Coverage for /Users/vincentdavis/VersionControl/biopython/Bio/SeqIO/SffIO.py : 56%

594 statements | 333 run | 261 missing | 0 excluded

```
# Copyright 2009-2016 by Peter Cock.  All rights reserved.
# Based on code contributed and copyright 2009 by Jose Blanca (COMAV-UPV).
#
# This code is part of the Biopython distribution and governed by its
# license.  Please see the LICENSE file that should have been included
# as part of this package.
"""Bio.SeqIO support for the binary Standard Flowgram Format (SFF) file format.

SFF was designed by 454 Life Sciences (Roche), the Whitehead Institute for
Biomedical Research and the Wellcome Trust Sanger Institute. SFF was also used
as the native output format from early versions of Ion Torrent's PGM platform
as well. You are expected to use this module via the Bio.SeqIO functions under
the format name "sff" (or "sff-trim" as described below).

For example, to iterate over the records in an SFF file,

    >>> from Bio import SeqIO
    >>> for record in SeqIO.parse("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff"):
    ...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    ...
    E3MFGYR02JWQ7T 265 tcagGGTCTACATGTTGGTT...
    E3MFGYR02JA6IL 271 tcagTTTTTTTTGGAAAGGA...
    E3MFGYR02JHD4H 310 tcagAAAGACAAGTGGTATC...
    E3MFGYR02GFKUC 299 tcagCGGCCGGGCCTCTCAT...
    E3MFGYR02FTGED 281 tcagTGGTAATGGGGGGAAA...
    E3MFGYR02FR9G7 261 tcagCTCCGTAAGAAGGTGC...
    E3MFGYR02GAZMS 278 tcagAAAGAAGTAAGGTAAA...
    E3MFGYR02HHZ8O 221 tcagACTTTCTTCTTTACCG...
    E3MFGYR02GPGB1 269 tcagAAGCAGTGGTATCAAC...
    E3MFGYR02F7Z7G 219 tcagAATCATCCACTTTTTA...

Each SeqRecord object will contain all the annotation from the SFF file,
including the PHRED quality scores.

    >>> print("%s %i" % (record.id, len(record)))
    E3MFGYR02F7Z7G 219
    >>> print("%s..." % record.seq[:10])
    tcagAATCAT...
    >>> print("%r..." % (record.letter_annotations["phred_quality"][:10]))
    [22, 21, 23, 28, 26, 15, 12, 21, 28, 21]...

Notice that the sequence is given in mixed case, the central upper case region
corresponds to the trimmed sequence. This matches the output of the Roche
tools (and the 3rd party tool sff_extract) for SFF to FASTA.

    >>> print(record.annotations["clip_qual_left"])
    4
    >>> print(record.annotations["clip_qual_right"])
    134
    >>> print(record.seq[:4])
    tcag
    >>> print("%s...%s" % (record.seq[4:20], record.seq[120:134]))
    AATCATCCACTTTTTA...CAAAACACAAACAG
    >>> print(record.seq[134:])
    atcttatcaacaaaactcaaagttcctaactgagacacgcaacaggggataagacaaggcacacaggggataggnnnnnnnnnnn

The annotations dictionary also contains any adapter clip positions
(usually zero), and information about the flows. e.g.

    >>> len(record.annotations)
    11
    >>> print(record.annotations["flow_key"])
    TCAG
    >>> print(record.annotations["flow_values"][:10])
    (83, 1, 128, 7, 4, 84, 6, 106, 3, 172)
    >>> print(len(record.annotations["flow_values"]))
```

```
    400
    >>> print(record.annotations["flow_index"][:10])
    (1, 2, 3, 2, 2, 0, 3, 2, 3, 3)
    >>> print(len(record.annotations["flow_index"]))
    219

Note that to convert from a raw reading in flow_values to the corresponding
homopolymer stretch estimate, the value should be rounded to the nearest 100:

    >>> print("%r..." % [int(round(value, -2)) // 100
    ...                  for value in record.annotations["flow_values"][:10]])
    ...
    [1, 0, 1, 0, 0, 1, 0, 1, 0, 2]...

If a read name is exactly 14 alphanumeric characters, the annotations
dictionary will also contain meta-data about the read extracted by
interpretting the name as a 454 Sequencing System "Universal" Accession
Number. Note that if a read name happens to be exactly 14 alphanumeric
characters but was not generated automatically, these annotation records
will contain nonsense information.

    >>> print(record.annotations["region"])
    2
    >>> print(record.annotations["time"])
    [2008, 1, 9, 16, 16, 0]
    >>> print(record.annotations["coords"])
    (2434, 1658)

As a convenience method, you can read the file with SeqIO format name "sff-trim"
instead of "sff" to get just the trimmed sequences (without any annotation
except for the PHRED quality scores and anything encoded in the read names):

    >>> from Bio import SeqIO
    >>> for record in SeqIO.parse("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff-trim"):
    ...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    ...
    E3MFGYR02JWQ7T 260 GGTCTACATGTTGGTTAACC...
    E3MFGYR02JA6IL 265 TTTTTTTTGGAAAGGAAAAC...
    E3MFGYR02JHD4H 292 AAAGACAAGTGGTATCAACG...
    E3MFGYR02GFKUC 295 CGGCCGGGCCTCTCATCGGT...
    E3MFGYR02FTGED 277 TGGTAATGGGGGGAAATTTA...
    E3MFGYR02FR9G7 256 CTCCGTAAGAAGGTGCTGCC...
    E3MFGYR02GAZMS 271 AAAGAAGTAAGGTAAATAAC...
    E3MFGYR02HHZ8O 150 ACTTTCTTCTTTACCGTAAC...
    E3MFGYR02GPGB1 221 AAGCAGTGGTATCAACGCAG...
    E3MFGYR02F7Z7G 130 AATCATCCACTTTTTAACGT...

Looking at the final record in more detail, note how this differs to the
example above:

    >>> print("%s %i" % (record.id, len(record)))
    E3MFGYR02F7Z7G 130
    >>> print("%s..." % record.seq[:10])
    AATCATCCAC...
    >>> print("%r..." % record.letter_annotations["phred_quality"][:10])
    [26, 15, 12, 21, 28, 21, 36, 28, 27, 27]...
    >>> len(record.annotations)
    3
    >>> print(record.annotations["region"])
    2
    >>> print(record.annotations["coords"])
    (2434, 1658)
    >>> print(record.annotations["time"])
    [2008, 1, 9, 16, 16, 0]

You might use the Bio.SeqIO.convert() function to convert the (trimmed) SFF
reads into a FASTQ file (or a FASTA file and a QUAL file), e.g.

    >>> from Bio import SeqIO
    >>> try:
    ...     from StringIO import StringIO # Python 2
    ... except ImportError:
    ...     from io import StringIO # Python 3
    ...
```

```
    >>> out_handle = StringIO()
    >>> count = SeqIO.convert("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff",
    ...                       out_handle, "fastq")
    ...
    >>> print("Converted %i records" % count)
    Converted 10 records

The output FASTQ file would start like this:

    >>> print("%s..." % out_handle.getvalue()[:50])
    @E3MFGYR02JWQ7T
    tcagGGTCTACATGTTGGTTAACCCGTACTGATT...

Bio.SeqIO.index() provides memory efficient random access to the reads in an
SFF file by name. SFF files can include an index within the file, which can
be read in making this very fast. If the index is missing (or in a format not
yet supported in Biopython) the file is indexed by scanning all the reads -
which is a little slower. For example,

    >>> from Bio import SeqIO
    >>> reads = SeqIO.index("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff")
    >>> record = reads["E3MFGYR02JHD4H"]
    >>> print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    E3MFGYR02JHD4H 310 tcagAAAGACAAGTGGTATC...
    >>> reads.close()

Or, using the trimmed reads:

    >>> from Bio import SeqIO
    >>> reads = SeqIO.index("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff-trim")
    >>> record = reads["E3MFGYR02JHD4H"]
    >>> print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    E3MFGYR02JHD4H 292 AAAGACAAGTGGTATCAACG...
    >>> reads.close()

You can also use the Bio.SeqIO.write() function with the "sff" format. Note
that this requires all the flow information etc, and thus is probably only
useful for SeqRecord objects originally from reading another SFF file (and
not the trimmed SeqRecord objects from parsing an SFF file as "sff-trim").

As an example, let's pretend this example SFF file represents some DNA which
was pre-amplified with a PCR primers AAAGANNNNN. The following script would
produce a sub-file containing all those reads whose post-quality clipping
region (i.e. the sequence after trimming) starts with AAAGA exactly (the non-
degenerate bit of this pretend primer):

    >>> from Bio import SeqIO
    >>> records = (record for record in
    ...            SeqIO.parse("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff")
    ...            if record.seq[record.annotations["clip_qual_left"]:].startswith("AAAGA"))
    ...
    >>> count = SeqIO.write(records, "temp_filtered.sff", "sff")
    >>> print("Selected %i records" % count)
    Selected 2 records

Of course, for an assembly you would probably want to remove these primers.
If you want FASTA or FASTQ output, you could just slice the SeqRecord. However,
if you want SFF output we have to preserve all the flow information - the trick
is just to adjust the left clip position!

    >>> from Bio import SeqIO
    >>> def filter_and_trim(records, primer):
    ...     for record in records:
    ...         if record.seq[record.annotations["clip_qual_left"]:].startswith(primer):
    ...             record.annotations["clip_qual_left"] += len(primer)
    ...             yield record
    ...
    >>> records = SeqIO.parse("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff")
    >>> count = SeqIO.write(filter_and_trim(records, "AAAGA"),
    ...                     "temp_filtered.sff", "sff")
    ...
    >>> print("Selected %i records" % count)
    Selected 2 records

```

```
We can check the results, note the lower case clipped region now includes the "AAAGA"
sequence:

    >>> for record in SeqIO.parse("temp_filtered.sff", "sff"):
    ...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    ...
    E3MFGYR02JHD4H 310 tcagaaagaCAAGTGGTATC...
    E3MFGYR02GAZMS 278 tcagaaagaAGTAAGGTAAA...
    >>> for record in SeqIO.parse("temp_filtered.sff", "sff-trim"):
    ...     print("%s %i %s..." % (record.id, len(record), record.seq[:20]))
    ...
    E3MFGYR02JHD4H 287 CAAGTGGTATCAACGCAGAG...
    E3MFGYR02GAZMS 266 AGTAAGGTAAATAACAAACG...
    >>> import os
    >>> os.remove("temp_filtered.sff")

For a description of the file format, please see the Roche manuals and:
http://www.ncbi.nlm.nih.gov/Traces/trace.cgi?cmd=show&f=formats&m=doc&s=formats

"""

from __future__ import print_function

from Bio.SeqIO.Interfaces import SequenceWriter
from Bio import Alphabet
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
import struct
import sys
import re

from Bio._py3k import _bytes_to_string, _as_bytes
_null = b"\0"
_sff = b".sff"
_hsh = b".hsh"
_srt = b".srt"
_mft = b".mft"
_flag = b"\xff"


def _check_mode(handle):
    """Ensure handle not opened in text mode.

    Ensures mode is not set for Universal new line
    and ensures mode is binary for Windows
    """
    # TODO - Does this need to be stricter under Python 3?
    mode = ""
    if hasattr(handle, "mode"):
        mode = handle.mode
        if mode == 1:
            # gzip.open(...) does this, fine
            return
        mode = str(mode)

    if mode and "U" in mode.upper():
        raise ValueError("SFF files must NOT be opened in universal new "
                         "lines mode. Binary mode is recommended (although "
                         "on Unix the default mode is also fine).")
    elif mode and "B" not in mode.upper() \
            and sys.platform == "win32":
        raise ValueError("SFF files must be opened in binary mode on Windows")


def _sff_file_header(handle):
    """Read in an SFF file header (PRIVATE).

    Assumes the handle is at the start of the file, will read forwards
    though the header and leave the handle pointing at the first record.
    Returns a tuple of values from the header (header_length, index_offset,
    index_length, number_of_reads, flows_per_read, flow_chars, key_sequence)

    >>> with open("../../Tests/Roche/greek.sff", "rb") as handle:
    ...     values = _sff_file_header(handle)
```

```
    ...
    >>> print(values[0])
    840
    >>> print(values[1])
    65040
    >>> print(values[2])
    256
    >>> print(values[3])
    24
    >>> print(values[4])
    800
    >>> values[-1]
    'TCAG'

    """
    _check_mode(handle)
    # file header (part one)
    # use big endiean encdoing   >
    # magic_number               I
    # version                    4B
    # index_offset               Q
    # index_length               I
    # number_of_reads            I
    # header_length              H
    # key_length                 H
    # number_of_flows_per_read   H
    # flowgram_format_code       B
    # [rest of file header depends on the number of flows and how many keys]
    fmt = '>4s4BQIIHHHB'
    assert 31 == struct.calcsize(fmt)
    data = handle.read(31)
    if not data:
        raise ValueError("Empty file.")
    elif len(data) < 31:
        raise ValueError("File too small to hold a valid SFF header.")
    magic_number, ver0, ver1, ver2, ver3, index_offset, index_length, \
        number_of_reads, header_length, key_length, number_of_flows_per_read, \
        flowgram_format = struct.unpack(fmt, data)
    if magic_number in [_hsh, _srt, _mft]:
        # Probably user error, calling Bio.SeqIO.parse() twice!
        raise ValueError("Handle seems to be at SFF index block, not start")
    if magic_number != _sff:  # 779314790
        raise ValueError("SFF file did not start '.sff', but %r" % magic_number)
    if (ver0, ver1, ver2, ver3) != (0, 0, 0, 1):
        raise ValueError("Unsupported SFF version in header, %i.%i.%i.%i"
                         % (ver0, ver1, ver2, ver3))
    if flowgram_format != 1:
        raise ValueError("Flowgram format code %i not supported"
                         % flowgram_format)
    if (index_offset != 0) ^ (index_length != 0):
        raise ValueError("Index offset %i but index length %i"
                         % (index_offset, index_length))
    flow_chars = _bytes_to_string(handle.read(number_of_flows_per_read))
    key_sequence = _bytes_to_string(handle.read(key_length))
    # According to the spec, the header_length field should be the total number
    # of bytes required by this set of header fields, and should be equal to
    # "31 + number_of_flows_per_read + key_length" rounded up to the next value
    # divisible by 8.
    assert header_length % 8 == 0
    padding = header_length - number_of_flows_per_read - key_length - 31
    assert 0 <= padding < 8, padding
    if handle.read(padding).count(_null) != padding:
        import warnings
        from Bio import BiopythonParserWarning
        warnings.warn("Your SFF file is invalid, post header %i byte "
                      "null padding region contained data." % padding,
                      BiopythonParserWarning)
    return header_length, index_offset, index_length, \
        number_of_reads, number_of_flows_per_read, \
        flow_chars, key_sequence


def _sff_do_slow_index(handle):
    """Generates an index by scanning though all the reads in an SFF file (PRIVATE).
```

```
363
364     This is a slow but generic approach if we can't parse the provided index
365     (if present).
366
367     Will use the handle seek/tell functions.
368     """
369     handle.seek(0)
370     header_length, index_offset, index_length, number_of_reads, \
371         number_of_flows_per_read, flow_chars, key_sequence \
372         = _sff_file_header(handle)
373     # Now on to the reads...
374     read_header_fmt = '>2HI4H'
375     read_header_size = struct.calcsize(read_header_fmt)
376     # NOTE - assuming flowgram_format==1, which means struct type H
377     read_flow_fmt = ">%iH" % number_of_flows_per_read
378     read_flow_size = struct.calcsize(read_flow_fmt)
379     assert 1 == struct.calcsize(">B")
380     assert 1 == struct.calcsize(">s")
381     assert 1 == struct.calcsize(">c")
382     assert read_header_size % 8 == 0  # Important for padding calc later!
383     for read in range(number_of_reads):
384         record_offset = handle.tell()
385         if record_offset == index_offset:
386             # Found index block within reads, ignore it:
387             offset = index_offset + index_length
388             if offset % 8:
389                 offset += 8 - (offset % 8)
390             assert offset % 8 == 0
391             handle.seek(offset)
392             record_offset = offset
393         # assert record_offset%8 == 0 # Worth checking, but slow
394         # First the fixed header
395         data = handle.read(read_header_size)
396         read_header_length, name_length, seq_len, clip_qual_left, \
397             clip_qual_right, clip_adapter_left, clip_adapter_right \
398             = struct.unpack(read_header_fmt, data)
399         if read_header_length < 10 or read_header_length % 8 != 0:
400             raise ValueError("Malformed read header, says length is %i:\n%r"
401                              % (read_header_length, data))
402         # now the name and any padding (remainder of header)
403         name = _bytes_to_string(handle.read(name_length))
404         padding = read_header_length - read_header_size - name_length
405         if handle.read(padding).count(_null) != padding:
406             import warnings
407             from Bio import BiopythonParserWarning
408             warnings.warn("Your SFF file is invalid, post name %i byte "
409                           "padding region contained data" % padding,
410                           BiopythonParserWarning)
411         assert record_offset + read_header_length == handle.tell()
412         # now the flowgram values, flowgram index, bases and qualities
413         size = read_flow_size + 3 * seq_len
414         handle.seek(size, 1)
415         # now any padding...
416         padding = size % 8
417         if padding:
418             padding = 8 - padding
419             if handle.read(padding).count(_null) != padding:
420                 import warnings
421                 from Bio import BiopythonParserWarning
422                 warnings.warn("Your SFF file is invalid, post quality %i "
423                               "byte padding region contained data" % padding,
424                               BiopythonParserWarning)
425         # print("%s %s %i" % (read, name, record_offset))
426         yield name, record_offset
427     if handle.tell() % 8 != 0:
428         raise ValueError(
429             "After scanning reads, did not end on a multiple of 8")
430
431
432 def _sff_find_roche_index(handle):
433     """Locate any existing Roche style XML meta data and read index (PRIVATE).
434
435     Makes a number of hard coded assumptions based on reverse engineered SFF
436     files from Roche 454 machines.
```

```
437
438     Returns a tuple of read count, SFF "index" offset and size, XML offset
439     and size, and the actual read index offset and size.
440
441     Raises a ValueError for unsupported or non-Roche index blocks.
442     """
443     handle.seek(0)
444     header_length, index_offset, index_length, number_of_reads, \
445         number_of_flows_per_read, flow_chars, key_sequence \
446         = _sff_file_header(handle)
447     assert handle.tell() == header_length
448     if not index_offset or not index_offset:
449         raise ValueError("No index present in this SFF file")
450     # Now jump to the header...
451     handle.seek(index_offset)
452     fmt = ">4s4B"
453     fmt_size = struct.calcsize(fmt)
454     data = handle.read(fmt_size)
455     if not data:
456         raise ValueError("Premature end of file? Expected index of size %i at offest %i, found nothing"
457                          % (index_length, index_offset))
458     if len(data) < fmt_size:
459         raise ValueError("Premature end of file? Expected index of size %i at offest %i, found %r"
460                          % (index_length, index_offset, data))
461     magic_number, ver0, ver1, ver2, ver3 = struct.unpack(fmt, data)
462     if magic_number == _mft:  # 778921588
463         # Roche 454 manifest index
464         # This is typical from raw Roche 454 SFF files (2009), and includes
465         # both an XML manifest and the sorted index.
466         if (ver0, ver1, ver2, ver3) != (49, 46, 48, 48):
467             # This is "1.00" as a string
468             raise ValueError("Unsupported version in .mft index header, %i.%i.%i.%i"
469                              % (ver0, ver1, ver2, ver3))
470         fmt2 = ">LL"
471         fmt2_size = struct.calcsize(fmt2)
472         xml_size, data_size = struct.unpack(fmt2, handle.read(fmt2_size))
473         if index_length != fmt_size + fmt2_size + xml_size + data_size:
474             raise ValueError("Problem understanding .mft index header, %i != %i + %i + %i + %i"
475                              % (index_length, fmt_size, fmt2_size, xml_size, data_size))
476         return number_of_reads, header_length, \
477             index_offset, index_length, \
478             index_offset + fmt_size + fmt2_size, xml_size, \
479             index_offset + fmt_size + fmt2_size + xml_size, data_size
480     elif magic_number == _srt:  # 779317876
481         # Roche 454 sorted index
482         # I've had this from Roche tool sfffile when the read identifiers
483         # had nonstandard lengths and there was no XML manifest.
484         if (ver0, ver1, ver2, ver3) != (49, 46, 48, 48):
485             # This is "1.00" as a string
486             raise ValueError("Unsupported version in .srt index header, %i.%i.%i.%i"
487                              % (ver0, ver1, ver2, ver3))
488         data = handle.read(4)
489         if data != _null * 4:
490             raise ValueError(
491                 "Did not find expected null four bytes in .srt index")
492         return number_of_reads, header_length, \
493             index_offset, index_length, \
494             0, 0, \
495             index_offset + fmt_size + 4, index_length - fmt_size - 4
496     elif magic_number == _hsh:
497         raise ValueError("Hash table style indexes (.hsh) in SFF files are "
498                          "not (yet) supported")
499     else:
500         raise ValueError("Unknown magic number %r in SFF index header:\n%r"
501                          % (magic_number, data))
502
503
504 def ReadRocheXmlManifest(handle):
505     """Reads any Roche style XML manifest data in the SFF "index".
506
507     The SFF file format allows for multiple different index blocks, and Roche
508     took advantage of this to define their own index block which also embeds
509     an XML manifest string. This is not a publicly documented extension to
510     the SFF file format, this was reverse engineered.
```

```
511
512     The handle should be to an SFF file opened in binary mode. This function
513     will use the handle seek/tell functions and leave the handle in an
514     arbitrary location.
515
516     Any XML manifest found is returned as a Python string, which you can then
517     parse as appropriate, or reuse when writing out SFF files with the
518     SffWriter class.
519
520     Returns a string, or raises a ValueError if an Roche manifest could not be
521     found.
522     """
523     number_of_reads, header_length, index_offset, index_length, xml_offset, \
524         xml_size, read_index_offset, read_index_size = _sff_find_roche_index(
525             handle)
526     if not xml_offset or not xml_size:
527         raise ValueError("No XML manifest found")
528     handle.seek(xml_offset)
529     return _bytes_to_string(handle.read(xml_size))
530
531
532 # This is a generator function!
533 def _sff_read_roche_index(handle):
534     """Reads any existing Roche style read index provided in the SFF file (PRIVATE).
535
536     Will use the handle seek/tell functions.
537
538     This works on ".srt1.00" and ".mft1.00" style Roche SFF index blocks.
539
540     Roche SFF indices use base 255 not 256, meaning we see bytes in range the
541     range 0 to 254 only. This appears to be so that byte 0xFF (character 255)
542     can be used as a marker character to separate entries (required if the
543     read name lengths vary).
544
545     Note that since only four bytes are used for the read offset, this is
546     limited to 255^4 bytes (nearly 4GB). If you try to use the Roche sfffile
547     tool to combine SFF files beyound this limit, they issue a warning and
548     omit the index (and manifest).
549     """
550     number_of_reads, header_length, index_offset, index_length, xml_offset, \
551         xml_size, read_index_offset, read_index_size = _sff_find_roche_index(
552             handle)
553     # Now parse the read index...
554     handle.seek(read_index_offset)
555     fmt = ">5B"
556     for read in range(number_of_reads):
557         # TODO - Be more aware of when the index should end?
558         data = handle.read(6)
559         while True:
560             more = handle.read(1)
561             if not more:
562                 raise ValueError("Premature end of file!")
563             data += more
564             if more == _flag:
565                 break
566         assert data[-1:] == _flag, data[-1:]
567         name = _bytes_to_string(data[:-6])
568         off4, off3, off2, off1, off0 = struct.unpack(fmt, data[-6:-1])
569         offset = off0 + 255 * off1 + 65025 * off2 + 16581375 * off3
570         if off4:
571             # Could in theory be used as a fifth piece of offset information,
572             # i.e. offset =+ 4228250625L*off4, but testing the Roche tools this
573             # is not the case. They simple don't support such large indexes.
574             raise ValueError("Expected a null terminator to the read name.")
575         yield name, offset
576     if handle.tell() != read_index_offset + read_index_size:
577         raise ValueError("Problem with index length? %i vs %i"
578                          % (handle.tell(), read_index_offset + read_index_size))
579
580 _valid_UAN_read_name = re.compile(r'^[a-zA-Z0-9]{14}$')
581
582
583 def _sff_read_seq_record(handle, number_of_flows_per_read, flow_chars,
584                          key_sequence, alphabet, trim=False):
```

```
585     """Parse the next read in the file, return data as a SeqRecord (PRIVATE)."""
586     # Now on to the reads...
587     # the read header format (fixed part):
588     # read_header_length     H
589     # name_length            H
590     # seq_len                I
591     # clip_qual_left         H
592     # clip_qual_right        H
593     # clip_adapter_left      H
594     # clip_adapter_right     H
595     # [rest of read header depends on the name length etc]
596     read_header_fmt = '>2HI4H'
597     read_header_size = struct.calcsize(read_header_fmt)
598     read_flow_fmt = ">%iH" % number_of_flows_per_read
599     read_flow_size = struct.calcsize(read_flow_fmt)
600 
601     read_header_length, name_length, seq_len, clip_qual_left, \
602         clip_qual_right, clip_adapter_left, clip_adapter_right \
603         = struct.unpack(read_header_fmt, handle.read(read_header_size))
604     if clip_qual_left:
605         clip_qual_left -= 1  # python counting
606     if clip_adapter_left:
607         clip_adapter_left -= 1  # python counting
608     if read_header_length < 10 or read_header_length % 8 != 0:
609         raise ValueError("Malformed read header, says length is %i"
610                          % read_header_length)
611     # now the name and any padding (remainder of header)
612     name = _bytes_to_string(handle.read(name_length))
613     padding = read_header_length - read_header_size - name_length
614     if handle.read(padding).count(_null) != padding:
615         import warnings
616         from Bio import BiopythonParserWarning
617         warnings.warn("Your SFF file is invalid, post name %i "
618                       "byte padding region contained data" % padding,
619                       BiopythonParserWarning)
620     # now the flowgram values, flowgram index, bases and qualities
621     # NOTE - assuming flowgram_format==1, which means struct type H
622     flow_values = handle.read(read_flow_size)  # unpack later if needed
623     temp_fmt = ">%iB" % seq_len  # used for flow index and quals
624     flow_index = handle.read(seq_len)  # unpack later if needed
625     seq = _bytes_to_string(handle.read(seq_len))  # TODO - Use bytes in Seq?
626     quals = list(struct.unpack(temp_fmt, handle.read(seq_len)))
627     # now any padding...
628     padding = (read_flow_size + seq_len * 3) % 8
629     if padding:
630         padding = 8 - padding
631         if handle.read(padding).count(_null) != padding:
632             import warnings
633             from Bio import BiopythonParserWarning
634             warnings.warn("Your SFF file is invalid, post quality %i "
635                           "byte padding region contained data" % padding,
636                           BiopythonParserWarning)
637     # Follow Roche and apply most aggressive of qual and adapter clipping.
638     # Note Roche seems to ignore adapter clip fields when writing SFF,
639     # and uses just the quality clipping values for any clipping.
640     clip_left = max(clip_qual_left, clip_adapter_left)
641     # Right clipping of zero means no clipping
642     if clip_qual_right:
643         if clip_adapter_right:
644             clip_right = min(clip_qual_right, clip_adapter_right)
645         else:
646             # Typical case with Roche SFF files
647             clip_right = clip_qual_right
648     elif clip_adapter_right:
649         clip_right = clip_adapter_right
650     else:
651         clip_right = seq_len
652     # Now build a SeqRecord
653     if trim:
654         if clip_left >= clip_right:
655             # Raise an error?
656             import warnings
657             from Bio import BiopythonParserWarning
658             warnings.warn("Overlapping clip values in SFF record, trimmed to nothing",
```

```
659                             BiopythonParserWarning)
660             seq = ""
661             quals = []
662         else:
663             seq = seq[clip_left:clip_right].upper()
664             quals = quals[clip_left:clip_right]
665         # Don't record the clipping values, flow etc, they make no sense now:
666         annotations = {}
667     else:
668         if clip_left >= clip_right:
669             import warnings
670             from Bio import BiopythonParserWarning
671             warnings.warn("Overlapping clip values in SFF record", BiopythonParserWarning)
672             seq = seq.lower()
673         else:
674             # This use of mixed case mimics the Roche SFF tool's FASTA output
675             seq = seq[:clip_left].lower() + \
676                 seq[clip_left:clip_right].upper() + \
677                 seq[clip_right:].lower()
678         annotations = {"flow_values": struct.unpack(read_flow_fmt, flow_values),
679                        "flow_index": struct.unpack(temp_fmt, flow_index),
680                        "flow_chars": flow_chars,
681                        "flow_key": key_sequence,
682                        "clip_qual_left": clip_qual_left,
683                        "clip_qual_right": clip_qual_right,
684                        "clip_adapter_left": clip_adapter_left,
685                        "clip_adapter_right": clip_adapter_right}
686     if re.match(_valid_UAN_read_name, name):
687         annotations["time"] = _get_read_time(name)
688         annotations["region"] = _get_read_region(name)
689         annotations["coords"] = _get_read_xy(name)
690     record = SeqRecord(Seq(seq, alphabet),
691                        id=name,
692                        name=name,
693                        description="",
694                        annotations=annotations)
695     # Dirty trick to speed up this line:
696     # record.letter_annotations["phred_quality"] = quals
697     dict.__setitem__(record._per_letter_annotations,
698                      "phred_quality", quals)
699     # Return the record and then continue...
700     return record
701
702 _powers_of_36 = [36 ** i for i in range(6)]
703
704
705 def _string_as_base_36(string):
706     """Interpret a string as a base-36 number as per 454 manual."""
707     total = 0
708     for c, power in zip(string[::-1], _powers_of_36):
709         # For reference: ord('0') = 48, ord('9') = 57
710         # For reference: ord('A') = 65, ord('Z') = 90
711         # For reference: ord('a') = 97, ord('z') = 122
712         if 48 <= ord(c) <= 57:
713             val = ord(c) - 22  # equivalent to: - ord('0') + 26
714         elif 65 <= ord(c) <= 90:
715             val = ord(c) - 65
716         elif 97 <= ord(c) <= 122:
717             val = ord(c) - 97
718         else:
719             # Invalid character
720             val = 0
721         total += val * power
722     return total
723
724
725 def _get_read_xy(read_name):
726     """Extract coordinates from last 5 characters of read name."""
727     number = _string_as_base_36(read_name[9:])
728     return divmod(number, 4096)
729
730 _time_denominators = [13 * 32 * 24 * 60 * 60,
731                       32 * 24 * 60 * 60,
732                       24 * 60 * 60,
```

```
733                     60 * 60,
734                     60]
735
736
737 def _get_read_time(read_name):
738     """Extract time from first 6 characters of read name."""
739     time_list = []
740     remainder = _string_as_base_36(read_name[:6])
741     for denominator in _time_denominators:
742         this_term, remainder = divmod(remainder, denominator)
743         time_list.append(this_term)
744     time_list.append(remainder)
745     time_list[0] += 2000
746     return time_list
747
748
749 def _get_read_region(read_name):
750     """Extract region from read name."""
751     return int(read_name[8])
752
753
754 def _sff_read_raw_record(handle, number_of_flows_per_read):
755     """Extract the next read in the file as a raw (bytes) string (PRIVATE)."""
756     read_header_fmt = '>2HI'
757     read_header_size = struct.calcsize(read_header_fmt)
758     read_flow_fmt = ">%iH" % number_of_flows_per_read
759     read_flow_size = struct.calcsize(read_flow_fmt)
760
761     raw = handle.read(read_header_size)
762     read_header_length, name_length, seq_len \
763         = struct.unpack(read_header_fmt, raw)
764     if read_header_length < 10 or read_header_length % 8 != 0:
765         raise ValueError("Malformed read header, says length is %i"
766                          % read_header_length)
767     # now the four clip values (4H = 8 bytes), and read name
768     raw += handle.read(8 + name_length)
769     # and any padding (remainder of header)
770     padding = read_header_length - read_header_size - 8 - name_length
771     pad = handle.read(padding)
772     if pad.count(_null) != padding:
773         import warnings
774         from Bio import BiopythonParserWarning
775         warnings.warn("Your SFF file is invalid, post name %i "
776                       "byte padding region contained data" % padding,
777                       BiopythonParserWarning)
778     raw += pad
779     # now the flowgram values, flowgram index, bases and qualities
780     raw += handle.read(read_flow_size + seq_len * 3)
781     padding = (read_flow_size + seq_len * 3) % 8
782     # now any padding...
783     if padding:
784         padding = 8 - padding
785         pad = handle.read(padding)
786         if pad.count(_null) != padding:
787             import warnings
788             from Bio import BiopythonParserWarning
789             warnings.warn("Your SFF file is invalid, post quality %i "
790                           "byte padding region contained data" % padding,
791                           BiopythonParserWarning)
792         raw += pad
793     # Return the raw bytes
794     return raw
795
796
797 class _AddTellHandle(object):
798     """Wrapper for handles which do not support the tell method (PRIVATE).
799
800     Intended for use with things like network handles where tell (and reverse
801     seek) are not supported. The SFF file needs to track the current offset in
802     order to deal with the index block.
803     """
804     def __init__(self, handle):
805         self._handle = handle
806         self._offset = 0
```

```
807
808     def read(self, length):
809         data = self._handle.read(length)
810         self._offset += len(data)
811         return data
812
813     def tell(self):
814         return self._offset
815
816     def seek(self, offset):
817         if offset < self._offset:
818             raise RuntimeError("Can't seek backwards")
819         self._handle.read(offset - self._offset)
820
821     def close(self):
822         return self._handle.close()
823
824
825 # This is a generator function!
826 def SffIterator(handle, alphabet=Alphabet.generic_dna, trim=False):
827     """Iterate over Standard Flowgram Format (SFF) reads (as SeqRecord objects).
828
829         - handle - input file, an SFF file, e.g. from Roche 454 sequencing.
830           This must NOT be opened in universal read lines mode!
831         - alphabet - optional alphabet, defaults to generic DNA.
832         - trim - should the sequences be trimmed?
833
834     The resulting SeqRecord objects should match those from a paired FASTA
835     and QUAL file converted from the SFF file using the Roche 454 tool
836     ssfinfo. i.e. The sequence will be mixed case, with the trim regions
837     shown in lower case.
838
839     This function is used internally via the Bio.SeqIO functions:
840
841     >>> from Bio import SeqIO
842     >>> for record in SeqIO.parse("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "sff"):
843     ...     print("%s %i" % (record.id, len(record)))
844     ...
845     E3MFGYR02JWQ7T 265
846     E3MFGYR02JA6IL 271
847     E3MFGYR02JHD4H 310
848     E3MFGYR02GFKUC 299
849     E3MFGYR02FTGED 281
850     E3MFGYR02FR9G7 261
851     E3MFGYR02GAZMS 278
852     E3MFGYR02HHZ8O 221
853     E3MFGYR02GPGB1 269
854     E3MFGYR02F7Z7G 219
855
856     You can also call it directly:
857
858     >>> with open("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "rb") as handle:
859     ...     for record in SffIterator(handle):
860     ...         print("%s %i" % (record.id, len(record)))
861     ...
862     E3MFGYR02JWQ7T 265
863     E3MFGYR02JA6IL 271
864     E3MFGYR02JHD4H 310
865     E3MFGYR02GFKUC 299
866     E3MFGYR02FTGED 281
867     E3MFGYR02FR9G7 261
868     E3MFGYR02GAZMS 278
869     E3MFGYR02HHZ8O 221
870     E3MFGYR02GPGB1 269
871     E3MFGYR02F7Z7G 219
872
873     Or, with the trim option:
874
875     >>> with open("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "rb") as handle:
876     ...     for record in SffIterator(handle, trim=True):
877     ...         print("%s %i" % (record.id, len(record)))
878     ...
879     E3MFGYR02JWQ7T 260
880     E3MFGYR02JA6IL 265
```

```
    E3MFGYR02JHD4H 292
    E3MFGYR02GFKUC 295
    E3MFGYR02FTGED 277
    E3MFGYR02FR9G7 256
    E3MFGYR02GAZMS 271
    E3MFGYR02HHZ8O 150
    E3MFGYR02GPGB1 221
    E3MFGYR02F7Z7G 130

    """
    if isinstance(Alphabet._get_base_alphabet(alphabet),
                  Alphabet.ProteinAlphabet):
        raise ValueError("Invalid alphabet, SFF files do not hold proteins.")
    if isinstance(Alphabet._get_base_alphabet(alphabet),
                  Alphabet.RNAAlphabet):
        raise ValueError("Invalid alphabet, SFF files do not hold RNA.")
    try:
        assert 0 == handle.tell(), "Not at start of file, offset %i" % handle.tell()
    except AttributeError:
        # Probably a network handle or something like that
        handle = _AddTellHandle(handle)
    header_length, index_offset, index_length, number_of_reads, \
        number_of_flows_per_read, flow_chars, key_sequence \
        = _sff_file_header(handle)
    # Now on to the reads...
    # the read header format (fixed part):
    # read_header_length     H
    # name_length            H
    # seq_len                I
    # clip_qual_left         H
    # clip_qual_right        H
    # clip_adapter_left      H
    # clip_adapter_right     H
    # [rest of read header depends on the name length etc]
    read_header_fmt = '>2HI4H'
    read_header_size = struct.calcsize(read_header_fmt)
    read_flow_fmt = ">%iH" % number_of_flows_per_read
    read_flow_size = struct.calcsize(read_flow_fmt)
    assert 1 == struct.calcsize(">B")
    assert 1 == struct.calcsize(">s")
    assert 1 == struct.calcsize(">c")
    assert read_header_size % 8 == 0  # Important for padding calc later!
    # The spec allows for the index block to be before or even in the middle
    # of the reads. We can check that if we keep track of our position
    # in the file...
    for read in range(number_of_reads):
        if index_offset and handle.tell() == index_offset:
            offset = index_offset + index_length
            if offset % 8:
                offset += 8 - (offset % 8)
            assert offset % 8 == 0
            handle.seek(offset)
            # Now that we've done this, we don't need to do it again. Clear
            # the index_offset so we can skip extra handle.tell() calls:
            index_offset = 0
        yield _sff_read_seq_record(handle,
                                   number_of_flows_per_read,
                                   flow_chars,
                                   key_sequence,
                                   alphabet,
                                   trim)
    _check_eof(handle, index_offset, index_length)


def _check_eof(handle, index_offset, index_length):
    """Check final padding is OK (8 byte alignment) and file ends (PRIVATE).

    Will attempt to spot apparent SFF file concatenation and give an error.

    Will not attempt to seek, only moves the handle forward.
    """
    offset = handle.tell()
    extra = b""
    padding = 0
```

```
955
956     if index_offset and offset <= index_offset:
957         # Index block then end of file...
958         if offset < index_offset:
959             raise ValueError("Gap of %i bytes after final record end %i, "
960                              "before %i where index starts?"
961                              % (index_offset - offset, offset, index_offset))
962         # Doing read to jump the index rather than a seek
963         # in case this is a network handle or similar
964         handle.read(index_offset + index_length - offset)
965         offset = index_offset + index_length
966         assert offset == handle.tell(), \
967             "Wanted %i, got %i, index is %i to %i" \
968             % (offset, handle.tell(), index_offset, index_offset + index_length)
969
970     if offset % 8:
971         padding = 8 - (offset % 8)
972         extra = handle.read(padding)
973
974     if padding >= 4 and extra[-4:] == _sff:
975         # Seen this in one user supplied file, should have been
976         # four bytes of null padding but was actually .sff and
977         # the start of a new concatenated SFF file!
978         raise ValueError("Your SFF file is invalid, post index %i byte "
979                          "null padding region ended '.sff' which could "
980                          "be the start of a concatenated SFF file? "
981                          "See offset %i" % (padding, offset))
982     if padding and not extra:
983         # TODO - Is this error harmless enough to just ignore?
984         import warnings
985         from Bio import BiopythonParserWarning
986         warnings.warn("Your SFF file is technically invalid as it is missing "
987                       "a terminal %i byte null padding region." % padding,
988                       BiopythonParserWarning)
989         return
990     if extra.count(_null) != padding:
991         import warnings
992         from Bio import BiopythonParserWarning
993         warnings.warn("Your SFF file is invalid, post index %i byte "
994                       "null padding region contained data: %r"
995                       % (padding, extra), BiopythonParserWarning)
996
997     offset = handle.tell()
998     assert offset % 8 == 0, \
999         "Wanted offset %i %% 8 = %i to be zero" % (offset, offset % 8)
1000    # Should now be at the end of the file...
1001    extra = handle.read(4)
1002    if extra == _sff:
1003        raise ValueError("Additional data at end of SFF file, "
1004                         "perhaps multiple SFF files concatenated? "
1005                         "See offset %i" % offset)
1006    elif extra:
1007        raise ValueError("Additional data at end of SFF file, "
1008                         "see offset %i" % offset)
1009
1010
1011 # This is a generator function!
1012 def _SffTrimIterator(handle, alphabet=Alphabet.generic_dna):
1013     """Iterate over SFF reads (as SeqRecord objects) with trimming (PRIVATE)."""
1014     return SffIterator(handle, alphabet, trim=True)
1015
1016
1017 class SffWriter(SequenceWriter):
1018     """SFF file writer."""
1019
1020     def __init__(self, handle, index=True, xml=None):
1021         """Creates the writer object.
1022
1023             - handle - Output handle, ideally in binary write mode.
1024             - index - Boolean argument, should we try and write an index?
1025             - xml - Optional string argument, xml manifest to be recorded in the index
1026               block (see function ReadRocheXmlManifest for reading this data).
1027         """
1028         _check_mode(handle)
```

```
1029            self.handle = handle
1030            self._xml = xml
1031            if index:
1032                self._index = []
1033            else:
1034                self._index = None
1035
1036        def write_file(self, records):
1037            """Use this to write an entire file containing the given records."""
1038            try:
1039                self._number_of_reads = len(records)
1040            except TypeError:
1041                self._number_of_reads = 0  # dummy value
1042                if not hasattr(self.handle, "seek") \
1043                        or not hasattr(self.handle, "tell"):
1044                    raise ValueError("A handle with a seek/tell methods is "
1045                                     "required in order to record the total "
1046                                     "record count in the file header (once it "
1047                                     "is known at the end).")
1048            if self._index is not None and \
1049                    not (hasattr(self.handle, "seek") and hasattr(self.handle, "tell")):
1050                import warnings
1051                warnings.warn("A handle with a seek/tell methods is required in "
1052                              "order to record an SFF index.")
1053                self._index = None
1054            self._index_start = 0
1055            self._index_length = 0
1056            if not hasattr(records, "next"):
1057                records = iter(records)
1058            # Get the first record in order to find the flow information
1059            # we will need for the header.
1060            try:
1061                record = next(records)
1062            except StopIteration:
1063                record = None
1064            if record is None:
1065                # No records -> empty SFF file (or an error)?
1066                # We can't write a header without the flow information.
1067                # return 0
1068                raise ValueError("Must have at least one sequence")
1069            try:
1070                self._key_sequence = _as_bytes(record.annotations["flow_key"])
1071                self._flow_chars = _as_bytes(record.annotations["flow_chars"])
1072                self._number_of_flows_per_read = len(self._flow_chars)
1073            except KeyError:
1074                raise ValueError("Missing SFF flow information")
1075            self.write_header()
1076            self.write_record(record)
1077            count = 1
1078            for record in records:
1079                self.write_record(record)
1080                count += 1
1081            if self._number_of_reads == 0:
1082                # Must go back and record the record count...
1083                offset = self.handle.tell()
1084                self.handle.seek(0)
1085                self._number_of_reads = count
1086                self.write_header()
1087                self.handle.seek(offset)  # not essential?
1088            else:
1089                assert count == self._number_of_reads
1090            if self._index is not None:
1091                self._write_index()
1092            return count
1093
1094        def _write_index(self):
1095            assert len(self._index) == self._number_of_reads
1096            handle = self.handle
1097            self._index.sort()
1098            self._index_start = handle.tell()  # need for header
1099            # XML...
1100            if self._xml is not None:
1101                xml = _as_bytes(self._xml)
1102            else:
```

```
1103            from Bio import __version__
1104            xml = "<!-- This file was output with Biopython %s -->\n" % __version__
1105            xml += "<!-- This XML and index block attempts to mimic Roche SFF files -->\n"
1106            xml += "<!-- This file may be a combination of multiple SFF files etc -->\n"
1107            xml = _as_bytes(xml)
1108        xml_len = len(xml)
1109        # Write to the file...
1110        fmt = ">I4BLL"
1111        fmt_size = struct.calcsize(fmt)
1112        handle.write(_null * fmt_size + xml)  # fill this later
1113        fmt2 = ">6B"
1114        assert 6 == struct.calcsize(fmt2)
1115        self._index.sort()
1116        index_len = 0  # don't know yet!
1117        for name, offset in self._index:
1118            # Roche files record the offsets using base 255 not 256.
1119            # See comments for parsing the index block. There may be a faster
1120            # way to code this, but we can't easily use shifts due to odd base
1121            off3 = offset
1122            off0 = off3 % 255
1123            off3 -= off0
1124            off1 = off3 % 65025
1125            off3 -= off1
1126            off2 = off3 % 16581375
1127            off3 -= off2
1128            assert offset == off0 + off1 + off2 + off3, \
1129                "%i -> %i %i %i %i" % (offset, off0, off1, off2, off3)
1130            off3, off2, off1, off0 = off3 // 16581375, off2 // 65025, \
1131                off1 // 255, off0
1132            assert off0 < 255 and off1 < 255 and off2 < 255 and off3 < 255, \
1133                "%i -> %i %i %i %i" % (offset, off0, off1, off2, off3)
1134            handle.write(name + struct.pack(fmt2, 0,
1135                                            off3, off2, off1, off0, 255))
1136            index_len += len(name) + 6
1137        # Note any padding in not included:
1138        self._index_length = fmt_size + xml_len + index_len  # need for header
1139        # Pad out to an 8 byte boundary (although I have noticed some
1140        # real Roche SFF files neglect to do this depsite their manual
1141        # suggesting this padding should be there):
1142        if self._index_length % 8:
1143            padding = 8 - (self._index_length % 8)
1144            handle.write(_null * padding)
1145        else:
1146            padding = 0
1147        offset = handle.tell()
1148        assert offset == self._index_start + self._index_length + padding, \
1149            "%i vs %i + %i + %i" % (offset, self._index_start,
1150                                    self._index_length, padding)
1151        # Must now go back and update the index header with index size...
1152        handle.seek(self._index_start)
1153        handle.write(struct.pack(fmt, 778921588,  # magic number
1154                                 49, 46, 48, 48,  # Roche index version, "1.00"
1155                                 xml_len, index_len) + xml)
1156        # Must now go back and update the header...
1157        handle.seek(0)
1158        self.write_header()
1159        handle.seek(offset)  # not essential?
1160
1161    def write_header(self):
1162        # Do header...
1163        key_length = len(self._key_sequence)
1164        # file header (part one)
1165        # use big endiean encdoing   >
1166        # magic_number               I
1167        # version                    4B
1168        # index_offset               Q
1169        # index_length               I
1170        # number_of_reads            I
1171        # header_length              H
1172        # key_length                 H
1173        # number_of_flows_per_read   H
1174        # flowgram_format_code       B
1175        # [rest of file header depends on the number of flows and how many keys]
1176        fmt = '>I4BQIIHHHB%is%is' % (
```

```
1177            self._number_of_flows_per_read, key_length)
1178        # According to the spec, the header_length field should be the total
1179        # number of bytes required by this set of header fields, and should be
1180        # equal to "31 + number_of_flows_per_read + key_length" rounded up to
1181        # the next value divisible by 8.
1182        if struct.calcsize(fmt) % 8 == 0:
1183            padding = 0
1184        else:
1185            padding = 8 - (struct.calcsize(fmt) % 8)
1186        header_length = struct.calcsize(fmt) + padding
1187        assert header_length % 8 == 0
1188        header = struct.pack(fmt, 779314790,  # magic number 0x2E736666
1189                             0, 0, 0, 1,  # version
1190                             self._index_start, self._index_length,
1191                             self._number_of_reads,
1192                             header_length, key_length,
1193                             self._number_of_flows_per_read,
1194                             1,  # the only flowgram format code we support
1195                             self._flow_chars, self._key_sequence)
1196        self.handle.write(header + _null * padding)
1197
1198    def write_record(self, record):
1199        """Write a single additional record to the output file.
1200
1201        This assumes the header has been done.
1202        """
1203        # Basics
1204        name = _as_bytes(record.id)
1205        name_len = len(name)
1206        seq = _as_bytes(str(record.seq).upper())
1207        seq_len = len(seq)
1208        # Qualities
1209        try:
1210            quals = record.letter_annotations["phred_quality"]
1211        except KeyError:
1212            raise ValueError("Missing PHRED qualities information for %s" % record.id)
1213        # Flow
1214        try:
1215            flow_values = record.annotations["flow_values"]
1216            flow_index = record.annotations["flow_index"]
1217            if self._key_sequence != _as_bytes(record.annotations["flow_key"]) \
1218                    or self._flow_chars != _as_bytes(record.annotations["flow_chars"]):
1219                raise ValueError("Records have inconsistent SFF flow data")
1220        except KeyError:
1221            raise ValueError("Missing SFF flow information for %s" % record.id)
1222        except AttributeError:
1223            raise ValueError("Header not written yet?")
1224        # Clipping
1225        try:
1226            clip_qual_left = record.annotations["clip_qual_left"]
1227            if clip_qual_left < 0:
1228                raise ValueError("Negative SFF clip_qual_left value for %s" % record.id)
1229            if clip_qual_left:
1230                clip_qual_left += 1
1231            clip_qual_right = record.annotations["clip_qual_right"]
1232            if clip_qual_right < 0:
1233                raise ValueError("Negative SFF clip_qual_right value for %s" % record.id)
1234            clip_adapter_left = record.annotations["clip_adapter_left"]
1235            if clip_adapter_left < 0:
1236                raise ValueError("Negative SFF clip_adapter_left value for %s" % record.id)
1237            if clip_adapter_left:
1238                clip_adapter_left += 1
1239            clip_adapter_right = record.annotations["clip_adapter_right"]
1240            if clip_adapter_right < 0:
1241                raise ValueError("Negative SFF clip_adapter_right value for %s" % record.id)
1242        except KeyError:
1243            raise ValueError("Missing SFF clipping information for %s" % record.id)
1244
1245        # Capture information for index
1246        if self._index is not None:
1247            offset = self.handle.tell()
1248            # Check the position of the final record (before sort by name)
1249            # Using a four-digit base 255 number, so the upper bound is
1250            # 254*(1)+254*(255)+254*(255**2)+254*(255**3) = 4228250624
```

```
1251            # or equivalently it overflows at 255**4 = 4228250625
1252            if offset > 4228250624:
1253                import warnings
1254                warnings.warn("Read %s has file offset %i, which is too large "
1255                              "to store in the Roche SFF index structure. No "
1256                              "index block will be recorded." % (name, offset))
1257                # No point recoring the offsets now
1258                self._index = None
1259            else:
1260                self._index.append((name, self.handle.tell()))
1261
1262        # the read header format (fixed part):
1263        # read_header_length     H
1264        # name_length            H
1265        # seq_len                I
1266        # clip_qual_left         H
1267        # clip_qual_right        H
1268        # clip_adapter_left      H
1269        # clip_adapter_right     H
1270        # [rest of read header depends on the name length etc]
1271        # name
1272        # flow values
1273        # flow index
1274        # sequence
1275        # padding
1276        read_header_fmt = '>2HI4H%is' % name_len
1277        if struct.calcsize(read_header_fmt) % 8 == 0:
1278            padding = 0
1279        else:
1280            padding = 8 - (struct.calcsize(read_header_fmt) % 8)
1281        read_header_length = struct.calcsize(read_header_fmt) + padding
1282        assert read_header_length % 8 == 0
1283        data = struct.pack(read_header_fmt,
1284                           read_header_length,
1285                           name_len, seq_len,
1286                           clip_qual_left, clip_qual_right,
1287                           clip_adapter_left, clip_adapter_right,
1288                           name) + _null * padding
1289        assert len(data) == read_header_length
1290        # now the flowgram values, flowgram index, bases and qualities
1291        # NOTE - assuming flowgram_format==1, which means struct type H
1292        read_flow_fmt = ">%iH" % self._number_of_flows_per_read
1293        read_flow_size = struct.calcsize(read_flow_fmt)
1294        temp_fmt = ">%iB" % seq_len  # used for flow index and quals
1295        data += struct.pack(read_flow_fmt, *flow_values) \
1296            + struct.pack(temp_fmt, *flow_index) \
1297            + seq \
1298            + struct.pack(temp_fmt, *quals)
1299        # now any final padding...
1300        padding = (read_flow_size + seq_len * 3) % 8
1301        if padding:
1302            padding = 8 - padding
1303        self.handle.write(data + _null * padding)
1304
1305
1306 if __name__ == "__main__":
1307     print("Running quick self test")
1308     filename = "../../Tests/Roche/E3MFGYR02_random_10_reads.sff"
1309     with open(filename, "rb") as handle:
1310         metadata = ReadRocheXmlManifest(handle)
1311
1312     from io import BytesIO
1313
1314     with open(filename, "rb") as handle:
1315         sff = list(SffIterator(handle))
1316     with open(filename, "rb") as handle:
1317         sff_trim = list(SffIterator(handle, trim=True))
1318
1319     from Bio import SeqIO
1320     filename = "../../Tests/Roche/E3MFGYR02_random_10_reads_no_trim.fasta"
1321     fasta_no_trim = list(SeqIO.parse(filename, "fasta"))
1322     filename = "../../Tests/Roche/E3MFGYR02_random_10_reads_no_trim.qual"
1323     qual_no_trim = list(SeqIO.parse(filename, "qual"))
1324
```

```
    filename = "../../Tests/Roche/E3MFGYR02_random_10_reads.fasta"
    fasta_trim = list(SeqIO.parse(filename, "fasta"))
    filename = "../../Tests/Roche/E3MFGYR02_random_10_reads.qual"
    qual_trim = list(SeqIO.parse(filename, "qual"))

    for s, sT, f, q, fT, qT in zip(sff, sff_trim, fasta_no_trim,
                                   qual_no_trim, fasta_trim, qual_trim):
        # print("")
        print(s.id)
        # print(s.seq)
        # print(s.letter_annotations["phred_quality"])

        assert s.id == f.id == q.id
        assert str(s.seq) == str(f.seq)
        assert s.letter_annotations[
            "phred_quality"] == q.letter_annotations["phred_quality"]

        assert s.id == sT.id == fT.id == qT.id
        assert str(sT.seq) == str(fT.seq)
        assert sT.letter_annotations[
            "phred_quality"] == qT.letter_annotations["phred_quality"]

    print("Writing with a list of SeqRecords...")
    handle = BytesIO()
    w = SffWriter(handle, xml=metadata)
    w.write_file(sff)  # list
    data = handle.getvalue()
    print("And again with an iterator...")
    handle = BytesIO()
    w = SffWriter(handle, xml=metadata)
    w.write_file(iter(sff))
    assert data == handle.getvalue()
    # Check 100% identical to the original:
    filename = "../../Tests/Roche/E3MFGYR02_random_10_reads.sff"
    with open(filename, "rb") as handle:
        original = handle.read()
        assert len(data) == len(original)
        assert data == original
        del data

    print("-" * 50)
    filename = "../../Tests/Roche/greek.sff"
    with open(filename, "rb") as handle:
        for record in SffIterator(handle):
            print(record.id)
    with open(filename, "rb") as handle:
        index1 = sorted(_sff_read_roche_index(handle))
    with open(filename, "rb") as handle:
        index2 = sorted(_sff_do_slow_index(handle))
    assert index1 == index2
    try:
        with open(filename, "rb") as handle:
            print(ReadRocheXmlManifest(handle))
        assert False, "Should fail!"
    except ValueError:
        pass

    with open(filename, "rb") as handle:
        for record in SffIterator(handle):
            pass
        try:
            for record in SffIterator(handle):
                print(record.id)
            assert False, "Should have failed"
        except ValueError as err:
            print("Checking what happens on re-reading a handle:")
            print(err)

    """
    # Ugly code to make test files...
    index = ".diy1.00This is a fake index block (DIY = Do It Yourself), which is allowed under the SFF standard.\0
    padding = len(index)%8
    if padding:
        padding = 8 - padding
```

```
1399     index += chr(0)*padding
1400     assert len(index)%8 == 0
1401
1402     # Ugly bit of code to make a fake index at start
1403     records = list(SffIterator(
1404         open("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "rb")))
1405     out_handle = open(
1406         "../../Tests/Roche/E3MFGYR02_alt_index_at_start.sff", "w")
1407     index = ".diy1.00This is a fake index block (DIY = Do It Yourself), which is allowed under the SFF standard.\0
1408     padding = len(index)%8
1409     if padding:
1410         padding = 8 - padding
1411     index += chr(0)*padding
1412     w = SffWriter(out_handle, index=False, xml=None)
1413     # Fake the header...
1414     w._number_of_reads = len(records)
1415     w._index_start = 0
1416     w._index_length = 0
1417     w._key_sequence = records[0].annotations["flow_key"]
1418     w._flow_chars = records[0].annotations["flow_chars"]
1419     w._number_of_flows_per_read = len(w._flow_chars)
1420     w.write_header()
1421     w._index_start = out_handle.tell()
1422     w._index_length = len(index)
1423     out_handle.seek(0)
1424     w.write_header() # this time with index info
1425     w.handle.write(index)
1426     for record in records:
1427         w.write_record(record)
1428     out_handle.close()
1429     records2 = list(SffIterator(
1430         open("../../Tests/Roche/E3MFGYR02_alt_index_at_start.sff", "rb")))
1431     for old, new in zip(records, records2):
1432         assert str(old.seq)==str(new.seq)
1433     i = list(_sff_do_slow_index(
1434         open("../../Tests/Roche/E3MFGYR02_alt_index_at_start.sff", "rb")))
1435
1436     # Ugly bit of code to make a fake index in middle
1437     records = list(SffIterator(
1438         open("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "rb")))
1439     out_handle = open(
1440         "../../Tests/Roche/E3MFGYR02_alt_index_in_middle.sff", "w")
1441     index = ".diy1.00This is a fake index block (DIY = Do It Yourself), which is allowed under the SFF standard.\0
1442     padding = len(index)%8
1443     if padding:
1444         padding = 8 - padding
1445     index += chr(0)*padding
1446     w = SffWriter(out_handle, index=False, xml=None)
1447     # Fake the header...
1448     w._number_of_reads = len(records)
1449     w._index_start = 0
1450     w._index_length = 0
1451     w._key_sequence = records[0].annotations["flow_key"]
1452     w._flow_chars = records[0].annotations["flow_chars"]
1453     w._number_of_flows_per_read = len(w._flow_chars)
1454     w.write_header()
1455     for record in records[:5]:
1456         w.write_record(record)
1457     w._index_start = out_handle.tell()
1458     w._index_length = len(index)
1459     w.handle.write(index)
1460     for record in records[5:]:
1461         w.write_record(record)
1462     out_handle.seek(0)
1463     w.write_header() # this time with index info
1464     out_handle.close()
1465     records2 = list(SffIterator(
1466         open("../../Tests/Roche/E3MFGYR02_alt_index_in_middle.sff", "rb")))
1467     for old, new in zip(records, records2):
1468         assert str(old.seq)==str(new.seq)
1469     j = list(_sff_do_slow_index(
1470         open("../../Tests/Roche/E3MFGYR02_alt_index_in_middle.sff", "rb")))
1471
1472     # Ugly bit of code to make a fake index at end
```

```
1473     records = list(SffIterator(
1474         open("../../Tests/Roche/E3MFGYR02_random_10_reads.sff", "rb")))
1475     with open("../../Tests/Roche/E3MFGYR02_alt_index_at_end.sff", "w") as out_handle:
1476         w = SffWriter(out_handle, index=False, xml=None)
1477         # Fake the header...
1478         w._number_of_reads = len(records)
1479         w._index_start = 0
1480         w._index_length = 0
1481         w._key_sequence = records[0].annotations["flow_key"]
1482         w._flow_chars = records[0].annotations["flow_chars"]
1483         w._number_of_flows_per_read = len(w._flow_chars)
1484         w.write_header()
1485         for record in records:
1486             w.write_record(record)
1487         w._index_start = out_handle.tell()
1488         w._index_length = len(index)
1489         out_handle.write(index)
1490         out_handle.seek(0)
1491         w.write_header() # this time with index info
1492     records2 = list(SffIterator(
1493         open("../../Tests/Roche/E3MFGYR02_alt_index_at_end.sff", "rb")))
1494     for old, new in zip(records, records2):
1495         assert str(old.seq)==str(new.seq)
1496     try:
1497         print(ReadRocheXmlManifest(
1498             open("../../Tests/Roche/E3MFGYR02_alt_index_at_end.sff", "rb")))
1499         assert False, "Should fail!"
1500     except ValueError:
1501         pass
1502     k = list(_sff_do_slow_index(
1503         open("../../Tests/Roche/E3MFGYR02_alt_index_at_end.sff", "rb")))
1504     """
1505
1506     print("Done")
```