

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 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:

```
1  #!/usr/bin/env python3
2  """
3  Author : kyclark
4  Date   : 2019-02-19
5  Purpose: Rock the Casbah
6  """
7
8  import argparse
9  import sys
10
11
12  # -----
13  def get_args():
14      """get command-line arguments"""
15      parser = argparse.ArgumentParser(
16          description='Argparse Python script',
17          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
18
19      parser.add_argument(
20          'positional', metavar='str', help='A positional argument')
21
22      parser.add_argument(
23          '-a',
24          '--arg',
25          help='A named string argument',
```

```

26         metavar='str',
27         type=str,
28         default='')
29
30     parser.add_argument(
31         '-i',
32         '--int',
33         help='A named integer argument',
34         metavar='int',
35         type=int,
36         default=0)
37
38     parser.add_argument(
39         '-f', '--flag', help='A boolean flag', action='store_true')
40
41     return parser.parse_args()
42
43
44 # -----
45 def warn(msg):
46     """Print a message to STDERR"""
47     print(msg, file=sys.stderr)
48
49
50 # -----
51 def die(msg='Something bad happened'):
52     """warn() and exit with error"""
53     warn(msg)
54     sys.exit(1)
55
56
57 # -----
58 def main():
59     """Make a jazz noise here"""
60     args = get_args()
61     str_arg = args.arg
62     int_arg = args.int
63     flag_arg = args.flag
64     pos_arg = args.positional
65
66     print('str_arg = {}'.format(str_arg))
67     print('int_arg = {}'.format(int_arg))
68     print('flag_arg = {}'.format(flag_arg))
69     print('positional = {}'.format(pos_arg))
70
71

```

```

72 # -----
73 if __name__ == '__main__':
74     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
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 extension "{}" 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 extension ".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
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 extension "{}" 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 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.

```
1  #!/usr/bin/env python3
2  """
3  Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
4  Purpose: Check the first/few records of a delimited text file
5  """
6
7  import argparse
8  import csv
9  import os
10 import re
11 import sys
12
13
14 # -----
15 def get_args():
16     """Get command-line arguments"""
17     parser = argparse.ArgumentParser(
18         description='Check a delimited text file',
19         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
20
21     parser.add_argument('file', metavar='FILE', help='Input file')
22
23     parser.add_argument(
24         '-s',
25         '--sep',
26         help='Field separator',
27         metavar='str',
28         type=str,
29         default='')
30
31     parser.add_argument(
32         '-f',
33         '--field_names',
34         help='Field names (no header)',
35         metavar='str',
36         type=str,
37         default='')
38
39     parser.add_argument(
40         '-l',
41         '--limit',
42         help='How many records to show',
43         metavar='int',
44         type=int,
```



```

45         default=1)
46
47     parser.add_argument(
48         '-d',
49         '--dense',
50         help='Not sparse (skip empty fields)',
51         action='store_true')
52
53     parser.add_argument(
54         '-n',
55         '--number',
56         help='Show field number (e.g., for awk)',
57         action='store_true')
58
59     parser.add_argument(
60         '-N',
61         '--no_headers',
62         help='No headers in first row',
63         action='store_true')
64
65     return parser.parse_args()
66
67
68 # -----
69 def main():
70     """main"""
71     args = get_args()
72     file = args.file
73     limit = args.limit
74     sep = args.sep
75     dense = args.dense
76     show_numbers = args.number
77     no_headers = args.no_headers
78
79     if not os.path.isfile(file):
80         print("{} is not a file".format(file))
81         sys.exit(1)
82
83     if not sep:
84         _, ext = os.path.splitext(file)
85         if ext == '.csv':
86             sep = ','
87         else:
88             sep = '\t'
89
90     with open(file) as csvfile:

```

```

91         dict_args = {'delimiter': sep}
92
93     if args.field_names:
94         regex = re.compile(r'\s*,\s*')
95         names = regex.split(args.field_names)
96         if names:
97             dict_args['fieldnames'] = names
98
99     if args.no_headers:
100         num_flds = len(csvfile.readline().split(sep))
101         dict_args['fieldnames'] = list(
102             map(lambda i: 'Field' + str(i), range(1, num_flds + 1)))
103         csvfile.seek(0)
104
105     reader = csv.DictReader(csvfile, **dict_args)
106
107     for i, row in enumerate(reader, start=1):
108         vals = dict(
109             [x for x in row.items() if x[1] != '']) if dense else row
110         flds = vals.keys()
111         longest = max(map(len, flds))
112         fmt = '{:' + str(longest + 1) + '}: {}'
113         print('// ***** Record {} ***** //'.format(i))
114         n = 0
115         for key, val in vals.items():
116             n += 1
117             show = fmt.format(key, val)
118             if show_numbers:
119                 print('{:3} {}'.format(n, show))
120             else:
121                 print(show)
122
123         if i + 1 == limit:
124             break
125
126
127     # -----
128     if __name__ == '__main__':
129         main()
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

FASTA

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

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

```
...
```

For this exercise, I'll use a few reads from the Global Ocean Sampling Expedition (<https://www.gos.noaa.gov/>)

```
$ 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=SRR066139.70645
raw_id=FG67BMZ02PUFIF  ATTTACAATAATTTAATAAAATTAACATA-
GAAATAAAATATTGTATGAAAATATGTTAAATAATGAAAGTTTTTCAGATCGTTTAATAATATTTTCT
```

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, but

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

```
...
```

```
$ 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 output

```
...
```

```
$ ./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 see in the next chapter.

```
$ 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 sequences

```
$ 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 77 if args.outfmt else args.infmt 78 out_dir = args.outdir 79 num_seqs = args.num 80 81 if not os.path.isfile(in_file): 82 die('-file "{}" is not a file'.format(in_file)) 83 84 if os.path.dirname(os.path.abspath(in_file)) == os.path.abspath(out_dir): 85 die('-outdir "{}" cannot be the same as input files'.format(out_dir)) 86 87 if num_seqs < 1: 88 die("-num cannot be less than one") 89 90 if not os.path.isdir(out_dir): 91 os.mkdir(out_dir) 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
 1  #!/usr/bin/env python3
 2  """
 3  Author:  Ken Youens-Clark
 4  Purpose: Split FASTA files
 5  NB:      If you have FASTQ files, maybe just use "split"?
 6  """
 7
 8  import argparse
 9  import os
10  import sys
11  from Bio import SeqIO
12
13
14  # -----
15  def get_args():
16      """get args"""
17      parser = argparse.ArgumentParser(
18          description='Split FASTA/Q files',
19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
20
21      parser.add_argument('file', help='FASTA input file(s)', nargs='+')
22
23      parser.add_argument(
24          '-f',
25          '--input_format',
26          help='Input file format',
27          type=str,
28          metavar='FORMAT',

```

```

29         choices=['fasta', 'fastq'],
30         default='fasta')
31
32     parser.add_argument(
33         '-F',
34         '--output_format',
35         help='Output file format',
36         type=str,
37         metavar='FORMAT',
38         choices=['fasta', 'fastq'],
39         default='fasta')
40
41     parser.add_argument(
42         '-n',
43         '--sequences_per_file',
44         help='Number of sequences per file',
45         type=int,
46         metavar='NUM',
47         default=50)
48
49     parser.add_argument(
50         '-o',
51         '--out_dir',
52         help='Output directory',
53         type=str,
54         metavar='DIR',
55         default='fasplit')
56
57     return parser.parse_args()
58
59
60 # -----
61 def warn(msg):
62     """Print a message to STDERR"""
63     print(msg, file=sys.stderr)
64
65
66 # -----
67 def die(msg='Something bad happened'):
68     """warn() and exit with error"""
69     warn(msg)
70     sys.exit(1)
71
72
73 # -----
74 def main():

```

```

75     """main"""
76     args = get_args()
77     files = args.file
78     input_format = args.input_format
79     output_format = args.output_format
80     out_dir = args.out_dir
81     seqs_per_file = args.sequences_per_file
82
83     if not os.path.isdir(out_dir):
84         os.mkdir(out_dir)
85
86     if seqs_per_file < 1:
87         die('--sequences_per_file "{}" cannot be less than one'.format(
88             seqs_per_file))
89
90     num_files = 0
91     num_seqs_written = 0
92     for i, file in enumerate(files, start=1):
93         print('{:3d}: {}'.format(i, os.path.basename(file)))
94         num_files += 1
95         num_seqs_written += process(
96             file=file,
97             input_format=input_format,
98             output_format=output_format,
99             out_dir=out_dir,
100             seqs_per_file=seqs_per_file)
101
102     print('Done, processed {} sequence{} from {} file{} into "{}".format(
103         num_seqs_written, ' ' if num_seqs_written == 1 else 's', num_files, ' '
104         if num_files == 1 else 's', out_dir))
105
106
107 # -----
108 def process(file, input_format, output_format, out_dir, seqs_per_file):
109     """
110     Spilt file into smaller files into out_dir
111     Optionally convert to output format
112     Return number of sequences written
113     """
114     if not os.path.isfile(file):
115         warn("{} is not valid".format(file))
116         return 0
117
118     basename, ext = os.path.splitext(os.path.basename(file))
119     out_fh = None
120     i = 0

```

```

121     num_written = 0
122     nfile = 0
123     for record in SeqIO.parse(file, input_format):
124         if i == seqs_per_file:
125             i = 0
126             if out_fh is not None:
127                 out_fh.close()
128                 out_fh = None
129
130             i += 1
131             num_written += 1
132             if out_fh is None:
133                 nfile += 1
134                 path = os.path.join(out_dir,
135                                     basename + '.' + '{:04d}'.format(nfile) + ext)
136                 out_fh = open(path, 'wt')
137
138             SeqIO.write(record, out_fh, output_format)
139
140     return num_written
141
142
143 # -----
144 if __name__ == '__main__':
145     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  """

```

```

3 Author: Ken Youens-Clark <kyclark@email.arizona.edu>
4 Purpose: Parse the GFF output of Prodigal
5 """
6
7 import argparse
8 import os
9 import sys
10
11
12 # -----
13 def get_args():
14     """get args"""
15     parser = argparse.ArgumentParser(
16         description='Prodigal GFF parser',
17         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
18
19     parser.add_argument('gff', metavar='FILE', help='Prodigal GFF file')
20
21     parser.add_argument(
22         '-m',
23         '--min',
24         help='Min score',
25         metavar='float',
26         type=float,
27         default=0)
28
29     return parser.parse_args()
30
31
32 # -----
33 def warn(msg):
34     """Print a message to STDERR"""
35     print(msg, file=sys.stderr)
36
37
38 # -----
39 def die(msg='Something bad happened'):
40     """warn() and exit with error"""
41     warn(msg)
42     sys.exit(1)
43
44
45 # -----
46 def main():
47     """main"""
48     args = get_args()

```

```

49     gff_file = args.gff
50     min_score = args.min
51
52     if not os.path.isfile(gff_file):
53         die('GFF "{}" is not a file'.format(gff_file))
54
55     flds = [
56         'seqname', 'source', 'feature', 'start', 'end', 'score', 'strand',
57         'frame', 'attribute'
58     ]
59
60     for line in open(gff_file):
61         if line.startswith('#'):
62             continue
63
64         vals = line.rstrip().split('\t')
65         rec = dict(zip(flds, vals))
66         attrs = {}
67
68         for x in rec['attribute'].split(';'):
69             if '=' in x:
70                 key, value = x.split('=')
71                 attrs[key] = value
72
73         score = attrs.get('score')
74         if score is not None and float(score) >= min_score:
75             print('{} {}'.format(rec['seqname'], score))
76
77
78     # -----
79     if __name__ == '__main__':
80         main()

```