Programming Lab II
Handout 5

Swarit Jasial, Dimitar Yonchev, Dr. Martin Vogt
`martin.vogt@bit.uni-bonn.de`

May 29, 2018

Deadline: June 14, 2018

Sequence data can be found in the `handout-5` folder accessible from the sciebo folder: `https://uni-bonn.sciebo.de/s/meq78OszilhzvH6`

1. *(10 pts) Reading and writing sequences in FASTA format*
   In previouse exercises you were already tasked with reading (at least a single) sequence(s) from FASTA files. Having a collection of functions available for reading and writing sequence data is quite handy. There are a wide variety of formats available. However, the FASTA format is both popular and simple. Thus, it is well suited for writing a set of functions for dealing with sequences in this format. It does not matter whether the sequence(s) repesented consist of DNA, RNA or amino acids; however, there might be some conventions on file name extensions when dealing with one or the other type of sequence. More importantly, a file may contain a single sequence or more than one sequence and functions for reading and writing sequences in FASTA format should be able to deal with both cases:

   (a) *(1 pt) Read a single sequence*
   Write a function `single_fasta_sequence(filename)` that reads a sequence given in FASTA format from a file containing a single sequence. The function should return a tuple containing the header line as first element (without the starting > and without the end-of-line character) and the sequence as a string as second element (without any end-of-line characters). Note, that a sequence can span multiple lines but you should return the sequence as a single string. The function should be usable in the following manner:

   ```
   f = open("ecoli-genome.fna")
   hd,seq = read_single_fasta_sequence(f)
   f.close()
   print("header:",hd)
   print("sequence length:",len(seq))
   ```

   What is the result for `ecoli-genome.fna`?

   (b) *(2 pts) Reading multiple sequences from a file*
   A FASTA file can contain more than one sequence. Write a function `fasta_list(filename)` that reads all sequences from a FASTA file and returns a list of tuples, each tuple containing the header as the first and the sequence as the second element.

   Note, that a function written this way would normally first read the complete data contained in the FASTA file and return all the data in a single data structure, i.e., a list of tuples.

   (c) *(2 pts) Using generators to read entries from FASTA files*
   FASTA files can be very large and calling the function would use a lot of memory.

However, if you have a FASTA file (e.g. `ecoli-genes.ffn`) containing a number of sequences, often you just want to perform some operation on each sequence separately (maybe something a simple as determine its length). In this case, it is not really necessary to first store all the elements in memory, rather it is preferable to have a mechanism that would yield one element (consisting of the header information and the sequence) at a time without reading all the data first. We already encountered such a mechanism when reading lines from a file:

> The following code first reads all the lines of a file in a list and requires at least as much memory as the file size because the complete contents of the file are stored in `all_lines`.
>
> ```
> infile = open("words.txt")
> all_lines = infile.readlines()
> max = 0
> for line in all_lines:
>         if len(line) > max:
>                 max = len(line)
> print("The longest line has",max,"characters.")
> ```
>
> As a more memory-efficient alternative, the following code only reads one line at a time from the file and uses much less memory because the whole contents of the file are never stored in an object.
>
> ```
> infile = open("words.txt")
> max = 0
> for line in infile:
>         if len(line) > max:
>                 max = len(line)
> print("The longest line has",max,"characters.")
> ```

In this vein, it would be conceptually very nice if one could write something like:

```
max_length = 0
name = None
for header, sequence in fasta_sequences("fasta.ffn"):
        if len(sequence) > max_length:
                max_length = len(sequence)
                name = header
print("The longest gene is",name, "and contains",max_length,"nucleobases.")
```

Or, in a more functional way:

```
name,seq = max(fasta_sequences("fasta.ffn"),key= lambda x: len(x[1]))
max_length = len(seq)
print("The longest gene is",name,"and contains",max_length,"nucleobases.")
```

Write a function `fasta_sequences(filename)` that can be used in the manner described above using a self written generator using the `yield` keyword. For understanding the generator concept and `yield`, see for instance:

- `http://pythoncentral.io/python-generators-and-yield-keyword`
- `http://www.python-course.eu/generators.php`

(d) *(1 pt)*
Test the functions written in (b) and (c) on `ecoli-proteome.faa`. Read the Fasta file and print the header and the length of the shortest and longest amino acid sequence.

(e) *(2 pts) Writing FASTA sequences*
Now write a function `write_fasta(outfile,header,sequence)` that writes the `sequence` with `header` to the **opened file** `outfile` in FASTA format. Make sure the the written header line starts with > and that the sequence is split into lines containing exactly 70 symbols (the final line may contain less than 70 symbols but should not be empty).

Your function should be usable in the following manner

`open/close` syntax	`with` syntax
`f = open("sillySequence.faa","w")`	`with open("sillySequence.faa","w") as f:`
`hd = "python"`	`    hd = "python"`
`seq = "FLYINGCIRCVS"`	`    seq = "FLYINGCIRCVS"`
`write_fasta(f,hd,seq)`	`    write_fasta(f,hd,seq)`
`f.close()`	

If written this way, the function can also be used to write multiple sequences to a single file. E.g., the following piece of code writes short amino acid sequences of an albatross, a lumberjack, and a dead parrot to the file `nudgenudge.faa`.

```
with open("nudgenudge.faa","w") as f:
    write_fasta(f,"albatross","WHATFLAVQRISIT")
    write_fasta(f,"lumberjack","ISLEEPALLNIGHTANDIWQRKALLDAY")
    write_fasta(f,"deadparrot","NQRWEGIANPLVE")
```

(f) *(2 pts) Creating a module: Putting things together.*
Put the definitions of the previous functions in a single file named `fastatools.py`. If you would run this script by itself you would observe no effect because all it does is define the functions. However, these functions might be actually useful as part of another script. Or, if you run it from the notebook the functions can be used from within the notebook. You can use the `import` statement, which you already know from Python standard modules such as `math`:

```
from fastatools import single_fasta_sequence
f = open("ecoli-genome.fna")
species,genome = single_fasta_sequence(f)
f.close()
print("The genome of",species,"contains",len(genome),"nucleotides.")
```

or

```
import fastatools
f=open("truth.faa","w")
fastatools.write_fasta(f,"theking","ELVISISALIVE")
fastatools.write_fasta(f,"liverpoolfour","PAVLISDEAD")
f.close()
```

Python will find and import `fastatools.py` if it is in the current directory. You may want to use the functions defined here in the following exercise by `import`ing them.

Open reading frames

Ex.2 *(3 pts) Complementary DNA*
Write a script `cdna.py` that takes two command line parameters. The first is an input file containing a DNA sequence in FASTA format. Read the sequence and generate the cDNA (complementary DNA strand) and write it to the file given as second parameter. You might want to add "cDNA" to the header of the output file.

Note: You can test your program on the small sample file `ecoli-genome-sample.fna` before running it of the big `ecoli-genome.fna` file.

Ex.3 *(12 pts + 5 pts) Finding genes in prokaryotes: Open Reading Frames*
The composition of the DNA of prokaryotes is much simpler than that of eukaryotes:

- There are few non-coding regions of DNA in the genome.
- Protein-coding genes do not include introns, i.e., parts of the RNA that are removed after transcription.

- Promoter regions upstream of the start of transcription are more highly conserved. E.g., the Pribnow box, a `TATAAT` consensus motif at position -10 (i.e., 10 base pairs before the start of transcription) and the `TTGACA` motif at position -35.

The most basic approach for identifying genes is by identifying open reading frames (ORFs) in the DNA.

The four nucleotides of the DNA A,C,G,T are used to encode the amino acid sequences making up a protein. There are 20 amino acids and each amino acid is encoded by a word of length 3 of nucleotides called a codon. $64 (= 4^3)$ different words of length three can be made from the four nucleotides. 61 of these code for amino acids and most of the amino acids can be encoded by more than one codon (e.g., 6 codons code for the leucine but only one encodes tryptophane). 3 codons (TAA, TAG, TGA) do not code for any amino acid. These are so-called stop codons, indicating the end of the amino acid sequence. Another codon (ATG) is the start codon indicating the start of a protein coding sequence. This codon also codes for the amino acid sequence methionine and the codon can also occur not at the beginning of a protein. (Often, the starting methionine will be excised after translation so that the final protein sequences do not have to start with methionine.)

Hence, in order to find protein coding sequences in the DNA one should look for segments of DNA that start with ATG and end in TAA,TAG, or TGA. However these need to be in the same *reading frame*. I.e., the distance between them needs to be a multiple of 3 because the translation will always proceed in steps of 3 nucleotides and will stop at one of the stop codons. Such a sequence from start to stop codon is called an *open reading frame*.

For a single strand of DNA there are three reading frames. One can either start at the first, second, or third nucleotide and proceed from there in steps of three. However, DNA is double-stranded consisting of the DNA sequence on one strand and the complementary sequence on the other strand pairing the nucleotides A with T and C with G. Genes can be found on both strands in varying reading frames. (Note that the reading direction of the DNA sequence on the complementary strand is *reversed*.) Genes may even overlap, i.e., have overlapping nucleotide sequences on the same strand of DNA in different reading frames or occupy corresponding parts on the complementary strand.

(a) *(12 pts) ORF finder*
Write a script `orf_finder.py` that takes two command line parameters. The first is an input file containing a DNA sequence. The second is an output file that should contain all the (longest) open reading frames found in DNA (from both strands). Note that for each stop codon you should only write one open reading frame: use the longest open reading frame ending at the stop codon. E.g., for the sequence

`GATGCGTAATGGATTCGATGCATGCGGCGTGAACTAGTCTAACG`

one open reading frame should be `ATGGATTCGATGCATGCGGCGTGA` and not `ATGCATGCGGCGTGA`. Note, that the blue start codons are not considered for the stop codon `TGA` because they are in other reading frames. Also near the beginning, the sequence `ATGCGTAA` is no proper reading frame because the start and stop codon are not in the same reading frame. The two other open reading frames to be found here are:

`ATGCGTAATGGATTCGATGCATGCGGCGTGAACTAG`

and

`ATGCGGCGTGAACTAGTCTAA`

In the output FASTA file, each open reading frame should have a header that identifies the genome and its position in the genome by giving the start and end position. The first word in a header of a FASTA sequence is an identifier. Take the identifier from the genome sequence and append a sequence identifier in the format `:123-455` for an ORF on the given strand from positon 123 (First nucleotide of the start codon) to 455 (last nucleotide of the stop codon) and `:c455-123` for an ORF on the complementary strand from 455 to 123. Use this as a new header for each ORF. (Start counting nucleotides from 1 and not from 0.)

Write the open reading frames in the order the appear on the genome of `ecoli-genome.fna` and store them in a file `ecoli-orf.ffn`.

Note: While writing your program and before using it on the whole genome `ecoli-genome.fna` you should try it on the small sample file `ecoli-genome-sample.fna` and compare the output to `ecoli-orf-sample.ffn` provided in the `proglab2` folder.

(b) *(Optional: 5 pts)* Make sure your program is not only *effective* but also *efficient*, i.e., it should be fast and not use unneccessary (implicit or explicit) nested loops.

Ex.4 *(10 pts) Assessing the quality of the ORF approach*

Just looking for open reading frames is a very simple method for trying to identify new genes. The proteome of *Escherichia coli* is known and the gene coding regions are available in FASTA format. The gene coding regions can be found in the file `ecoli-genes-standard.ffn`. This file contains those genes from the file `ecoli-genes.ffn` that correspond to a single continuous DNA region ending in one of the standard stop codons. Note, that this file also contains some genes (ca. 12%) with alternative start codons.

If you compare the lengths of the files from the previous exercise `ecoli-orf.ffn` and `ecoli-genes-standard.ffn` you should notice that the ORF file is much larger so probably a lot of non-coding ORFs were identified by our program. The following scripts will allow you to investigate how successful the program was.

First, we would like to find out how many gene-coding ORFs were identified and how many we missed. The header for each ORF in our output file contains the start and end position of each ORF. Likewise, these positions are given in the file `ecoli-genes-standard.ffn`.

(a) *(3 pts) Get gene sequence positions from headers*

Write a function `get_sequence_positions(fasta_file)` that can retrieve the sequence positions from the headers of the sequences in `ecoli-orf.ffn` and `ecoli-genes-standard.ffn`. In both files, sequence positons should have been stored basically in the same manner at the end of the identifier, in `ecoli-genes.ffn`:

```
>gi|556503834|ref|NC_000913.3|:190-255 Escherichia coli str. K-12 substr. MG1655
```

and in `ecoli-orf.ffn`:

```
>gi|556503834|ref|NC_000913.3|:190-255
```

The results should be stored in a dictionary where the key is the *end* position and the value is the start positon. The end position is used as key because a gene-coding ORF has to end at the stop codon but the start of the real gene might be one of several methionine codons ATG found inside the ORF.

(b) *(5 pts) Correctly predicted ORFs*

Write a script `correct_orfs.py` taking two command line parameters, the first should be a FASTA file of open reading frames and the second a file of genes. It should calculate and print the following things:

- total number of open reading frames
- total number of genes
- total number and ratio of open reading frames correctly predicting a gene (with correct start codon)
- total number and ratio of open reading frames correctly predicting at least the stop codon of a gene (i.e. the ORF might be longer because the real start of transcription is determined by another start codon in the ORF.)
- number of missed genes, i.e., genes of `ecoli-genes-standard.ffn` whose stop codon does not correspond to any ORF stop codon.

(c) *(2 pts) Taking ORF length into account*
These results might seem pretty disappointing. However, one simple aspect of open reading frames that has not been taken into account is their length. The average length of a bacterial protein is around 250 amino acids (corresponding to 750 nucleotides). Modify your previous script so that you can give an additional parameter determining the minimum length of an ORF/gene as a parameter. All ORFs smaller than the given parameter should not be considered genes. Check your results for minimum sizes of 50, 100, 150, 200, 250, 300, and 350 amino acids. (Remember that you need to multiply the number of amino acids by 3 to get the number of nucleotides.)

Ex.5 *Bonus: (5+5 pts) Optimizing the classification performance*
Given a minimum size, we can now classify each ORF as either gene coding or non-coding. A classification of an ORF will be considered correct even if the start codons do not match, i.e., only the stop codon has to correspond to the stop codon of a gene. Thus we will predict an ORF to code for a gene if it has a certain minimum length else we predict it as non-coding. For each prediction parameter (minimum length of ORF) one can now determine whether the prediction is correct using the file of genes `ecoli-genes-normal.ffn` or `ecoli-genes.ffn`.

Thus if we have an ORF and predict it to be a gene it can either be a true positive (TP) (it is indeed a gene) or a false positive (FP) (it is not a gene). If it is too small and it is predicted not to be a gene it can either be a true negative (TN) (not a gene) or a false negative (FN) (is a gene).

There are a number possibilities to measure the performance of this classifier. Because there are much more non-coding than coding ORFs, we will use the balanced accuracy. This avoids favoring the classification of every ORF as non-gene simply because there are much more non-coding than coding ORFs.

The balanced accuracy is given by the formula: $\text{acc}_{\text{balanced}} = 0.5 \cdot \left(\frac{TP}{TP+FN} + \frac{TN}{TN+FP} \right)$

Write a modified version of the script that is able to determine the minimum ORF length that gives the best balanced accuracy.

- *(Optional: 5 pts) Challenge:* Make your program as efficient as possible!