FastaIO.jl Documentation

Release 0.2-dev

Carlo Baldassi

June 17, 2013

CONTENTS

This module provides ways to parse and write files in FASTA format in Julia. It is designed to be lightweight and fast; the parsing method is inspired to kseq.h. It can read and write files on the fly, keeping only one entry at a time in memory, and it can read and write gzip-compressed files.

Here is a quick example for reading a file:

```
julia> using FastaIO

julia> FastaReader("somefile.fasta") do fr
         for (desc, seq) in fr
           println("$desc : $seq")
         end
       end
```

And for writing:

```
julia> using FastaIO

julia> FastaWriter("somefile.fasta") do fw
         for s in [">GENE1", "GCATT", ">GENE2", "ATTAGC"]
           write(fw, s)
         end
       end
```

CHAPTER

ONE

INSTALLATION

You can install FastaIO from Julia's package manager:

```
julia> Pkg.add("FastaIO")
```

CHAPTER

TWO

USAGE

2.1 Introductory notes

For both reading and writing, there are quick methods to read/write all the data at once: `readfasta()` and `writefasta()`. These, however, require all the data to be stored in memory at once, which may be impossible or undesirable for very large files. Therefore, for both reading and writing, the preferred way is actually to use specialized types, `FastaReader` and `FastaWriter`, which have the ability to process one entry (description + sequence data) at a time (the writer can actually process one char at a time); however, note that these two object types are not symmetric: the reader acts as an iterable object, while the writer behaves similarly to an `IO` stream.

2.2 The FASTA format

The FASTA format which is assumed by this module is as follows:

1. description lines must start with a > character, and cannot be empty
2. only one description line per entry is allowed
3. all characters must be ASCII
4. whitespace is not allowed within sequence data (except for newlines) and at the beginning or end of the description

When writing, description lines longer than 80 characters will trigger a warning message; sequence data is formatted in lines of 80 characters each; extra whitespace is silently discarded. No other restriction is put on the content of the sequence data, except that the > character is forbidden.

When reading, almost no explicit checks are performed to test that the data actually conforms to these specifications.

2.3 The sequence storage type

When reading FASTA files, the container type used to store the sequence data can be chosen (as an optional argument to `readfasta()` or as a parametric type of `FastaReader`). The default is `ASCIIString`, which is the most memory-efficient and the fastest; another performance-optimal option is `Vector{Uint8}`, which is a less friendly representation, but has the advantage of being mutable. Any other container `T` for which `convert(::Type{T}, ::Vector{Uint8})` is defined can be used (e.g. `Vector{Char}`, or a more specialized `Vector{AminoAcid}` if you use the BioSeq package), but the conversion will generally slightly reduce the performance.

2.4 Reading files

readfasta (*filename::String*[, *sequence_type::Type = ASCIIString*])
readfasta (*io::IO*[, *sequence_type::Type = ASCIIString*])

This function parses a whole FASTA file at once and stores it into memory. The result is a `Vector{Any}` whose elements are tuples consisting of `(description, sequence)`, where `description` is an `ASCIIString` and `sequence` contains the sequence data, stored in a container type defined by the *sequence_type* optional argument (see *this section* for more information).

FastaReader{T} (*filename::String*)
FastaReader{T} (*io::IO*)

This creates an object which is able to parse FASTA files, one entry at a time. `filename` can be a plain text file or a gzip-compressed file (it will be autodetected from the content). The type `T` determines the output type of the sequences (see *this section* for more information) and it defaults to `ASCIIString`.

The data can be read out by iterating the `FastaReader` object:

```
for (name, seq) in FastaReader("somefile.fasta")
    # do something with name and seq
end
```

As shown, the iterator returns a tuple containing the description (always an `ASCIIString`) and the data (whose type is set when creating the `FastaReader` object (e.g. `FastaReader{Vector{Uint8}}(filename)`).

The `FastaReader` type has a field `num_parsed` which contains the number of entries parsed so far.

Other ways to read out the data are via the `readentry()` and `readall()` functions.

FastaReader{T} (*f::Function*, *filename::String*)

This format of the constructor is useful for do-notation, i.e.:

```
FastaReader(filename) do fr
  # read out the data from fr
end
```

which ensures that the `close()` function is called; however, this is not crucial, since the function will be called anyway by the finalizer when the `FastaReader` object goes out of scope and is garbage-collected.

readentry (*fr::FastaReader*)

This function can be used to read entries one at a time:

```
fr = FastaReader("somefile.fasta")
name, seq = readentry(fr)
```

See also the `eof()` function.

readall (*fr::FastaReader*)

This function extends `Base.readall()`: it parses a whole FASTA file at once, and returns an array of tuples, each one containing the description and the sequence (see also the `readfasta()` function).

rewind (*fr::FastaReader*)

This function rewinds the reader, so that it can restart the parsing again without closing and re-opening it. It also resets the value of the `num_parsed` field.

eof (*fr::FastaReader*)

This function extends `Base.eof()` and tests for end-of-file condition; it is useful when using `readentry()`:

```
fr = FastaReader("somefile.fasta")
while !eof(fr)
```

```
    name, seq = readentry(fr)
    # do something
end
close(fr)
```

close (*fr::FastaReader*)
This function extends `Base.close()` and closes the stream associated with the `FastaReader`; the reader must not be used any more after this function is called.

2.5 Writing files

writefasta (*filename::String*, *data*[, *mode::String = "w"*])
writefasta ([*io::IO = OUTPUT_STREAM*], *data*)
This function dumps data to a FASTA file, auto-formatting it so to follow the specifications detailed in *this section*. The `data` can be anything which is iterable and which produces `(description, sequence)` tuples upon iteration, where the `description` must be convertible to an `ASCIIString` and the `sequence` can be any iterable object which yields elements convertible to ASCII characters (e.g. an `ASCIIString`, a `Vector{Uint8}` etc.).

Examples:

```
writefasta("somefile.fasta", [("GENE1", "GCATT"), ("GENE2", "ATTAGC")])
writefasta("somefile.fasta", ["GENE1" => "GCATT", "GENE2" => "ATTAGC"])
```

If the `filename` ends with `.gz`, the result will be a gzip-compressed file.

The `mode` flag determines how the `filename` is open; use `"a"` to append the data to an existing file.

FastaWriter (*filename::String*[, *mode::String = "w"*])
FastaWriter ([*io::IO = OUTPUT_STREAM*])
FastaWriter (*f::Function*, *args...*)
This creates an object which is able to write formatted FASTA files which conform to the specifications detailed in *this section*, via the `write()` and `writeentry()` functions.

The third form allows to use do-notation:

```
FastaWriter("somefile.fasta") do fw
  # write the file
end
```

which is strongly recommended since it ensures that the `close()` function is called at the end of writing: this is crucial, as failing to do so may result in incomplete files (this is done by the finalizer, so it will still happen automatically if the `FastaWriter` object goes out of scope and is garbage-collected, but there is no guarantee that this will happen if Julia exits).

If the `filename` ends with `.gz`, the result will be gzip-compressed.

The `mode` flag can be used to set the opening mode of the file; use `"a"` to append to an existing file.

The `FastaWriter` object has an `entry::Int` field which stores the number of the entry which is currently being written.

writeentry (*fw::FastaWriter*, *description::String*, *sequence*)
This function writes one entry to the FASTA file, following the specifications detailed in *this section*. The `description` is without the initial `'>'` character. The `sequence` can be any iterable object whose elements are convertible to ASCII characters.

Example:

```
FastaWriter("somefile.fasta") do fw
  for (desc,seq) in [("GENE1", "GCATT"), ("GENE2", "ATTAGC")]
    writeentry(fw, desc, seq)
  end
end
```

write (*fw::FastaWriter*, *item*)

This function extends `Base.write()` and streams items to a FASTA file, which will be formatted according to the specifications detailed in *this section*.

When using this method, description lines are marked by the fact that they begin with a `'>'` character; anything else is assumed to be part of the sequence data.

If `item` is a `Vector`, `write` will be called iteratively over it; if it is a `String`, a newline will be appended to it and it will be dumped. For example the following code:

```
FastaWriter("somefile.fasta") do fw
  for s in [">GENE1", "GCA", "TTT", ">GENE2", "ATTAGC"]
    write(fw, s)
  end
end
```

will result in the file:

```
>GENE1
GCATTT
>GENE2
ATTAGC
```

If `item` is not a `Vector` nor a `String`, it must be convertible to an ASCII character, and it will be piped into the file. For example the following code:

```
data = """
  >GENE1
  GCA
  TTT
  >GENE2
  ATT
  AGC
  """

FastaWriter("somefile.fasta") do fw
  for ch in data
    write(fw, ch)
  end
end
```

will result in the same file as above.

close (*fw::FastaWriter*)

This function extends `Base.close()` and it should always be explicitly used for finalizing the `FastaWriter` once the writing has finished, unless the do-notation is used when creating it.

PYTHON MODULE INDEX

f

PYTHON MODULE INDEX

f