Parsing with Python

We'll use the term "parsing" to mean deriving meaning from structured text. For example, we can use `argparse` to find meaning from command-line arguments that may or may not have flags or be defined by positions. In this chapter, we'll look at common file file formats in bioinformatics like CSV, FASTA/Q, and GFF.

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:

```
#!/usr/bin/env python3
"""
Author : kyclark
Date   : 2019-02-19
Purpose: Rock the Casbah
"""

import argparse
import sys


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Argparse Python script',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument(
        'positional', metavar='str', help='A positional argument')

    parser.add_argument(
        '-a',
        '--arg',
        help='A named string argument',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-i',
        '--int',
        help='A named integer argument',
        metavar='int',
        type=int,
        default=0)

    parser.add_argument(
        '-f', '--flag', help='A boolean flag', action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """Make a jazz noise here"""
    args = get_args()
    str_arg = args.arg
    int_arg = args.int
    flag_arg = args.flag
    pos_arg = args.positional

    print('str_arg = "{}"'.format(str_arg))
    print('int_arg = "{}"'.format(int_arg))
    print('flag_arg = "{}"'.format(flag_arg))
    print('positional = "{}"'.format(pos_arg))


```

```
# --------------------------------------------------
if __name__ == '__main__':
    main()
```

If you run without any arguments or with `-h|--help`, you get a usage statement:

```
$ ./test.py
usage: test.py [-h] [-a str] [-i int] [-f] str
test.py: error: the following arguments are required: str
[cholla@~/work/biosys-analytics/lectures/09-python-parsing]$ ./test.py -h
usage: test.py [-h] [-a str] [-i int] [-f] str

Argparse Python script

positional arguments:
  str                A positional argument

optional arguments:
  -h, --help         show this help message and exit
  -a str, --arg str  A named string argument (default: )
  -i int, --int int  A named integer argument (default: 0)
  -f, --flag         A boolean flag (default: False)
```

And the `argparse` module is able to turn the command line arguments into useful information:

```
$ ./test.py -a foo -i 42 -f ABCDE
str_arg = "foo"
int_arg = "42"
flag_arg = "True"
positional = "ABCDE"
```

If you try to write the code to parse `-a foo -i 42 -f ABCDE`, you will quickly appreciate how much effort using this module will save you!

CSV Files

"CSV" stands for "comma-separated values" and describes structured text that looks like:

```
foo,bar,baz
flip,burp,quux
```

More generally, these are values that are separated by some marker. Commas are typical but can cause problems when a comma can be a legitimate value, e.g., in addresses or formatted numbers, so tabs are often used as delimiters. Tab-delimited files may have the extension ".tsv," ".dat," ".tab", or ".txt." Usually CSV files have ".csv" and are especially common in the R/Pandas world.

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 to parse the output from Centrifuge (http://www.ccb.jhu.edu/software/centrifuge/). Despite the name, this module parses any line-oriented, delimited text, not just CSV files.

For this, we'll use some data from a study from Yellowstone National Park (https://www.imicrobe.us/#/samples/1378). For each input file, Centrifuge creates two tab-delimited 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
```

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

```
$ cat -n read_count_by_taxid.py
#!/usr/bin/env python3
"""Counts by taxID"""

import csv
import os
import sys
from collections import defaultdict
```

```

args = sys.argv[1:]

if len(args) != 1:
    print('Usage: {} SAMPLE.SUM'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

sum_file = args[0]

_, ext = os.path.splitext(sum_file)
if not ext == '.sum':
    print('File extention "{}" is not ".sum"'.format(ext))
    sys.exit(1)

counts = defaultdict(int)
with open(sum_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        taxID = row['taxID']
        counts[taxID] += 1

print('\t'.join(['count', 'taxID']))
for taxID, count in counts.items():
    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.centrifuge.sum
count   taxID
6432    321327
80  321332
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
#!/usr/bin/env python3
"""Counts by tax name"""

import csv
import os
import sys
from collections import defaultdict

args = sys.argv[1:]

if len(args) != 1:
    print('Usage: {} SAMPLE.SUM'.format(os.path.basename(sys.argv[0])))
    sys.exit(1)

sum_file = args[0]

basename, ext = os.path.splitext(sum_file)
if not ext == '.sum':
    print('File extention "{}" is not ".sum"'.format(ext))
    sys.exit(1)

tsv_file = basename + '.tsv'
if not os.path.isfile(tsv_file):
    print('Cannot find expected TSV "{}"'.format(tsv_file))
    sys.exit(1)

tax_name = {}
with open(tsv_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        tax_name[row['taxID']] = row['name']

counts = defaultdict(int)
with open(sum_file) as csvfile:
    reader = csv.DictReader(csvfile, delimiter='\t')
    for row in reader:
        taxID = row['taxID']
        counts[taxID] += 1

print('\t'.join(['count', 'taxID']))
for taxID, count in counts.items():
    name = tax_name.get(taxID) or 'NA'
    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 like this:

```
$ head oceanic_mesopelagic_zone_biome.csv
Analysis,Pipeline version,Sample,MGnify ID,Experiment type,Assembly,ENA run,ENA WGS sequence
MGYA00005220,2.0,ERS490373,MGYS00000410,metagenomic,,ERR599044,
MGYA00005081,2.0,ERS490507,MGYS00000410,metagenomic,,ERR599005,
MGYA00005208,2.0,ERS492680,MGYS00000410,metagenomic,,ERR598999,
MGYA00005133,2.0,ERS490633,MGYS00000410,metagenomic,,ERR599154,
MGYA00005272,2.0,ERS488769,MGYS00000410,metagenomic,,ERR599062,
MGYA00005209,2.0,ERS490714,MGYS00000410,metagenomic,,ERR599124,
MGYA00005243,2.0,ERS493822,MGYS00000410,metagenomic,,ERR599051,
MGYA00005117,2.0,ERS491980,MGYS00000410,metagenomic,,ERR599132,
MGYA00005135,2.0,ERS493705,MGYS00000410,metagenomic,,ERR599152,
```

I'd rather see it formatted vertically:

```
$ tabchk.py oceanic_mesopelagic_zone_biome.csv
// ****** Record 1 ****** //
Analysis             : MGYA00005220
Pipeline version     : 2.0
Sample               : ERS490373
MGnify ID            : MGYS00000410
Experiment type      : metagenomic
Assembly             :
ENA run              : ERR599044
ENA WGS sequence set :
```

Sometimes I have many more fields and lots of missing values, so I can use the `-d` flag to the program indicates to show a "dense" matrix, i.e., leave out the empty fields:

```
$ tabchk.py -d oceanic_mesopelagic_zone_biome.csv
// ****** Record 1 ****** //
Analysis         : MGYA00005220
Pipeline version : 2.0
Sample           : ERS490373
MGnify ID        : MGYS00000410
Experiment type  : metagenomic
ENA run          : ERR599044
```

Here is the `tabchk.py` program I wrote to do that. The program is generally useful, so I added it to the main `bin` directory of the repo so that you can use that if you have already added it to your $PATH.

```
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
Purpose: Check the first/few records of a delimited text file
"""

import argparse
import csv
import os
import re
import sys


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Check a delimited text file',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', metavar='FILE', help='Input file')

    parser.add_argument(
        '-s',
        '--sep',
        help='Field separator',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-f',
        '--field_names',
        help='Field names (no header)',
        metavar='str',
        type=str,
        default='')

    parser.add_argument(
        '-l',
        '--limit',
        help='How many records to show',
        metavar='int',
        type=int,
        default=1)

    parser.add_argument(
        '-d',
        '--dense',
        help='Not sparse (skip empty fields)',
        action='store_true')

    parser.add_argument(
        '-n',
        '--number',
        help='Show field number (e.g., for awk)',
        action='store_true')

    parser.add_argument(
        '-N',
        '--no_headers',
        help='No headers in first row',
        action='store_true')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    file = args.file
    limit = args.limit
    sep = args.sep
    dense = args.dense
    show_numbers = args.number
    no_headers = args.no_headers

    if not os.path.isfile(file):
        print('"{}" is not a file'.format(file))
        sys.exit(1)

    if not sep:
        _, ext = os.path.splitext(file)
        if ext == '.csv':
            sep = ','
        else:
            sep = '\t'

    with open(file) as csvfile:
        dict_args = {'delimiter': sep}

        if args.field_names:
            regex = re.compile(r'\s*,\s*')
            names = regex.split(args.field_names)
            if names:
                dict_args['fieldnames'] = names

        if args.no_headers:
            num_flds = len(csvfile.readline().split(sep))
            dict_args['fieldnames'] = list(
                map(lambda i: 'Field' + str(i), range(1, num_flds + 1)))
            csvfile.seek(0)

        reader = csv.DictReader(csvfile, **dict_args)

        for i, row in enumerate(reader, start=1):
            vals = dict(
                [x for x in row.items() if x[1] != '']) if dense else row
            flds = vals.keys()
            longest = max(map(len, flds))
            fmt = '{:' + str(longest + 1) + '}: {}'
            print('// ****** Record {} ****** //'.format(i))
            n = 0
            for key, val in vals.items():
                n += 1
                show = fmt.format(key, val)
                if show_numbers:
                    print('{:3} {}'.format(n, show))
                else:
                    print(show)

            if i + 1 == limit:
                break


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

## FASTA

Now let's finally get into parsing good, old FASTA files. We're going to need to install th

```
```

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

For this exercise, I'll use a few reads from the Global Ocean Sampling Expedition (https://i

$ iget /iplant/home/shared/imicrobe/projects/26/samples/578/CAM_SMPL_GS108.fa

Since that file is 725M, I've added a sample to the repo in the `examples` directory.

$ head -2 CAM_SMPL_GS108.fa >CAM_READ_0231669761 /library_id="CAM_LIB_GOS108XLRVAL-4F-1-400" /sample_id="CAM_SMPL_GS108" raw_id=SRA_ID=SRR066139.70645 raw_id=FG67BMZ02PUFIF ATTTACAATAATTTAATAAAATTAACTA-GAAATAAAATATTGTATGAAAATATGTTAAATAATGAAAGTTTTTCAGATCGTTTAATAATATTTTTCT

The format of a FASTA file is:

* A record starts with a header row which has `>` as the first character on a line
* The string following the `>` up until the first whitespace is the record ID
* Anything following the ID up to the newline can be the "description," but here we see this
* Any line after a header that does not start with `>` is the sequence. The sequence may be

We **could** write our own FASTA parser, and we would definitely learn much along the way, b

There is a useful program called `seqmagick` that will give you information like the followi

```
$ seqmagick info *.fa
name              alignment    min_len   max_len   avg_len  num_seqs
CAM_SMPL_GS108.fa FALSE             47       594    369.65       499
CAM_SMPL_GS112.fa FALSE             50       624    383.50       500
```

You can install it like so:

$ python -m pip install seqmagick

Let's write a toy program to mimic part of the output. We'll skip the "alignment" and just d

```
$ ./seqmagique.py *.fa
name              min_len    max_len    avg_len    num_seqs
CAM_SMPL_GS108.fa         47        594 369.45            500
CAM_SMPL_GS112.fa         50        624 383.50            500
```

The code to produce this builds on our earlier skills of lists and dictionaries as we will p

$ cat -n seqmagique.py 1 #!/usr/bin/env python3 2 “”" 3 Author: Ken Youens-Clark kyclark@email.arizona.edu 4 Purpose: Mimic seqmagick, print stats on FASTA sequences 5 “”" 6 7 import os 8 import sys 9 import numpy as np 10 from Bio import SeqIO 11 12 files = sys.argv[1:] 13 14 if not files: 15 print(‘Usage: {} F1.fa [F2.fa…]’.format(os.path.basename(sys.argv[0]))) 16 sys.exit(1) 17 18 info = [] 19 for file in files: 20 lengths = [] 21 for record in SeqIO.parse(file, ‘fasta’): 22 lengths.append(len(record.seq)) 23 24 info.append({ 25 ‘name’: os.path.basename(file), 26 ‘min_len’: min(lengths), 27 ‘max_len’: max(lengths), 28 ‘avg_len’: ‘{:.2f}’.format(np.mean(lengths)), 29 ‘num_seqs’: len(lengths) 30 }) 31 32 if info: 33 longest_file_name = max([len(f[‘name’]) for f in info]) 34 fmt = ‘{:’ + str(longest_file_name) + ‘} {:10} {:10} {:10} {:10}’ 35 flds = [‘name’, ‘min_len’, ‘max_len’, ‘avg_len’, ‘num_seqs’] 36 print(fmt.format(*flds)) 37 for rec in info: 38 print(fmt.format(*[rec[fld] for fld in flds])) 39 else: 40 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 seque
```

$ cat -n subset_fastx.py 1 #!/usr/bin/env python3 2 “”" 3 Author: Ken Youens-Clark kyclark@email.arizona.edu 4 Purpose: Subset FASTA/Q files 5 “”" 6 7 import argparse 8 import os 9 import sys 10 from Bio import SeqIO 11 12 13 # ———————————————— 14 def get_args(): 15 “”“get args”“” 16 parser = argparse.ArgumentParser(17 description=‘Split FASTA files’, 18 formatter_class=argparse.ArgumentDefaultsHelpFormatter) 19 20 parser.add_argument(‘file’, help=‘Input file’, metavar=‘FILE’) 21 22 parser.add_argument(23 ‘-f’, 24 ‘–infmt’, 25 help=‘Input file format’, 26 type=str, 27 metavar=‘FMT’, 28 choices=[‘fasta’, ‘fastq’], 29 default=‘fasta’) 30 31 parser.add_argument(32 ‘-F’, 33 ‘–outfmt’, 34 help=‘Output file format’, 35 type=str, 36 metavar=‘FMT’, 37 default=None) 38 39 parser.add_argument(40 ‘-n’, 41 ‘–num’, 42 help=‘Number of records per file’, 43 type=int, 44 metavar=‘NUM’, 45 default=500000) 46 47 parser.add_argument(48 ‘-o’, 49 ‘–outdir’, 50 help=‘Output directory’, 51 type=str, 52 metavar=‘DIR’, 53 default=‘subset’) 54 55 return parser.parse_args() 56 57 # ———————————————— 58 def warn(msg): 59 “”“Print a message to STDERR”“” 60 print(msg, file=sys.stderr) 61 62 63 # ———————————————— 64 def die(msg=‘Something bad happened’): 65 “”“warn() and exit with error”“” 66 warn(msg) 67 sys.exit(1) 68 69 70 # ———————————————— 71 def main(): 72 “”“main”“” 73 args = get_args() 74 in_file = args.file 75 in_fmt = args.infmt 76 out_fmt = args.outfmt if args.outfmt else args.infmt 77 out_dir = args.outdir 78 num_seqs = args.num 79 80 if not os.path.isfile(in_file): 81 die(‘–file “{}” is not a file’.format(in_file)) 82 83 if os.path.dirname(os.path.abspath(in_file)) == os.path.abspath(out_dir): 84 die(‘–outdir “{}” cannot be the same as input files’.format(out_dir)) 85 86 if num_seqs < 1: 87 die(“–num cannot be less than one”) 88 89 if not os.path.isdir(out_dir): 90 os.mkdir(out_dir) 91 92 basename = os.path.basename(in_file) 93 out_file = os.path.join(out_dir, basename) 94 out_fh = open(out_file, ‘wt’) 95 num_written = 0 96 97 for record in SeqIO.parse(in_file, in_fmt): 98 SeqIO.write(record, out_fh, out_fmt) 99 num_written += 1 100 101 if num_written == num_seqs: 102 break 103 104 print(‘Done, wrote {} sequence{} to “{}”’.format(105 num_written, ‘’ if num_written == 1 else ’s’, out_file)) 106 107 108 #

109 if **name** == ‘**main**’: 110 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 fa_split.py
#!/usr/bin/env python3
"""
Author:  Ken Youens-Clark
Purpose: Split FASTA files
NB:      If you have FASTQ files, maybe just use "split"?
"""

import argparse
import os
import sys
from Bio import SeqIO


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Split FASTA/Q files',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('file', help='FASTA input file(s)', nargs='+')

    parser.add_argument(
        '-f',
        '--input_format',
        help='Input file format',
        type=str,
        metavar='FORMAT',
        choices=['fasta', 'fastq'],
        default='fasta')

    parser.add_argument(
        '-F',
        '--output_format',
        help='Output file format',
        type=str,
        metavar='FORMAT',
        choices=['fasta', 'fastq'],
        default='fasta')

    parser.add_argument(
        '-n',
        '--sequences_per_file',
        help='Number of sequences per file',
        type=int,
        metavar='NUM',
        default=50)

    parser.add_argument(
        '-o',
        '--out_dir',
        help='Output directory',
        type=str,
        metavar='DIR',
        default='fasplit')

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    files = args.file
    input_format = args.input_format
    output_format = args.output_format
    out_dir = args.out_dir
    seqs_per_file = args.sequences_per_file

    if not os.path.isdir(out_dir):
        os.mkdir(out_dir)

    if seqs_per_file < 1:
        die('--sequences_per_file "{}" cannot be less than one'.format(
            seqs_per_file))

    num_files = 0
    num_seqs_written = 0
    for i, file in enumerate(files, start=1):
        print('{:3d}: {}'.format(i, os.path.basename(file)))
        num_files += 1
        num_seqs_written += process(
            file=file,
            input_format=input_format,
            output_format=output_format,
            out_dir=out_dir,
            seqs_per_file=seqs_per_file)

    print('Done, processed {} sequence{} from {} file{} into "{}"'.format(
        num_seqs_written, '' if num_seqs_written == 1 else 's', num_files, ''
        if num_files == 1 else 's', out_dir))


# --------------------------------------------------
def process(file, input_format, output_format, out_dir, seqs_per_file):
    """
    Spilt file into smaller files into out_dir
    Optionally convert to output format
    Return number of sequences written
    """
    if not os.path.isfile(file):
        warn('"{}" is not valid'.format(file))
        return 0

    basename, ext = os.path.splitext(os.path.basename(file))
    out_fh = None
    i = 0
    num_written = 0
    nfile = 0
    for record in SeqIO.parse(file, input_format):
        if i == seqs_per_file:
            i = 0
            if out_fh is not None:
                out_fh.close()
                out_fh = None

        i += 1
        num_written += 1
        if out_fh is None:
            nfile += 1
            path = os.path.join(out_dir,
                                basename + '.' + '{:04d}'.format(nfile) + ext)
            out_fh = open(path, 'wt')

        SeqIO.write(record, out_fh, output_format)

    return num_written


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

You can run this on the FASTA files in the `examples` directory to split them into files of 50 sequences each:

```
$ ./fa_split.py *.fa
  1: CAM_SMPL_GS108.fa
  2: CAM_SMPL_GS112.fa
Done, processed 1000 sequences from 2 files into "fasplit"
$ ls -lh fasplit/
total 1088
-rw-r--r--  1 kyclark  staff    22K Feb 19 15:41 CAM_SMPL_GS108.0001.fa
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS108.0002.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS108.0003.fa
-rw-r--r--  1 kyclark  staff    23K Feb 19 15:41 CAM_SMPL_GS108.0004.fa
-rw-r--r--  1 kyclark  staff    22K Feb 19 15:41 CAM_SMPL_GS108.0005.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS108.0006.fa
-rw-r--r--  1 kyclark  staff    29K Feb 19 15:41 CAM_SMPL_GS108.0007.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS108.0008.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS108.0009.fa
-rw-r--r--  1 kyclark  staff    24K Feb 19 15:41 CAM_SMPL_GS108.0010.fa
-rw-r--r--  1 kyclark  staff    26K Feb 19 15:41 CAM_SMPL_GS112.0001.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0002.fa
```

```
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS112.0003.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0004.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0005.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0006.fa
-rw-r--r--  1 kyclark  staff    28K Feb 19 15:41 CAM_SMPL_GS112.0007.fa
-rw-r--r--  1 kyclark  staff    29K Feb 19 15:41 CAM_SMPL_GS112.0008.fa
-rw-r--r--  1 kyclark  staff    27K Feb 19 15:41 CAM_SMPL_GS112.0009.fa
-rw-r--r--  1 kyclark  staff    16K Feb 19 15:41 CAM_SMPL_GS112.0010.fa
```

We can verify that things worked:

```
$ for file in fasplit/*; do echo -n $file && grep '^>' $file | wc -l; done
fasplit/CAM_SMPL_GS108.0001.fa      50
fasplit/CAM_SMPL_GS108.0002.fa      50
fasplit/CAM_SMPL_GS108.0003.fa      50
fasplit/CAM_SMPL_GS108.0004.fa      50
fasplit/CAM_SMPL_GS108.0005.fa      50
fasplit/CAM_SMPL_GS108.0006.fa      50
fasplit/CAM_SMPL_GS108.0007.fa      50
fasplit/CAM_SMPL_GS108.0008.fa      50
fasplit/CAM_SMPL_GS108.0009.fa      50
fasplit/CAM_SMPL_GS108.0010.fa      50
fasplit/CAM_SMPL_GS112.0001.fa      50
fasplit/CAM_SMPL_GS112.0002.fa      50
fasplit/CAM_SMPL_GS112.0003.fa      50
fasplit/CAM_SMPL_GS112.0004.fa      50
fasplit/CAM_SMPL_GS112.0005.fa      50
fasplit/CAM_SMPL_GS112.0006.fa      50
fasplit/CAM_SMPL_GS112.0007.fa      50
fasplit/CAM_SMPL_GS112.0008.fa      50
fasplit/CAM_SMPL_GS112.0009.fa      50
fasplit/CAM_SMPL_GS112.0010.fa      50
```

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  """
```

```
Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
Purpose: Parse the GFF output of Prodigal
"""

import argparse
import os
import sys


# --------------------------------------------------
def get_args():
    """get args"""
    parser = argparse.ArgumentParser(
        description='Prodigal GFF parser',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('gff', metavar='FILE', help='Prodigal GFF file')

    parser.add_argument(
        '-m',
        '--min',
        help='Min score',
        metavar='float',
        type=float,
        default=0)

    return parser.parse_args()


# --------------------------------------------------
def warn(msg):
    """Print a message to STDERR"""
    print(msg, file=sys.stderr)


# --------------------------------------------------
def die(msg='Something bad happened'):
    """warn() and exit with error"""
    warn(msg)
    sys.exit(1)


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    gff_file = args.gff
    min_score = args.min

    if not os.path.isfile(gff_file):
        die('GFF "{}" is not a file'.format(gff_file))

    flds = [
        'seqname', 'source', 'feature', 'start', 'end', 'score', 'strand',
        'frame', 'attribute'
    ]

    for line in open(gff_file):
        if line.startswith('#'):
            continue

        vals = line.rstrip().split('\t')
        rec = dict(zip(flds, vals))
        attrs = {}

        for x in rec['attribute'].split(';'):
            if '=' in x:
                key, value = x.split('=')
                attrs[key] = value

        score = attrs.get('score')
        if score is not None and float(score) >= min_score:
            print('{} {}'.format(rec['seqname'], score))


# --------------------------------------------------
if __name__ == '__main__':
    main()
```