Assignment 2

Due Feb 28 by 1pm **Points** 8

Recent updates

13 Feb The assignment submission system, MarkUs, will be available soon. We will send an announcement when it is ready.

Assignment 2: Cryptography

Due Date: Fri 28 Feb 2020 before 1pm.

You must only use the material described in weeks Weeks 1 through 6 of CSC108.

Introduction

Bruce Schneier (https://www.schneier.com/blog/about/), a renowned security expert, designed an encryption algorithm called **Pontifex** (https://en.wikipedia.org/wiki/Solitaire_(cipher)) that generates **pseudo-random numbers** (https://en.wikipedia.org/wiki/Pseudorandomness) by manipulating a deck of playing cards according to a set of rules. It was used in Neal Stephenson's novel Cryptonomicon. In the book, two field agents use this algorithm to encrypt and decrypt messages that they don't want to enemy to be able to decipher.

In the book, they each have a deck where the cards are in the same order, so by following the algorithm they generate the same sequence of numbers. Each number can be used to either encrypt or decrypt a single letter, so they follow the algorithm once for each letter in the message.

This assignment has you complete a Python program that implements an interesting encryption algorithm. The details of the algorithm are explained later in this handout. You'll need to read them carefully and understand them. Feel free to come to office hours and try out the algorithm yourself.

The purpose of this assignment is to practice using programming concepts you have seen in the course so far, including (but not limited to) lists and list methods and loops. By the end of it, we hope you are comfortable manipulating lists and writing loops.

You will also get the experience of translating an algorithm described in English into Python code.

Programming almost always involves this kind of translation.

Learning Goals

- Read and understand an intricate algorithm that manipulates a set of cards.
- Translate the intricate algorithm into Python.
- Use the **Function Design Recipe** to plan, implement, and test functions that manipulate lists using item assignment, slicing, and loops.
- Design and write function bodies using lists and list methods and loops.
- Practice good programming style.

Definitions: Encrypting and Decrypting Messages

In this section, we'll define some terms that are used when discussing encryption and decryption, and then present the high-level steps to take when applying an encryption algorithm to a message.

Suppose you are trying to pass an English language message to a friend. The contents of the message is so private, you don't want anyone other than you or your friend to be able to understand it. The message you are trying to pass, in a form that is understandable to any reader, is called the *plaintext*.

A hidden form of the message that can't be readily understood by anyone is called the *ciphertext*. *Encryption* is the process of converting plaintext into ciphertext, and *decryption* is the process of retrieving the original plaintext from the ciphertext.

In this assignment, the encryption algorithm we use encrypts (or decrypt) one letter of the message at a time.

First, we will remove all characters that are not letters from the plaintext message and also convert all lowercase letters to uppercase. This means that whitespace, digits and punctuation are lost in the encryption process, leaving only uppercase letters of the alphabet. Next, we convert the letters to numbers based on their place in the alphabet (A is represented as 0, B as 1, C as 2, ..., Y as 24, and Z as 25). We'll use the resulting sequence of these numbers as input to the encryption algorithm.

After the letters are converted to numbers (remember, we're only converting capital letters here), the encryption algorithm generates a special number for each position in the plaintext message. These special numbers are called *keystream* values. To encrypt each letter, the numeric letter value and the corresponding keystream value are **added**, modulo 26, and then the resulting number is converted to its letter form. The two numbers that are added are the number corresponding to the letter in the plaintext, and the keystream value for the position of the letter in the plaintext.Ω

Decryption is just the inverse of encryption. Start by converting the letters in the ciphertext to be decoded to numbers, in the same way as the original plaintext message was converted. Then, use the encryption algorithm again to generate a special number for each position in the ciphertext. **Subtract** these special numbers from the numbers from the ciphertext, again modulo 26. Finally, convert the resulting numbers to letters to recover the plaintext message.

The only other requirement for the encryption and decryption processes to work is that both encryption and decryption use the same sequence of keystream values. In our algorithm, we'll use a representation of a deck of playing cards to generate keystream values. The encryption and decryption algorithms need to start with the same decks.

Background: The Caesar Cipher Algorithm

Don't skip this. Understanding this background and reading and understanding the program we describe here will help you a lot. **Read the code we describe here!**

Julius Caesar (https://en.wikipedia.org/wiki/Julius_Caesar) used an encryption algorithm (now called the **Caesar cipher** (https://en.wikipedia.org/wiki/Caesar_cipher)) to encrypt some of his correspondence, in an attempt to keep his secrets from others. He would shift each letter from the alphabet by 3 places (using our new terminology, we use 3 for every keystream value): `A` becomes `D`, `B` becomes `E`, `C` becomes `F`, and so on. Letters at the end of the alphabet would wrap around to the beginning: `W` becomes `Z`, but then `X` becomes `A`, `Y` becomes `B`, and `Z` becomes `C`.

Here is a short message (penned by A. A. Milne):

```
PEOPLE SAY NOTHING IS IMPOSSIBLE BUT I DO NOTHING EVERY DAY
```

When encrypting this, P → S, E → H, O → R, and so on. Here is the encrypted version of this message produced by applying the Caesar cipher:

```
SHRSOH VDB QRWKLQJ LV LPSRVVLEOH EXW L GR QRWKLQJ HYHUB GDB
```

To decrypt an encrypted message, shift by 3 places in the other direction. S → P, H → E, R → O, and so on.

The Caesar cipher is a terrible encryption algorithm because it is very easy to break, but that also makes it a great example to start with.

We provide a module containing functions that encrypt and decrypt using the Caesar cipher algorithm: caesar_cipher.py. **We strongly recommend that you understand this code because you are going to write code that is remarkably similar to it.** The code uses `ord` to convert a letter to a number and `chr` to convert a number to a letter. **You're going to need to do very similar things in this assignment, so please study the provided Caesar cipher code. Ask on Piazza for help understanding this!**

The Encryption Algorithm

You'll be writing functions that implement an *encryption algorithm*. When used together, your functions will encrypt and decrypt text. Such an algorithm describes a process that turns plaintext into what looks like gibberish (the 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. An example of ciphertext is:
`WMAVOZGTBYAVIZIDQBNTWCNNDBOXFVUUWJMZCMBCANAMEQWEJ`
Hmmm. How can that ciphertext be decrypted, and what does it say? You'll have to do the assignment to find out!

You'll implement an encryption algorithm that carries out encryption and decryption using a modified deck of playing cards. A standard deck has 52 cards, but we are going to use a deck with one card for each letter of the alphabet, plus two extra cards. For the English alphabet, our deck will have 26 + 2 = 28 cards. For the Greek alphabet, it would have 24 + 2 = 26 cards.

Rather than use actual card ranks and suits, we're going to use the integers from 1 to the number of cards (for English, 1 to 28). We will call the cards with the two highest values *jokers* (for English, the jokers are the cards with numbers 27 and 28). In our program, we will represent the cards as a list of integers. The top card of the deck will be at index 0.

The algorithm that you will implement is an example of a *stream cipher*. This means that every time you complete a *round* of the algorithm, you generate an integer — a keystream value that you will use to encrypt (or decrypt) a single letter.

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

The steps for one round of the keystream value generation algorithm for processing an English message are as follows:

1. Find the small joker in the deck. (The small joker is the card with the second highest value, which is 27 in our context.) Swap this card with the card that is directly under it in the deck, so that the small joker moves down one position. If the small joker was found to be at the bottom of the deck, then instead swap the small joker with the top card. It may helpto think of the deck as being circular so that the card "under" the bottom card is the top card.
2. Find the big joker in the deck. (The big joker is the card with the highest value, which is 28 in our context.) Move the big joker two cards down the deck by performing two card swaps. As before, think of the deck as being circular.
3. Recall that the two cards with the highest two values are the two jokers. Call the joker closest to the top of the deck the *first* joker, and the one closest to the bottom the *second* joker. (Note this is independent of which is the big or small joker.) Swap the stack of cards above the first joker with the stack of cards below the second joker. This is called a **triple cut**. *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 value on the bottom card in the deck. Let's represent the value of the bottom card by the symbol v. If the bottom card is the big joker, take v to be the value of the small joker instead. (In our example with English, if the bottom card is 28, use v = 27 instead.) Then, take the group of v cards from the top of the deck, and insert that group just above the bottom card in the deck.
5. Look at the value on the top card in the deck. Let's represent the value of that top card by the symbol w. If w is the big joker, take w to be the value of the small joker instead. (Again, in our example with English, if the value on the top card is 28, use w = 27 instead.) Starting from the top, find the card that is at index w in the deck. If the card at position w is a joker, continue the current round back at Step 1. Otherwise, remember the value on the card at position w in the deck — it is the keystream value generated for the current round. This keystream value will be in the range of 1 to the size of the deck minus two, inclusive (1 to 26 in our example with English). *Why can't it be a joker?*

To generate the next 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 position in the text to be encrypted or decrypted.

Example: Completing one round of the keystream value generation algorithm

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 that could be used for encryption of English text. The top card has value 1. The bottom card has value 26. Since this deck is for the English alphabet, there are 28 cards and the two jokers are 27 and 28.

Initial deck:

```
Index:       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Card value:  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

             ^                                                                                ^
             |                                                                                |
          1 is the                                                                      26 is the
          top card                                                                     bottom car
d
```

Step 1: Swap the small joker with the card that is under it in the deck. So, we swap 27 and 2, to get this deck:

```
Index:       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Card value:  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 the big joker two places down the deck by first swapping it with 3 and then swapping it with 6. It ends up between 6 and 9:

```
Index:       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Card value:  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. Find every card that is above the highest joker in the deck (in this case: every card to the left of 28). Now find every card to the right of the lowest joker in the deck (in this case: every card to the right of 27). Now take these two groups of cards and switch their positions. In this case, the cards between 5 and 26 switched places with the cards betwee 1 and 6:

```
Index:       0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

```
Card value:  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: Find the bottom card and take its value (6 in this example). Take the first 6 cards from the top of the deck (5, 8, 11, 14, 17, and 20). Put them directly on top of the bottom card:

```
Index:        0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Card value: 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. Find the card at index 23 in the deck which in this case is 11. Since 11 isn't a joker card (it isn't 27 or 28), we are done with the round. The generated keystream value is 11.

```
Index:        0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

Card value: 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
```

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.

Example: Encrypting and Decrypting

Let's say we want to encrypt the plaintext message Hundred Acre Wood. We start by cleaning the message by removing non-letters and capitalizing the letters, which gives us HUNDREDACREWOOD. Next, we convert these letters to numbers:

```
            H  U  N  D  R  E  D  A  C  R  E  W  O  O  D
            8 21 14  4 18  5  4  1  3 18  5 23 15 15  4
```

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

```
            7  8  4  6  3 21 14 19  3 11 12  2  1 17 21
```

Now add the numbers from the letters and the generated keystream values together pairwise, modulo 26. (If the letters were not from English, the 26 would be replaced by the number of letters in the alphabet.) The "modulo 26" part means that if the sum of the pair of numbers is greater than 25, then subtract 26 from the sum. (For example, 1 + 8 = 9, which is not greater than 25, so (1 + 8) modulo 26 is 9. But, 11 + 17 = 28, which is greater than 25, so we compute 28 - 26 = 2, and (11 + 17) modulo 26 is 2.) Then, convert the resulting sums to letters to produce the encrypted message (the ciphertext):

```
        8  21 14  4 18  5  4  1  3 18  5 23 15 15  4
  +  7   8  4  6  3 21 14 19  3 11 12 2   1 17 21
     ---------------------------------------------
     15  3 18 10 21  0 18 20  6  3 17 25 16  6 25
      O  C  R  J  U  Z  R  T  F  C  Q  Y  P  F  Y
```

To decrypt this message, the recipient would start with the same deck with which the encryption was started, and generate the same 15-number keystream. They would then convert the ciphertext to numbers. Then, instead of adding corresponding numbers from the keystream and ciphertext, they would pairwise-subtract the keystream from the ciphertext, modulo 26. (Again, if the letters were not from English, the 26 would be replaced by the number of letters in the alphabet.) That is, if the subtraction gives you a negative number, add 26 to the result. (For example, 9 - 8 = 1, which is not negative, so (9 - 8) modulo 26 is 1. But 2 - 17 = -15, which is negative, so add 26 to get 11, and then (2 - 17) modulo 26 is 11.) The decryption back to HUNDREDACREWOOD looks like this:

```
     15  3 18 10 21  0 18 20  6  3 17 25 16  6 25
  -   7  8  4  6  3 21 14 19  3 11 12  2  1 17 21
     ---------------------------------------------
      8 21 14  4 18  5  4  1  3 18  5 23 15 15  4
      H  U  N  D  R  E  D  A  C  R  E  W  O  O  D
```

Files to Download

Please download the **Assignment 2 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 a program that builds and displays a user interface to run the encryption and decryption process. It will work properly once you have completed the assignment.

A nearly-empty file: `cipher_functions.py`

You'll put all of the functions that you write in this file. It's imported by `cipher_program.py`. It includes the header for one of the functions you are to write and also two constants: `ENCRYPT` and `DECRYPT`.

A file for keeping track of your testing: `cipher_tester.py`

This file contains a set of tests for function `clean_message`, one of the functions you will write. We encourage you to use this file to test all of your A2 functions, following the example we have given. Read it carefully, there are several useful tips in the module docstring.

It may help to think about three kinds of tests:

- Your docstring examples that you created while following the FDR
- More general-case tests for typical situations that you envision
- Edge cases: tests that are at the limits of what is allowed, such as empty strings, empty lists, jokers at the top or bottom of the deck, and so on.

The style checker: `a2_checker.py` and related files

This file contains a set of tests to make sure you're following the Python style guidelines and your return types. It does not check for correctness; you will need to do that yourself. This program uses `checker_generic.py` and the contents of the `pyta` subdirectory.

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

As you know, we are representing a deck of cards in Python as a list of integers. A deck mustl contain at least 3 cards, since every alphabet has at least 1 letter, and there are 2 jokers. We provide sample decks in `deck1.txt` and `deck2.txt`. As you come up with interesting scenarios, you might want to create more deck files. (But you won't submit them.)

Some message files: `message_file1.txt` and `secret?.txt`

Messages to encrypt or decrypt will be stored in text files. Message text files contain multiple messages - one per line. Imagine that we're encrypting or decrypting a message file that contains multiple lines. 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 lines of a message file.

The starter code archive contains one text file named `message_file1.txt` that contains two plaintext messages. There are also seven text 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`. The encrypted version of `message_file1.txt` is in `secret1.txt`.

File `secret7.txt` contains the ciphertext from "The Encryption Algorithm" section of the handout. You can decrypt the message in `secret7.txt` using the deck in file `deck1.txt`.

Design Notes

The main goal of our program is to encrypt and decrypt text. As you'll see in the table below, we designed our code by breaking down the major tasks of encrypting and decrypting text into smaller subtasks. For example, we have functions to encrypt and decrypt a single letter. Those functions need a keystream value, so we have a function to generate that value. And to generate a keystream value, we follow the steps of the chosen algorithm, so we included a function for each of those steps. As part of completing those steps, cards are swapped multiple times, so we introduced a function to swap cards. In addition, we have functions to read messages and decks from files and store them using the data structures we selected.

In general, we aimed for short functions, each with a key purpose. We also introduced functions in order to eliminate duplicate code, such as when code (e.g., swap cards) was needed in more than one of our functions.

What to do

We have performed a top-down design on the algorithm presented in the handout and have come up with quite a few functions. Write all functions in the file `cipher_functions.py`.

We present the functions in what we think is a sensible order, but you're welcome to jump around as you work. You'll notice that the functions near the top of the table don't require you to use lists.

Follow the Function Design Recipe we have been using in class. We expect your docstrings to contain a type contract, a description, and two examples, where appropriate.

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 part marks for correctly implementing some of the functions.

Preconditions

Preconditions come in two forms: the type contract and the function description.

The parameter types in the type contracts serve as preconditions: you should assume that the argument types match the expected parameter types.

The first sentence in each function description below describes the preconditions.

Except for function `is_valid_deck`, each function with a deck of cards as a parameter has as a precondition that the parameter is a valid, non-empty deck. A valid deck contains every integer from 1 up to the number of cards in the deck, with no duplicate numbers. The number of cards in the deck will depend on the chosen alphabet, unless otherwise specified. As such, your functions must handle any alphabet that contains at least one letter. Function `is_valid_deck` will be given a `List[int]`, but that list may or may not represent a valid deck.

Functions to design and develop

List of functions to implement in `cipher_functions.py`.

Function name: (Parameter types) -> Return type	**Full Description (paraphrase for your docstring)**
`clean_message` `(str) -> str`	The parameter represents a message that may contain any characters, including letters, punctuation, and spaces. Return a copy of the message that includes only its alphabetic characters, where each of those characters has been converted to uppercase. This function only needs to work for the 26 character English alphabet.
`encrypt_letter` `(str, int) -> str`	The first parameter represents a single uppercase letter and the second represents a keystream value. Apply the keystream value to the letter to encrypt the letter, and return the result. **This function only needs to work for the 26-character English**

	alphabet.
`decrypt_letter` `(str, int) -> str`	The first parameter represents a single uppercase letter and the second represents a keystream value. Apply the keystream value to the letter to decrypt the letter, and return the result. **This function only needs to work for the 26-character English alphabet.**
`is_valid_deck` `(List[int]) -> bool`	The parameter represents a candidate deck of cards that may or may not be valid. A valid deck contains every integer from 1 up to the number of cards in the deck. Return `True` if and only if the candidate deck is a valid deck of cards.
`swap_cards` `(List[int], int) -> None`	The first parameter represents a valid deck of cards and the second parameter represents a valid non-negative 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 this function doesn't return anything. The deck is to be mutated.
`get_small_joker_value` `(List[int]) -> int`	The parameter represents a valid deck of cards. Return the value of the small joker (value of the second highest card) for the given deck of cards.
`get_big_joker_value` `(List[int]) -> int`	The parameter represents a valid deck of cards. Return the value of the big joker (value of the highest card) for the

	given deck of cards.
`move_small_joker` `(List[int]) -> None`	The first parameter represents a valid deck of cards. This is Step 1 of the algorithm. Swap the small joker with the card that follows it. Treat the deck as circular. Note that this function doesn't return anything. The deck is to be mutated.
`move_big_joker` `(List[int]) -> None`	The parameter represents a valid deck of cards. This is Step 2 of the algorithm. Move the big joker two cards down the deck. Treat the deck as circular. Note that this function doesn't return anything. The deck is to be mutated.
`triple_cut` `(List[int]) -> None`	The parameter represents a deck of cards. This is Step 3 of the algorithm. Do a triple cut on the deck. Note that this function doesn't return anything. The deck is to be mutated.
`insert_top_to_bottom` `(List[int]) -> None`	The parameter represents a valid deck of cards. This is Step 4 of the algorithm. Examine the value of 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 the big joker, use the value of the small joker as the number of cards. Note that this function doesn't return anything. The deck is to be mutated.
	The parameter represents a valid deck of cards.

`get_card_at_top_index` `(List[int]) -> int`	This is Step 5 of the algorithm. Using the value of the top card as an index, return the card in the deck at that index. Special case: if the top card is the big joker, use the value of the small joker as the index.
`get_next_keystream_value` `(List[int]) -> int`	The parameter represents a valid deck of cards. Repeat all five steps of the algorithm until a valid keystream value is produced, then return that valid keystream value.
`process_messages` `(List[int], List[str], str) -> List[str]`	The first parameter represents a valid deck of cards. The second represents a list of messages. The third parameter is either `ENCRYPT` or `DECRYPT` — the constants defined at the top of file `cipher_functions.py`. Return a list of encrypted messages if the third parameter is `ENCRYPT` or decrypted messages if the third parameter is `DECRYPT`. The messages are returned in the same order as they are given. Note that the first parameter may also be mutated during a function call.

A2 Checker

We have provided a checker program, `a2_checker.py`, that tests two things:

- whether your code follows the Python and CSC108 Python Style Guidelines.
- whether your functions have the correct parameter and return types, and
- whether your functions are named correctly, have the correct number of parameters, and return the correct types.

To run the checker, open `a2_checker.py` and run it. Note: the checker file should be in the **same** directory as your `cipher_functions.py`, as provided in the starter code zip file. Be sure to scroll up to the top and read all messages.

If the checker passes for both style and types:

- Your code follows the style guidelines.
- Your function names, number of parameters, and return types match the assignment specification. **This does not mean that your code works correctly in all situations.** We will run a *different*

set of tests on your code once you hand it in, so be sure to thoroughly test your code yourself before submitting.

If the checker fails, carefully read the message provided:

- It may have failed because your code did not follow the style guidelines. Review the error description(s) and fix the code style. Please see the **PyTA documentation (http://www.cs.toronto.edu/~david/pyta/)** . for more information about errors.
- It may have failed because:
    - you are missing one or more functions,
    - one or more of your functions is misnamed,
    - one or more of your functions has the incorrect number or type of parameters, or
    - one of more of your function return types does not match the assignment specification.

  Read the error message to identify the problematic function, review the function specification in the handout, and fix your code.

Make sure the checker passes before submitting.

Running the checker program on Markus

In addition to running the checker program on your own computer, run the checker on MarkUs as well. You will be able to run the checker program on MarkUs 5 times every 24 hours (note: we may have to reduce this number if MarkUs has any issues with this load). This can help to identify issues such as uploading the incorrect file.

First, submit your work on MarkUs. Next, click on the "Automated Testing" tab and then click on "Run Tests". Wait for a minute or so, then refresh the webpage. Once the tests have finished running, you'll see results for the Style Checker and Type Checker components of the checker program (see both the Automated Testing tab and results files under the Submissions tab). Note that these are not actually marks -- just the checker results. If there are errors, edit your code, run the checker program again on your own machine to check that the problems are resolved, resubmit your assignment on MarkUs, and (if time permits) after the 24 hour period has elapsed, rerun the checker on MarkUs.

Testing your Code

We strongly recommend that you test each function as soon as you write it.

You might be tempted to just plow ahead and write all of the functions and hope everything *works, but then it will be difficult to determine whether your program is encrypting/decrypting correctly.*

As you get more insight into your program and how it should represent and manipulate a deck of cards, revisit your tests to see if you can improve or extend them.

We strongly recommend that you write and test the functions one or two at a time.

As usual, follow the Function Design Recipe. Once you've implemented a function, run it on your examples in the docstring, and think about other possible tests. Here are a few tips:

- Be careful that you test the right thing. Some functions return values while others modify the deck in place. Be clear on what the functions are doing before concluding that your tests work.
- 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 test it.

When you're ready to test using the overall algorithm (`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. Remember that when you encrypt a message file and then want to decrypt it, you need to start the decryption with the same deck that started the encryption.

Remember to run the checker!

Additional requirements

- Do **not** call `print`, `input`, or `open`, except within the `if __name__ == '__main__'` block.
- Do **not** modify or add to the import statements provided in the starter code.
- Do **not** add any code outside of a function definition.
- Do **not** mutate objects unless specified.
- Do **not** use Python language features for sorting that we haven't covered in this course, like the optional parameter `key` in function `sorted`.

Marking

These are the aspects of your work that will be marked for Assignment 2:

- **Correctness (80%):** Your functions should perform as specified in this assignment handout. 7 Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests, not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.
- **Coding style (20%):**
  - Make sure that you follow the **Python Style Guidelines** that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with one exception: docstrings may be evaluated separately. For each occurrence of a **PyTA error (http://www.cs.toronto.edu/~david/pyta/)** , a 1 mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.
  - You are encouraged to write helper functions, both to avoid repetitive code and to make the program easier to read. If a function body is more than about 20 lines long, introduce helper functions to do some of the work, even if they will only be called once.
  - All functions, including helper functions, should have complete docstrings including preconditions when you think they are necessary.
  - Your variable names should be meaningful and your code as simple and clear as possible.
  - Hard-coding 27s and 28s in multiple places in your code is not a good idea. We will test with decks of different sizes, so always be sure to calculate the value of the two jokers based on the size of the deck. (Hint: you have to write two functions that calculate the values of the two jokers. Call them!)

No Remark Requests

No remark requests will be accepted. A syntax error could result in a grade of 0 on the assignment. Before the deadline, you are responsible for running your code and the checker program to identify and resolve any errors that will prevent our tests from running.

What to Hand In

The very last thing you do before submitting should be to run the checker program 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 file `cipher_functions.py`. Your file must be named exactly as given (no capital letters, and please don't change the filename).

NWAJOPQD!