Project #1

Objective

To reinforce the basics of Python programming learned from the Shaw book except for Object Oriented programming.

Due

Friday February 9th at 5pm PST. Please create a new Project01 folder on your GitHub repository and upload your final script once completed. Remember, you should not consult with other students to work on this project unless you do so using the #project channel on Slack. This is because the communication via Slack is preserved and everyone can see it. You may ask and answer questions on Slack but do not post code. If you finish the project early, please wait to upload it to GitHub until noon on Friday at the earliest. To get you started, you will be given a script that has the functions you need already documented and ready to go. Your job is to fill them in.

Tasks

One of the most common sequence file formats is the FASTQ format. For those of you who have worked with DNA sequence files you will be familiar with these. The FASTQ format contains DNA sequences that were produced by high-throughput DNA sequencers. The file uses a specific format. Each DNA sequence in the file is referred to as a **read**, and each read is defined in the FASTQ format using four lines. Here is an example for a single read:

```
@cluster_2:UMI_ATTCCG
TTTCCGGGGCACATAATCTTCAGCCGGGCGC
+
9C;=;=<9@4868>9:67AA<9>65<=>591
```

The first line always begins with an @ symbol and contains the unique name of the read. The second line contains the DNA sequence for that read. The third line always begins with a + character and is usually left blank or optionally contain the unique name for the read. Finally, the fourth line contains quality data for the read. Each individual nucleotide in the sequence has an associated quality score. These scores range from 0-93. Typically, a quality score of 20 is bare minimum quality, 30 is typically what is wanted, and anything above 30 is even better. Rather than store numbers to represent nucleotide quality, the FASTQ format uses a shortcut by storing characters that represent quality scores. Remember in the Shaw book when character encoding was discussed? Shaw mentioned the ASCII encoding. It was one of the first character encodings but was limited to the English language. Here ASCII is used to encode the quality scores. You can find ASCII tables easily online. Here's an example: https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html

In the ASCII table, notice that the ! character maps to the decimal value 33, and the " character maps to the decimal value 34. You can calculate the quality score in a modern FASTQ file by simply converting the character to its decimal value and subtracting 33. The Shaw book told you how to do this. If interested, you can learn more about the FASTQ format from these references: https://en.wikipedia.org/wiki/FASTQ_format, http://maq.sourceforge.net/fastq.shtml

Often, DNA sequencers will generate reads where both the beginning and end of the reads are lower quality than the middle. Usually the ends are worse than the beginning. However, when we want to use those reads in downstream analysis (such as for alignment to a genome sequence), we don't want those low-quality nucleotides on the ends to confuse the alignment. Therefore, it is common practice to trim

off the low-quality ends (both at the beginning and the end) of every read. One common tool to do this is trimmomatic (http://www.usadellab.org/cms/?page=trimmomatic, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4103590/).

Your job is to create very simple trimming program that does the following:

1. Open a FASTQ file
2. Process each read from the file and remove low-quality nucleotides from the beginning of the read (you don't need to remove from the end of the read).
3. Your script will then write out a new FASTQ file that contains the reads but with their trimmed sequences (not the original sequence)
4. When trimming a read, you only need to trim from the beginning up until you see the first high-quality base of 30 or above. Once you see the first high quality base you can stop trimming.
5. If a read is less than 30 nucleotides after trimming, then it is too short and that read should not be included in the final output file.
6. Write a new FASTQ file that contains only the trimmed sequences that are at least 30 nucleotides long.

You will be given:

1. A starter script to get you going.
    a. Notice that this starter script is well commented with what is known as Sphnix reST compatible docstring. This allows the Sphinx program to create nice web formatted documentation for your code simply by reading the comments. We won't be generating that documentation, but it's good practice to start documented well now!
    b. Also notice that this script already has the functions for you. You need to fill them out and figure out how to connect them.
2. An input FASTQ file to test with.
3. An example output FASTQ file to check your results for correctness.

Criteria that your script must meet:

1. Your script should be named Project01.py.
2. Fill in the top of the starter script where it has a space for your name.
3. Your script must receive as input 4 arguments in this order:
    a. The filename of the input FASTQ file.
    b. The filename for the new output FASTQ file that your script will generate.
    c. The minimum quality score that a nucleotide must have to not be trimmed (this should be set to 30)
    d. The minimum read size after trimming to include the read in the final output (this should be set to 30).
4. Your script will therefore be executed using a command like this:

```
python3.6 Project01.py SP1.fq SP1.trim.fq 30 30
```

5. Aside from creating the new trimmed FASTQ output, your script should also output to the terminal (i.e. STDOUT) the following text as it proceeds.

```
Opening SP1.fq file for reading...
Opening SP1.trim.fq file for writing...
XXXX reads were found
XXXX reads were removed
XXXX reads were trimmed and kept
```

```
Done.
```

Where XXXX contains appropriate counts.

6. You should retain all the commenting and "docstrings" in the start script given to you. And even with that left in, your script should be no longer than 185 lines. If you find your code is getting too long then you may be over complicating it.

Grading

You will be graded on the following scale

1. Your program runs without any errors: 70 points.
2. Your program generates an output file identical to the sample output file provided: 20 points.
3. Your script meets all the criteria specified above: 10 points.
4. 5 points will be subtracted for each day late.
5. You can earn 15 extra credit points by adding a working function titled trim_read_end() that removes low quality nucleotides from the end of the sequence. The same minimum sequence quality and minimum string size should be used. If you add this function, then your script can go beyond the 185 lines.

Hints to help you out:

1. The readline() function will include the trailing carriage return when it returns a line from a file. This is therefore not actually part of the DNA sequence or the quality scores. Think, how can you deal with it?
2. Each individual character in a string can be accessed just like a list, for example, if you have a string object named xyz, then xyz[0] will access the first character of string xyz.
3. The author did not mention this, but you can pull out a subset of a string by using a colon when indexing a string. For example, to pull out characters 5-10 of the string you would do the following: xyz[4:11] (4 because lists are zero indexed so that gets us character 5 and 11 because this notation includes everything up to but not including the 11^{th} character, and because we want character 10 we must set it to 11). Also, you can pull out everything from character 5 to the end with this: xyz[4:]. Think how you might use this to get this project to work.

Best Practices

Avoid trying to write your program in one large sitting. If you do, odds are you will spend an inordinate amount trying to figure out where bugs are occurring. Rather, program in steps.

- Start at the beginning and do one simple atomic thing.
- Test your program by running it to make sure that one thing works.
- Once you know that thing works, then move on to the next thing.
- This will help tremendously because you will work out bugs as you go along.
- If you get an error, believe it! Error messages tell you exactly what is going on. Make sure you understand what the error messages mean before you start trying to apply a fix. Otherwise you may waste a lot of time trying to fix something that wasn't really a problem…
- Don't spend too much time on a bug. You should not spend hours trying to figure something out. Always try to resolve bugs first because this is how you learn, but don't spend too long on a bug. Rather, send a message on the slack #project channel. You can post your error messages and ask for help--just don't post code.