Assignment 1:

Due: 11:55 PM November 13, 2016

Introduction

The goal of this assignment is to gain practice with some important Python and computer science concepts:

- reading and understanding algorithms (this can be quite a bit of work!)
- understanding a top-down design
- composing multiple functions that together implement the algorithm
- working with strings and lists

You'll be implementing a program to encrypt and decrypt text using an *encryption algorithm*. Such an algorithm describes a process that turns plaintext into what looks like gibberish (we'll call this *ciphertext*), and decrypts the ciphertext back into plaintext. For example, your program will be able to take a secret message written in ciphertext, and figure out what it says. ... Like:
`XHXZCHRYTMWEHOPPB`
Hmmm. How can that ciphertext be decrypted, and what does it say? You'll have to do the assignment to find out!

We decided to write a main program that is separate from the set of helper functions that the main program uses.

The Encryption Algorithm

The encryption algorithm carries out encryption and decryption using a deck of cards[1]. However, we're going to make two simplifications to a regular card deck. First, rather than use a standard deck of 52 cards, we're going to use 28 cards. Second, rather than use actual card ranks and suits, we're going to use the numbers from 1 to 28. We will call 27 and 28 the *jokers*. We're doing this because it's not necessary to know anything about playing cards to understand this algorithm, so we might as well use integers rather than card names.

The algorithm is an example of a *stream cipher*. What that means is that every time you complete a **round** of the algorithm, you get one *keystream value*. This stream of values is then used, in combination with plaintext or ciphertext, to encrypt or decrypt, respectively. You will complete one round of the algorithm for each letter in the text to be encrypted or decrypted.

Each round of the algorithm consists of (one or more repetitions of) five steps; once all steps in a round are complete, one keystream value is available and the next round of keystream generation can begin.

Here is the algorithm; an example follows.

Begin with a deck of cards, which is really just a permutation of the integers

We'll be doing lots of rounds, one for each letter in a message, and so we decided to have a function that does all the steps in a round.

Each of the steps in a round is independent, and so we decided to have a separate function for each step.

from 1 to 28. The steps for one round are as follows:

1. Find the card with value 27. Swap this card with the following card (so that it moves down one position). If the card with value 27 is at the bottom end of the deck, then swap the 27 with the top card. That is, think of the deck as circular, so that the card following the bottom card is the top card.
2. Find the card with value 28. Move it two cards down by performing two swaps, again treating the deck as circular.
3. Recall that the two cards with values 27 and 28 are the two jokers. Swap the cards above the first joker (the joker closest to the top of the deck) with the cards below the second joker. This is called a **triple cut**. *Think about what happens if a joker is at the top or bottom of the deck? What if the jokers are next to each other? Is this a problem?*
4. Look at the bottom card from the deck. That card will have some value `v`. If `v` is 28, use a value `v` of 27 in this step. Then, take `v` cards from the top of the deck, and put them above the bottom-most card in the deck.
5. Look at the top card from the deck. That card will have some value `v`. If `v` is 28, use a value `v` of 27 in this step. Starting from the top, find the card that is at position `v` in the deck (the topmost card is at position 0). If the card on which you land is a joker, continue the current round at step 1. Otherwise, remember the value of the card on which you landed; this is the keystream value generated for the current round. This keystream value will be in the range 1-26 inclusive. *(Thought question: why can't it be a 27 or a 28?)*

Steps 1 and 2 involve swapping cards, so we'll have a helper function that does that; both step 1 and step 2 will call that function.

If we get a joker as a potential keystream value, we might have to repeat all these steps. We decided to have a function that generates the next potential keystream value, and a separate function that calls that one until a non-joker is produced.

To generate another keystream value, we take the deck as it is after step 5 and run another round of the algorithm. We need to generate one keystream value for each character in the text to be encrypted or decrypted.

Example: Completing one Round

Let's go through an example of how the algorithm generates keystream values. Consult the descriptions of the steps given above as you follow along with this example. We'll illustrate the first round of the algorithm, but encourage you to do another round by-hand so you really understand what's happening.

Consider the following deck. The top card has value 1. The bottom card has value 26.

```
Top                                                                Bottom
 1 4 7 10 13 16 19 22 25 28 3 6 9 12 15 18 21 24 27 2 5 8 11 14 17 20 23 26
```

Step 1: Swap 27 with the value following it. So, we swap 27 and 2:

```
 1 4 7 10 13 16 19 22 25 28 3 6 9 12 15 18 21 24 2 27 5 8 11 14 17 20 23 26
                                                 ^^^^
```

Step 2: Move 28 two places down the list. It ends up between 6 and 9:

```
 1 4 7 10 13 16 19 22 25 3 6 28 9 12 15 18 21 24 2 27 5 8 11 14 17 20 23 26
                           ^^^^^^
```

Step 3: Do the triple cut. Everything above the first joker (28 in this case) goes to the bottom of the deck, and everything below the second (27) goes to the top:

```
5 8 11 14 17 20 23 26 28 9 12 15 18 21 24 2 27 1 4 7 10 13 16 19 22 25 3 6
^^^^^^^^^^^^^^^^^^^^^                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Step 4: The bottom card is 6. The first 6 cards of the deck are 5, 8, 11, 14, 17, and 20. They go just ahead of 6 at the bottom end of the deck:

```
23 26 28 9 12 15 18 21 24 2 27 1 4 7 10 13 16 19 22 25 3 5 8 11 14 17 20 6
                                                         ^^^^^^^^^^^^^^^
```

Step 5: The top card is 23. Thus, our generated keystream value is the card at position 23. This card has value 11. Since this value 11 isn't 27 or 28, we are done with the round.

As a self-test, you should carry out the next round of the algorithm to find the second keystream value. Its value is 9; be sure you get the same value to convince yourself that you understand the five steps.

Encrypting and Decrypting

To encrypt a message, remove all non-letters from the message and convert any lowercase letters to uppercase. Next, convert the letters to numbers (A becomes 0, B becomes 1, ..., Y becomes 24, and Z becomes 25). Then, use the algorithm to generate the same number of values as there are letters in the message. Add the corresponding pairs of numbers, modulo 26. Finally, convert the resulting numbers back to letters.

Decryption is just the reverse of encryption. Start by converting the message to be decoded to numbers. Using the same card ordering as was used to encrypt the message, generate one keystream value for each character in the message. (Because the same starting deck of cards was used, the same keystream will be generated.) Subtract the keystream values from the message numbers, again modulo 26. Finally, convert the numbers to letters to recover the message.

We'll be cleaning a lot of these messages to remove non-letters and converting to uppercase, so we decided to have a function for that.

We also are encrypting a single letter at a time, so we decided to have a function for that. Same for decryption.

Example: Encrypting and Decrypting

Let's say we want to encrypt the message `Lake Hylia`. Removing non-letters and capitalizing the letters gives us `LAKEHYLIA`. Next, convert these letters to numbers:

```
  L  A  K  E  H  Y  L  I  A
 11  0 10  4  7 24 11  8  0
```

Since we have nine letters, nine keystream values are required. Rather than go through nine rounds of the algorithm here, let's just assume that the nine generated keystream values are as follows:

```
 12 8 17 25 1 14 15 13 20
```

Now add the two groups together pairwise, modulo 26. This means that if the sum of a pair is greater than 25, just subtract 26 from it. (For example, 1 + 8 = 9, but 11 + 17 = 28 - 26 = 2.) Then, convert these numbers to letters to arrive at the encrypted message:

```
     11  0 10  4  7 24 11  8  0
+    12  8 17 25  1 14 15 13 20
     --------------------------
     23  8  1  3  8 12  0 21 20
      X  I  B  D  I  M  A  V  U
```

To decrypt this message, the recipient would start with the same deck with which the encryption was started, and generate the same nine-number keystream. They would then convert the encrypted message to numbers. Then, instead of adding corresponding numbers from the keystream and message, they would pairwise-subtract the keystream from the message, modulo 26. That is, if the subtraction gives you a negative number, add 26 to the result. (For example, 9 - 8 = 1, and 2 - 17 = -15, but since it is negative you add -15 + 26 = 11.) The decryption back to `LAKEHYLIA` looks like this:

```
     23  8  1  3  8 12  0 21 20
-    12  8 17 25  1 14 15 13 20
     --------------------------
     11  0 10  4  7 24 11  8  0
      L  A  K  E  H  Y  L  I  A
```

Of course we can never get all the way back to `Lake Hylia` because we don't know how to restore any lowercase letters or where to insert any spaces.

Files to Download

Please download the Assignment 1 Data Files and extract the zip archive. The following paragraphs explain the files you have been given.

The main program: `cipher_program.py`

This file contains the definition of three constants, an empty function called `main`, and a call on function `main`. More on this later.

A nearly-empty file: `cipher_functions.py`

You'll write almost all of the functions here. It's imported by `cipher_program.py`. It includes two constants: `JOKER1` and `JOKER2`. In order to make your code more readable, use these constants instead of using `27` and `28` lest you incur the wrath of the people marking your program!

deck files: `deck1.txt` and `deck2.txt`

A deck of cards will be represented in Python as a list of numbers. We provide a sample deck in `deck1.txt`. Notice that this file has multiple lines: a deck file can have any number of lines. Another sample file `deck2.txt` is a different file, but if you look closely you will notice that it represents the same deck. Your

We decided to store a deck in a text file, so we'll have a function that function for reading in a deck must work for both these deck files and any others that we might use for testing. In particular, the numbers might be in a different order and there could be one or more numbers on each line.

reads a deck file and builds a list of integers.

Some message files: `message-encrypted.txt`, `message-decrypted.txt` and `secret?.txt`

Messages to encrypt or decrypt will be stored in text files. Message text files contain one message per line. Imagine that we're encrypting or decrypting a message file that contains multiple lines (i.e. multiple messages). The first line is encrypted (or decrypted) using the deck in the configuration as specified in a deck file. Subsequent lines of that same file are encrypted (or decrypted) starting with the configuration of the deck following the previous encryption (or decryption). That is, the deck is not reset between message lines of a message file.

We decided to store messages in text files, so we'll have a function that reads a message file and builds a list of strings.

The starter code archive contains a pair of text files: `message-encrypted.txt` and `message-decrypted.txt` that contain both an encrypted and decrypted version of the same message, and several other files named `secret?.txt` (`secret1.txt`, `secret2.txt`, etc.) containing ciphertext that you can decrypt. Some of them contain multiple messages, one message per line. They were all encrypted using the deck in `deck1.txt`.

We need to process a list of messages, so we decided to have a function that does this. We also decided to have a helper function that processes a single message.

A type-checking file: `typechecker.py`

To help you test, we provide a type checker. This program calls all of the functions that appear in the table below (except for function `main` and the two file-reading functions) and makes sure that your functions accept the proper number of parameters and return the proper type. **Passing the type-checker says nothing of the correctness of your functions.** It only says that your functions have the correct number of parameters and that the functions return the right type of value.

For example, if your `clean_message` function has one parameter and **always** returns the empty string, it will pass the type checker even though it is completely incorrect. Passing the type checker means that our own tests will be able to call your functions properly when we go mark your assignment. That's all it means. Test carefully on your own as explained more toward the end of this handout!

Your Tasks

We have performed a top-down design on the handout and have come up with quite a few functions. Write all but the last one in file `cipher_functions.py`. (The last one, `main`, will go in `cipher_program.py`.)

We present them in what we think is a sensible order, but you're welcome to jump around as you work. Notice that the functions near the top of the table don't require you to use lists or to read from a file. In fact, you already know all the Python instructions you need to complete these functions.

Please follow the function design recipe we have been using in class. We will mark your docstrings in addition to your code, so we expect the docstrings to contain a description, examples, and type contract.

Wrath of the markers and all that.

We will be testing and marking each of these functions individually. So even if you can't complete the entire program and correctly encrypt and decrypt messages, you can earn marks for correctly implementing many of the functions.

List of functions to implement for A1.

Function name: (Parameter types) -> Return type	**Description**
`clean_message: (str) -> str`	The parameter is a message (i.e. it is a `str` that contains only one line of text). Return a copy of the message that includes only its alphabetical characters, where each of those characters has been converted to uppercase.
`encrypt_letter: (str, int) -> str`	The first parameter is a single character and the second represents a keystream value. Apply the keystream value to the character to encrypt the character, and return the result.
`decrypt_letter: (str, int) -> str`	The first parameter is a single uppercase character and the second represents a keystream value. Apply the keystream value to the character to decrypt the character, and return the result.
`swap_cards: (list of int, int) -> NoneType`	The first parameter represents a deck of cards and the second represents an index into the deck. Swap the card at the index with the card that follows it. Treat the deck as circular: if the card at the index is on the bottom of the deck, swap that card with the top card. (Note that in this and all functions that do something to the deck, the function doesn't return anything. The deck is to be mutated.)
`move_joker_1: (list of int) -> NoneType`	The parameter represents a deck of cards. This is step 1 of the algorithm. Find `JOKER1` and swap it with the card that follows it. Treat the deck as circular.
`move_joker_2: (list of int) -> NoneType`	The parameter represents a deck of cards. This is step 2 of the algorithm. Find `JOKER2` and move it two cards down. Treat the deck as circular.
`triple_cut: (list of int) -> NoneType`	The parameter represents a deck of cards. This is step 3 of the algorithm. Find the two jokers and do a triple cut.

`insert_top_to_bottom: (list of int) -> NoneType`	The parameter represents a deck of cards. This is step 4 of the algorithm. Look at the bottom card of the deck; move that many cards from the top of the deck to the bottom, inserting them just above the bottom card. Special case: if the bottom card is JOKER2, use JOKER1 as the number of cards.
`get_card_at_top_index: (list of int) -> int`	The parameter represents a deck of cards. This is step 5 of the algorithm. Look at the top card. Using that value as an index, return the card in that deck at that index. Special case: if the top card is JOKER2, use JOKER1 as the index.
`get_next_value: (list of int) -> int`	The parameter represents a deck of cards. This is the function that does all five steps of the algorithm. Return the next potential keystream value.
`get_next_keystream_value: (list of int) -> int`	The parameter represents a deck of cards. This is the function that repeats all five steps of the algorithm (call `get_next_value` to get potential keystream values!) until a valid keystream value (a number in the range 1-26) is produced.
`process_message: (list of int, str, str) -> str`	The first parameter represents a deck of cards. The second represents a message to encrypt or decrypt based on the third parameter, which is either `'e'` (to encrypt) or `'d'` (to decrypt). Return the encrypted or decrypted message. Note that the message might contain non-letters.
`process_messages: (list of int, list of str, str) -> list of str`	The first parameter represents a deck of cards. The second represents a list of messages to encrypt or decrypt based on the third parameter, which is either `'e'` (to encrypt) or `'d'` (to decrypt). Return the list of encrypted or decrypted messages.
`read_messages: (file open for reading) -> list of str`	The parameter represents an open message file, which contains one message per line. Read and return the contents of the file as a list of messages. Strip the newline from each line. Note: you don't have to provide example calls for functions that read files.
`read_deck: (file open for reading) -> list of int`	The parameter represents an open deck file, which contains the numbers 1 through 28 in some order. Read and return the contents of the file. Do not hard-code the number 28 anywhere; just read all of the integers from the deck file. Note: you don't have to provide example calls for functions that read files.

`main:`
`() -> NoneType`
	Open and read the deck file called `DECK_FILENAME`. Use it to encrypt or decrypt the messages in the file called `MSG_FILENAME`; if `MODE` is `'e'` you should encrypt, and if `MODE` is `'d'` you should decrypt. Print the encrypted or decrypted messages one per line. **This is the only function that prints anything.**

Testing your Code

It is strongly recommended that you test each function as you write it. (You might alternatively decide to just plow ahead and write all of the functions and hope everything works, but then it will be really difficult to determine whether your program is encrypting/decrypting correctly. If you get garbage when decrypting our ciphertext, then what? The bug(s)could be anywhere.)

Here is what we recommend for testing your A1 code:

- Start a file `cipher_tests.py`. Import `cipher_functions`.
- Call your function with inputs for which you've already determined what the function should do, and print the results of calling the functions. The examples you came up with during the function design recipe are great starting points. For some of the functions you can use small card decks -- you don't have to use full 28-card decks all the time. For example, working with a deck of five or six cards produces output that is much easier to read. Performing a triple cut on only a few cards is more manageable than observing the effects on a full 28-card deck. *Thought question: You need to be careful about using a smaller deck for functions insert_top_to_bottom, get_card_at_top_index and any functions that call these two functions. Why will some test cases be problematic?*
- Be careful that you test the right thing. Some functions return values; others modify the deck in-place. Be clear on what the functions are doing before determining whether your tests work.
- Can you think of any edge cases for your functions? Will each function always work, or are there special cases to consider? Test each function carefully.
- Once you are happy with the behaviour of a function, move to the next function, implement it, and start testing by adding new tests to your `cipher_tests.py` file.

When you're ready to test the algorithm overall (`cipher_program.py`), start with a one- or two-letter message and make sure it matches what you get by hand. Then work up to longer messages from there.

We are not asking you to submit your testing code as part of your submission. We are recommending that you write these tests to maintain your own sanity.

Remember to run the type checker!

Marking

These are the aspects of your work that will be marked for Assignment 1:

- **Correctness:** Your functions should perform as specified in this assignment handout.
- **Formatting style:** Make sure that you follow PEP-8 Python style guidelines.
- **Programming style:** Your variable names should be meaningful and your code as simple and clear as

possible.
- **No magic numbers:**: Hard-coding 27s and 28s in multiple places in your code is not a good idea. (Today, you know that they refer to the jokers. But will you remember that next month??) Use the constants instead of magic numbers. Converting this program to work with Welsh (which has 29 letters) should be trivial (assuming we had a version of Python with string functions built for the language).
- **Commenting**: be sure to include accurate docstrings (that follow the design recipe we have been using) and include internal comments.

The correctness of your code will count for approximately 50 percent of this assignment. The other 50 percent will be allocated for properly demonstrating your understanding of the code. (That is to say, even if you can't get all of the functions working, you can still pass the assignment by providing good documentation).

What to Submit

The very last thing you should do before submitting is to run the type-check module one last time. Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `cipher_functions.py` and `cipher_program.py`. Your files must be named exactly as given here (no capital letters, and please don't change the filenames).

RXLKVTNV!

[1] Bruce Schneier, a computer scientist who is considered by many to be a computer security guru, designed this algorithm. It was used in Neal Stephenson's novel *Cryptonomicon*.