```
# pipeline to generate read counts and phenotype scores directly from gzipped
sequencing data

import os
import sys
import gzip
import multiprocessing
import fnmatch
import glob
import argparse

### Sequence File to Trimmed Fasta Functions ###

def parallelSeqFileToCountsParallel(fastqGzFileNameList, fastaFileNameList,
countFileNameList, processPool, libraryFasta, startIndex=None, stopIndex=None,
test=False):

    if len(fastqGzFileNameList) != len(fastaFileNameList):
        raise ValueError('In and out file lists must be the same length')

    arglist = zip(fastqGzFileNameList, fastaFileNameList, countFileNameList,
[libraryFasta]*len(fastaFileNameList),

[startIndex]*len(fastaFileNameList),[stopIndex]*len(fastaFileNameList),
[test]*len(fastaFileNameList))

    readsPerFile = processPool.map(seqFileToCountsWrapper, arglist)

    return zip(countFileNameList,readsPerFile)


def seqFileToCountsWrapper(arg):
    return seqFileToCounts(*arg)


def seqFileToCounts(infileName, fastaFileName, countFileName, libraryFasta,
startIndex=None, stopIndex=None, test=False):
    printNow('Processing %s' % infileName)

    fileType = None

    for fileTup in acceptedFileTypes:
        if fnmatch.fnmatch(infileName,fileTup[0]):
            fileType = fileTup[1]
            break

    if fileType == 'fqgz':
        linesPerRead = 4
        infile = gzip.open(infileName,mode='rt')
    elif fileType == 'fq':
        linesPerRead = 4
        infile = open(infileName)
    elif fileType == 'fa':
        linesPerRead = 2
        infile = open(infileName)
    else:
        raise ValueError('Sequencing file type not recognized!')


    seqToIdDict, idsToReadcountDict, expectedReadLength = 
parseLibraryFasta(libraryFasta)

    curRead = 0
    numAligning = 0

    with open(fastaFileName,'w') as unalignedFile:
        for i, fastqLine in enumerate(infile):
            if i % linesPerRead != 1:
                continue

            else:
                seq = fastqLine.strip()[startIndex:stopIndex]

                if i == 1 and len(seq) != expectedReadLength:
                    raise ValueError('Trimmed read length does not match expected 
reference read length')

                if seq in seqToIdDict:
                    for seqId in seqToIdDict[seq]:
                        idsToReadcountDict[seqId] += 1

                    numAligning += 1

                else:
                    unalignedFile.write('>%d\n%s\n' % (i, seq))

                curRead += 1

                #allow test runs using only the first N reads from the fastq file
                if test and curRead >= testLines:
                    break

    with open(countFileName,'w') as countFile:
        for countTup in (sorted(zip(idsToReadcountDict.keys(), 
idsToReadcountDict.values()))):
            countFile.write('%s\t%d\n' % countTup)

    printNow('Done processing %s' % infileName)

    return curRead, numAligning, numAligning * 100.0 / curRead
```

```
### Map File to Counts File Functions ###

def parseLibraryFasta(libraryFasta):
    seqToIds, idsToReadcounts, readLengths = dict(), dict(), []

    curSeqId = ''
    curSeq = ''

    with open(libraryFasta) as infile:
        for line in infile:
            if line[0] == '>':
                if curSeqId != '' and curSeq != '':
                    if curSeq not in seqToIds:
                        seqToIds[curSeq] = []
                    seqToIds[curSeq].append(curSeqId)

                    idsToReadcounts[curSeqId] = 0

                    readLengths.append(len(curSeq))

                curSeqId = line.strip()[1:]
                curSeq = ''

            else:
                curSeq += line.strip().upper()

        #at the end, add the final item that was not covered in the loop
        if curSeqId != '' and curSeq != '':
            if curSeq not in seqToIds:
                seqToIds[curSeq] = []
            seqToIds[curSeq].append(curSeqId)

            idsToReadcounts[curSeqId] = 0

            readLengths.append(len(curSeq))

    if len(seqToIds) == 0 or len(idsToReadcounts) == 0 or readLengths[0] == 0:
        raise ValueError('library fasta could not be parsed or contains no 
sequences')
    elif max(readLengths) != min(readLengths):
        print(min(readLengths), max(readLengths))
        raise ValueError('library reference sequences are of inconsistent lengths')

    return seqToIds, idsToReadcounts, readLengths[0]


### Utility Functions ###
def parseSeqFileNames(fileNameList):
    infileList = []
    outfileBaseList = []

    for inputFileName in fileNameList:                    #iterate through entered
filenames for sequence files
        for filename in glob.glob(inputFileName):         #generate all possible
files given wildcards
            for fileType in list(zip(*acceptedFileTypes))[0]:    #iterate through
allowed filetypes
                if fnmatch.fnmatch(filename,fileType):
                    infileList.append(filename)

outfileBaseList.append(os.path.split(filename)[-1].split('.')[0])

    return infileList, outfileBaseList

def makeDirectory(path):
    try:
        os.makedirs(path)
    except OSError:
        #printNow(path + ' already exists')
        pass

def printNow(printInput):
    print(printInput)
    sys.stdout.flush()

### Global variables ###
acceptedFileTypes = [('*.fastq.gz','fqgz'),
                    ('*.fastq', 'fq'),
                    ('*.fq', 'fq'),
                    ('*.fa', 'fa'),
                    ('*.fasta', 'fa'),
                    ('*.fna', 'fa')]


testLines = 10000


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Process raw sequencing data from
screens to counts files in parallel')
    parser.add_argument('Library_Fasta', help='Fasta file of expected library
reads.')
    parser.add_argument('Out_File_Path', help='Directory where output files should
be written.')
    parser.add_argument('Seq_File_Names', nargs='+', help='Name(s) of sequencing
file(s). Unix wildcards can be used to select multiple files at once. The script
will search for all *.fastq.gz, *.fastq, and *.fa(/fasta/fna) files with the given
wildcard name.')
```

```
    parser.add_argument('-p','--processors', type=int, default = 1)
    parser.add_argument('--trim_start', type=int)
    parser.add_argument('--trim_end', type=int)
    parser.add_argument('--test', action='store_true', default=False, help='Run the
entire script on only the first %d reads of each file. Be sure to delete or move
all test files before re-running script as they will not be overwritten.' %
testLines)

    args = parser.parse_args()
    #printNow(args)

    ###catch input mistakes###
    numProcessors = max(args.processors, 1)

    infileList, outfileBaseList = parseSeqFileNames(args.Seq_File_Names)
    if len(infileList) == 0:
        sys.exit('Input error: no sequencing files found')

    try:
        seqToIdDict, idsToReadcountDict, expectedReadLength =
parseLibraryFasta(args.Library_Fasta)

        printNow('Library file loaded successfully:\n\t%.2E elements (%.2E unique
sequences)\t%dbp reads expected' \
                % (len(idsToReadcountDict), len(seqToIdDict), expectedReadLength))

    except IOError:
        sys.exit('Input error: library fasta file not found')

    except ValueError as err:
        sys.exit('Input error: ' + err.args[0])


    trimmedFastaPath = os.path.join(args.Out_File_Path,'unaligned_reads')
    makeDirectory(trimmedFastaPath)
    countFilePath = os.path.join(args.Out_File_Path,'count_files')
    makeDirectory(countFilePath)

    fastaFileNameList = [outfileName + '_unaligned.fa' for outfileName in
outfileBaseList]
    fastaFilePathList = [os.path.join(trimmedFastaPath, fastaFileName) for
fastaFileName in fastaFileNameList]
    countFilePathList = [os.path.join(countFilePath,outfileName + '_' +
os.path.split(args.Library_Fasta)[-1] + '.counts') for outfileName in
outfileBaseList]

    pool = multiprocessing.Pool(min(len(infileList),numProcessors))

    try:
        resultList = parallelSeqFileToCountsParallel(infileList, fastaFilePathList,
countFilePathList, pool, args.Library_Fasta, args.trim_start, args.trim_end,
args.test)
    except ValueError as err:
        sys.exit('Error while processing sequencing files: ' + ' '.join(err.args))

    for filename, result in resultList:
        print(filename + ':\n\t%.2E reads\t%.2E aligning (%.2f%%)' % result)

    pool.close()
    pool.join()

    printNow('Done processing all sequencing files')
```