# HigherOrderFunctions2.py
# All from Chapter 3 of "CS for All" at Harvey Mudd College.

import sys
import random
sys.setrecursionlimit(20000) # allow functions to recurse 20,000 times.

# Map

# Very often we want to apply a function to a list, for example given

def double(n):
    return n * 2

def doubleAll(L):
    if L == []:  
        return []
    else:
        return [double(L[0])] + doubleAll(L[1:])

# Eg,

## >>> doubleAll(range(10))
## [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
## >>> doubleAll(['cat', 'bird', 'dog'])
## ['catcat', 'birdbird', 'dogdog']

# Python has a built in function
#
# map( f, List) that maps f across (applies f to every element of) List, 
# to produce a new list. Egs,

## >>> map(double, range(10))
## [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
## >>> map(lambda x: x+1, range(10))
## [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
## >>> map(lambda x: x*2, range(10))
## [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

## >>>

# Can you write a recursive definition of map()? 

# List Comprehensions

# Python has this syntax for specifying what it calls, 
# "list comprehensions". These serve a purpose similar to map(). 
# In general, the syntax is 
# 
# [ f(x) for x in L ] where f is a function and L is a list.
#
# Egs,

## >>> [double(x) for x in [1,2,3]]
## [2, 4, 6]
## >>> [x+1 for x in [1,2,3]]
## [2, 3, 4]
## >>> [len(w) for w in ["one", "two", "three", "four", "five", "six"]]
## [3, 3, 5, 4, 4, 3]

# We can also use list comprehensions for filtering. Eg.,
```python
## [w for w in ['one', 'two', 'three', 'four', 'five', 'six'] if len(w)==4]
## ['four', 'five']
##
## and we can do both mapping and filtering, eg.,
##
## >>> [double(x) for x in range(1000) if x % 31 == 0]
## [0, 62, 124, 186, 248, 310, 372, 434, 496, 558, 620, 682, 744, 806, 868,
## 930, 992, 1054, 1116, 1178, 1240, 1302, 1364, 1426, 1488, 1550, 1612, 1674,
## 1736, 1798, 1860, 1922, 1984]
##
## >>> [x/31 for x in range(1000) if x % 31 == 0]
## [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
## 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
##
## # Reduce

# Say we want to sum up or multiply up a list of numbers; we can use a
# higher-order function called reduce() for this.
#
# Eg, say we have two binary functions,

def add(x,y):
    return x + y

def multiply(x,y):
    return x * y

# We can then use reduce() to compute the sum (or product) of a list
# of numbers. Eg.,
##
## >>> reduce(add, [1,2,3,4,5,6])
## 21
## >>> reduce(multiply, [1,2,3,4,5,6])
## 720
##

# Functions as Results

# Consider the function, scale():

def scale(factor):
    return lambda(x): factor * x

# What if the result of scale(2) ?

g = scale(2)

##
## >>> g(3)
## 6
## >>> g(8)
## 16
##
## So, scale() is a function that, when applied to a number, returns
# another function! So, we can name functions, pass them as arguments,
# and return them as the results of other functions. Functions really DO
# have the same civil rights as other types of values.
#
# Btw, we can ask what type a value has:
##
## >>> type(5)
## <type 'int'>
## >>> int("5")
def scale(factor):
    def multiply(x):
        return n * x
    return multiply

# And btw, we can nest function definitions and then return them. Eg.,
def scale(factor):
    def multiply(x):
        return n * x
    return multiply

# Back to RSA Cryptography.
#
# Recall:
#
# We want two functions: encrypt() and decrypt(). If x is our message (eg. a number) we will encrypt it using the RSA function
#
# f(x) = x ** e mod n
#
# where e and n are carefully chosen numbers. If we choose e and n appropriately, a receiver can decrypt the encrypted message even. Others cannot, even if they know e and n.

# How do we choose e and n?
#
# (1) We choose two different large prime numbers p and q at random.
# (2) n = p*q.
# (3) Compute e in two steps.
#     (a) Let m = (p - 1)*(q - 1)
#     (b) e = a random prime < m and not a divisor of m.

# Eg from the text, let's work with small primes for our example
p = 3
q = 5
n = 3 * 5 = 15
m = (3 - 1)*(5 - 1) = 8
Now choose
# e = 3, since it's < n (ie < 15).
# Now we can encrypt any number < n; let's encrypt 13.
codedMessage = f(x) = f(13) = 13 ** 3 mod 15 = 7

# Now, how does our receiver decrypt this 7 to get the original 13?
# Well, when we computed e, we should have also computed a decryption exponent d, which has two properties:
# (1) 1 < d < m - 1 (ie, d lies between 1 and m - 1).
# (2) (e * d) mod m = 1
(We call $d$ the "multiplicative inverse of $e$; and there is just one $d$ with these properties.) Then we can retrieve our original message with,

g(y) = y ** d \mod n \quad (y \text{ is our coded message})

In our example, $e = 3$ and $m = 8$, so $d = 3$ (only coincidentally same as $e$).
So, $g(7) = 7 ** 3 \mod 15 = 13$ (and 13 was indeed our original message!)

Notice, that

(1) $e$ and $n$ are "public keys", anyone who knows them can encode a message
that only one with $d$ can read; but no one else can (even if they have
the public keys and a coded message).
(2) $d$ is a "private key. One must have this $d$ to decode a coded message.

Hence the name public/private key encryption.

Notice that, if we know $e$ and $n$, we "can" compute the factors $p$ and $q$ and
so eventually find $d$. But this factorization of $n$ to find the large primes
$p$ and $q$ (as opposed to our small example $p$ and $q$) is computationally hard,
meaning it would take a long time even with powerful computers. (One wonders
how long it takes the NSA; they are trying!)

Now we are sophisticated enough to write a function that computes
and returns both an RSA encoder and its corresponding decoder.

```python
def makeEncoderDecoder():
    """ Returns two functions: an RSA encryption function and
    its corresponding decryption function.
    """
    # Choose 2 primes:
p, q = random.sample(primeSieve(range(2, 10)), 2)

    n = p*q
    m = (p-1)*(q-1)
    print "Maximum number that can be encrypted is ", n-1

    # Choose a random prime for e:
    e = random.choice(primeSieve(range(2, m)))
    if m % e == 0: # If $e$ divides $m$, it won't work!
        print ("Please try again")
        return
    else:
        d = inverse(e, m) # compute d
    encoder = lambda x: (x**e) % n # encryption function
    decoder = lambda y: (y**d) % n # decryption function
    return [encoder, decoder]

def inverse(e, m):
    """ The inverse of $e$ mod $m$.
    """
    # There's only one! Pick it out.
    return filter(lambda d: e * d % m == 1, range(1,m))[0]

def sift(divisor, List):
    """ Returns the list of number List, but where all numbers in it
    that are evenly divided by divisor are sifted out.
    """
    return filter(lambda x: x % divisor != 0, List)
def primeSieve(numbers):
    """ Returns a list of all primes in numbers, using a prime sieve. """
    if numbers == []:
        return []
    else:
        prime = numbers[0]  # the first number is prime.
        return [prime] + primeSieve(sift(prime, numbers[1:]))

# >>> encrypt, decrypt = makeEncoderDecoder()
# Maximum number that can be encrypted is 20
# Please try again
#
# Traceback (most recent call last):
#  File "<pyshell#38>" , line 1, in <module>
#    encrypt, decrypt = makeEncoderDecoder()
# TypeError: 'NoneType' object is not iterable
# >>> encrypt, decrypt = makeEncoderDecoder()
# Maximum number that can be encrypted is 34
# >>> coded = encrypt(33)
# >>> coded
# 3L
# >>> decrypt(coded)
# 33L
# >>>

# Of course, we'd want to use much larger primes and write the program
# so the computation of d is repeatedly computed until it's valid.