**Mohamad Alayouni, Andrew Maddox and Monique Nichols**

**Practical Computer Concepts for Bioinformatics**

**Final Project**

**December 17th, 2023**

The Code

import re

**# Part 1: FASTA reading**

**# Written by Andrew Maddox**

#this function finds all headers and retrieves

#the sequences associated with them in a FASTA file. A list containing

#the uppercase sequence is stored in a dictionary with each header

#as keys, then the dictionary is returned. This relies on

#the helper function \_\_get\_file\_contents and \_\_validate\_file\_contents.

#Returns a dictionary or None if the user quits. Ensures

#the selected file is in FASTA format.

def get\_seq\_dict()->dict:

contents = \_\_get\_file\_contents()

seq\_dict = dict()

#Group 1 gives the header, group 2 gives the sequence in valid files

pattern = r"(>\S+.\*)\n([ATGCatgc\s]+)"

#check to see if user wants to quit, indicated by empty string

if contents == "":

return seq\_dict

if \_\_validate\_file\_contents(contents):

#file is valid

#this loop creates the dictionary with

#header keys and values of a list with the sequence

for m in re.finditer(pattern, contents):

header = m.group(1)

#Remove whitespace characters in sequences

seq = re.sub(r"\s", "", m.group(2))

seq = seq.upper()

seq\_dict.update({header:[seq]})

else:

#file is invalid and new file needs selection

get\_seq\_dict()

return seq\_dict

def \_\_validate\_file\_contents(contents : str)-> bool:

#Search pattern to check for invalid characters, g1-header, g3-invalid chars

validate\_char\_pattern = r"(>\S+.\*)\n([ATGCatgc\s]\*)([^ATGCatgc\s>]+)([ATGCatgc\s]\*) "

#Ensure the file has fasta headers, return None means it is invalid

validate\_headers\_pattern = r"(>\S+.\*)\n"

#check to see if selected file is a FASTA file by format

if re.search(validate\_headers\_pattern, contents) == None:

print("ERROR: Selected file does not have FASTA formatted headers.\n")

return False

#check for invalid characters in the sequence of the file

else:

m = re.search(validate\_char\_pattern, contents)

if(m is not None):

#there are invalid chars in a sequence

print("ERROR: Selected file contains invalid characters for a FASTA sequence.\n")

return False

else:

#reached if file contents are valid

return True

#this function takes in the user input to validate that

#the file selected by the user exists, then returns the

#contents of the file. Returns an empty string if the user quits.

def \_\_get\_file\_contents()->list:

#sentinel variable used to control the cycle of user input and validation

flag = 1

while flag == 1:

try:

#try to read the entered file name and return the contents

fname = input("Enter a FASTA file name: ")

file = open(fname, 'r')

contents = file.read()

file.close()

return contents

except OSError:

#notify user of error, ask them to proceed or quit, and validate their input

flag = input("ERROR: File is unreadable or not found. Enter 1 to continue or 0 to quit.")

while flag != '0' and flag != '1':

flag = input("ERROR: Invalid input. Enter 1 to continue or 0 to quit.")

if flag == '1':

return \_\_get\_file\_contents()

#only reached in the case where a user wants to quit the program

return ""

**# Part 2: ORF isolation**

**# Written by Monique Nichols**

# For use in ORF\_Finder to produce reverse complement of given sequence

def reverse\_compliment(sequence):

# Dictionary of base translations

base\_dict = {

'A':'T',

'T':'A',

'C':'G',

'G':'C'

}

counter = 0

Translated = '' #empty string to add translated bases to

while counter < len(sequence):

Translated += base\_dict[sequence[counter]] # translating sequence using dictionary base-by-base

counter += 1

ReverseCompliment = '' # empty string to append reversed bases to

for base in Translated:

ReverseCompliment = base + ReverseCompliment # append each new base to the left of the previous - reversed!

return ReverseCompliment

# For use in ORF\_Finder, splits sequence into in-frame codons and checks to make sure none are premature STOP codons

def STOP\_Check(found\_orf):

counter = 0

while counter <len(found\_orf)-3: # Don't want to include actual STOP codon in the True/False check below, so -3

substr = found\_orf[counter:counter+3] # splitting into codon and checking against STOPs

if substr == 'TAA' or substr == 'TAG' or substr == 'TGA':

return False # If False passed to ORF\_Finder, ORF will not be included in final output

counter += 3

return True

def ORF\_Finder(FASTA\_dict):

valid = False

min\_length = ''

while not valid:

# try-except to make sure the user-entered minimum bp value is valid. Must be greater than 49 and an integer

try:

min\_length = input('What is the desired minimum ORF length? Press Enter to default to 50bp: ')

min\_length = int(min\_length)

if min\_length < 50:

print('ERROR: Minimum value is 50, please try again.')

valid = False

else:

valid = True

# if input was not a number:

except ValueError:

if min\_length == '': #Default = 50

min\_length = 50

valid = True

else:

print('ERROR: Non-integer value entered, please try again.')

valid = False

return\_list = [] # blank list that will be filled and returned

for key in FASTA\_dict:

fwd\_seq = FASTA\_dict.get(str(key)) # forward sequence is the one in the input dictionary

rev\_seq = reverse\_compliment(fwd\_seq[0]) # reverse complement function above

FASTA\_dict[key].extend([rev\_seq]) # add the reverse complement to the same key, so the header is still connected

seq\_type = 0 # bool to determine if we are working with the forward or reverse strand (1st vs 2nd sequence in list)

for seq in FASTA\_dict.get(key):

seq\_starts = [m.start(0) for m in re.finditer('ATG', seq)] # find all START match objects

for start in seq\_starts: # querying each start

sub\_seq = seq[start:] # getting only sequence from start codon forward

stop\_codons = re.compile(('TAA|TAG|TGA')) #creating a regex search that accepts any of the three stop codons

seq\_ends = [m.start(0) for m in stop\_codons.finditer(sub\_seq)] # list of match object STOP positions

for end in seq\_ends: #for each STOP codon match, slice out string from the start codon to the stop

temp\_ORF = sub\_seq[:end+3]

if seq\_type == 0: #Forward strand, frames 1-3 (see seq\_type = 0 statement above)

# All found start-stop sequences screened against user input minimum ORF length,

# the STOP\_Check function which detects premature stop codons within frame

# and the stop codon is ensured to be within frame of start codon

if len(temp\_ORF) % 3 == 0 and len(temp\_ORF) >= min\_length and STOP\_Check(temp\_ORF):

if len(seq[:start]) % 3 == 0: #FRAME = 1

# append list of valid ORF characteristics to return\_list for output function

# (start + 1) accounts for Python counting from 0

return\_list.append([key,temp\_ORF,'FRAME = 1','POS = '+str(start+1),'LEN = '+str(len(temp\_ORF))])

elif len(seq[:start]) % 3 == 1: #FRAME = 2

return\_list.append([key,temp\_ORF,'FRAME = 2','POS = '+str(start+1),'LEN = '+str(len(temp\_ORF))])

elif len(seq[:start]) % 3 == 2: #FRAME = 3

return\_list.append([key,temp\_ORF,'FRAME = 3','POS = '+str(start+1),'LEN = '+str(len(temp\_ORF))])

# second sequence in FASTA\_dict.get(key) is always the reverse strand sequence (seq\_type != 0)

else:

# if the seq is divisible by 3, FRAME = 1. (ATG start)

if len(temp\_ORF) % 3 == 0 and len(temp\_ORF) >= min\_length and STOP\_Check(temp\_ORF):

if len(seq[:start]) % 3 == 0: #FRAME = 4

return\_list.append([key,temp\_ORF,'FRAME = 4','POS = '+str(0-start-1),'LEN = '+str(len(temp\_ORF))])

elif len(seq[:start]) % 3 == 1: #FRAME = 5

return\_list.append([key,temp\_ORF,'FRAME = 5','POS = '+str(0-start-1),'LEN = '+str(len(temp\_ORF))])

elif len(seq[:start]) % 3 == 2: #FRAME = 6

return\_list.append([key,temp\_ORF,'FRAME = 6','POS = '+str(0-start-1),'LEN = '+str(len(temp\_ORF))])

seq\_type += 1 # making sure we fail the if statement above that writes positions for forward strand

return return\_list

**# Part 3 : Formatting the output to a FASTA format**

**#Written by Mohamad Alayouni**

# Edited by Andrew

# Defining the ‘print\_orfs’ function which considers the three parameters header, orfs, and sequence

def print\_orfs(nested\_list : list):

# Obtains a file name for the output file from the user

valid = False

output\_file\_name = ""

while not valid:

output\_file\_name = input("Please enter a file name for output: ")

# Check for .txt extension, add it if not present in file name

if output\_file\_name[len(output\_file\_name)-4:] != ".txt":

output\_file\_name = output\_file\_name + ".txt"

# Check if file already exists

# Try block concludes if the file already exists

try:

f = open(output\_file\_name, 'r')

f.close()

valid = False

print(f"ERROR: The file {output\_file\_name} already exists. Please try again.")

# Reached if selected file name does not already exist

except:

valid = True

# Open file for writing

outfile = open(output\_file\_name, 'w', encoding = "UTF-8")

# Extract info for each orf, format them, print them to the file

for orf in nested\_list:

accession = orf[0]

orf\_sequence = orf[1]

frame = orf[2]

pos = orf[3]

length = orf[4]

# Creates the final formatted header for an orf entry

formatted\_header = f"{accession} | {frame} {pos} {length}"

# Uses regex to split sequence into a list of 3 char chunks

codons = re.findall("...", orf\_sequence)

# Counts the number of codons in a line

count = 0

orf = ""

# Print the codons to the output file, 15 codons per line

for i in range(len(codons)):

orf = orf + codons[i] + " "

count += 1

if count == 15:

orf = orf.rstrip() + "\n"

count = 0

outfile.write(formatted\_header + "\n" + orf.rstrip() +"\n")

# Close file after all writing is completed

outfile.close()

# Main function that ties all functions together for the gestalt function of isolating

# orfs from a file and printing them to another file.

def main():

fasta\_dict = get\_seq\_dict()

orf\_list = ORF\_Finder(fasta\_dict)

print\_orfs(orf\_list)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Work Summary

**Solving the Problem:**

Mohamad: I first began by coding the whole program and tested it out multiple times until it worked. Then I realized it can be split into three different sections, and the section I chose to enhance and simplify was the coding section that is involved with formatting how the output will be displayed. I decided to use the for loop to iterate over each ORF identified, and then made the frame, pos, and length represent the frame number, the starting position, and the length of each ORF, respectively. I used the if-else statement to check for the frame number, so that if the frame is less than or equals to 3 it would indicate that the ORF is on the template strand and print a positive value, but if it was more than 3 it would indicate that the ORF began in the reverse strand and show a negative value. Print statements were used to display information about the ORF. I used the count variable to keep track of the number of codons printed per line. In my code I initialized the variables ‘valid’ to ‘false’ and an empty string ‘output\_file\_name,’ the code then enters a ‘while’ loop that continues until the ‘valid’ variable becomes ‘true,’ this was used to ensure that when the program prompts the user to enter a name for the output file, the name entered is not a file name that already exists.

Andrew: I had read the project before doing Discussion 13, so I wrote my answer to Discussion 13 with the intention of reusing portions of that code in the project. I improved upon it with some more error handling and file validation. It seemed to me that the best way to transfer the sequence information between our modules was to use a dictionary with keys and values which would be simple and meaningful to the next person who used it. I decided that the dictionary should be composed of keys made from the sequence headers because it gives an easy way to organize them with ‘primary keys’ and the values should be the sequences extracted from the input files. I made sure to remove any whitespace in the sequences and force them to uppercase to avoid any inconsistencies they may cause. Like my groupmates, I either coded other parts of the project or outlined sections other than my own to evaluate how my portion would interact with the others and plan it better.

Monique: I first read through the instructions, bulleted-out what was being asked and what steps would need to be taken to do what was asked and then wrote down the logic: I knew the input would be a dictionary, so I wrote through how I would loop through getting each key (FASTA header) and value (sequence). I then would need to use regex to find all instances of ATG in each sequence and every instance of stop codon that followed each start codon identified. I would slice out the sequence from start-stop, make sure it met the minimum length criteria and that there were no premature stop codons, then determine if the frame is valid and use remainder math to determine the frame type. Since I had a lot of position information available to me from the regex and slicing I figured I would also determine the position and length of each ORF; to keep all of the information together I decided to output a list of lists. After I had the workflow thought out and coded for the forward strand, I wrote a function to create each reverse strand from the full input sequence and added it as a second value to each key so that I could go through the same logic with the reverse strand sequence. Since there are 6 different frames, there are 6 conditional statements.

**Division of Work:**

After some back and forth between how we can collaborate as a team on the code and make our individual contributions, it came to our attention that the code could be split into three different parts. Part 1 deals with reading the input file, parsing the FASTA sequence, and then storing them in a suitable structure. Part 2 involves writing the functions to find the open reading frame for each of the 6 different reading frames. Lastly, part 3 involves writing the functions for formatting the output.

**Obstacles:**

Mohamad: I think one of the biggest obstacles so far is communication, since canvas does not notify me whenever someone adds a message to the group discussion board. This became particularly obvious when I saw that Andrew brought up the idea to start this final project on Nov 5th, but I did not see his message until the beginning of this week

Furthermore, since I approached writing my code by first writing the whole code for this project then splitting it into three sections, part of my code which is the third and last section has stubs which includes some of the code which was supposed to be from part 1 and part 2 of the code.

Andrew: As Mohamad stated, the communication took a little bit to become established, but it was consistent once it started. My biggest issue was thinking about all the different errors I would need to handle and the best way to go about doing so. Generally, the biggest obstacle is creating a unified program with three different authors, though. We each have an idea of what we want to do for our sections and we have to think ahead to make unification easier. Something else I personally struggled with was overstepping and trying to take on more of the project. I also ran into a weird bug in Notepad where some of the output files I produced while testing were somehow encoded as Chinese unicode characters in Notepad. I was able to properly view the files in Notepad++ and I confirmed that the files were printing properly by printing to the console as well.

Monique: My biggest obstacle was a misunderstanding of what the ORF frames meant. I understood in principle but coded the whole project thinking that the frames were split up starting from ATG due to being misled by an online resource; I was getting many more output ORFs than the sample output which was frustrating until I learned that the frame started from the beginning of the sequence. After that things were smooth sailing. As Mohamed mentioned, the Canvas Group tab was difficult to navigate at first due to lack of notifications which hindered the time we ended up having to talk about the project since we both missed Andrew’s initial message.

**Future Improvements:**

Mohamad: In part 3 of the code which was focused on the format of the output and prompting the used to input the minimum number of bases they desire to be in their ORF, I think one of the improvements that could be made is to make the separate output ORF’s generated by 2-3 blank line spaces so that the viewer can clearly distinguish one generated ORF from the other. Another improvement that could help make the code clearer, is to replace ATG with ‘Start’ and the three stop codons with ‘stop.’ Another possible improvement which could help differentiate one codon from another easily would be to color each codon differently. Furthermore, the output could be provided in a way that the longest ORF’s found would be at the top and would go down in a descending order to the shortest at the bottom.

Andrew: I think it would be interesting to set up the program such that select sequences can be selected for ORF finding from the input file or it can be processed as a batch. This would not change part 2 and 3, but there could be some user selection in part 1 to select the sequences of interest and pass them along in a dictionary for processing. Using a GUI for file selection would also be very helpful to reduce the possibility of errors stemming from mistyping a file name. Selecting multiple files for ORF finding would also be feasible to accomplish without changing parts 2 and 3 while making minimal changes to the input process in part 1.

Monique: I agree with what my teammates have said, especially on color-coding ORF attributes and creating a GUI. I think there is room for more error-handling with user input; it may not be very apparent to us now but with more scrutiny and use of the code we could definitely find ways to break it. I also think it is possible to make the ORF\_Finder function more concise by creating other functions to do some of the repetitive information gathering for each of the 6 frames. Finally, depending on what the end user wants to do with this information I think we could make the output file more user-friendly by making it into a well-formatted Excel sheet; if there is a large output and they want to sort/filter/highlight entries, this might be useful.