`. Make sure it is the gene ID and not the exon ID.

How can I count overlaps with features other than genes/exons? If you have GFF file listing your features, use it together with the `--type` and `--idattr` options. If your feature intervals need to be computed, you are probably better off writing your own counting script (provided you have some knowledge of Python). Follow the tutorial in the other pages of this documentation to see how to use HTSeq for this.

How should I cite htseq-count in a publication? Please cite HTSeq as follows: S Anders, T P Pyl, W Huber: *HTSeq — A Python framework to work with high-throughput sequencing data*. bioRxiv 2014. doi: 10.1101/002824. (This is a preprint currently under review. We will replace this with the reference to the final published version once available.)

CHAPTER 15

Version history

Version 0.8.0

2017-06-07

This release adds a few options to `htseq-count`:

- `--nonunique` handles non-uniquely mapped reads
- `--additional-attr` adds an optional column to the output (typically for human-readable gene names)
- `--max-reads-in-buffer` allows increasing the buffer size when working with paired end, coordinate sorted files

Moreover, `htseq-count` can now take more than one input file and prints the output with one column per input file.

Finally, parts of the code have been streamlined or modernized, documentation has been moved to readthedocs, and other minor changes.

Version 0.7.2

2017-03-24

This release effectively merges the Python2 and Python3 branches.

Enhancements:

- `pip install HTSeq` works for both Python 2.7 and 3.4+

Version 0.7.1

2017-03-16

Enhancements:

- installs from PyPI

Version 0.7.0

2017-02-07

Enhancements:

- understands SAMtools optional field B (used sometimes in STAR aligner)
- write fasta files in a single line
- better docstrings thanks to SWIG 3

Bugfixes:

- fixed tests and docs in .rst files

Support bumps:

- supports pysam >=0.9.0

New maintainer: Fabio Zanini.

Version 0.6.1

2014-02-27

- added parser classes for BED and Wiggle format

Patch versions:

- 0.6.1p1 (2014-04-13)
  - Fixed incorrect version tag
- 0.6.1p2 (2014-08-09)
  - some improvements to documentation

Version 0.6.0

2014-02-26

- Several changes and improvements to htseq-count:
  - BAM files can now be read natively. (New option `--format`)
  - Paired-end SAM files can be used also if sorted by position. No need any mroe to sort by name. (New option `--order`.)
  - Documentation extended by a FAQ section.
  - Default for `--minaqual` is now 10. (was: 0)
- New chapter in documentation, with more information on counting reads.
- New function `pair_SAM_alignments_with_buffer` to implement pairing for position-sorted SAM files.

Version 0.5.4

2013-02-20

Various bugs fixed, including

- GFF_Reader interpreted the constructor's "end_included" flag in the wrong way, hence the end position of intervals of GFF features was off by 1 base pair before
- htseq-count no longer warns about missing chromosomes, as this warning was often misleading. Also, these reads are no properly included in the "no_feature" count.
- default for "max_qual" in "htseq-qa" is now 41, to accommodate newer Illumina FASTQ files
- BAM_Reader used to incorrectly label single-end reads as paired-end

Patch versions:

- v0.5.4p1 (2013-02-22):
  - changed default for GFF_Reader to end_included=True, which is actually the correct style for Ensemble GTF files. Now the behavious should be as it was before.
- v0.5.4p2 (2013-04-18):
  - fixed issue blocking proper built on Windows
- v0.5.4p3 (2013-04-29):
  - htseq-count now correctly skips over "M0" cigar operations
- v0.5.4p4 (2013-08-28):
  - added `.get_original_line()` function to `VariantCall`
  - firex a bug with reads not being read as paired if they were not flagged as proper pair
- v0.5.4p5 (2013-10-02/2013-10-10):
  - parsing of GFF attribute field no longer fails on quoted semicolons
  - fixed issue with get_line_number_string

Version 0.5.3

2011-06-29

- added the '–stranded=reverse' option to htseq-count

Patch versions:

- v0.5.3p1 (2011-07-15):
  - fix a bug in pair_sam_Alignment (many thanks for Justin Powell for finding the bug and suggesting a patch)
- v0.5.3p2 (2011-09-15)
  - fixed a bug (and a documentation bug) in trim_left/right_end_with_quals
- v0.5.3p3 (2011-09-15)
  - p2 was built improperly
- v0.5.3p5 (2012-05-29)

  - added 'to_line' function to VariantCall objects and 'meta_info' function to VCF_Reader objects to print VCF-lines / -headers respectively
- v0.5.3p5b (2012-06-01) - added 'flag' field to SAM_Alignment objects and fixed 'get_sam_line' function of those
- v0.5.3p6 (2012-06-11) - fixed mix-up between patches p3, p4 and p5
- v0.5.3p7 (2012-06-13) - switched global pysam import to on-demand version
- v0.5.3p9ur1 (2012-08-31) - corrected get_sam_line: tab isntead of space between optional fields

Version 0.5.2

2011-06-24

- added the '–maxqual' option to htseq-qa

Version 0.5.1

2011-05-03

- added steps method to GenomicArray

Patch versions:

- v0.5.1p1 (2011-05-11):
  - fixed a bug in step_vector.h causing linkage failure under GCC 4.2
- v0.5.1p2 (2011-05-12):
  - fixed pickling
- v0.5.1p3 (2011-05-22):
  - fixed quality plot in htseq-qa (top pixel row, for quality score 40, was cut off)

Version 0.5.0

2011-04-21

- refactoring of GenomicArray class:
  - field `step_vectors` replaced with `chrom_vector`. These now contain dicts of dicts of `ChromVector` objects rather than `StepVector` ones.
  - `chrom_vectors` is now always a dict of dict, even for unstranded GenomicArrays to make it easier to loop over them. (The inner dict has either keys `"+"` and `"-"`, or just one key, `"."`.)
  - The new `ChromVector` class wraps the actual vector and supports three different storage modes: `step`, `ndarray` and `memmap`, the latter two being numpy arrays, without and with memory mapping.
  - The `GenomicArray` constructor now take two new arguments, one for the storage class, one for the memmap directory (if needed).
  - The `add_value` methods had been replaced with an `__iadd__` method, to enable the += semantics.
  - Similarily, += for `GenomicArrayOfSets` adds an element to the sets.

  - Instead of `get_steps`, now use `steps`.
- new parser class `VCF_Reader` and record class `VariantCall`
- new parser class `BAM_Reader`, to add BAM support (including indexed random access) (requires PySam to be installed)
- new documentation page *A detailed use case: TSS plots*
- `Fasta_Reader` now allows indexed access to Fasta files (requires Pysam to be installed)
- peek function removed

Patch Versions:

- v0.5.0p1 (2011-04-22):
  - build was incomplete; fixed
- v0.5.0p2 (2011-04-22):
  - build was still faulty; new try
- v0.5.0p3 (2011-04-26)
  - fixed regression bug in htseq-count

Version 0.4.7

2010-12-22

- added new option `-o` (or `--samout`) to htseq-count

Patch versions:

- Version 0.4.7p1 (2011-02-14)
  - bug fix: GFF files with empty attribute fiels are now read correctly
- Version 0.4.7p2 (2011-03-13)
  - fixed assertion error in pair_SAM_alignment, triggered by incorrect flags
- Version 0.4.7p3 (2011-03-15)
  - fixed problem due to SAM_Alignment.peek (by removing the method)
- Version 0.4.7p4 (2011-03-18)
  - removed left-over debugging print statement

Version 0.4.6

2010-12-09

- pair_SAM_alignments now handles multiple matches properly
- SAM_Alignments now allows access to optional fields via the new methods optional_field and optional_fields
- htseq-count now skips reads that are non-uniquely mapped according to the ‘NH’ optional field
- updated documentation

Patch versions:

- Version 0.4.6p1 (2010-12-17)
  - updated htseq-count documentation page
  - htseq-count now accepts '-' as SAM file name
- Version 0.4.6p2 (2012-12-21)
  - corrected a bug in htseq-count regarding the handling of warnings and added SAM_Reader.peek.

Version 0.4.5

2010-08-30

- correction to GenomicArray.get_steps() when called without arguments
- correction to FileOrSequence.get_line_number_string
- removed use of urllib's quote and unquote in GFF parsing/writing
- GFF_Reader now stores "meta information"
- qa.py now gives progress report
- auto add chrom now also works on read access
- refactored CIGAR parser
- added bool fields to SAM_Alignment for all flag bits

Patch versions:

- Version 0.4.5p1 (2010-10-08)
  - correction of a mistake in CIGAR checking, misreading symbol "N"
- Version 0.4.5p2 (2010-10-13)
  - Sequence.add_bases_to_count_array and hence htseq-qa now accepts '.' instead of 'N' in a fastq file
- Version 0.4.5p3 (2010-10-20)
  - fixed error reporting for PE in htseq-count
- Version 0.4.5p4 (2010-10-21)
  - fixed another error reporting for PE in htseq-count
- Version 0.4.5p5 (2010-10-28)
  - Not only 'N' but also 'S' was read the wrong way. Fixed.
  - Cython had some odd way handling properties overloading attributes, which caused issues with 'Alignment.read'. Worked around.
- Version 0.4.5p6 (2010-11-02)
  - write_to_fastq should not break lines. Fixed.
- Version 0.4.5p7 (2010-11-16)
  - added fallback to distutils in case setuptools in unavailable
  - fixed documentation of '-a' option to htseq-count

Version 0.4.4

2010-05-19

- StepVectors (and hence also GenomicArrays) now notice if, when setting the value of a step, this value is equal to an adjacent step and merge the steps.
- GenomicArray's constructor now allows the special value `"auto"` for its first arguments in order to start without chromosomes and automatically add them when first encountered.

Patch versions:

- Version 0.4.4p1 (2010-05-26):
  - minor change to make it run on Python 2.5 again
  - changed 'str' to 'bytes' at various places, now compiles with Cython 0.12 (but no longer with Cython 0.11 and Python 2.5)
- Version 0.4.4p2 (2010-06-05):
  - change to SAM parser: if flag "query unmapped is set" but RNAME is not "*", a warning (rather than an error) is issued
- Version 0.4.4p3 (2010-06-25)
  - again removed an "except sth as e"
- Version 0.4.4p4 (2010-07-12)
  - dto.
- Version 0.4.4p5 (2010-07-13)
  - rebuilt with Cython 0.12.1 (previous one was accidently built with Cython 0.11.1, causing it to fail with Python 2.5)
- Version 0.4.4p6 (2010-07-21)
  - fixed bug in error reporting in count.py
  - losened GFF attribute parsing
  - changed "mio" to "millions" in qa output
  - improved error reporting in GFF parser
  - made SAM parsing more tolerant

Version 0.4.3

2010-05-01

New argument to constructer of GFF_Reader: `end_include`

- Version 0.4.3-p1 (2010-05-04): version number was messed up; fixed
- Version 0.4.3-p2 (2010-05-15): fixed '-q' option in htseq-count
- Version 0.4.3-p3 (2010-05-15): parse_GFF_attribute_string can now deal with empty fields; score treated as float, not int

- Version 0.4.3-p3 (2010-05-15): - parse_GFF_attribute_string can now deal with empty fields; score treated as float, not int - fixed bug in SAM_Reader: can now deal with SAM files with 11 columns - SAM_Alignment._tags is now a list of strings
- Version 0.4.3-p4 (2010-05-16): bumped version number again just to make sure

Version 0.4.2

2010-04-19

Bug fixes to htseq-count and pair_SAM_alignments. Bumped version number to avoid confusion.

- Version 0.4.2-p1 (2010-04-20): there was still a bug left in htseq-count, fixed.
- Version 0.4.2-p2 (2010-04-26): bug fix: adapter trimming failed if the adapter was completely included in the sequence
- Version 0.4.2-p3
- Version 0.4.2-p4 (2010-04-29): bug fix: error in warning when htseq-count encountered an unknown chromosome
- Version 0.4.2-p5 (2010-04-30): bug fixes: error in warning when PE positions are mismatched, and misleading error when calling get_steps with unstranded interval in a stranded array

Version 0.4.1

2010-04-19

Bug fixes:

- Fixed bug in `htseq-count`: CIGAR strings with gaps were not correctly handled
- Fixed bug in Tour (last section, on counting): An wrong indent, and accidental change to the `exons` variable invalidated data.
- SolexaExportReader no longer complains about multiplexing (indexing) not being supported.
- Mention link to example data in Tour.
- Fix installation instructions. (`--user` does not work for Python 2.5.)

Enhancements:

- Paired-end support for SAM_Alignment.
- "_as_pos" attributes for GenomicInterval

Version 0.4.0

2010-04-07

First "official" release, i.e., uploaded to PyPI and announced at SeqAnswers

Version 0.3.7

2010-03-12

First version that was uploaded to PyPI

CHAPTER 16

Notes for Contributors

If you intend to contribute to the development of HTSeq, these notes will help you to get started.

Source code

The source code is on Github. To check out the repository, use

```
git clone https://github.com/simon-anders/htseq.git
```

Languages

HTSeq is mostly written in Python and is compatible with both Python 2.7 and Python 3.4 and above. However, the codebases for Python 2/3 are separate and development happens mainly on the Python 3 branch.

A good part of HTSeq is actually not written in Python but in Cython. In case you don't know it yet: Cython, a fork from Pyrex, is a kind of Python compiler. You annotate Python code with additional type informations (the lines starting with `cdef` in the source code). Cython will then transform the Cython source file (with extension `pyx`) into a C file, which calls the appropriate funnctions of Python's C API. Without type annotation, this looks and feels the same as normal Python and is not really faster, either. With type annotation, significant performance gains are possible, especially in inner loops.

A small part, namely the StepVector class, is written in C++ and exported with SWIG. (SWIG, the "Simple Wrapper and Interface Generator" is a very useful tool to generate C/C++ code to wrap an existing C/C++ library such that it becomes accessible as a native library within a number of scripting languages.) I am not so happy with this any more (the abstraction panelty of the object-oriented SWIG wrapping turned out to be a bit high) and ultimatively want to rewrite this part.

Build process

HTSeq follows the standard python packaging guidelines and relies on a `setup.py` script that is simultaneously compatible withPython 2 and 3. To build the code, run:

```
python setup.py build
```

and to install:

```
python setup.py install
```

If you are not modifying the low-level C/C++/Cython interfaces, you can do without Cython and SWIG. This is how users normally install HTSeq using `pip`. If you do modify those files, the `setup.py` has a preprocessing step that calls Cython and/or SWIG if these programs are found. You set the `SWIG` and `CYTHON` environment variables to point to your executables if you have special requirements.

To test during development, HTSeq relies on Continuous Integration (CI), at the moment Travis CI is set up.

To build the documentation, Sphinx was used. Just go into the appropriate `doc` folder and call:

```
make html
```

to regenerate the documentation. Docs are stored on readthedocs.

Distributing

To wrap up a package, call:

```
python setup.py sdist
```

This makes a directory `dists` and in there, a tarball with all the source files (Python and C/C++). If you are a maintainer of HTSeq, you can upload this file onto PyPI on the testing server. Then, you should run the Tracis CI tests that try to install HTSeq directly from PyPI (without the source code). If all goes well, you can upload the tar file onto the live PyPI server.

Files

The package contains source files for Python 2 and 3 in separate folders. Within each of those folders, the following files are found:

`HTSeq/__init__.py`: The outer face of HTSeq. This file defines the name space of HTSeq and contains the definition of all classes without performance-critical methods. The file imports `_HTSeq` in its own namespace, so that, for the user, it does not matter whether an object is defined here or in `_HTSeq.pyx`.

`src/HTSeq/_HTSeq.pyx`: The core of HTSeq. All classes with perfomance-critical methods are defined here. For most of it, this file looks as a normal Python file. Only where performance is critical, type annotation has been added. See the Cython manual for details.

`src/HTSeq/_HTSeq.pxd`: The “header file” for `_HTSeq.pyx`. It contains the type annotation for all the fields of the classes defined in `_HTSeq.pyx`. If a user would want to write her own Cython code, she could use Cython’s `cimport` directive to import this header file and so make Cython aware of the typed definitions of fields and methods in `_HTSeq.pyx`, which may improve performance because it allows Cython to kick out all unnecessary type checking.

`HTSeq/_HTSeq_internal.py`: There are a few limitation to the standard Python code allowed in Cython files; most importantly, the `yield` statement is not yet supported. Hence, `_HTSeq.pyx` imports this file, and whenever a method in `_HTSeq.pyx` needs a `yield`, it calls a function which is put in here.

`src/step_vector.h`: The C++ `step_vector` class template. As this is a pure template, there is no `step_vector.cc` file with definitions. If you want to use a `step_vector` in a C++ project, this is all you need.

`src/StepVector.i`: An input file to SWIG, which produces the Python wrapper around `step_vector.h`, i.e., the `StepVector` module containing the `StepVector` class. Note that this file contains not only SWIG directives but also Python and come C++ code.

`src/AutoPyObjPtr.i`: A very small SWIG library that allows SWIG-wrapped C++ container classes to store Python objects in a way that Python's garbage collector is happy with.

`HTSeq/scripts/count.py` and `HTSeq/scripts/qa.py`: The source code for the stand-alone scripts `htseq-count` and `htseq-qa`. They reside in the sub-package `HTSeq.scripts`, allowing to call the scripts with, e.g., `python -m HTSeq.scripts.qa`.

`scripts/htseq-count` and `scripts/htseq-qa`: Short stubs to call the scripts from the command line simply as, e.g., `htseq-qa`.

`doc/`: this documentation, in Sphinx reStructuredText format, and a Makefile to drive Sphinx.

`test/test.py` Performs all the deoctests in the documentation, using the example data in the `example_data` directory.

Furthermore, there are these files to support development:

`setup.py`: A typical setuptools setup.py file.

Finally, there are these files

`VERSION`: a one-line text-fil with the version number. It is read by `setup.py`, used by `build_it` to generate the one-line Python file `HTSeq/_version.py` and also used when building the documentation.

`MANIFEST.in`: Brings some files to the attention of `setup.py sdist` which would otherwise not be included

`LICENCE`: The GPL, v3

`README.md`: Points the user to the web site.

and these directories

`example_files/`: a few example files to be use by the doctests in the documentation.

- genindex

Index