22C:16 (CS:1210) Project 1

Introduction

Cryptology (from the Greek: crypto = "hidden" and -ology = "study") is the study of codes and secret writing (see also cryptography = "hidden" + "writing", and cryptanalysis = "hidden" + "untie"). Long of interest for its obvious military and political application, it provides a fascinating study of the impact of technology on society; for those who are interested in the history of cryptography, we recommend "The Code Book" by Simon Singh, also author of "Fermat's Enigma," which has more than passing relevance to our second homework.

In this assignment, we will be decoding strings that represent secret messages so that they can be understood. Each secret message will be provided to you in a file that contains a combination of *instructions* and *secret text*. Your program will have to interpret the instructions and apply them to the secret text to produce the output. The assignment provides you an excellent opportunity to practice programming skills pertaining to strings and lists. Here, sequences of characters make up strings which represent both the instructions and the secret messages. Your program will be manipulating the latter guided by the former. Note that your assignment is to write Python functions to handle these manipulations (as detailed below) and not to rely on existing Python string manipulation libraries. We realize that Python libraries already exist to handle some of the manipulations described here; the point of the assignment is to learn about how to manipulate strings from scratch, not how to use the Python string library. Later in the semester, you'll have plenty of opportunity to use the Python string module.

This is an assignment best approached in stages, where each stage adds a little more functionality to your program until, eventually, all of the assigned requirements are met. Working in stages lets you test your code each step of the way, so that you can be confident that any errors you may find have been introduced only in your most recent changes. For this assignment, we'll help you by defining the recommended stages for you; one of the most important problem solving skills you can develop is the ability to decide how to break a larger problem down in this fashion so as to facilitate the development of a solution.

The Big Idea

Your program should contain a function with the header

```
def driver(filename):
```

As the name suggests, this function "drives" your entire program. This function takes a single filename as parameter, and opens and reads the indicated file, and follows the instructions contained within to decode a secret message, that is also contained in the file. Your program will not contain a main program and a user would have to call the `driver` function appropriately to get your program going.

Here is an example to illustrate the contents of the input file. The input file contains one or more entries, called *stanzas*. Each stanza looks like:

```
-5d&xyr
3rjyx~~x%st++my+~~U%~~~hsjlwj+rj%jmy%kt%yxj+y+%f%xn%xn++mY
```

where the first line contains the instructions that need to be applied to the rest of the lines in the stanza. The file may contain multiple stanzas, separated by a blank line, where each stanza consists of a (different) instruction set and (one or more) lines of message.

The Basic Instruction Set

These instructions constitute a little language, where:

`r`	reverses the text;
`-<n>`	shifts characters by subtracting `<n>` from their ASCII value;
`+<n>`	shifts characters by adding `<n>` to their ASCII value;
`d<char>`	removes all occurrences of the character `<char>` from the text;
`d<n>\<string>`	removes all occurrences of characters in the `<n>`-length `<string>` from the text;
`s<char1><char2>`	substitutes all occurrences of `<char1>` with `<char2>`;
`s<n1>\<string1>\<n2>\<string2>`	substitutes all occurrences of the `<n1>`-length `<string1>` with the `<n2>`-length `<string2>`;
`x<char>`	replaces consecutive repeated `<char>` characters with single `<char>` character.

Here `<n>`, `<n1>`, and `<n2>` refer to a positive integer, `<char>`, `<char1>`, and `<char2>` to a single character, and `<string>`, `<string1>`, and `<string2>` to arbitrary sequences of characters. Also note that the character "\" plays a special role as a delimiter separating different parts of the instructions "`d`" and "`s`."
So the instruction line:

```
-5d&xyr
```

means:

1. subtract 5 from the ASCII value of each character ("`-5`")
2. delete all "`&`" characters ("`d&`")
3. reduce all multiple "`y`" characters to a single "`y`" ("`xy`")
4. reverse the string ("`r`")

so:

```
3rjyx~~x%st++my+~~U%~~~hsjlwj+rj%jmy%kt%yxj+y+%f%xn%xn++mY
```

becomes (after "`-5`"):

```
.metsyys no&&ht&yyP yyycnegre&me eht fo tse&t& a si si&&hT
```

becomes (after "`d&`"):

```
 .metsyys nohtyyP yyycnegreme eht fo tset a si sihT
```

becomes (after "`xy`"):

```
 .metsys nohtyP ycnegreme eht fo tset a si sihT
```

becomes (after "`r`"):

```
 This is a test of the emergency Python system.
```

Notes on this Language

- Note that the instructions can occur in any order, but that they must be applied in the order they are given. For example, the instruction sequence:

  ```
  -5sps
  ```

  produces a markedly different result than:

  ```
  sps-5
  ```

- Also note that the `+/-` operations must be performed on the underlying ASCII values of the characters. To do this, you'll have to rely on the Python built-in functions `ord()` and `chr()`. For this assignment, we'll use only the 95 characters with ASCII value between 32 (the space character) and 126 (the "~" character), inclusive; you'll have to use modulo arithmetic and some addition/subtraction to keep everything within that range. This also means that any character appearing in the instructions will also have ASCII value in the range 32 through 126 (inclusive).

- The positive integers that appear in the instructions can have more than one digit. For example, `-12+50d11\xyzuvwrstpq` is a valid sequence of instructions that is asking us to first subtract 12 from the ASCII values of characters, then add 50, and then remove all occurrences of the last 11 lower case characters from the message. To avoid any ambiguity in the interpretation of instructions, we assume that in any situation where a number is expected in an instruction, the number is made up of all the consecutive digits that appear. For example, the first instruction in the above sequence is "`-12`" and not "`-1`."

- The fact that "\" is a special character raises questions about how instructions that delete or substitute the character "\" might look. This is not too hard to figure out. For example, if we wanted to delete all occurrences of "\" we could simply use the instruction "`d\`". In this instance, the "\" is not playing the role of a delimiter because there is no positive integer immediately following the "`d`." Alternately, the instruction "`d1\\`" would serve the same purpose.

Special Instructions

To the set of instructions described above, we now add three more special instructions. These instructions are special because they must appear as the last instruction in a set of instructions, and because only one of these can appear in each stanza.

- `m<filename>` use the text in the file named `<filename>` as a mono-alphabetic cipher to decode the line.

- `a<keyword>` apply an autokey cipher with keyword `<keyword>` to decode the line;

- `o<filename>` use the text in the file named `<filename>` as a one-time pad to decode the line;

Monoalphabetic Ciphers

A monoalphabetic cipher, or substitution cipher (`http://en.wikipedia.org/wiki/Substitution_cipher`) is a scheme that maps each letter in a message to another letter. The specified file will contain a single line of 95 characters, corresponding (in order) to the encrypted characters from `chr(32)` through `chr(126)`. So, for example, assume the file contains the 95 characters:

```
,lfU|z ~:sY>/48TK#w0^uiB'kXFALC<q[SG$d+x;QRZ3N9jaH&r7*‘teOM}WP_cgD]y6v@V2h\pJ!b(.{m%I1)=-E?o5"n
```

if we line these characters up under the 95 ASCII characters between 32 and 126, inclusive, we have

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
,lfU|z ~:sY>/48TK#w0^uiB'kXFALC<q[SG$d+x;QRZ3N9jaH&r7*`teOM}WP_cgD]y6v@V2h\pJ!b(.{m%I1)=-E?o5"n
```

So, for example, the letter "`A`" is encrypted as "`[`" and "`\`" is encrypted as "`W`". So, to decrypt the encrypted message "`2vJJ(`", you look for the "`2`" in the lower line and take the corresponding letter in the upper line (here, "`h`"); thus "`2vJJ(`" becomes "`hello`".

Monoalphabetic ciphers may be the kind of ciphers you might have played with as a child (e.g., your "secret decoder ring"), but they are not every secure. Not how every time the same letter appears, it is mapped to the same encrypted letter: if we analyze the frequency with which the encrypted letters occur, and compare these frequencies to the expected frequencies of letters in the target language, it fairly quickly becomes obvious what the decoded letters should be.

Autokey Ciphers

An autokey cipher (`http://en.wikipedia.org/wiki/Autokey_cipher`) essentially uses the message string to encode itself. Here, we'll consider a somewhat simplified version of an autokey cipher. First, we line up the text we wish to encrypt over a string composed of the keyword and the original text like this:

```
This is a test of the emergency Python system.
blarneyThis is a test of the emergency Python system.
```

where "`blarney`" is the chosen secret key. Next, we add each of the corresponding characters together (again, considering only characters between 32 and 126, inclusive, and performing modulo arithmetic) to get:

```
7UKfnOmi6h^Ys^soH iNYte]Lr\NTc_m6l\N^RysJmiN]|
```

To decode this message, we begin by subtracting the chosen secret key:

```
7UKfnOmi6h^Ys^soH iNYte]Lr\NTc_m6l\N^RysJmiN]|
blarney
This is
```

Once the key is subtracted, we proceed by subtracting the newly-decoded string from the rest of the string, one character at a time, offset by the length of the key.

Autokey ciphers seem pretty slick, but they are also fairly easily broken, using an extension of the frequency analysis described earlier that takes advantage of the fact that the key is generally fairly short, so that the message is eventually used, displaced, to encode itself.

One-time Pad Ciphers

A one-time pad (`http://en.wikipedia.org/wiki/One-time_pad`) extends the autokey idea by using a key that is at least as long as the message you want to encode, thereby making cryptanalysis impossible. The one-time pad gets its name from the fact that you are expected to use the key only once and never again for subsequent messages. This code is the only truly unbreakable code: the problem with the one-time pad is that you need to share a very long key (the works of Shakespeare? Dante's *Inferno*?) so that you can send messages of any interesting length. For this reason, we'll store the one-time pad (a SINGLE very long line of characters) in a separate file.

To encode a message with a one-time pad, simply align the message with the random string from the one-time pad and add them together, character by character (again, sticking to ASCII characters with decimal values from 32 to 126, inclusive). Decryption works the same way, except you'll have to subtract instead of add.

Stages of the Project

This project will be broken into four separate stages. At each stage, you will turn in a portion of your code; if you don't trust your solution, you will be provided with a solution for the previous stage before going on to the next stage so that failing to complete one stage will not make the rest of the problem impossible to solve.

Stage 1: Due March 13, 4:59 pm

In the first stage, you should set up the general structure of your program, with the `driver` function taking the name of a file to decrypt, opening that file, and decrypting the lines contained therein (n.b., here, you will have to be careful to remove any trailing newline characters from the input line to avoid decrypting it as well). For this stage, the only cipher directives you will have to deal with are "`-`", and "`+`."

As noted before, your program should not make use of the Python string library, rather, you should write your own functions to perform the required operations on strings as sequences. In addition to the `driver` function mentioned earlier, you will have to provide and use the function with the following header:

```
# Returns a new string that is equivalent to the argument string
# shifted by n ASCII values, where n may be a positive or negative int.
def shift (string, n):
```

This functions should be used in your main code to perform the string operations needed for decoding: be careful to ensure you are always working on integer values between 32 and 126, corresponding to ASCII characters starting with " " (space) up to "~" (tilde).
What to submit? A single python file called `project1.1.py`

Stage 2: Due March 27, 4:59 pm

In the second stage, you should extend your program to decipher the messages which make use of all of the directives in Stage 1 plus the new cipher directives "`r`", "`d`", "`s`", and "`x`". The functions you should implement and use are:

```
# Returns a new string that is a copy of the string provided but in
# reverse order.
def flip (string):

# Returns a new string that is a copy of the string provided but with
# all characters in the target string removed.
def flush (string, target):

# Returns a new string that is a copy of the string provided but with
# every occurrence of the substring s1 replaced with the substring s2.
def swap (string, s1, s2):

# Returns a new string that is a copy of the string provided but with
# multiple consecutive occurrences of character c reduced to a single
# character c.
def trim (string, c):
```

What to submit? A single python file called `project1.2.py`

Stage 3: Due April 3, 4:59 pm

In the third and final stage, you should extend your program to handle the three ciphers (monoaphabetic substitution cipher, autokey cipher, and one-time pad cipher). Now your program should be able to handle the entire set of instructions defined in this handout. For this stage of the program we are not requiring any specific functions.
What to submit? A single python file called `project1.3.py`

Final Remarks

For each stage we will provide test files and also solutions. When working on Stage i, $i > 1$, students should feel free to use the code we provide for Stage $(i - 1)$. Visit the course website regularly for updates.

At this point (March 6th) you can take Stage 1's requirements as fixed. However, we reserve the right to make *minor* tweaks to requirements of subsequent stages as students point out ambiguities or errors in the requirements. We will freeze all stages in a week.
