Recursion

APRIL 11TH, 2012
What is recursion?

- In most programming languages including Python, a function can call itself.
- Functions that call themselves are called recursive functions.
- Why might this be useful??
  - Many computational problems can be solved by:
    - Solving smaller versions on the problem and
    - Combining solutions of the smaller problems
- Recursion is a natural way of implementing this problem solving approach.
The First Example

- Computing the factorial of \( n \)
- Recall: \( n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \)
- Hence, \( n! = n \times (n-1)! \)
- So to compute \( 4! \), we need to compute \( 3! \) first and then multiply by 4 the result of our computation.
- To compute \( 3! \), we need to compute \( 2! \) first and multiply by 3 the result of our computation.
- And so on…
But wait!

- What about computing $1!$? We would compute $0!$ and multiply the answer by $1$. Then, what about computing $0!$?
- You’ll notice that if we keep this up, we will go forever.
- So we need a base case. This refers to a problem that is so small that we “know” the answer to the problem right away, i.e., without any computation (or with very little computation).
A complete solution

- For $n > 0$, $n! = n \times (n-1)!$ (Recursive case)
- For $n = 0$, answer is 1 (Base case)

This can be translated into a recursive function easily:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```
Advice

- You should practice tracing the recursive functions you write, especially in the beginning.
- You should make sure that your base cases cover all of the “bottom” cases; otherwise there is a big danger of infinite loops.
The *Fibonacci sequence* is 1, 1, 2, 3, 5, 8, 13, 21, ...

More precisely, \( F(1) = 1, \ F(2) = 1, \) and for any \( n > 2, \ F(n) = F(n-1) + F(n-2). \)

Notice that the definition of this sequence itself is recursive and so computing the \( n \)th Fibonacci number can be easily solved by a recursive function.