Recursion

APRIL 18TH, 2014
What is recursion?

- In most programming languages, a function can call itself.
- A function that calls itself is called a recursive function.
- Why might this be useful??
- Many computational problems can be solved by:
  - solving smaller versions of the problem
  - combining the solutions (of the smaller problems) to obtain the solution of the bigger problem.
- Recursion is a natural way of implementing this problem-solving approach.
Recursion Outside Computer Science

- Search for “recursion” on Google and you will be asked the question “Did you mean recursion?”
- Jokes aside, recursion plays an important role in natural languages, fractals, etc.
Write a function to compute the factorial of a given positive integer \( n \).

A simple solution:

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    mult = 1
    for i in range(2, n+1):
        mult = mult*i
    return mult
```
The Recursive Approach

- Note that
  \[ n! = (n-1)! \times n \]

- Thus to compute the factorial of \( n \), we just have to compute the factorial of \( n-1 \) and then multiply the answer by \( n \).

- Example: \( 4! = 3! \times 3 \)
Recursive Function for Factorial (Version 1)

def factorial(n):
    return factorial(n-1) * n

- To compute 4!, we first compute 3!
- To compute 3!, we first compute 2!
- To compute 2!, we first compute 1!
- To compute 1!, we first compute 0!
- To compute 0!, we first compute -1!
- ...

- You’ll notice that this goes on forever – we have infinite loop!
Base Case

- To complete a recursive function, it is not enough to specify how the problem is solved in terms of smaller problems.
- It is also necessary to enumerate the smallest problem(s) and specify how to solve these.
- This is called the **base case** of the recursion.
- Without a base case, we will have *infinite descent*!
- The base case for the factorial function could be $n = 0$. In this case, we know the answer is 1.
Recursive Function for Factorial (Version 2)

def factorial(n):
    # Base Case
    if n == 0:
        return 1
    # Recursive Case
    return factorial(n-1) * n

Notes:
- Look, no loops (while- or for-)! Recursive function calls can often replace looping behavior.
- Recursive functions are typically organized into one or more *recursive cases* and one or more *base cases*. 
Our Second Example

- 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.
The fibonacci function

def fibonacci(n):
    # Base cases
    if n == 1 or n == 2:
        return 1
    else:
        # Recursive case
        return fibonacci(n-1) + fibonacci(n-2)