Recursion in Python

One of the fundamental ideas of computer science is to divide a complicated problem into one or more simpler pieces, solving them, and using their solution to compute a solution to the original problem. When the simpler sub-problems are instances of the original problem, this technique is called *recursion*. The functional abstraction enables us to implement this method in functions that call themselves on simpler input somewhere in the function body.

Recursive Functions

A function is called *recursive* if the body of that function calls the function itself, either directly or indirectly. That is, the process of executing the body of a recursive function may in turn require applying that function again. Recursive functions do not use any special syntax in Python, but they do require some care to define correctly.

As an introduction to recursive functions, we begin with the task of converting an English word into its Pig Latin equivalent. Pig Latin is a secret language: one that applies a simple, deterministic transformation to each word that veils the meaning of the word. Thomas Jefferson was supposedly an early adopter. The Pig Latin equivalent of an English word moves the initial consonant cluster (which may be empty) from the beginning of the word to the end and follows it by the "-ay" vowel. Hence, the word "pun" becomes "unpay", "stout" becomes "outstay", and "all" becomes "allay".

Our goal in this section is to define a set of functions to produce the Pig Latin version of a word. We use strings to represent words, but we need a way of separately accessing the first letter of a word and the remaining letters. We can use the expression `s[0]` to obtain the first letter of a string `s` and `s[1:]` to get the remaining letters:

```
>>> 'pig'[0]
'p'
>>> 'pig'[1:]
'ig'
```

The bracket notation is very general and will be discussed at length in Chapter 2, but this is all we need for now. We can now define functions to produce the Pig Latin version of a word:

```
>>> def pig_latin(w):
        """Return the Pig Latin equivalent of a lowercase
English word w."""
        if starts_with_a_vowel(w):
            return w + 'ay'
        return pig_latin(w[1:] + w[0])
>>> def starts_with_a_vowel(w):
        """Return whether w begins with a vowel."""
        c = w[0]
        return c == 'a' or c == 'e' or c == 'i' or c ==
'o' or c == 'u'
```

The idea behind this definition is that the Pig Latin variant of a string that starts with a consonant is the same as the Pig Latin variant of another string: that which is created by moving the first letter to the end. Hence, the Pig Latin word for "sending" is the same as for "endings" (*endingsay*), and the Pig Latin word for "smother" is the same as the Pig Latin word for "mothers" (*othersmay*). Moreover, moving one consonant from the beginning of the word to the end results in a simpler problem with fewer initial consonants. In the case of "sending", moving the "s" to the end gives a word that starts with a vowel, and so our work is done.

This definition of `pig_latin` is both complete and correct, even though the `pig_latin` function is called within its own body.

```
>>> pig_latin('pun')
'unpay'
```

The idea of being able to define a function in terms of itself may be disturbing; it may seem unclear how such a "circular" definition could make sense at all, much less specify a well-defined process to be carried out by a computer. We can, however, understand precisely how this recursive function applies successfully using our environment model of computation. The environment diagram and expression tree that depict the evaluation of `pig_latin('pun')` appear below.

```
def pig_latin(w):
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])

def starts_with_a_vowel(w):
    c = w[0]
    return c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u'

pig_latin('pun')
```

Edit code

< Back Step 1 of 15 Forward >

The steps of the Python evaluation procedures that produce this result are:

1. The `def` statement for `pig_latin` is executed, which
    A. Creates a new `pig_latin` function with the stated body, and
    B. Binds the name `pig_latin` to that function in the current (global) frame.
2. The `def` statement for `starts_with_a_vowel` is executed similarly.
3. The call expression `pig_latin('pun')` is evaluated by
    A. Evaluating the operator and operand sub-expressions by
        I. Looking up the name `pig_latin` that is bound to the `pig_latin` function.

        II. Evaluating the operand string literal to the string `'pun'`.

    B. Applying the function `pig_latin` to the argument `'pun'` by

        I. Adding a local frame,
        II. Binding the formal parameter `w` to the argument `'pun'` in that frame, and
        III. Executing the body of `pig_latin` in the environment that starts with that frame:

            a. The initial conditional statement has no effect, because the header expression evaluates to `False`.
            b. The final return expression `pig_latin(w[1:] + w[0])` is evaluated by

                1. Looking up the name `pig_latin` that is bound to the `pig_latin` function,
                2. Evaluating the operand expression to the string `'unp'`,
                3. Applying `pig_latin` to the argument `'unp'`, which returns the desired result from the suite of the conditional statement in the body of `pig_latin`.

As this example illustrates, a recursive function applies correctly, despite its circular character. The `pig_latin` function is applied twice, but with a different argument each time. Although the second call comes from the body of `pig_latin` itself, looking up that function by name succeeds because the name `pig_latin` is bound in the environment before its body is executed.

This example also illustrates how Python's recursive evaluation procedure can interact with a recursive function to evolve a complex computational process with many nested steps, even though the function definition may itself contain very few lines of code. Some examples are quite long indeed: the word *scythe* results in the longest terminating `pig_latin` process among words that appear in Shakespeare's works, ignoring punctuation.

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