Assignment 3: Transform

CompSci 101 Fall 2021

Due: October 21, 2021

Overview

This assignment is about understanding how to transform data from one format to another. You'll need to use the starter code from this zip file and create a project from it. The zip file is linked here. Download the zip file, unzip the file, move files to desired location, then go into

Pycharm and select *File* > *Open…* then select the unzipped folder. Choose either New Window or This Window.

Note: You'll have two modules, ***Transform.py*** and ***Vowelizer.py***, as well as a folder data holding files you can use to encrypt or decrypt. When you run the module Transform, new data files may be created.

Learning Goals

1. Understanding code you did not write that uses libraries.
2. Processing files.
3. Using global variables.
4. Practicing string manipulation.
5. Practice the 7-steps, where steps 1-4 are the harder part.

Completing the Assignment

Note that Part 0 does not have anything to submit.

- **Part 0:** Run the module ***Transform.py*** which uses the module ***Vowelizer.py***. Make sure you understand how these modules work together by answering the questions below.
    - Questions similar to these could appear on exams, but you don't need to submit your answers as part of this assignment.
- **Part 1:** Complete the module ***PigLatin.py***, test its functions within that module, then use the ***Transform.py*** module to create Pig-Latinized files.
- **Part 2:** Complete the module ***CaesarCipher.py***, test its functions within that module, then use the ***Transform.py*** module to create encrypted and decrypted files.
- **(Optional Challenge) Part 3:** Complete the module ***Shuffleizer.py*** and demonstrate that encrypt and decrypt work. This will be described at the very end of this document.

Part 0: Transform

You'll use the starter code for this assignment to understand how the different modules fit together. The first part is to understand how the modules fit together by running the program, reading the program, and answering questions.

Your goal is to run the program in ***Transform.py*** to see how it works by transforming words by removing vowels and then trying to restore the words after vowels have been removed. This uses the ***Vowelizer.py*** module you're given.

Module Transform.py

OBJECTIVE

After reading this description and running the ***Transform.py*** module you're given, you should be able to answer these questions. You don't need to submit your answers as part of this assignment, but you should be able to answer questions similar to these on exams.

1. What is the conceptual difference between **`Vowelizer.encrypt`** and **`Vowelizer.decrypt`**?
2. Which of the **`Vowelizer.encrypt`** and **`Vowelizer.decrypt`** functions runs more quickly and why?
3. When is the value **`NT_WORD`** returned by **`Vowelizer.decrypt`**?
4. How is **`Vowelizer.encrypt`** called from the ***Transform.py*** module?
5. When does **`decrypt(encrypt(w)) == w`**, i.e., for what words w does this happen?

DESCRIPTION

Transform.py reads a file of words that the user chooses when the program is run and transforms that file using the following steps (which you can see by reading the code in the module):

A. The user chooses a file using the FileDialog to pick a file.
B. The **`doTransform`** function calls the function **`readFileAsLines`** to read the file chosen by the user to get a list of line-lists, where each line-list represents a line in the file read.
C. Every word in each line is transformed, by calling the function passed to **`doTransform`** that returns a transformed-version of its word parameter.
D. When every word in every line has been transformed, the lines are written to another file, one with a suffix specified by a parameter to **`doTransform`**. The file is written using the function **`writeFileAsLines`**.

EXAMPLE

For example, to remove vowels from each word and create a file with an ***-nvw*** suffix, this code in the main section of Transform works (this is the code you start with).

```
doTransform("-nvw", Vowelizer.encrypt)
```

For example, here are the first ten lines of **`data/twain.txt`**, Mark Twain's ***The Notorious Jumping Frog of Calaveras County:***

```
The Notorious Jumping Frog of Calaveras County

In compliance with the request of a friend of mine, who wrote me
from the East, I called on good-natured, garrulous old Simon Wheeler,
and inquired after my friend's friend, Leonidas W. Smiley, as requested
to do, and I hereunto append the result. I have a lurking suspicion
that Leonidas W. Smiley is a myth; that my friend never knew
such a personage; and that he only conjectured that if I asked
old Wheeler about him, it would remind him of his infamous Jim
Smiley, and he would go to work and bore me to death with some
```

Here are the ten lines of **`data/twain-nvw.txt`**, the no-vowel version of the same file as produced by calling the **`encrypt`** function of the Vowelizer.py module. Note that each word in

this transformed file has no vowels. This is because the function **`Vowelizer.encrypt`** returns a version of the string parameter it's passed without vowels.

```
Th Ntrs Jmpng Frg f Clvrs Cnty

n cmplnc wth th rqst f  frnd f mn, wh wrt m
frm th st,  clld n gd-ntrd, grrls ld Smn Whlr,
nd nqrd ftr my frnd's frnd, Lnds W. Smly, s rqstd
t d, nd  hrnt ppnd th rslt.  hv  lrkng sspcn
tht Lnds W. Smly s  myth; tht my frnd nvr knw
sch  prsng; nd tht h nly cnjctrd tht f  skd
ld Whlr bt hm, t wld rmnd hm f hs nfms Jm
Smly, nd h wld g t wrk nd br m t dth wth sm
```

Reading this no-vowel file and trying to transform it back to normal using the call below from the main block in Transform.py doesn't quite produce the original:

```
doTransform("-rvw", Vowelizer.decrypt)
```

Here's the decrypted version created by choosing **`twain-nvw.txt`** using the call above to create **`twain-nvw-rvw.txt`** (the nvw is "no-vowel" and the rvw is "re-vowel"):

```
oath antares jumping fargo fe cleavers county

ainu compliance with oath request fe friend fe NT_WORD who wart aim
farm oath NT_WORD called ainu NT_WORD NT_WORD aloud samoan NT_WORD
aeneid enquired after amy NT_WORD NT_WORD landis NT_WORD NT_WORD as
requested
at NT_WORD aeneid hornet append oath NT_WORD have lurking suspicion
tahiti landis NT_WORD seemly as NT_WORD tahiti amy friend never knew
such NT_WORD aeneid tahiti ah inlay conjectured tahiti fe asked
aloud whaler abate NT_WORD at wailed remained ham fe haas infamous jaime
NT_WORD aeneid ah wailed age at work aeneid bar aim at death with sam
```

Notice that some words in this version are the same as in the original, but others are not at all correct and some are missing. Be sure you can answer the questions at the beginning of this section before continuing. (You are not required to submit your answers to these questions.)

Part 1: PigLatin.py

Background

One example of very simple data transformation is translating an English word into Pig Latin. If you are not familiar with Pig Latin, you can read more about it here. A comprehensive list of

rules for translating English to Pig Latin is provided below. If you consider yourself a Pig Latin pro, you might consider using this search engine interface.

Implementing the Module

You must create a module named ***PigLatin.py***. Make sure you create it with a main program block. In this module, you'll write functions **`encrypt`** and **`decrypt`** as described below which should mirror the conceptual functionality of these functions in ***Vowelizer.py***. You must test these functions in the main program block of ***PigLatin.py***. Once you're confident that the functions work, you can import the module into ***Transform.py*** and test **`PigLatin.encrypt`** and **`PigLatin.decrypt`** on files.

Follow the broad steps below in implementing this module. In writing these functions, it will be ***very beneficial*** to write helper functions (e.g. **`isVowel`** or **`isAllVowels`** or **`firstVowelIndex`**).

1. Using the rules for creating Pig Latin below, write a function **`encrypt`** so that **`encrypt(w)`** returns the Pig Latin form of any string **`w`**.
    a. Test **`encrypt`** by calling it from the main block in ***PigLatin.py*** and printing the return value for several different strings.
    b. Make sure you've covered upper and lower case letters, as well as all the vowel cases described in the rules for Pig Latin.
2. Using the same rules, but in reverse, write a function **`decrypt`** so that **`decrypt(w)`** returns a best-guess at the original word that generated the Pig Latin string **`w`**.
    a. For example, **`decrypt("uke-Day")`** should return **`"Duke"`** and **`decrypt("apple-way")`** could return either **`"wapple"`** or **`"apple"`** depending on how you write the code -- both are correct, but your code should be consistent in how it treats these cases.
    b. Test **`decrypt`** by calling it from the main block in ***PigLatin.py*** and printing the return value for several different strings.

It should be true that **`encrypt(decrypt(w)) == w`** if **`w`** is in Pig Latin form. But, as you've seen in the example above, it's possible that **`decrypt(encrypt(w))`** will be different from **`w`** for some English words: Consider ***"wand"*** and ***"and"***, which are both encrypted/Pig-Latinized as ***"and-way"***, or ***"weasel"*** and ***"easel"***, which are both encrypted as ***"easel-way"***.

When you submit this module, make sure that the code you used to test your functions is still in the main program block.

Rules for Pig Latin

Follow these rules to convert a word into Pig Latin. We're using a hyphen to facilitate translating back from Pig Latin to English. In creating Pig Latin you will **not** be concerned with punctuation, so treat every character as either a vowel or non-vowel; punctuation falls into the second category. ***The rules below describe cases with lowercase characters, but your code must work with uppercase and lowercase letters.*** In the rules below, 'A' is a vowel just as 'a' is, for example. Your code should not change the case of any letters.

1. If a word begins with a vowel: 'a', 'e', 'i', 'o', or 'u', then append the string "-way" to form the Pig Latin equivalent.

Word	Pig Latin Version
anchor	anchor-way
oasis	oasis-way
umbrella	umbrella-way
AWOL	AWOL-way

2. If a word begins with a non-vowel (we will call this a consonant, but it could be a number, a punctuation mark, or something else), move the prefix ***before the first vowel*** to the end with lowercase "ay" appended. Use a hyphen and treat 'y' as a vowel in these cases. If 'y' is the first letter of a word, it should then be considered a consonant.

word	Pig Latin Version
computer	omputer-cay
yesterday	esterday-yay
STRENGTH	ENGTH-STRay
my	y-may
rhythm	ythm-rhay
"always!"	always!"-"ay

3. Words that begin with 'qu' should be treated as though the 'u' is a consonant. Note that not all of these words have a vowel immediately after 'qu'.

word	Pig Latin Version
quiz	iz-quay
queue	eue-quay
quay	ay-quay
quran	an-quray

4. ***If a word contains no vowels, it should be treated as though it starts with a vowel.*** For example, "zzz" will be encrypted to "zzz-way".

5. ***It is possible that different words will be transformed to the same Pig Latin form.*** For example, "it" encrypts to "it-way", but "wit" also encrypts to "it-way" using the rules above.

On very rare occasions, a word may not conform to these rules. If you find such a word, post on Piazza to confirm the proper encryption.

Running Transform with PigLatin Functions

When you've got these functions working, you should test them by importing the ***PigLatin*** module into ***Transform*** (see how ***Vowelizer*** is imported). Then encrypt the provided file **`data/twain.txt`** to create **`data/twain-pig.txt`**. Then use this Pig-Latinized file and the **`decrypt`** function to create **`data/twain-pig-upg.txt`**. You should create these with the code below in the main program block of ***Transform.py*** (on two different runs, not at the same time).

```
doTransform("-pig", PigLatin.encrypt)
doTransform("-upg", PigLatin.decrypt)
```

You'll submit these two files when you're submitting code and text files for this assignment.

Part 2: Caesar Cipher

Background

Encryption is a very common form of data transformation in which data is encoded using a "key". Ideally, only someone with the key can decrypt the encryption to access the original data, which protects it from anyone who doesn't have the key. This part of the assignment focuses on the Caesar Cipher encryption technique. To encode a string, each letter in the string is "shifted" by the same designated number of places. The shift or rotation number is the "key" because you can only decrypt encoded text if you know how much all the letters were shifted by.

The example provided here is from the Wikipedia page for Caesar Cipher and uses a shift or rotation of 23 (Wikipedia calls this a right shift -- we'll use only that kind of shift here). Note that if A is the 0th character and Z the 25th, then X is the 23rd character of the alphabet.

```
Plain text:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher Text:     XYZABCDEFGHIJKLMNOPQRSTUVW
```

To encrypt a word or a phrase, you look up each letter of the word in the plain text string and use the corresponding ciphertext letter to encrypt. To encrypt the string "QUICK", you'd look up the Q, find N; look up the U, find R; and so on --- so that "QUICK" is encrypted as "NRFZH".

Implementing the Module

You must create a module named ***CaesarCipher.py***. Make sure you create it with a main program block. In this module, you should create a function **`encrypt`** as described below that mirrors the conceptual functionality of this function in ***Vowelizer.py***. You'll want to test this function, and any helper functions you create, in the main program block of ***CaesarCipher.py***. When you're confident that the function works, you can import the module into ***Transform.py*** and test **`encrypt`** on files.

You're also required to implement a function **`setShift`** that will be called from the ***Transform.py*** module to set the values of global variables used in **`encrypt`**. You'll also call this function from within the ***CaesarCipher.py*** module. You'll also write a function **`findShift`** described below for help in finding the shift of an already encrypted file. Follow the broad steps below in implementing the module.

Making **`encrypt`** Work

1. Create *five global variables*. These should be defined at the top of the module, before any functions. Use the code below for these, the values of these variables will change when you call the function **`setShift`**. Note that with the code below, the value of global variable **`shifted_upper`** will be "DEFGHIJKLMNOPQRSTUVWXYZABC".

```
shift = 3
lower_alph = "abcdefghijklmnopqrstuvwxyz"
upper_alph = lower_alph.upper()
shifted_lower = lower_alph[3:] + lower_alph[:3]
shifted_upper = upper_alph[3:] + upper_alph[:3]
```

2. Implement the function **`encrypt`** so that the call **`encrypt(w)`** returns a version of w encrypted using a Caesar Cipher based on the values of the global variables defined above, e.g., by a shift of 3 -- but in your code, use only **`lower_alph`** and the other string variables in implementing encrypt. The general idea is:
    a. If the character is a lowercase letter, use **`lower_alph.index(ch)`** to find the index where the character occurs. Then use this index to find the encrypted character in **`shifted_lower`**. For example, if **`lower_alph.index('h')`** evaluates to 7 and **`shifted_lower[7]`** is **`'k'`**, then **`'h'`** encrypts to **`'k'`**.
    b. Use this idea to encrypt the string parameter by encrypting each letter and creating a new (encrypted) string to return.
    c. ***Your function should only encrypt alphabetic letters (uppercase and lowercase). Other characters, including spaces, should just "pass through" unchanged.*** For example, "Hat 7" would encrypt to "Kdw 7".
3. Test the **`encrypt`** function by calling it from the main program block of the ***CaesarCipher.py*** module and printing the value returned by **`encrypt`** for several strings with encryptions that you can verify by hand.
4. Implement the function **`setShift`** to set the values of the three global variables **`shift`**, **`shifted_lower`**, and **`shifted_upper`** depending on the int parameter to

setShift. For example, the call **setShift(3)** should set the variables to what they are when the program starts. The call **setShift(6)** should set **shifted_lower** to the string "ghijklmnopqrstuvwxyzabcdef". You can use the logic that's used in the example shown above in Step 1. You must define the three variables as global within the function **setShift**. See **Vowelizer.decrypt** for an example of using global variables.

5. Test your **setShift** and **encrypt** functions by calling them several times with different inputs and printing the results in the main program block.

Decrypting Files that Have Been Encrypted

You won't need to implement a **decrypt** function, since decryption of an encrypted word is done instead by another encryption with a different (complementary) shift. Note that "Hat 7" encrypts to "Kdw 7" with a shift of 3, as described above. Encrypting "Kdw 7" with a shift of 23 yields "Hat 7", the original string. So, you can test whether **setShift** works by encrypting with a given shift, then "decrypting" the result by encrypting again using a complementary shift. For example, the code below will print "jolbk" "zebra" in the only print statement executed.

```
setShift(10)
ew = encrypt("zebra")
setShift(16)
w = encrypt(ew)
print(ew,w)
```

Implement a function **findShift** that takes a string of CaesarCipher-encrypted words as a parameter and ***returns the shift that was originally used to encrypt the words*** (not the shift you used to decrypt them). You'll do this by using brute-force to try all possible 26 shifts (0-25) and choosing the shift that yields the most actual words. The idea behind this is described in more detail in the section on “eyeball”/brute-force decryption below.

For example, the call **findShift("Zkdw grhv wkh ira vdb?")** should return 3, indicating that a shift of 3 was used to create the words in the string parameter **"Zkdw grhv wkh ira vdb?"**. Using this string and a shift of 23, the call **encrypt("Zkdw grhv wkh ira vdb?")** returns "What does the fox say?" This is because 3 and 23 complement each other as do 10 and 16 in the example above.

Use the following steps/ideas for implementing **findShift**.

1. First, read all the words in **data/lowerwords.txt** (use **os.path.join()** for the autograder; see example below) and store the words in a list and/or set local variable in the body of the **findShift** function.
    a. To create the filename, use the following code:

```
import os.path
file = os.path.join("data","lowerwords.txt")
f = open(file)
```

   b. You may need to use the **`.strip()`** method on the strings before putting them in a list/set in order to make sure there is no whitespace left over from the file. You can do this with a list comprehension:

      ```
      wordsClean = [w.strip() for w in f.read().split()]
      ```

      See the function **`loadlower()`** in Vowelizer.py for another example, but you don't need to use global variables here like **`loadlower()`** does.

2. Next, call **`setShift`** for all 26 possible shifts (0-25) and encrypt all the "words" in the string parameter passed to **`findShift`**. You can use any approach to do this, but using a list comprehension makes this simple to code. Consider, for example, **`[encrypt(w) for w in st.split()]`** where **`st`** is the string parameter.
3. For each shift in Step 2, find out how many of the encrypted strings are real words. One simple approach is to find the set intersection of each list of encrypted "words" from Step 2 with the set of real words read from **`lowerwords.txt`** in Step 1. The shift that yields the largest set intersection is the one that decrypts the text. From this number, determine and ***return the shift that was originally used to encrypt the text***.
   a. The note on "eyeball" decryption explains checking encrypted strings for words.
4. Test your **`findShift`** function by calling it repeatedly, passing previously-encrypted strings as parameters.

Running Transform with CaesarCipher Functions

When you've got these functions working, you should test them by importing the ***CaesarCipher*** module into Transform (see how ***Vowelizer*** and ***PigLatin*** are imported). Then write code and run the ***Transform.py*** module to encrypt the file **`data/twain.txt`** into **`data/twain-csr.txt`** ***using a shift of 18***. You'll need to call **`CaeserCipher.setShift`** before calling **`doTransform`** **(see below).**

You should use the lines and select the file **`data/twain.txt`** to generate the file you'll submit, **`twain-csr.txt`**:

```
CaesarCipher.setShift(18)
doTransform("-csr", CaesarCipher.encrypt)
```

You'll submit **`twain-csr.txt`** when submitting code and text files for this assignment.
To ensure that your **`findShift`** function works correctly, you should verify that a shift of 18 was used to encrypt **`twain.txt`** above. The output of the code below should be 18:

```
file = os.path.join("data","twain-csr.txt")
f = open(file)
st = f.read()
shift = CaesarCipher.findShift(st)
print(shift)
```

Once **`findShift`** passes the test above, you should use it to determine the shift used in creating **`data/decryptme-csr.txt`**. When you've discovered this value, you must subtract it from 26 and use that shift to decrypt the file and create **`decryptme-csr-uncsr.txt`**, ***which***

you will submit. If you open the properly-decrypted file within Pycharm, you'll see a quote by Donald Knuth regarding Computer Science.

Submitting and Grading

Checklist Before Submitting

1. Make sure you can answer the questions for Part 0 in understanding the ***Transform.py*** module. You won't submit anything, but there could be exam questions based on these.
2. You will submit six files total: two modules and four text files. The two modules are ***PigLatin.py*** and ***CaesarCipher.py***. The four text files are **`twain-pig.txt`**, **`twain-pig-upg.txt`**, **`twain-csr.txt`**, and **`decryptme-csr-uncsr.txt`**.
    a. ***PigLatin.py***:
        i. Make sure you have **`encrypt`** and **`decrypt`** functions working and in the module's main program block (and only the main program block), there is code you used to test those functions.
    b. ***CaesarCipher.py***:
        i. Make sure that **`setShift`**, **`encrypt`**, and **`findShift`** are working and that the code you wrote to test these functions is in the main program block. When using the **`lowerwords.txt`** file, make sure to use **`os.path.join()`** for the autograder's **`findShift`** test case.
    c. **`twain-pig.txt`** and **`twain-pig-upg.txt`:**
        i. Submit the files **`twain-pig.txt`** and **`twain-pig-upg.txt`** created by using ***Transform*** and ***PigLatin*** together.
    d. **`twain-csr.txt`** and **`decryptme-csr-uncsr.txt`:**
        i. Submit **`twain-csr.txt`** created using a shift of 18.
        ii. Submit **`decryptme-csr-uncsr.txt`** created after finding the shift used to create it.
3. **Each function you write must have a docstring comment describing what it does.**
4. You should have a comment with your name at the top of each module you submit.

This assignment is out of 40 points.

Autograded portion [31 points]

- [12] ***PigLatin.py***
    - [6] **`encrypt`** function takes one string as its parameter and correctly encrypts test strings
    - [6] **`decrypt`** function takes one string as its parameter and correctly decrypts test strings
- [14] ***CaesarCipher.py***
    - [2] **`setShift`** function behaves correctly and takes one int parameter
    - [6] **`findShift`** function behaves correctly and takes a string of encrypted words as parameter and returns the correct shift

    - [6] **`encrypt`** behaves correctly and takes a string parameter
- [5] Submitting .txt files and correctness of CaesarCipher .txt files
    - [1] Submitted all .txt files
    - [2] Correctness of **`twain-csr.txt`**
    - [2] Correctness of **`decryptme-csr-uncsr.txt`**

Hand-graded portion [9 points]

- [2] Quality of main program code
    - [1] Main program block in ***PigLatin.py*** contains calls to **`encrypt`** and **`decrypt`** as evidence of testing your functions
    - [1] Main program block in ***CaesarCipher.py*** contains calls to **`setShift`**, **`encrypt`**, and **`findShift`** as evidence of testing your functions
- [4] Correctness of PigLatin .txt files
    - [2] **`twain-pig.txt`**
    - [2] **`twain-pig-upg.txt`**
- [3] Rest of submission
    - [3] points for docstring comments/name

Submission Instructions

1. Log in to Gradescope.
2. Under the CompSci 101 dashboard, click Assignment 3.
3. Click Submit Programming Assignment, click “Click to browse”, and select the required files. Click “Browse Files” again to select and submit more files.
4. Once all files are selected click Upload.

You can submit more than once. Your last submission will be graded.

“Eyeball” or Brute-Force Decryption

Finding the shift used to encrypt a string can be done by trying all possible shifts and seeing which one yields the most real words. For example, using all twenty-six shifts of the string

```
st = "Bxvncrvnb rc'b njbh cx lxdwc oaxv 1-10, kdc wxc jufjhb"
```

with this code

```
for sh in range(26):
    setShift(sh)
    print(sh, encrypt(st))
```

results in this output:

```
0  Bxvncrvnb rc'b njbh cx lxdwc oaxv 1-10, kdc wxc jufjhb
1  Cywodswoc sd'c okci dy myexd pbyw 1-10, led xyd kvgkic
2  Dzxpetxpd te'd pldj ez nzfye qczx 1-10, mfe yze lwhljd
3  Eayqfuyqe uf'e qmek fa oagzf rday 1-10, ngf zaf mximke
```

```
4  Fbzrgvzrf vg'f rnfl gb pbhag sebz 1-10, ohg abg nyjnlf
5  Gcashwasg wh'g sogm hc qcibh tfca 1-10, pih bch ozkomg
6  Hdbtixbth xi'h tphn id rdjci ugdb 1-10, qji cdi palpnh
7  Iecujycui yj'i uqio je sekdj vhec 1-10, rkj dej qbmqoi
8  Jfdvkzdvj zk'j vrjp kf tflek wifd 1-10, slk efk rcnrpj
9  Kgewlaewk al'k wskq lg ugmfl xjge 1-10, tml fgl sdosqk
10 Lhfxmbfxl bm'l xtlr mh vhngm ykhf 1-10, unm ghm teptrl
11 Migyncgym cn'm yums ni wiohn zlig 1-10, von hin ufqusm
12 Njhzodhzn do'n zvnt oj xjpio amjh 1-10, wpo ijo vgrvtn
13 Okiapeiao ep'o awou pk ykqjp bnki 1-10, xqp jkp whswuo
14 Pljbqfjbp fq'p bxpv ql zlrkq colj 1-10, yrq klq xitxvp
15 Qmkcrgkcq gr'q cyqw rm amslr dpmk 1-10, zsr lmr yjuywq
16 Rnldshldr hs'r dzrx sn bntms eqnl 1-10, ats mns zkvzxr
17 Sometimes it's easy to count from 1-10, but not always
18 Tpnfujnft ju't fbtz up dpvou gspn 1-10, cvu opu bmxbzt
19 Uqogvkogu kv'u gcua vq eqwpv htqo 1-10, dwv pqv cnycau
20 Vrphwlphv lw'v hdvb wr frxqw iurp 1-10, exw qrw dozdbv
21 Wsqixmqiw mx'w iewc xs gsyrx jvsq 1-10, fyx rsx epaecw
22 Xtrjynrjx ny'x jfxd yt htzsy kwtr 1-10, gzy sty fqbfdx
23 Yuskzosky oz'y kgye zu iuatz lxus 1-10, haz tuz grcgey
24 Zvtlaptlz pa'z lhzf av jvbua myvt 1-10, iba uva hsdhfz
25 Awumbquma qb'a miag bw kwcvb nzwu 1-10, jcb vwb iteiga
```

By "eyeballing" this output, you should see that the shift of 17 yields the most real words. This means that a shift of 26-17 = 9 was originally used to encrypt the text. Note: The shift of 1 includes the word "led", but the shift of 17 yields many more words.

Optional Part 3: Challenge Transform

This module does not earn you any extra points, but it does represent a challenge for you to work on.

Create a module ***Shuffleizer.py*** in which you write both an **`encrypt`** and a **`decrypt`** function. The idea is to leave the first and last letters of a word in place while shuffling the other letters of the word.

For **`encrypt`**, turn a word like "wonder" into "wnoedr" or "wdnoer" where the first and last letters are the same, but the other letters have been randomly shuffled. The function **`random.shuffle()`** from the random library/module will shuffle a list in place, e.g., **`random.shuffle(letters)`** will shuffle a list named **`letters`**. You can use the function **`list`** to convert a string to a list, and then **`join`** to convert back to a string after shuffling.

For **`decrypt`**, use ideas like those in ***Vowelizer.py*** to find a word that could be that represented by the parameter to **`decrypt`**. Unlike the function in ***Vowelizer***, this function should, in theory, always be able to return a word. However, not every word in text files will necessarily be found in the lexicon of words in **`lowerwords.txt`** -- so you can return return the string **`"NT_WORD"`** if a word can't be found when **`decrypt`** is called. Note that some

encrypted words have more than one valid decryption; for example, “cmals” could be decrypted to “clams” or “calms”. In these cases, your **`decrypt`** function can return any valid decryption.

Hint for `decrypt`: You may find it useful to use the **`sorted`** function to compare an encrypted word with potential decryptions. Calling this function and passing it a string parameter returns a list of the letters in the string in lexicographical (alphabetical) order. For example, the function call **`sorted("rhythm")`** returns **`['h','h','m','r','t','y']`**.