Introduction to Cryptography

Today, we will examine the basics of *cryptography*, or encoding and decoding secret messages. We will look at some simple cipher systems that use *modular arithmetic*. Next week, we will extend these ideas to the other encryption algorithms.

To create an encoded message using our example cipher systems, we first need to convert letters into numbers. We will work with the following scheme.

a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9
k	l	m	n	o	p	q	r	s	t
10	11	12	13	14	15	16	17	18	19
u	v	w	x	y	z				
20	21	22	23	24	25				

Table 1

So, to translate the word "secret" into numbers, we would have $18, 4, 2, 17, 4, 19$ as our numeric values. Then, we will apply some sort of encoding function to these values, and finally, we will translate back to letters. To do this easily in python, we will use commands that we have used before.

- `ord(x)` – returns the ascii code number for the character stored in x
- `map(function,list)` – applies the given function to every element in the given list
- `chr(n)` – returns the character whose ascii code number is n

Translating letters to numbers and back again

Recall how to define functions to easily translate words (strings of characters) to numbers. The `ord` function will give us the ascii code for any given character. Here is an example of using `ord`.

```
>>> ord("a")
97
>>> letter = "h"
>>> ord(letter)
104
>>> initial = "d"
>>> ord(initial)
100
```

A function that nicely handles the shift by 97 is given below. Type this in and save it.

```
def StringToInteger(x):
    return(ord(x)-97)
```

** **Try 1:** Write a function using `chr` to convert from numbers between 0 and 25 back to letters. Name your function `IntegerToString`.

Recall how to use the `map` command. This command is quite powerful and handy. To see how it works, consider the following example of squaring numbers in a list.

```
>>>data = [1,2,3,5]
>>>def f(x):
       return(x*x)
>>>map(f,data)
[1,4,9,25]
```

We can similarly apply our function `StringToInteger` to any given string. For example,

```
>>>plain = "attackatdawn"
>>>map(StringToInteger,plain)
[0, 19, 19, 0, 2, 10, 0, 19, 3, 0, 22, 13]
```

The Caesar Shift

Legend states that Julius Caesar used a simple encoding scheme in which messages were encrypted by shifting letters down by three in the alphabet. After translating our message (without spaces) into numbers, this is easy for us to do by just adding 3 to all of the numbers. BUT, what if the numbers go beyond 25? In this case, we want to wrap around and begin the letters again at "a." We can do this using *mod* or the remainder when dividing by 26. (This is called *modular arithmetic*.)

For example, "k" is translated to 10. We add 3 to get 13, and translate back to "n." So, "k" is encoded as "n." As another example, consider "y." This letter translates to 24, and we add 3 to get 27. BUT, this is outside of our table above. So, we take the remainder when dividing by 26. That is, $27 \bmod 26 = 1$. Translating back to letters gives "b." So, "y" is encoded as "b."

To decode the Caesar Shift, we invert the process and subtract 3.

** **Try 2:** Write python code to encode and decode the string "attackatdawn" using the Caesar shift. It will be helpful to define functions `encode` and `decode` first and then use the `map` command to apply these functions to your string.

An Affine Cipher

Encoding

The art to creating a good code is to find ways to make the encoding method complicated to decode but still possible to decode. The first step on this road is to switch from just shifting by 3 to a more complicated function. In this section, we will convert our message to numbers, multiply these numbers by something, add a shift, and then convert back to letters.

For this section, we will encode letters by applying the following function to the number equivalent of a letter.

$$c = 15 * p + 3 \pmod{26} \tag{1}$$

Here p represents the message, or *plaintext*, and c represents the encoded message, or *ciphertext*. So, to encode "t", we translate to the number 19. Then,

$$c = 15 * 19 + 3 = 288 = 2 \pmod{26}$$

since dividing 288 by 26 leaves a remainder of 2.
****Try 3:** Write the function above into your python program for encoding messages. Remember that `mod` can be performed in python using the % command. (I.e., 288 % 26)

Decoding

To decode a message using the system above, we just need to invert the function in equation (1). To do so, we must subtract 3 on both sides and divide by 15. BUT, we are working only with integers, so we CANNOT divide by 15 as usual. So, the question becomes: is there an integer b so that $15 * b = 1 \pmod{26}$?

The answer to this question is "Yes," since 15 and 26 are relatively prime (that is, they share no common prime factors). The result is that for 15, $b = 7$. Then, check

$$15 * 7 = 105 = 1 \pmod{26}$$

since dividing 105 by 26 gives a remainder of 1. So, 7 is the multiplicative inverse of 15 in this system.

So, our decoding function is

$$p = 7 * (c - 3) \pmod{26} \tag{2}$$

****Try 4:** Write the function above into your python program for decoding messages. Remember that `mod` can be performed in python using the % command.