fqfa

Release 1.2.2

Feb 21, 2023

Contents

fqfa is a pure Python library for bioinformatics and computational biology. It implements parsers for FASTA and FASTQ files as well as some commonly-used sequence operations.

To cite fqfa please refer to "fqfa: A pure Python package for genomic sequence files".

Install fqfa from PyPI using pip:

```
pip3 install fqfa
```

To install the package for development purposes, include the optional dependencies:

```
pip3 install fqfa[dev]
```

Building a local copy of the documentation requires the following additional packages:

```
pip3 install sphinx
pip3 install sphinx-rtd-theme
```

CHAPTER 1

Usage examples

This page contains some example use cases for fqfa. They are formatted as `doctest` tests.

1.1 Basic sequence validation

The validators in `fqfa.validator.validator` return a match object if the sequence validates or None if the sequence doesn't validate. This means that they can be used in simple if statements.

This validator only accepts the standard DNA bases, so the input sequence is invalid.

```
>>> if dna_bases_validator("ACGTNW"):
...     print("valid!")
... else:
...     print("invalid!")
invalid!
```

This validator accepts all IUPAC bases, so the input sequence is valid.

```
>>> if dna_characters_validator("ACGTNW"):
...     print("valid!")
... else:
...     print("invalid!")
valid!
```

The validators only accept strings (or bytes), as they are based on regular expressions. Attempting to validate anything else results in a `TypeError`.

```
>>> if dna_characters_validator(42):
...     print("valid!")
... else:
...     print("invalid!")
Traceback (most recent call last):
    ...
TypeError: expected string or bytes-like object
```

Default validators only accept uppercase characters, so mixed-case or lowercase input is invalid.

```
>>> if dna_bases_validator("ACgT"):
...     print("valid!")
... else:
...     print("invalid!")
invalid!
```

Case-insensitive validators can be created using `create_validator()`.

```
>>> case_insensitive_validator = create_validator(DNA_BASES, case_sensitive=False)
>>> if case_insensitive_validator("ACgT"):
...     print("valid!")
... else:
...     print("invalid!")
valid!
```

1.2 Translating FASTA sequences

fqfa implements a function to parse individual records from FASTA files.

```
>>> fasta_string = """
... >test record
... ACGAAA
... TAA
...
... >another record here
... ACANaa
... """
>>> for header, seq in parse_fasta_records(StringIO(fasta_string)):
...     print(header)
...     print(seq)
test record
ACGAAATAA
another record here
ACANaa
```

These sequences can be validated and/or transformed using utility functions in the library and rewritten as FASTA output.

```
>>> fasta_string = """
... >test record
... ACGAAA
... TAA
... """
>>> output_file = StringIO()
>>> for header, dna_seq in parse_fasta_records(StringIO(fasta_string)):
...     if dna_bases_validator(dna_seq):
...         protein_seq, _ = translate_dna(dna_seq)
...         write_fasta_record(output_file, header, protein_seq)
>>> output_file.seek(0)
0
>>> print(output_file.read())
>test record
TK*
```

1.3 Filtering paired-end FASTQ reads on sequence quality

fqfa can read single FASTQ files or a pair of FASTQ files in parallel.

```
>>> fastq_fwd = """\
... @TEST:123:456 AAA
... ACGTAA
... +
... AAA!CD
... @TEST:999:888 AAA
... AAACCC
... +
... ABABAB
... """
>>> fastq_rev = """\
... @TEST:123:456 BBB
... TTTTTT
... +
... ACACAC
... @TEST:999:888 BBB
... GGGCCC
... +
... BBBAAA
... """
>>> for fwd, rev in parse_fastq_pe_reads(StringIO(fastq_fwd), StringIO(fastq_rev)):
...     print(f"{fwd.sequence}-{rev.sequence}")
ACGTAA-TTTTTT
AAACCC-GGGCCC
```

The `parse_fastq_reads()` and `parse_fastq_pe_reads()` functions are generators that return `FastqRead` objects, so the relevant class methods can be used for filtering or trimming of reads as they are processed.

```
>>> fastq_fwd = """\
... @TEST:123:456 AAA
... ACGTAA
... +
... AAA!CD
... @TEST:999:888 AAA
... AAACCC
... +
... ABABAB
... """
>>> fastq_rev = """\
... @TEST:123:456 BBB
... TTTTTT
... +
... ACACAC
... @TEST:999:888 BBB
... GGGCCC
... +
... BBBAAA
... """
>>> for fwd, rev in parse_fastq_pe_reads(StringIO(fastq_fwd), StringIO(fastq_rev)):
...     if fwd.min_quality() > 20 and rev.min_quality() > 20:
...         print(f"{fwd.sequence}-{rev.sequence}")
AAACCC-GGGCCC
```

CHAPTER 2

File handling

fqfa implements several functions to help open FASTA and FASTQ data files. This includes functions for validating file names as well as for opening compressed file handles. Currently fqfa supports opening files compressed with bzip2 or gzip. Generally speaking, gzip is faster and more widely-supported by other bioinformatics software, but bzip2 offers slightly better compression that may be relevant for large FASTQ files that are not frequently accessed.

The generator functions for FASTA and FASTQ files take open file handles as their arguments, supporting the use of `open_compressed()`.

2.1 FASTA files

fqfa has basic support for FASTA files. This is designed for small FASTA files such as those containing gene or plasmid sequences. fqfa does not use or create FASTA index (`.fai`) files.

The generator function below that parses FASTA files is slightly more flexible than the FASTA specification. Specifically, it ignores any lines before the first FASTA record, allowing for comments or other metadata at the start of the file, and allows any amount of leading or trailing whitespace in the sequence (including blank lines within a record).

No validation is performed on the sequences, but fqfa implements a set of *callable validators* that can be used.

2.2 FASTQ files

fqfa supports reading FASTQ files either singly or as a pair (for paired-end data). Reads are returned as `FastqRead` objects. These objects support several basic operations, such as in-place read trimming and calculating quality-based values. The sequence and headers are stored as strings, and the quality values are stored as a list of integers.

Note that there are no FASTQ output functions, because the `__str__()` method formats a `FastqRead` object as a standard FASTQ record. Generating a FASTQ output file is as simple as printing all the objects.

CHAPTER 3

Sequence validation

fqfa implements regular expression-based sequence validators. There are several commonly-used validators based on *IUPAC codes*, as well as a function for creating new callable validators from a string or list of characters. This `create_validator()` function can also be used to create case-insensitive versions of the provided validators.

3.1 IUPAC codes

fqfa includes the International Union of Pure and Applied Chemistry (IUPAC) notation for degenerate bases. A mapping between single- and three-letter amino acid codes is also included. Validation based on single-letter amino acid codes can be accomplished by using the keys of the mapping.

3.1.1 DNA sequences

3.1.2 RNA sequences

3.1.3 Amino acid sequences

CHAPTER 4

Utility functions

fqfa provides basic utility functions for working with biological sequences as strings. For efficiency, these functions assume that any required validation (such as making sure all the characters string are valid bases) has already been performed.

fqfa has a copy of the *standard translation table* and alternative translation tables can be imported using `ncbi_genetic_code_to_dict()`.

4.1 Nucleotide sequence utility functions

4.2 Coding sequence translation

4.3 Sequence type inference functions

CHAPTER 5

Performance comparison

This page contains some performance and usage comparisons for processing FASTQ files with fqfa and pyfastx.

In these benchmarks, fqfa is comparable to pyfastx, although pyfastx has made substantial performance improvements since fqfa was written, particularly when reading gzip-compressed input files.

The results are derived from Jupyter notebooks. If you'd like to run this code yourself, the notebooks are available with the fqfa documentation in `fqfa/docs/notebooks`. The file used in the benchmark is from the Enrich2 example dataset. To run the benchmarks as written, you will have to decompress the bz2 file and also create a gzipped version.

This section includes examples of usage that are common in my work, primarily in processing files of barcode reads for high-throughput functional genomic assays. pyfastx includes many other functions that are not demonstrated here.

5.1 Benchmarking for raw FASTQ files

```
import pyfastx
from fqfa.fastq.fastq import parse_fastq_reads
```

5.1.1 Benchmark 1: list of reads

This code creates a list containing all the reads in the file. Note that the data structures for the reads are quite different, with two being package-specific objects and one being a tuple.

pyfastx with index

Much of the time spent in the first example is likely spent building the `.fxi` index file. This file enables direct access into the FASTQ file, which we are not using here. The index is quite large, much larger than the reads in this case:

```
334M    BRCA1_input_sample.fq
 48M    BRCA1_input_sample.fq.bz2
511M    BRCA1_input_sample.fq.fxi
 68M    BRCA1_input_sample.fq.gz
513M    BRCA1_input_sample.fq.gz.fxi
```

```
%time reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq")]
for x in reads[:5]:
    print(repr(x))
del reads
```

```
CPU times: user 6.69 s, sys: 993 ms, total: 7.68 s
Wall time: 7.73 s
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1 with length of 16
```

pyfastx without index

This is by far the fastest for just reading data from the file, but it doesn't perform any extra computation or quality value conversion.

```
%time reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq", build_index=False)]
for x in reads[:5]:
    print(x)
del reads
```

```
CPU times: user 1.42 s, sys: 417 ms, total: 1.83 s
Wall time: 1.93 s
('140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1', 'CCCGTGGCCTTTTCCA',
↪'B@CFFFFFHHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1', 'TTTGGTAAAGGGTAAC',
↪'BBCFFDFFHHHHDHIJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1', 'AATAATGTATGTACCT',
↪'BC@FFFFEFHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1', 'CTATTGCGTGTGATCT',
↪'BCCFFFFFHHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1', 'ACCCCTACCCTCTGCC',
↪'BBBFFFFFHHHHHJJJ')
```

fqfa

Unlike pyfastx, fqfa takes an open file handle rather than a file name. In these examples, this is addressed using a context created by a with statement.

```
with open("BRCA1_input_sample.fq") as handle:
    %time reads = [x for x in parse_fastq_reads(handle)]
for x in reads[:5]:
    print(x)
del reads
```

```
CPU times: user 26.7 s, sys: 1.03 s, total: 27.8 s
Wall time: 27.8 s
@140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1
CCCGTGGCCTTTTCCA
+
B@CFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1
TTTGGTAAAGGGTAAC
+
BBCFFDFFHHHHDHIJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1
AATAATGTATGTACCT
+
BC@FFFFEFHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1
CTATTGCGTGTGATCT
+
BCCFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1
ACCCCTACCCTCTGCC
+
BBBFFFFFHHHHHJJJ
```

5.1.2 Benchmark 2: summarized quality statistics

This code calculates the median average read quality for all reads in the file.

```
from statistics import mean, median
```

pyfastx with index

pyfastx provides integer quality values as part of its FASTQ read data structure.

```
%time read_quals = [mean(x.quali) for x in pyfastx.Fastq("BRCA1_input_sample.fq")]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 54.8 s, sys: 630 ms, total: 55.5 s
Wall time: 55.9 s
Median average quality is 37.5
```

pyfastx without index

The timing here is quite a bit closer to the others, since the conversion and calculation has not already been performed as part of processing the input file.

```
%time read_quals = [mean([ord(c) - 33 for c in x[2]]) for x in pyfastx.Fastq("BRCA1_
↪input_sample.fq", build_index=False)]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 53.9 s, sys: 95.4 ms, total: 54 s
Wall time: 54 s
Median average quality is 37.5
```

fqfa

This code uses the `average_quality()` method implemented by the FastqRead class.

```
with open("BRCA1_input_sample.fq") as handle:
    %time read_quals = [x.average_quality() for x in parse_fastq_reads(handle)]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 1min 19s, sys: 146 ms, total: 1min 19s
Wall time: 1min 19s
Median average quality is 37.5
```

5.1.3 Benchmark 3: filtering reads on quality

This code creates a list of reads for which all bases are at least Q20. The performance and usage in this section is quite a bit faster than Benchmark 2 following recent performance improvements in pyfastx.

pyfastx with index

```
%time filt_reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq") if min(x.quali)␣
↪>= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 5.75 s, sys: 556 ms, total: 6.3 s
Wall time: 6.32 s
Kept 3641707 reads after applying filter.
```

pyfastx without index

```
%time filt_reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq", build_
↪index=False) if min([ord(c) - 33 for c in x[2]]) >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 6.71 s, sys: 472 ms, total: 7.18 s
Wall time: 7.25 s
Kept 3641762 reads after applying filter.
```

fqfa

This code uses the `min_quality()` method implemented by the FastqRead class.

```
with open("BRCA1_input_sample.fq") as handle:
    %time filt_reads = [x for x in parse_fastq_reads(handle) if x.min_quality() >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 30.9 s, sys: 4.38 s, total: 35.3 s
Wall time: 1min 15s
Kept 3641762 reads after applying filter.
```

5.2 Benchmarking for gzip-compressed FASTQ files

```
import pyfastx
from fqfa.fastq.fastq import parse_fastq_reads
from fqfa.util.file import open_compressed
```

5.2.1 Benchmark 1: list of reads

This code creates a list containing all the reads in the file. Note that the data structures for the reads are quite different, with two being package-specific objects and one being a tuple.

pyfastx with index

Much of the time spent in the first example is likely spent building the `.fxi` index file. This file enables direct access into the FASTQ file, which we are not using here. The index is quite large, much larger than the reads in this case:

```
334M    BRCA1_input_sample.fq
 48M    BRCA1_input_sample.fq.bz2
511M    BRCA1_input_sample.fq.fxi
 68M    BRCA1_input_sample.fq.gz
513M    BRCA1_input_sample.fq.gz.fxi
```

```
%time reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq.gz")]
for x in reads[:5]:
    print(repr(x))
del reads
```

```
CPU times: user 9.1 s, sys: 1.05 s, total: 10.1 s
Wall time: 10.2 s
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1 with length of 16
<Read> 140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1 with length of 16
```

pyfastx without index

This is by far the fastest for just reading data from the file, but it doesn't perform any extra computation or quality value conversion.

```
%time reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq.gz", build_
↪index=False)]
for x in reads[:5]:
    print(x)
del reads
```

```
CPU times: user 2.59 s, sys: 312 ms, total: 2.9 s
Wall time: 2.9 s
('140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1', 'CCCGTGGCCTTTTCCA',
↪'B@CFFFFFHHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1', 'TTTGGTAAAGGGTAAC',
↪'BBCFFDFFHHHHDHIJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1', 'AATAATGTATGTACCT',
↪'BC@FFFFEFHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1', 'CTATTGCGTGTGATCT',
↪'BCCFFFFFHHHHHJJJ')
('140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1', 'ACCCCTACCCTCTGCC',
↪'BBBFFFFFHHHHHJJJ')
```

fqfa

Unlike pyfastx, fqfa takes an open file handle rather than a file name. In these examples, this is addressed using a context created by a with statement.

```
with open_compressed("BRCA1_input_sample.fq.gz") as handle:
    %time reads = [x for x in parse_fastq_reads(handle)]
for x in reads[:5]:
    print(x)
del reads
```

```
CPU times: user 30.8 s, sys: 881 ms, total: 31.6 s
Wall time: 31.6 s
@140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1
CCCGTGGCCTTTTCCA
+
B@CFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1
TTTGGTAAAGGGTAAC
+
BBCFFDFFHHHHDHIJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1
AATAATGTATGTACCT
+
BC@FFFFEFHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1
CTATTGCGTGTGATCT
+
BCCFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1
ACCCCTACCCTCTGCC
+
BBBFFFFFHHHHHJJJ
```

5.2.2 Benchmark 2: summarized quality statistics

This code calculates the median average read quality for all reads in the file.

```
from statistics import mean, median
```

pyfastx with index

pyfastx provides integer quality values as part of its FASTQ read data structure.

Note: this step ran for over an hour without completing, so timing information is not provided.

```
%time read_quals = [mean(x.quali) for x in pyfastx.Fastq("BRCA1_input_sample.fq.gz")]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 53.9 s, sys: 323 ms, total: 54.2 s
Wall time: 54.2 s
Median average quality is 37.5
```

pyfastx without index

The timing here is quite a bit closer to the others, since the conversion and calculation has not already been performed as part of processing the input file.

```
%time read_quals = [mean([ord(c) - 33 for c in x[2]]) for x in pyfastx.Fastq("BRCA1_
↪input_sample.fq.gz", build_index=False)]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 55.9 s, sys: 15.4 ms, total: 55.9 s
Wall time: 56 s
Median average quality is 37.5
```

fqfa

This code uses the `average_quality()` method implemented by the FastqRead class.

```
with open_compressed("BRCA1_input_sample.fq.gz") as handle:
    %time read_quals = [x.average_quality() for x in parse_fastq_reads(handle)]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 1min 23s, sys: 55.6 ms, total: 1min 23s
Wall time: 1min 23s
Median average quality is 37.5
```

5.2.3 Benchmark 3: filtering reads on quality

This code creates a list of reads for which all bases are at least Q20. The performance and usage in this section is quite a bit faster than Benchmark 2 following recent performance improvements in pyfastx.

pyfastx with index

Note: this step ran for over an hour without completing, so timing information is not provided.

```
%time filt_reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq.gz") if min(x.
↪quali) >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 6.17 s, sys: 360 ms, total: 6.53 s
Wall time: 6.53 s
Kept 3641707 reads after applying filter.
```

pyfastx without index

```
%time filt_reads = [x for x in pyfastx.Fastq("BRCA1_input_sample.fq.gz", build_
↪index=False) if min([ord(c) - 33 for c in x[2]]) >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 7.24 s, sys: 620 ms, total: 7.86 s
Wall time: 7.87 s
Kept 3641762 reads after applying filter.
```

fqfa

This code uses the `min_quality()` method implemented by the FastqRead class.

```
with open_compressed("BRCA1_input_sample.fq.gz") as handle:
    %time filt_reads = [x for x in parse_fastq_reads(handle) if x.min_quality() >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 31.2 s, sys: 660 ms, total: 31.9 s
Wall time: 31.9 s
Kept 3641762 reads after applying filter.
```

5.3 Benchmarking for bzip2-compressed FASTQ files

```
from fqfa.fastq.fastq import parse_fastq_reads
from fqfa.util.file import open_compressed
```

5.3.1 Benchmark 1: list of reads

This code creates a list containing all the reads in the file. Note that the data structures for the reads are quite different, with two being package-specific objects and one being a tuple.

Because pyfastx does not support bzip2, these results are most useful for comparing with fqfa's gzip benchmarks.

fqfa

Unlike pyfastx, fqfa takes an open file handle rather than a file name. In these examples, this is addressed using a context created by a with statement.

```
with open_compressed("BRCA1_input_sample.fq.bz2") as handle:
    %time reads = [x for x in parse_fastq_reads(handle)]
for x in reads[:5]:
    print(x)
del reads
```

```
CPU times: user 42.2 s, sys: 1.05 s, total: 43.3 s
Wall time: 43.4 s
@140313_SN743_0432_AC3TTHACXX:4:1101:5633:2224:1#0/1
CCCGTGGCCTTTTCCA
+
B@CFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6580:2239:1#0/1
TTTGGTAAAGGGTAAC
+
BBCFFDFFHHHHDHIJ
@140313_SN743_0432_AC3TTHACXX:4:1101:6929:2242:1#0/1
AATAATGTATGTACCT
+
BC@FFFFEFHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:13004:2221:1#0/1
CTATTGCGTGTGATCT
+
BCCFFFFFHHHHHJJJ
@140313_SN743_0432_AC3TTHACXX:4:1101:14034:2219:1#0/1
ACCCCTACCCTCTGCC
+
BBBFFFFFHHHHHJJJ
```

5.3.2 Benchmark 2: summarized quality statistics

This code calculates the median average read quality for all reads in the file.

```
from statistics import median
```

fqfa

This code uses the `average_quality()` method implemented by the FastqRead class.

```
with open_compressed("BRCA1_input_sample.fq.bz2") as handle:
    %time read_quals = [x.average_quality() for x in parse_fastq_reads(handle)]
print(f"Median average quality is {median(read_quals)}")
del read_quals
```

```
CPU times: user 1min 35s, sys: 277 ms, total: 1min 35s
Wall time: 1min 35s
Median average quality is 37.5
```

5.3.3 Benchmark 3: filtering reads on quality

This code creates a list of reads for which all bases are at least Q20. The performance and usage in this section is quite similar to Benchmark 2.

fqfa

This code uses the `min_quality()` method implemented by the FastqRead class.

```
with open_compressed("BRCA1_input_sample.fq.bz2") as handle:
    %time filt_reads = [x for x in parse_fastq_reads(handle) if x.min_quality() >= 20]
print(f"Kept {len(filt_reads)} reads after applying filter.")
del filt_reads
```

```
CPU times: user 43 s, sys: 784 ms, total: 43.8 s
Wall time: 43.8 s
Kept 3641762 reads after applying filter.
```