Affine ciphers

We assign to each letter $A-Z$ a number $0-25$. After this coding we work in $\mathbb{Z}/26\mathbb{Z}$. In this context an affine cipher is a map $f : \mathbb{Z}/26\mathbb{Z} \rightarrow \mathbb{Z}/26\mathbb{Z}$ given by $f(x) = ax + b$ with $a$ and 26 relatively prime. The numbers $a$ and $b$ are our secret keys. In the following program correspond to the variables $\textbf{key1}$ and $\textbf{key2}$.

```python
alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
message = 'MYSECRET'
key1 = 3
key2 = 0

encrypted = ''
for c in message:
    loc = alph.find(c)
    encrypted += alph[Mod(key1*loc+key2,26)]
print message
print encrypted
```

If you are not a skilled Pythonist or Sage you will appreciate the following comments:

```python
# Start with and empty encrypted message
# we'll add character with a loop
encrypted = ''
# This is the loop c is each character in the message
for c in message:
    # Search the position of c in alph.
    # This is the code corresponding to c
    loc = alph.find(c)
    # Computes (key1*loc+key2 modulo 26 and
    # append the corresponding character to encrypted
    encrypted += alph[Mod(key1*loc+key2,26)]
# Print the original and the encrypted messages
print message
print encrypted
```

Reusing is part of the Python philosophy and we can recycle our program more easily using functions. In the computer science jargon this is a kind of encapsulation. We call the function using the message and the keys and we do not care about the internal definition of the alphabet or the access to it.

```python
def encrypt(message, key1,key2):
    alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    encrypted = ''
    for c in message:
        loc = alph.find(c)
        encrypted += alph[Mod(key1*loc+key2,26)]
    print message,'->',encrypted
```
Now we encrypt with

```python
# ENCRYPT MESSAGES
encrypt('MYSECRET', 3, 0)
```

that gives KUCMGZMF.

The inverse function of \( f(x) = ax + b \) is \( g(x) = a^{-1}(x - b) = a^{-1}x - a^{-1}b \). We have to use it to decrypt messages

```python
encrypt('KUCMGZMF', 3. inverse_mod(26), 0)
```
gives MYSECRET.

In general

```python
encrypt(..., key1.inverse_mod(26), -key1.inverse_mod(26)*key2)
```

inverts

```python
encrypt(..., key1, key2)
```

Note that we really need \( a = key1 \) and 26 to be relatively prime because we have to compute the multiplicative inverse of \( a \) in \( \mathbb{Z}/26\mathbb{Z} \) to apply the inverse map.

We can add new characters to our alphabet (the usual ASCII code employs 256 characters, one byte) modifying the modulo.

Here we have an example:

```python
1 def encrypt27(message, key1, key2):
2     alph = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ?'
3     encrypted = '
4     for c in message:
5         loc = alph.find(c)
6         encrypted += alph[Mod(key1*loc+key2,27)]
7     print message, '->', encrypted
8
9 # ENCRYPT MESSAGES
10 encrypt27('HOWAREYOU?', 2, 0)
11 encrypt27('OBRAHIVBNZ', 2. inverse_mod(27), 0)
```

If we encrypt with

```python
encrypt27('HOWAREYOU?', 2, 0)
```

we decrypt with

```python
encrypt27('OBRAHIVBNZ', 2. inverse_mod(27), 0)
```