```
"""Step 1: Read .FASTQ.gz files and ID features

Updated 9 December 2016
(Ready to go)s
Script uses sample data from "../data/sample_data"
Sarah Fortune & JoAnn Flynn labs (VWL)
"""

import numpy as np
import pandas as pd
import regex
import os
import sys
import gzip
import sqlalchemy as sqla
import types
import regex


"""Experiment name to name output files"""
EXPERIMENT = "sample"

"""Directory path(s) to input data
I would recommend putting data (especially large sets) in a separate,
local folder.
"""
INPUT_DIRECTORIES = ["../data/sample_data"]

"""Directory path to save output"""
OUTPUT_DIR = "../output"

"""File path for qtag reference csv.
    csv should have two columns, ordered [qtag id, sequence]
"""
QTAG_CSV = "../helpers/qtag_ref.csv"

"""Required modules; checked by check_input()"""
required_modules = [
    np, pd, regex
    , gzip, sqla
    , os, sys, types
]


"""Motifs used to search for features

Motifs are strings with constant "handles"
    and variable barcode sequences to capture in parentheses

BARCODE_MOTIF(str) for barcode
MCOUNT_MOTIF(str) for molecular counter
INDEX_MOTIF(str) for .fastq.gz-like naming format
    used to parse index names

"""
BARCODE_MOTIF = "CGA([ACTG]{3})C([ACTG]{4})AATTCGATGG"
MCOUNT_MOTIF = "C([ACTG]{3})C([ACTG]{3})C([ACTG]{3})GCGCAACGCG"
```

```
INDEX_MOTIF = "(.+)_S\d{1,3}_L\d{3}_R(\d)_\d{3}\.fastq\.gz"
# ASSERT USER INPUTS ARE VALID TYPES, EXIST (for paths), AND ARE NON-NULL


def check_modules():
    try:
        for mod in required_modules:
            assert mod
            assert type(mod)==types.ModuleType, mod.__name__+" is not a module."

    except (NameError, AssertionError) as e:
        print e
        print "Please ensure the module has been imported and named correctly \
    and has not been redefined."

    else:
        print "All modules have been imported successfully."

class Index(object):

    """class Index(idx,reads,rexs)

    contains tallies from each pair of raw index reads files

    idx(str): index name
    reads(list): list of file paths for fwd and rev reads,
        parsed as file0, file1
    rexs(dict): regex objects to find

    Constructs:
    tname(str): index name with only alphanumeric characters for db table

    Returns: Index(obj)
    """
    def __init__(self, idx, reads, rexs):
        """ init_search(self)

        Opens raw compressed files and initializes iterreads search.
        Returns qbm keyed dict with list of scores as values.
        """
        self.idx = idx
        self.file0, self.file1 = reads[:2]
        self.tname = regex.sub('[^0-9a-zA-Z]+',"",idx)
        self.rexs = rexs

    def search_index(self):
        # open raw .fastq.gz (compressed) files
        try:
            read0 = gzip.open(self.file0)
            read1 = gzip.open(self.file1)
        except Exception, e:
            print "Cannot open read files for %s.\nAborting with Exception: %s"%(self.idx,e)
        else:
            # init file reading
            self.qbm_dict = self.iterreads(read0, read1)
            return self
```

```
    def iterreads(self, read0, read1):
        """ iterreads(self, read0, read1)

        Reads through opened file pairs and records feature search results

        read0(file): opened compressed file for forward read
        read1(file): opened compressed file for reverse read

        Returns dictionary of counts with keys as feature sequences (as tuple)
        and list of paired base quality scores of barcodes and molecular counters
        for each read. Function returns counts to init_search().
        """
        # init output dict and line counter
        line = 0
        counts = {}
        # in chunk, first tuple will contain fwd and rev read sequences,
        # and second tuple with contain base QS for relevant
        # (barcode, molecular counter) features
        chunk = [(),()]
        # iterating through paired read files
        for r0, r1 in zip(read0, read1):
            # if line contains read sequences, save to chunk
            if line == 1:
                chunk[0] = (r0, r1)
            # if line contains base QS, save to chunk and search the reads
            elif line == 3:
                chunk[1] = (r0, r1)
                key, scores = self.search_read(chunk)
                # from search_reads, save with key as feature seqs as tuple, and
                # value as tuple of base QS for barcode and molecular counter features
                counts.setdefault(key,[])
                counts[key].append(scores)
                # line set to -2 to account for (a) += 1 at end of loop, and
                # (b) to read and ignore the fourth row of the read set
                # so that new read set will begin at 0.
                line = -1
            line += 1
        return counts

    def search_read(self, chunk):
        """ search_read(self,chunk)

        Given forward and reverse sequences and base QS for
        a read, search for features using regex motif objects.

        Returns key as tuple of feature sequences, and
        values as the barcode and molecular counter base QS.
        Function returns key and values to iterreads(self, read0,read1).
        """
        # parses chunk values
        seq0, seq1 = chunk[0]
        qs0, qs1 = chunk[1]
        # searches for features (q: qtag, b: barcode, m:molecular counter)
        q = regex.search(self.rexs['q'],seq1)
        b = regex.search(self.rexs['b'],seq0)
        m = regex.search(self.rexs['m'],seq0)
```

```
        # set defaults for scores as 'None'
        # 'None' is used as string to avoid using different types
        # for non-null and null values (i.e. float v str)
        qtag = "None"
        barcode = 'None'
        mcount = 'None'
        bscore = ""
        mscore = ""

#         this is also more verbose and ugly but again,
#         to make code clearer i've written it like this

        """get name of the captured group (i.e. qid) if match
            if barcode and/or molecular counter are found, extract
            the sequence parts (one per group separated by the constant handle)
            and join to form one string, and get base QS for the relevant region
            region includes the entire motif region to also check handles
        """
        if q:
            qtag = q.lastgroup
        if b:
            barcode = "".join(b.groups())
            bscore = qs0[b.start():b.end()]
        if m:
            mcount = "".join(m.groups())
            mscore = qs0[m.start():m.end()]
        # construct key and spans tuples for handoff
        key = (qtag,barcode,mcount)
        scores = [bscore, mscore]
        return key, scores

    def construct_qbm_keyed_df(self, delete_dict=False):

        qbm = pd.DataFrame(self.qbm_dict.keys(), columns=['qtag','barcode','mcount'])
        qbm = qbm.reindex(columns=np.concatenate([qbm.columns,['scores']]))
        qbm.loc[:,'key'] = qbm.apply(lambda x: tuple(x[['qtag','barcode','mcount']]), axis=1)
        qbm.scores = qbm.apply(lambda x: self.qbm_dict[x['key'] ], axis=1)

        self.qbm = qbm
        if delete_dict:
            del self.qbm_dict
        return self

def load_qtags(qtag_csv,id_col=0,sequence_col=1):
    """Load_qtags parses csv,returns pd.DataFrame with qtag id and sequence"""
    # load qtag csv into df
    try:
        qtag_df = pd.DataFrame.from_csv(qtag_csv)
        qtag_df.reset_index(inplace=True)
        df_cols = len(qtag_df.columns)

        assert df_cols == 2, "Incorrect number of columns (%d cols) "%(df_cols)
        assert id_col != sequence_col, "id_col and sequence_col have been assigned the same v
alue "
        assert len(qtag_df) > 0, "Empty dataframe "
    except IOError as e:
        print "Cannot find qtag file at %s. Aborting with Exception: %s."%(qtag_csv,e)
```

```
            sys.stdout.flush()
            sys.exit(1)
        except AssertionError as e:
            sys.stdout.write(e+"\n")
            sys.stdout.write("in qtag file at %s. Aborting with Exception: %s."%(qtag_csv,e))
            sys.stdout.flush()
            sys.exit(1)
        else:
            qtag_df.columns = ['qid','seq']
        # format and wrangle df to output
        qtag_df.qid = qtag_df.qid.apply(lambda x: 'q'+str(x))
        qtag_df.seq = qtag_df.seq.str.upper()
        qtag_df.set_index('seq',inplace=True)
        return qtag_df


    def make_rexs(barcode_motif, mcount_motif, qtags):
        '''Construct regex motifs for features and return dict of names and regex objs'''
        # construct qtag motif from qtag_df, with capture groups named by qid
        qtag_motif = "|".join(['(?P<%s>%s)'%(q.qid,seq) for seq,q in qtags.iterrows()])
        qtag_regex = regex.compile(qtag_motif, flags=regex.I)
        # construct barcode and molecular counter motifs from user input
        barcode_regex = regex.compile(barcode_motif, flags=regex.I)
        mcount_regex = regex.compile(mcount_motif, flags=regex.I)
        return {'q':qtag_regex,'b':barcode_regex,'m':mcount_regex}

    def create_indexes(root, rexs):
        '''Finds relevant files with index motif and returns dict of Index obj'''
        indexes = {}
        # checks that directory exists

        if os.path.isdir(root):
            for directory, sub, files in os.walk(root):
                for f in files:
                    # check if file should be read (i.e. .fastq.gz)
                    term = regex.search(INDEX_MOTIF, f)
                    # if valid
                    if term and term[0]!='Undetermined':
                        # get capture groups index name (idx), read num {0,1}
                        idx, read = term.groups()
                        read = int(read)
                        # add file entry to output dict
                        indexes.setdefault(idx, ["",""])
                        indexes[idx][int(read)-1] = directory+"/"+f
        for idx in indexes:
            indexes[idx] = Index(idx, indexes[idx], rexs)
        return indexes



    class Counts(object):
        """ Counts object(self, idx, counts)

        used to manipulate tallied data from Index
        to get final, non-thresholded but filtered library-ID-barcode counts

        Requires:
```

```
    idx(str):  index name
    counts(dict): dict containing feature combinations (qtag-barcode-molecularcounter)
        as keys, and list of base QS for barcode and/or molecular counter regions as values.
        (counts dict is generated by Index object)
    """

    def __init__(self, idx):
        """init object with idx dict"""
        self.idx = idx

    @staticmethod
    def count_readsPF(scores):
        """ID reads with min QS < 30 and returns number of readsPF"""
        readsPF = 0
        # base score chars that represent QS >= 30
        pass_scores = "[^\?@ABCDEFGHI]"
        for readQS in scores:
            # search for any chars that are not in the high-QS char set
            qs_0 = regex.search(pass_scores,readQS[0])
            qs_1 = regex.search(pass_scores,readQS[1])
            # if both reads are good, add to PF count
            if (not qs_0) and (not qs_1):
                readsPF += 1
        return readsPF

    def filter_reads(self, qbm):
        """filter out reads by QS and drop empty data rows"""
        # trim down df to include only valid reads (all features present)
        # to cut down on processing time
        qbm = qbm.loc[ (qbm.qtag!="None")
                      &(qbm.barcode!="None")
                      &(qbm.mcount!="None")]
        # counts reads PF and remove those with no reads PF (cut time)
        qbm['readsPF'] = qbm.scores.apply(self.count_readsPF)
        qbm = qbm.loc[qbm.readsPF > 0]
        # count reads
        qbm['reads'] = qbm.scores.apply(lambda x: len(x))
        self.filtered_qbm = qbm
        return self


    def count_molecs(self):
        """from qbm data, counts number of molecular counters per qb"""
        filtered_qbm = self.filtered_qbm
        # count molecs(PF) and sum readsPF
        pivoted = pd.pivot_table(filtered_qbm, index=['qtag','barcode'],values='readsPF', agg
func=[sum, len])
        # formatting
        pivoted.rename(columns={'sum':'readsPF','len':'mcountsPF'},inplace=True)
        pivoted.sort_values(by=['mcountsPF','readsPF'],ascending=False, inplace=True)
        pivoted.reset_index(inplace=True)
        pivoted['idx'] = self.idx
        self.filtered_qb = pivoted
        return self

if_exists = 'replace'
overwrite = True
```

```
open_type = 'wb' if overwrite else 'ab'
col_order = ["idx","qtag","barcode","mcountsPF","readsPF"]
# f = open(output_csv_fpath, open_type)
INPUT_DIRECTORIES = ["%s/%s"%(MAIN_DIRECTORY,d)
                     for d in ['18014','18114'] ]


def run(db_name=None):
    # build required constants and regex
    qtags = load_qtags(QTAG_CSV)
    rexs = make_rexs(BARCODE_MOTIF, MCOUNT_MOTIF, qtags)
    # init sqldb
    db_name = db_name.split(".db")[0] if db_name!=None else "counts_%s"%EXPERIMENT
    db_filepath = 'sqlite:///%s/%s.db'%(OUTPUT_DIR, db_name)
    engine = sqla.create_engine(db_filepath)
#     tables = engine.table_names()
    tables = []
    # iterate through directories provided
    for directory in INPUT_DIRECTORIES:
        run_name = regex.split(r"\/",directory)[-1]
        indexes = create_indexes(directory, rexs)
        iterum = 1
        output_csv_fpath = "%s/filtered-%s-%s.csv"%(OUTPUT_DIR,EXPERIMENT,run_name)
        header = True
#         f = open(output_csv_fpath, open_type)
        # iterate through each index
        for idx_name in indexes:
            sys.stdout.write("\nStarting %d of %d: %s"%(iterum, len(indexes), idx_name))
            sys.stdout.flush()
s                index = indexes[idx_name]
                # search and ID features from raw files
                index.search_index()
                index.construct_qbm_keyed_df()
                # count
                sys.stdout.write("...")
                sys.stdout.flush()
                # init Counts obj, QS filter reads and count molecs
                counts = Counts(idx_name)
                counts.filter_reads(index.qbm)
                if len(counts.filtered_qbm) > 0:
                    counts.count_molecs()

                    # save to sqldb
                    conn = engine.connect()
                    counts.filtered_qb.to_sql(idx_name, engine, if_exists=if_exists)
                    conn.close()
                    # write to file
                    csv = counts.filtered_qb.to_csv(header=header)
                    header = False
                    f.write(csv)
                    f.flush()
                sys.stdout.write("Finished.")
                sys.stdout.flush()
            iterum+=1

    # clean up
    engine.dispose()
```

```
    sys.stdout.write('\nJob complete\n')
    sys.stdout.flush()

if __name__=='__main__':
    check_modules()
    run()
```