Mutual Recursion in Python

When a recursive procedure is divided among two functions that call each other, the functions are said to be *mutually recursive*. As an example, consider the following definition of even and odd for non-negative integers:

- a number is even if it is one more than an odd number
- a number is odd if it is one more than an even number
- 0 is even

Using this definition, we can implement mutually recursive functions to determine whether a number is even or odd:

```
1  def is_even(n):
2      if n == 0:
3          return True
4      else:
5          return is_odd(n-1)
6
7  def is_odd(n):
8      if n == 0:
9          return False
10     else:
11         return is_even(n-1)
12
```

```
13  result = is_even(4)
```

Edit code

< Back Step 1 of 18 Forward >

Mutually recursive functions can be turned into a single recursive function by breaking the abstraction boundary between the two functions. In this example, the body of `is_odd` can be incorporated into that of `is_even`, making sure to replace `n` with `n-1` in the body of `is_odd` to reflect the argument passed into it:

```
>>> def is_even(n):
        if n == 0:
            return True
        else:
            if (n-1) == 0:
                return False
            else:
                return is_even((n-1)-1)
```

As such, mutual recursion is no more mysterious or powerful than simple recursion, and it provides a mechanism for maintaining abstraction within a complicated recursive procedure.

As another example of mutual recursion, consider a two-player game in which there are `n` initial pebbles on a table. The players take turns, removing either one or two pebbles from the table, and the player who removes the final pebble wins. Suppose that Alice and Bob play this game, each using a simple strategy:

- Alice always removes a single pebble
- Bob removes two pebbles if an even number of pebbles is on the table, and one otherwise

Given `n` initial pebbles and Alice starting, who wins the game?

A natural decomposition of this problem is to encapsulate each strategy in its own function. This allows us to modify one strategy without affecting the other, maintaining

the abstraction barrier between the two. In order to incorporate the turn-by-turn nature of the game, these two functions should call each other at the end of each turn.

```
>>> def play_alice(n):
        if n == 0:
            print("Bob wins!")
        else:
            play_bob(n-1)
>>> def play_bob(n):
        if n == 0:
            print("Alice wins!")
        elif is_even(n):
            play_alice(n-2)
        else:
            play_alice(n-1)
```

Two observations can be made from the above functions. First, a recursive procedure need not return any value. In this case, a different string is printed to the screen depending on who wins. Second, multiple recursive calls may appear in the body of a function. Here, `play_bob` may call `play_alice` from two locations in the body. However, in this example, each call to`play_bob` calls `play_alice` at most once. In the next section, we consider what happens when a single function call makes multiple direct recursive calls.

Source: http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#mutual-recursion