```
#!/usr/bin/python

from __future__ import division
import math,re,time,sys,os,random
import cPickle as cpickle
import urllib
from Bio import Seq as seq
from Bio import SeqIO as seqio
import assign_to_ORFs as gff
from Bio import SeqUtils
import warnings
import gc

fl = sys.stdout.flush
warnings.filterwarnings('error')

# Change these values to alter the unique string that the program uses
to separate mutant from WT data.
M = 'M'
WT = 'WT'


class ORF():

    def __init__(self,id,c1,c2,strand,name='',note=''):
        self.note = note
        self.name = name
        self.id = id
        c1 = int(c1)
        c2 = int(c2)
        if strand == '+':
            self.start = min([c1,c2])
            self.end = max([c1,c2])
        elif strand == '-':
            self.start = max([c1,c2])
            self.end = min([c1,c2])
        else: raise ValueError('Strand neither + nor -')
        self.strand = strand
        self.regions = []
        self.five_prime_UTR = []
        self.three_prime_UTR = []

    def __hash__(self):
        return int(self.name[2:])

    def coords(self):
        c = [self.start,self.end]
        c.sort()
        return c
```

```
    def start_end_list(self,c1c2):
        return self.start_end(c1c2[0],c1c2[1])

    def start_end(self,c1,c2):
        if self.strand == '+':
            start = min([c1,c2])
            end = max([c1,c2])
        elif self.strand == '-':
            start = max([c1,c2])
            end = min([c1,c2])
        else: raise ValueError('Strand neither + nor -')
        return [start,end]

    def get_intron_sites(self,pos=0):
        sites = []
        for section in self.regions:
            if section[0] == 'intron':
                accept.append([self.strand,section[0][pos]])
        return sites

    def add_region(self,typeof,c1,c2):
        c1 = int(c1)
        c2 = int(c2)
        self.regions.append([typeof,self.start_end(c1,c2)])

    def add_UTR(self,typeof,c1,c2):
        c1 = int(c1)
        c2 = int(c2)
        if typeof == 'three_prime_UTR':
            self.three_prime_UTR = self.start_end(c1,c2)
        elif typeof == 'five_prime_UTR':
            self.five_prime_UTR = self.start_end(c1,c2)

    def ORF_region(self):
        return [self.start,self.end]

    def get_full_length(self):
        return self.start_end_list(self.five_prime_UTR +
self.three_prime_UTR + self.ORF_region())

    def is_in_range(self,r,coord):
        r.sort()
        if coord < r[0] or coord > r[1]:
            return False
        else:
            return True

    def is_exon(self,coord):
        for item in self.regions:
            if self.is_in_range(item[1],coord) and item[0] == 'CDS':
```

```
                return item[1]
        return []

    def is_intron(self,coord):
        for item in self.regions:
            if self.is_in_range(item[1],coord) and item[0] ==
'intron':
                return item
        return False

    def is_UTR(self,coord):
        if self.three_prime_UTR:
            if self.is_in_range(self.three_prime_UTR,coord):
                return 'three_prime_UTR'
        if self.five_prime_UTR:
            if self.is_in_range(self.five_prime_UTR,coord):
                return 'five_prime_UTR'
        return ''

    def get_coding_sequence(self,chr,debug = False):
        final = seq.Seq('')
        # slices of seq object are done this way: AACAAC [1:4] gives
ACA
        self.regions.sort(key = lambda x: x[1][0])

        for item in self.regions:
            if item[0] == 'CDS':
                final = final + chr.seq[min(item[1]) - 1:
max(item[1])  ]

        if self.strand == '-':
            final = final.reverse_complement()

        if debug:
            # test whether it is of the right length
            length = 0
            for item in self.regions:
                if item[0] == 'CDS':
                    l2 = abs(item[1][1] - item[1][0]) + 1
                    l1 = len(chr.seq[min(item[1]) - 1:
max(item[1])  ] )
                    if l1 != l2:
                        print "Extracted", l1, "Calced", l2
                    length += l2
            if length - len(final) != 0:
                print 'expected - returned SHOULD BE ZERO:', (length-
len(final))
            fl()

        if str(final) == '':
```

```
            print 'BREAK BREAK BREAK'
        return final


    def get_transcript_position(self,coord):
        dist = 0 # 1-indexed
        if self.strand == '+':
            self.regions.sort(key = lambda x: x[1][0])
        else:
            self.regions.sort(key = lambda x: -x[1][0])
        for item in self.regions:
            if item[0] == 'CDS':
                if self.is_in_range(item[1],coord):
                    if self.strand == '-':
                        dist += abs(coord - item[1][1]) + 1
                    else:
                        dist += abs(coord - item[1][0]) + 1
                    break
                else:
                    if self.strand == '-':
                        pass
                    dist += (1 + abs(item[1][0] - item[1][1]))
        return dist

    def test_transcript_position(self,coord,chr):
        internal_pos = self.get_transcript_position(coord)
        internal_seq = self.get_coding_sequence(chr)
        used =  str(internal_seq[internal_pos-5:internal_pos+5].seq)
        external_seq = chr[coord-4 : coord+6]
        if self.strand == '-':
            external_seq = external_seq.reverse_complement()
        if str(external_seq.seq) != used:
            print 'not equal'
            print used
            print str(external_seq.seq)
            print coord
            print internal_pos
            j = []
            start = 10000000000
            print self.regions
            for item in self.regions:
                for this in item[1]:
                    if int(this) < start:
                        start = int(this)
            for item in self.regions:
                j.append([])
                j[-1].append(int(item[1][0]) - start)
                j[-1].append(int(item[1][1]) - start)
            for item in self.regions:
                for this in item[1]:
```

```
                    if abs(int(this)-coord)<10:
                        print abs(int(this)-coord), "YES"
            print j
            print ''

def make_index_from_gff(gff_loc):
    '''Argument is file handle to gff file. Returns dict-of-list-of-
lists:

DICT 1: chromosome, LIST 3 (!)

LIST 2: [lower, upper limits ] , LIST 3

LIST 3: ORFs # [start, end] , [ region 1 : start,end,kind ] , [ region
2 : start,end,kind ] ... '''
    try:
        feat_file = open(gff_loc+'.pk1','r')
        feature_dict = cpickle.load(feat_file)
        feat_file.close()

    except ValueError:
        feature_dict = {}
        no_ORF_yet = [] # store lines in case we have to do
categorization a second time

        # separate the file by lines and tabs
        gff_loc = open(gff_loc,'r')
        file_contents = [ line.split('\t') for line in
gff_loc.readlines() if not line.startswith('#') ]
        for item in range(0,len(file_contents)):
            file_contents[item][0] =
gff.sensibleChroms(file_contents[item][0])
        # line[2] has feature type: chromosome / contig [not helpful],
ORF ['parent'], three_prime_UTR, five_prime_UTR, CDS, intron
        # catch each alternative
        count = 0

        for line in file_contents:

            # update on progress
            count += 1
            if count % 10000 == 0:
                print count
                # flush
                fl()

            #do actual work
            if line[2] in ['chromosome','contig']:
                # this makes it work ONLY BECAUSE chroms/contigs are
at start of file. if this changes, need to change following code.
```

```
                if gff.sensibleChroms(line[0]) not in feature_dict:
                    feature_dict[gff.sensibleChroms(line[0])] = []

            elif line[2] == 'ORF':
                name = gff.getTokenFromInformation(line[8],'Name')
                id = gff.getTokenFromInformation(line[8],'ID')
                note =
urllib.unquote(gff.getTokenFromInformation(line[8],'Note'))
                if id and (id != name):
                    name = id
                if not note:
                    note = ''
                this_ORF =
ORF(gff.getTokenFromInformation(line[8],'ID'),int(line[3]),int(line[4]
),line[6],name=name,note=note) # name, coord1, coord3, strand
                if gff.sensibleChroms(line[0]) in feature_dict:

feature_dict[gff.sensibleChroms(line[0])].append(this_ORF)
                else:
                    feature_dict[gff.sensibleChroms(line[0])] =
[ this_ORF ]

            elif line[2] in ['CDS','intron']:
                for item in
range(len(feature_dict[gff.sensibleChroms(line[0])])-1,-1,-1):
                    if ( gff.getTokenFromInformation(line[8],'Parent')
== feature_dict[gff.sensibleChroms(line[0])][item].name ):
                        feature_dict[gff.sensibleChroms(line[0])]
[item].add_region(line[2],line[4],line[3])
                        break

            elif line[2][-3:] == 'UTR':
                coords = [ int(c) for c in line[3:4] ]
                for item in
range(len(feature_dict[gff.sensibleChroms(line[0])])-1,-1,-1):
                    span = feature_dict[gff.sensibleChroms(line[0])]
[item].coords()
                    if max(coords) + 1 == span[0]:
                            feature_dict[gff.sensibleChroms(line[0])]
[item].add_UTR(line[2],line[4],line[3])
                            break
                    elif min(coords) - 1 == span[1]:
                            feature_dict[gff.sensibleChroms(line[0])]
[item].add_UTR(line[2],line[4],line[3])
                            break

        outfile = open(gff_loc.name+'.pk1','wb')
        cpickle.dump(feature_dict,outfile,-1)
        outfile.close()
```

```
    return feature_dict

def get_chroms(file):
    '''load chromosomes from fasta file called file
Returns list of SeqRecords'''
    chroms = []
    fasta = seqio.parse(file,'fasta')
    while True:
        try:
            chroms.append(fasta.next())
        except StopIteration:
            break
    for chrom in range(0,len(chroms)):
        chroms[chrom].name = gff.sensibleChroms(chroms[chrom].name)
        chroms[chrom].description =
gff.sensibleChroms(chroms[chrom].description)
        chroms[chrom].id = gff.sensibleChroms(chroms[chrom].id)

    return chroms

def get_particular_chrom(chroms,id):
    ''' finds the chromosome with (sanitized) name ID, returns it'''
    for chr in chroms:
        if gff.sensibleChroms(chr.id) == str(id):
            return chr

def findref(codon,ref,bloc=0):
    loc = str(codon).find(ref)
    if loc != -1:
        print loc+bloc
        fl()
        findref(codon[loc+1:],ref,bloc=loc)
    else:
        return

def is_synonymous(coord,chrom,ref,alt,orf_dictionary,debug=False):
    '''arguments: coordinate, chromosome, reference,
            alternate nucleotide.
       Returns [ref_aa,alt_aa,nonsynonymous/synonymous/splice/intron/
3-UTR/5-UTR].
    '''

    # get chromosome of interest
    chrom.id = gff.sensibleChroms(chrom.id,show=True)

    for item in orf_dictionary[chrom.id]: # lookup by chromosome
        coords = item.get_full_length()
        coords.sort()
        is_intergenic = True
```

```
        if coord > coords[0] and coord < coords[1]:
            is_intergenic = False
            # test it
            if debug:
                if item.strand == '-':
                    ref = str(seq.Seq(ref).complement())
                    alt = str(seq.Seq(alt).complement())
            if item.is_exon(coord):
                trans = item.get_coding_sequence(chrom,debug=debug)
                pos = item.get_transcript_position(coord)
                loc = pos - 1
                codon = trans[loc - (loc%3): loc + 3-loc%3]
                adjcodon = codon.tomutable()
                try:
                    adjcodon[loc%3] = alt
                    adjcodon = adjcodon.toseq()
                except:
                    sys.stdout.flush()
                    time.sleep(1)
                if str(codon)[loc%3] != ref:
                    # diagnostics -- should never need to be displayed
                    print 'STRAND ', item.strand
                    print 'CODON', str(codon)
                    print 'REF', ref, 'ALT', alt
                    print 'LOC%3',loc%3
                    print 'REGIONS',item.regions
                    print 'COORD' ,coord
                    print 'CHROM ID',chrom.id
                    for location in range(0,len(trans[loc-15:loc
+15])):
                        if str(trans[location]) == ref:
                            print location, 'YES',
str(trans[location])
                        else:
                            print location, 'NO', str(trans[location])
                #check synonymous
                try:
                    refAA = str(codon.translate())
                    altAA = str(adjcodon.translate())
                except:
                    print 'TRUNCATED CODING REGION\n'
                    print 'STRAND ', item.strand
                    print 'CODON', str(codon)
                    print 'REF', ref, 'ALT', alt
                    print 'LOC%3',loc%3
                    print 'REGIONS',item.regions
                    print 'COORD' ,coord
                    print 'CHROM ID',chrom.id

                #[refAA,altAA] =
```

```
[SeqUtils.seq3(refAA),SeqUtils.seq3(altAA)]
                if refAA == altAA:
                    return
[refAA,altAA,'SILENT',item.id,item.name,item.note]
                else:
                     return
[refAA,altAA,'NONSYN',item.id,item.name,item.note]
            # it's in it and coding
            elif item.is_intron(coord):
                intron = item.is_intron(coord)
                if abs(coord - intron[1][0]) < 5 or abs(coord -
intron[1][1]) < 5:
                    return ['-','-',
'SPLICE',item.id,item.name,item.note]
                else:
                    return ['-','-',
'INTRON',item.id,item.name,item.note]
            elif item.is_UTR(coord):
                return
['-','-',item.is_UTR(coord),item.id,item.name,item.note]
    return ['-','-','INTGEN','','','']

def run_diagnostic_test(chroms,orfs):
    random.seed()
    # choose random positions, and check if transcripts have the same
bases in those positions
    for ii in range(0,1):
        randchrom = random.choice(chroms)
        randpos = random.randint(1,len(randchrom)) # acting like GFF
file
        # handle as if in file
        j = is_synonymous(randpos,randchrom,randchrom[randpos -
1],'G',orfs,debug=True)
    print 'random tests complete'

    print is_synonymous(2367969,get_particular_chrom(chroms,
7),'C','A',orfs,debug=True)
    print is_synonymous(3089658,get_particular_chrom(chroms,
5),'G','T',orfs)
    print is_synonymous(3089631,get_particular_chrom(chroms,
5),'G','C',orfs)
    print is_synonymous(3092196,get_particular_chrom(chroms,
5),'G','T',orfs)
    print is_synonymous(4541552,get_particular_chrom(chroms,
8),'G','A',orfs)
    print is_synonymous(4544119,get_particular_chrom(chroms,
8),'G','T',orfs)
    print is_synonymous(4546694,get_particular_chrom(chroms,
8),'G','T',orfs)
```

```
def add_seq_to_dict(f,dict_of_seqs):
    counter = 0
    # this section: check every SNP and collect data
    sites_visited = []
    snp_set = set()
    discards = 0 # for non-interesting SNPs
    snp_contents = [] # will contain one item for every SNP -- see lab
book page 43
    # loop over all SNPs in a VCF file
    f_toread = open(f,'r')
    piece = f_toread.readline()
    while piece:
        if not piece.startswith('#'):
            item = piece.split('\t')
            counter += 1
            if counter % 1000 == 0: print counter
            # check whether the SNP is interesting
            # not actually debugging, but need this feature for GATK
output
            chrom =
get_particular_chrom(chroms,gff.sensibleChroms(item[0]))
            pos = int(item[1])
            ref = item[3]
            qual = float(item[5])
            alt = (item[4]).split(',')
            depth = (item[9]).split(':')[1].split(',')

            for each in range(0,len(alt)):
                nuc = alt[each]
                covr = [depth[0], depth[each+1] ]
                synon = is_synonymous(pos,
get_particular_chrom(chroms, gff.sensibleChroms(item[0])), ref, nuc,
orfs, debug=True)
                # good quality and interesting
                if qual > 50 and synon[2] != 'SILENT' and synon[2] !=
'INTGEN':
                    # make a record of the SNP

snp_contents.append([ [gff.sensibleChroms(item[0]), pos] , qual, covr,
synon[2], synon[3]])
                    sites_visited.append(pos) # faster to look up in a
list of ints -- look up in snp_contents iff coords match
                    SNP = gff.SNP([qual]+synon[0:3]+
[gff.sensibleChroms(item[0]),pos,ref,nuc]+covr+synon[4:])
                    snp_set.add(SNP)
        piece = f_toread.readline()
    dict_of_seqs[f[0:-4]] = [snp_contents,sites_visited,snp_set]


def sort_set(snp_set):
```

```
    l = list(snp_set)
    l.sort(key= lambda x: (int(x.disp[-4])/x.covr,-x.qual ))
    return l

if __name__ == "__main__":
    # We will eventually need to be able to check SNPs against the
versions present in each chromosome.
    # Here, we load that data.

    # sys.argv should be (1) reference sequence location, (2) feature
databse location, (3) save location

    chroms = get_chroms(sys.argv[1])
    orfs = make_index_from_gff(sys.argv[2])
    save_directory = sys.argv[3]

    # run unit tests to check for program integrity
    run_diagnostic_test(chroms,orfs)

    # 'files' gets a list of files in directory
    files = os.listdir('.')
        # remove all but the files we want
    count = 0
    while True:
        try:
            # if file is not a pickle or vcf file
            if not files[count].endswith('.pk') and \
                not files[count].endswith('snps.vcf'):
                # then disregard it
                files.pop(count)
            # if file is a vcf file
            elif files[count].endswith('snps.vcf'):
                # if file has corresponding pickle file:
                if files[count][:-3] +'.pk' in files:
                    # disregard the vcf
                    files.pop(count)
                else: # no pickle file, move on
                    count += 1
            else: # it's a pickle file, move on
                count += 1
        except IndexError:
            break

    files.sort()
    dict_of_seqs = {}
    #print files
    # files has '.pk' if it exists, otherwise '.vcf'

    # open pickle files we can find
    # for testing:
```

```
    for fn in range(0,len(files)):
        try:
            f=files[fn]
            init = open(save_directory+'/'+ f[:-3]+'.pk','rb')
            if f.find(WT) != -1:
                dict_of_seqs.update(cpickle.load(init))
                init.close()
            files[fn] = f[0:-3] + '.pk'

        except IOError: print 'No .pk for', f[0:-3] + '.pk'

    k = dict_of_seqs.keys()
    # k has '.pk'
    # write a pickle file for all the ones we missed
    for fn in range(0,len(files)):
        f=files[fn]
        if (f[0:-4] + '.pk') not in files and not f.endswith('.pk'):
            dummy = {}
            add_seq_to_dict(f,dummy)
            save_location = open(save_directory+'/'+f[:-4]
+'.pk','wb')
            cpickle.dump(dummy,save_location,-1)
            save_location.close()
            files[fn] = f[:-4]+'.pk'

    # do SNP subtraction and write out
    for sequence in dict_of_seqs.keys():
        rpa = sequence[0:-3]
        if sequence.find(M) != -1: # mutant
            print 'Processing', sequence
            # subtract everything
            subtr=open(save_directory+'/'+rpa+excess+'_subtr_all','w')

            for item in dict_of_seqs.keys():
                if item.find(WT) != -1:
                    dummy[sequence[0:-3]][2] = \
                        dummy[sequence[0:-3]][2] - dict_of_seqs[item]
[2]
            for item in sort_set(dummy[sequence[0:-3]][2]):
                subtr.write(''.join((str(item),'\n')))
            subtr.close()
        gc.collect()
```