**Genomic Data Manipulation**

BIO508 Spring 2014

Problems 04

**Python: Regular Expressions and Data Stores**

1. (0) In this problem set, you're going to search the *E. coli* genome for the Pribnow box or "-10" sequence in gene promoters, i.e. the six nucleotides TATAAT roughly 10 bases upstream of the transcription start site (analogous to the TATA box in eukaryotes). This will require A) downloading the *E. coli* genome, B) obtaining a file listing all *E. coli* gene start and stop sites, C) reading those files into Python data structures, D) extracting each gene's upstream promoter sequence, and E) printing out the ones that match the -10 consensus sequence (or its reverse complement). Your two main input files will be KEGG's version of the *E. coli* genome and its gene positions, which we've provided at:

<http://huttenhower.sph.harvard.edu/moodle/mod/resource/view.php?id=45>

<http://huttenhower.sph.harvard.edu/moodle/mod/resource/view.php?id=46>

These are just plain text files that we'll be reading into Python, so open them up from a web browser first and take a look at them yourself. Always look at your data! The genome is a single big nucleotide sequence in FASTA format:

<http://en.wikipedia.org/wiki/FASTA_format>

And the gene positions are in KEGG's POS format, which is reasonably similar to the standard GFF format:

<http://en.wikipedia.org/wiki/General_feature_format>

We'll write a whole set of utility functions for obtaining, parsing, and searching these files that'll culminate in a (partial) list of *E. coli*'s Pribnow-containing genes.

1. (2) Write a function called reFASTA that takes one argument, a string, and performs a regular expression search to A) identify whether the given string is a FASTA header line and B) if so, return the header itself (after stripping off the leading > and any leading/trailing whitespace). The header should of course be captured as a group for return, and you should return the empty string if the RE does not match. For those not familiar with FASTA files, take a look at the file linked above and the corresponding Wikipedia page. They're plain text files containing nucleotide or amino acid sequences, with specific sequences identified by header lines that must start with an initial > character at the beginning of the line, followed by an arbitrary identifier for the following sequence.

reFASTA( "> id" ) ⇒ "id"

reFASTA( ">id id" ) ⇒ "id id"

reFASTA( "> i dee " ) ⇒ "i dee"

reFASTA( "ATGGC" ) ⇒ ""

1. (4) Write a function called rePOS that takes one argument, a string, and performs a regular expression search to identify whether the string is a KEGG POS file format position (as described above). Positions are in the fourth column (which doesn't matter for this problem!) and are either of the form start..end (e.g. 123..456) or the form complement(end..start) (e.g. complement(123..456)). Note that POS files don't reverse the start and end indices for features on the reverse strand, which I find to be a bit confusing (but which also doesn't matter for this problem). Your RE should capture three groups: A) the initial word complement if present, B) the first number (before the periods), and C) the second number (after the periods). rePOS should return a list of all three groups if the RE matches and None otherwise (the first group can and almost certainly will be empty if the position is not a complement). Some hints to remember:
   1. In an RE, ? means, "zero or one of the preceding thing," and can also be read as, "the preceding thing is optional."
   2. To match a literal parenthesis character in an RE, escape it with a backslash. For example, r'\(' matches the string "(", and r'\(\)' matches "()". This prevents the RE from thinking you want the parenthesis to be part of a capture group; for example, r'(\(\))' will capture the string "()".
   3. Ditto for a period: to match a literal period character, escape it with a backslash. For example, r'\.' matches the string ".", and r'\..\.' matches a period followed by any one character followed by another period.
   4. Putting this all together, you want to match a regular expression that consists of:
      1. Either zero or one copies of the whole capture group containing complement.
      2. An optional literal left parenthesis.
      3. A capture group containing one or more digits.
      4. Two literal periods.
      5. A capture group containing one or more digits.
   5. For comparison purposes, the RE I used for this is exactly 30 characters long, excluding the r'', and contains four left parentheses, three right parentheses, two question marks, two pluses, two periods, and five backslashes. Oh, and the word "complement"; I'll let you figure out what the last two characters are.

rePOS( "1..23" ) ⇒ [None, "1", "23"]

rePOS( "complement(45..67)" ) ⇒ ["complement", "45", "67"]

rePOS( "compliementary" ) ⇒ None

1. (5) One more utility function before we start putting these all together! Since we're going to be looking through genome sequences for regulatory elements matching a particular pattern, let's write a promoter function that, given a gene's location in the genome, returns its upstream promoter sequence. promoter should take three arguments: strGenome, a string containing the entire genome sequence of an organism; aiPos, a list containing exactly two integers of the form [iFirst, iLast] (iFirst < iLast if the gene's on the forward strand, but iFirst > iLast if it's on the reverse; this takes the place of the "complement" encoding used above, and we'll see why in a bit); and iLength, an integer indicating how many nucleotides of promoter sequence to return (i.e. the length of the string). Although you *can* write promoter with a single line of code using some nutty Python trickery, I find it to be much more comprehensible using *seven lines*:
   1. If we've been asked for a promoter on the forward strand (that is, aiPos[0] < aiPos[1])...
   2. The index of the first nucleotide we do want, iBegin, is the gene start minus the requested length *minus one*, since gene positions are *inclusive* and *one indexed*, but Python strings are *zero indexed*.
   3. The index of the first nucleotide we don't want, iEnd, is the gene start *minus one*. For example, in the genome "abcdef", the gene "cd" is at position [3, 4], so the promoter sequence of length two is "ab" beginning at string index 0 and ending before string index 2.
   4. Otherwise, we're on the reverse strand (and the gene start aiPos[0] is greater than the end aiPos[1])...
   5. The index of iBegin is aiPos[0] (we don't need to add one because Python strings start at index 0 and gene positions start at nucleotide 1).
   6. The index of iEnd is aiPos[0] plus the requested length. For example, in the genome "abcdef", the gene "dc" is at position [4, 3], so the promoter sequence of length two is "ef" beginning at string index 4 and ending before string index 6.
   7. Finally, outside of the whole if block, return the string slice strGenome[iBegin:iEnd].

Note that we're not dealing with reverse complementation here at all, for reasons we'll see below. Just slice out and return a chunk of genome!

promoter( "abcdefghijklmnopqrstuvwxyz", [22, 23], 20 ) ⇒ "bcdefghijklmnopqrstu"

promoter( "abcdefghijklmnopqrstuvwxyz", [3, 2], 20 ) ⇒ "defghijklmnopqrstuvw"

promoter( "abcdefghijklmnopqrstuvwxyz", [8, 4], 10 ) ⇒ "ijklmnopqr"

promoter( "abcdefghijklmnopqrstuvwxyz", [3, 4], 2 ) ⇒ "ab"

1. (4) Ok, now you get to see why you're messing around with all of these utility functions! Two big ones left to go, a function that reads a FASTA file into Python and one that reads a POS file. We'll start with the latter (it's a bit simpler); write a function readPOS that takes one argument, an input file stream (*not* the string filename itself, but an already-opened stream!), and returns a dictionary of the form {strGene : [iFirst, iLast]}, where strGene is a string gene name, iFirst and iLast are integers, iFirst < iLast if strGene is on the forward strand, and iFirst > iLast if it's on the reverse. POS files contain five tab-delimited columns: a gene name, a comma-separated list of synonymous names, its start index, its position (as described in rePOS), and its length (note that this format is redundant, since the start and length can easily be calculated from the position). So for example, if readPOS read a POS file stream containing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| G1 | N1a,N1b | 1 | 1..2 | 2 |
| G2 | N2 | 3 | complement(3..4) | 2 |

then it would return the dictionary:

{"G1" : [1, 2], "G2" : [4, 3]}

Note that the gene synonyms, start, and length information are extraneous for our purposes and are thus generally ignored. Implement readPOS by filling in the blanks in the following code:

|  |
| --- |
| def readPOS( fileIn ):  hashRet = {}  for strLine in \_\_\_\_\_\_:  astrLine = \_\_\_\_\_\_\_.strip( ).\_\_\_\_\_( "\t" )  strID, strNames, strStart, strPos, strLength = \_\_\_\_\_\_\_\_  astrPOS = rePOS( \_\_\_\_\_\_ )  if astrPOS:  \_\_\_\_\_\_\_, strOne, strTwo = \_\_\_\_\_\_\_  iOne, iTwo = \_\_\_(strOne), \_\_\_(\_\_\_\_\_\_)  \_\_\_\_\_\_\_[strID] = [\_\_\_\_, \_\_\_\_] if strComp else [iOne, iTwo]  return \_\_\_\_\_\_\_ |

Note that it's probably easiest to write and test this function using a small, handmade POS file rather than the whole big e.coli.pos. You can create such a thing either by using Excel or OpenOffice and saving as tab-delimited text, or by typing text and tabs directly into jEdit and saving the file. Don't forget to call readPOS( open( strFile ) ) rather than passing the filename in directly; this format makes it easy to test using calls of the form readPOS( sys.stdin ) if you'd prefer (which I would!)

1. (5) Next, you should write readFASTA, which works exactly like readPOS, but for FASTA files. readFASTA should take one argument, an input file stream containing a FASTA file, and return a dictionary pairing sequence IDs from the header lines with their actual sequences. FASTA files are *not* tab-delimited, but instead each line (except headers) contains a chunk of sequence (that has to be concatenated with the preceding and/or following lines to create the whole sequence). Header lines, which start with >, provide an identifier for the subsequent sequence. To make this concrete, if you input the following FASTA file to readFASTA:

|  |
| --- |
| >G1  ATTA  TTAT  >G2  CGAT  CGAT  TTTT |

then it would return the dictionary:

{"G1" : "ATTATTAT", "G2" : "CGATCGATTTTT"}

The general process for doing this will again be to loop through each line of the file. If it's a header, we'll save any sequence we've stored up for the previous gene, remember the new gene ID, and start accumulating a new sequence. If it's not a header, we'll concatenate the line (sans trailing whitespace) onto the accumulating sequence for the current gene. Implement this function by filling in the blanks:

|  |
| --- |
| def readFASTA( fileIn ):  hashRet = {}  strID = strSeq = ""  for \_\_\_\_\_\_\_ in fileIn:  strLine = strLine.\_\_\_\_\_( )  strNew = reFASTA( \_\_\_\_\_\_\_ )  if strNew:  if \_\_\_\_\_\_:  hashRet[\_\_\_\_\_] = \_\_\_\_\_\_  strID = strNew  strSeq = \_\_  else:  \_\_\_\_\_\_ += \_\_\_\_\_\_\_  if strSeq:  hashRet[strID] = strSeq  return \_\_\_\_\_\_\_ |

* 1. (2) In a comment or docstring in your Python file, explain why the last two lines preceding the return are necessary.

1. (3) Ok, one last function that will take advantage of the hard work performed by readPOS and readFASTA. Write a function findS70 that takes two arguments, hashFASTA (a dictionary as returned by readFASTA) and hashPOS (a dictionary as returned by readPOS). In each line of the output file this function will print out each gene ID whose promoter contains the -10 sequence TATAAT within 20bp of its transcription start site (or the reverse complement ATTATA if the gene's on the reverse strand). Also included in the same line would be that gene’s starting and ending indices and its promoter sequence. We'll use two Python tricks in order to make this function compact:
   1. You can iterate over the key/value pairs in a dictionary using the construct for key, value in hash.items( ). This combines parallel assignment using variable names separated by commas with the items collection, which is a list of lists of length two containing all key/value pairs in a dictionary (just like keys( ) is a list of keys alone and values( ) a list of values).
   2. You've seen this before, but don't forget that you can use a single-line ternary form of if of the form *body1* if *test* else *body2*. This is more compact than the four-line version if *test*:/*body1*/else:/*body2* and can be used anyplace where a single expression is expected.

Combining these two reminders, implement findS70 by filling in the following blanks:

|  |
| --- |
| def findS70( hashFASTA, hashPOS ):  for strID, strGenome in hashFASTA.items( ):  for strGene, aiPos in hashPOS.\_\_\_\_\_\_\_\_:  strPromoter = promoter( \_\_\_\_\_\_\_\_\_, \_\_\_\_\_, 20 )  strRE = r'TATAAT' if ( \_\_\_\_\_\_\_\_ < \_\_\_\_\_\_\_\_ ) else r'ATTATA'  if re.\_\_\_\_\_\_( strRE, strPromoter ):  print( "\t".join( [strGene, \_\_\_(aiPos[0]), \_\_\_(aiPos[1]),  strPromoter] ) ) |

1. (1) In a comment or docstring in your Python file, explain why this function is called findS70.
2. (2) Implement findS70 without using the re module (those aren't very interesting search patterns as far as REs go, are they?). Name your new function findS70NoRe.
3. Congratulations - you've filled up your punch card, and the next one's free:

|  |
| --- |
| def main( ):  if len( sys.argv ) < 2:  raise Exception( "Usage: problems04.py <e.coli.pos> < <e.coli.fasta>" )  strPOS = sys.argv[1]  hashFASTA = readFASTA( sys.stdin )  hashPOS = readPOS( open( strPOS ) )  findS70( hashFASTA, hashPOS ) |

* 1. (2) Run this main function to find the -10 sequences in the *E. coli* genome:

python problems04.py e.coli.pos < e.coli.fasta

List what you find in a comment or docstring in your submitted file. The output should contain four tab-delimited columns as printed above in findS70. Hint: I get exactly 12 hits when I run this, although your mileage may vary slightly.

* 1. (2) This binding site is very common in prokaryotic genomes; based on the search method we're using here, why might we be getting so few hits?
  2. (6) Take a close look at the *E. coli* data on KEGG and the corresponding genome annotations at UCSC (<http://microbes.ucsc.edu>) and BioMart (<http://www.biomart.org>). Also check out PromEC (<http://margalit.huji.ac.il/promec/>). Why are we *really* not getting as many hits as you might expect?
  3. (C:\Users\eblis\Desktop\pipette.png) Find and parse another set of *E. coli* annotations that give you a more accurate overview of the -10 promoter sequences in the genome.

1. (C:\Users\eblis\Desktop\pipette.png) Change the implementation of findS70 so that it looks for and scores a full PSSM for the -10 sequence rather than the hard-coded TATAAT consensus. How often do you expect this sequence to occur in the genome by chance? How often do you expect it to occur within a 20bp promoter window? Don't forget reverse complements!