Lab 04 Files and Modules

Deliverables (50 pts total):

- sequenceAnalysis.py module - 31 total points
- classes included:
  - NucParams,
  - ProteinParams,
  - FastAreader
- genomeAnalyzer.py – 15 total points
- Lab 4 notebook with Inspection Intro Cell and Inspection Results Cell completed(4 pts)
- possible extra credit – 10 additional points
- Due: Monday February 5, 2024 11:55pm

Congratulations. You have started to build an inventory of some pretty useful functions. Because these are written as classes, you can easily reuse them. Your ProteinParam class is your first deposit into your very own sequenceAnalysis toolkit. In Python, these toolkits are called modules.

We are also going to start using real sequence files. The fastA format, described here: en.wikipedia.org/wiki/FASTA_format is very convenient to use and fully capable of storing sequences of many types. You will be reading these from an input file for this assignment.

Genomic analysis

There are a few things that we can do that mirror and extend the analyses that we did previously on protein sequences. We can calculate composition statistics on the genome (gc content for example), we can calculate relative codon usage in the genome, and we can calculate amino acid composition by translating those codons used the genome.

For this lab, I have provided a NucParams class, with the required methods that it implements (see below). You will need to design and write those methods, and these are to be placed in a file called sequenceAnalysis.py This is a ***module*** that you can use from now on.

You will also need to place the ProteinParams class from Lab 3 into this module. This class will not be used for this assignment, but place it into your toolbox.

I have written the FastAreader class. It is included below. Keep it as is part of your module for now, you may decide to keep it somewhere else later.

The input file for this assignment will be named testGenome.fa, and is available in Canvas. You will not need to submit testGenome.fa, but it will be necessary for your testing. For development and testing, create a new directory (Lab04) and place the data file (testGenome.fa), your Lab04 notebook, your program (genomeAnalyzer.py), your new module (sequenceAnalysis.py).

Hints

- Python modules have the *.py* extension as files, but when they are imported, the name without the extension is used in the import statement in your program.
- File placement: Make sure to place your notebook, program, sequenceAnalysis module and the required data files in the same folder. This will allow Python to find them. Read over the FastAreader usage to see how to specify file names that you can use for your data.

Codon frequency calculations

Notice that NucParams does all of the counting you need. It is responsible for counts of codons and their translated amino acids.

Your genomeAnalyzer.py program has the task of determining which codons are preferred for each of the amino acids and calculating the relative percentage. For any given amino acid, the relative codon usage (percentages) should sum to 100.0%. Notice that Methionine and Tryptophan only have 1 codon that codes for those, so these will have relative codon usages of 100%.

For example: Lysine is coded by both AAA (607) and AAG (917) (example counts in parentheses). From our aaComposition() method, we are given the aaComposition dictionary and we can lookup 'K' to find 1524 counts (these came from those 607+917 codons). We can then calculate 607/1524 for AAA and 917/1524 for AAG. The associated percentages are thus: 39.8 for AAA and 60.2 for AAG.

AAA = 607/1524 * 100 = 39.8%

AAG = 917/1524 * 100 = 60.2%

Design specification - sequenceAnalysis.py

NucParams class

__init__

The constructor of the class has one optional parameter, a sequence of type string. It may include upper or lower case letters of the set (ACGTUN) or whitespace. These will be gene sequences and they begin in frame 1. In other words the first 3 letters of the sequence encode the first AA of the sequence. Carefully consider in what form this class should maintain its data. Is a string the best structure? This class (NucParams) is intended to be very similar to ProteinParam. Make sure to read addSequence() before making this decision, and remember that objects of this class may need to handle an arbitrarily large number of sequences (hint: dictionaries are good). As a second hint, notice that **init** and addSequence are doing VERY similar things - you could just make one of them do most of the work.

addSequence() - 5 pts

This method must accept new sequences, from the (ACGTUN) alphabet, and each sequence can be presumed to start in frame 1. This data must be added to the data that you were given with the **init** method (if any).

aaComposition() – 6 pts

This method will return a dictionary of counts over the 20 amino acids and stop codons. This dictionary is VERY similar to the lab 3 aaComposition, though you must decode the codon first. The translation table from codon to AA is provided. You are counting amino acids by translating from the proper codon table.

nucComposition() - 10 pts

This method returns a dictionary of counts of valid nucleotides found in the analysis. (ACGTNU). If you were given RNA nucleotides, they should be counted as RNA nucleotides. If you were given DNA nucleotides, they should be counted as DNA nucleotides. Any N bases found should be counted also. Invalid bases are to be ignored in this dictionary.

codonComposition() – 10 pts

This dictionary returns counts of codons. Presume that sequences start in frame 1, accept the alphabet (ACGTUN) and store codons in RNA format, along with their counts. **Any codons found with invalid bases should be discarded**. Discard codons that contain N bases. This means that all codon counts are stored as RNA codons, even if the input happens to be DNA. If you discard a codon, make sure to not alter the frame of subsequent codons.

nucCount()

This returns an integer value, summing every valid nucleotide (ACGTUN) found. This value should exactly equal the sum over the nucleotide composition dictionary.

Design specification - genomeAnalyzer.py

This program must import your sequenceAnalysis module. It is responsible for preparing the summaries and final display of the data.

Input must be from STDIN

Your FastaReader object will read data from sys.stdin if it is not given a filename. Notice that the filename is specified as a parameter to your main() function. See the last line of the cell containing main() in this notebook:

- main('testGenome.fa').

When you move main() to genomeAnalyzer.py, you can delete the parameter:

- main() this will cause the filename parameter to be *None* (default parameter) and your FastaReader object will interpret a filename of *None* as a request to use sys.stdin. (see the doOpen() method for further info).

Your notebook will be used for inspections. For the notebook, you will need to specify the input filename. For genomeAnalyzer.py, you will need to delete the parameter.

Output format - 15 pts

The function to output the results of your analysis has specific formatting rules that you must follow to get full credit. These rules are as follows:

- First line: sequence length = X.XX Mb with two digits after the decimal and labeled Mb (you need to calculate the number of bases in Mb).
- second line is blank
- third line: GC content = XX.X% as a percentage with one digit after the decimal
- fourth line is blank
- lines 5 - 68 are the output statistics on relative codon usage for each codon ordered by codon within each amino acid group as follows:

XXX : A F (D)

where XXX is the three letters for an RNA codon, A is the 1-letter amino acid code, F is relative codon frequency, use (5.1f) for the format, and D is for codon count, use the format (6d). There is a single space between each of these fields. For example (this is not representative of any real genome):

sequence length = 3.14 Mb GC content = 60.2% UAA : + 32.6 (1041) UAG : + 38.6 (1230) UGA : + 28.8 (918) GCA : A 14.1 (10605) GCC : A 40.5 (30524) GCG : A 30.5 (22991) GCU : A 14.9 (11238) UGC : C 67.2 (4653) UGU : C 32.8 (2270) ...

To get full credit on this assignment, your code needs to:

- Run properly (execute and produce the correct output).
- Include any assumptions or design decisions you made in writing your code
- contain proper docstrings for the program, classes, modules and any public functions.
- Contain in-line comments

Extra credit - 10 pts possible

You now have a very powerful set of classes for evaluating genomes. Write a compareGenomes.py program that compares GC content, aaComposition and relative codon bias of 2 genomes. You will have a halophile genome and a hyperthermophile genome to compare.

Submit your code using canvas

Congratulations, you have finished your fourth lab assignment!

Inspection Intro

Provide design level information for your inspection team here. How do you input data to avoid having to read every sequence from the genome into memory? Where are your composition dictionaries initialized? How does data get added to those composition dictionaries?

Nuc Params

In [1]:
```
class NucParams:
    rnaCodonTable = {
        # RNA codon table
        # U
        'UUU': 'F', 'UCU': 'S', 'UAU': 'Y', 'UGU': 'C',  # 0x0
        'UUC': 'F', 'UCC': 'S', 'UAC': 'Y', 'UGC': 'C',  # 0x1
        'UUA': 'L', 'UCA': 'S', 'UAA': '-', 'UGA': '-',  # 0x2
        'UUG': 'L', 'UCG': 'S', 'UAG': '-', 'UGG': 'W',  # 0x3
        # C
        'CUU': 'L', 'CCU': 'P', 'CAU': 'H', 'CGU': 'R',  # 0x4
        'CUC': 'L', 'CCC': 'P', 'CAC': 'H', 'CGC': 'R',  # 0x5
        'CUA': 'L', 'CCA': 'P', 'CAA': 'Q', 'CGA': 'R',  # 0x6
        'CUG': 'L', 'CCG': 'P', 'CAG': 'Q', 'CGG': 'R',  # 0x7
        # A
        'AUU': 'I', 'ACU': 'T', 'AAU': 'N', 'AGU': 'S',  # 0x8
        'AUC': 'I', 'ACC': 'T', 'AAC': 'N', 'AGC': 'S',  # 0x9
        'AUA': 'I', 'ACA': 'T', 'AAA': 'K', 'AGA': 'R',  # 0xA
        'AUG': 'M', 'ACG': 'T', 'AAG': 'K', 'AGG': 'R',  # 0xB
        # G
        'GUU': 'V', 'GCU': 'A', 'GAU': 'D', 'GGU': 'G',  # 0xC
        'GUC': 'V', 'GCC': 'A', 'GAC': 'D', 'GGC': 'G',  # 0xD
        'GUA': 'V', 'GCA': 'A', 'GAA': 'E', 'GGA': 'G',  # 0xA
        'GUG': 'V', 'GCG': 'A', 'GAG': 'E', 'GGG': 'G'  # 0xG
        }
    dnaCodonTable = {key.replace('U','T'):value for key, value in rnaCodonTable.items()}

    def __init__ (self, inString=''):
        pass

    def addSequence (self, inSeq):
        pass
    def aaComposition(self):
        return self.aaComp
    def nucComposition(self):
        return self.nucComp
    def codonComposition(self):
        return self.codonComp
    def nucCount(self):
        return sum(self.nucComp.values())
```

FastAreader

In [34]:
```
import sys
class FastAreader :
    ''' 
    Define objects to read FastA files.

    instantiation:
    thisReader = FastAreader ('testTiny.fa')
    usage:
    for head, seq in thisReader.readFasta():
        print (head,seq)
    '''
    def __init__ (self, fname=None):
        '''constructor: saves attribute fname '''
        self.fname = fname

    def doOpen (self):
        ''' Handle file opens, allowing STDIN.'''
        if self.fname is None:
            return sys.stdin
        else:
            return open(self.fname)

    def readFasta (self):
        ''' Read an entire FastA record and return the sequence header/sequence'''
        header = ''
        sequence = ''

        with self.doOpen() as fileH:

            header = ''
            sequence = ''

            # skip to first fasta header
            line = fileH.readline()
            while not line.startswith('>') :
                line = fileH.readline()
            header = line[1:].rstrip()

            for line in fileH:
                if line.startswith ('>'):
                    yield header,sequence
                    header = line[1:].rstrip()
                    sequence = ''
                else :
                    sequence += ''.join(line.rstrip().split()).upper()

        yield header,sequence
```

Main

Here is a jupyter framework that may come in handy

In [ ]:
```
def main (fileName=None):
    myReader = FastAreader(fileName)
    myNuc = NucParams()
    for head, seq in myReader.readFasta() :
        myNuc.addSequence(seq)

    # sort codons in alpha order, by Amino Acid

    # calculate relative codon usage for each codon and print
    for nuc1 in nucs:
        ...
        print ('{:s} : {:s} ({:5.1f}) ({:6d})'.format(nuc, aa, val*100, thisCodonComp[nuc]))

if __name__ == "__main__":
    main('testGenome.fa') # make sure to change this in order to use stdin
```

Inspection Results

Who is your inspection team? What did they find? How did you decide to incorporate their suggestions?