# INTRODUCTION TO COMPUTER SCIENCE I
# Fall 2014
# PROJECT 3A
# Creating a substitution cipher program

*Encryption* is the process of encoding a message so that only those with specialized information—the corresponding *decryption algorithm and key*—can read the encoding message. Most forms of encryption somehow scramble the symbols that represent the message such that it is difficult to recreate that scrambling in order to then translate the message back into a "normal" form and read it.

We are going to implement a classic *cryptographic cipher*—a form of encryption and decryption. Once you've written that program, you will be able to obtain a secret message and then read its contents. That message will lead you to *Project 3b* (perhaps indirectly), in which you will *cryptanalyze* a second message—that is, crack its code, without the proper key, and read it anyway. More on that after break. . .

## 1 The substitution cipher

When performing encryption on a message, we begin with the *cleartext* of that message: a straightforward, common representation of that message. For example, if the message is normally English text, then the *cleartext* is just the plain, unmodified written form of that text. We then choose a particular *cipher*—a form of encryption and decryption—along with a *key*: a unique value (often a number or a passphrase) that is one of the inputs to the cipher. We then encrypt the cleartext using our chosen key to produce a *ciphertext*—a scrambled and thus obscured version of that same message.

The recipient of the ciphertext can then apply the cipher's decryption, using the key once again to transform the scrambled for of the message back into the cleartext. Knowing the cipher that was used is not enough to decrypt that message; the key is also necessary.[1]

A *substitution cipher* is one that replaces each possible cleartext character with a corresponding ciphertext character. For example, we may replace all occurences `A` in the cleartext with `q`'s in the ciphertext, each of the `B`'s with `R`'s, and so on. The cleartext message is then encrypted by replacing each of its original, cleartext characters with the corresponding ciphertext characters. For example, if we were considering only the 26 uppercase letters of the English alphabet, then one possible correspondence could be:

| **Cleartext char** | A | B | C | D | E | F | G | . . . | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ciphertext char** | Q | D | A | E | H | Y | W | . . . | B | Z | P | I | M | S | U | V |

That is, to form a ciphertext alphabet, we take the plaintext alphabet and *permute* its characters. The permutation cannot be purely random, because that same permutation must be used for both encryption and decryption. Thus, the permutation must be determined in part by the *key* $k$ that is

[1]Well, most of the time. If the cipher isn't sufficiently *strong*, we may be able to "crack the code" and read the message without prior knowledge of the key.

chosen for a given encryption/decryption. Specifically, one must use a *pseudorandom number generator (PRNG)*, where the *seed* for that algorithm is $k$, in order to gernerate the same permutation twice.[2]

We can create a list containing each of the characters. We may then use $k$ as a seed to randomly permute those characters. Note also that once we create the randomly ordered list, we can use it to map cleartext to ciphertext characters and back again.

## 2 Pseudo-Random Number Generator background

In order to create this random permutation of characters, you will need to use the random number functions (e.g., `random.randrange()`). However, for this task, you must understand a bit more about how these functions work.

### 2.1 What is a PRNG?

The `random` functions in Python implement a *pseudo-random number generator*. Functions like `random.randrange()` have been written by someone else, but they are still normal functions, written using the same Python data types and operations that you have been using. Critically, Python (and indeed all computers) are *deterministic* machines—given the same input, they produce the same output every time. How can a deterministic machine produce a random result? It can't.

That is why we refer to these functions as being **pseudo**-random: they are, in fact, deterministic algorithms that merely produce unpredictable results, since they cannot produce truly random results. These functions "remember" the last random value, and use it to calculate the next one. The calculation is sufficiently complex that, without examining the function itself, it would be hard to guess what the next pseudo-random number would be based on the previous one. However, if you **do** examine the function, guess the next pseudo-random number based on the previous one is trivial.

### 2.2 How do you control a PRNG?

Sometimes, we would like for the sequence of (seemingly) random numbers produced by these functions to be *repeatable*—to get the same sequence of unpredictable values out of them. To achieve this end, there is a useful function that Python's pseudo-random number generator (PRNG) provides:

```
random.seed(value)
```

By calling this function, you may specify a `value` (say, some integer, like $5$). This function is used by the PRNG as though it were the most recent random value produced. Thus, by *seeding* the PRNG, you bring it to a fixed place in the sequence of numbers that it can produce. Each time you

[2]Keep in mind the number of possible permutations. For the 26 uppercase letters, there are $26! \approx 4 \times 10^{26}$ permutations; Python has 100 printable characters, there are $100! \approx 9.3 \times 10^{157}$ permutations. Given that there are estimated to be about $10^{80}$ subatomic particles in the observable universe, you may have difficulty intuitively grasping just how staggering a number of permutations that is.

seed the PRNG, you can then call `random.randrange()` and get the exact same sequence of values.

## 2.3 How do you use a PRNG?

Create and run the following code in a module (which you could name, say, `test.py` if you wanted):

```
import random

def main ():
  random.seed(13)
  x = random.randrange(0, 100)
  y = random.randrange(0, 100)
  z = random.randrange(0, 100)
  print(x)
  print(y)
  print(z)

main()
```

What happens when you run this module twice, thrice, or more times than that?[3] You will see the **same output** each time. By using `random.seed()` to set the PRNG to a fixed state (where 13 is the last value it thinks that it produced), the "random" numbers that follow are identical each time.

Thus, any time you need to make the selection of random values repeatable, you need only seed the PRNG to some specific value. In the context of scrambling an alphabet of letters for encryption/decryption, the *key* could well be the seed. Hint hint.

## 3 Your assignment

To get started in creating your own encryption program, do the following:

1. Create a new module named `encrypt.py`. When it runs, it should prompt the user for the following information, like so (with example responses shown):

   ```
   Original file?  groceries.txt
   Encrypted file? groceries.txt.enc
   Key?            42
   ```

   With these responses, the program should read the contents of the file `groceries.txt` and perform a substitution on it. That is, it should create a *dictionary* that maps cleartext characters to ciphertext characters, and then use that map to create a new file, `groceries.txt.enc`, with the message translated to the ciphertext substitutions.

[3]If *twice* is "two times", and *thrice* is "three times", then what is "four times"? *Quice?*

2. Create a companion module named `decrypt.py` that, when run, should invert the work of your encryption module, like so:

   ```
   Encrypted file? groceries.txt.enc
   Decrypted file? groceries-decrypted.txt
   Key?            42
   ```

   If the key value provided is correct, the decryption module should decrypt `groceries.txt.enc` (for example) into a new, cleartext file `groceries-decrypted.txt` which, if all goes correctly, should be identical to the original `groceries.txt`.

   This program should create the exact same dictionary that the encryption program does, but then **invert it** into a new dictionary, where the values of the original become the keys, and the keys become the values. Of course, your work from Lab 6 should provide a handy function for that inversion.

## 4 Submitting your work

### 4.1 Your supersecret key

To complete the assignment, you must encrypt something for me (more details below in Section 4.2), where the encryption key that you use is a simple kind of summation on your username. Specifically, go through the characters of your Amherst College username. For each character, add the following to a running total:

- *If the character is a letter, add its position number in the alphabet.* So, `a` is 1, `b` is 2, ..., `z` is 26. Note that all letters in a username are lowercase, so you need not consider uppercase.
- *If the character is a digit, add* ***the negative*** *of its value.* So, `1` is -1, `2` is -2, ..., `9` is -9. Yes, `0` is -0 which is, of course, 0.

The number that you get from this running sum **is your supersecret key**.

### 4.2 Submitting

Use the CS submission site, to upload the following for Project 3a:

1. Your `decrypt.py` module.
2. Your `encrypt.py` code, **but in encrypted form**, so really, your `encrypt.py.enc` file. It should be encrypted using the *supersecret key* described above in Section 4.1.

**This assignment is due by Friday, Dec-05, at 11:59 pm.**