Assignment 4

Objectives: Practice making functions and importing modules. Become familiar with lists and basic tools to use them.

Note: Include DocStrings in each script you submit analogous to the following:

```
"""This script takes a song and makes it into a cannon.


Submitted by Mauricio Arias, NetID ma6918
[Include a short explanation here of how the script does that.]
"""
```

Part 1. Old encryption methods and RSA encryption
Task 1.1. Make an old encryption function.
We will make very simple encryption and decryption functions that rely on shifting the letters in a string. For this we will use a list that contains the alphabet: `alphabet`. Lists in Python have several operations they can perform themselves; we call this type of operations "methods." We saw in class how to join two lists with the + operator. That is available because lists provide a method called "__add__". A more complex method is "index." This operator returns the position in which an element is located. We will study this in more detail later. So, `alphabet.index("a")` yields 0 since "a" is at the beginning of the list (see alphabet.txt) and we call this position 0. Notice the period separating the list from the name of the method. Python provides the function `len()` that returns how many elements are in a list. To retrieve the element at a specific position, say `pos`, we use the following notation `alphabet[pos]`. To add an element at the end we use the method append() or add a list with a single element.

Make a module in which you define the list `alphabet`: only use the characters provided in the file alphabet.txt. In this module make a function that takes a string (**assume** it only uses characters in `alphabet`) and a number x. This function returns a string in which each character is replaced by the character that is x positions to the right. As an example shift("a", 4) is "e," shift("B", 6) is "H" and so on and so forth. Note: if the new position runs off the end of the list, make it continue at the beginning of the list. For example, shift("#", 2) is "b." Make also a function that decrypts a string so encrypted.

Make a script that uses this module to encrypt or decrypt a user provided string by using a "secret key," which is a global variable defined in the main script. The script should keep asking whether encryption (E) or decryption (D) is desired. A non-announced option "*" would request an integer key greater than 0 to be entered: verify that condition. This key will be used for all operations until it is changed again. It is initially 3. Only those three values are accepted: simply pressing enter terminates the program. After that selection it should request a non-empty string to process and print the result of the operation.

Submit your script as `[NetID]_simple_encryption.py. (2.5 pts)`

Task 1.2. Complete an RSA encryption script.
The module provided by utilities_RSA provides some functions that are crucial for RSA encryption. We made one previously: gcd.

To the utilities module, add a function raise_mod() that takes 3 positive integers: return 0 otherwise. The three integers are a base, an exponent and a modulus. Calculate a product by following this pseudocode:

Make `product` = 1
Make `factor` = `base`
Iterate over all the bits of the exponent starting with the least significant one:
    if the current bit is 1, multiply `product` by `factor` and take mod `modulus`
    make `factor` = `factor`^{2} mod `modulus`
Return `product`

Note: Notice that the loop is essentially the same as extracting the bits as we did in Assignment 2. This function calculates base^{exp} mod modulus. (If you are so inclined, try to understand it. It uses the fact that `k * m mod modulus` is equivalent to (`k mod modulus) * (m mod modulus) mod modulus`.) It is particularly useful when the numbers are very big.

In RSA, to encrypt one uses a public key. To decrypt one uses a secret or private key. The public key is freely accessible while the private key is kept secret. (Notice the difference with the system in Task 1.1.) Calculating the private key is trivial if one starts with two large primes. However, these large primes are also kept secret. The process to generate the private key uses three numbers: two large primes (p_1 and p_2) and one small one (p_3). It is based on a very old result $A^{tot(p1 * p2) / gcd(p1 - 1, p2 - 1)} = 1 \text{ mod } (p_1*p_2)$ based on Euler's work. With this result it is easy to obtain two numbers key_1 and key_2 that make $(A^{key1})^{key2} = A$ mod n. (I might add some details for those interested in a supplemental file.) The first key is called the public key and it is used for encoding, while the second key is called the private key and it is used for decoding.

To encrypt one makes cipher = A^{key1} mod n, where A is an integer that represents the message. To decrypt one takes cipher^{key2} mod n, which yields the original message. (Check why if you desire using the results mentioned in the previous paragraph.)

Make a script that encrypts and then decrypts a number using RSA. Ask the user for two large prime numbers. The small prime number or public key will be taken as $2^8 + 1$. Calculate $n = p_1 * p_2$. The number to be encrypted has to be smaller than that. (Notice that n is not secret and the prime numbers can be obtained by factoring n. This is a very time consuming operation due to the large prime numbers used though.)

Calculate the private key by using the function get_private_key(p_1, p_2, public_key) from the utilities module. Ask the user for a number to be encrypted. Encrypt the number and decrypt it afterwards to show the user that the system works. Keep asking until the user presses Enter without giving numbers.

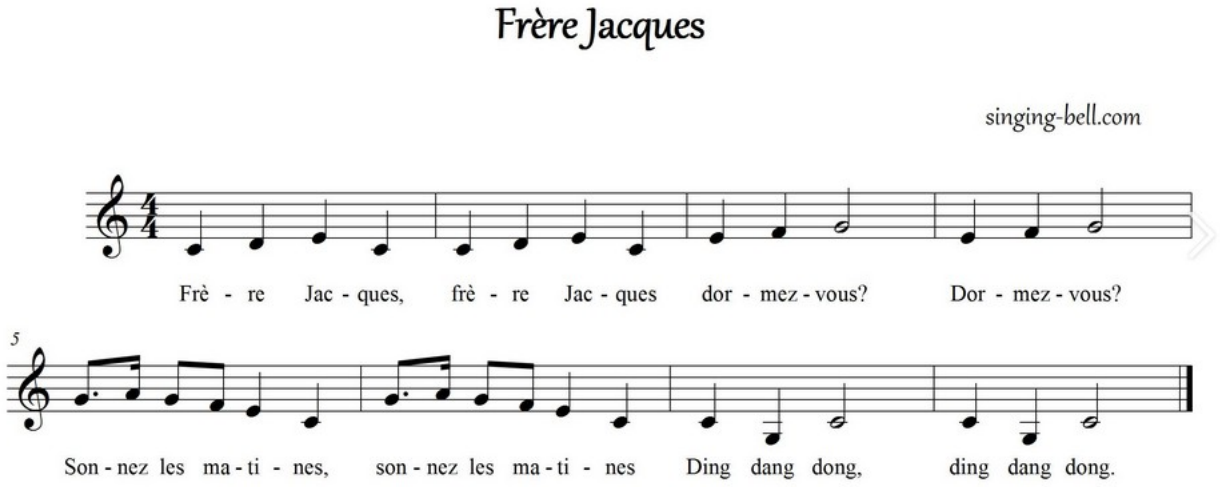
Submit your script as `[NetID]_utilities_RSA.py` and `[NetID]_basic_RSA.py. (2.5 pts)`

Part 2. Making a file that plays a simple song.

Task 2.1. For this task we will reproduce a song according to instructions provided in two lists: one list contains the notes for a song and the other list contains the duration for each note. A song is provided in the module songs.py.

Follow these steps:

Import WAV_utilities_v2 from a lib folder. Do not use **from … import *.** That is unacceptable.
Import the songs file from the local folder.
Load the song into global variables: there is only one song there at this time.
Calculate the total duration for the song.
Calculate the total number of samples.
Start a `for` loop and keep track of how long the current note has played.
Play each note for the duration specified.
Go through all the notes switching notes once the time for each note is exhausted.



Submit your script as `[NetID]_player.py`. Submit also your wav file as `[NetID]_song.wav`. (2.5 pts)

Task 2.2. For this task we will make a music_utilities module with a tool to transpose songs. Sometimes it is useful to change the key for a song. To do this all notes in a song are shifted equivalently in one direction. The easiest way to describe this is probably by using the idea of semitones for the equal temperament system. In this music system an octave is composed of 12 tones equally spaced. These correspond to the ones listed in the utilities file. Given a frequency for a tone, the frequency of the next semitone is very easy to calculate: to move up a semitone multiply the frequency by $2^{1/12}$. To go down a semitone do the inverse operation. (Note: in other tonal systems this process is different.)

Use this utility in a script in which the user inputs the number of semitones to transpose and gets a song appropriately transposed.

Submit your new module as `[NetID]_music_tones.py`. Submit your script as `[NetID]_transposing_player.py`. Submit also a transposed song in a wav file as `[NetID]_[X]_transposed_song.wav`, where X is the number of semitones transposed. (2.5 pts)