```
# NO QTAG ERRORS ALLOWED
import numpy as np
import pandas as pd
import regex
import os,sys
import gzip
import sqlalchemy as sqla

'''CONSTANTS defined by user'''

EXPERIMENT = "2016-08-04-nates1"
INPUT_DIRECTORIES = ["../data/nate"]
OUTPUT_DIR = "../output"

QTAG_CSV = "../helpers/qtags_var.csv"

GTAG_MOTIF = "CGA(?P<gtag>[ACTG]{3})C(?P<gtag>[ACTG]{4})AATTCGATGG"
MCOUNT_MOTIF = "C(?P<mcount>[ACTG]{3})C(?P<mcount>[ACTG]{3})C(?P<mcount>[ACTG]{3})GCGCA
FILE_MOTIF = "(?P<sample>.+)_(?P<sample_barcode>.+)_L(?P<lane>\d{3})_R(?P<read_number>\
READ_REF_DEFAULT = {'q':1, 'g':0, 'm':0}

IF_SQLTABLE_EXISTS = 'replace'
DEFAULT_DB_NAME = "counts-%s.db"%EXPERIMENT

'''HELPERS'''
# used only to make regex motifs, but
# not nested to preserve qtag loading functionality if desired
def load_qtags(qtag_csv):
    try:
        qtagdf = pd.DataFrame.from_csv(qtag_csv).reset_index()
        qtagdf.rename(columns={'qtag_seq':'seq', 'qtag_num':'qid'}, inplace=True)
        qtagdf.qid = qtagdf.qid.apply(lambda x: "q%s"%str(x))
        qtagdf.seq = qtagdf.seq.str.upper()
        qtagdf.set_index('seq', inplace=True)
    # TO DO: CHECK FOR DUPLICATE SEQUENCES OR NAMES
    except IOError as e:
        print "Unable to load qtag file, with error:", e
        sys.exit(1)
    return qtagdf


# construct regex motif dict for read search
def make_rexs(qtag_csv):
    # load and construct qtag motif as OR list of each qtag seq (named)
    qtags = load_qtags(qtag_csv)
    qtag_phrases = qtags.apply(lambda x: '(?P<%s>%s)'%(x.qid, x.name) , axis=1)
    qtag_motif = "|".join( qtag_phrases.values )
    # return compiled motifs for qtag, gtag (barcode), and molec counter, resp.
```

```
        return {'q':regex.compile(qtag_motif, flags=regex.I),
                'g':regex.compile(GTAG_MOTIF, flags=regex.I),
                'm':regex.compile(MCOUNT_MOTIF, flags=regex.I)}

    # this looks gross but works for now; make pretty later
    def get_file_list(root):
        fpath_temp_a = []
        fil_temp_a = []
        # construct list of files and their infodict, as tuples:
        # (i.e. <sample>_<sample_barcode>_L<lane>_R<read_number>_<set_number>)
        for direct, sub, fil in os.walk(root):
            fpaths = np.array( [ "%s/%s"%(direct,f)  for f in fil] )
            to_append = np.array([regex.search(FILE_MOTIF,f) for f in fil ])
            fil_temp_a.append( to_append )
            fpath_temp_a.append(fpaths)

        fil_temp_b = np.concatenate(fil_temp_a)
        fpath_temp_b = np.concatenate(fpath_temp_a)
        fil_temp_c = fil_temp_b[np.nonzero(fil_temp_b)]
        fpath_temp_c = fpath_temp_b[np.nonzero(fil_temp_b)]
        files = np.array( [(fp, fil.groupdict()) for (fp, fil) in zip(fpath_temp_c, fil_tem
        return files

    def init_indexes(root):
        files = get_file_list(root)
        ### FIX :  files list item fmt:  (fpath, fil.str)
        indexes = dict([(f[1]['sample'],["",""]) for f in files])
        for fpath, match in files:
            if match['sample']!='Undetermined':
                # assumes 2 reads (fwd and reverse)
                indexes[match['sample']][int(match['read_number'])-1] = fpath
        if len(indexes) == 0:
            print "Empty index list. No valid files. Please check your input directory and
            sys.exit(1)
        # convert idx entry list of files to Index object
        for idx, idx_paths in indexes.items():
            indexes[idx] = Index(idx, idx_paths)
        return indexes
    # modified opening .gz file with error/exception catching
    # 15 aug 2016

    # with zip(gzip.open(self.file0), gzip.open(self.file1)) as f0, f1:
    def open_gz(fpath):
        try:
            f_gen = gzip.open(fpath)
            return f_gen
        except EnvironmentError as e:
            print '%s "%s". Please check your file and/or directory paths. Skipping index.
                    e.strerror, e.filename, e.errno)
        except TypeError as e:
            print "TypeError: %s. Skipping index."%e
```

```
        except BaseException as e:
            print 'Other error: %s. Skipping index.'%e
        return None
    def sysprint(msg,tab_num=0):
        tabs = "".join(["\t" for t in range(tab_num)])
        sys.stdout.write("%s%s\n"%(tabs, msg))
        sys.stdout.flush()
        return

    ''' Index

    Updated 15 August 2016 -- need to test all class methods together,
    but otherwise cleaned

    - Added read_ref option for UX flexibility
    - Streamlined count_reads logic and flow, calling motif_search
    - Major modifications to motif search; generalize search and feature extraction for eac

    '''
    class Index(object):

        # defining read_ref as instance variable so that
        # if user uses multiple read rexs or refs, changing
        # var won't affect previously defined objects

        def __init__(self, idx, fpaths, read_ref=READ_REF_DEFAULT):
            self.idx = idx
            self.file0, self.file1 = fpaths
            # read_ref as dict
            self.read_ref = read_ref
            self.tname = regex.sub('[^0-9a-zA-Z]+',"",idx)

        # so ugly i'm cringing but should probably not change it
        # for this v1 version
        def count_reads(self):
            counts = {}
            # such that line 1 is seq, line 3 is qs
            line = 0
            entry_len = 4
            gz0, gz1 = [open_gz(self.file0), open_gz(self.file1)]
            if gz0 and gz1:
                chunk = [(),()]
                for r0,r1 in zip(gz0, gz1):
                    if line==1: chunk[0] = (r0,r1)  # sequence
                    elif line==3: chunk[1] = (r0,r1)  # q scores
                    if line+1 > entry_len:
                        key,qscores = self.motif_search(chunk[0],chunk[1])
                        counts.setdefault(key,[])
                        counts[key].append(qscores)
                        chunk = [(),()]
                        line = -1
```

```
                    line += 1

            return counts

        def motif_search(self, seqs, qscores, order=['q','g','m']):
            keys = ['None' for _ in order]
            qs_seqs = ""
            searches = [(feature, read, regex.search(REXS[feature], seqs[read]))
                        for feature, read in self.read_ref.items() ]

            for feature, i in zip( order, range(len(order)) ):
                r = self.read_ref[feature]
                search = regex.search(REXS[feature], seqs[r])
                if search:
                    match = search.capturesdict()
                    extracted = filter(lambda x: len(match[x])>0, match)
                    if len(extracted) == 1:
                        k = extracted[0]
                        keys[i] = k if feature=='q' else "".join(match[k])
                        qs_seqs += "" if feature=='q' else qscores[r][search.start():search
                    else:
                        print "Error: non-unique sequence"
            return tuple(keys), qs_seqs

    ''' Counts

    NOTE ON QSCORE FORMATS (ref. fn calculate_count_minscore)
    Q-scores for Illumina 1.8+ (most recent as of Aug 2016) ranges
    from 33 to 73 (Phred+33 system). P, the probability of erroneous base call,
    is defined as:  P(erroneous base call) = 10 ^ (Qphred / -10), i.e.
    for Illumina 1.8+, P = 10^( (QS-33)/-10 ).

    The minimum QS cutoff is set at an error probability
    of 10^-3 (standard for Illumina system).
    '''
    class Counts(object):
        def __init__(self, idx, directory):
            self.idx = idx
            self.directory = directory

        '''STATIC METHOD GENERATORS'''

        # generator for parsing raw df qgm keys
        # row is pd.Series
        @staticmethod
        def parse_qgm_key(row, order=['q','g','m']):
            # parse qgm key
            for feature, seq in zip(order, row[0]):
                row[feature] = seq
            return row
```

```
        # generator for calculating and counting read minscores (PF)
        # row is pd.Series
        @staticmethod
        def calculate_count_minscore(row):
            # define variables as null values
            keys = [ 'reads_total','reads_pf','molec_passed' ]
            vals = [0,0,False]
            q, g, m = row[['q','g','m']]
            valid = (q!='None') and (g!='None') and (m!='None')
            # if key is valid (i.e. non-null q, g, and m), count
            if valid:
                try:
                    min_qscores = np.array([])
                    # get min qscores for each read_qs in row (i.e. per read)
                    for read_qs in row[1]:
                        read_min_qs = 0 if len(read_qs)==0 else np.min(
                                              [ord(s) for s in read_qs])
                        min_qscores = np.append(min_qscores,read_min_qs)
                    # calculate values for variables
                    vals = [ len(min_qscores), #reads_total
                             len(min_qscores[np.where(min_qscores>=63)]), #reads_pf
                             True if max(min_qscores) > 0 else False ] #molec_passed
                except Exception as e:
                    print e
                    print row
                    sys.exit(1)
            # assign new values to row
            for k, v in zip(keys, vals) : row[k] = v
            return row

        '''counts number of molecular counters and
        reads that PF per qg group is pd.DataFrame'''
        @staticmethod
        def count_qg(group):
            s = pd.Series()
            molecs = group.loc[group.reads_pf>0].molec_passed
            s['molecs'], s['reads'] = np.sum(molecs), np.sum(group.reads_pf)
            return s

        '''INSTANCE FUNCTIONS'''

        '''construct_qgm_df

        creates pd.DataFrame from Index.counts_dict, and
        1) parses qgm key after calling df;
        2) calculates min read qscore for each read; and
        3) counts reads PF and drops qscore seqs to save memory.
        4) classify qgm as 'pass' or 'fail' QC if at least one read PF
        5) returns qgm_df to export to db if desired without saving to memory
        '''
        def construct_qgm_counts_df(self, counts):
```

```
            # create df from counts;
            # abbr qgmcounts_df to df for easier digestion
            df = pd.DataFrame.from_dict(counts.items())
            # count and consolidate qg
            df = df.apply(self.parse_qgm_key, axis=1)
            df = df.apply(self.calculate_count_minscore, axis=1)
            keep = ['q','g','m','reads_total','reads_pf','molec_passed']
            return df[keep]

        ''' construct final filtered df from qgm_counts_df
        1) per qg, counts no. molecs PF, reads PF
         (as 'molecs' and 'reads', respectively)
        2) classify qg FILTER to eliminate extraneous data, ie.
         must satisfy (a) molecs > 0, and (b) all features non-null
        '''
        def construct_filtered_df(self, qgm_counts_df):
            # returns bool for the passing conditions
            # ie. (has molecs, and all features present/non-null)
            def pass_conditions(x):
                cond1 = x.molecs > 0
                cond2 = not('None' in x[['q','g','m']].values)
                return True if cond1 and cond2 else False

            # again, abbr qg_counts_df to df for easier digestion
            df = qgm_counts_df.groupby(['q','g'], as_index=False).apply(self.count_qg)
            df.loc[:,'passed'] = df.apply(lambda x: pass_conditions(x), axis=1)

            # exclude QC fails, sort and re-index
            df = df.loc[df.passed==True]
            df.reset_index(inplace=True)
            df.sort_values(by=['molecs'], ascending=False, inplace=True)
            # for a polished df (re-index to make idx numbers ascending)
            df.reset_index(inplace=True, drop=True)
            self.filtered_counts = df
            print df.head()
            print self.filtered_counts.head()
            return df

    def run(db_name=DEFAULT_DB_NAME, quiet=False):
        stats = {}

        # define output files
        db_name = db_name.split(".db")[0]
        db_path = 'sqlite:///%s/%s.db'%(OUTPUT_DIR, db_name)
        engine = sqla.create_engine(db_path)

        # generate a new csv file and open
        filtered_fpath = '%s/filtered-%s'%(OUTPUT_DIR,EXPERIMENT)
        open('%s.csv'%filtered_fpath, 'a').close()
        f = open('%s.csv'%filtered_fpath, 'a')
        header = True
```

```
    # iterate through directories/indexes
    for directory in INPUT_DIRECTORIES:
        # prep indexes
        indexes = init_indexes(directory)
        sysprint('\nStarting directory: %s'%directory.split("/")[-1:][0])

        # run analysis for each index
        idx_names = indexes.keys()[:5]
        i = 1
        while i < len(idx_names):
            idx_name, index = idx_names[i], indexes[idx_names[i]]
            idx_message = '\nIndex %d of %d: %s'%(i,len(indexes),idx_name)
            sysprint(idx_message)

            # actually execute analysis
            counts_dict = index.count_reads()
            sysprint('...counted')

            # init counts object with idx name, directory
            counts = Counts(idx_name, directory)
            qgm_counts_df = counts.construct_qgm_counts_df(counts_dict)
            sysprint('...qgm done')

            # save qgm_counts_df
            conn = engine.connect()
            qgm_counts_df.to_sql(idx_name, conn, if_exists='replace')
            conn.close()
            sysprint('...saved')

            # construct and save filtered_df
            counts.construct_filtered_df(qgm_counts_df)
            counts.filtered_counts['idx'] = idx_name
            counts.filtered_counts['directory'] = directory.split("/")[-1:]
            sysprint('...filtered')

            # write to output files for ref
            counts.filtered_counts.to_csv(f, header=header)
            sysprint('...filtered to csv.')
            header = False
            i+=1

    f.close()
    engine.dispose()
    sysprint('Job complete\n')
    return stats

'''EXECUTE SCRIPT'''
REXS = make_rexs(QTAG_CSV)
stats = run()
```

```
'''TEST CELLS'''

test = '9615-01_S9_L001_R1_001.fastq.gz'
REXS = make_rexs(QTAG_CSV)
directory = INPUT_DIRECTORIES[0]
indexes = init_indexes(directory)
testi = indexes.values()[1]
counts_dict = testi.count_reads()

testcount = Counts(testi.idx)
qgm = testcount.construct_qg_df(counts_dict)

'''NEED TO DEAL WITH / REWRITE'''

def get_stats(self):
    valid = self.df.loc[(self.df.qtag!='None')&
                        (self.df.gtag!='None')&
                        (self.df.mcount!='None')]
    idxstats = {'total reads': len(self.df),
        'mcounts with qtag, gtag and mcount': len(valid.groupby(['qtag','gtag','mcount'
        'reads with qtag, gtag and mcount': len(valid),
        'reads with only no qtag': self.get_read_counts(self.df, False, True, True),
        'reads with only no gtag': self.get_read_counts(self.df, True, False, True),
        'reads with only no mcount': self.get_read_counts(self.df, True, True, False),
        'reads with only mcount': self.get_read_counts(self.df,False,False,True),
        'reads with only barcode': self.get_read_counts(self.df, False,True,False),
        'reads with only qtag': self.get_read_counts(self.df, True,False,False),
        'reads with no qtag, barcode or mcount': self.get_read_counts(self.df,False,Fal
    }
    return idxstats

pd.DataFrame.from_dict(data_stats).T.to_csv("%s/%s_stats.csv"%(OUTPUT_DIR,EXPERIMENT))

'''CODE RESOURCES / IDEAS FROM THE NETIZENS'''

'''conditional import'''
try:
    import json
except ImportError:
    import simplejson as json

'''concatenating multiple files into one file'''
filenames = ['file1.txt', 'file2.txt', ...]
with open('path/to/output/file', 'w') as outfile:
    for fname in filenames:
        with open(fname) as infile:
            for line in infile:
                outfile.write(line)

'''idea for UX flexibility and ease'''
def format_motif(user): return
```

```
'''example of xlsxwriter'''
import xlsxwriter

# Create an new Excel file and add a worksheet.
workbook = xlsxwriter.Workbook('demo.xlsx')
worksheet = workbook.add_worksheet()

# Widen the first column to make the text clearer.
worksheet.set_column('A:A', 20)

# Add a bold format to use to highlight cells.
bold = workbook.add_format({'bold': True})

# Write some simple text.
worksheet.write('A1', 'Hello')

# Text with formatting.
worksheet.write('A2', 'World', bold)

# Write some numbers, with row/column notation.
worksheet.write(2, 0, 123)
worksheet.write(3, 0, 123.456)

# Insert an image.
worksheet.insert_image('B5', 'logo.png')

workbook.close()

'''OLD CODE I MIGHT NEED'''

'''test run used 17 Aug 2016 to debug run'''

def testrun(db_name=DEFAULT_DB_NAME, quiet=False):
    stats = {}
    counts_output = pd.DataFrame()
    dfoutput = pd.DataFrame()
    # define output files
    db_name = db_name.split(".db")[0]
    db_path = 'sqlite:///%s/%s.db'%(OUTPUT_DIR, db_name)
    engine = sqla.create_engine(db_path)

    filtered_fpath = '%s/filtered-%s'%(OUTPUT_DIR,EXPERIMENT)
    try:
        # iterate through directories/indexes
        for directory in INPUT_DIRECTORIES:
            # prep indexes
            indexes = init_indexes(directory)
            sysprint('\nStarting directory: %s'%directory.split("/")[-1:][0])
            # run analysis for each index
            idx_names = indexes.keys()[:2]
```

```
            i = 1
            while i < len(idx_names):
                idx_name, index = idx_names[i], indexes[idx_names[i]]
                idx_message = '\nIndex %d of %d: %s'%(i,len(indexes),idx_name)
                sysprint(idx_message)

                # actually execute analysis
                counts_dict = index.count_reads()
                sysprint('...counted')

                # init counts object with idx name, directory
                counts = Counts(idx_name, directory)

                qgm_counts_df = counts.construct_qgm_counts_df(counts_dict)
                dfoutput = qgm_counts_df

                counts.construct_filtered_df(qgm_counts_df)
                counts.filtered_counts['idx'] = idx_name
                counts.filtered_counts['directory'] = directory
                sysprint('...filtered to csv.')
                header = False
                i+=1
        except Exception as e:
            print e

            pass
        sysprint('Job complete\n')
        return counts_output, dfoutput
```