Command-line Arguments

If you have not already, I encourage you to copy the "new_py.py" script into your `$PATH` and then execute it with the `-a` argument to start a new script with `argparse`:

```
$ ./new_py.py -a test
Done, see new script "test.py."
```

If you check out the new script, it has a `get_args` function that will show you how to create named arguments for strings, integers, booleans, and positional arguments:

```
$ cat -n test.py
     1    #!/usr/bin/env python3
     2
     3    import argparse
     4    import os
     5    import sys
     6
     7    # --------------------------------------------------
     8    def get_args():
     9        parser = argparse.ArgumentParser(description='Argparse Python script')
    10        parser.add_argument('positional', metavar='str', help='A positional argument')
    11        parser.add_argument('-a', '--arg', help='A named string argument',
    12                            metavar='str', type=str, default='')
    13        parser.add_argument('-i', '--int', help='A named integer argument',
    14                            metavar='int', type=int, default=0)
    15        parser.add_argument('-f', '--flag', help='A boolean flag',
    16                            action='store_true')
    17        return parser.parse_args()
    18
    19    # --------------------------------------------------
    20    def main():
    21        args = get_args()
    22        str_arg = args.arg
    23        int_arg = args.int
    24        flag_arg = args.flag
    25        pos_arg = args.positional
    26
    27        print('str_arg = "{}"'.format(str_arg))
    28        print('int_arg = "{}"'.format(int_arg))
    29        print('flag_arg = "{}"'.format(flag_arg))
    30        print('positional = "{}"'.format(pos_arg))
    31
    32    # --------------------------------------------------
    33    if __name__ == '__main__':
```

```
    34        main()
```

If you run without any arguments, you get a nice usage statement:

```
$ ./test.py
usage: test.py [-h] [-a str] [-i int] [-f] str
test.py: error: the following arguments are required: str
$ ./test.py foobar
str_arg = ""
int_arg = "0"
flag_arg = "False"
positional = "foobar"
$ ./test.py -a XYZ -i 42 -f foobar
str_arg = "XYZ"
int_arg = "42"
flag_arg = "True"
```

Please RTFM ([https://docs.python.org/3/library/argparse.html](https://docs.python.org/3/library/argparse.ht
to find all the other Fine things you can do with argparse.

CSV Files

Delimited text files are a standard way to distribute non/semi-hierarchical data – e.g., records that can be represented each on one line. (When you get into data that have relationships, e.g., parents/children, then structures like XML and JSON are more appropriate, which is not to say that people haven't sorely abused this venerable format, e.g., GFF3.) Let's first take a look at the `csv` module in Python ([https://docs.python.org/3/library/csv.html](https://docs.python.org/3/library/csv.html%29)
to parse the output from Centrifuge ([http://www.ccb.jhu.edu/software/centrifuge/](http://www.ccb.jhu.edu/soft

For this, we'll use some data from a study from Yellowstone National Park ([https://www.imicrobe.us/sample/view/1378](https://www.imicrobe.us/sample/view/1378%29%29.
For each input file, Centrifuge creates two output files: 1) a file ("YELLOWSTONE_SMPL_20723.sum") showing the taxonomy ID for each read it was able to classify and 2) a file ("YELLOWSTONE_SMPL_20723.tsv") of the complete taxonomy information for each taxonomy ID. One record from the first looks like this:

```
      readID: Yellowstone_READ_00007510
       seqID: cid|321327
       taxID: 321327
       score: 640000
2ndBestScore: 0
   hitLength: 815
 queryLength: 839
  numMatches: 1
```

One from the second looks like this:

```
          name: synthetic construct
         taxID: 32630
       taxRank: species
    genomeSize: 26537524
      numReads: 19
numUniqueReads: 19
     abundance: 0.0
```

Read Count by taxID

Let's write a program that shows a table of the number of records for each "taxID":

```
$ cat -n read_count_by_taxid.py
     1    #!/usr/bin/env python3
     2    """Counts by taxID"""
     3
     4    import csv
     5    import os
     6    import sys
     7    from collections import defaultdict
     8
     9    args = sys.argv[1:]
    10
    11    if len(args) != 1:
    12        print('Usage: {} SAMPLE.SUM'.format(os.path.basename(sys.argv[0])))
    13        sys.exit(1)
    14
    15    sum_file = args[0]
    16
    17    _, ext = os.path.splitext(sum_file)
    18    if not ext == '.sum':
    19        print('File extention "{}" is not ".sum"'.format(ext))
    20        sys.exit(1)
    21
    22    counts = defaultdict(int)
    23    with open(sum_file) as csvfile:
    24        reader = csv.DictReader(csvfile, delimiter='\t')
    25        for row in reader:
    26            taxID = row['taxID']
    27            counts[taxID] += 1
    28
    29    print('\t'.join(['count', 'taxID']))
```

```
    30    for taxID, count in counts.items():
    31        print('\t'.join([str(count), taxID]))
```

As always, it prints a "usage" statement when run with no arguments. It also uses the `os.path.splitext` function to get the file extension and make sure that it is ".sum." Finally, if the input looks OK, then it uses the `csv.DictReader` module to parse each record of the file into a dictionary:

```
$ ./read_count_by_taxid.py
Usage: read_count_by_taxid.py SAMPLE.SUM
$ ./read_count_by_taxid.py YELLOWSTONE_SMPL_20723.tsv
File extention ".tsv" is not ".sum"
$ ./read_count_by_taxid.py YELLOWSTONE_SMPL_20723.sum
count    taxID
80    321332
6432    321327
19    32630
```

That's a start, but most people would rather see the a species name rather than the NCBI taxonomy ID, so we'll need to go look up the taxIDs in the ".tsv" file:

```
$ cat -n read_count_by_tax_name.py
     1    #!/usr/bin/env python3
     2    """Counts by tax name"""
     3
     4    import csv
     5    import os
     6    import sys
     7    from collections import defaultdict
     8
     9    args = sys.argv[1:]
    10
    11    if len(args) != 1:
    12        print('Usage: {} SAMPLE.SUM'.format(os.path.basename(sys.argv[0])))
    13        sys.exit(1)
    14
    15    sum_file = args[0]
    16
    17    basename, ext = os.path.splitext(sum_file)
    18    if not ext == '.sum':
    19        print('File extention "{}" is not ".sum"'.format(ext))
    20        sys.exit(1)
    21
    22    tsv_file = basename + '.tsv'
    23    if not os.path.isfile(tsv_file):
    24        print('Cannot find expected TSV "{}"'.format(tsv_file))
    25        sys.exit(1)
    26
```

```
    27    tax_name = {}
    28    with open(tsv_file) as csvfile:
    29        reader = csv.DictReader(csvfile, delimiter='\t')
    30        for row in reader:
    31            tax_name[row['taxID']] = row['name']
    32
    33    counts = defaultdict(int)
    34    with open(sum_file) as csvfile:
    35        reader = csv.DictReader(csvfile, delimiter='\t')
    36        for row in reader:
    37            taxID = row['taxID']
    38            counts[taxID] += 1
    39
    40    print('\t'.join(['count', 'taxID']))
    41    for taxID, count in counts.items():
    42        name = tax_name.get(taxID) or 'NA'
    43        print('\t'.join([str(count), name]))
$ ./read_count_by_tax_name.py YELLOWSTONE_SMPL_20723.sum
count    taxID
6432    Synechococcus sp. JA-3-3Ab
80    Synechococcus sp. JA-2-3B'a(2-13)
19    synthetic construct
```

tabchk

A huge chunk of my time is spent doing ETL operations – extract, transform, load – meaning someone sends me data (Excel or delimited-text, JSON/XML), and I put it into some sort of database. I usually want to inspect the data to see what it looks like, and it's hard to see the data when it's in columnar format. I'd rather see it formatted vertically. E.g., here is an export of some iMicrobe sample annotations:

```
$ tabchk.py -d imicrobe-blast-annots.txt
// ****** Record 1 ****** //
sample_id             : 1
project_name          : Acid Mine Drainage Metagenome
ontologies            : ENVO:01000033 (oceanic pelagic zone biome),ENVO:00002149 (sea water)
project_id            : 1
source_mat_id         : 33
sample_type           : metagenome
sample_description    : ACID_MINE_02 - 5-Way (CG) Acid Mine Drainage Biofilm
sample_name           : ACID_MINE_02
collection_start_time : 2002-03-01 00:00:00.0
collection_stop_time  : 2002-03-01 00:00:00.0
```

```
site_name             : Richmond Mine
latitude              : 40.666668
longitude             : -122.51667
site_description      : Acid Mine Drainage
country               : UNITED STATES
region                : Iron Mountain, CA
library_acc           : JGI_AMD_5WAY_IRNMTN_LIB_20020301
sequencing_method     : dideoxysequencing (Sanger)
dna_type              : gDNA
num_of_reads          : 180713
habitat_name          : waste water
temperature           : 42
sample_acc            : JGI_AMD_5WAY_IRNMTN_SMPL_20020301
```

If you open that file in a text editor, you will see there are many more column headers than are shown here. The lines are extremely long, and it's a pretty sparse matrix of data. The `-d` flag to the program indicates to show a "dense" matrix, i.e., leave out the empty fields. I find the above format much easier to read. Here is the program to do that:

```
$ cat -n tabchk.py
     1    #!/usr/bin/env python3
     2
     3    import argparse
     4    import csv
     5    import os
     6    import sys
     7
     8    # --------------------------------------------------
     9    def get_args():
    10        parser = argparse.ArgumentParser(description='Check a delimited text file')
    11        parser.add_argument('file', metavar='str', help='File')
    12        parser.add_argument('-s', '--sep', help='Field separator',
    13                            metavar='str', type=str, default='\t')
    14        parser.add_argument('-l', '--limit', help='How many records to show',
    15                            metavar='int', type=int, default=1)
    16        parser.add_argument('-d', '--dense', help='Not sparse (skip empty fields)',
    17                            action='store_true')
    18        return parser.parse_args()
    19
    20    # --------------------------------------------------
    21    def main():
    22        args = get_args()
    23        file = args.file
    24        limit = args.limit
    25        sep = args.sep
    26        dense = args.dense
```

```
27
28          if not os.path.isfile(file):
29              print('"{}" is not a file'.format(file))
30              sys.exit(1)
31
32          with open(file) as csvfile:
33              reader = csv.DictReader(csvfile, delimiter=sep)
34
35              for i, row in enumerate(reader):
36                  vals = dict([x for x in row.items() if x[1] != '']) if dense else row
37                  flds = vals.keys()
38                  longest = max(map(len, flds))
39                  fmt = '{:' + str(longest + 1) + '}: {}'
40                  print('// ****** Record {} ****** //'.format(i+1))
41                  for key, val in vals.items():
42                      print(fmt.format(key, val))
43
44                  if i + 1 == limit:
45                      break
46
47      # --------------------------------------------------
48      if __name__ == '__main__':
49          main()
```

FASTA

Now let's finally get into parsing good, old FASTA files. The `argparse` and `csv` modules are standard in Python, but for FASTA we're going to need to install the BioPython ([http://biopython.org/](http://biopython.org/%29) module. This should work for you:

```
$ python3 -m pip install biopython
```

For this exercise, I'll use a few reads from the Global Ocean Sampling Expedition ([https://imicrobe.us/sample/view/578](https://imicrobe.us/sample/view/578%29):

```
$ head -5 CAM_SMPL_GS108.fa
>CAM_READ_0231669761 /library_id="CAM_LIB_GOS108XLRVAL-4F-1-400" /sample_id="CAM_SMPL_GS108'
ATTTACAATAATTTAATAAAATTAACTAGAAATAAAATATTGTATGAAAATATGTTAAATAATGAAAGTTTTT
CAGATCGTTTAATAATATTTTTCTTCCATTTTGCTTTTTTCTAAAATTGTTCAAAAACAAACTTCAAAGGAAA
ATCTTCAAAATTTACATGATTTTATATTTAAACAAATAGAGTTAAGTATAAGAGAAATTGGATATGGTGATGC
TTCAATAAATAAAAAAATGAAAGAGTATGTCAATGTGATGTACGCAATAATTGACAAAGTTGATTCATGGGAA
```

Let's write a script that mimics `seqmagick`:

```
$ seqmagick info CAM_SMPL_GS108.fa
```

```
name              alignment    min_len   max_len   avg_len  num_seqs
CAM_SMPL_GS108.fa FALSE            168       440    325.60         5
```

We'll skip the "alignment" and just do min/max/avg lengths and the number of sequences. You can pretty much copy and paste the example code from http://biopython.org/wiki/SeqIO. Here is the output from our script, "seqmagique.py":

```
$ ./seqmagique.py *.fa
name                 min_len    max_len    avg_len   num_seqs
CAM_SMPL_GS108.fa        168        440      325.6          5
CAM_SMPL_GS112.fa         71        517      398.4          5
```

The code to produce this builds on our earlier skills of lists and dictionaries as we will parse each file and save a dictionary of stats into a list, then we will iterate over that list at the end to show the output.

```
$ cat -n seqmagique.py
     1    #!/usr/bin/env python3
     2    """Mimic seqmagick, print stats on FASTA sequences"""
     3
     4    import os
     5    import sys
     6    from statistics import mean
     7    from Bio import SeqIO
     8
     9    files = sys.argv[1:]
    10
    11    if len(files) < 1:
    12        print('Usage: {} F1.fa [F2.fa...]'.format(os.path.basename(sys.argv[0])))
    13        sys.exit(1)
    14
    15    info = []
    16    for file in files:
    17        lengths = []
    18        for record in SeqIO.parse(file, "fasta"):
    19            lengths.append(len(record.seq))
    20
    21        info.append({'name': os.path.basename(file),
    22                     'min_len': min(lengths),
    23                     'max_len': max(lengths),
    24                     'avg_len': mean(lengths),
    25                     'num_seqs': len(lengths)})
    26
    27    if len(info):
    28        longest_file_name = max([len(f['name']) for f in info])
    29        fmt = '{:' + str(longest_file_name) + '} {:>10} {:>10} {:>10} {:>10}'
    30        flds = ['name', 'min_len', 'max_len', 'avg_len', 'num_seqs']
    31        print(fmt.format(*flds))
```

```
32        for rec in info:
33            print(fmt.format(*[rec[fld] for fld in flds]))
34    else:
35        print('I had trouble parsing your data')
```

FASTA subset

Sometimes you may only want to use part of a FASTA file, e.g., you want the first 1000 sequences to test some code, or you have samples that vary wildly in size and you want to sub-sample them down to an equal number of reads. Here is a Python program that will write the first N samples to a given output directory:

```
$ cat -n fa_subset.py
     1    #!/usr/bin/env python3
     2    """Subset FASTA files"""
     3
     4    import argparse
     5    import os
     6    import sys
     7    from Bio import SeqIO
     8
     9    # --------------------------------------------------
    10    def get_args():
    11        """get args"""
    12        parser = argparse.ArgumentParser(description='Split FASTA files')
    13        parser.add_argument('fasta', help='FASTA input file', metavar='FILE')
    14        parser.add_argument('-n', '--num', help='Number of records per file',
    15                            type=int, metavar='NUM', default=500000)
    16        parser.add_argument('-o', '--out_dir', help='Output directory',
    17                            type=str, metavar='DIR', default='subset')
    18        return parser.parse_args()
    19
    20    # --------------------------------------------------
    21    def main():
    22        """main"""
    23        args = get_args()
    24        fasta = args.fasta
    25        out_dir = args.out_dir
    26        num_seqs = args.num
    27
    28        if not os.path.isfile(fasta):
    29            print('--fasta "{}" is not valid'.format(fasta))
    30            sys.exit(1)
```

```
31
32        if os.path.dirname(fasta) == out_dir:
33            print('--outdir cannot be the same as input files')
34            sys.exit(1)
35
36        if num_seqs < 1:
37            print("--num cannot be less than one")
38            sys.exit(1)
39
40        if not os.path.isdir(out_dir):
41            os.mkdir(out_dir)
42
43        basename = os.path.basename(fasta)
44        out_file = os.path.join(out_dir, basename)
45        out_fh = open(out_file, 'wt')
46        num_written = 0
47
48        for record in SeqIO.parse(fasta, "fasta"):
49            SeqIO.write(record, out_fh, "fasta")
50            num_written += 1
51
52            if num_written == num_seqs:
53                break
54
55        print('Done, wrote {} sequence{} to "{}"'.format(
56            num_written, '' if num_written == 1 else 's', out_file))
57
58    # --------------------------------------------------
59    if __name__ == '__main__':
60        main()
```

FASTA splitter

I seem to have implemented my own FASTA splitter a few times in as many languages. Here is one that writes a maximum number of sequences to each output file. It would not be hard to instead write a maximum number of bytes, but, for the short reads I usually handle, this works fine. Again I will use the BioPython `SeqIO` module to parse the FASTA files

```
$ cat -n fasplit.py
     1    #!/usr/bin/env python3
     2    """split FASTA files"""
     3
     4    import argparse
```

```
 5  import os
 6  from Bio import SeqIO
 7
 8  # --------------------------------------------------
 9  def main():
10      """main"""
11      args = get_args()
12      fasta = args.fasta
13      out_dir = args.out_dir
14      max_per = args.num
15
16      if not os.path.isfile(fasta):
17          print('--fasta "{}" is not valid'.format(fasta))
18          exit(1)
19
20      if not os.path.isdir(out_dir):
21          os.mkdir(out_dir)
22
23      if max_per < 1:
24          print("--num cannot be less than one")
25          exit(1)
26
27      i = 0
28      nseq = 0
29      nfile = 0
30      out_fh = None
31      basename, ext = os.path.splitext(os.path.basename(fasta))
32
33      for record in SeqIO.parse(fasta, "fasta"):
34          if i == max_per:
35              i = 0
36              if out_fh is not None:
37                  out_fh.close()
38                  out_fh = None
39
40          i += 1
41          nseq += 1
42          if out_fh is None:
43              nfile += 1
44              path = os.path.join(out_dir, basename + '.' + str(nfile) + ext)
45              out_fh = open(path, 'wt')
46
47          SeqIO.write(record, out_fh, "fasta")
48
49      print('Done, wrote {} sequence{} to {} file{}'.format(
50          nseq, '' if nseq == 1 else 's',
```

```
51            nfile, '' if nfile == 1 else 's'))
52
53    # --------------------------------------------------
54    def get_args():
55        """get args"""
56        parser = argparse.ArgumentParser(description='Split FASTA files')
57        parser.add_argument('-f', '--fasta', help='FASTA input file',
58                            type=str, metavar='FILE', required=True)
59        parser.add_argument('-n', '--num', help='Number of records per file',
60                            type=int, metavar='NUM', default=50)
61        parser.add_argument('-o', '--out_dir', help='Output directory',
62                            type=str, metavar='DIR', default='fasplit')
63        return parser.parse_args()
64
65    # --------------------------------------------------
66    if __name__ == '__main__':
67        main()
```

If you type `make` in the "python/fasta-splitter" directory, you should see:

```
$ make
./fasplit.py -f POV_L.Sum.O.1000m_reads.fa -o pov -n 100
Done, wrote 2061 sequences to 21 files
```

We can verify that things worked:

```
$ for file in *; do echo -n $file && grep '^>' $file | wc -l; done
POV_L.Sum.O.1000m_reads.1.fa     100
POV_L.Sum.O.1000m_reads.10.fa     100
POV_L.Sum.O.1000m_reads.11.fa     100
POV_L.Sum.O.1000m_reads.12.fa     100
POV_L.Sum.O.1000m_reads.13.fa     100
POV_L.Sum.O.1000m_reads.14.fa     100
POV_L.Sum.O.1000m_reads.15.fa     100
POV_L.Sum.O.1000m_reads.16.fa     100
POV_L.Sum.O.1000m_reads.17.fa     100
POV_L.Sum.O.1000m_reads.18.fa     100
POV_L.Sum.O.1000m_reads.19.fa     100
POV_L.Sum.O.1000m_reads.2.fa     100
POV_L.Sum.O.1000m_reads.20.fa     100
POV_L.Sum.O.1000m_reads.21.fa      61
POV_L.Sum.O.1000m_reads.3.fa     100
POV_L.Sum.O.1000m_reads.4.fa     100
POV_L.Sum.O.1000m_reads.5.fa     100
POV_L.Sum.O.1000m_reads.6.fa     100
POV_L.Sum.O.1000m_reads.7.fa     100
POV_L.Sum.O.1000m_reads.8.fa     100
POV_L.Sum.O.1000m_reads.9.fa     100
```

GFF

Two of the most common output files in bioinformatics, GFF (General Feature Format) and BLAST's tab/CSV files do not include headers, so it's up to you to merge in the headers. Additionally, some of the lines may be comments (they start with "#" just like bash and Python), so you should skip those. Further, the last field in GFF is basically a dumping ground for whatever else the data provider felt like putting there. Usually it's a bunch of "key=value" pairs, but there's no guarantee. Let's take a look at parsing the GFF output from Prodigal:

```
$ cat -n parse_prodigal_gff.py
     1    #!/usr/bin/env python3
     2
     3    import argparse
     4    import csv
     5    import os
     6    import sys
     7
     8    # --------------------------------------------------
     9    def get_args():
    10        parser = argparse.ArgumentParser(description='Parse Prodigal GFF')
    11        parser.add_argument('gff', metavar='GFF', help='Prodigal GFF output')
    12        parser.add_argument('-m', '--min', help='Minimum score',
    13                            metavar='float', type=float, default=0.0)
    14        return parser.parse_args()
    15
    16    # --------------------------------------------------
    17    def main():
    18        args = get_args()
    19        gff_file = args.gff
    20        min_score = args.min
    21        flds = 'seqname source feature start end score strand frame attribute'.split()
    22
    23        for line in open(gff_file):
    24            if line[0] == '#':
    25                continue
    26
    27            row = dict(zip(flds, line.split('\t')))
    28            attrs = {}
    29            if 'attribute' in row:
    30                for fld in row['attribute'].split(';'):
    31                    if '=' in fld:
    32                        name, val = fld.split('=')
    33                        attrs[name] = val
    34
    35            if 'score' in attrs and float(attrs['score']) > min_score:
```

```
36                print(row)
37                break
38
39    # ------------------------------------------------
40    if __name__ == '__main__':
41        main()
```