Bio723P Coding for Scientists Final Assignment, 2021/22

This assignment counts towards 100% of the marks for this module. The assignment is divided into four questions, each carrying 25 marks.

Student name and number

Enter your full name and student number in the cell below

full name: Jack Coutts

student number: 210653579

Deadline and submission

Complete this notebook with your code, compress it and submit it as a single .zip or .gz file through the QMPlus page by the following deadline:

```
Friday Nov 5th, 24:00 midnight
```

Submission checklist

- Notebook is written in Python 3 and can be opened and run on the EECS/SBCS JupyterHub.
- Any import statements are grouped in the cell provided at the end of this introduction; name and version of the library are given in a comment. There should be no other import statements in the notebook.
- Notebook functions pass all tests when cells are run with a fresh kernel from top to bottom.
- Notebook only contains well commented answer code and the original tests. Please remove your own test code.
- Notebook includes the full name and student number of the author.
- Zip file containing the notebook is free from errors.

Please refrain from editing the text of the assignment and the test code provided. Feel free to insert text or code cells as needed, but please clean up rough work that you do not wish to be marked.

Marking

Your work will be marked **automatically**. Marks will be based on:

```
completeness and correctness: 100%
```

For automatic marking to function appropriately, each of your answers **must** pass the tests provided when running the notebook from top to bottom on a fresh kernel. Code that does not pass the tests or is not run (directly or indirectly, e.g. a function calling another function) by the functions you are asked to implement will **not** be marked.

The test code provided with each question does **not** ensure that your submission is correct; it only ensures that it can be marked automatically. You will need to test your code independently for correctness; however, please remove your test code from your final submission, and do not submit any test data; the marking script will test your code with its own choice of data.

How to answer questions

This assignment contains four questions, that can in principle be tackled in any order. However, we suggest that you answer Question 3 before attempting Question 4.

Each question will require you to code a few functions. As you progress through the subquestions, you may have to rely on your answer to previous subquestions to perform some operations. Please call the relevant functions and avoid duplicating code.

Use of libraries

This assignment is designed to be feasible using only the functions in the Python standard library, and we recommend that you do not use anything else. However, you are free to use any library available in PyPI if you think that would make your work easier. If you do decide to use any additional libraries, these must be installable from PyPI using *pip*. Specify in the cell below all necessary import statements; for functions that are not part of the standard library, include the name and version of the library in a comment.

In [1]:
```
# Please put all your import statements here, according to the example. Include a comment
# the library and its version. Do not include import statements in the rest of the code

# Example:
# Biopython v 1.78
# import Bio # this would obviously be uncommented if you are importing Biopython
```

Preliminary reading

Please read the PDF file `BIO723P_Assignment_Background.pdf` distributed with this assigment carefully before you start coding.

Question 1: Primers and melting temperature

Question 1.a: Reading a DNA sequence from a file

Write a python function called `readDNAsequence` that takes as its argument the name of a file. When passed the name of a FASTA file, the function should read the file, discard the header and return the sequence as a string. Your code should raise `BadSequenceException` (defined below) if the sequence part of the file contains characters that are not one of the letters A, C, T, G, U. All U nucleotides should be replaced by T in the returned string (for simplicity, we will be working with T only throughout the rest of this assignment).

In [2]:
```
# Run this cell to define the exception
class BadSequenceException(Exception):
    pass
```

In [3]:

```
def readDNAsequence(filename):

    # Open file and set to read
    with open(filename, 'r') as f:

        # Create empty string which will
        # be filled as the output
        protein=''

        # Remove header lines with >
        # Finish when you reach an empty line
        # For other lines remove supplementary characters
        # , change to upper case, and replace U with T
        for line in f:
            if '>' in line:
                continue
            elif line =='':
                break
            else:
                p=line.rstrip().upper().replace('U','T')

                # Raise the correct error is non-nucleotide
                for i in p:
                    if i not in 'ATCG':
                        raise BadSequenceException(i, ' is not a nucleotide - incorrect se

                # If all requirements satified add charcter to string
                protein+=p

        # Output string of nucleotides
        return protein
```

In [4]:

```
# Test code
import os
with open("test9876345.fas", "wt") as _OUTF:
    _OUTF.write("> test\n")
    _OUTF.write("ACTG\n")
_seq=readDNAsequence("test9876345.fas")
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
os.remove("test9876345.fas")
print("OK")
```

```
OK
```

Question 1.b: Computing the complement of a sequence

Write a function called `complement` that takes a string containing a DNA sequence as its only parameter and returns the complement of the sequence in a string. The function should raise `BadSequenceException` if the argument sequence contains anything else than the four characters A, C, T, G. Do not reverse the string; for the avoidance of doubt, if the input string starts with A then the complement string should start with T.

In [5]:

```
# Your code here

def complement(string):

    # Define complementary pairs
    comps={'A':'T', 'T': 'A', 'C': 'G', 'G': 'C'}

    # Turn sequence string to list
    lst=list(string.upper())

    # Check for non-nucleotides
```

```
    for i in lst:
        if i not in comps:
            raise BadSequenceException (i, ' is not a nucleotide - incorrect sequence.')

    # Change nucleotides in sequence to their complement
    # then turn back into string
    seq=(''.join([comps.get(i) for i in lst]))

    return seq
```

In [6]:

```
# Test code
_seq=complement("ACGTTTCG")
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
print("OK")
```

```
OK
```

Question 1.c: Extracting primers

Write a function called `primer` that takes three parameters: a DNA sequence called `sequence`, an integer `length` that is 20 by default and a Boolean value `forward` that is `True` by default. When `forward` is `True` (or is not passed), the function should return a Forward primer for the sequence passed as `sequence`; when it is `False`, it should return a Reverse primer. The length of the primer is specified by `length`; if this is not passed, a primer of length 20 should be returned. Refer to the Background document for a definition of primers and how to compute them (for the avoidance of doubt, if the sequence string ends with a C, then the reverse primer string should start with a G). If the sequence is shorter than `length` nucleotides, your code should raise a `BadSequenceException`.

In [7]:

```
# Your code here

def primer(sequence, length=20, forward=True):
    # Sequence is too short
    slen=len(sequence)
    if slen<length:
        raise BadSequenceException ('Invalid sequence - sequence shorter than primer lengt
    #Forward primer
    if forward==True:
        front=sequence[:length]
        return front

    #Reverse Primer
    else:
        back=sequence[-length:]
        c=complement(back)
        rv=c[::-1]
        return rv
```

In [8]:

```
# Test code
_seq=primer(sequence="AAAAATTTTTCCCCCGGGGGAAAAA", length= 20, forward=False)
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
print("OK")
```

```
OK
```

Question 1.d: Computing the melting temperature

Write a function called `meltingTemp` that takes a string representing a primer as its argument. The function should return the melting temperature of the primer in degrees Celsius according to the equation given in the

Background document. If the sequence contains characters other than A, C, T, G, the function should raise a `BadSequenceException` .

In [9]:
```
# Your code here

def meltingTemp(primerstr):
    # Exception
    primerstr=primerstr.upper()
    for i in primerstr:
        if i not in 'ATCG':
            raise BadSequenceException (i, ' is not a nucleotide - incorrect sequence.')
    #Metling temp
    A=primerstr.count('A')
    T=primerstr.count('T')
    C=primerstr.count('C')
    G=primerstr.count('G')
    tmp=4*(G+C)+(2*(A+T))
    return tmp
```

In [10]:
```
# Test code
_temp=meltingTemp("AAAAATTTTTCCCCCGGGGG")
assert ((type(_temp) is type(0.0)) or
        (type(_temp) is type(0))), "Return value is not a number: %r" % _temp
print("OK")
```

```
OK
```

Question 1.e: Putting it all together

Write a function called `sequencePCRtemp` that takes a string containing the name of a FASTA file as its argument. The function should return the average melting temperature of the two primers of the sequence as a `float` .

In [11]:
```
# Your code here

def sequencePCRtemp(fasta):
    #Read sequence
    s=readDNAsequence(fasta)
    #Find forward primers
    forw=primer(s)
    #Find reverse Primers
    rev=primer(s, forward=False)
    #Forward melting temp
    m_forw=meltingTemp(forw)
    #Reverse melting temp
    m_rev=meltingTemp(rev)
    #Average of the melting temps
    avmt=float((m_forw+m_rev)/2)

    return avmt
```

In [12]:
```
# Test code
import os
with open("test9876346.fas", "wt") as _OUTF:
    _OUTF.write("> test\n")
    _OUTF.write("AAAAACCCCCTTTTTGGGGGAAAAA\n")
_temp=sequencePCRtemp("test9876346.fas")
assert type(_temp) is type(0.0), "Return value is not a float: %r" % _temp
```

```
os.remove("test9876346.fas")
print("OK")
```

```
OK
```

Question 2: Translation and reading frames

Question 2.a: Reading frames

Write a function `translate` that takes a string containing a DNA sequence as its input and outputs a Python dictionary containing the translation of the sequence in all possible reading frames. The keys of the dictionary should be `f1`, `f2`, `f3` for the three forward frames and `r1`, `r2` and `r3` for the reverse reading frames; the value of each key should be the translation of the sequence in the corresponding frame. For simplicity and ease of debugging, **do not complement the sequence** when computing the reverse reading frames; just reverse it. Use an asterisk (`*`) to represent stop codons. Always translate the entire sequence.

In [13]:

```
# Your code here

def translate(sequence):

    #Translation dictionary
    dictionary= {
    'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
    'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
    'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
    'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
    'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
    'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
    'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
    'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
    'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
    'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
    'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
    'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
    'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
    'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
    'TAC':'Y', 'TAT':'Y', 'TAA':'*', 'TAG':'*',
    'TGC':'C', 'TGT':'C', 'TGA':'*', 'TGG':'W'}

    #Check sequence only contains nucletides
    for i in sequence:
        if i not in 'ATCGatcg':
            raise BadSequenceException (i, ' is not a nucleotide - incorrect sequence.')

    #Convert sequence to list
    lst=list(sequence.upper())
    rlst=list(sequence.upper())
    rlst.reverse()

    #Create a dictionary for the reading frames
    seqlen=len(lst)
    codonlen=3
    codons={'f1':'', 'f2':'', 'f3':'', 'r1':'', 'r2':'', 'r3':''}
    codons['f1']=[''.join(lst[i:i + codonlen]) for i in range(0, seqlen, codonlen)]
    codons['f2']=[''.join(lst[i:i + codonlen]) for i in range(1, seqlen, codonlen)]
    codons['f3']=[''.join(lst[i:i + codonlen]) for i in range(2, seqlen, codonlen)]
    codons['r1']=[''.join(rlst[i:i + codonlen]) for i in range(0, seqlen, codonlen)]
    codons['r2']=[''.join(rlst[i:i + codonlen]) for i in range(1, seqlen, codonlen)]
    codons['r3']=[''.join(rlst[i:i + codonlen]) for i in range(2, seqlen, codonlen)]
```

```
    #Translate the reading frames and convert to strings
    def translatel(ls):
        a=ls
        z=''
        for m in a:
            if m in dictionary:
                z+=''.join([dictionary[m]])
        return z

    f1=codons['f1']
    f2=codons['f2']
    f3=codons['f3']
    r1=codons['r1']
    r2=codons['r2']
    r3=codons['r3']

    codons['f1']=translatel(f1)
    codons['f2']=translatel(f2)
    codons['f3']=translatel(f3)
    codons['r1']=translatel(r1)
    codons['r2']=translatel(r2)
    codons['r3']=translatel(r3)

    return codons
```

In [14]:

```
# Test code
_seqdic=translate("ACTGACTGACTGACTGACTGACTG")
assert type(_seqdic)==type(dict()), "Return value is not a dictionary: %r" % _seqdic
assert set(_seqdic.keys())==set(['f1', 'f2', 'f3', 'r1', 'r2', 'r3']), \
    "Output dictionary has incorrect/missing keys: %r"  % _seqdic.keys()
assert type(_seqdic['f1'])==type(""), \
    "Output dictionary values should be strings, not %r" % type(_seqdic['f1'])
print("OK")
```

```
OK
```

Question 2.b: Locating an ORF

Write a function called `openReadingFrame` that takes a string containing an aminoacid sequence as its argument and returns a string containing the aminoacids between the first Methionine (included) and the first STOP codon that follows it (excluded). Assume the stop codon is represented by an asterisk (*) as would be returned by `translate` above. If either the Methionine or the STOP codon are missing, your function should return an empty string.

In [15]:

```
# Your code here

def openReadingFrame(aminos):

    #Find length of amino acid string
    length=len(aminos)
    #Find first M
    start=aminos.find('M')
    #Find first * between first M and end of sequence
    stop=aminos.find('*', start, length)
    #If * or M not found return empty string
    if start == -1 or stop == -1:
        return ''
    #Return oepn reading frame in a string
    else:
        orf=aminos[start:stop]
        return orf
```

In [16]:
```
# Test code
_seq=openReadingFrame("AMCAPP*L")
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
print("OK")
```

```
OK
```

Question 2.c: Translating a sequence

Write a function called `candidateProtein` that takes a string containing a DNA sequence as its input and outputs the string of aminoacids corresponding to the longest ORF, as extracted by `openReadingFrame` above.

In [17]:
```
# What if there are two the same length

def candidateProtein(sequence):
    #Translate sequence
    frames=translate(sequence)
    #Find open reading frame for each frame
    orfs=[openReadingFrame(frames['f1']),
          openReadingFrame(frames['f2']),
          openReadingFrame(frames['f3']),
          openReadingFrame(frames['r1']),
          openReadingFrame(frames['r2']),
          openReadingFrame(frames['r3'])]
    #Sort orfs in decending order
    orfs.sort(reverse=True, key=len)
    #Select the longest orf - first in list
    longestORF=str(orfs[0])
    return longestORF
```

In [18]:
```
# Test code
_seq=candidateProtein("ATGACTGCTGGGTAG")
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
print("OK")
```

```
OK
```

Question 2.d: Writing a FASTA file

Write a function called `writeFASTA` that takes three string arguments called, in the order, `sequence`, `description` and `filename`. Argument `sequence` should contain an aminoacid sequence. Argument `description` should contain a description (eg name of protein, organism, etc). Argument `filename` should contain a file name. Your code should create the file with the name requested, write to it the description as a FASTA header (i.e. starting with the character `>`) and write the sequence to the file. Long sequences should be formatted over several lines. The function should not return any value.

In [19]:
```
# Your code here

def writeFASTA(sequence, description, filename):
    #Open file
    OUTF=open(filename, 'w')
    linelen=60
    position=0
    #Write header
    OUTF.write('> '+description+'\n')
    #Wriet out sequence
```

```
    while sequence[position:position+linelen]!='':
        OUTF.write(sequence[position:position+linelen]+"\n")
        position+=linelen
    #Close file
    OUTF.close()
```

In [20]:

```
# Test code
import os
import os.path
_rv=writeFASTA(sequence="TESTTESTTESTTESTTEST",
               description="test sequence",
               filename="test9876347.fas")
assert type(_rv) is type(None), "Function should not return anything; it returns %r" % _rv
_fe=os.path.isfile("test9876347.fas")
assert _fe, "Cannot find output file - has it been created?"
os.remove("test9876347.fas")
print("OK")
```

```
OK
```

Question 2.e: Putting it all together

Write a function called `maximalORF` that takes as its argument string `inputfile` containing the name of an input file, string `outputfile` with the name of an output file and string `proteinname` with a description of a candidate protein. The function should read a DNA sequence from the input file and write the candidate protein corresponding to the longest ORF to the output file, in FASTA format. The string supplied in `proteinname` should provide the header of the FASTA file. The function should not return any value.

In [21]:

```
# Your code here

def maximalORF(inputfile, outputfile, proteinname):
    #Read sequence
    s=readDNAsequence(inputfile)
    #Fine the orf
    orf=candidateProtein(s)
    #Write orf to output file
    writeFASTA(orf, proteinname, outputfile)
```

In [22]:

```
# Test code
import os
import os.path
with open("test9876348.fas", "wt") as _OUTF:
    _OUTF.write("> test\n")
    _OUTF.write("ATGACTGCTGGGTAG\n")
_rv=maximalORF(inputfile="test9876348.fas", outputfile="test9876349.fas",
               proteinname="test protein")
assert type(_rv) is type(None), "Function should not return anything; it returns %r" % _rv
_fe=os.path.isfile("test9876349.fas")
assert _fe, "Cannot find output file - has it been created?"
os.remove("test9876348.fas")
os.remove("test9876349.fas")
print("OK")
```

```
OK
```

Question 3: Classification of aminoacids

Question 3.a: Reading an aminoacid sequence

Write a function called `readAAsequence` that takes as its argument the name of a file. When passed the name of a FASTA file containing an aminoacid sequence, the function should read the file, discard the header and return the sequence as a string.

In [23]:
```
def readAAsequence(filename):
    #Open file and write out sequence
    with open(filename, 'r') as f:
        protein=''
        aminoa='ARNDBCQEZGHILKMFPSTWYV'
        for i in f:
            if '>' in i:
                continue
            if i =='':
                break
            else:
                protein+=i.rstrip().upper()
        # Mainly done for 3c but raises an error if invalid characters
        for i in protein:
            if i not in aminoa:
                raise BadSequenceException ('Invalid sequence')
        return protein
```

In [24]:
```
# Test code
import os
with open("test9876350.fas", "wt") as _OUTF:
    _OUTF.write("> test\n")
    _OUTF.write("TESTTESTTESTTESTTEST\n")
_seq=readAAsequence("test9876350.fas")
os.remove("test9876350.fas")
assert type(_seq) is type(""), "Return value is not a string: %r" % _seq
print ("OK")
```

```
OK
```

Question 3.b: Aminoacid usage statistics

Write a function called `AAtypes` that takes as its argument a string containing a sequence of aminoacids. The function should compute, for each protein, the fraction of aminoacids that are polar, small and hydrophobic and return that as a list. For instance, if the chain of a protein contains 35% polar residues, 15% small and 50% hydrophobic residues, the function should return `[0.35, 0.15, 0.5]` (the order is important). Refer to the Venn diagram in the Assessment Background file for the classification of aminoacids (other Venn diagrams may differ). Note that some aminoacids belong to more than one category, so that the sum of these three numbers can be larger than 1.

In [25]:
```
# Your code here

def AAtypes(aminos):
    #length of sequence
    length=len(aminos)
    #Dictionaries
    polar=['C','T','S','N','D','Q','E','R','K','H','Y','W']
    small=['P','V','A','C','T','G','S','D','N']
    hydph=['K','H','W','Y','F','T','C','A','M','L','I','V']
    #Count occurances
    p=0
    s=0
    h=0
    for i in aminos:
        if i in polar:
```

```
            p+=1
        if i in small:
            s+=1
        if i in hydph:
            h+=1
    #Find fractions and put in list
    fract=[(p/length),(s/length),(h/length)]
    return fract
```

In [26]:

```
# Test code
_aatypes=AAtypes("TESTTESTTESTTESTTEST")
assert type(_aatypes) is type([]), "Return value is not a list: %r" % _aatypes
assert len(_aatypes)==3, "Returned list should contain 3 values"
print("OK")
```

```
OK
```

Question 3.c: Processing multiple sequences

Write a function called `AAtypetable` that takes a list of strings called `filelist` and a separate string called `outputfile`. Each element of `filelist` represents the name of a FASTA file containing an aminoacid sequence. For each filename, the function should load the sequence, compute the fraction of aminoacids that are polar, small and/or hydrophobic, and finally output this to the text file specified by `outputfile` in TSV (tab-separated values) tabular format. The table should contain, on each line, the name of the input file and then the percentages of polar, small and hydrophobic residues (in that order). Include a comment at the beginning of the file that starts with a `#` and specifies the content of each column, as follows (numbers and file names shown here are random):

```
# Filename      Polar     Small     Hydro
Pxxx.fasta      0.52      0.22      0.33
Qx3Z.fas        0.47      0.35      0.38
...
```

Include at least two decimal places in your output; do not add any quotes or other spurious characters. Do not worry if the columns do not all look aligned; as long as consecutive values on the same line are separated by a tab, that's fine. If some of the files in `filelist` do not exist, contain invalid data or otherwise cause an error, your function should ignore those files and continue. However, the names of those files should be returned by your function in a list. If all files are processed without errors your function should return an empty list.

In [27]:

```
def AAtypetable(filelist, outputfile):

    # Check for files that do not exist or cause another random error in
    # readAAsequence or AAtypes they will be caught here and added to the list

    def checkfail(filelist):
        failure=[]
        for i in filelist:
            try:
                a=readAAsequence(i)
                b=AAtypes(a)
            except:
                failure+=[i]
        return failure

    #Remove from the filelist any files that have been flagged with errors
    #Create a list containing the filenames and ratios
    #Remove sublists
```

```
    def ratiolist(filelist):
        a=checkfail(filelist)
        for i in filelist:
            if i in a:
                filelist.remove(i)
        p=[]
        for i in filelist:
            a=[i, AAtypes(readAAsequence(i))]
            p+=a

        liist = []
        for i in p:
            if type(i)==str:
                i=[i]
            liist += i

        return liist

    # Using the output list from ratiolist
    # Write to a file with the specified filename
    # One line wirtten at a time with tabs and new lines written in the string

    def writeTSV(lists, filename):
        with open(filename, 'wt') as f:
            head='# Filename'+'\t'+'Polar'+'\t'+'Small'+'\t'+'Hydro'+'\n'
            f.write(head)
            count=0
            for i in lists:

                if type(i)!= str:
                    i=str(i)
                if count <=2:
                    f.write(i + '\t')
                    count+=1
                elif count ==3:
                    f.write(i + '\n')
                    count-=3

    # Check files for errors

    b=checkfail(filelist)

    # Produce list with names and ratios

    ratios=ratiolist(filelist)

    # Write output TSV file

    writeTSV(ratios, outputfile)

    # Return list with files causing an error

    return b
```

In [28]:

```
# Test code
import os
import os.path
for fname in ["_Ptest123.fas", "_Ptest456.fas"]:
    with open(fname, "wt") as _OUTF:
        _OUTF.write("> test\n")
        _OUTF.write("TESTTESTTESTTESTTEST\n")
_rv=AAtypetable(filelist=['_Ptest123.fas','_Ptest456.fas'], outputfile="_table_test789.txt
_fe=os.path.isfile("_table_test789.txt")
```

```
assert _fe, "Cannot find output file - has it been created?"
os.remove("_table_test789.txt")
os.remove("_Ptest123.fas")
os.remove("_Ptest456.fas")
assert type(_rv) is type([]), "Function should return a list; it returns %r" % _rv
print("OK")
```

```
OK
```

Question 4: Clustering proteins based on aminoacid usage

Question 4.a: Computing the Euclidean distance

Write a function called `distance` that takes two lists of `float` of equal length as its arguments. The function should return the Euclidean distance between the two lists, considered as vectors in a space of dimensionality equal to the length of the lists. That is to say, it should compute the element by element difference between the two lists, square that, sum all the squares and take a square root of the final sum (see the equation here). Your function should raise a `DimensionalityException` if the two lists passed are not of the same length, or if they are both empty; otherwise it should return a `float` .

In [29]:
```
# Run this cell to define the exception
class DimensionalityException(Exception):
    pass
```

In [30]:
```
# Your code here

def distance(lista, listb):

    alen=len(lista)
    blen=len(listb)
    if alen!=blen:
        raise DimensionalityException ('Lists not same length')
    elif alen==0 or blen==0:
        raise DimensionalityException ('Lists empty')

    #squared difference between points
    sqdiff=0
    # zip into iterable tuples
    it=zip(lista,listb)
    for a,b in it:
        sqdiff += (float(a) - float(b))**2

    # sq root of sum of squared difference
    ed = sqdiff**0.5
    return ed
```

In [31]:
```
# Test code
_dist=distance([1.0, 2.0], [1.5, -1.0])
assert type(_dist) is type(0.0), "Return value is not a float: %r" % _dist
print("OK")
```

```
OK
```

Question 4.b: Reading data in tabular format

Write a function called `readTable` that takes a string called `filename` as its argument. The function should read from the file specified in the argument a TSV table formatted as described at the end of Question 3 (you do not necessarily have to answer Question 3 in order to do this; you can create one such table containing arbitrary values using an editor). The function should return a dictionary with the filenames as its keys; the value associated to each key should be a list of floats, containing in order the percentages of polar, small and hydrophobic AAs, as listed in the table. So for instance, if the argument file contains the table used as an example in Question 3, `readTable` should return a dictionary containing at least the following items:

```
{ "Pxxx.fasta": [0.52, 0.22, 0.33],  "Qx3Z.fas": [0.47, 0.35, 0.38],
... }
```

In [32]:

```
# Your code here

def readTable(filename):

    # Read the entire file
    # Turn new line to tab
    # Split into list at tabs
    extract = open(filename).read().replace('\n','\t').split('\t')

    # Now group them 4 at a time to produce sublists
    sub = [extract[i:i+4] for i in range(0, (len(extract)), 4)]

    # Remove unwanted sublists
    for i in sub:
        if i ==['']:
            sub.remove(i)
        if '# Filename' in i:
            sub.remove(i)

    # Turn back into single list - remove sublists
    clean = []
    for i in sub:
        if type(i)==str:
            i=[i]
        clean += i

    # Turn into a dictionary with the first string every 4
    # being the key for a list of the following three
    dictionary = {clean[i]: clean[i+1:i+4] for i in range(0, len(clean), 4)}
    return dictionary
```

In [33]:

```
# Test code
import os
with open("_testTable123.tsv", "wt") as OUTF:
    OUTF.write("# Filename\tPolar\tSmall\tHydro\n")
    OUTF.write("Pxxx.fas\t0.52\t0.22\t0.33\n")
    OUTF.write("Pyyy.fas\t0.47\t0.35\t0.38\n")
_tab=readTable(filename="_testTable123.tsv")
assert type(_tab) is type({}), "Return value is not a dict: %r" % _tab
assert ("Pxxx.fas" in _tab) and ("Pyyy.fas" in _tab), "Return dict missing keys"
os.remove("_testTable123.tsv")
print("OK")
```

```
OK
```

Question 4.c: Computing a distance matrix

Write a function called `distanceMatrix` that takes as its argument two strings called `inputfile` and `outputfile`, used to pass the names of the input and output file. The function should read from the input file a TSV table created as specified at the end of Question 3 (again, you do not necessarily have to answer Question 3 in order to do this; you can create a table with made-up data using an editor). Suppose data for N proteins are read. Your function should then create an output TSV file containing a matrix of size NxN (plus one row and one column for labels). The rows and columns of the matrix should correspond to individual protein file names, so that each matrix entry corresponds to a pair of proteins, say Pxxx and Pyyy. Such entry should contain the distance between the `[polar, small, hydro]` lists (considered as 3 dimensional vectors) corresponding to Pxxx and Pyyy. If done correctly, your matrix should be symmetric and have zeroes on the diagonal. Format the matrix with row and column labels as in the following 3-protein example (that contains random numbers and filenames):

```
# Filename      Pxxx.fas  Pyyy.fas  Pzzz.fas
Pxxx.fas        0.00      2.34      1.51
Pyyy.fas        2.34      0.00      1.82
Pzzz.fas        1.51      1.82      0.00
```

The exact alignment is not important, as long as the data are separated by tabs and the first line starts with a hash (`#`). Likewise, the order in which the proteins are listed is not important, provided the same order is used along the rows and along the columns. Do not add any quotes or other spurious characters. Your function should not return any value.

In [34]:

```
# Your code here

def distanceMatrix(inputfile, outputfile):

    # Produce a dictionary with filename as key
    # and ratios as values
    inp=readTable(inputfile)

    # Create list of ratios of the dict
    lst=[]
    for i in inp.values():
        lst+=i

    # Create sub lists for each file
    slst=[lst[i:i+3] for i in range(0,len(lst),3)]

    # Create dictionary where each filename corresponds to
    # a list of distances
    dict3={}
    for i in inp.keys():
        dict3[i]=[]
    for i in dict3.keys():
        for a in slst:
            dict3[i]+=[distance(inp[i], a)]

    # Create lists of of dictionary3
    dlist = []
    for key, val in dict3.items():
        dlist.append([key] + val)

    # Write the TSV file using the list/sublists
    with open(outputfile, 'wt') as f:
        # Add the '#' filename
        head='# Filename'+'\t'
        f.write(head)
```

```
        # Write the column headers
        l=len(inp.keys())
        n=0
        for i in inp.keys():
            if n<(l-1):
                f.write(i+'\t')
                n+=1
            elif n==(l-1):
                f.write(i+'\n')

        # Write in the rows
        count=0
        for a in dlist:
            for i in a:
                if count <(l):
                    f.write(str(i) + '\t')
                    count+=1
                elif count ==(l):
                    f.write(str(i) + '\n')
                    count-=l
```

In [35]:

```
# Test code
import os
import os.path
with open("_testTable123.tsv", "wt") as OUTF:
    OUTF.write("# Filename\tPolar\tSmall\tHydro\n")
    OUTF.write("Pxxx.fas\t0.52\t0.22\t0.33\n")
    OUTF.write("Pyyy.fas\t0.47\t0.35\t0.38\n")
_rv=distanceMatrix(inputfile="_testTable123.tsv", outputfile="_testTable123out.tsv")
_fe=os.path.isfile("_testTable123out.tsv")
assert _fe, "Cannot find output file - has it been created?"
with open("_testTable123out.tsv", "rt") as INF:
    _fl=INF.readline()
assert _fl[0]=='#', "First line of output file does not start with a #"
os.remove("_testTable123.tsv")
os.remove("_testTable123out.tsv")
assert type(_rv) is type(None),"Function should not return any value; it returns %r" % _rv
print("OK")
```

```
OK
```

An application of this type of analysis

Now that you have this code running, try downloading a few FASTA sequences for the antifreeze proteins of cold water fish. Then download the sequences of other families of related proteins (eg Hemoglobin, Insulin and P53) from other species. Process these sequences according to Questions 3 and 4. Have a look at the resulting distance matrix - is any of these groups of proteins characterised by a distinctive pattern of usage of aminoacids? NOTE: no marks are assigned for this section.

Examiners

Dr Fabrizio Smeraldi *(f.smeraldi@qmul.ac.uk)*; Prof Conrad Bessant *(c.bessant@qmul.ac.uk)*

In [41]:

```
exec(open("../marking.py").read())
```

In [42]:

```
mark()
```

```
.............................../tmp/ipykernel_9024/1185055892.py:8: ResourceWarning: unc
```

```
losed file <_io.TextIOWrapper name='test138hnb-13876.txt' mode='r' encoding='UTF-8'>
  extract = open(filename).read().replace('\n','\t').split('\t')
ResourceWarning: Enable tracemalloc to get the object allocation traceback
./tmp/ipykernel_9024/1185055892.py:8: ResourceWarning: unclosed file <_io.TextIOWrapper na
me='test354hfy-37596.txt' mode='r' encoding='UTF-8'>
  extract = open(filename).read().replace('\n','\t').split('\t')
ResourceWarning: Enable tracemalloc to get the object allocation traceback
.
--------------- Marking Q1a -------------
> Total marks for Q1a: 5 out of 5
--------------- Marking Q1b -------------
> Total marks for Q1b: 5 out of 5
--------------- Marking Q1c -------------
> Total marks for Q1c: 5 out of 5
--------------- Marking Q1d -------------
> Total marks for Q1d: 5 out of 5
--------------- Marking Q1e -------------
> Total marks for Q1e: 5 out of 5
--------------- Marking Q2a -------------
> Total marks for Q2a: 8 out of 8
--------------- Marking Q2b -------------
> Total marks for Q2b: 5 out of 5
--------------- Marking Q2c -------------
> Total marks for Q2c: 5 out of 5
--------------- Marking Q2d -------------
> Total marks for Q2d: 4 out of 4
--------------- Marking Q2e -------------
> Total marks for Q2e: 3 out of 3
--------------- Marking Q3a -------------
> Total marks for Q3a: 5 out of 5
--------------- Marking Q3b -------------
> Total marks for Q3b: 8 out of 8
--------------- Marking Q3c -------------
> Total marks for Q3c: 12 out of 12
--------------- Marking Q4a -------------
> Total marks for Q4a: 7 out of 7
--------------- Marking Q4b -------------
List elements not float
List elements not float
List elements not float
> Total marks for Q4b: 5 out of 8
--------------- Marking Q4c -------------
> Total marks for Q4c: 10 out of 10
-----------------------------------------
------------- Marking summary -------------
-----------------------------------------
Question 1:     25 marks out of 25
Question 2:     25 marks out of 25
Question 3:     25 marks out of 25
Question 4:     22 marks out of 25
-----------------------------------------
Total marks: 97 out of 100
----------------------------------------------------------------------
Ran 35 tests in 0.021s

OK
```

In [ ]: