Parsing and processing with Python and R

We'll return to the rawseqs.sam file that we produced on Monday (the alignment of the CYP2D6 primers to chr22).

Use `less` to explore the sam file format. The -S option will make it easier to view a very wide tab-delimited file with `less`.

sam (sequence alignment/map) format

header with sequence names (@SQ) and information about the program used (@PG), plus a bunch of tab-delimited data:

QNAME	Query template/pair NAME
FLAG	bitwise FLAG
RNAME	Reference sequence NAME
POS	1-based leftmost POSition/coordinate of clipped sequence
MAPQ	MAPping Quality (Phred-scaled)
CIGAR	extended CIGAR string
RNEXT	mate reference sequence name ('=' if same as RNAME)
PNEXT	1-based mate poistion
TLEN	inferred Template LENgth (insert size)
SEQ	query SEQuence on the same strand as the reference
QUAL	query QUALity (ASCII-33 gives the Phred base quality)
OPT	variable OPTional fields in the format TAG:VTYPE:VALUE

The fourth column contains the position of each alignment.

Exercise:
Write a Python program to determine whether there are any sequences that occur more than once in this file.

Tips:
Initialize a dictionary. You're going to use the sequences as keys and the number of times they've been observed, as values.
For each line, check to see whether the sequence in that sam record has been seen before. Is there a key in the dictionary that matches this sequence?
If it is already in the dictionary, you can increment the value for that key, and if not, you can add it with the value "1".

```
#!/usr/bin/python

import sys

seqnames = {}
samfile = file(sys.argv[1], "r")
for line in samfile:
```

```
    fields = line.split("\t")
    # check to make sure that there are enough columns before
    # we check the 10th one
    if len(fields) > 10:
        # if the sequence is present in the dictionary already,
        # has_key will be true
        if seqnames.has_key(fields[9]):
            seqnames[fields[9]] += 1
        #so if it's not present, add it
        else:
            seqnames[fields[9]] = 1

#loop through all of the keys and print those whose values (counts)
are greater than 1
for keys in seqnames:
    if seqnames[keys] > 1:
        print seqnames[keys], keys

samfile.close()
```

Exercise:
Write a one-line unix command to count the number of times each sequence occurs in the file.

Tips:
use the uniq utility to summarize the column of aligned sequences.
the uniq command has a -c option, to count the number of times a value is seen
the uniq command only works as expected, if you have sorted data.

More practice in handling files in Python. Remember that the command file opens the designated file (passed as a string or variable). There is another piece to this command, that specifies the mode in which the file is opened, e.g. this command opens a file named filename.txt in read mode:

```
filehandle = file("filename.txt", "r")
```

'r' --> read only
'w' --> write only (overwrite a file with the same name)
'a' --> append to the existing file (do not overwrite)
'r+' --> open a file both in read and write mode
'b' --> binary mode
't' --> ascii mode

Exercise:
Write a Python script that takes two command line arguments. The first is the name of an existing file, and the second is the desired name for an output file. The program should count the number of fasta-formatted sequences in the input file, by opening the file and counting how

many lines start with “>” (fasta deflines), and then it should write the final count into a new output file, using the name passed from the command line.

Tips:
The string object in Python has a built in function, startswith, that is called like line.startswith(“N”) and will return true or false, according to whether the letters in quotes are at the beginning of the string.
You’ll need to use a counter, so remember to initialize a variable (set it to 0) that will act as a counter as your program loops through the lines of the file.

R is a statistical programming language and has some quirks but is well worth the effort!

It is a free, open source, widely available, well-supported statistical computing & graphics language. Over the last 10 years it has become useful, relevant, and now critical to computational genomics, thanks in large part to the Bioconductor project.

Stated goals of the Bioconductor project are:

To provide widespread access to a broad range of powerful statistical and graphical methods for the analysis of genomic data.
To facilitate the inclusion of biological metadata in the analysis of genomic data, e.g. literature data from PubMed, annotation data from Entrez genes.
To provide a common software platform that enables the rapid development and deployment of extensible, scalable, and interoperable software.
To further scientific understanding by producing high-quality documentation and reproducible research.
To train researchers on computational and statistical methods for the analysis of genomic data.

http://www.r-project.org/ R project home page

http://www.bioconductor.org/ Bioconductor -- biology/genomics/biostatistics treasure trove

http://www.rstudio.com/ RStudio -- graphical interface for the R language

Start RStudio (Applications -> RStudio)
In RStudio, use Session -> Set Working Directory and choose the directory holding your new file. This can also be done from the R command prompt, using setwd.

Let’s explore the nuts and bolts of R for a minute. R makes a very good calculator (better than Python, actually, as it stores floating points by default).

```
> 1+1
```

```
[1] 2
> 3/4
[1] 0.75
> 9999999/10000000
[1] 0.9999999
> 1+2*4
[1] 9
> (1+2)*4
[1] 12
> 42
[1] 42
```

and some comparisons:

```
> 1 = 1
Error in 1 = 1 : invalid (do_set) left-hand side to assignment
> 1 == 1
[1] TRUE
> 1 != 1
[1] FALSE
> 1 > 1
[1] FALSE
> 1 < 1
[1] FALSE
> 1 >= 1
[1] TRUE
> x
Error: object 'x' not found
```

If R cannot evaluate something that you have typed, it complains. At times these errors are useful.

```
> x <- 3
> x
[1] 3
> y <- 4
> y
[1] 4
> x + y
[1] 7
> z <- x + y
> z
[1] 7
> y <- -100
> z
[1] 7
```

```
> z == 7
[1] TRUE
> Z == 7
Error: object 'Z' not found
> z = 6
> z == 7
[1] FALSE
```

Let's look at those blast results again.

```
> tab <- read.table("primerblast.txt", stringsAsFactors=F)
```

We are using the stringsAsFactors option in there, that we'll discuss later. For now, explore the object that was created.

Note that RStudio completes parentheses and quotes for you, and highlights a quote or parenthesis if it is the second in a pair.

What did we do? Let's look at the tab object. The head command works in R, too, but you need parentheses:

```
> head(tab)
                         V1    V2      V3 V4 V5 V6 V7 V8       V9       V10      V11  V12
1 chr22:42126499-42126521 chr22 100.000 23  0  0  1 23 42126500 42126522 2.04e-05 42.8
2 chr22:42126499-42126521 chr22 100.000 23  0  0  1 23 42140204 42140226 2.04e-05 42.8
3 chr22:42126499-42126521 chr22 100.000 16  0  0  1 16 39755857 39755842 1.30e-01 30.1
4 chr22:42126499-42126521 chr22 100.000 14  0  0  7 20 15336511 15336498 1.60e+00 26.5
5 chr22:42126499-42126521 chr22 100.000 14  0  0  7 20 16206870 16206857 1.60e+00 26.5
6 chr22:42126499-42126521 chr22  89.474 19  2  0  2 20 24917262 24917280 1.60e+00 26.5
```

So it looks like . . . a table. But R knows its way around tables, so this is a more complex structure than it seems; in fact, it's a data frame.

```
> str(tab)
'data.frame': 247 obs. of  12 variables:
 $ V1 : chr  "chr22:42126499-42126521" "chr22:42126499-42126521"
"chr22:42126499-42126521" "chr22:42126499-42126521" ...
 $ V2 : chr  "chr22" "chr22" "chr22" "chr22" ...
 $ V3 : num  100 100 100 100 100 ...
 $ V4 : int  23 23 16 14 14 19 17 13 13 16 ...
 $ V5 : int  0 0 0 0 0 2 1 0 0 1 ...
 $ V6 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ V7 : int  1 1 1 7 7 2 2 7 1 3 ...
 $ V8 : int  23 23 16 20 20 20 18 19 13 18 ...
 $ V9 : int  42126500 42140204 39755857 15336511 16206870 24917262 25287640 27112356
27432697 28933321 ...
 $ V10: int  42126522 42140226 39755842 15336498 16206857 24917280 25287624 27112368
27432709 28933306 ...
 $ V11: num  2.04e-05 2.04e-05 1.30e-01 1.60 1.60 1.60 1.60 5.50 5.50 5.50 ...
```

```
 $ V12: num  42.8 42.8 30.1 26.5 26.5 26.5 26.5 24.7 24.7 24.7 …
```

What is that str command, by the way?

```
> ?str
> help(str)
```

will do the same thing, and both will tell you that str displays the structure of an R object.

Like most languages, R has several types/modes of objects, including logical, integer, double, raw, complex, character, and more. R also has classes, including matrix, array, vector, data.frame, and many more. Classes are more complex containers for objects and are often extended by particular R packages.

See http://stat.ethz.ch/R-manual/R-devel/doc/manual/R-lang.html#Objects for more details.

A data frame is a lot like a matrix except that each "column" can have a different type (character, numeric etc). Data frames have rows and columns, just like a matrix.

```
> nrow(tab)
[1] 247
> ncol(tab)
[1] 12
> dim(tab)
[1] 247  12
```

We can access elements of the data frame in several different ways.

```
> tab[1,1]
[1] "chr22:42126499-42126521"
> tab[1,2]
[1] "chr22"
> tab[1,3]
[1] 100
> tab[1,4]
[1] 23
> tab[1,]
                     V1    V2  V3 V4 V5 V6 V7 V8       V9       V10       V11  V12
1 chr22:42126499-42126521 chr22 100 23  0  0  1 23 42126500 42126522 2.04e-05 42.8

> tab[,4]
  [1] 23 23 16 14 14 19 17 13 13 16 23 13 13 13 23 23 23 14 14 22 14 17 13 13 13 13 13
 [28] 13 13 13 23 23 23 16 23 23 23 19 16 18 15 15 14 14 17 14 14 14 14 14 14 14 14 19
 [55] 14 14 13 13 18 13 13 13 13 16 16 13 13 13 18 16 13 13 13 18 13 18 13 13 13 13 13
 [82] 13 13 18 13 16 13 13 13 13 13 13 18 18 13 23 21 18 14 13 16 16 16 13 16 18 16 18
[109] 13 13 23 14 14 18 13 13 20 13 23 21 21 16 19 14 14 13 18 16 13 13 23 23 23 19 23
[136] 16 16 23 22 18 18 18 18 18 15 14 16 16 13 13 18 16 23 13 13 23 21 21 21 14 14 23
[163] 19 17 17 24 14 16 13 13 13 13 13 13 16 13 13 13 23 17 21 13 23 23 23 15 14 14 14
[190] 19 17 16 16 13 16 13 23 23 23 15 15 15 15 15 18 14 17 17 17 17 19 19 14 14 17 19
```

```
[217] 19 14 14 16 13 16 13 16 13 16 13 16 13 13 16 13 13 13 13 13 13 13 23 23 14 14
[244] 13 23 18 13
```

RStudio has a terrific built-in viewer for tabular data:

```
> View(tab)
```

This data frame really needs some column names.

```
> colnames(tab)
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11" "V12"
> currentnames <- colnames(tab)
> currentnames
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11" "V12"
> str(currentnames)
 chr [1:12] "V1" "V2" "V3" "V4" "V5" "V6" "V7" "V8" "V9" "V10" ...
> currentnames[3]
[1] "V3"
> colnames(tab) <- c("primername", "subject", "percentid", "length",
"mismatches", "gaps", "querystart", "querystop", "subjectstart",
"subjectstop", "evalue", “bit_score")
> head(tab)
```

Now we can use some convenient and specialized notation. Instead of percent identity being tab[,3] (which is positionally determined and dangerous to rely on), we can use its name.

```
> head(tab[,3])
> head(tab$percentid)
```

We can subset this table in many ways: “physically” (rows/columns), logically (satisfying some condition), or a combination of both.

Make a new table, containing only alignments with no gaps or mismatches:

```
> summary(tab$percentid)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  82.61   94.12  100.00   97.19  100.00  100.00
> sum(tab$percentid==100)
[1] 157
> sel <- tab$percentid==100
> head(sel)
[1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
> length(sel)
[1] 247
> tabperfect <- tab[sel,]
```

```
> dim(tabperfect)
[1] 157  12
> head(tabperfect)
              primername subject percentid length mismatches gaps querystart querystop
1 chr22:42126499-42126521   chr22       100     23          0    0          1        23
2 chr22:42126499-42126521   chr22       100     23          0    0          1        23
3 chr22:42126499-42126521   chr22       100     16          0    0          1        16
4 chr22:42126499-42126521   chr22       100     14          0    0          7        20
5 chr22:42126499-42126521   chr22       100     14          0    0          7        20
8 chr22:42126499-42126521   chr22       100     13          0    0          7        19
  subjectstart subjectstop   evalue bit_score
1     42126500    42126522 2.04e-05      42.8
2     42140204    42140226 2.04e-05      42.8
3     39755857    39755842 1.30e-01      30.1
4     15336511    15336498 1.60e+00      26.5
5     16206870    16206857 1.60e+00      26.5
8     27112356    27112368 5.50e+00      24.7
> summary(tabperfect$percentid)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    100     100     100     100     100     100
> tabperfect <- tab[tab$percentid==100,]
```

We can combine data by rows and columns:

```
> smallertab <- cbind(tabperfect$subject, tabperfect$bit_score)
```

But this did something weird.

```
> head(smallertab)
     [,1]    [,2]
[1,] "chr22" "42.8"
[2,] "chr22" "42.8"
[3,] "chr22" "30.1"
[4,] "chr22" "26.5"
[5,] "chr22" "26.5"
[6,] "chr22" "24.7"
> str(smallertab)
 chr [1:157, 1:2] "chr22" "chr22" "chr22" "chr22" "chr22" "chr22" …
```

What are those quotes? and what's that structure?

We could also use the functions that are intended for data frames, as these will preserve the mode of each vector:

```
> smallertab <- subset(tabperfect, select=c(subject, bit_score))
> head(smallertab)
  subject bit_score
1   chr22      42.8
2   chr22      42.8
```

```
3    chr22      30.1
4    chr22      26.5
5    chr22      26.5
8    chr22      24.7
```

Exercise:
Read in the file, rawseqs.sam.
Tips:
R won't like the file format! using the skip option you can tell it to skip a given number of leading lines (this file has three short lines in the header). Also, the sam format provides for an arbitrary number of comment fields. You will have to use the fill option (fill=T) to tell R to just fill any short lines with NAs to make them all the same length.
also, be sure to use stringsAsFactors=F. If time permits we'll explore why this is needed.

Exercise:
Give your data frame column names, at least up through column 11.
Make a new data frame that contains only the columns with chromosome, position, and the aligned sequences.

Exercise:
Make a data frame with the read name and alignment position of every read that aligns between chr22 42141116 and 42150170 (inclusive), but only if the alignment is 51 characters long. This can be a series of select statements and subsetting commands.

Exercise:
Do any sequences occur more than once in this file?

Tips:
R has a "unique" function but it doesn't do counting like the unix function does. R does have a "table" function that will be somewhat helpful.
Try looking through the help for the "unique" function. There is a related function that will tell you which entries occur more than once.