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.]
"""
```

Include also one-line DocStrings for your functions.

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 can join two lists with the + operator: [1, 100] + [5] yields [1, 100, 5]. That operation 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 all this in more detail later. So in a provided list, `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, let's say `pos`, we use the following notation `alphabet[pos]`. To add an element at the end we use the method append() or concatenate 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) returns "e," shift("B", 6) returns "H" and so on and so forth. 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. An unannounced option "*" requests a secret key (type `int`) 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 (E, D and *) 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`. Don't forget to submit your module as `[NetID]_encryption_module.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 a simple version of one previously: gcd. (As an exercise, if you want, try writing a version of the Extended Euclidean Algorithm. Don't include it in your code but we can talk about it during office hours.)

To the utilities module, add a function raise_mod() that takes 3 positive integers: return 0 otherwise. The three integers are *base*, *exponent* and *modulus*. The function provides the result of base raised to the exponent mod modulus: base**exp mod modulus. Using small numbers, test the result generated by your code against this basic formula. However, unlike the formula, the code below handles very large numbers well. Calculate `product` by following the pseudocode paying special attention to the indentation:

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. (See Assignment 2.) This function calculates base^{exp} mod modulus. (If you are so inclined, try to understand it: see also the supplemental file.) This shortcut is particularly useful when the numbers are very big as in RSA.

In RSA, to encrypt one uses a *public key*, which is shared freely. To decrypt one uses a *private key*, which is kept secret. See the details for how they are used below. (Notice the difference with the system in Task 1.1. In particular what happens if the encrypted message is captured while in transit?)

It turns out that calculating a suitable private key is trivial if one starts with two large primes. Therefore, these large primes *are kept secret*. The process to generate the private key uses three numbers: the two large primes (p_1 and p_2) and one small one (p_3) aka the public key. It is based on a very old result based on Euler's work and his totient (tot) function, namely $A^{tot(p1 * p2) / gcd(p1 - 1, p2 - 1)} = 1 \text{ mod } (p_1 * p_2)$. With this result, it is easy to obtain two numbers key_1 and key_2 that make a reversible process $(A^{key1})^{key2} = A \text{ mod } n$ possible. (See 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 a message represented by an integer A, make cipher = A^{public_key} mod n. To decrypt, take M = $cipher^{private\_key}$ mod n, which yields the original message back, i.e. M = A. (Check *why* if you desire or read the supplemental file.)

Make a script that encrypts and then decrypts a number using RSA. Ask the user for two large prime numbers from the provided large-prime-numbers table: we won't check that they are primes. The small prime number or public key will be taken as $2^8 + 1$. Calculate $n = p_1 * p_2$. Ask for a number smaller than n to be encrypted: verify that it is; otherwise decryption won't work. (Notice that *n is not secret* and the prime numbers can be obtained by factoring n. This whole system is amazingly simple. However, this is a very time consuming operation due to the large prime numbers used. *The security of RSA relies on this!*)

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.

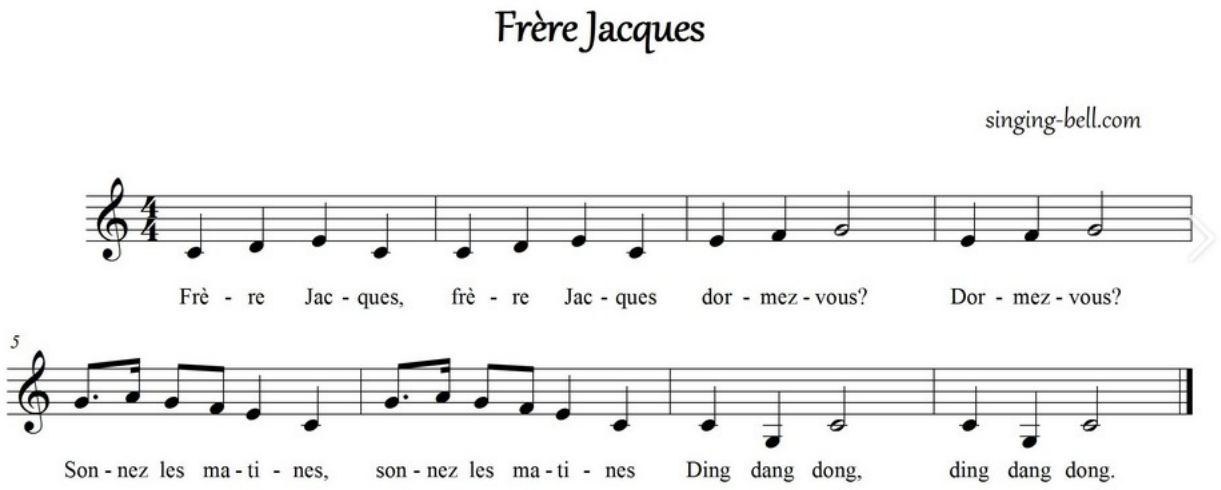
Submit your script as `[NetID]_utilities_RSA.py` and `[NetID]_basic_RSA.py`. (2.5 pts)

Part 2. Make a .wav file that plays a simple song.
These tasks build upon the framework we set up for assignment 2. Borrow code and strategies from there. I recommend you use the published solution for assignment 2 as your framework: see the content section in Brightspace.

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. Lists for a song are provided in the module songs.py. (Later in the semester, we will learn much better tools to store a song.)

Using the file provided as solution to assignment 2 as a foundation, **follow these steps**:

Import WAV_utilities_v2 from a Lib folder. Do not use from … import *. That is unacceptable.
Import the songs file. (Indicate in a comment where this file resides.)
Load the song into global variables: notes and duration. There is only one song there at this time.
Calculate the total duration for the song.
Calculate the total number of samples based on the total duration.
Start a `for` loop for all the samples.
Inside of the loop keep updating how long the current note has played.
Keep playing the same note for all the iterations necessary to exhaust the time allotted for it.
Go through all the notes switching notes once the time for the previous 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.)

Notice that moving up 2 semitones is the same as moving up one semitone and then moving up another one. Handle the general case in your code. Check the help files if you get stuck or after you are done for comparison purposes.

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)

Optional challenge for fun only
If you want a challenge make a canon using the song. One way to do that is to play the song two times. While it is playing, another repetition of the song starts when the main song is half way through its first repetition. They both play (different parts of the song) at the same time and the notes playing simultaneously, while different, complement each other rather nicely. To play two notes simultaneously, simply add the numbers of both sound samples into one composite_sample number before encoding the latter in the file. Alternatively, you can play the main song on the left channel and the overlaid song in the right channel to add stereo sound to the experience.

Again don’t submit this for the assignment but you can share it with me via a link to your google drive in an email or during office hours.