ST1507 DATA STRUCTURES AND ALGORITHMS (AI)

ASSIGNMENT ONE (CA1)

~ Morse Code Message Analyzer ~

Project by:

Joshua Yap

2011857

DAAA/03

# Program:

## Description:

This Morse Code Message Analyzer contains 4 options, changing of the printing mode, conversion of the user input text into morse text, a way to analyse an encoded message from an input file and subsequently save it to an output file, and a way to exit and end the programme.

Text

Description automatically generated

### Option 1

#### How to use:

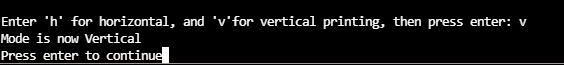
Input of 1, Change printing mode:

If option 1 is selected, the user is prompted with the following.

Text

Description automatically generated

The use will now be able to input their desired printing mode. Horizontal means that the morse is printed horizontally in one line, while vertical prints each morse letter vertically, top to down. Here, it changes the output based on the user input.



#### Error Check:

If the user inputs a wrong input, it will change the printing mode to the default mode of ‘horizontal’, and sends the user back to the options page, where the user can re-enter ‘1’ and change the printing mode again.

Text

Description automatically generated

### Option 2

#### Description

Option 2 is for printing from text that the user inputs and encodes it as morse. By default, it prints horizontally.

How to use:

Input a line of text that you want to encode

Example for Outputs:

Text

Description automatically generatedA picture containing background pattern

Description automatically generated

Limitations:

Input is limited to 1 line, ‘\n’ does not register to prints 2 row or more rows.

Special characters are not saved and instead are ignored.

### Option 3:

#### Description:

Option 3 reads a file containing morse, translate it and stores the frequency, locations, word, morse, and initial position into a pandas data frame. It then prints the message in its entirety, followed by the frequency, morse and position of each word, and finally a summarised message.

#### How to use:

The user is prompted to enter an input file (file with the morse), and an output file. If the output file does not exist, it will create one based on the naming convention input by the user.

Graphical user interface, text

Description automatically generated

Then, the programme will decode, group by frequency, produce the essential message, print the outputs, and then save it into the indicated output file:

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

#### Limitations:

Limited to saving as a horizontal print.

### Option 4:

Description  
option 4 exits the programme.

#### How to use:

Input ‘4’ when prompted in the main screen

Text

Description automatically generated

#### Limitations:

User will have to run the programme again manually after exiting through option 4

# Program Code:

## Files:

Text

Description automatically generatedFiles made are as shown here.

\_\_init\_\_.py allows for classes to be imported from other files.

ui.py is where the main programme is run, it imports files for the options.

For printing of the menu, ui.py also imports indexClass from index.py.

For option 1, ui.py imports the printModeMe function from index.py.

For option 2, ui.py imports function ‘encode’ from printsHorizontal.py or function ‘encode’ from printsVertical.py

For option 3, ui.py imports option3.py, which in turn imports essentialSort.py, frequencySort.py, wordCheck.py and printsLetter.py.

morse\_sos.txt is where the input morse coded message for option 3 is stored.

out.txt was where the output for one iteration of the code was stored.

stop\_words.txt is where words that are to be filtered out are stored to be used in essentialSort.py.

Some files will have 2 or more classes contained in them. The reason for this is that if more than 1 class is needed for a specific function, such as the storing of word, morse and location found within wordCheck.py, they will be placed together so the relevant code will be easier to change instead of moving between different files to fix bugs.

The reason for 2 files that sort the message for option 3 is due to the different requirements of printing the words based on frequency and the essential message.

For frequency, all words need to be stored, and printed in the format of ‘morse, leave line, word, frequency, locations’, while the essential message requires only the words to be printed in order of frequency. Thus, for ease of use of either sorting methods, 2 files dedicated to sorting will be used.

The sortedListF and sortedListE functions use the \_\_lt\_\_ taken from the class wordMessage or wordClass to compare two different word nodes. The word nodes inherit its values from wordCheck from wordCheck.py and stores the relevant ones for the class.

The ‘split’ function was overloaded and made to return a specific list for use in analyse.

Polymorphism is used in wordCheck, where if a word occurs once, a default value is assigned to frequency. Else, the returned frequency corresponds to the number of occurrences. It is also used here to store a set of words, corresponding morse, locations, and initial row and word position.

As the morse dictionary needs to be used in 3 different methods, it was stored separately in its own file. It is a dictionary of lists, where the key index is the letter or number, and the key value is a 2d list of the morse. For example, ‘s’ in the dictionary is stored as 'S': [['.','.','.']].

For option 3, after calling analyse and wordCheck, the output was stored as a pandas data frame.

## Class made and reasons

2 similar files, essentialSort.py and frequencySort.py were made. The Node class of both files are the same. As it is only 4 lines of code, dedicating a new file for it was not deemed necessary. The wordMessage and wordClass functions differ in that they store different values. wordClass stores the word, its frequency and the location in the row and column as a pair in a bracket while wordMessage stores the word, frequency, and absolute value of its position in the row and the nth word. Likewise, SortedListE and SortedListF differ in that SortedListE first filters words out.

The class wordCheck stores the word, frequency, location pair, morse, and position in the row and the column it is in. It works in tandem with the analyse class. If analyse detects that a word has already been added, it will activate addNewLocation in wordCheck, updating the locations and frequency.

In analyse, the code assumes that any word inserted is new, and performs a check by comparing to words in passedWords. If the word has already been passed before, the frequency is updated, and the new location is added. It then updates newEntry to be false, so that a new set will not be added using this word.

## Problems faced and respective solutions:

As option 2 requires 2 different printing methods, a way was needed to store the morse and letter, then be able to be used in printing for either method. Thus, a dictionary containing lists was used. This is because a list allows for the storage of a key and a value together in the same index. For example, ‘S’ : [[‘.’, ‘.’ , ‘.’ ]]. Thus, to obtain the respective key using a value or value using a key is easy to implement. This results in a simplistic way to code the encoding to morse functions and the decoding method.

The values are contained as a list is because of the printing methods. For use in horizontal printing, a .join method was used to combine the list into a single string. In the case of ‘S’, it is ‘…’. For vertical printing, np.transpose was sufficient to change the morse to be top down.

However, the vertical printing posed another challenge. To combine 2 words, concatenate was used. Furthermore, as the morse stored in this method does not all have the same length, empty spaces must be manually added to the top of the transposed list. After doing so, the morse can be printed row by row.

As a while loop was used to determine whether the appended list is has the same number of rows as the new morse letter or the new morse is the same length as the message, it can vary in time for the programme to run. If all the morse has the same length, it will have a complexity of 0. If the morse has varying levels of length, it can have a complexity of O(n^2). A method where the morse stored is of the same length was considered, but ultimately rejected as it could not dynamically change the number of rows the output would have when inputting a new morse character.

If the printing mode the user input was invalid, it will need to be changed back to a default state. The printing mode will also need a default state since the user may not change the printing mode. Thus, the printing mode was initialised as horizontal and would default back to horizontal if a bad input was passed.

Lastly, for analysis in option 3, a way was needed to store the word, corresponding morse, locations appeared, and frequency. Locations refers to the position in the row of text and the row the word appears in. Morse of an entire word was also required to be stored after being translated from morse. To solve this, the morse was converted to text, thus creating the first part of the analysis. For the storing of the values stated before, the word was translated back into morse, and stored with the relevant word.

To store location, enumerate was needed to store position in the row or the row it was in as an integer. Splitting by row, then by each word helps facilitate this enumeration. As enumeration returns a tuple of ‘integer, word’, tuple indexing was used to pass either the word or the integer as needed. Then, a check will be initiated at every word passed in. If the word appeared previously, only the location would be passed. Else, it would pass the word and position to wordClass. The passing of the new location would then increment the frequency.

For the sorting by initial position for use in essential message, the initial position of the word was also stored in wordCheck.

The values then needed to be stored together in rows, and only some values had to be obtained at a time for use in later in option 3. Which means fetching of 2 different sets of information from this storage was required. Thus, a pandas data frame was utilised as it allowed a set of values to be stored and relevant values can be retrieved easily.

# Appendix:

## essentialSort.py

class Node:

    # Constructor

    def \_\_init\_\_(self):

        self.nextNode = None

class wordMessage(Node):

    #required information

    def \_\_init\_\_(self,word,frequency,coorX,coorY):

        self.word = word

        self.freq = frequency

        self.coorX = coorX

        self.coorY = coorY

        super().\_\_init\_\_()

    def \_\_lt\_\_(self, otherNode):

        if otherNode == None:

            raise TypeError('\'<\' not supported between instances of \'AnalyzeWord\' and \'NoneType\'')

        #comparison checks for the essential message

        elif self.freq == otherNode.freq:

            if self.coorX == otherNode.coorX:

                return self.coorY < otherNode.coorY

            else:

                return self.coorX < otherNode.coorX

        return self.word < otherNode.word

    #output per word node

    def \_\_str\_\_(self):

        s= f'{self.word}'

        return s

class sortedListE:

    def \_\_init\_\_(self):

        self.headNode = None

        self.currentNode = None

    #cheks for first node, filters out if in stop\_words.txt

    def \_\_appendToHead(self, newNode):

        if  self.headNode.word not in filterList:

            oldHeadNode = self.headNode

            self.headNode = newNode

            self.headNode.nextNode = oldHeadNode

    #insert function

    def insert(self, newNode):

        if newNode.word not in filterList:

            # If list is currently empty

            if self.headNode == None:

                self.headNode = newNode

                return

            # Check if it is going to be new head

            if newNode > self.headNode:

                self.\_\_appendToHead(newNode)

                return

            #goes to next nodes

            leftNode = self.headNode

            rightNode = self.headNode.nextNode

            while rightNode != None:

                if newNode == rightNode:

                    if newNode < rightNode:

                        leftNode.nextNode = newNode

                        newNode.nextNode = rightNode

                        return

                elif newNode < rightNode:

                    leftNode.nextNode = newNode

                    newNode.nextNode = rightNode

                    return

                leftNode = rightNode

                rightNode = rightNode.nextNode

    # When there is one wordSet remaining, it will have to be added to the end

            leftNode.nextNode = newNode

    #printing of the output

    def \_\_str\_\_(self):

        # We start at the head

        output =""

        node= self.headNode

        firstNode = True

        output=[]

        while node != None:

            if firstNode:

                output = "\'"+node.\_\_str\_\_()

                firstNode = False

            else:

                output += (' ' + node.\_\_str\_\_())

            node= node.nextNode

        return output+"\'"

#initialises the sort using a for loop

class startSort:

    def sort(wordSets,l):

        for wordSet in wordSets:

            l.insert(wordMessage(wordSet[0],wordSet[1],wordSet[2],wordSet[3]))

        return(l)

import os

#initialises the classes needed for determining essntial message

class startEssentialClass:

    def startEssential(listThings):

        #opens the stop words file

        here = os.path.dirname(os.path.abspath(\_\_file\_\_))

        filename = os.path.join(here, 'stop\_words.txt')

        f = open(filename, "r")

        stopWords=f.read().upper()

        #makes the list of stop words global, so it does not need to be manually passed into the above classes

        global filterList

        filterList=stopWords.split('\n')

        #initialises SortedList

        l = sortedListE()

        #starting sort

        filteredMessage=startSort.sort(listThings,l)

        return filteredMessage

## frequencySort.py

from essentialSort import Node

class wordClass(Node):

    #initialise values

    def \_\_init\_\_(self, word, morseWord, locations,freq):

        self.word = word

        self.freq=freq

        self.locations=locations

        self.morseWord=morseWord

        super().\_\_init\_\_()

    #checks if words have different frequencies, then their length, then their alphabet order

    def \_\_lt\_\_(self, otherNode):

        if otherNode == None:

            raise TypeError('\'<\' not supported between instances of \'AnalyzeWord\' and \'NoneType\'')

        elif self.freq == otherNode.freq:

            if len(str(self.word)) == len(str(otherNode.word)):

                return str(self.word).upper() < str(otherNode.word).upper()

            else:

                return len(str(self.word)) < len(str(otherNode.word))

        return self.freq > otherNode.freq

    def \_\_str\_\_(self):

        s= f'{self.morseWord}\n {self.word} ({self.freq}) {self.locations}\n'

        return s

class sortedListF:

    def \_\_init\_\_(self):

        self.headNode = None

        self.currentNode = None

    def \_\_appendToHead(self, newNode):

        oldHeadNode = self.headNode

        self.headNode = newNode

        self.headNode.nextNode = oldHeadNode

    def insert(self, newNode):

        # If list is currently empty

        if self.headNode == None:

            self.headNode = newNode

            return

        # Check if it is going to be new head

        if newNode < self.headNode:

            self.\_\_appendToHead(newNode)

            return

        #goes to next nodes

        leftNode = self.headNode

        rightNode = self.headNode.nextNode

        while rightNode != None:

            if newNode == rightNode:

                if newNode < rightNode:

                    leftNode.nextNode = newNode

                    newNode.nextNode = rightNode

                    return

            elif newNode < rightNode:

                leftNode.nextNode = newNode

                newNode.nextNode = rightNode

                return

            leftNode = rightNode

            rightNode = rightNode.nextNode

    # When there is one wordFreq left, it will have to be added to the end

        leftNode.nextNode = newNode

    def \_\_str\_\_(self):

        # We start at the head

        output =""

        node= self.headNode

        firstNode = True

        while node != None:

            if firstNode:

                output = f"Words with frequency: {node.freq}\n"+node.\_\_str\_\_()

                firstNode = False

            else:

                output += (' ' + node.\_\_str\_\_())

            currentNode=node

            node = node.nextNode

            if node != None:

                if currentNode.freq>node.freq:

                    output+=(f"Words with frequency: {node.freq}\n")

        return output+"\'"

#inserts each word and its corresponding information

class startSortF:

    def sort(wordFreqs,l):

        for wordFreq in wordFreqs:

            l.insert(wordClass(wordFreq[0],wordFreq[1],wordFreq[2],wordFreq[3]))

        return(l)

class startWord:

    #initialises sortedListF

    def startWordSort(listThings):

        l = sortedListF()

        filteredMessage=startSortF.sort(listThings,l)

        return filteredMessage

## index.py

class indexClass():

    #menu layout + obtaining user input

    def mess():

        print('\*'\*57, '\n\*\t ST1507 DSAA: MorseCode Message Analyzer\t\*\n\*', '-'\*53, '\*\n\*',

              ' '\*53, '\*\n\*\t Done by: Joshua Yap', ' '\*26, '\*\n\*\t Class: DAAA/03', ' '\*31, '\*')

        print('\*'\*57, '\n\*\nPlease Select your choice (\'1\',\'2\',\'3\',\'4\'):',

              '\n\t1.Change Printing Mode', '\n\t2.Convert Morse Code to Text',

              '\n\t3.Analyze Morse Code Message', '\n\t4.Exit')

        inputNum = input('Enter your choice: ')

        return inputNum

    #for print mode, returns the letter and the full word of the print mode

    def printModeMe(printMode):

        while printMode != 'h' or printMode != 'v':

            if str(printMode) == 'h':

                printModeLong = 'Horizontal'

                printMode='h'

                return printMode, printModeLong

            elif str(printMode) == 'v':

                printModeLong = 'Vertical'

                printMode='v'

                return printMode, printModeLong

            else:

                print('invalid input, printing mode will change to default')

                printModeLong = 'Horizontal'

                printMode='h'

                return printMode, printModeLong

## morseDict.py

# dictonary for letter/number to morse

morse = {

        'A': [['.','-']],

        'B': [['-','.','.','.']],

        'C': [['-','.','-','.']],

        'D': [['-','.','.']],

        'E': [['.']],

        'F': [['.','.','-','.']],

        'G': [['-','-','.']],

        'H': [['.','.','.','.']],

        'I': [[ '.','.']],

        'J': [['.','-','-','-']],

        'K': [['-','.','-']],

        'L': [['.','-','.','.']],

        'M': [['-','-']],

        'N': [['-','.']],

        'O': [['-','-','-']],

        'P': [['.','-','-','.']],

        'Q': [['-','-','.','-']],

        'R': [['.','-','.']],

        'S': [['.','.','.']],

        'T': [['-']],

        'U': [['.','.','-']],

        'V': [['.','.','.','-']],

        'W': [['.','-','-']],

        'X': [['-','.','.','-']],

        'Y': [['-','.','-','-']],

        'Z': [['-','-','.','.']],

        '1': [['.','-','-','-','-']],

        '2': [['.','.','-','-','-']],

        '3': [['.','.','.','-','-']],

        '4': [['.','.','.','.','-']],

        '5': [['.','.','.','.','.']],

        '6': [['-','.','.','.','.']],

        '7': [['-','-','.','.','.']],

        '8': [['-','-','-','.','.']],

        '9': [['-','-','-','-','.']],

        '0': [['-','-','-','-','-']]

         }

## option3.py

from printsLetter import decodeMorse

from wordCheck import analyse

from essentialSort import startEssentialClass

from frequencySort import startWord

import pandas as pd

import os

here = os.path.dirname(os.path.abspath(\_\_file\_\_))

#base code for option3

class option3Class():

    def option3Code(inFile, outName):

        decoded=decodeMorse.decodePrep(inFile)

        stringList=''

        #for decoded message

        print('\n'\*2,'\*'\*3,'Decoded Morse Text')

        print(decoded)

        stringList+=decoded

        #for analysis

        outPutCheck=analyse.getFrequency(decoded)

        my\_list = list(map(lambda x: x.split(' '), outPutCheck))

        df = pd.DataFrame(my\_list[:-1], columns =['morse', 'Word', 'Frequency', 'Locations','Initial X', 'Initial Y'])

        #for frequency

        dfWords=df[['Word','morse','Locations','Frequency']]

        WordsList=dfWords.values.tolist()

        frequency=startWord.startWordSort(WordsList)

        print(frequency)

        stringList+=str(frequency)

        #for essential

        dfCoor=df[['Word','Frequency', 'Initial X', 'Initial Y']]

        listCoor=dfCoor.values.tolist()

        messageE=startEssentialClass.startEssential(listCoor)

        print('\n'\*2,'\*'\*20,'\nEssential Message')

        print(messageE)

        stringList=stringList+'\n \*\*\*Essential Message'+'\n'+str(messageE)

        #for output

        currentLocation=os.path.join(here,outName)

        f= open(currentLocation,"w+")

        f.write(stringList)

## printsHorizontal.py

from morseDict import morse

# reading file that has message to be encoded

class option2h():

 # encoding function

    def encode(data):

        encodedWord = ''

        mCodedLine=[]

        #ensures that the user input is in all capitals

        data=data.upper()

        #split

        test = [line.split() for line in data.split("\n")]

        #iterates through row

        for row in test:

            mCoded = []

            #iterates through each word

            for word in row:

                wordSplit=list(word)

                #iterates through letter in word

                for letter in wordSplit:

                    if letter.isalpha():

                        morseLetter = ''.join(str(item) for innerlist in morse[letter.upper()] for item in innerlist)

                        mCoded.append(morseLetter)

                    elif letter.isdigit():

                        morseLetter = ''.join(str(item) for innerlist in morse[letter] for item in innerlist)

                        mCoded.append(morseLetter)

                mCoded.append(' ')

            mCodedLine.append((','.join(mCoded)).strip(', '))

        encodedWord='\n'.join(mCodedLine)#as each record in mCodedLine is one line, need to combine them by \n

        return encodedWord

## printsLetter.py

import os

import re

from morseDict import morse

# reading file that has message to be encoded

here = os.path.dirname(os.path.abspath(\_\_file\_\_))

# decoding part, opeing file

class decodeMorse():

    def decodePrep(fileInput):

        filename = os.path.join(here, fileInput)

        file = open(filename, 'r')

        content = file.read()

        #input coded message here, splits string by the ',' and '\n', keeps them after seperation

        morseTxt = re.split('([,|\n])', content)

        #the line above seperates the string with seperators ',' or '\n' in this case

        noC = list(filter(lambda a: a != ',', morseTxt)) #removes ',', since not needed

        new=[]

        l=0

        #checks if it is a possible morse

        for l in enumerate(noC):

            if [list(noC[l])] in morse.values():

                new.append([list(noC[l])])

            else:

                new.append(noC[l])

            l+=1

        return decodeMorse.decode(new)

    # decoding function

    def decode(data):

        encodedWord = ''

        for letter in data:

            if letter in morse.values():#checks if letter is in the values

                indexL=list(morse.values()).index(letter) #index of matching value

                encodedLetter = list(morse.keys())[indexL] #key of index obtained above

                encodedWord += encodedLetter

            else: #in the event of a \n, , ect, it will be added into the decoded message here

                encodedWord += str(letter)

        return encodedWord

## printsVertical.py

import numpy as np

from morseDict import morse

# encoding function

class option2v():

    def encode(data):

        data=data.upper()

        mCoded=[[]]

        #initialises the first letter and correspinding morse for appending later

        if data[0:1] in morse.keys():

            morseLetter =  np.transpose(morse[data[0:1]])

            mCoded=morseLetter.copy()

        #combining the morse

        for letter in data[1:]:

            #checks if it is a valid letter/number

            if letter in morse.keys():

                morsedLetter=np.transpose(morse[letter])

                #checks if same length

                if len(mCoded) != len(morsedLetter):

                    #checks if the message has fewer rows

                    if len(mCoded) < len(morsedLetter):

                        blank = np.empty((1, len(mCoded[1])), dtype='str')

                        blank[:] = ' '

                        while len(mCoded) != len(morsedLetter):

                            mCoded = np.append(blank, mCoded, axis=0)

                    #checks if the new letter has fewer rows

                    elif len(mCoded) > len(morsedLetter):

                        blank = np.empty((1, 1), dtype='str')

                        blank[:] = ' '

                        while len(morsedLetter)!=len(mCoded):

                            morsedLetter=np.append(blank,morsedLetter, axis=0)

            #replaces bad inputs with a space

            else:

                morsedLetter=np.empty((len(mCoded),1), dtype='str')

                morsedLetter[:] = ' '

            #conceating the morse to the side instead of to the bottom

            mCoded = np.concatenate((mCoded, morsedLetter), axis=1)

        return print('\n'.join([''.join(['{:2}'.format(item) for item in row])

                        for row in mCoded]))

## ui.py

from printsVertical import option2v

from printsHorizontal import option2h

from option3 import option3Class

from index import indexClass

import os

##########################################

# main programme

pMode = 'h'

#Initial printing of the main page

num=indexClass.mess()

while num != 4:

    try:

        num = int(num)

        #checks for input

        if num > 0 and num < 4:

            #initialising the print mode

            pModeOut = indexClass.printModeMe(pMode)

            pMode=pModeOut[0]

            printMode=pModeOut[1]

            #option 1, print mode

            if num == 1:

                print(f'Current mode is {pMode}\n')

                pMode = input('Enter \'h\' for horizontal, and \'v\''

                                  'for vertical printing, then press enter: ')

                #printModeMe returns a tuple. which corresponds to the letter or a whole word

                pModeOut = indexClass.printModeMe(pMode)

                pMode=pModeOut[0]

                printMode=pModeOut[1]

                print(f'Mode is now {printMode}')

                input('Press enter to continue')

            #option 2, checks for priting mode, default is h

            elif num == 2:

                message=input('Type the message you want to convert to morse code:\n')

                if pMode=='v':

                    outputMorse=option2v.encode(message)

                #option2h requires manually printing after calling encode

                elif pMode=='h':

                    outputMorse=option2h.encode(message)

                    print(outputMorse)

                input('Press enter to continue')

            elif num == 3:

                inputFile=input('Enter input file (please include .txt at the end): ')

                #obtains current diectory location and checks for input file

                here = os.path.dirname(os.path.abspath(\_\_file\_\_))

                inputExists=os.path.join(here,inputFile)

                #input file check

                if os.path.exists(inputExists):

                    #output file does not need a check as it can be manually created

                    outputFile=input('Enter output file (.txt will be added automacially): ')

                    outputFile=str(outputFile)+'.txt'

                    l=option3Class.option3Code(inputFile, outputFile)

                    input('Press enter to continue')

                else:

                    print('Bad input, file does not exist')

            num = indexClass.mess()

        elif num != 4:

            print('error, input a valid number')

            num = indexClass.mess()

    except ValueError:

        print('error, input a valid number')

        num = indexClass.mess()

print('Bye, thanks for using ST1507 DSAA: MorseCode Message Analyzer')

## wordCheck.py

from printsHorizontal import option2h

class wordCheck():

    #values that are stored

    def \_\_init\_\_(self, word, coorX, coorY, frequency=1):

        self.\_\_word = word

        self.\_\_morse=option2h.encode(self.\_\_word)

        self.\_\_frequency = frequency

        #brackets are placed here for printing later

        self.\_\_locations = [(coorX, coorY)]

        #for use in essential message, storing the first occurence of the word

        self.\_\_initialX=coorX

        self.\_\_initialY=coorY

    #if a word occurs more than once, this will be called

    def addNewLocation(self, coorX, coorY):

        self.\_\_frequency = self.\_\_frequency+1

        self.\_\_locations.append((coorX, coorY))

    #custom split

    def split(self,x):

        return [self.\_\_morse, self.\_\_word, self.\_\_frequency,

        self.\_\_locations,self.\_\_initialX,self.\_\_initialY]

        #to check if a word as occured more than once

    def getWord(self):

        return self.\_\_word

class analyse():

    def getFrequency(message):

        message=message.upper()

        message = message.split("\n")

        passedWords = []

        #loop considers the coorX (xth row), and coorY (yth position)

        for coorX in enumerate(message):

            #splits the row into words

            row = coorX[1].split(" ")

            for coorY in enumerate(row):

                #assumes that the new entry is true initially

                newEntry = True

                #checks if word has occured before

                for i in passedWords:

                    if coorY[1] == i.getWord():

                        #adds the new location

                        i.addNewLocation(coorX[0], coorY[0])

                        newEntry = False

                        break

                #if word has not occured prior

                if newEntry == True:

                    passedWords.append(wordCheck(coorY[1], coorX[0], coorY[0]))

        return passedWords