CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #1-2

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 pm

Getting Started

To help you get started on this assignment, we have given you a skeleton Python file named `trifid.py`. This file contains several *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You must fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `1-2.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

This assignment is worth a total of 20 points. You must earn at least 16 points (80%) in order to pass (receive credit for) this assignment.

- At the top of the `trifid.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Assignment #1-2)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters, or that use different parameter names) can't be graded by our automated grading system, and may receive a grading penalty.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

The Trifid Cipher

The *Trifid cipher* (not to be confused with the creatures from the classic science-fiction film "The Day of the Triffids") is an algorithm that enciphers a plaintext message by encoding each letter as a three-digit number and then breaking up and rearranging the digits from each letter's encoded form. For this assignment, you will create a set of Python functions that can encode messages using this cipher (these functions are described in detail following the general explanation of how the Trifid cipher works; you can skip ahead to those function specifications if you want to, but they may make more sense if you read the general explanation first).

The Trifid cipher works as follows:

1. Use an encryption key to construct a lookup table that will translate each letter into a 3-digit number. The lookup table (technically, there are three separate lookup tables that work together) requires 27 characters in all; in addition to the 26 letters of the alphabet, we will use the '!' character as the 27th value.
2. Process the plaintext message one letter at a time, obtaining its equivalent number from the lookup table and writing the digits of that number along each of three rows (one digit per line/row).
3. After the plaintext message has been converted to digits, break each row of digits into a series of 5-digit groups. Recombine the groups into a single string of digits in the following order: the first groups of each row, followed by the second groups of each row, etc.
4. Each group of three digits from this combined string is then translated into a new letter via the lookup table.
5. Finally, we break up the ciphertext into 5-letter groups. If the final group has fewer than 5 letters, we add Xs as necessary to pad it to a length of 5.

Example

Suppose that we want to use the Trifid cipher to encrypt the message "Sherlock Holmes and Doctor Watson are investigating" using the secret key "deduction". After capitalizing every letter in the key, we write out its unique letters, in the order they appear, followed by the remainder of the alphabet:

DEUCTIONABFGHJKLMPQRSVWXYZ!

These letters will be arranged into three 3x3 grids as follows:

Table 1			
	1	2	3
1	D	E	U
2	C	T	I
3	O	N	A

Table 2			
	1	2	3
1	B	F	G
2	H	J	K
3	L	M	P

Table 3			
	1	2	3
1	Q	R	S
2	V	W	X
3	Y	Z	!

We can use these tables to obtain the 3-digit trigram for each letter in the plaintext message. For example, the letter "S" translates to 313 (table 3, row 1, column 3). Each trigram is written vertically below the original letter (the message has been broken into 5-letter groups):

```
SHERL OCKHO LMESA NDDOC TORWA TSONA REINV ESTIG ATING
32132 11221 22131 11111 11331 13111 31113 13112 11112
12113 32223 33113 31132 23123 21333 11232 11221 32231
31221 11311 12233 21111 21223 23123 22321 23233 32323
```

To encrypt the message, we take each "block" (in this case, all three rows of a 5-character group) and concatenate the rows into a single string of (15) digits. Every three-digit group in this string, reading left to right, is then translated back into a new letter using the same letter-trigram equivalences as before:

```
321321211331221 = 321 321 211 331 221 = V V B Y H
112213222311311 = 112 213 222 311 311 = E G J Q Q
221313311312233 = 221 313 311 312 233 = H S Q R P
111113113221111 = 111 113 113 221 111 = D U U H D
113312312321223 = 113 312 312 321 223 = U R R V K
131112133323123 = 131 112 133 323 123 = O E A X I
311131123222321 = 311 131 123 222 321 = Q O I J V
131121122123233 = 131 121 122 123 233 = O C T I P
111123223132323 = 111 123 223 132 323 = D I K N X
```

This gives us a final enciphered message of `VVBYH EGJQQ HSQRP DUUHD URRVK OEAXI QOIJV OCTIP DIKNX`.

Part I: Constructing the Lookup Table (10 points)

Complete the `buildEncipheringTable()` function, which takes a string argument (the encryption key) and returns a dictionary that maps (uppercase) letters to three-digit trigrams. Use the pseudocode algorithm below as a guide:

1. General Preparation
   - Convert the key to uppercase and use the string method `replace()` to remove any spaces. The `replace()` method takes two arguments: the character sequence to be replaced, and the replacement value. In this case, use `" "` (a single space) as the search value and `""` (the empty string) as its replacement, e.g., `myKey.replace(" ", "")`.
   - Construct a list of available (not yet used) letters of the alphabet (use `string.ascii_uppercase` plus the placeholder character `"!"`).
   - Create a list to hold your three initial lookup tables. This list should contain four elements: an arbitrary integer (or some other value) to serve as a placeholder, followed by three empty lists (the placeholder value is there so that the indices of the three lookup table lists will begin with 1 rather than 0).
   - Create a variable to track the index of the current lookup table (this will initially be 1).
2. Create the Lookup Tables
   (a) For each letter in the encryption key, check to see if it is in the list of available/unused letters. If it is, use the `remove()` method to remove it from that list and append it to the current lookup table list. If the length of the current lookup table list is now 9, increment the "current lookup table" variable to point to the next list. Note that if the encryption key contains multiple copies of a given letter, that letter will only be added to the lookup table(s) one time.
   (b) Once you have processed the entire encryption key, add the remaining values from the "unused letters" list to the current lookup table, moving to the next lookup table each time the current table's list reaches a length of 9 elements.

3. Create the Dictionary of Trigrams
   (a) Create a new, empty dictionary to store your letter-trigram mappings.
   (b) For each of the three lookup tables/lists in order (indices 1, 2, and 3 in your list from Step 1), calculate the letter trigrams as follows:
      - Your new dictionary key will be the letter at that index
      - The trigram for that letter is calculated as follows:
        - `firstDigit` = the index of the current table in the list of tables (e.g., 1, 2, or 3)
        - `letterIndex` = the index of the current letter in the current table
        - `secondDigit` = (`letterIndex` // 3) + 1
        - `thirdDigit` = (`letterIndex` % 3) plus 1
        - `trigram` = `firstDigit` * 100 + `secondDigit` * 10 + `thirdDigit`
   (c) Return your newly-filled dictionary.

Examples:

Note: the examples below do not show the entire lookup table that will be generated from the encryption key. Instead, they show what the resulting lookup table will return for a given plaintext character.

Function Call	**Plaintext Letter**	**Corresponding Trigram**
`buildEncipheringTable("DRAGON")`	`"R"`	112
`buildEncipheringTable("DRAGON")`	`"I"`	213
`buildEncipheringTable("DRAGON")`	`"Z"`	332
`buildEncipheringTable("NEPTUNE")`	`"B"`	131
`buildEncipheringTable("NEPTUNE")`	`"J"`	222
`buildEncipheringTable("NEPTUNE")`	`"V"`	321
`buildEncipheringTable("CHALLENGER")`	`"E"`	122
`buildEncipheringTable("CHALLENGER")`	`"Q"`	233
`buildEncipheringTable("CHALLENGER")`	`"T"`	312

Part II: Performing the Encryption Process (10 points)

Finally, we need to write the code that will actually encipher the user's message using the selected encryption key. Complete the `encipher()` function, which takes two string arguments (the plaintext message and the encryption key), according to the following algorithm:

1. Call your `buildEncipheringTable()` function from Part I to create the dictionary of trigrams.
2. Create three variables (`row_1`, `row_2`, and `row_3`), and assign the empty string to each. These variables will hold parts of the enciphered message.
3. Convert the plaintest message to uppercase and remove any spaces using the string method `replace()`.
4. For each letter in the message:
   (a) Get the letter's corresponding trigram from the dictionary
   (b) Append the first digit of the trigram (as a string) to `row_1`
   (c) Append the second digit of the trigram (as a string) to `row_2`
   (d) Append the third digit of the trigram (as a string) to `row_3`
5. Use the `invert()` helper method that we provided you to create a new dictionary from your original letter-trigram dictionary, and create a new variable (called `combined` in the steps that follow) that is initially set to the empty string.
6. Use a loop to fill `combined` with the contents of `row_1`, `row_2`, and `row_3`, five letters at a time. In other words, when you are done, `combined` should hold the first 5 letters of `row_1`, the first 5 letters of `row_2`, the first 5 letters of `row_3`, the second 5 letters of `row_1`, the second 5 letters of `row_2`, etc.
7. Create a new string variable to hold the final, enciphered message.
8. Read the contents of `combined` in 3-character chunks. Convert each "chunk" to a 3-digit integer and use the resulting trigram as a key into your dictionary from Step 5 to get the next enciphered letter.
9. If the length of your final ciphertext is not evenly divisible by 5, append `"X"` characters to the end until it is.
10. Finally, use the string method `join()` to create a new string where the ciphertext is divided into 5-letter blocks, each separated by a single space, and return the result.

Examples:

Function Call	**Final Ciphertext**
`encipher("TOBEORNOTTOBE", "HAMLET")`	`LEZYL WHPID FZAXX`
`encipher("SPACETHEFINALFRONTIER", "KIRK")`	`VKWTE VKFBP HINDF NRGUM RXXXX`
`encipher("FOUR SCORE AND SEVEN YEARS AGO", "LINCOLN")`	`HPHIY IIRG! NBUQB NNERE PHCPW`
`encipher("The Helvetii compelled by the want of everything sent ambassadors to him about a surrender", "caesar")`	`VHCSU NWHSW IKFVP RCKZN GIQDJ TWDWR EESMF ZMCTS CPJDT SCKMV CIGIR FLDAI HOBFX QQSLF AAGBE RXXXX`
`encipher("Alan Turing was a leading participant in the breaking of German ciphers at Bletchley Park", "ENIGMA")`	`GCKCW VESRV UMHOC EEPIZ GQXEZ EFUQU BBEMW FFDGB GNFPV KMGYI ENJAO KRINB BHCCG TFTMK VOXFY`