Parsing with Python

> Programming is the art of doing one thing at a time. – Michael Feathers

We'll use the term "parsing" to mean deriving meaning from structured text. In this chapter, we'll look at parsing command-line arguments and common file file formats in bioinformatics like CSV, FASTA/Q, and GFF.

Command-line Arguments

If you've been using `new_py.py -a` to create new programs, you've already been using a parser – one that uses the `argparse` module to derive meaning from command-line arguments that may or may not have flags or be defined by positions. Let's create a new program and see how it works:

```
$ 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 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.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
```

```
    """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.py

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',
```

```
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 == limit:
124                  break
125
126
127  # -----------------------------------------------
128  if __name__ == '__main__':
129      main()
```

BLAST's tab-delimited output (`-outfmt 6`) does not include headers, so I have this alias:

```
alias blast6chk='tabchk.py -f "qseqid,sseqid,pident,length,mismatch,gapopen,qstart,qend,ssta
```

tabget.py

Here's a program that extracts columns from a delimited text file using the column names instead of the number (yes, I know we could just use `awk`):

```
$ cat -n tabget.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : Ken Youens-Clark <kyclark@gmail.com>
     4  Date   : 2018-11-16
     5  Purpose: Get fields from a tab/csv file
     6  """
     7
     8  import argparse
     9  import csv
    10  import os
    11  import re
    12  import sys
    13
    14
    15  # --------------------------------------------------
    16  def get_args():
    17      """get command-line arguments"""
    18      parser = argparse.ArgumentParser(
    19          description='Argparse Python script',
    20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    21
    22      parser.add_argument(
    23          'file', nargs='+', metavar='FILE', help='Input file(s)')
    24
    25      parser.add_argument(
    26          '-d',
    27          '--delimiter',
    28          help='Field delimiter',
    29          metavar='str',
    30          type=str,
    31          default='')
    32
    33      parser.add_argument(
    34          '-f',
    35          '--field',
    36          help='Name of field(s)',
    37          metavar='str',
    38          type=str,
    39          default='')
    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      files = args.file
62      default_delim = args.delimiter
63      field_names = re.split('\s*,\s*', args.field)
64
65      for file in files:
66          with open(file, 'rt') as fh:
67              delim = default_delim
68              if not delim:
69                  _, ext = os.path.splitext(file)
70                  if ext == '.csv':
71                      delim = ','
72                  else:
73                      delim = '\t'
74
75              reader = csv.DictReader(fh, delimiter=delim)
76
77              print(delim.join(field_names))
78
79              for row in reader:
80                  flds = list(map(lambda f: row[f], field_names))
81                  print(delim.join(flds))
82
83
84  # --------------------------------------------------
85  if __name__ == '__main__':
86      main()
```

tab2json.py

At some point I must have needed to turn a flat, delimited text file into a hierarchical, JSON structured, but I cannot at this moment remember why. Anyway, here's a program that will do that.

```
$ cat -n tab2json.py
     1  #!/usr/bin/env python3
     2  """
     3  Author:  Ken Youens-Clark <kyclark@gmail.com>
     4  Purpose: Convert a delimited text file to JSON
     5  """
     6
     7  import argparse
     8  import csv
     9  import json
    10  import os
    11  import re
    12  import sys
    13
    14
    15  # --------------------------------------------------
    16  def get_args():
    17      """get args"""
    18      parser = argparse.ArgumentParser(
    19          description='Argparse Python script',
    20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    21
    22      parser.add_argument(
    23          'tabfile', metavar='str', nargs='+', help='A positional argument')
    24
    25      parser.add_argument(
    26          '-s',
    27          '--sep',
    28          help='Field separator',
    29          metavar='str',
    30          type=str,
    31          default='\t')
    32
    33      parser.add_argument(
    34          '-o',
    35          '--outdir',
    36          help='Output dir',
    37          metavar='str',
    38          type=str,
    39          default='')
```

```
40
41      parser.add_argument(
42          '-i',
43          '--indent',
44          help='Indent level',
45          metavar='int',
46          type=int,
47          default=2)
48
49      parser.add_argument(
50          '-n',
51          '--normalize_headers',
52          help='Normalize headers',
53          action='store_true')
54
55      return parser.parse_args()
56
57
58  # --------------------------------------------------
59  def main():
60      """main"""
61      args = get_args()
62      indent_level = args.indent
63      out_dir = args.outdir
64      fs = args.sep
65      norm_hdr = args.normalize_headers
66      tabfiles = args.tabfile
67
68      if len(tabfiles) < 1:
69          print('No input files')
70          sys.exit(1)
71
72      if indent_level < 0:
73          indent_level = 0
74
75      if out_dir and not os.path.isdir(out_dir):
76          os.makedirs(out_dir)
77
78      for i, tabfile in enumerate(tabfiles, start=1):
79          basename = os.path.basename(tabfile)
80          filename, _ = os.path.splitext(basename)
81          dirname = os.path.dirname(os.path.abspath(tabfile))
82          print('{:3}: {}'.format(i, basename))
83          write_dir = out_dir if out_dir else dirname
84          out_path = os.path.join(write_dir, filename + '.json')
85          out_fh = open(out_path, 'wt')
```

```
 86
 87          with open(tabfile) as fh:
 88              reader = csv.DictReader(fh, delimiter=fs)
 89              if norm_hdr:
 90                  reader.fieldnames = list(map(normalize, reader.fieldnames))
 91              out_fh.write(json.dumps(list(reader), indent=indent_level))
 92
 93
 94  # --------------------------------------------------
 95  def normalize(hdr):
 96      return re.sub(r'[^A-Za-z0-9_]', '', hdr.lower().replace(' ', '_'))
 97
 98
 99  # --------------------------------------------------
100  if __name__ == '__main__':
101      main()
```

FASTA

Now let's finally get into parsing good, old FASTA files. We're going to need to install the BioPython (http://biopython.org/) module to get a FASTA parser. 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/#/samples/578). You can download the full file with this command:

```
$ 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 -5 CAM_SMPL_GS108.fa
>CAM_READ_0231669761 /library_id="CAM_LIB_GOS108XLRVAL-4F-1-400" /sample_id="CAM_SMPL_GS108'
ATTTACAATAATTTAATAAAATTAACTAGAAATAAAATATTGTATGAAAATATGTTAAAT
AATGAAAGTTTTTCAGATCGTTTAATAATATTTTTCTTCCATTTTGCTTTTTTCTAAAAT
TGTTCAAAAACAAACTTCAAAGGAAAATCTTCAAAATTTACATGATTTTATATTTAAACA
AATAGAGTTAAGTATAAGAGAAATTGGATATGGTGATGCTTCAATAAATAAAAAAATGAA
```

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 space has been set up as key/value pairs of metadata

- Any line after a header that does not start with `>` is the sequence. The sequence may be one long line or many shorter lines.

We **could** write our own FASTA parser, and we would definitely learn much along the way, but let's not and instead use the BioPython `SeqIO` (sequence input-output) module to read and write all the different formats. FASTA is one of the most common, but other formats may include FASTQ (FASTA but with "Quality" scores for the base calls), GenBank, EMBL, and more. See https://biopython.org/wiki/SeqIO for an exhaustive list.

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 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         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 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  """
     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 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 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
```

```
import sys
from Bio import SeqIO


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

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

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

    parser.add_argument(
        '-F',
        '--outfmt',
        help='Output file format',
        type=str,
        metavar='FMT',
        default=None)

    parser.add_argument(
        '-n',
        '--num',
        help='Number of records per file',
        type=int,
        metavar='NUM',
        default=500000)

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

```

```
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()
```

Here is a version that will randomly select some percentage of the reads from the input file. I had to write this version because we had created an artificial metagenome from a set of known organisms, and I was testing a program with input of various numbers of reads. I did not realize at first that, in creating the artificial set, reads from each organism had been added in blocks. Since I was taking all my reads from the top of the file down, I was mostly getting just the first few species. Randomly selecting reads when there are potentially millions of records is a bit tricky, so I decided to use a non-deterministic approach where I just roll the dice and see if the number I get on each read is less than the percentage of reads I want to take. This program will also stop at a given number of reads so you could use it to randomly subset an unevenly sized number of samples down to the same number of reads per sample.

```
$ cat -n random_subset.py
     1  #!/usr/bin/env python3
     2  """
     3  Author:  Ken Youens-Clark <kyclark@email.arizona.edu>
     4  Purpose: Probabalistically subset FASTQ/A
     5  """
     6
     7  import argparse
     8  import os
     9  import re
    10  import sys
    11  from random import randint
    12  from Bio import SeqIO
    13
    14
    15  # --------------------------------------------------
    16  def get_args():
    17      """get args"""
    18      parser = argparse.ArgumentParser(
    19          description='Randomly subset FASTQ',
    20          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    21
    22      parser.add_argument('file', metavar='FILE', help='FASTQ/A file')
```

```
23
24      parser.add_argument(
25          '-p',
26          '--pct',
27          help='Percent of reads',
28          metavar='int',
29          type=int,
30          default=50)
31
32      parser.add_argument(
33          '-m',
34          '--max',
35          help='Maximum number of reads',
36          metavar='int',
37          type=int,
38          default=0)
39
40      parser.add_argument(
41          '-f',
42          '--input_format',
43          help='Intput format',
44          metavar='IN_FMT',
45          type=str,
46          choices=['fastq', 'fasta'],
47          default='')
48
49      parser.add_argument(
50          '-F',
51          '--output_format',
52          help='Output format',
53          metavar='OUT_FMT',
54          type=str,
55          choices=['fastq', 'fasta'],
56          default='')
57
58      parser.add_argument(
59          '-o',
60          '--outfile',
61          help='Output file',
62          metavar='FILE',
63          type=str,
64          default='')
65
66      return parser.parse_args()
67
68
```

```
69  # --------------------------------------------------
70  def warn(msg):
71      """Print a message to STDERR"""
72      print(msg, file=sys.stderr)
73
74
75  # --------------------------------------------------
76  def die(msg='Something bad happened'):
77      """warn() and exit with error"""
78      warn(msg)
79      sys.exit(1)
80
81
82  # --------------------------------------------------
83  def main():
84      """main"""
85      args = get_args()
86      file = args.file
87      pct = args.pct
88      out_file = args.outfile
89      max_num_reads = args.max
90      min_num = 0
91      max_num = 100
92
93      if not os.path.isfile(file):
94          die('"{}" is not a file'.format(file))
95
96      in_fmt = args.input_format
97      if not in_fmt:
98          _, ext = os.path.splitext(file)
99          in_fmt = 'fastq' if re.match('\.f(ast)?q$', ext) else 'fasta'
100
101     out_fmt = args.output_format or in_fmt
102
103     if not min_num < pct < max_num:
104         msg = '--pct "{}" must be between {} and {}'
105         die(msg.format(pct, min_num, max_num))
106
107     if not out_file:
108         base, _ = os.path.splitext(file)
109         out_file = '{}.sub{}.{}'.format(base, pct, out_fmt)
110
111     out_fh = open(out_file, 'wt')
112     num_taken = 0
113     total_num = 0
114
```

```
115     with open(file) as fh:
116         for rec in SeqIO.parse(fh, in_fmt):
117             total_num += 1
118             if randint(min_num, max_num) <= pct:
119                 num_taken += 1
120                 SeqIO.write(rec, out_fh, out_fmt)
121                 if max_num_reads > 0 and num_taken == max_num_reads:
122                     break
123
124     out_fh.close()
125
126     print('Wrote {} of {} ({:.02f}%) to "{}"'.format(
127         num_taken, total_num, num_taken / total_num * 100, out_file))
128
129
130 # --------------------------------------------------
131 if __name__ == '__main__':
132     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
```

FASTQ

FASTA (sequence) plus "quality" scores for each base call gives us "FASTQ." Here is an example:

```
$ head -4 !$
head -4 input.fastq
@M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
TTTCTGTGCCAGCAGCCGCGGTAAGACAGAGGTGGCGAGCGTTGTTCGGATTTACTGGGCGTAAAGCGCGGGTAGGCGGTTCGGCCAGTCAC
+
CCCCCGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGEFGGFEGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGFGGC
```

Because of inherent logical flaws in this file format, the only sane representation is for the record to consist of four lines:

1. header ('@', ID, desc, yadda yadda yadda)
2. sequence
3. spacer
4. quality scores (phred 33/64)

Here is what the record looks like:

```
>>> from Bio import SeqIO
>>> rec = list(SeqIO.parse('input.fastq', 'fastq'))[0]
>>> rec = list(SeqIO.parse('input.fastq', 'fastq'))[0]
>>> print(rec)
ID: M00773:480:000000000-BLYPT:1:2106:12063:1841
Name: M00773:480:000000000-BLYPT:1:2106:12063:1841
Description: M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
Number of features: 0
Per letter annotation for: phred_quality
Seq('TTTCTGTGCCAGCAGCCGCGGTAAGACAGAGGTGGCGAGCGTTGTTCGGATTTA...CGC', SingleLetterAlphabet())
```

But this looks pretty much like a FASTA file, so where is the quality information? We have to look here (http://biopython.org/DIST/docs/api/Bio.SeqIO.QualityIO-module.html):

```
>>> print(rec.format("qual"))
>M00773:480:000000000-BLYPT:1:2106:12063:1841 1:N:0:AGGCGACCTTA
34 34 34 34 34 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 36 37 38
38 37 36 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 37 38 38 38 38 38 38 38 38 38 38 38
38 38 38 38 38 38 38 38 36 38 38 38 38 38 38 38 36 38 38 38
38 38 35 38 35 38 38 38 38 38 38 38 38 38 38 38 38 37 35 38
38 38 38 38 38 38 37 37 37 37 35 37 38 22 37 37 38 38 38 38
38 38 38 38 38 22 36 38 38 38 38 38 35 38 38 36 38 38 38 38
38 38 28 36 37 38 35 38 38 37 38 38 35 36 38 38 38 38 37 37
```

```
34 20 26 36 36 35 37 36 37 38 36 38 37 34 37 38 36 36 34 34
23 30 20 34 36 36 9 25 20 9 26 30 37 38 38 37 38 34 34 37
38 32 37 37 38 38 38 35 38 38 37 37 38 34 35 36 34 38 38 38
38 36 26 36 36 23 36 34 28 18 24 15 26 20 22 20 29 23 27 10
24 37 38 38 37 34 27 23 34 38 37 37 25 24 10 24 11 27 35 20
8
```

We can combine the bases and their quality scores into a list of tuples (which can naturally become a dictionary):

```
>>> list(zip(rec.seq, rec.format('qual')))
[('T', '>'), ('T', 'M'), ('T', '0'), ('C', '0'), ...
>>> for base, qual in zip(rec.seq, rec.format('qual')):
...   print('base = "{}" qual = "{}"'.format(base, qual))
...   break
...
base = "T" qual = ">"
```

The scores are based on the ordinal represenation of the quality characters' ASCII values. Cf:

- https://www.rapidtables.com/code/text/ascii-table.html
- https://www.drive5.com/usearch/manual/quality_score.html

We can convert FASTQ to FASTA by simply changing the leading "@" in the header to ">" and then removing lines 3 and 4 from each record. Here is an [g]awk one-liner to do that:

```
#!/bin/gawk -f

### fq2fa.awk
##
## Copyright Tomer Altman
##
### Desription:
##
## Given a FASTQ formatted file, transform it into a FASTA nucleotide file.

(FNR % 4) == 1 || (FNR % 4) == 2 { gsub("^@", ">"); print }
```

Can you write one in Python?

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()
```

XML

Here's an example that looks at XML from the NCBI taxonomy. Here is what the raw file looks like:

```
$ head ena-101.xml
<?xml version="1.0" encoding="UTF-8"?>
<SAMPLE alias="SAMD00024455" accession="DRS018892" broker_name="DDBJ">
     <IDENTIFIERS>
          <PRIMARY_ID>DRS018892</PRIMARY_ID>
          <EXTERNAL_ID namespace="BioSample">SAMD00024455</EXTERNAL_ID>
          <SUBMITTER_ID namespace="">SAMD00024455</SUBMITTER_ID>
     </IDENTIFIERS>
     <TITLE>Surface water bacterial community from the East China Sea Site 100</TITLE>
     <SAMPLE_NAME>
          <TAXON_ID>408172</TAXON_ID>
```

The whitespace in XML is not significant and simply bloats the size of the file, so often you will get something that is unreadable. I recommend you install the program `xmllint` to look at such files. If you inspect the file, you can see that XML gives us a way to represent hierarchical data unlike CSV files which are essentially "flat" (unless you start sticking things like lists and key/value pairs [dictionaries]). We need to use a specific XML parser and use accessors that look quite a bit like file paths. There is a "root" of the XML from which we can descend into the structure to find data. Here is a program that will extract various parts of the XML.

```
$ cat -n xml_ena.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : kyclark
     4  Date   : 2019-02-22
     5  Purpose: Rock the Casbah
     6  """
     7
     8  import argparse
     9  import os
    10  import sys
    11  from xml.etree.ElementTree import ElementTree
    12
    13
    14  # --------------------------------------------------
    15  def get_args():
    16      """get command-line arguments"""
    17      parser = argparse.ArgumentParser(
    18          description='Argparse Python script',
    19          formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    20
    21      parser.add_argument('xml', metavar='XML', help='XML input', nargs='+')
    22
    23      parser.add_argument(
    24          '-o',
```

```
25          '--outdir',
26          help='Output directory',
27          metavar='str',
28          type=str,
29          default='out')
30
31      return parser.parse_args()
32
33
34  # --------------------------------------------------
35  def warn(msg):
36      """Print a message to STDERR"""
37      print(msg, file=sys.stderr)
38
39
40  # --------------------------------------------------
41  def die(msg='Something bad happened'):
42      """warn() and exit with error"""
43      warn(msg)
44      sys.exit(1)
45
46
47  # --------------------------------------------------
48  def main():
49      """Make a jazz noise here"""
50      args = get_args()
51      xml_files = args.xml
52      out_dir = args.outdir
53
54      if not os.path.isdir(out_dir):
55          os.makedirs(out_dir)
56
57      for file in xml_files:
58          print('>>>>>>', file)
59          tree = ElementTree()
60          root = tree.parse(file)
61
62          d = []
63          for key, value in root.attrib.items():
64              d.append(('sample.' + key, value))
65
66          for id_ in root.find('IDENTIFIERS'):
67              d.append(('id.' + id_.tag, id_.text))
68
69          for attr in root.findall('SAMPLE_ATTRIBUTES/SAMPLE_ATTRIBUTE'):
70              d.append(('attr.' + attr.find('TAG').text, attr.find('VALUE').text))
```

```
    71
    72          for key, value in d:
    73              print('{:25}: {}'.format(key, value))
    74
    75          print()
    76
    77  # --------------------------------------------------
    78  if __name__ == '__main__':
    79      main()
$ ./xml_ena.py ena-101.xml
>>>>>> ena-101.xml
sample.alias              : SAMD00024455
sample.accession          : DRS018892
sample.broker_name        : DDBJ
id.PRIMARY_ID             : DRS018892
id.EXTERNAL_ID            : SAMD00024455
id.SUBMITTER_ID           : SAMD00024455
attr.sample_name          : 100A
attr.collection_date      : 2013-08-15/2013-08-28
attr.depth                : 0.5m
attr.env_biome            : coastal biome
attr.env_feature          : natural environment
attr.env_material         : water
attr.geo_loc_name         : China:the East China Sea
attr.lat_lon              : 29.3 N 122.08 E
attr.project_name         : seawater bacterioplankton
attr.BioSampleModel       : MIMARKS.survey.water
attr.ENA-SPOT-COUNT       : 54843
attr.ENA-BASE-COUNT       : 13886949
attr.ENA-FIRST-PUBLIC     : 2015-02-15
attr.ENA-LAST-UPDATE      : 2018-08-15
```

SwissProt

The SwissProt format is one, like GenBank and EMBL, that allows for detailed annotation of a sequence whereas FASTA/Q are primarily devoted to the sequence/quality and sometimes metadata/annotations are crudely shoved into the header line. Parsing SwissProt, however, is no more difficult thanks to the `SeqIO` module. Most of the interesting non-sequence data is in the `annotations` which is a dictionary where the keys are strings like "accessions" and "keywords" and the values are ints, strings, and lists.

Here is an example program to print out the accessions, keywords, and taxonomy in a SwissProt record:

```
$ cat -n swissprot.py
```

```
#!/usr/bin/env python3

import argparse
import sys
from Bio import SeqIO


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

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

    return parser.parse_args()


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


# --------------------------------------------------
def main():
    """main"""
    args = get_args()
    file = args.file

    for i, record in enumerate(SeqIO.parse(file, "swiss"), start=1):
        print('{:3}: {}'.format(i, record.id))
        annotations = record.annotations

        for annot_type in ['accessions', 'keywords', 'taxonomy']:
            if annot_type in annotations:
                print('\tANNOT {}:'.format(annot_type))
                val = annotations[annot_type]
                if type(val) is list:
                    for v in val:
                        print('\t\t{}'.format(v))
                else:
                    print('\t\t{}'.format(val))

```

```
    47
    48
    49  # ------------------------------------------------
    50  if __name__ == '__main__':
    51      main()
$ ./swissprot.py input.swiss
  1: G5EEM5
    ANNOT accessions://
        Nematoda
        Chromadorea
        Rhabditida
        Rhabditoidea
        Rhabditidae
        Peloderinae
        Caenorhabditis
```

You should look at the sample "input.swiss" file to get a greater understanding of what is contained.

JSON

JSON stands for JavaScript Object Notation, and it has become the lingua franca of data exchange on the Internet. For our example, I will use the JSON that is returned by https://www.imicrobe.us/api/v1/samples/578. We need to `import json` and use `json.load` to read from an open file handle (there is also `loads` – load string) to parse the data from JSON into a Python dictionary. We could `print` that, but it's not nearly as pretty as printing the JSON which we can do with `json.dumps` (dump string) and the keyword argument `indent=4` to get nice indentation.

```
$ cat -n json_parse.py
     1  #!/usr/bin/env python3
     2
     3  import json
     4
     5  file = '578.json'
     6  data = json.load(open(file))
     7  print(json.dumps(data, indent=4))
$ ./json_parse.py | head -12
{
    "sample_id": 578,
    "project_id": 26,
    "sample_acc": "CAM_SMPL_GS108",
    "sample_name": "GS108",
    "sample_type": "Metagenome",
    "sample_description": "GS108",
```

```
    "url": "",
    "creation_date": "2018-07-06T04:43:09.000Z",
    "project": {
        "project_id": 26,
        "project_code": "CAM_PROJ_GOS",
```

If you `head 578.json`, you will see there is no whitespace, so this is a nicer way to look at the data; however, if all we wanted was to look at pretty JSON, we could do this:

```
$ python -m json.tool 578.json
```