**Test file to start with:**

aaAaa b

op

Documentation of the code as it develops:

Started with calculating character stream without map/reduce (to get a grasp on the logic the code would need)

import time  
import math  
  
  
def count\_total\_characters(a\_character\_list):  
 return len(a\_character\_list)  
  
  
def build\_character\_list(a\_file):  
 the\_list = []  
 for every\_line in a\_file: # breaks it up by line  
 for every\_character in every\_line: # counts newline '\n' character as well. minus those is 229  
 the\_list.append(every\_character.capitalize()) # don't want to recount upper/lower case  
  
 return the\_list  
  
  
def build\_unique\_character\_list(character\_list):  
 uniques\_found = []  
 for some\_character in character\_list:  
 if some\_character in uniques\_found:  
 continue  
 else:  
 uniques\_found.append(some\_character)  
  
 return uniques\_found  
  
  
def count\_uniques(a\_character\_list, uniques\_list): # can use microsoft word find function to check accuracy  
 count = 0  
 linked\_dictionary = {}  
 for someUnique in uniques\_list:  
 for aCharacter in a\_character\_list:  
 if aCharacter == someUnique:  
 count += 1  
 # go through entire character list  
 # print("Total found for " + str(someUnique) + ": " + str(count))  
 linked\_dictionary[someUnique] = count  
 count = 0  
  
 return linked\_dictionary  
  
  
def find\_probabilities(a\_unique\_dictionary, total\_characters):  
 the\_probabilities = {}  
 for each\_key in a\_unique\_dictionary:  
 amount\_of\_each\_key\_found = a\_unique\_dictionary[each\_key]  
 probability\_of\_each\_key = amount\_of\_each\_key\_found / total\_characters  
 probability\_of\_each\_key = round(probability\_of\_each\_key, 4)  
 the\_probabilities[each\_key] = probability\_of\_each\_key  
 return the\_probabilities  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 myTextFile = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile.txt", "r")  
  
 characterList = build\_character\_list(myTextFile)  
 # print("Total lines found: " + str(lineCount)) # total: 4  
 # print("Total characters found: " + str(characterCount)) # total: 233  
 # print(len(characterList)) # returns 233  
 TotalCharacters = count\_total\_characters(characterList)  
 print(TotalCharacters)  
  
 uniqueCharacters = build\_unique\_character\_list(characterList)  
 print(len(uniqueCharacters)) # returns 36  
 # print("Total unique characters found: " + str(len(uniqueCharacters))) # total: 36 (39 if not capitalized)  
 myDictionary = count\_uniques(characterList, uniqueCharacters)  
 print(myDictionary) # show whole dictionary  
 myProbabilities = find\_probabilities(myDictionary, TotalCharacters)  
 print(myProbabilities)  
  
 sumOfTheInformation = 0  
 for eachUniqueCharacterFound in myDictionary:  
 numberOfTimesItOccurs = myDictionary[eachUniqueCharacterFound]  
 if eachUniqueCharacterFound in myProbabilities:  
 probabilityOfTheCharacter = myProbabilities[eachUniqueCharacterFound]  
 informationForThisCharacter = round((numberOfTimesItOccurs \* (-probabilityOfTheCharacter) \*  
 math.log(probabilityOfTheCharacter, 2)), 6)  
 sumOfTheInformation += informationForThisCharacter  
  
 sumOfTheInformation = round(sumOfTheInformation, 6) # this value will reduce with groups of characters  
 print("TOTAL SUM OF INFORMATION: " + str(sumOfTheInformation))  
  
 runTime = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(runTime) + " nanoseconds")  
 print(str(runTime / 1000) + " microseconds (μs)")  
 print(str(runTime / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(runTime / 1000000000, 2)) + " seconds (rounded)")

Now will start analyzing for a map/reduce process…

MAPPER maps a value to another value

REDUCER takes a list of values and returns a single value

**Begin by counting the occurrences of each character using Map/Reduce**

REDUCER takes a list of values and returns a single value

\*\*attempt 1: count map/reduce

import time  
from functools import reduce  
  
lenMapper = len  
  
  
def len\_reducer(p, c):  
 return p+c  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 myTextFile = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt", "r")  
 fileString = myTextFile.read()  
 lengthOfEachCharacter = map(lenMapper, fileString) # just 1 per character  
  
 reduced = reduce(len\_reducer, lengthOfEachCharacter)  
  
 print(reduced)  
  
 runTime = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(runTime) + " nanoseconds")  
 print(str(runTime / 1000) + " microseconds (μs)")  
 print(str(runTime / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(runTime / 1000000000, 2)) + " seconds (rounded)")

\*\* attempt 2: using lambda expressions

import time  
from functools import reduce  
'''  
REQUIREMENTS FOR: COUNTING THE OCCURRENCES OF EACH CHARACTER  
\* know what each character is (in the file) (MAY NOT BE NECESSARY)  
\* be able to return a count of each of those characters (REDUCE SHOULD DO THIS; it returns a single value)  
\*\* start by being able to return the count of a chosen character  
# example: how many a's are in my file? REDUCE should return this  
\*\* now be able to return the count of ANY character  
# example: how many 'INPUT\_VALUE' are in my file?  
  
obj\_total = sum(obj.count for obj in list\_all\_objects())  
  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope;   
\* PURE functional programs only depend on their input (no internal state)  
'''  
  
# lambda count  
isLetterA = lambda a: a == 'a' or a == 'A'  
# lambda check  
isTrue = lambda a: a == True  
  
# lambda equivalent  
isEquivalent = lambda a, b: a == b  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
 FILE\_STRING = FILE\_STRING # use upper() or lower() to help in count  
 print(FILE\_STRING)  
 myLetterIterable = map(isLetterA, FILE\_STRING)  
 myLetterTuple = tuple(myLetterIterable)  
 print(myLetterTuple)  
  
 myTrueFilterIterator = filter(isTrue, myLetterTuple)  
 myTrueFilterTuple = tuple(myTrueFilterIterator)  
 NUMBER\_OF\_A = len(myTrueFilterTuple)  
 print(NUMBER\_OF\_A)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

\*\* attempt 3: using a generator expression and dictionary expression

import time  
from functools import reduce  
'''  
REQUIREMENTS FOR: COUNTING THE OCCURRENCES OF EACH CHARACTER  
\* know what each character is (in the file) (MAY NOT BE NECESSARY)  
\* be able to return a count of each of those characters (REDUCE SHOULD DO THIS; it returns a single value)  
\*\* start by being able to return the count of a chosen character  
# example: how many a's are in my file? REDUCE should return this  
\*\* now be able to return the count of ANY character  
# example: how many 'INPUT\_VALUE' are in my file?  
  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope;   
\* PURE functional programs only depend on their input (no internal state)  
'''  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
 myFileIterator = iter(FILE\_STRING)  
  
 # my generator expressions  
 CHARACTER\_COUNTER = (FILE\_STRING.count(x) for x in FILE\_STRING)  
 CHARACTER\_PRINTER = (x for x in FILE\_STRING)  
 print(tuple(CHARACTER\_COUNTER)) # returns a tuple of the counts of EVERY character -> shows duplicates  
 print(tuple(CHARACTER\_PRINTER)) # returns EVERY character -> shows duplicates  
 # my dictionary expression  
 MY\_DICTIONARY = {(x, FILE\_STRING.count(x)) for x in FILE\_STRING}  
 print(MY\_DICTIONARY) # returns EACH character -> no duplicates!  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

we’re finding the probabilities pc of each character

import time  
from functools import reduce  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope;   
\* PURE functional programs only depend on their input (no internal state)  
'''  
# lambda sum  
myTotal = lambda x, y: x + y  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
 myFileIterator = iter(FILE\_STRING)  
  
 # my generator expressions  
 CHARACTER\_COUNTER = (FILE\_STRING.count(x) for x in FILE\_STRING)  
 CHARACTER\_PRINTER = (x for x in FILE\_STRING)  
 # print(tuple(CHARACTER\_COUNTER)) # returns a tuple of the counts of EVERY character -> shows duplicates  
 # print(tuple(CHARACTER\_PRINTER)) # returns EVERY character -> shows duplicates  
 # my dictionary expression  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS = dict([(x, FILE\_STRING.count(x)) for x in FILE\_STRING])  
 print(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS) # returns EACH character -> no duplicates - ONLY UNIQUES!!  
 NUMBER\_OF\_UNIQUES = len(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
 print(NUMBER\_OF\_UNIQUES)  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = (MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS[the\_counts] for the\_counts  
 in MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
   
 MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 print(MY\_TOTAL\_CHARACTERS)  
 MY\_PROBABILITY\_OF\_CHARACTER\_FINDER = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 (MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS[x] for x  
 in MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS))  
 print(tuple(MY\_PROBABILITY\_OF\_CHARACTER\_FINDER))  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

now we’re finding the INFORMATION OF EACH CHAR

import time  
import math  
from functools import reduce  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope;   
\* PURE functional programs only depend on their input (no internal state)  
'''  
# lambda sum  
myTotal = lambda x, y: x + y  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
 myFileIterator = iter(FILE\_STRING)  
  
 # my generator expressions  
 CHARACTER\_COUNTER = (FILE\_STRING.count(x) for x in FILE\_STRING)  
 CHARACTER\_PRINTER = (x for x in FILE\_STRING)  
 # print(tuple(CHARACTER\_COUNTER)) # returns a tuple of the counts of EVERY character -> shows duplicates  
 # print(tuple(CHARACTER\_PRINTER)) # returns EVERY character -> shows duplicates  
 # my dictionary expression  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS = dict([(x, FILE\_STRING.count(x)) for x in FILE\_STRING])  
 print(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS) # returns EACH character -> no duplicates - ONLY UNIQUES!!  
 NUMBER\_OF\_UNIQUES = len(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
 print(NUMBER\_OF\_UNIQUES)  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS.get(x)),  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
  
 MY\_TOTAL\_CHARACTERS = FILE\_STRING.\_\_len\_\_()  
  
 MY\_PROBABILITY\_OF\_CHARACTER\_FINDER = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 # iterator is spent, need to remake  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS.get(x)),  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITY\_OF\_CHARACTER\_FINDER)  
 print(MY\_INFORMATION\_OF\_EACH\_CHARACTER.\_\_next\_\_())  
 '''  
 print(tuple(MY\_PROBABILITY\_OF\_CHARACTER\_FINDER))  
 '''  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

NOW FIND TOTAL INFORMATION WITH REDUCE

import math  
import time  
import math  
from functools import reduce  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
# lambda sum for reduce  
myTotal = lambda x, y: x + y  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
  
 # my generator expressions  
 CHARACTER\_COUNTER = (FILE\_STRING.count(x) for x in FILE\_STRING)  
 CHARACTER\_PRINTER = (x for x in FILE\_STRING)  
 # print(tuple(CHARACTER\_COUNTER)) # returns a tuple of the counts of EVERY character -> shows duplicates  
 # print(tuple(CHARACTER\_PRINTER)) # returns EVERY character -> shows duplicates  
 # my dictionary expression  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS = dict([(x, FILE\_STRING.count(x)) for x in FILE\_STRING])  
 # print(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS) # returns EACH character -> no duplicates - ONLY UNIQUES!!  
 # NUMBER\_OF\_UNIQUES = len(MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
 # print(NUMBER\_OF\_UNIQUES)  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS.get(x)),  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
  
 MY\_TOTAL\_CHARACTERS = FILE\_STRING.\_\_len\_\_()  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # PROBABILITY DISTRIBUTION  
 # iterator is spent, need to remake  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS.get(x)),  
 MY\_DICTIONARY\_COUNTS\_OF\_CHARACTERS)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 print(MY\_TOTAL\_INFORMATION\_IN\_STREAM) # TOTAL INFORMATION IN THE FILE  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

**NOW: TRY TO DIVIDE INTO pairs/triples of characters**

\*\*First: rework to eliminate dictionary (unnecessary)

import time  
import math  
from functools import reduce  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
# lambda sum for reduce  
myTotal = lambda x, y: x + y  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns()  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
 # use SET to grab uniques  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING)  
 # CONSIDER FILE\_STRING.UPPER() OR FILE\_STRING.LOWER() to reduce load  
 # print(MY\_UNIQUES\_TUPLE) # CAN CHECK SET OF ALL UNIQUE CHARS HERE IF YOU WANT  
  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS))  
 MY\_TOTAL\_CHARACTERS = FILE\_STRING.\_\_len\_\_()  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # PROBABILITY DISTRIBUTION  
  
 # iterator is spent to build probabilities iterator, need to remake  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 print(MY\_TOTAL\_INFORMATION\_IN\_STREAM) # TOTAL INFORMATION IN THE FILE  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # end time - start time to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")

\*\* NEXT IS THE DIVISIONS AND PROBABILITIES

import time  
import math  
from functools import reduce  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
  
# lambda sum for reduce --------------------------------  
myTotal = lambda x, y: x + y  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------  
  
 # use SET to grab uniques  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING) # USED FOR SINGLES  
 # need to pass PAIRS and TRIPLES to count new values for NC -- instead of count char, count pairs  
 # [line[i:i+n] for i in range(0, len(line), n)] where n = 2  
 MY\_PAIRS\_TUPLE = set(FILE\_STRING[i: i+2] for i in range(0, len(FILE\_STRING), 2)) # USED FOR PAIRS  
 MY\_TRIPLES\_TUPLE = set(FILE\_STRING[i: i+3] for i in range(0, len(FILE\_STRING), 3)) # USED FOR TRIPLES  
 # CONSIDER FILE\_STRING.UPPER() OR FILE\_STRING.LOWER() to reduce load  
 # print(MY\_UNIQUES\_TUPLE) # CAN CHECK SET OF ALL UNIQUE CHARS HERE IF YOU WANT  
  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE) # COUNTS SINGLES  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS))  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS))  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS))  
 MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR  
 print("TOTAL NUMBER OF CHARACTERS: " + str(MY\_TOTAL\_CHARACTERS) + " (used to solve for pc)")  
  
 # iterator is spent to build MY\_TOTAL\_CHARACTERS, need to remake  
 # reset iterator  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR AGAIN  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = map(lambda x: round(x/MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # PROBABILITY DISTRIBUTIONS  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)) + " (single characters)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_PAIR)) + " (pairs)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)) + " (triples)")  
  
 # reset BOTH iterators FOR ALL  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT AGAIN  
  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_INFORMATION\_OF\_EACH\_PAIR = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR)  
 MY\_INFORMATION\_OF\_EACH\_TRIPLE = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)  
  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 MY\_TOTAL\_INFORMATION\_PAIRS = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_PAIR), 3)  
 MY\_TOTAL\_INFORMATION\_TRIPLES = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_TRIPLE), 3)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # END TIME - START TIME to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")  
  
 # TOTAL INFORMATION IN THE FILE  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_IN\_STREAM) + " (SINGLES)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_PAIRS) + " (PAIRS)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_TRIPLES) + " (TRIPLES)")

**total information in the stream does indeed go down**

TIME FOR CONCURRENCY!!!!!!!!

\*\*\*\*\*\*\*\*\*FIRST DOCUMENT THE INFO **WITHOUT \*\*\*\*\*\*CONCURRENCY**

**INFORMATION IN DATA STREAM**: 4.107 **(SINGLES)**

**INFORMATION IN DATA STREAM**: 2.257 **(PAIRS)**

**INFORMATION IN DATA STREAM**: 1.328 **(TRIPLES)**

TOTAL NUMBER OF CHARACTERS: 10 (used to solve for pc)

PROBABILITY DISTRIBUTION: (0.1, 0.4, 0.1, 0.1, 0.1, 0.1, 0.1) (single characters)

PROBABILITY DISTRIBUTION: (0.1, 0.1, 0.1, 0.2, 0.1) (pairs)

PROBABILITY DISTRIBUTION: (0.1, 0.1, 0.1, 0.1) (triples)

\*\*\* going to run **3 TIMES** for an average on the runtime

**RUN 1:**

374626 nanoseconds

374.626 microseconds (μs)

0.000374626 seconds

0.0 seconds (rounded)

**RUN 2:**

254027 nanoseconds

254.027 microseconds (μs)

0.000254027 seconds

0.0 seconds (rounded)

**RUN 3:**

245046 nanoseconds

245.046 microseconds (μs)

0.000245046 seconds

0.0 seconds (rounded)

**AVERAGE TIME:**

374626 nanoseconds + 254027 nanoseconds + 245046 nanoseconds = 873699 / 3

**= 291233 nanoseconds**

**= 291.233 microseconds (μs)**

**= 0.000291233 seconds**

\*\*\*\*\* DOCUMENT THE INFO **WITH**

**\*\*\*\*\*\*CONCURRENCY**

STARTING WITH 2 THREADS

import time  
import math  
from functools import reduce  
from multiprocessing import Pool # THIS IS FOR CONCURRENCY ------------------  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
  
# lambda sum for reduce --------------------------------  
myTotal = lambda x, y: x + y  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/testFile2.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------  
 MY\_WORKER\_POOL = Pool(2) # THIS IS FOR CONCURRENCY ------- NUMBER IS NUMBER OF THREADS  
 # USE MY\_WORKER\_POOL.MAP INSTEAD OF MAP --- NEED 2, 4, 8, 16, 32, AND 64 THREADS; NEED TO RUN 3 TIMES FOR AVERAGE  
  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING) # USED FOR SINGLES  
 # need to pass PAIRS and TRIPLES to count new values for NC -- instead of count char, count pairs, triples...  
 MY\_PAIRS\_TUPLE = set(FILE\_STRING[i: i + 2] for i in range(0, len(FILE\_STRING), 2)) # USED FOR PAIRS  
 MY\_TRIPLES\_TUPLE = set(FILE\_STRING[i: i + 3] for i in range(0, len(FILE\_STRING), 3)) # USED FOR TRIPLES  
  
 # COUNTS SINGLES  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE) # PAIRS  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 # TRIPLES  
  
 MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR  
 print("TOTAL NUMBER OF CHARACTERS: " + str(MY\_TOTAL\_CHARACTERS) + " (used to solve for pc)")  
  
 # iterator is spent to find MY\_TOTAL\_CHARACTERS, need to remake  
 # reset iterator  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 # SPENT ITERATOR AGAIN  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # PROBABILITY DISTRIBUTIONS  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)) + " (single characters)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_PAIR)) + " (pairs)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)) + " (triples)")  
  
 # reset BOTH iterators FOR ALL  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT AGAIN  
  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = MY\_WORKER\_POOL.map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_INFORMATION\_OF\_EACH\_PAIR = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR)  
 MY\_INFORMATION\_OF\_EACH\_TRIPLE = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)  
  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 MY\_TOTAL\_INFORMATION\_PAIRS = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_PAIR), 3)  
 MY\_TOTAL\_INFORMATION\_TRIPLES = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_TRIPLE), 3)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # END TIME - START TIME to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")  
  
 # TOTAL INFORMATION IN THE FILE  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_IN\_STREAM) + " (SINGLES)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_PAIRS) + " (PAIRS)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_TRIPLES) + " (TRIPLES)")

RUN 1: 355809 nanoseconds

RUN 2: 286101 nanoseconds

RUN 3: 379758 nanoseconds

\*\*\*\*\* 4 THREADS

RUN 1: 225802 nanoseconds

RUN 2: 377619 nanoseconds

RUN 3: 241198 nanoseconds

\*\*\*\*\* 8

RUN 1: 249323 nanoseconds

RUN 2: 429366 nanoseconds

RUN 3: 375909 nanoseconds

\*\*\*\*\* 16

RUN 1: 241626 nanoseconds

RUN 2: 358375 nanoseconds

RUN 3: 414397 nanoseconds

\*\*\*\*\*\* This seems to be working…

For a file as small as the test file we’re using… the addition of threads seems to only INCREASE the time, not decrease it.

\*\*\*\*\* will now try running on the actual project file….

First run will be WITHOUT concurrency… (0 THREADS)

import time  
import math  
from functools import reduce  
from multiprocessing import Pool # THIS IS FOR CONCURRENCY ------------------  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
  
# lambda sum for reduce --------------------------------  
myTotal = lambda x, y: x + y  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/WarAndPeace.txt",  
 "r")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------  
 MY\_WORKER\_POOL = Pool(2) # THIS IS FOR CONCURRENCY ------- NUMBER IS NUMBER OF THREADS  
 # USE MY\_WORKER\_POOL.MAP INSTEAD OF MAP --- NEED 2, 4, 8, 16, 32, AND 64 THREADS; NEED TO RUN 3 TIMES FOR AVERAGE  
  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING) # USED FOR SINGLES  
 # need to pass PAIRS and TRIPLES to count new values for NC -- instead of count char, count pairs, triples...  
 MY\_PAIRS\_TUPLE = set(FILE\_STRING[i: i + 2] for i in range(0, len(FILE\_STRING), 2)) # USED FOR PAIRS  
 MY\_TRIPLES\_TUPLE = set(FILE\_STRING[i: i + 3] for i in range(0, len(FILE\_STRING), 3)) # USED FOR TRIPLES  
  
 # COUNTS SINGLES  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE) # PAIRS  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 # TRIPLES  
  
 MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR  
 print("TOTAL NUMBER OF CHARACTERS: " + str(MY\_TOTAL\_CHARACTERS) + " (used to solve for pc)")  
  
 # iterator is spent to find MY\_TOTAL\_CHARACTERS, need to remake  
 # reset iterator  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 # SPENT ITERATOR AGAIN  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # PROBABILITY DISTRIBUTIONS  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)) + " (single characters)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_PAIR)) + " (pairs)")  
 print("PROBABILITY DISTRIBUTION: " + str(tuple(MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)) + " (triples)")  
  
 # reset BOTH iterators FOR ALL  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT AGAIN  
  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_INFORMATION\_OF\_EACH\_PAIR = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR)  
 MY\_INFORMATION\_OF\_EACH\_TRIPLE = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)  
  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 MY\_TOTAL\_INFORMATION\_PAIRS = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_PAIR), 3)  
 MY\_TOTAL\_INFORMATION\_TRIPLES = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_TRIPLE), 3)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # END TIME - START TIME to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")  
  
 # TOTAL INFORMATION IN THE FILE  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_IN\_STREAM) + " (SINGLES)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_PAIRS) + " (PAIRS)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_TRIPLES) + " (TRIPLES)")

RESULTS: error with log…

ERROR was with MY PROBABILITIES – get rid of round… it was creating 0 values and trying to log them

MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = map(lambda x: x / MY\_TOTAL\_CHARACTERS,  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR AGAIN  
 # print(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER))  
 '''  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = map(lambda x: round(x / MY\_TOTAL\_CHARACTERS, 3),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
 '''  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 '''  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
 '''

TOTAL NUMBER OF CHARACTERS: 3227581 (used to solve for pc)

RESULTS:

112013085818 nanoseconds

112013085.818 microseconds (μs)

112.013085818 seconds

112.01 seconds (rounded)

INFORMATION IN DATA STREAM: 743699.526 (SINGLES)

INFORMATION IN DATA STREAM: 160299.742 (PAIRS)

INFORMATION IN DATA STREAM: 52336.552 (TRIPLES)

\*\*\* Now going to grab the average time for 0 threads

\*\*\*\*\* 0 THREADS

RUN 1: 112.013085818 seconds

RUN 2: 119.011069424 seconds

RUN 3: 110.80892328 seconds

AVERAGE = 341.8330785 / 3 = 113.9443595 seconds

TIME FOR CONCURRENCY!!!

import time  
import math  
from functools import reduce  
from multiprocessing import Pool # THIS IS FOR CONCURRENCY ------------------  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
  
# lambda sum for reduce --------------------------------  
myTotal = lambda x, y: x + y  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/WarAndPeace.txt",  
 "r", encoding="utf8")  
 FILE\_STRING = MY\_TEXT\_FILE.read()  
  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------  
 MY\_WORKER\_POOL = Pool(2) # THIS IS FOR CONCURRENCY ------- NUMBER IS NUMBER OF THREADS  
 # USE MY\_WORKER\_POOL.MAP INSTEAD OF MAP --- NEED 2, 4, 8, 16, 32, AND 64 THREADS; NEED TO RUN 3 TIMES FOR AVERAGE  
  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING) # USED FOR SINGLES  
 # need to pass PAIRS and TRIPLES to count new values for NC -- instead of count char, count pairs, triples...  
 MY\_PAIRS\_TUPLE = set(FILE\_STRING[i: i + 2] for i in range(0, len(FILE\_STRING), 2)) # USED FOR PAIRS  
 MY\_TRIPLES\_TUPLE = set(FILE\_STRING[i: i + 3] for i in range(0, len(FILE\_STRING), 3)) # USED FOR TRIPLES  
  
 # COUNTS SINGLES  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS))  
  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE) # PAIRS  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)),  
 MY\_TRIPLES\_TUPLE) # TRIPLES  
  
 MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR  
 # print("TOTAL NUMBER OF CHARACTERS: " + str(MY\_TOTAL\_CHARACTERS) + " (used to solve for pc)")  
  
 # iterator is spent to find MY\_TOTAL\_CHARACTERS, need to remake  
 # reset iterator  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = MY\_WORKER\_POOL.map(lambda x: x / MY\_TOTAL\_CHARACTERS,  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 # SPENT ITERATOR AGAIN  
   
 # print(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER))  
  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = MY\_WORKER\_POOL.map(lambda x: x / MY\_TOTAL\_CHARACTERS,  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = MY\_WORKER\_POOL.map(lambda x: x / MY\_TOTAL\_CHARACTERS,  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map((lambda x: FILE\_STRING.count(x)), MY\_TRIPLES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_INFORMATION\_OF\_EACH\_PAIR = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR)  
 MY\_INFORMATION\_OF\_EACH\_TRIPLE = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)  
  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 MY\_TOTAL\_INFORMATION\_PAIRS = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_PAIR), 3)  
 MY\_TOTAL\_INFORMATION\_TRIPLES = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_TRIPLE), 3)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # END TIME - START TIME to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")  
  
 # TOTAL INFORMATION IN THE FILE  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_IN\_STREAM) + " (SINGLES)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_PAIRS) + " (PAIRS)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_TRIPLES) + " (TRIPLES)")

\*\*\* Now going to grab the average time for thread pools

**ERROR:** pickling???

def run\_dill\_encoded(payload):  
 fun, args = dill.loads(payload)  
 return fun(\*args)  
  
  
def apply\_async(pool, fun, args):  
 payload = dill.dumps((fun, args))  
 return pool.apply\_async(run\_dill\_encoded, (payload,))

# COUNTS SINGLES  
NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = apply\_async(MY\_WORKER\_POOL, lambda x: FILE\_STRING.count(x),  
 MY\_PAIRS\_TUPLE)

This runs but isn’t iterable for reduce

\*\* found a solution which has to use a GLOBAL file…

import time  
import math  
from functools import reduce  
from multiprocessing import Pool

also can’t use lambda…

def file\_count(c):  
 global FILE\_STRING  
 return FILE\_STRING.count(c)

MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/WarAndPeace.txt",  
 "r", encoding="utf8")  
  
FILE\_STRING = MY\_TEXT\_FILE.read()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------

MY\_WORKER\_POOL = Pool(processes=2) # THIS IS FOR CONCURRENCY ------- NUMBER IS NUMBER OF THREADS

NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_UNIQUES\_TUPLE)

Final code for Concurrency…

import time  
import math  
from functools import reduce  
from multiprocessing import Pool  
  
'''  
REQUIREMENTS FOR: COUNTING THE TOTAL NUMBER OF CHARACTERS  
\* know count of each unique (ALREADY DONE)  
\* return sum of these counts  
  
PROBABILITY DISTRIBUTION:  
\* probability p of some character c (pc) is the NUMBER OF OCCURRENCES OF C divided by TOTAL CHARACTERS IN DATA STREAM  
\*\* so pc = nc / total c  
\* probability distribution is the probability of EACH CHARACTER in data stream  
'''  
  
'''  
REQUIREMENTS FOR: A FUNCTIONAL PARADIGM  
\* need to use map/reduce  
\* functional paradigms use CONSTANTS outside their scope; doesn't rely on mutable variables -- passes returned results  
\* from function to function   
\*\* PURE functional programs only depend on their input (no internal state)  
'''  
  
  
def probabilities(count):  
 global MY\_TOTAL\_CHARACTERS  
 return count / MY\_TOTAL\_CHARACTERS  
  
  
def file\_count(char):  
 global FILE\_STRING  
 return FILE\_STRING.count(char)  
  
  
# lambda sum for reduce --------------------------------  
myTotal = lambda x, y: x + y  
  
MY\_TEXT\_FILE = open("C:/Users/Mikaela/Documents/Spring2020/CS441 Progrm LangDes&Imp/Final Project/WarAndPeace.txt",  
 "r", encoding="utf8")  
  
FILE\_STRING = MY\_TEXT\_FILE.read()  
MY\_TOTAL\_CHARACTERS = len(FILE\_STRING)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 startClock = time.perf\_counter\_ns() # EXCLUDE FILE I/O FOR THE TIMER --------------------------  
 MY\_WORKER\_POOL = Pool(processes=2) # THIS IS FOR CONCURRENCY ------- NUMBER IS NUMBER OF THREADS  
 # USE MY\_WORKER\_POOL.MAP INSTEAD OF MAP --- NEED 2, 4, 8, 16, 32, AND 64 THREADS; NEED TO RUN 3 TIMES FOR AVERAGE  
  
 MY\_UNIQUES\_TUPLE = set(x for x in FILE\_STRING) # USED FOR SINGLES  
 # need to pass PAIRS and TRIPLES to count new values for NC -- instead of count char, count pairs, triples...  
 MY\_PAIRS\_TUPLE = set(FILE\_STRING[i: i + 2] for i in range(0, len(FILE\_STRING), 2)) # USED FOR PAIRS  
 MY\_TRIPLES\_TUPLE = set(FILE\_STRING[i: i + 3] for i in range(0, len(FILE\_STRING), 3)) # USED FOR TRIPLES  
  
 # COUNTS SINGLES  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_UNIQUES\_TUPLE)  
 # print(tuple(NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS))  
  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_PAIRS\_TUPLE) # PAIRS  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map(file\_count,  
 MY\_TRIPLES\_TUPLE) # TRIPLES  
  
 # MY\_TOTAL\_CHARACTERS = reduce(myTotal, NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS) # SPENT ITERATOR  
 # print("TOTAL NUMBER OF CHARACTERS: " + str(MY\_TOTAL\_CHARACTERS) + " (used to solve for pc)")  
  
 # iterator is spent to find MY\_TOTAL\_CHARACTERS, need to remake  
 # reset iterator  
 # NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_UNIQUES\_TUPLE)  
  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER = MY\_WORKER\_POOL.map(probabilities,  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS)  
 # SPENT ITERATOR AGAIN  
  
 # print(tuple(MY\_PROBABILITIES\_OF\_EACH\_CHARACTER))  
  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR = MY\_WORKER\_POOL.map(probabilities,  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS) # SPENT  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE = MY\_WORKER\_POOL.map(probabilities,  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS) # SPENT  
  
 # RESET NC AGAIN  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_UNIQUES\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_PAIRS\_TUPLE)  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS = MY\_WORKER\_POOL.map(file\_count, MY\_TRIPLES\_TUPLE)  
  
 # map this function: (nc)(−pc)lg(pc) to each NC  
 MY\_INFORMATION\_OF\_EACH\_CHARACTER = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_CHARACTER\_C\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_CHARACTER)  
 MY\_INFORMATION\_OF\_EACH\_PAIR = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_PAIR\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_PAIR)  
 MY\_INFORMATION\_OF\_EACH\_TRIPLE = map((lambda nc, pc: round(nc \* (-pc) \* math.log(pc, 2), 3)),  
 NUMBER\_OF\_TIMES\_EACH\_TRIPLE\_OCCURS,  
 MY\_PROBABILITIES\_OF\_EACH\_TRIPLE)  
  
 # get the summation  
 MY\_TOTAL\_INFORMATION\_IN\_STREAM = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_CHARACTER), 3)  
 MY\_TOTAL\_INFORMATION\_PAIRS = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_PAIR), 3)  
 MY\_TOTAL\_INFORMATION\_TRIPLES = round(reduce(myTotal, MY\_INFORMATION\_OF\_EACH\_TRIPLE), 3)  
  
 RUN\_TIME = ((time.perf\_counter\_ns()) - startClock) # END TIME - START TIME to get duration  
 print(str(RUN\_TIME) + " nanoseconds")  
 print(str(RUN\_TIME / 1000) + " microseconds (μs)")  
 print(str(RUN\_TIME / 1000000000) + " seconds") # to return seconds  
 # 2 decimals w/ str(round(answer, 2))  
 print(str(round(RUN\_TIME / 1000000000, 2)) + " seconds (rounded)")  
  
 # TOTAL INFORMATION IN THE FILE  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_IN\_STREAM) + " (SINGLES)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_PAIRS) + " (PAIRS)")  
 print("INFORMATION IN DATA STREAM: " + str(MY\_TOTAL\_INFORMATION\_TRIPLES) + " (TRIPLES)")

Results: (for a Pool of 2…)

70348807890 nanoseconds

70348807.89 microseconds (μs)

70.34880789 seconds **BIG JUMP!!**

70.35 seconds (rounded)

INFORMATION IN DATA STREAM: 743699.526 (SINGLES)

INFORMATION IN DATA STREAM: 160299.742 (PAIRS)

INFORMATION IN DATA STREAM: 52336.552 (TRIPLES)

**\*\*\*AVERAGES\*\*\***

\*\*\*\*\* 2 THREADS

RUN 1: 70.34880789 seconds

RUN 2: 68.556980977 seconds

RUN 3: 68.56214363 seconds

AVERAGE = 207.4679326 / 3 = 69.15597754 seconds

\*\*\*\*\* 4 THREADS

RUN 1: 48.660048846 seconds

RUN 2: 47.938612678 seconds

RUN 3: 47.875360353 seconds

AVERAGE = 144.4740219 / 3 = 48.15800729 seconds

\*\*\*\*\* 8 THREADS

RUN 1: 47.571287324 seconds

RUN 2: 47.750695475 seconds

RUN 3: 47.074456033 seconds

AVERAGE = 142.3964388/ 3 = 47.46547961 seconds

\*\*\*\*\* 16 THREADS

RUN 1: 47.45276913 seconds

RUN 2: 47.535462656 seconds

RUN 3: 47.589933512 seconds

AVERAGE = 142.578165298/ 3 = 47.526055099333333333333333333333 seconds

\*\*\*\*\* 32 THREADS

RUN 1: 48.099712915 seconds

RUN 2: 48.104893957 seconds

RUN 3: 48.147236941 seconds

AVERAGE = 144.351843813/ 3 = 48.117281271 seconds

\*\*\*\*\* 60 THREADS (64 didn’t work…)

RUN 1: 50.140585623 seconds

RUN 2: 50.53693208 seconds

RUN 3: 50.303801896 seconds

AVERAGE = 150.981319599/ 3 = 50.327106533 seconds

**GRAPH:**

Graph will show EXECUTION TIME in seconds, as a FUNCTION OF THE NUMBER OF THREADS…

SUMMARY:

**The min (fastest result)** was with 8 and 16 threads, with the **absolute fastest being 8 threads**.

**The biggest improvement** was seen **between 0 and 2 threads**, **improving by a factor of ~ 1.6**

The second biggest improvement was seen between 2 and 4 threads, improving by a factor of ~ 1.4

Prior to the results, the expectation was that there would be ***consistent improvement*** due to having more threads to help with the load. As the threads increased… this was proven to be wrong, since after 8 threads, the **execution time only increased.**

Taking a look at the power point slides from class, the guess as to why comes from **extremes:** at one end there were not enough threads, and at the other end there were too many. The **benefit of having multiple threads to handle the problem started to be outmatched by the cost of having to create & use multiple threads.**

Threads for concurrency (x values) were chosen by the pattern 2n were n was in range 1 through 6. Creating a graph for n we see:

There seems to be a limit as the execution time approaches 47. It does not go below this point. I.e.: **Lim f(n) as n -> 3= 47**

The graph resembles the shape of : **f(x) = (-lnx)**

**Troubles in the process:**

There was an issue with trying to use 64 threads. This would not run (not sure as to why). 60 threads were used as the max instead.

That is all.