Assignment - In progress

Add attachment(s), then choose the appropriate button at the bottom.

Title	Fasta and FastQ files
Due	Oct 3, 2016 11:55 pm
Number of resubmissions allowed	0
Status	Not Started
Grade Scale	Points (max 25.0)
Modified by instructor	Oct 1, 2016 3:35 pm

Instructions

Sequence

A sequence is an ordered list of symbols. It might describe the composition of a biopolymer. From this sequence we can infer some of the properties of the molecule, like it's isoelectic point or possibly it's location in the cell, or maybe it's function though each of these require a body of previous experimental knowledge to which we can make sequence comparisons. We might also be able to infer properties of subsequences, like a binding motif in a DNA sequence or a region of a protein sequence that may form a transmembrane helix.

Typical sequences are of nucleic acids or of proteins, using the alphabets of [ACGTNU] or [ACDEFGHIKLMNPQRSTVWY], respectively. We can also create sequences using characters of any alphabet, possibly representing the sequential shapes of a folded protein, or an encoding of the ordered phonemes in a bird song. The important properties of a sequence include the encoded characters of its alphabet and the ordered list of instances of those characters.

FASTA

A FASTA file is a collection of sequences with identifying labels.

A line that has a '>' character in the first column is an id line, with the id starting in the second column and running until white space (space, tab, or end-of-line). IDs end just before whitespace (including end of line). Comma, or any other character is legal in an ID. Note: no space is allowed between the '>' and the id. If you have a space after the '>', you have an empty id string, which is illegal in many fasta-reading programs.

The rest of the id line after the id is a comment and is frequently used to provide more information about the sequence. An id line identifies the sequence that follows, which consists of all lines until the next id line (or end-of-file). There is no standard way to provide comments other than on the id line (some, but not all, FASTA-reading programs ignore lines beginning with '#'). There is also no standard way to associate multiple ids with a sequence.

Lines that do not have a '>' in the first column are sequence lines. White space is ignored on sequence lines, but all other characters are interpreted as being part of the sequence, including characters that seem nonsensical for the alphabet of the sequence. A single sequence can span many lines, and it is quite common for the characters of the sequence to be grouped into sets of 10 characters. Putting 5 sets of 10 characters each on each line makes hand-counting easier---handy when you are debugging a program. To avoid breaking poorly written programs, it is a good idea not to make sequences lines extremely long.

Blank lines and lines containing just whitespace should be treated as legal anywhere in a FASTA file, even before the first ID line. Note: NCBI documentation states that blank lines are not allowed within the body of a FASTA record (ID with associated sequence), and we will take a more permissive stance.

FASTQ

FASTQ (see: The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants, Cock PJ et. al, 2009, NAR) records include an ID line and sequence information, similar to a FASTA file; and they also include a measure of quality associated with each symbol. FASTQ files are organized in 2 parts, the first has an ID and sequence, the second may repeat the ID and includes the quality scores. The ID/sequence section starts with @, and the quality section starts with +. The ID portion of the quality section is usually a copy of the ID section for the sequence section.

```
@identifier comment
ACGTACGTACGTN
+
HHHHHHHHHHHHB
```

The quality alphabet encodes a score that is mapped to the ASCII table, such that a single character can be used to encode a quality score for each character of the sequence. There are two basic mappings to the ASCII table - wikipedia has a nice description. The original Sanger mapping made use of ASCII characters range(33:73) to describe quality scores in range (0:40) and this encoding is called PHRED+33. Solexa (now Illumina) decided to use an offset of 64, creating PHRED+64 using ASCII range (64:104) to represent quality scores in range 0:40. Technically the upper bound on these ranges can go larger but this is not typically done.

The character value 33 corresponds to an ASCII "!", and the number 64 corresponds to ASCII "@", and each of these in PHRED+33 or PHRED+64 mappings (respectively), corresponds to a quality score of 0, which is the lowest quality score possible (exception: pre v 1.3). These names: "PHRED+33" or "PHRED+64", are used to name the specific representation of the quality scoring, and they remind us how the mapping can be computed. If you have a Qscore of 0, and you want it's character mapping for PHRED+33, you can add 33 to the quality score and convert that to a character. You may find the built-in python functions chr() and ord() useful for this task--example: chr(Q+33), or chr(Q + ord('!')).

In at least one version, Illumina decided that the lowest quality score should be "2" even when assigned to a Base read of N--this case means the machine could provide no information for that base, yet claim a P(error) < 1 !!. Some downstream programs get confused when we give them a base read of "N" with a quality score of 2.

Finally, Illumina versions before 1.3 specified the quality as -10*log10(P/(1-P)) (clarification: pre 1.3 Illumina is Solexa). This makes little difference where P(error) < 0.3, and where it does matter, the PHRED+64 mapping starts at ASCII 59 mapping Qscores of -1 through -5 (Ascii characters ;<=>?) which correspond to: 0.3 < P(error) < 1 for those 5 codes. These codes need to be fixed to appropriate PHRED+33 or PHRED+64 mappings, where the code reflects P(error).

PHRED Q Scores

What do the quality values refer to? In all cases except the early Solexa data (before version 1.3), the score is an error probability (denoted P(error) or P(e)). The score is encoded as the log10 of that error probability, and.. in order to make use of the ASCII encoding, the log10-prob is then multiplied by -10. (-10*log10(P)). If you see a "!" character in a PHRED+33 encoding, corresponding to a Quality score of 0, then this means that there is a probability of 1 that the base called in the corresponding sequence position is an error. In PHRED+64 encoding, a "@" encodes a quality score of 0 which we interpret as P(error)=1 that this base is in error. In other words.. that base is not known.

Examples:

Here is an example of a FASTA file with two sequences

```
>1914
masmtggqqm gripgnsprM VLLESEQFLT ELTRLFQKCR SSGSVFITLK
KYDgrtkpip rkssvEGLEP AENKCLLRAT DGKRKISTVV SSKEVNKFQM
AYSNLLRANM DGLKKRdkkn kskkskpAQG GEQKLiseed dsagspmpqF
QTWEEFSRAA EKLYLADPMK VRVVLKYRHV DGNLCIKVTD DLVCLVYRTD
QAQDVKKIEK FHSQLMRLMV AKESRNVtme te
>1a2xB
gdEEKRNRAITARRQHLKSVMLQIAATELEKEEgrreaekqnylaeh
```

Here is an example of a FASTQ file with 2 sequences:

```
@HWI-ST611_0189:1:1101:1237:2140#0/1
ACTAGCTGTCCTTGGTGCCCGAGTGTATTGAAAGTTGATTCCCTTATAGATGTTCGTTTTCCACACAACTCTGTAGG
CACCANCAATNACTAGTAGATCG
+HWI-ST611_0189:1:1101:1237:2140#0/1
___`cccceeeeehh`dcfhe`dQb`deehehebY^aedfdhhhhe_ebaa]cebe_eehfhS__ede_V\HHV^^^
acccBBBBBBBBBBBBBBBBBBB
@HWI-ST611_0189:1:1101:1178:2158#0/1
ACTAGCTATTGATGGTGCCTACAGTCTCAGTTGAGGGGAAANNNNCGNGNNNCNNNNNNNNNNNNNANNNCNGNNNA
GNNNNNNNNNNNNNNNNGCCNANN
+HWI-ST611_0189:1:1101:1178:2158#0/1
bbbeeeeeggfegiidggggghghhhhifhghhhiiiifghBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBB
```

Notice that the quality scores go from "B" to "b", meaning Q2 through Q34. This tells us immediately that this is PHRED+64 encoded, and it is using "B" as its lowest score even when the corresponding sequence letter is N. So, all of those "B"s should be considered as P(error) = 1. A string of "B" on the 3' end, further means that all of the read sequence should be disregarded even if it has apparently valid bases.

Suggested Reading

In order to do this assignment, you will need to learn some Python. You should read Chapter 4 "The Python Language" from Python in a Nutshell, though it is rather dry and lacks examples. You don't need all of Chapter 4 for this assignment, but try reading at least the sections "Data types", "Variables and other references", "Sequence operations", "The print statement", and "Control flow statements".

The following selections from the on-line documentation will probably give you most of what you need and include more tutorial examples.

Tutorial Docs:

http://docs.python.org/tutorial/interpreter.html
http://docs.python.org/tutorial/introduction.html
http://docs.python.org/tutorial/controlflow.html
http://docs.python.org/tutorial/inputoutput.html

Understanding Strings (And More Generally, Sequences):

http://docs.python.org/library/stdtypes.html#string-methods
http://docs.python.org/library/stdtypes.html#string-formatting-operations
http://docs.python.org/library/stdtypes.html#sequence-types-str-unicode-list-tuple-buffer-xrange

Useful Standard Library Modules:

http://docs.python.org/library/string.html
http://docs.python.org/library/re.html
http://docs.python.org/library/sys.html

You also might find it useful to learn some Python idioms (standard ways of expressing commonly used bits of code). Perhaps http://jaynes.colorado.edu/PythonIdioms.html would be useful.

PYTHON Assignment 1:

Read a FASTQ file that may have mixed upper and lower case letters, make them all upper case. We might see any of "*", ".", "n" or "N" all meaning unknown base. Make all of these "N". *Note: Standard FASTQ only defines [ACGTUN] as sequence characters.*

The program should be a "filter" program that reads from STDIN and outputs to STDOUT. We will not delete any characters, so the final length of each line should remain unchanged.

Implement options that specify an input of PHRED+33 or PHRED+64 and a desired output of PHRED+33 or PHRED+64. Use:

- -P33in (--PHRED33input),
- -P64in (--PHRED64input)
- -P64Bin (--PHRED64Binput with B offset in quality values)
- -P64SOLin (--PHRED64SOLinput with SOLEXA interpretation of Q score)
- -P33out (--PHRED33output) [make this your default output - SANGER FASTQ]
- -P64out (--PHRED64output)

In the case of the PHRED64 Solexa mapping (--PHREDSOLin), remap these scores taking into consideration the pre 1.3, Solexa quality mapping and assuming that the output will be a mapping of P(error), rather than P(error)/(1- P(error)).

For the PHRED64 B case, all "N"s should have an associated Q0, and all 3' terminal "B"s should coerce their associated bases in the sequence to be "N"s, as well as having an associated Q0 quality score.

Think carefully about boundary conditions: an empty file, an id line with a zero-length sequence, a mis-ordered FASTQ file (ID lines of sequence and quality lines are both specified but not equal). For this exercise we will not allow quality lines without an ID specified. Remember to ignore the comment fields though.

The program should also report to stderr (not stdout) any anomalies that might be found, and the assumptions that are made about the input file. Is it a PHRED+33 or PHRED+64 file ? How many FASTQ records were found? How many BAD records were found (BAD IDs, insufficient number of quality scores, sequence without quality or quality section without sequence).

The error messages to stderr should be fairly terse. You probably want to report only once errors that violate an assumption, such as.. PHRED+33 file assumed, quality scores out of range.

I would like to see messages like

ERROR: '@' not in first column for 'HWI-ST611_0189:1:1101:1237:2140#0/1'

ERROR: not FASTQ format, sequence line before first id line.
WARNING: input has no sequences.
WARNING: input has 3 empty sequences.

Call your program "fastqPrep.py". Your file should also be an executable python program, which requires two things:

- The first line of the file should be

  #!/usr/bin/env python3

  to indicate what program is to interpret the code. On SOE computational machines you will get version 3.5.2 of python using:

  #!/usr/bin/env python3

- The unix execute permissions need to be set for user, group, and other with "chmod ugo+x fastqPrep.py".

Only executable files and directories should be made executable. The command chmod go+rX * will make all files readable by "group" and "other", and will make executable for everyone those that are executable by the user. All directories on the path down to the file need to be readable and executable for me to be able to access the files.

Test Files

I will provide a few test files in eCommons::resources

Hints:

- Organize your program into "paragraphs" and separate the paragraphs with blank lines. A block comment at the beginning of each paragraph saying what the paragraph does (like a topic sentence) makes maintenance much easier.
- Indent your comments just like code---don't allow the comments to break up the structure that is shown by the indenting. It is a good idea to associate comments with "then" and "else" code blocks, rather than with "if" statements.
- To aid in clarity, it is a good idea to use the increment operators (+=, -=, *=, /=, .=) rather than repeating the left-hand side of an equation. This also helps cut down on the possibility of mistyping the left-hand side, which would result in an error that would be very difficult to detect.
- The program should use sys.exit(1)to return a non-zero exit code when it detects a fatal error. This will make it easier for shell scripts and make files to use the fastqPrep.py program. You will need "import sys" near the beginning of your program to give you access to this function.

- The string method line.isspace() might help in detecting blank lines.
- Use something like

  ```
  for line in sys.stdin:
      line = line.rstrip()  # strip right terminal whitespace
  ```

  to read the input.

  Don't read the entire file into memory, but process a line at a time. In general, bioinformatics programs may have huge data sets to deal with, and you should get into the habit of keeping only essential information in memory.

  Files created under MS-DOS (and Windows) follow the convention that lines are terminated by a carriage return (^M) followed by a line feed (^J). Unix conventions call for only a single character (^J) to end a line. Some programs get confused by the extraneous ^M characters, so it is a good idea for filter programs to remove them. (The notation ^M means: ctrl-M, and is a specific ASCII character - chr(13), and ^J is ctrl-J, which is chr(10)).

- Use line.translate() string method to do the substitutions. You can use str.maketrans() to create the translation table. (the maketrans() function is in the str module, so make sure you "import" it)
- Using the .split() method makes peeling off the ID from the id line easy.
- You can use either

  print ("WARNING: read", num_empty, "empty sequences", file = sys.stderr)

  or

  sys.stderr.write("WARNING: read %d empty sequences\n" % num_empty)

  for output of error messages. The write method has the advantage of not needing to use backslashes to continue onto a new line, but you have to use explicit spaces, newlines, and string concatenation to get a single argument for write().

Important Points

Document your code clearly! It is particularly important to have a block comment at the beginning of the program that documents what the program is supposed to do, as this is often the only way of telling what a year-old python script is for. Your documentation should be fairly complete, giving all the user full information about how to use the program, and should be clear enough that someone new to bioinformatics can still grasp what it does.

1. The first block comment of the program should contain several things:

   I will expect all programs to have such an initial block comment.

   - Your name and the date
   - The name of the program
   - Usage instructions for the program
   - An external view of the program that explains what task it accomplishes (not an internal view that explains how it works or what data structures it uses).
2. Turn in your code and the input and output from one run (short) showing that the code works. Submit at eCommons.

Bonus Points

It would be very nice to infer what the input quality format is. Since most FASTQ records will make use of a quality specifier that is unique to a given format, this is a very useful feature. In this case, make sure to write to stderr which assumption is being made !! Properly handle all of the input mappings automatically. The input file could be P64SOL, in which case you need to properly handle the remapping of quality scores from solexa to illumina. The input may be a clean PHRED+64 that uses character mappings Q0 through Q40, starting at ASCII "!" . The input might also have a low bound "B" quality score that must be remapped to Q0. In this case, all N's should have an associated Q0, and any 3' terminal strings of "B" should be remapped to Q0 and the associated bases chould be should be coerced to N. Or, you may be seeing a clean PHRED+33. Hint - a nice way to handle this is to read FASTQ records until you get enough data to properly discriminate, then close and reopen the file.

Instead of processing a line at a time, you can try processing a sequence at a time (4-lines). In later assignments it will be useful to have a generator that reads fastq files and yields a list containing (id, comment, sequence). This generator should be an an object of its own !!

Optional

--verbose

reports every sequence that has a different length from the first sequence (report the ID), and report any sequence whose sequence length does not match its quality length.

--fasta-out

write a fasta file without any of the quality information

--require-unique-ids

Check that each id in the FASTQ file is different from all previous ones (a set may be a good data structure for this).

Note that the Python module argparse is required for parsing the command-line arguments.

There are simple, fast, easy-to-understand functions for a lot of string operations
(examples: line.startswith('>') or line.isspace()), rather than use of regular expressions.