```
class ProteinParam :
# These tables are for calculating:
#     molecular weight (aa2mw), along with the mol. weight of H2O (mwH2O)
#     absorbance at 280 nm (aa2abs280)
#     pKa of positively charged Amino Acids (aa2chargePos)
#     pKa of negatively charged Amino acids (aa2chargeNeg)
#     and the constants aaNterm and aaCterm for pKa of the respective termini
#  Feel free to move these to appropriate methods as you like

# As written, these are accessed as class attributes, for example:
# ProteinParam.aa2mw['A'] or ProteinParam.mwH2O

    aa2mw = {
        'A': 89.093,  'G': 75.067,  'M': 149.211, 'S': 105.093, 'C': 121.158,
        'H': 155.155, 'N': 132.118, 'T': 119.119, 'D': 133.103, 'I': 131.173,
        'P': 115.131, 'V': 117.146, 'E': 147.129, 'K': 146.188, 'Q': 146.145,
        'W': 204.225,  'F': 165.189, 'L': 131.173, 'R': 174.201, 'Y': 181.189
        }

    mwH2O = 18.015
    aa2abs280= {'Y':1490, 'W': 5500, 'C': 125}

    aa2chargePos = {'K': 10.5, 'R':12.4, 'H':6}
    aa2chargeNeg = {'D': 3.86, 'E': 4.25, 'C': 8.33, 'Y': 10}
    aaNterm = 9.69
    aaCterm = 2.34

    def __init__ (self, protein):
        protein=protein.upper()
        self.protein=''.join([aa for aa in protein if aa in ProteinParam.aa2mw])
        #protein will convert to uppercase
        self.proteinDict={aa: self.protein.count(aa) for aa in ProteinParam.aa2mw}
        #cleans the input string to include only valid amino acids and stores a
count dictionary of each of them
        pass

    def aaCount (self):
        #return the total number of valid amino acids in the input sequence
        return sum(self.proteinDict.values())
        pass

    def pI (self):
        #search for pH value with a net charge of approximately 0 using binary
search and using it to calculate the isoelectric point rounded to two decimal
places
        low=0.0
        high=14.0
        while (high-low)>0.01:
            mid=(high+low)/2
            if self._charge_(mid)>0:
                low=mid
            else:
                high=mid
        return round((high+low)/2,2)
        pass

    def aaComposition (self) :
        #Return the composition that we already calculated in init
        return self.proteinDict
```

```
        pass

    def _charge_ (self, pH):
        #return the net charge of the protein at a specific pH
        #calculates it by using the pKa values of charged amino acids and terminal
groups
        pos_charge=0.0
        neg_charge=0.0
        for aa, pKa in self.aa2chargePos.items():
            pos_charge+=self.proteinDict[aa]*(10**pKa)/(10**pKa+10**pH)
        for aa, pKa in self.aa2chargeNeg.items():
            neg_charge+=self.proteinDict[aa]*(10**pH)/(10**pKa+10**pH)
        #add terminal charges
        pos_charge+=(10**self.aaNterm)/(10**self.aaNterm+10**pH)
        neg_charge+=(10**pH)/(10**self.aaCterm+10**pH)
        return pos_charge-neg_charge
        pass

    def molarExtinction (self, cystine=True):
        #calculate and return the molar extinction coefficient using Y, W, and C if
cystine is true
        extinction=self.proteinDict['Y']*self.aa2abs280['Y']+\
                    self.proteinDict['W']*self.aa2abs280['W']
        if cystine:
            extinction+=self.proteinDict['C']*self.aa2abs280['C']
        return extinction
        pass

    def massExtinction (self):
        #return the mass extinction coefficient which is molar extinction divided
by molecular Weight, and returns 0.0 of the molecular weight is 0
        myMW =  self.molecularWeight()
        return self.molarExtinction() / myMW if myMW else 0.0

    def molecularWeight (self):
        #return the total molecular weight of the sequence and accounts for water
loss
        total_mass=sum(self.aa2mw[aa]*count for aa, count in
self.proteinDict.items())
        num_peptide_bonds=self.aaCount()-1
        return total_mass-(num_peptide_bonds*self.mwH2O) if self.aaCount()>0 else
0.0
        pass

# Please do not modify any of the following.  This will produce a standard output
that can be parsed

import sys
def main():
    inString = input('protein sequence?')
    while inString :
        myParamMaker = ProteinParam(inString)
        myAAnumber = myParamMaker.aaCount()
        print ("Number of Amino Acids: {aaNum}".format(aaNum = myAAnumber))
        print ("Molecular Weight: {:.1f}".format(myParamMaker.molecularWeight()))
        print ("molar Extinction coefficient:
{:.2f}".format(myParamMaker.molarExtinction()))
        print ("mass Extinction coefficient:
{:.2f}".format(myParamMaker.massExtinction()))
```

```
        print ("Theoretical pI: {:.2f}".format(myParamMaker.pI()))
        print ("Amino acid composition:")

        if myAAnumber == 0 : myAAnumber = 1  # handles the case where no AA are
present

        for aa,n in sorted(myParamMaker.aaComposition().items(),
                           key= lambda item:item[0]):
            print ("\t{} = {:.2%}".format(aa, n/myAAnumber))

        inString = input('protein sequence?')

if __name__ == "__main__":
    main()


class NucParams:
    rnaCodonTable = {
    # RNA codon table
    # U
    'UUU': 'F', 'UCU': 'S', 'UAU': 'Y', 'UGU': 'C',  # UxU
    'UUC': 'F', 'UCC': 'S', 'UAC': 'Y', 'UGC': 'C',  # UxC
    'UUA': 'L', 'UCA': 'S', 'UAA': '-', 'UGA': '-',  # UxA
    'UUG': 'L', 'UCG': 'S', 'UAG': '-', 'UGG': 'W',  # UxG
    # C
    'CUU': 'L', 'CCU': 'P', 'CAU': 'H', 'CGU': 'R',  # CxU
    'CUC': 'L', 'CCC': 'P', 'CAC': 'H', 'CGC': 'R',  # CxC
    'CUA': 'L', 'CCA': 'P', 'CAA': 'Q', 'CGA': 'R',  # CxA
    'CUG': 'L', 'CCG': 'P', 'CAG': 'Q', 'CGG': 'R',  # CxG
    # A
    'AUU': 'I', 'ACU': 'T', 'AAU': 'N', 'AGU': 'S',  # AxU
    'AUC': 'I', 'ACC': 'T', 'AAC': 'N', 'AGC': 'S',  # AxC
    'AUA': 'I', 'ACA': 'T', 'AAA': 'K', 'AGA': 'R',  # AxA
    'AUG': 'M', 'ACG': 'T', 'AAG': 'K', 'AGG': 'R',  # AxG
    # G
    'GUU': 'V', 'GCU': 'A', 'GAU': 'D', 'GGU': 'G',  # GxU
    'GUC': 'V', 'GCC': 'A', 'GAC': 'D', 'GGC': 'G',  # GxC
    'GUA': 'V', 'GCA': 'A', 'GAA': 'E', 'GGA': 'G',  # GxA
    'GUG': 'V', 'GCG': 'A', 'GAG': 'E', 'GGG': 'G'  # GxG
    }
    dnaCodonTable = {key.replace('U','T'):value for key, value in
rnaCodonTable.items()}

    def __init__ (self, inString=''):
        self.nucComp={}
        self.codonComp={}
        self.aaComp={}
        if inString: #if string is given, call addSequence method
            inString=inString.upper().strip() #removes white space and capitalizes
all input
            self.addSequence(inString)
        pass

    def addSequence (self, inSeq):
        nucString=''.join(inSeq.upper().split()) #removes white spaces and
capitalizes all input
        rnaString=nucString.replace('T', 'U') #replaces all 'T' with 'U' for RNA
string
```

```
        for nuc in nucString: #initializes nucleotide counts
            if nuc in 'ACGTU':
                self.nucComp[nuc]=self.nucComp.get(nuc, 0)+1

      #iterating from position 0 to end of rnaString
        for index in range(0, len(rnaString)-2, 3):
            codon=rnaString[index:index+3]

        #sets codon to 3 letter increments
            if 'N' in codon or len(codon)!=3:
                continue
        #codon counts
            self.codonComp[codon]=self.codonComp.get(codon, 0)+1
        #amino acid counts
            aa=self.rnaCodonTable.get(codon, '-')
            self.aaComp[aa]=self.aaComp.get(aa, 0)+1
        pass
    def aaComposition(self):
        return self.aaComp
    def nucComposition(self):
        return self.nucComp
    def codonComposition(self):
        return self.codonComp
    def nucCount(self):
        return sum(self.nucComp.values())


import sys
class FastAreader :
    '''
    Define objects to read FastA files.

    instantiation:
    thisReader = FastAreader ('testTiny.fa')
    usage:
    for head, seq in thisReader.readFasta():
        print (head,seq)
    '''
    def __init__ (self, fname=None):
        '''contructor: saves attribute fname '''
        self.fname = fname

    def doOpen (self):
        ''' Handle file opens, allowing STDIN.'''
        if self.fname is None:
            return sys.stdin
        else:
            return open(self.fname)

    def readFasta (self):
        ''' Read an entire FastA record and return the sequence header/sequence'''
        header = ''
        sequence = ''

        with self.doOpen() as fileH:

            header = ''
            sequence = ''
```

```
    # skip to first fasta header
    line = fileH.readline()
    while not line.startswith('>') :
        line = fileH.readline()
    header = line[1:].rstrip()

    for line in fileH:
        if line.startswith ('>'):
            yield header,sequence
            header = line[1:].rstrip()
            sequence = ''
        else :
            sequence += ''.join(line.rstrip().split()).upper()

yield header,sequence
```