HTSeq – A Python framework to work with high-throughput sequencing data

Simon Anders*, Paul Theodor Pyl and Wolfgang Huber

Genome Biology Unit, European Molecular Biology Laboratory, 69111 Heidelberg, Germany

* e-mail: sanders@fs.tum.de

Draft of 2014-02-19

ABSTRACT

A large choice of tools exists for many standard tasks in the analysis of high-throughput sequencing (HTS) data. However, once a project deviates from standard work flows, custom scripts are needed. We present HTSeq, a Python library to facilitate the rapid development of such scripts. HTSeq offers parsers for many common data formats in HTS projects, as well as classes to represent data such as genomic coordinates, sequences, sequencing reads, alignments, gene model information, variant calls and more, and provides data structures that allow for querying via genomic coordinates. We also present htseq-count, a tool developed with HTSeq that preprocesses RNA-Seq data for differential expression analysis by counting the overlap of reads with genes.

1 INTRODUCTION

The rapid technological advance in high-throughput sequencing (HTS) has led to the development of many new kinds of assays, each of which requires the development of a suitable bioinformatical analysis pipeline. For the recurring "big tasks" in a typical pipeline, such as alignment and assembly, the bioinformatics practitioner can chose from a range of published tools. For more specialised tasks, and in order to interface between existing tools, customised scripts often need to be written.

For instance, before reads are aligned, various preprocessing steps may need to be performed on the raw reads, such as trimming low-quality ends, removing adapter sequences or demultiplexing. Once reads have been aligned, downstream analysis typically attempts to relate the alignment to feature metadata such as gene models.

In the case of RNA-Seq data a typical downstream analysis is calculating the number of alignments overlapping certain features (gene, exons, etc.). An example for ChIP-Seq data would be the generation of aggregate coverage profiles relative to transcription start sites or other features of interest.

While collections of scripts, and some specialised tools with graphical user interfaces, are available for some of these recurring smaller tasks, users are often restricted by the limited scope and flexibility of such monolithic tools. Only the most common and widely used analysis work flows can be performed without the need to write custom scripts for some of the tasks.

Here we present HTSeq, a Python library to facilitate the rapid development of scripts for processing and analysis of high-throughput sequencing (HTS) data. HTSeq includes parsers for common file formats for a variety of types of input data and offers a general platform for a diverse range of tasks. A core component of HTSeq is a container class that simplifies working with data associated with genomic coordinates, i.e., values attributed to genomic positions (e.g., read coverage) or to genomic intervals (e.g., genomic features such as exons or genes). Two stand-alone applications developed with HTSeq are distributed with the package, namely htseq-qa for read quality assessment and htseq-count for preprocessing RNA-Seq alignments for differential expression calling.

Most of the features described in the following sections have been available since the initial release of the HTSeq package in 2010. Since then, the package and especially the htseq-count script have found considerable use in the research community. The present article provides a description of the package and also reports on recent improvements.

HTSeq comes with extensive documentation, including a tutorial that demonstrates the use of the core classes of HTSeq and discusses several important use cases in detail. The documentation, as well as HTSeq's design, is geared towards allowing users with only moderate Python knowledge to create their own scripts, while shielding more advanced internals (such as the use of Cython, Behnel *et al.* (2011), to enhance performance) from the user.

HTSeq is released as open-source software under the GNU General Public Licence and available from `http://www-huber.embl.de/HTSeq` or from the Python Package Index `https://pypi.python.org/pypi/HTSeq`.

2 COMPONENTS OF HTSEQ

HTSeq is designed as an object-oriented framework that uses classes to encapsulate data and provide services. Its classes fall into four categories: Parsers, data records, auxiliary classes and containers.

2.1 Parser and record objects

Currently, HTSeq provides parsers for reference sequences (FASTA), short reads (FASTQ), short-read alignments (the SAM/BAM format and some legacy formats), feature, annotation and score data (GFF/GTF, VCF, BED, Wiggle).

HTSeq provides a parser for each of the supported file types and a record class for each type of data. Parser objects are tied to files and act as iterator generators which can be used e.g. in the head of a `for` loop, to populate the loop variable for each iteration with an instance of the appropriate record class. For instance, a *FastqParser* object is connected to a FASTQ file and when used as an iterator, it generates objects of type *SequenceWithQuality* that represent the

records in the FASTQ file and provide the data in slots for the read name, read sequence and quality string.

A core idea is that the loop body does not need to know about specifics of the data source because the parser enforces clearly defined conventions. For example, it is sufficient to inform the *FastqParser* object on construction which of the various encoding standards for the base-call quality scores the file uses. The parser will decode and convert the scores as necessary, and any downstream code may safely assume that the integer array representing the quality scores is always in the standard Sanger/Phred scale.

Another example of conflicting conventions concerns genomic coordinates. Some formats index the first base pair of a chromosome with zero (e.g., the BED format), others with one (e.g., the GFF format). In some formats, an interval is meant to include both start and end positions, in other formats, the base pair under the "end" coordinate is not part of the interval. HTSeq sticks to one convention, namely to start indexing sequences with zero and not to include endpoints (as is standard in Python). Parser classes adjust coordinates where needed, and hence, whenever HTSeq objects present genomic coordinates in the HTSeq classes for this purpose, *GenomicInterval* and *GenomicPosition*, they are guaranteed to follow this convention. When writing a position or interval into a file, the conversion is reversed to follow the convention of the output file format.

Typically, files are processed sequentially; however, support of random access for FASTA and BAM files is provided, too. For this feature, HTSeq builds on PySam (the Python bindings to the samtools API; Heger *et al.*) to leverage the indexing functionality provided by samtools (Li *et al.*, 2009).

Note that in all these cases the operating system's buffering and caching of disk accesses works to our advantage. Even though a typical HTSeq script may look like it accesses its input data files in every loop, most loop iterations will get their data from memory, so that the streaming fashion of data access carries no performance penalty.

2.2 The GenomicArray class

Data in genomics analyses is often associated with positions on a genome, i.e., coordinates in a reference assembly. One example for such data is read coverage: for each base pair or nucleotide of the reference genome, the number of read alignments overlapping that position are stored. Similarly, gene models and other genomic annotation data can be represented as objects describing features such as exons that are associated with genomic intervals, i.e., coordinate ranges in the reference.

A core component of HTSeq is the class *GenomicArray*, which is a container to store any kind of genomic-position dependent data. Each base-pair position on the genome can be associated with a value that can be efficiently stored and retrieved given the position.

Use cases for the *GenomicArray* class fall into two distinct categories: representing data and representing metadata. Coverage vectors are an example for the former: To generate them, one initializes a *GenomicArray* object with zeroes and iterates through a list of alignments, adding one to all positions covered by each of them. The tutorial in the online documentation gives detailed explanations and code examples for this and other use cases.

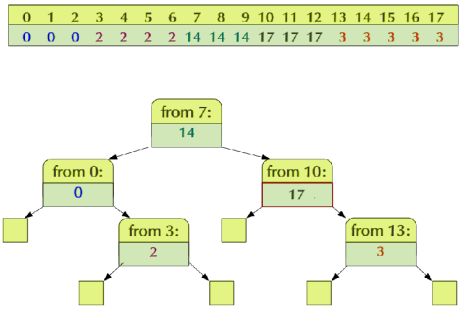


Fig. 1. Efficient storage of piecewise constant data: We wish to store data that conceptually form a one-dimensional array as depicted on top. The values are constant for certain index ranges, which we call steps, and we wish to store for each step just its starting index and its value. A balanced binary tree (bottom) allows for efficiently retrieval of the value given a position index.

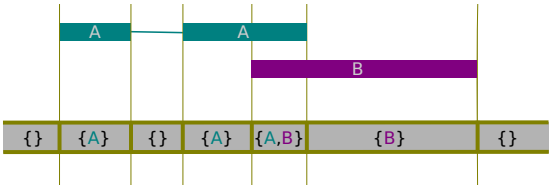


Fig. 2. Using the class *GenomicArrayOfSets* to represent overlapping annotation metadata. The indicated features are assigned to the array, which then represents them internally as steps, each step having as value a set whose elements are references to the features overlapping the step.

Especially for large genomes and a moderate number of reads the resulting vector will have steps, i.e., stretches of constant values. To store these efficiently, we represent each chromosome (or, alternatively, each strand of the chromosome) with an object of class *StepVector*. Such an object stores steps, i.e., intervals associated with a value, in a way that offers fast access while substantially reducing memory requirements. Internally this is done by using the `map` template of the C++ standard template library (Josuttis, 1999) (exposed to Python via SWIG; Beazley *et al.* (1996)), which is typically implemented as a red-black tree (Sedgewick, 1988). See Figure 1 for an example.

On the other hand, for large numbers of reads, the steps may become small and this storage strategy will be less efficient than a simple dense array. Hence, the user can alternatively ask the *GenomicArray* object to use standard NumPy arrays to represent the chromosomes. (NumPy (Oliphant *et al.*) is a widely used Python extension providing dense array functionality and linear algebra routines.) For larger genomes, a computer's RAM may be insufficient to keep the data for all chromosomes in memory simultaneously, and in that case the *memmap* feature of NumPy may be used, which transparently swaps currently unused parts of the data out to disk. As before, the choice of storage back-end is transparent, i.e., if the user changes it, no changes need to be made in the code using the *GenomicArray* objects.

Similar to NumPy arrays, a *GenomicArray* has a data type, which may either be a simple scalar type such as `int` or `float`, or `object`, i.e., a reference to arbitrary Python objects. The

latter is handy for the other main use case of genomic arrays, namely providing access to metadata. Given a genomic interval, for example, the interval a read was aligned to, it may be interesting to know which genomic features this interval overlaps.

For RNA-Seq data, determining the overlaps between aligned reads and exons is a common approach to estimating for each gene how many reads originated from its transcripts. This can be achieved by using the HTSeq *GFF_Parser* to read in a GFF file with exon annotations and a *GenomicArray* to store the exons at their respective genomic intervals. The second step is to iterate through the aligned reads (using e.g. the *BAM_Reader*) and finding the overlaps between the reads genomic intervals and the exons genomic intervals. To this end the *GenomicArray* object can be queried with a *GenomicInterval* object and will return all the steps overlapping that interval.

Here, dealing with the situation of a read overlapping with no feature or a single feature is usually straightforward, while an overlap with several features is more challenging to interpret. The possibility to code custom logic at this point is a core advantage of our approach of iterating sequentially through data rather than offering vectorized functions (see Discussion).

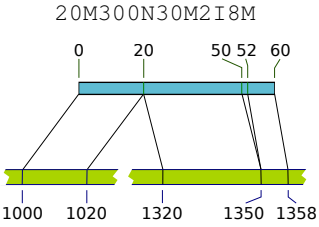
A variant of the *GenomicArray*, the class *GenomicArrayOfSets* facilitates dealing with overlapping genomic features. Figure 2.2 shows a situation with two features (e.g. genes) partially overlapping (and one feature is split by a gap, e.g., an intron). The depicted meta data can be stored in a *GenomicArrayOfSets* object `f` by performing three assignment operations, typically using the syntax "`f[iv] += a`". This adds the object `a` to the interval specified by the *GenomicInterval* object `iv`, in order to assign objects representing A (twice, once for each of its parts) and B to the corresponding intervals. Afterwards, the *GenomicArrayOfSets* object will contain steps as depicted in Figure 2.2, which carry as values sets of object references.

If afterwards `f[p]` is accessed (where `p` is a *GenomicPosition* object indicating a position within the overlap of A and B), `set([ "A", "B" ])` is returned. When querying with an interval, the return value is an iterator over all steps covered by this interval. In this manner, the *GenomicArrayOfSets* offers functionality to handle metadata associated with overlapping intervals and allows for easy, straight-forward querying.

3 DOCUMENTATION AND CODING STRATEGIES

HTSeq comes with extensive documentation to guide developers. Care has been taken to expect only moderate experience with Python from the reader. A "Tour" offers an overview over the classes and principles of HTSeq by demonstrating their use in simple examples. Then, two common use cases are discussed in detail to show how HTSeq can be applied to complex tasks.

The first use case is that of aggregate coverage profiles: Given ChiP-Seq data, e.g. from histone marks, we want to calculate the profile of these marks with respect to specific positions in the genome, such as transcription start sites (TSSs). This is achieved by aligning coverage data in windows centred on the TSSs and averaging over the TSSs of all genes or a subset thereof. The documentation discusses and compares three alternative strategies to combine the data on TSS positions with the data on read alignments.



type	size	query	reference
M	20	0–20	chr1: 1000–1020
N	300	20–20	chr1: 1020–1320
M	30	20–50	chr1: 1320–1350
I	2	50–52	chr1: 1350–1350
M	8	52–60	chr1: 1350–1358

Fig. 3. The detailed structure of a read alignment is presented as a list of objects of class *CigarOperation*, which interprets the CIGAR strings given in a SAM file and calculates which bases on both the read and the reference are affected by the individual operations. For example, the CIGAR string "20M300N30M2I8M", which indicates the complex mapping situation depicted here, is represented by five *CigarOperation* objects, each providing the data in one of the lines in the table (right).

As this serves to illustrate three typical ways to write scripts with HTSeq, we summarise them here briefly:

(i) In the case of relatively small genomes we suggest processing all read alignments into a tally of alignment coverage for the whole genome and storing it in a *GenomicArray*. The *GenomicArray* can then be queried with the genomic intervals corresponding to the selected TSSs to obtain the coverage profiles in windows around them, which are then added up in a vector that will hold the result.

(ii) The alternative is to proceed in the opposite order: The list of TSSs is obtained first and stored in a *GenomicArray* object. Then, the read alignments are processed sequentially: For each alignment, the *GenomicArray* is queried to find nearby TSSs and the result vector is incremented where a match is found.

(iii) The *BAM_Reader* offers random access on an indexed *BAM* file and can therefore be used to extract only the reads overlapping a specified interval. Using this functionality we can iterate through the TSS positions and query the *BAM* file for alignments in a window around each TSS and construct the profiles from this data.

The documentation demonstrates each of these approaches with worked-out code examples to illustrate these various manners of working with the HTSeq library.

The second use case discussed in detail is that of counting for each gene in a genome how many RNA-Seq reads map to it. In this context, the HTSeq class *CigarOperation* is demonstrated, which represents complex alignments in a convenient form (Figure 3). This section of the documentation also explains HTSeq's facilities to handle multiple alignments and paired-end data.

The remainder of the documentation provides references for all classes and functions provided by HTSeq, including those classes not used in the highlighted use cases of the tutorial part, such as the facilities to deal with variant-call format (*VCF*) files.

4 HTSEQ-QA AND HTSEQ-COUNT

We distribute two stand-alone scripts with HTSeq, which can be used from the shell command line, without any Python knowledge, and hence are also of use to practitioners without Python knowledge – and also illustrate potential applications of the HTSeq framework.

The script *htseq-qa* is a simple tool for initial quality assessment of sequencing runs. It produces plots as in Figure 4, which summarise the nucleotide compositions of the positions in the read and the base-call qualities.

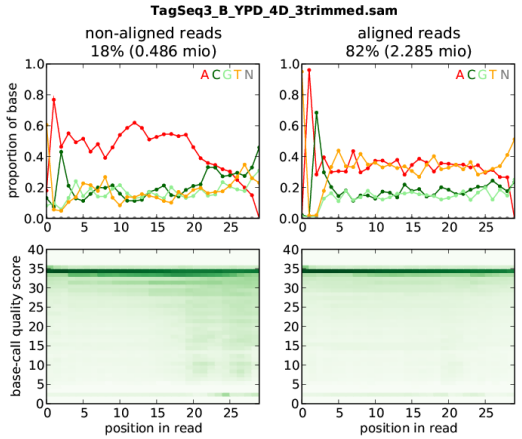


Fig. 4. Example of a quality report on a SAM file, produced by *htseq-qa*. The top panels show the nucleotide composition, the bottom panels the distribution of base-call quality scores, as they change with sequencing cycle shown from the 5' to the 3' end of the reads. The left panel shows statistics for reads that have not been aligned, the right panels for the aligned reads.

The script *htseq-count* is a tool for RNA-Seq data analysis: Given a SAM/BAM file and a GTF or GFF file with gene models, it counts for each gene how many aligned reads overlap its exons. These counts can then be used for gene-level differential expression analyses using methods such as *DESeq2* (Anders and Huber, 2010) or *edgeR* (Robinson *et al.*, 2010). As the script is designed specifically for *differential* expression analysis, only reads mapping unambiguously to a single gene are counted while reads aligned to multiple positions or overlapping with more than one gene are discarded. To see why this is desirable, consider two genes with some sequence similarity, one of which is differentially expressed while the other one is not. A read that maps to both genes equally well should be discarded, because if it were counted for both genes, the extra reads from the differentially expressed gene may cause the other gene to be wrongly called differentially expressed, too. Another design choice made with the downstream application of differential expression testing in mind is to count fragments, not reads, in case of paired-end data. This is because the two mates originating from the same fragment provide only evidence for one cDNA fragment and should hence be counted only once.

As the *htseq-count* script has found widespread use over the last three years, we note that we recently replaced it with an overhauled version, which now allows to process paired-end data without the need to sort the SAM/BAM file by read name first. See the documentation for a list of all changes to the original version.

5 DISCUSSION

HTSeq aims to fill the gap between performant but monolithic tools optimised for specialised tasks and the need to write data processing code for HTS application entirely from scratch. For a number of the smaller taks covered by HTSeq, good stand-alone solutions exist, e.g. *FastQC* (Andrews *et al.*, 2011) for quality assessment or *Trimmomatic* (Lohse *et al.*, 2012) for trimming of reads. If the specific approaches chosen by the developers of these tools are suitable for a user's application, they are easier to use. However, the need to write customised code will inevitably arise in many projects, and then, HTSeq aims to offer advantages over other programming libraries that focus on specific file formats, e.g. *PySam* (Heger *et al.*) and *Picard* (Wysoker *et al.*) for SAM/BAM files, by integrating parsers for many common file formats and fixing conventions for data interchange between them. HTSeq complements Biopython (Cock *et al.*, 2009), which provides similar functionality for more "classic" bioinformatics tasks such as sequence analysis and phylogenetic analyses but offers little support for HTS tasks.

While most uses of HTSeq will be the development of custom scripts for one specific analysis task, it can also be useful for writing more general tools which may then be deployed for use by the community. The *htseq-count* script, for example, prepares a count table for differential expression analysis, a seemingly easy task, which, however, becomes complicated when ambiguous cases have to be treated correctly. When HTSeq was first released in 2010, no other simple solution was available, but we note that by now, further tools for this task have appeared, including the *summerizeOverlap* function in the *GenomicRanges* Bioconductor package (Lawrence *et al.*, 2013) and the stand-alone tool *featureCount* (Liao *et al.*, 2013), which is implemented in C. Nevertheless, neither *htseq-count* nor the other tools offer much flexibility to deal with special cases, which is why the HTSeq documentation discusses in detail how users can write their own scripts for this important use case.

Interval queries are a recurring task in HTS analysis problems, and several libraries offer solutions for various programming languages, including BEDtools (Quinlan and Hall, 2010; Dale *et al.*, 2011) and IRanges/GenomicRanges (Lawrence *et al.*, 2013) Typically, these methods take two lists of intervals and report overlaps between intervals in the latter with intervals in the former list. HTSeq uses a different paradigm, namely that one list of intervals is read in and stored in a *GenomicArrayOfSets* object, and then the other intervals are queried one by one, in a loop. This explicit looping often simplifies development for users, and hence we prefer it over the use of specialized algorithms optimised for list-vs-list queries. The advantage is especially apparent in the read counting problem discussed above, where split reads, gapped alignments, ambiguous mappings etc. cause much need for treatment of special cases within the inner loop.

In conclusion, HTSeq offers a comprehensive solution to facilitate a wide range of programming tasks in the context of high-throughput sequencing data analysis.

REFERENCES

Anders, S. and Huber, W. 2010 Differential expression analysis for sequence count data. *Genome Biology*, **11**(10), R106.

Andrews, S. *et al.* 2011 FastQC – A quality control tool for high throughput sequence data.

Beazley, D. M. *et al.* 1996 SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pp. 129–139.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S. and Smith, K. 2011 Cython: The best of both worlds. *Computing in Science & Engineering*, **13**(2), 31–39.

Cock, P. J., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F. *et al.* 2009 Biopython: freely available Python tools for computational molecular biology and

bioinformatics. *Bioinformatics*, **25**(11), 1422–1423.

Dale, R. K., Pedersen, B. S. and Quinlan, A. R. 2011 Pybedtools: a flexible Python library for manipulating genomic datasets and annotations. *Bioinformatics*, **27**(24), 3423–3424.

Heger, A., Belgrad, T. G., Goodson, M., Goodstad, L. and Jacobs, K. pysam: samtools interface for python. http://code.google.com/p/pysam/.

Josuttis, N. M. 1999 *The C++ Standard Library*. Addison-Wesley.

Lawrence, M., Huber, W., Pages, H., Aboyoun, P., Carlson, M., Gentleman, R., Morgan, M. T. and Carey, V. J. 2013 Software for computing and annotating genomic ranges. *PLoS Computational Biology*, **9**(8), e1003 118.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G. and Durbin, R. 2009 The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, **25**(16), 2078–9.

Liao, Y., Smyth, G. K. and Shi, W. 2013 featureCounts: an efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics*.

Lohse, M., Bolger, A. M., Nagel, A., Fernie, A. R., Lunn, J. E., Stitt, M. and Usadel, B. 2012 RobiNA: a user-friendly, integrated software solution for RNA-Seq-based transcriptomics. *Nucleic Acids Research*, **40**(W1), W622–W627.

Oliphant, T. E. *et al.* Numpy. http://numpy.scipy.org.

Quinlan, A. R. and Hall, I. M. 2010 Bedtools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, **26**(6), 841–842.

Robinson, M. D., McCarthy, D. J. and Smyth, G. K. 2010 edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, **26**(1), 139–40.

Sedgewick, R. 1988 *Algorithms*. Addison-Wesley, 2nd edn.

Wysoker, A., Tibbetts, K., McCowan, M. and Fennell, T. Picard. http://picard.sourceforge.net/.