Assignment 1: Cryptography Suite

Due: 11:59:59 PM, October 9th

Overview

This assignment is all about building utilities for a cryptography suite. In this handout, we'll walk you through building a text-based cryptography tool. In this first assignment, we want to instill some good Pythonic practices from the beginning - so we encourage you to think critically about writing good Python code.

Expected Time: 4 hours (if it takes much longer than that, email us)

Selected Reading

Get a quick refresher by flipping through our slides from Weeks 1, 2, and 3 on stanfordpython.com

Starter Files

We've provided a starter zip file available on the website as a skeleton for this assignment. Here's an overview of what's in it:

1. `crypto.py`
   This is the main file you'll be working with. It will contain all the functions to decrypt/encrypt strings and also the logic behind the text-based console menu.
2. `feedback.txt`
   See the second-to-last section for some questions we'd love to ask you
3. Sample text files
   `caesar-plain.txt, caesar-cipher.txt, vigenere-plain.txt, vigenere-cipher.txt, railfence-plain.txt, railfence-cipher.txt`
   Text files to be used for testing the final portion of the program. In particular, the Vigenere key is `LEMON` and the railfence cipher uses three rails in these example files.

Cryptography Suite

Building the Ciphers

In this section, you will build out the cipher functions to encode and decode strings. We'll give a brief overview of each cipher and give some pointers on how it fits it into the starter files.

Caesar Cipher

A Caesar cipher involves shifting each character by three letters forward:

```
A -> D, B -> E, etc...
```

At the end of the alphabet, the cipher mapping wraps around the end, so:

```
X -> A, Y -> B, Z -> C.
```

For example, encrypting 'PYTHON' with a Caesar cipher gives

```
PYTHON
SBWKRQ
```

You can assume that the text passed to each of these functions will be comprised only of alphabetic characters. However, you should not assume anything about the case of these letters.

Hint: Consider using a dictionary for the caesar cipher mapping.

Implement the functions:

```
encrypt_caesar(plaintext)
decrypt_caesar(ciphertext)
```

Then test the functions by dropping into the Python interpreter `python3 -i crypto.py` and verifying that `encrypt_caesar("python")` returns "`SBWKRQ`".

Note: Windows users: Run `py -3`

Vigenere Cipher

A Vigenere cipher is a lot like a Caesar cipher, only every character in the plaintext is shifted by a different amount determined by a keyword. The keyword is repeated or truncated as necessary to fit the length of the plaintext. As an example, encrypting "ATTACK AT DAWN" with the key LEMON gives

```
Plaintext:      ATTACKATDAWN
```

```
Key:            LEMONLEMONLE
Ciphertext:     LXFOPVEFRNHR
```

Looking more closely, each letter in the ciphertext is the sum of the letters in the plaintext and the key. Thus, the first character of ciphertext is L due to the following calculations: A + L = 0 + 11 = 11 → L.

It may be useful to use the functions `ord` and `chr` which convert strings of length one to and from, respectively, their ASCII numerical equivalents.

Implement the methods:

```
encrypt_vigenere(plaintext, keyword)
decrypt_vigenere(ciphertext, keyword)
```

Then, try testing the methods using the interactive flag again.

Railfence Cipher

Below is a sample encryption of the plaintext “We are discovered. Flee at once.” using a railfence with 3 rails to generate the ciphertext “WECRLTEERDSOEEFEAOCAIVDEN”

```
WE ARE DISCOVERED. FLEE AT ONCE

W . . . E . . . C . . . R . . . L . . . T . . . E
. E . R . D . S . O . E . E . F . E . A . O . C .
. . A . . . I . . . V . . . D . . . E . . . N . .

WECRL TEERD SOEEF EAOCA IVDEN
```

The ciphertext is obtained by reading the rails from left to right, top to bottom.

Implement the functions:

```
encrypt_railfence(plaintext, num_rails)
decrypt_railfence(ciphertext, num_rails)
```

Consider using list comprehensions and slice syntax to simplify your implementation.

Console Menu

This program is user-driven, so your first step is to write a function that asks the user if they want to **(e)ncrypt** or **(d)ecrypt** a **(f)ile** or a **(s)tring**, and lastly whether to do so using a **(c)aesar** cipher, a **(v)igenere** cipher, or a **(r)ailfence** cipher. If the Vigenere or railfence ciphers are selected, either for encryption or decryption, you should make sure to ask the user for additional information. In the case of the Vigenere cipher, you should ask them for the phrase to be used as a secret key. In the case of the railfence cipher, you should request a positive integer number of rails.

The program should be case-insensitive. A sample run of the program might look like this (user input is bold and italic):

```
$ python3 crypto.py
Welcome to the Cryptography Suite!
*Input*
(F)ile or (S)tring? S
Enter the string to encrypt: I'm not dead yet
*Transform*
(E)ncrypt or (D)ecrypt? E
(C)aesar, (V)igenere, or (R)ailfence? v
Passkey? Python
Encrypting IMNOTDEADYET using Vigenere cipher with key PYTHON
...
*Output*
(F)ile or (S)tring? f
Filename? output.txt
Writing ciphertext to output.txt...
Again (Y/N)? Y
*Input*
(F)ile or (S)tring? F
Filename: secret_message.txt
*Transform*
(E)ncrypt or (D)ecrypt? D
(C)aesar, (V)igenere, or (R)ailfence? C
Decrypting contents of secret_message.txt using Caesar cipher
...
*Output*
(F)ile or (S)tring? S
The plaintext is: HELLOWORLD
```

If the user enters an invalid option to a prompt, you should reprompt until they enter an appropriate response.

For robustness, you should strip text of all non-alphabetic characters like spaces and punctuation, which should not be encrypted, before passing the text to the `encrypt_*` and `decrypt_*` functions. The function `str.isalpha()` may be helpful.

Implement the function:

```
run_suite()
```

which should run a single iteration of the cryptography suite, prompting the user for information about input, transformation, and output.

File I/O

No cryptography tool would be complete without the ability to encrypt and decrypt files. In this section, you will incorporate file input and output into the program.

You will need to update the console menu in order to allow the user to supply a filename to use.

You can assume that all filenames supplied by the user represent valid files with the appropriate read/write flags set.

It may be helpful to write methods to interact with files.

Implement the functions:

```
read_from_file(filename)
write_to_file(filename, content)
```

Recall that you can open a file for writing by passing the 'w' flag to open.

Extensions

The following section is an extension and is entirely optional. If you choose to give it a crack, give us some documentation and let us know how it went in your feedback!

Merkle-Hellman Knapsack Cryptosystem

Public-key cryptography is essential to modern society. One of the earliest public-key cryptosystems was the Merkle-Hellman Knapsack Cryptosystem, which relies on the NP-complete subset sum problem. Although it has been broken, it illustrates several

important concepts in public-key cryptography and can give you lots of practice with Pythonic constructs.

Read through the Wikipedia summary here, specifically the example section here. If you have any specific questions about how the cryptosystem works from a theory perspective, email us or ask a question on Piazza.

You have complete freedom in choosing how to implement this cryptosystem. We will be looking for good style and program design.

Automatic Decrypter

Suppose that you have access to ciphertext that you know has been encrypted using a Vigenere cipher. Furthermore, suppose that you know that the corresponding plaintext has been written using only words in /usr/share/dict/words, although you don't know the exact message. Finally, suppose that you know that someone has encrypted a message using a Vigenere cipher with a key drawn from a preset list of words. Can you still decrypt the ciphertext?

For many of the incorrect keys, the resulting plaintext will be gibberish. Thus, the bulk of this problem lies in evaluating how close to an English sentence a given sequence of letters is.

Your top-level function should be

```
decrypt_vigenere(ciphertext, possible_keys)
```

Besides that, you are free to implement this program however you see fit. However, think about the Python style guidelines before continuing.

You can test your method on "SEOEYMIDWTGFAGAXSEAOQYAIXTRROTY"

Beyond Text Files

So far, our encryption techniques have been applied solely to letters from the alphabet. However, it is possible to extend these encryption methods to work on arbitrary binary data, such as images, audio files, and more. For this extension, choose at least one of the encryption techniques and make it work on binary files. You will need to use the binary flag when reading from files, which you can read more about here. You may want to read about text sequence types compared with binary sequence types as well.

Feedback

We hope you have been enjoying the course so far, and would love to hear from you about how this first assignment went. It's the first assignment we've ever written from scratch, and there's definitely room for us to improve on structuring and pacing the various parts of the class and assignments.

To help us out, please answer the following questions in the feedback.txt file:

1. How long did this assignment take you to complete?
2. What has been the best part of the class so far?
3. What can we do to make this class more enjoyable for you?
4. What types of assignments would excite you in this class?

Thank you for being our Guinea pigs this quarter - we're learning from you guys as well as we teach this course!

Submitting

To submit an assignment, follow these steps:

1. Zip up any relevant files into a folder named "[SUNet ID]-assignment[Assign #]" (e.g. "skleung-assign1")
2. Go to stanfordpython.com/submit
3. Upload the zip file from Step #1
4. Confirm that the file was submitted by clicking on the confirmation link that the site should give you back.

Credit

Lots of credit to Python Tutorial, Learn Python the Hard Way, Google Python, MIT OCW 6.189, and Project Euler.