CRYPTOGRAPHY USING THE LOGISTIC GROWTH FUNCTION

For this PA, you'll write two programs that will use three of the functions you coded in Lab #11. One program will be for encryption and the other for decryption. Please see the info on the first page of PA #5 or PA #6 if you don't know the grading et al. routine yet!

Header information. Your program must include the following information in the header.

- Name
- Date
- Programming Assignment #7a or #7b
- Name of program
- Brief description and/or purpose of the program; include sources

Background

We want to devise a way of encrypting and decrypting messages so that only someone with the password can encode or decode them. In Lab #11, you used the Caesar cipher to encrypt and decrypt messages. The Caesar cipher uses a constant offset for each character, but it's just too easy to crack this cipher. In this PA, *you're going to use a different offset for every single character and, in addition, the offset will be nearly randomly determined.*

To do this, we'll use the logistic growth function you coded in lab. Recall that this function always returns values between 0 and 1, and if an amplitude close to 4 is used, the difference between the two initial populations differing only slightly will quickly diverge with passing generations as shown below for initial populations of 0.30001 and 0.30000 and an amplitude of 3.99:

```
Gen.         Pop. A                  Pop. B
1      0.8379159596010001      0.8379
2      0.5418930889452718      0.5419361241000001
3      0.9904974267035079      0.9904830323669228
4      0.03755497484497509     0.03761131589099112
                    [output deleted]
197    0.6269698396798223      0.011329613171045869
198    0.933175852648604       0.04469299961539693
199    0.248811135921798       0.17035518624909252
200    0.7457475767067793      0.5639238441008467
```

If we multiply the population values by 96 and then take their integer parts, we end up with the following:

```
Gen.    A     B
1       80    80
2       52    52
```

```
3       95    95
4       3     3
[output deleted]
197     60    1
198     89    4
199     23    16
200     71    54
```

Notice that in the second and third columns the values range from 1 to 95 (in fact, when more generations are used, they actually range between 0 and 95) and, thus, they work perfectly as offsets for cryptography! Initially the two sequences of offsets aren't random, but eventually they are. We can use the values for late generations as offsets for our ciphering, but this would be something of a hassle. Instead we'll use a more refined and fun way of generating the amplitude and what we'll call a seed using the function `gen_amp_seed()` compliments of Prof. John Schneider.

The non-void function `gen_amp_seed()` takes one string argument, a password, and returns an amplitude amp and a seed (x in the logistic growth function). In `gen_amp_seed()`, even a change of one bit in one byte in the password will result immediately in a completely different sequence of offsets (to the chagrin of many students!). What is important in using this function is to note what it returns (see More Info below for more information which you're free to ignore)!

Program Requirements

You'll write two programs, one for encryption and one for decryption. Because the encryption and decryption programs are so similar, grading emphasis will be on the encryption program. However, there is an extra task for the decryption program worth 5%, so be on the look out for it!

What you must do initially:

- Create a file called **`cipher.py`**.
- Copy **`gen_amp_seed()`** into this file.
- Copy the **`log_growth()`**, **`cipher_chr()`**, and **`decipher_chr()`** functions from Lab #11 into this file.

Note that you do not want to have a `main()` function in `cipher.py`.

Encryption Program

Recall that in your lab you coded a function to encrypt a single character called `cipher_chr()`. This function should now be in your `cipher.py` file which you're going to use as a module. In the first part of this program, you're going to code a function called `line2cipher()`. It's similar to the `clear2cipher()` function you wrote in your lab.

After `line2cipher()` is working correctly, add it to your `cipher.py` file. The rest of the encryption program consists of the program `pa7a.py` which is mostly a single function.

Let's look at the functions needed for the encryption program.

- **`line2cipher()`:** A non-void function with three parameters, an amplitude and seed and a string (line) of clear (not encrypted) text, which returns an amplitude, seed, and string (line) of ciphered (encrypted) text. What you need to do to code this function:
  - Define the function header with three parameters as described above.
  - Initialize a string accumulator as an empty string.
  - Create an iterating `for`-loop that loops through each character in the string passed to the function.
  - In the loop body:
    - Create an offset by multiplying the seed by 96 and using the `int()` function with the result.
    - Call the function `cipher_chr()` with the character and offset as arguments and add the returned result to the string accumulator.
    - Generate a new seed by calling the `log_growth()` function with the current amplitude and seed as arguments.
  - Return the amplitude, seed, and line of ciphered text. The values of amplitude and seed will be used the next time `line2cipher()` is called which will be to cipher the next line.
  - Add a docstring.

  See the example below to verify this function is working correctly and then add it to `cipher.py`.
- **`main()`:** A void function with no parameters that will be used in the program `py7a.py`. It prompts the user for a password, a clear text file name (input), and a cipher text file name (output). It calls `gen_amp_seed()`, and then for every line in the input file, it calls `line2cipher()` and prints the resulting ciphered text to the output file. To code this function:
  - Write the function header.
  - Prompt the user for a password (see example below).
  - Prompt the user for a clear text file name (see example below) to read and open it, all in one statement.
  - Prompt the user for a cipher text file name (see example below) to print to, and open it, all in one statement.
  - Call `gen_amp_seed()` to generate the amplitude and initial seed.
  - Create an iterating `for`-loop for each line in the input file.
  - In the loop body:
    - Call `line2cipher()` with three arguments, the amplitude and see, and the line to cipher. Strip the line to delete any whitespace or it will be ciphered, too! Assign the output to three lvalues. The amplitude and seed should have the same names as used as arguments because they'll be used in the next iteration. Call the ciphered line whatever you think is appropriate.

- Print the ciphered line to the output file. You can use the `.write()` method as well, but then you need to add a newline.

– Close both the input and output files.

– Add a docstring.

Next, you need to create `pa7a.py`. To do this:

- Create a file called `pa7a.py`.
- Add a header as instructed above and comments as needed.
- Import your `cipher.py` file using `from cipher import *`.
- Add your `main()` function.
- Add a call to main.

To run the example for `line2cipher()`, use the IDLE Shell window, and follow the instructions below. Note that your files must be in the same directory as your IDLE session. If you need help with this, contact me or a TA. As usual, what you enter is given in boldface.

```
>>> # Import cipher module.
>>> from cipher import *
>>> # Set password.
>>> password = ``I've got a secret!''
>>> # Call gen_amp_seed() to generate amplitude and seed.
>>> amp, seed = gen_amp_seed(password)
>>> amp, seed
(3.992407363218947, 0.1910140581380398)
>>> # Create line to cipher.
>>> line = 'This is a test.'
>>> amp, seed, ciphered_line = line2cipher(amp, seed, line)
>>> amp, seed
(3.992407363218947, 0.5416116867372278)
>>> ciphered_line
'fDd(`^;|nMtgz1~'
```

The last line is the ciphered text `'fDd(`^;|nMtgz1~'`. Notice that the space characters appearing in the fifth, eighth, and tenth positions were encrypted to `` ` ``, `|`, and `M`, i.e., 3 different characters. Also, the amplitude returned by `line2cipher()` is the same as the original amplitude passed to the function, but the seed is different. When you get the correct results, add **`line2cipher()`** to your **`cipher.py`** file.

To test `pa7a.py`, your encryption program, close your IDLE session and start a new one, or restart your IDLE session using the correct command in the Shell's Shell menu. Then open your `pa7a.py` program and run it. At the prompts in the Shell window, enter text as given below. You need to have downloaded **`clear.txt`** to use as the clear text input file.

```
Enter password: Oh woe is me.
Enter clear text (input) file name: clear.txt
Enter cipher text (output) file name: cipher.txt

lQQ]eWPyyiV_Cm1iVji,XMcml}cTsi*N{v@jmM
>=}m6k''<suz;^@{$Mn% Gf#fOof#$8xv($a=lozs5 b|.2 boo?k ow@e/

G%ImOc|Qw'fNtii0>gw$e:ZFh|tka85d#(@u!i(Edg$SSjl3c~9ck8`R
```

The ciphered text isn't part of the output but is provided for you to check your program. If your program is correct, you will get this **exact** ciphered text. Note that you'll need to open the text file to look at it using either Notepad (Windows) or TextEdit (macOS).

Decryption Program

Recall that in lab you coded a function to decrypt a single character called `decipher_chr()`. This function should now be in your `cipher.py` file. For the first part of the decryption program you're going to code a function called `line2clear()` which is similar to the `cipher2clear()` function your wrote in lab.

After `line2clear()` is working correctly, add it to your `cipher.py` file. The rest of the decryption program consists of the program `pa7b.py` which is mostly a single function.

Next, let's look at the functions needed for the decryption program.

- **`line2clear()`:** A non-void function with three parameters, an amplitude and seed and a string (line) of ciphered (encrypted) text, which returns an amplitude and seed and a line of clear text. This function is essentially a copy of `line2cipher()` except that it calls `decipher_chr()`. Thus, to code this function:
  - Make a copy of `line2cipher()` and call it `line2clear()`.
  - Change the lvalue names appropriately.
  - Change `cipher_chr()` to `decipher_chr()`.
  - Return the amplitude, seed, and line of clear text.
  - Change the docstring.

  That's all there is to it. See the example below to verify this function and then it add it to `cipher.py`. Next move on to the function for `pa7b.py`.

- **`main()`:** A void function with no parameters that is used in the program `pa7b.py`. This function is essentially a copy of the `main()` function in `pa7a.py` except that it calls `line2clear()`. Thus, to code this function:
  - Make a copy of your previous `main()` function.
  - Change the prompts for the input and output files appropriately.

- Change the lvalue names appropriately (actually, there's only one you should need to change).
- Change `line2cipher()` to `line2clear()`.
- Change the docstring.

To create `pa7b.py`:

- Make a copy of `pa7a.py` and call it `pa7b.py`
- Change header and comments as needed.
- Replace the `main()` function with the one you just wrote.

To run the example for `line2clear()`, use the IDLE Shell window, and follow the instructions below. Again, your files must be in the same directory as your IDLE session.

```
>>> # Import cipher module.
>>> from cipher import *
>>> # Convert clear text to cipher text.
>>> password = ``I've got a secret!''
>>> amp, seed = gen_amp_seed(password)
>>> amp, seed
(3.992407363218947, 0.1910140581380398)
>>> line2clear(amp, seed, 'fDd(`^;|nMtgz1~')
(3.992407363218947, 0.5416116867372278, 'This is a test.')
>>>
>>> # Try again, but make a small change to the password.
>>> password = ``I've got a secret.''
>>> amp, seed = gen_amp_seed(password)
>>> amp, seed
(3.9901230079996655, 0.5833452084069134)
>>> line2clear(amp, seed, 'fDd(`^;|nMtgz1~')
(3.9901230079996655, 0.05714244222193654, '.FY`cRo}j<;kj[ ')
```

As can be seen in the first part of this example, the `line2clear()` function successfully decrypts the cipher text. Perhaps more interestingly, we see what happens when the password is changed just a little. Even a small change in the password leads to radically different results. Hence, without the password, it would be very difficult to crack this result. Once you have the correct results, add `line2clear()` to your `cipher` module.

To test `pa7b.py`, your decryption program, close your IDLE session and start a new one, or restart your IDLE session using the correct command in the Shell's Shell menu. Then open your `pa7b.py` program and run it. At the prompts in the Shell window, enter text as given below.

```
Enter password: Oh woe is me.
Enter cipher text (input) file name: cipher.txt
Enter clear text (output) file name: myclear.txt
```

Compare the contents of the `clear.txt` and `myclear.txt` files. They should be identical. If not, debug your program!

REQUIRED TASK: I've included a secret message on our class website (**`secret.txt`**) and in Canvas. I'll post the password in Canvas Chat. Add the decrypted file to your assignment so your TA knows you were able to decrypt it. Call it **`no_secret.txt`**.

More Information

The following code demonstrates the behavior of `gen_amp_seed()`:

```
>>> # Import gen_amp_seed().
>>> from gen_amp_seed() import *
>>>
>>> # Call gen_amp_seed() with an argument of 'undercover'.
>>> password = input('Enter password: ')
Enter password: undercover
>>> gen_amp_seed(password)
(3.99901475987551, 0.5975199126206723)
>>>
>>> # Call gen_amp_seed() with an argument of 'agent99'.
>>> a, x = gen_amp_seed('agent99')
>>> print(a, x)
(3.9914234187336097, 0.20519277633205749)
```

We can use the seed to generate an offset:

```
>>> # Show how amplitude and seed can be used with logistic growth
>>> # function to obtain a suitable offset for encryption.
>>> offset = int(96 * x)
>>> print(offset)
19
>>> x = log_growth(a, x)
>>> offset = int(96 * x)
>>> print(offset)
62
```

The user is prompted for a password and responds with `undercover`. The function `gen_amp_seed()` is then called with the password argument. Then `gen_amp_seed()` returns the amplitude and seed values. When a different password `agent99` is used, the amplitude and seed values are different. In the next few lines we see how the seed can be used to obtain an offset and how the logistic growth function can be used to generate the next offset using the seed.

Recall that you open a file using the `open()` command. The first argument is the file name and the second argument indicates whether you are opening the file for reading or writing (both these

arguments are strings). To open the input file for reading, the second argument is the character `'r'`; to open the output file for writing, the second argument is the character `'w'`. `open()` returns a file object. When a file object is used as the iterable in a `for`-loop header, the loop variable takes on the value of a line in the file for each iteration of the loop, i.e., it is a string. This string variable includes the newline character at the end of each line. We remove the newline character using the `rstrip()` method (or `strip()`). Printing to an output file is straightforward. In the following example, we open an input file called `data.txt` and an output file called `result.txt`. Then in a `for`-loop, we iterate through each line of the input file, removing the whitespace at the end of each line and printing the result to the output file.

```
file_in = open('data.txt', 'r')
file_out = open('result.txt', 'w')
for line in file_in:
    print(line.rstrip(), file=file_out)
```

Submission information. Use Canvas to submit a zipped file called **`<first_initial+lastname>_pa7.zip`** that includes the following: Three programs (**`cipher.py`**, **`pa7a.py`**, and **`pa7b.py`**) and your **`no_secret.txt`** file (it will count 5%) in a *SINGLE* zip file. The grading rubric is available in Canvas.