Data Processing with Python

In this exercise, you will build a crude one-off parser for VCF data from the 1000 Genomes Project. To run in a reasonable time, we'll only work with a small portion of chromosome 22. (Note that the 1000 Genomes Project data is large, even when summarized in VCF. To deal with the data efficiently, you should use VCFtools if possible.)

The data are in the directory, dataprocessing. To switch to that directory in the terminal, use the command `cd dadiExercise/dataprocessing`

The VCF file begins with a long header, followed by thousands of long lines with the data. For something easier to work with, let's extract the first 300 lines into a smaller file:
`head –n 300 data.vcf > tiny.vcf`

We'll develop our code using `tiny.vcf`, so we can debug quickly. Open a new terminal window, change to the `dataprocessing` directory, and run `nano tiny.vcf` to view the file.

This VCF file includes data for all the individuals in the 1000 Genomes Project, organized by id. Our first task will be to determine which IDs we want data for. To do that, we'll use the `sample_info.txt` file provided by the project. This is a tab-separated file listing all individuals in the project.

In a new terminal window, change to the `dataprocessing` directory and edit a new script file: `nano parse_vcf.py`. This is where you'll write your Python script. Open another terminal window, change to the `dataprocessing` directory, and start iPython with `ipython --pylab`. In the iPython window, you can run your script at any time with `%run parse_vcf.py`. (Be sure to save your script when you change it!) If your script hangs, use `ctrl-C` to stop execution.

Remember that indentation is part of the Python syntax. It doesn't matter how many spaces you indent the lines inside a loop, but you must be consistent, otherwise you will get SyntaxErrors.

If you get lost and can't find your bug, you can use `nano` or `cat` to look at the example script in `dataprocessing/parse_vcf_example.py`. But you'll learn more if you don't peak unless you really need it!

1. To open `sample_info.txt`, add a line
   `fid = file('sample_info.txt', 'rU')` to your script[1]. The first line of the file is a header line. Read that line by adding the command
   `line = fid.readline()` to your script.

   Run your script with `%run`. Check the value of the variable `line`. You should see the header line, indicating the different columns in the file.

2. Next we'll use a `for` loop to run through all the remaining lines in the file. We'll also `split` each line by tabs. Add the following lines to your script.
   ```
   for line in fid:
       spl = line.split('\t')
   ```

[1] The 'U' option tells python to open the file so that either DOS or Unix line returns will be read correctly.

Run your script with `%run`. You should see that `spl` is now a list with elements corresponding to the columns in the data file, for the last row of data.

3. We only want to keep samples from the Yoruba population, denoted YRI. To do so, create an empty list *before your for loop*, using `tokeep = []`. Next, we need to use an if statement to keep only relevant rows. The population id is the 3rd entry in each line, and the sample id is the 1st entry. In your for loop, after the split line, add
```
if spl[2] == 'YRI':
        tokeep.append(spl[0])
```
Run your script. Your tokeep list should be 186 elements long, with the last element being `'NA19258'`.

Now we turn to the VCF file itself. Our first task will be to figure out which columns in the file correspond to Yoruba individuals.

4. At the new end of your script, open the VCF file using `fid = file('tiny.vcf')`. First we'll read through all lines in the file, looking for the line that begins with `#CHROM`.
```
line = ''
while not line.startswith('#CHROM'):
    line = fid.readline()
```
Run your script. Check that the variable `line` holds what you expected.

5. We'll loop through the elements in that line, checking whether any of them match the samples we want to keep. If they do, we'll save the column number to a variable called `sample_columns`. Add these lines to your script.
```
for ii, value in enumerate(line.split('\t')):
     if value in tokeep:
        sample_columns.append(ii)
```
Run your script. Check that sample_columns contains 186 entries.

Now we can parse the actual data lines. To input into our data into dadi, we create a dictionary that we'll call `data_dict`. The keys will identify each SNP, and the values will be of the form `{'calls': {'YRI': (173, 37)}, 'segregating': ['G', 'T']}`.

6. We'll carry on reading the file we've already opened. We'll read each line, split it, and pull out the two alleles.
```
for line in fid:
    spl = line.split('\t')
    ref, alt = spl[3:5]
```
Run your script. Check the `spl` result and check that `ref` and `alt` are correct.

7. We only want to keep simple SNPs, which have reference and alternate alleles of length 1. To do so, we'll use an `if` statement. Inside your loop, add the lines
```
if len(ref) != 1 or len(alt) != 1:
        continue
```
The `continue` command tells Python to skip back to the top of the for loop (and thus start processing the next line, if the `if` statement is satisfied.)

The keys in our dictionary will identify the chromosome and the position of the SNP. To create the string with this information in it, add the line below after your `if` statement.

```
key = '{0}_{1}'.format(spl[0], spl[1])
```

The data for each SNP goes into a dictionary, so next we'll create it and add it to data_dict.

```
snp_data = {}
data_dict[key] = snp_data
```

Run your script. Check that `data_dict` has length 43.

8. Now we need to collect data for each SNP. We've already pulled out the segregating alleles, so just add that to `snp_data`.

```
snp_data['segregating'] = [ref, alt]
```

Run your script. Check that `data_dict` entries now contain the segregating data.

9. The last and trickiest part is collecting allele counts for all the individuals in the `tokeep` list. In the column for each individual, the genotype is of the form `0|1`, where `|` separates the two alleles in the individual, `0` denotes reference, and `1` denotes alternate allele. To do so, add the following lines inside your for loop over rows. (Remember the all the code inside that loop needs to be indented another level to be inside the loop.)

```
totrefcount, totaltcount = 0,0
for ii in sample_columns:
    a1, a2 = spl[ii].split('|')
    if a1 == '0':
        totrefcount += 1
    if a1 == '1':
        totaltcount += 1
    if a2 == '0':
        totrefcount += 1
    elif a2 == '1':
        totaltcount += 1
snp_data['calls'] = {'YRI':[totrefcount, totaltcount]}
```

Run your script. Check that `data_dict` now contains call information. You should find that only 4 SNPs are segregating in the YRI population, with the highest alternative allele frequency being 10.

10. Change your script to run with `data.vcf` instead of `tiny.vcf`. At the end of your script use dadi to generate a frequency spectrum from your `data_dict` and plot it.

```
import dadi
fs = dadi.Spectrum.from_data_dict(data_dict,
        pop_ids=['YRI'],projections=[216], polarized=False)
dadi.Plotting.plot_1d_fs(fs)
dadi.Plotting.pylab.show()
```

Run your script. You should get a reasonably nice plot of your frequency spectrum.

11. OPTIONAL: Add logic to you script so SNPs that aren't segregating in YRI aren't added to the data_dict. This is just one more `if` statement.

12. OPTIONAL: Add `outgroup_allele` information to your data dict. You can get this from the `AA` field of the info field of the VCF file. Be sure to convert the outgroup calls to upper case, so dadi can match them to the segregating alleles.