HS 608 Project 2: String and List Utilities
Upload to Canvas (algPack.py and testcases.txt)

This project will develop your algorithmic thinking as you code a package of algorithms for **matrices of experiments, dna strings**, and **vectors of lab tests**. **No def main() is needed**, as your package is designed to be imported by others for use in their research (for example, investigation of genomic sequence).

You will define function bodies for each of the **10** algorithms described below. You may not change the function headers or descriptions. Be sure your code fulfills all the requirements of each function description. **Your code should pass my limited test cases**, but your **testing grade will be based on your own unit tests, which should be placed along with mine in function test_suite and called from if __name__ == '__main__':**

Your code **may use the bracket operator, concatenation operator, the len function and the append function**, but no other string or list functions/operations. NB: this means **NO slices, NO in operator, etc, unless you write the code for the operator you use.** The exception is that you are encouraged to use for loops throughout this assignment, so you may use the **in** operator within the for loop header (not the body).

For example, you may use:

```
for character in myString:
for i in range(start, len(lst), increment):
```

You must code the functions yourself, not use built-ins. Don't use anyone else's work work. You should be able to explain your code in a code walk through.

Feel free to call on functions you've written for one algorithm as you later code another string or list algorithm.

Grading:

1. Correctness and quality of solution: 75% of your grade: Your program should return the correct value for all possible inputs that are valid according to the function description. Your program should **never crash on valid input such as the empty string**. For functions that require an integer input, test your program for negative, zero and positive integers. Your solution **should not be unnecessarily complex and should be efficient** in cost of time and space. You may create and call helper functions.

2. Documentation: 5%: Include header documentation with your name, and a brief description of this collection of algorithms. Throughout your program, **use identifiers that are meaningful**. If there are any obscure constructs, be sure to clearly explain them. Each function **already has** a brief description of **what it accomplishes or returns**, **how its parameters are used (including type of parameters)**, **and preconditions**, if there are any. Therefore, you may keep these as your function descriptions; you do **not** have to write your own function descriptions.

3. Testing: 20%: For each function, test all its paths of execution. This is expected to be in the range of **65-100+ test cases**. If you have an error in your code that you do not detect with testing, you will lose both points for testing and points for correctness. On the other hand, **good testing that reveals errors results in full testing points, even if you don't successfully correct the error**. All test results should be uploaded in file testcases.txt.

Do your own work, come up with your own solution to each algorithm.
Here are the **9** function descriptions and headers, along with very limited Examples:

```
def colMean(m, col):
    """
    If col is valid return the mean of values in column col, else print "col out of bounds"
    and return.
    :param m: a matrix of numbers represented as a list of lists
    :param col: an integer that represents a valid column index of m
    :return: the float value that is the mean of the values in column col of m
    Example:
    >>> colMean([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2)
    4.0
    """
```

```
def colMode(m, col):
    """
    If col is valid return the mode of values in col else print "col out of bounds" and return.
    :param m: a matrix of integers represented as a list of lists
    :param col: an integer that represents a valid column index of m
    :return: the integer value that is the mode of the values in column col of m
    Example:
    >>> colMode([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2)
    3
    """
```

#Sample Standard Deviation Example Problem with steps to calculate:
#https://www.thoughtco.com/sample-standard-deviation-problem-609528

```
def colStandardize(m, col):
    """
    If col is valid return a new matrix identical to m except that the values in col are
    standardized, else print "col out of bounds" and return.
    :param m: a matrix of numbers represented as a list of lists
    :param col: an integer that represents a valid column index of m
    :return: a new matrix of the contents of m, with values in column col standardized
    Example:
    >>> colStandardize([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2)
    [[2, 4, 1.155], [1, 2, -0.577], [1, 2, -0.577]]
    """
```

```
def colMinMaxNormalize(m, col):
    """
    If col is valid return a new matrix identical to m except that the values in col are
    Normalized, else print "col out of bounds" and return.
    :param m: a matrix of numbers represented as a list of lists
    :param col: an integer that represents a valid column index of m
    :return: a new matrix of the contents of m with values in column col normalized between 0 and 1
    Example:
    >>> colMinMaxNormalize([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2)
    [[2, 4, 1], [1, 2, 0], [1, 2, 0]]
    """
```

```
def mutation(dna, index, newNT):
    """
    If index is valid return a string with that represents a SNP (single nucleotide
    polymorphism) of dna, else print "index out of bounds" and return None.
    :param dna: a string
    :param index: an integer such that    0 <= index < len(dna)
    :param newNT: a string to replace the character at index
    :return: a string composed of the characters of dna with the value at index replaced with newNT
    Example:
    >>> mutation("ACTCGG", 0, "G")
    "GCTCGG"
    """
```

```
def insertion (dna, index, newNTs):
    """
    If index is valid return a string that represents an insertion mutation of dna,
    else print "index out of bounds" and return None.
    :param dna: a string
    :param index: an integer such that    0 <= index <= len(dna)
    :param newNTs: a string to insert into dna at position index
    :return: a string composed of the characters of dna with the value at index replaced with newNT
    Examples:
    >>> insertion ("ACTCGG", 6, "AGC")
    "ACTCGGAGC"
    >>> insertion ("ACTCGG", 7, "AGC")
    "Index out of bounds"
    """
```

```
def deletion(dna, index, numNTdeleted ):
    """
    If index is valid return a string that represents a deletion mutation of dna,
    else print "index out of bounds" and return None.
    :param dna: a string
    :param index: an integer such that    0 <= index < len(dna)
    :param numNTdeleted: integer indicating how many characters to delete
    :return: a string composed of the characters of dna with up to numNTdeleted beginning at position index.
    Examples:
    >>> deletion("ACTCGG", 5, 2)
    "ACTCG"
    >>> deletion("ACTCGG", 1, 2)
    "ACGG”
    """
```

```
def euclideanDistance(v1, v2):
    """
    Return the euclidean distance between vectors of equal length
    :param v1: a vector of numbers represented as a list
    :param v2: a vector of numbers represented as a list
    :return: the float value that is the Euclidean distance between v1 and v2
    Examples:
    >>> euclideanDistance([3, 1], [6, 5])
    5
    >>> euclideanDistance([0, 0], [3, 4])
    5
    >>> euclideanDistance([3, 6, 1, 2, 8, 2, 1], [3, 6, 1, 2, 8, 2, 1])
    0
    """

def normalizeVector(v):
    """
    Return a new vector that is vector v normalized
    :param v: a vector of numbers represented as a list
    :return: a new vector equivalent to v scaled to length 1 (ie: a unit vector)
    Example:
    >>> normalizeVector([6, 8])
   [.6, .8]
    >>> normalizeVector([25,2,7,1,-5,12])
   [0.8585035246793065, 0.06868028197434452, 0.2403809869102058,
0.03434014098717226, -0.1717007049358613, 0.4120816918460671]
    """

def test_suite():
    """ Run the suite of tests for code in this module (this file).
    """
    test(abs(colMean([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2) - 4.0) < .0000001 )
    test(colMode([[2, 4, 6], [1, 2, 3], [1, 2, 3]], 2) == 3)
    # colStandardize (to be added)
    # colMinMaxNormalize(m, col): (to be added)

    #Proper normalize test for a short vector:
    test(abs(normalizeVector([6, 8])[0] - .6) < .0000001 )
    test(abs(normalizeVector([6, 8])[1] - .8) < .0000001 )
    #However, to test normalize you may paste in these correct vector elements, but to
whatever precision your system has
    test(normalizeVector([25, 2, 7, 1 ,-5 ,12]) == [0.8585035246793065,
0.06868028197434452, 0.2403809869102058, 0.03434014098717226,
-0.1717007049358613, 0.4120816918460671])
    test(abs(euclideanDistance([3, 1], [6, 5]) - 5) < .0000001)
    test(abs(euclideanDistance([0, 0], [3, 4]) - 5) < .0000001)
    test(euclideanDistance([3, 6, 1, 2, 8, 2, 1], [3, 6, 1, 2, 8, 2, 1]) == 0)

    #### Your own tests:  65 - 100+ tests  ####
    #Test above functions
    #Test 3 string functions
    #Test your helper functions
```