CISS451: Cryptography and Computer Security
Assignment 3

Objectives:

- Perform frequency analysis on textual data.
- Implement the substitution cipher.
- Implement tools to attack the substitution cipher.

In this assignment you will be downloading web contents for text mining. Therefore you will be creating directories, moving in and out of directories, unzipping files, reading files, etc.

Python: running linux commands using os.system

There are several ways to run linux commands in python. The following is the simplest. Run this python program:

```
import os
os.system('ls -la')
os.system('cd ~')
os.system('ls -la')
```

You can run several linux commands together:

```
import os
os.system('ls -la; cd ; ls -la')
os.system('cd ~')
os.system('ls -la')
```

All the above runs the linux commands in your bash. It's important to remember that each `os.system` executes in a single bash shell. Once the command is done, your context is restored. For instance if you `cd` to a different directory in one os.sytem function call, then after the function call, you are back to your original directory again. Go to your home directory and run this program:

```
import os
os.system('ls -la') # list your home directory
os.system('cd /')   # go somewhere else
os.system('ls -la') # you are back to your home directory again
```

If you really want to execute a command in a different directory, you can do this:

```
import os
os.system('ls -la') # list your home directory
os.system('cd /; ls -la') # go somewhere else and execute linux command
```

Python: files

You can write a string to a regular disk file and read the contents of a file into a string.

```
s = "hello world"
f = open('abc.txt', 'w')    # 'w' -- write access
f.write(s)
f.write("\n1 2 3\n\n")
f.write("%s\n" % 42)
f.close()

f = open('abc.txt', 'r')    # 'r' -- read access
s = f.read()
f.close()
print(s)
```

Open `abc.txt` and check it out.

Python: capturing and analyzing output from bash shell commands

In general for a process (process = an executing program) there are at least three files attached to the process: stdin (input), stdout, and stderr. In C++ they appear as follows:

```
int x;
std::cin >> x;                     # reading stdin (put into x)
std::cout << "from stdout ...\n"; # writing to stdout
std::cerr << "from stderr ...\n"; # writing to stderr
```

The std::cin is a file stream of data where the data is from the keyboard (by default). The std::cout and std::err are file streams of data where the data is sent to your console window (by default).

What's the point of stderr? Sometimes you want to send correct output to stdout and error messages to stderr. (Although it's perfectly fine if you want to send error messages to stdout.) For instance

```
int x, y;
std::cin >> x >> y;
if (y == 0)
{
    std::err << "y cannot be 0\n";
}
else
{
    std::cout << x / y << '\n';
}
```

This is the same for python. In python

```
import sys
x = input()                           # read string from stdin (put into x)
print("from stdout ... 0")            # write to stdout
sys.stdout.write("from stdout ... 1") # write to stdout
sys.stderr.write("from stderr ...")   # write to stderr
```

There are times when you want to analyze the output from your bash shell. To do that you redirect the output to files and then read the files either manually or using python. This redirects stdout to the file stdout.txt and then prints stdout.txt:

```
import os
os.system('ls -ls > stdout.txt')
```

```
f = open('stdout.txt', 'r')
s = f.read()
print(s)
```

This redirects stderr to stderr.txt and then prints stderr.txt:

```
import os
os.system('less nosuchfile 2> stderr.txt')
f = open('stdout.txt', 'r')
s = f.read()
print(s)
```

You can capture both the stdout and stderr by doing this:

```
import os
os.system('less nosuchfile > stdout.txt 2> stderr.txt')
```

You can also *combine* stdout and stderr into *one* file::

```
import os
os.system('less nosuchfile 1&2> std.txt')
```

Exercise 0.1. Write a python program so that when you run it the program executes the shell command `"ls -la"`, collects the data, and displays the total number of bytes used by the regular files in the current directory (regular means for instance don't include `.` and `..`).

Many linux commands are actually implemented in python modules. These are mainly found in the `os` module. For instance if you really want to change your linux current working directory permanently, you can do this:

```
import os
os.chdir('/')
```

and after the program ends, you will notice that you are now in directory /. Therefore this is very different from

```
import os
os.system('/')
```

which will only change the current working directory temporarily while executing `os.system`.

In a python program if you want to temporarily change your working directory you do this:

```
import os
cwd = os.getcwd()     # store current working directory
print(cwd)
os.chdir('/home')     # change to /home
# ... do some system work in /home
os.chdir(cwd)         # go back to where you came from
```

Python: running linux commands using `subprocess`

For a more sophisticated method, you can read up on the `subprocess` module. Here's an example:

```
import subprocess
p = subprocess.Popen(["ls", "-la"],
                     stdout=subprocess.PIPE,
                     stderr=subprocess.PIPE,
                     universal_newlines=True)
stdout, stderr = p.communicate()
print("stdout: %s" % stdout)
print("stderr: %s" % stderr)
```

Note that the command `"la -la"` is now broken up as `["la", "-la"]`.

Python: traversing a directory

The following shows you how to traverse a directory recursively. Suppose you are in directory `bleh`. First create a directory call `blah`. In `blah` create lots of directories recursively and put regular text files in the directories. Now go back to `bleh` and run this:

```
import os
for a, b, c in os.walk('blah'):
    print("----")
    print("a: %s\n" % a)
    print("b: %s\n" % b)
    print("c: %s\n" % c)
```

This traverses your file system recursively starting at path `blah`. The variables `a,b,c` are

1. the current directory your program is in right now
2. the list of all directories in the current directory
3. the list of all regular files in the current directory

Run the program and read the output carefully.

The files you are downloading requires unzipping. The linux command to unzip a file is `unzip [filename]`.

Python: downloading web content

There are many ways to download a URI (a web content).

First try this python program. This version is for python2:

```
import urllib
url = 'http://news.yahoo.com'
f = urllib.urlopen(url) # a file
s = f.read()            # a string (i.e., the news.yahoo.com webpage)
print(s[0:100])         # print first few characters of webpage
```

Here's the python3 version:

```
import urllib.request
url = 'http://news.yahoo.com'
f = urllib.request.urlopen(url)
s = f.read()
print(s[0:100])
```

You can of course save the webpage:

```
import urllib
s = urllib.urlopen("http://news.yahoo.com").read()
f = open("news-yahoo.html", "w")
f.write(s)
f.close()
```

You can now open the web page downloaded from the web using a browser. Or you can open the file for processing:

```
f = open("news-yahoo.html", 'r')
s = f.read()
print(s[:300])
```

You can also use the `wget` utility in linux:

```
import os
url = 'http://news.yahoo.com'
os.system('wget %s -O index.html' % url)
f = file('index.html', 'r')
s = f.read()
```

```
print(s[0:50])
```

The web server might block your download request if the server suspects that you are running a program and not using a browser. In that case you have to fool the server into believing that for instance you might want to do this:

```
import urllib.request
url = "http://www.gutenberg.org/dirs/1/0/2/1/10214/10214.txt"
req = urllib.request.Request(
    url,
    data=None,
    headers={
        'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_3) '+\
                      'AppleWebKit/537.36 (KHTML, like Gecko) '+\
                      'Chrome/35.0.1916.47 Safari/537.36'
    }
)
f = urllib.request.urlopen(req)
data = f.read().decode('utf-8')
print(data[:500])
```

You can also set your user agent in wget.

Python: exceptions

When something goes terribly wrong, a Python program can stop:

```
for i in range(-5, 5):
    print("%s %s" % (i, 1 / i))
```

If you want to have control over such errors and want the program to continue you can catch the error (or exception) and prevent Python from crashing the program.

```
for i in range(-5, 5):
    try:
        print("%s %s" % (i, 1 / i))
    except:
        print("something went wrong ... let's continue")
```

Python: strings

You can cut out substrings from a string easily:

```
s = "hello world"
t = s[3:6]
print(t)
t = s[4:10]
print(t)
```

It's not shocking that you can compare and concatenate strings:

```
s = "hello world"
t = s[3:5]
u = s[7:10]
v = t + u
print(v)
```

You can check if a string is a substring of another:

```
s = "hello world"
t = "wor"
u = "war"
print(t in s)
print(u in s)
```

You can typecast an int value to a string like this:

```
i = 42
s = "The answer to life, universe, and everything is " + str(i)
```

You can also do this:

```
i = 42
s = "The answer to life, universe, and everything is %s" % i
print(s)
```

Also try this:

```
i = 42
s = "The answer to life, universe, and everything is %s ... %s" % (i, i)
print(s)
```

For finding contents in a string and regular expressions

For finding contents in a string, you can do this:

```
s = 'my name is john.'
name = '[NO NAME]'
i = s.index(' is ')     # i points to " is "
if i != -1:
    i += 4              # i points to after " is "
    j = s.index(".", i) # find "." in s after index i
    name = s[i:j]

print(name)
```

A more sophisticated method is to use regular expressions:

```
import re
s = 'my name is john.'
name = '[NO NAME]'
p = re.compile(' is ([a-z]*)\.')
srch = p.search(s)
name = srch.group(1)
print(name)
```

The search object might have several matching groups of strings. To understand this concept try this:

```
import re
s = 'my name is john.'
name = '[NO NAME]'
p = re.compile('( is )([a-z]*)(\.)')
srch = p.search(s)
print(srch.group(0))
print(srch.group(1))
print(srch.group(2))
print(srch.group(3))
```

If you have not heard of regular expressions, then you will learn more about it in CISS445 (Programming Languages) and CISS362 (Automata). You can google "python re" and read the documentation on re. Do NOT use any other websites not mentioned in this assignments. (This applies to all assignments. If you are not sure, you can always ask me. Otherwise it would be considered plagiarism.)

Python Programming: Substitution cipher

Here's some code useful for breaking the substitution cipher.

```python
"""
tool for breaking substitution cipher
"""
import string
WIDTH=50

def get_decryption_key(key):
    """ Compute decryption key from encryption key """
    t = []
    for k,v in key.items(): t.append((v,k))
    return dict(t)


def E(key, plaintext):
    t  = ''
    for c in plaintext: t += key.get(c, '?')
    return t


def D(key, ciphertext):
    t = ''
    decrypt_key = get_decryption_key(key)
    for c in ciphertext: t += (decrypt_key.get(c, '-'))
    return t

#=============================================================
# Some tools for decryption process
#=============================================================
def add(key, a, b):
    """ adds a->b to encryption key """
    if a in key.keys():
        print("add %s -> %s error: %s already set" % (a, b, a))
        return
    if b in key.values():
        print("add %s -> %s error: %s already used" % (a, b, b))
        return
    key[a] = b


def delete(key, a):
    """ delete a->? from encryption key """
    if a not in key.keys():
        print("delete %s error: %s is not set" % (a, a))
        return
    del key[a]


def pprint(ciphertext, key, width=WIDTH):
    print((width + 2) * "=")
    print("ENCRYPTION KEY")
    print("==============")
    i = 0
    for char in string.ascii_lowercase:
        print("%s -> %s" % (char, key.get(char, '?')), end='')
        i += 1
        if i % 5 == 0: print()

    print()
    print()

    print("MESSAGES (ciphertext on top)")
    print("============================")
    plaintext = D(key, ciphertext)
    while plaintext != '':
        ciphertext0, ciphertext = ciphertext[:width], ciphertext[width:]
        print(ciphertext0)
        plaintext0, plaintext = plaintext[:width], plaintext[width:]
        print(plaintext0)
        print()

    print((width + 2) * "=")


def cleanup(s):
    """ Keep only a-zA-Z """
    t = ""
    for c in s:
        if c.isalpha():
            t += c.lower()
    return t


def freq(s, length=1):
    f = {}
    for c in s:
        f[c] = f.get(c, 0) + 1
    return f


if __name__ == "__main__":

    ciphertext = """
asdfkajsdflasdjfhaldjfhaludoiqrwerqwuehnfdmcz
xcaweriuahfalkjdsvc
"""
    ciphertext = cleanup(ciphertext)

    key = {}

    print("***** TEST 1")
    pprint(ciphertext, key)

    print()
    print("***** TEST 2 ... after adding c->d, z->a ...")
    print()
    add(key, 'c', 'd')
    add(key, 'z', 'a')
    pprint(ciphertext, key)

    print()
    print("***** TEST 3 ... after deleting c->d ...")
    print()
    delete(key, 'c')
    pprint(ciphertext, key)

    print()
    print("***** TEST 4 ... after clear ...")
    print()
    key = {} # clear
    pprint(ciphertext, key)
```

Take note of the usage examples on how to use the `add` and `delete` functions and how to clear a dictionary (i.e., `key = {}`)

You want to also write some code to perform frequency analysis on the ciphertext.

Speeding up Python with concurrency

If you're doing serious hacking, you usually need some for of concurrency in order to speed things up. There are many ways to speed up Python. But the easiest is to take a look at two modules: threading and multiprocessing. Just google for

`python concurrency multiprocessing threading tutorial`

or something similar and you will find lots of examples. The names of the modules tells you that the multiprocessing module uses multiple processes while the theading module uses multiple threads.

(Concurrency is not the only way to speed up Python programs. Designing and writing algorithmically efficient code as well as choosing the correct data structures are still the most important. But assuming you have already done that, then concurrency can help. Besides concurrency, there are many special python interpreters that can run faster than the standard python interpreter. Also, there are tools that allow you to embed C/C++ code inside python.)

Save this as a python program and run it:

```
from multiprocessing import Process

def hw(i):
    print("hello world from %s" % i)

if __name__ == '__main__':

    for i in range(10):
        p = Process(target=hw, args=(i,))
        p.start()
```

Each function call to `hw` will result in `hw` executing as a separate process.

OK. No big deal.

Now try this:

```
from multiprocessing import Process
import random
random.seed()
import time

def hw(i):
    time.sleep(random.uniform(0.01, 1.0)) # to force each process to run
```

```
                                                  # with different times
    print("hello world from %s" % i)

if __name__ == '__main__':

    for i in range(10):
        p = Process(target=hw, args=(i,))
        p.start()
```

Get it?

Now here's a program that computes a list of squares by getting different processes to execute `appendsquare` to insert the square of `i` into the `squares` list:

```
import time, random; random.seed()
from multiprocessing import Process, Manager

def appendsquare(i, squares):
    time.sleep(random.uniform(0.01, 1.0))
    squares.append(i * i)
    print("appended %s" % (i * i))

if __name__ == '__main__':

    manager = Manager()
    squares = manager.list([])

    ps = []
    for i in range(20):
        p = Process(target=appendsquare, args=(i, squares))
        p.start()
        ps.append(p)

    # wait for all process p in ps to stop
    for p in ps:
        p.join()

    print("squares: %s" % squares)
```

If you want the processes to work on the same list, you have to obtain the list like the above: the above `squares` is shared by the processes `p`'s.

`p.join()` means the main program will wait for `p` to end before it ends. If you don't execute the loop containing `p.join()`, then the main program can stop earlier then some process, resulting in that process trying to access the `squares` in the main program which have already died. This process will then crash.

Read the documentation/tutorials on sharing data carefully. You might also want to study the threading module too.

If you have a very long running process, then it's a good idea to design your program to be able to restart itself from where it stopped earlier. Otherwise if your program crashes (example if you lost network connectivity) you would have to restart the whole program. This means that you should continually store intermediate computations.

Q1. [Frequency analysis]

The goal is to performance frequency analysis on all English ebooks from gutenberg.org. Frequency analysis is performed on the 1-gram, 2-gram, 3-gram, and 4-gram. First go ahead and open your web browser and visit gutenberg.org and look for at least 2-3 book in plain text format.

First go to https://www.gutenberg.org/ and look around. Next, here's a simple hacking exercise for you: How do you view the plain text version of the book with id 10001? Who's the author and what's the title? (Think creatively.)

The goal is to do text mining for frequencies using a program. (You can't possible do it "by hand".)

Note that you should remove non-alphabet and replace uppercase with lowercase before performing the frequency analysis. Note that for the string `abcde`, there are four 2-grams, i.e., `ab`, `bc`, `cd`, `de`.

For each file from gutenberg, you should create a file named `data.txt`. (I suggest you have one directory for each book from gutenberg.org.) Here's my `data.txt` for gutenberg.org's file for id 10001:

```
Language: English
Author: Lucius Seneca
Title: Apocolocyntosis
length: 39515
string: theprojectgutenbergebookofapocolocyntosisb ... snipped
e: 4767
t: 3593
o: 3309
a: 2967
i: 2715
... snipped ...
th: 1141
he: 914
er: 664
in: 596
an: 533
... snipped ...
the: 645
and: 259
ing: 214
his: 174
you: 167
... snipped ...
```

```
tion: 135
with: 113
uten: 98
tenb: 97
nber: 97
... snipped ...
```

The 1-gram is listed starting with the one with the highest frequency. This is followed by the 2-gram frequency. Etc. The number of entries grows. So limit each n–gram category to at most 1000 entries.

The frequency analysis for each text file must be done using a C++ program. Call the program `freq.cpp`. The whole process is done using a python program. (The python program execute the C++ program for the frequency analysis of each file.) Your C++ program should accept a command line argument which is the path of the text file to be analyzed. The frequency data is printed. You can then run the program like this:

```
./a.out gutenberg/10001/10001.txt > gutenberg/10001/data.txt
```

(assuming you compiled your `freq.cpp` to `a.out`) redirecting the output to the text file `gutenberg/10001/data.txt`. Some hints will be given in class. Of course your python program can execute this C++ program. (Your C++ program to perform the frequency analysis will be at least twice as fast as the python version.) You can access the command line argument in your C++ program if your program looks like this:

```
#include <iostream>

int main(int argc, char ** argv)
{
    std::cout << argv[1] << '\n';
    return 0;
}
```

(This is in the CISS350 notes.) You should use C++ STL classes such as `std::unordered_map` (for hashtables) and `std::vector` (for arrays). See CISS350 notes.

After all the books are individually analyzed, you create a file `all_data.txt` that computes the frequencies for all books and their probabilities. The format of the file is below:

```
length: 1000000000
e: 1000, 0.5
t: 900, 0.4
o: 800, 0.3
```

```
... snipped ...
th: 1000, 0.5
he: 900, 0.4
er: 800, 0.3
... snipped ...
the: 1000, 0.5
and: 900, 0.4
ing: 800, 0.3
... snipped ...
tion: 1000, 0.5
with: 900, 0.4
uten: 800, 0.3
... snipped ...
```

where for the 2-, 3-, 4-grams, only the top 40 are included.

Once you are done, write a program that prints your frequency analysis, i.e., write a program (named `main.py`) that print a string. The string is the contents of your `all_data.txt`.

You submit your

- `main.py` – a dummy program that prints your frequency analysis
- `download.py` – a program that downloads all zipped text files from gutenberg.org. Make sure your program avoids re-downloading the same file (unless you know the file was changed or the download was incomplete). You might want to be able to download based on some id, or some id range, or a starting id. You might want to filter so that you only keep the books that you want. Or you can have another program to perform the filtering.
- `freq.cpp` – a C++ program that performs frequency analysis on a single text file from gutenberg.org.
- `all.py` – collects up all the frequency data and creates `all_data.txt`.
- `gutenberg.py` – this is the main program that calls all the other programs.

The above is only a suggestion for the break down of your program. Note that another thing you need to do is to organize your file system for the downloads. You do not want to be messy and have a confusing file system for all gutenberg files/directories.

That's it.

Start early because this will take some time.

Spoilers on the next page.

As an aside, you should google for gutenberg.org's mirror websites. These are "volunteer" websites that help gutenberg host their books. You should download your ebooks by performing a round robin on the mirrors. This will prevent overloading any mirror websites. Also, make sure you slow down the rate of your download to prevent overloading the mirrors. There are others downloading their books. Plus if you overload the servers, they will block you. Another thing you should do is to include some delays so that you are not constantly hitting the mirrors:

```
import time
import random; random.seed()
time.sleep(random.randrange(5, 20))
```

Spoilers: Mining Gutenberg.org

Here's one suggestion. You can go to gutenberg.org and just download some ebooks. Look at the URL format and see if you figure out a format for the URLs. With that you might be able to run a loop over the possible URL.

Obviously, this is not the only way. See if you can figure it out. (Spoilers on next page.)

Spoilers again

Google "gutenberg.org robot".

Q2. [Conditional probability]

Answer this question: Although `q` is infrequent, we know that `u` follows `q` with a very high probability. In order words, instead of talking about the probability of `qu`, we can talk about the following fact:

Given `q`, what is the probability that the next character is `u`?

First write a program to analyze all your text files from gutenberg.org and compute the above probability.

[NOTE: You need to count all `q[x]` where `x` is any character and you also need a count of `qu`. Divide to get the required probability.]

Finally write a program, named `main.py`, to simply print this number. If the required probabilty is 0.5, you write

```
print(0.5)
```

in `main.py`. The name of the program that computes the frequency should be named `cond2gram.py` The program reads the two characters `q` and `u` from command line. Here's a template for your `cond2gram.py` to show you how to read command line arguments:

```
import sys
import sys
x = sys.argv[1]
y = sys.argv[2]
print(x, y)
```

Run this program like this in your shell:

```
python cond2gram.py q u
```

and you should be able to figure out what's happening.

You submit

- `cond2gram.py`
- `main.py`

Q3. [Substitution cipher]

The goal is to decrypt the following ciphertext (i.e., you need to compute the plaintext) and also to discover the key used. The substitution cipher is used.

```
tjfgrltuutnspjjqieusregdkqurweipqnxwqpyerievserjru
qijwekfeiegpfgkrrqehneeltdbpkleutdtjegdlasjuqifgil
ftjwrsnwgvstejgtiquygrjeiktuepjwqfezeijwgjfwedfwtj
dekfgrqdnenqdutdeltdjwengoykytrrtqdfgraignjtngppkg
nnqyaptrwelgdluqijweierjtnqspldqjftrwgdkjwtdboejje
ijwgdjqoegrrqntgjelftjwykuitedltdqdequjwqrertdbspg
iglzedjsierfwtnwfeiejwedqiygpnqdltjtqdquwtrehtrjed
netdguefytdsjertwglfitjjedykdqjeagtlfwtjdekrotpppe
lwtyqsjjqjwengogdlreedwtylitzedjwiqsbwjwelgixderrt
dgzeikrwqijjtyeglenieatjutbsiewgleyeibeluiqyjweqat
syledgdltfgrfgpxtdblqfdjwerjieejftjwrweipqnxwqpyer
uqijfqrjieejrwerwsuupelgpqdbftjwgoedjognxgdlgdsdne
ijgtduqqjjwedbpgdntdbvstnxpkiqsdlwerjigtbwjedelwty
repuqsjgdlosirjtdjqgwegijkutjqupgsbwjei
```

Once you have hacked the cipher using the given tool, you copy the the output of the tool into a python program and print the output. Call the program `main.py`.

For instance suppose the cipher was

```
oiwdhxuifjhwuimafwuiiuaeenkavfwuojvffayfeuvvtuivjh
dtjhuaifwamedmmnmuafwuindhfqlnyjybafjjjmaoahtuimuy
fenxjjlhafdiulzdfazvjedfuenyuhfuiuldtjhwqvjkhvqeen
vuem
```

And using the program, you finally broke it:

```
====================================================
ENCRYPTION KEY
==============
a -> a   b -> z   c -> y   d -> l   e -> u
f -> m   g -> x   h -> w   i -> q   j -> g
k -> b   l -> e   m -> o   n -> h   o -> j
p -> t   q -> p   r -> i   s -> v   t -> f
u -> d   v -> c   w -> k   x -> r   y -> n
z -> s

MESSAGES (ciphertext on top)
============================
oiwdhxuifjhwuimafwuiiuaeenkavfwuojvffayfeuvvtuivjh
mrhungertonherfatherreallywasthemosttactlessperson

dtjhuaifwamedmmnmuafwuindhfqlnyjybafjjjmaoahtuimuy
uponearthafluffyfeatheryuntidycockatooofamanperfec

fenxjjlhafdiulzdfazvjedfuenyuhfuiuldtjhwqvjkhvqeen
tlygoodnaturedbutabsolutelycentereduponhisownsilly

vuem
self

=====================================================
```

You then copy the above into a program named `main.py` as like:

```
# File: main.py
print('''
=====================================================
ENCRYPTION KEY
==============
a -> a   b -> z   c -> y   d -> l   e -> u
f -> m   g -> x   h -> w   i -> q   j -> g
k -> b   l -> e   m -> o   n -> h   o -> j
p -> t   q -> p   r -> i   s -> v   t -> f
u -> d   v -> c   w -> k   x -> r   y -> n
z -> s

MESSAGES (ciphertext on top)
============================
oiwdhxuifjhwuimafwuiiuaeenkavfwuojvffayfeuvvtuivjh
mrhungertonherfatherreallywasthemosttactlessperson

dtjhuaifwamedmmnmuafwuindhfqlnyjybafjjjmaoahtuimuy
uponearthafluffyfeatheryuntidycockatooofamanperfec

fenxjjlhafdiulzdfazvjedfuenyuhfuiuldtjhwqvjkhvqeen
tlygoodnaturedbutabsolutelycentereduponhisownsilly

vuem
self

=====================================================
''')
```

[NOTE: You probably want to perform a frequency analysis of common 1-gram, common 2-grams. etc. on the above given ciphertex.]

SOLUTION.