Anatomy of Recursive Functions in Python

A common pattern can be found in the body of many recursive functions. The body begins with a *base case*, a conditional statement that defines the behavior of the function for the inputs that are simplest to process. In the case of `pig_latin`, the base case occurs for any argument that starts with a vowel. In this case, there is no work left to be done but return the argument with "ay" added to the end. Some recursive functions will have multiple base cases.

The base cases are then followed by one or more *recursive calls*. Recursive calls require a certain character: they must simplify the original problem. In the case of `pig_latin`, the more initial consonants in `w`, the more work there is left to do. In the recursive call,`pig_latin(w[1:] + w[0])`, we call `pig_latin` on a word that has one fewer initial consonant: a simpler problem. Each successive call to `pig_latin` will be simpler still until the base case is reached: a word with no initial consonants (assuming that the input contains a vowel somewhere).

Recursive functions express computation by simplifying problems incrementally. They often operate on problems in a different way than the iterative approaches that we have used in the past. Consider a function `fact` to compute `n` factorial, where for example`fact(4)` computes $4!=4 \cdot 3 \cdot 2 \cdot 1=24$.

A natural implementation using a `while` statement accumulates the total by multiplying together each positive integer up to `n`.

```
>>> def fact_iter(n):
        total, k = 1, 1
        while k <= n:
            total, k = total * k, k + 1
        return total
>>> fact_iter(4)
24
```

On the other hand, a recursive implementation of factorial can express `fact(n)` in terms of `fact(n-1)`, a simpler problem. The base case of the recursion is the simplest form of the problem: `fact(1)` is `1`.

```
1  def fact(n):
2      if n == 1:
3          return 1
4      return n * fact(n-1)
5
6  fact(4)
```

Edit code

< Back Step 1 of 14 Forward >

These two factorial functions differ conceptually. The iterative function constructs the result from the base case of 1 to the final total by successively multiplying in each term. The recursive function, on the other hand, constructs the result directly from the final term,`n`, and the result of the simpler problem, `fact(n-1)`.

As the recursion "unwinds" through successive applications of the *fact* function to simpler and simpler problem instances, the result is eventually built starting from the base case. The recursion ends by passing the argument `1` to `fact`; the result of each call depends on the next until the base case is reached.

The correctness of this recursive function is easy to verify from the standard definition of the mathematical function for factorial:

$$(n-1)!\ n!\ n! = (n-1)\cdot(n-2)\cdot\ldots\cdot 1 = n\cdot(n-1)\cdot(n-2)\cdot\ldots\cdot 1 = n\cdot(n-1)!$$

While we can unwind the recursion using our model of computation, it is often clearer to think about recursive calls as functional abstractions. That is, we should not care about how `fact(n-1)` is implemented in the body of `fact`; we should simply trust that it

computes the factorial of `n-1`. Treating a recursive call as a functional abstraction has been called a *recursive leap of faith*. We define a function in terms of itself, but simply trust that the simpler cases will work correctly when verifying the correctness of the function. In this example, we trust that `fact(n-1)` will correctly compute `(n-1)!`; we must only check that `n!` is computed correctly if this assumption holds. In this way, verifying the correctness of a recursive function is a form of proof by induction.

The functions *fact_iter* and *fact* also differ because the former must introduce two additional names, `total` and `k`, that are not required in the recursive implementation. In general, iterative functions must maintain some local state that changes throughout the course of computation. At any point in the iteration, that state characterizes the result of completed work and the amount of work remaining. For example, when `k` is `3` and total is `2`, there are still two terms remaining to be processed, `3` and `4`. On the other hand,*fact* is characterized by its single argument `n`. The state of the computation is entirely contained within the structure of the environment, which has return values that take the role of `total`, and binds `n` to different values in different frames rather than explicitly tracking `k`.

Recursive functions can rely more heavily on the interpreter itself, by storing the state of the computation as part of the environment, rather than explicitly using names in the local frame. For this reason, recursive functions are often easier to define, because we do not need to try to determine the local state that must be maintained across iterations. On the other hand, learning to recognize the computational processes evolved by recursive functions requires practice.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/functions.html#the-anatomy-of-recursive-functions