Introduction to Programming in Python

Recursion

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: June 4, 2021 at 11:05

Factorial Function

Consider the factorial function:

$$k! = 1 * 2 * \ldots * (k-1) * k.$$

This is often defined mathematically by the following *recurrence relation*:

$$1! = 1$$
$$n! = n * (n-1)!, \text{ for } n > 1$$

It is typically quite easy to implement a function in Python directly from the recurrence relation.

Factorial Function

$$1! = 1$$
$$n! = n * (n-1)!, \text{ for } n > 1$$

For example, to compute 5! we do the following:

$$\begin{aligned}
5! &= 5 * 4! && (1)\\
&= 5 * (4 * 3!) && (2)\\
&= 5 * (4 * (3 * 2!)) && (3)\\
&= 5 * (4 * (3 * (2 * 1!))) && (4)\\
&= 5 * (4 * (3 * (2 * 1))) && (5)\\
&= 5 * (4 * (3 * 2)) && (6)\\
&= 5 * (4 * 6) && (7)\\
&= 5 * 24 && (8)\\
&= 120 && (9)
\end{aligned}$$

Factorial Function

Mathematical definition:

$$1! = 1$$
$$n! = n * (n-1)!, \text{ for } n > 1$$

Here's a straightforward implementation in Python.

```
def fact (n):
    """ Factorial function. """
    if n == 1:                      # base case
        return 1
    else:
        return n * fact(n-1)        # recursive case
```

This function is **recursive** because it calls itself.

Can you see anything wrong with this? How might you fix it?

How to Think about Recursion

Whenever you're trying to think about a recursive problem, take the following three steps:

1. Think of the simplest instances of the problem, ones that can be solved directly.
2. For more complex instances, assume you can solve an instance that is *one step simpler*.
3. Think how you'd get from a solution of that simpler problem to a solution to your more complex problem.

Factorial Again

$$1! = 1$$
$$n! = n * (n-1)!, \text{ for } n > 1$$

1. The simplest case (base case) is $n = 1$.
2. The more complex case is $n!$. Assume you know how to solve a case that is one step simpler, $(n-1)!$.
3. If you could do that, you'd solve $n!$ by multiplying $n * (n-1)!$.

Another Example

How would you define a recursive function `listSum(lst)` to add together all the elements of a list of numbers.

1. In the simplest (base) case, `lst` is empty. Then `listSum(lst)` is 0.
2. Now suppose `lst` isn't empty, i.e., it's `[x, y,...,z]`, and assume we knew how to find `listSum([y,...,z])`.
3. Then we solve our problem by adding x to `listSum([y,...,z])`.

```
def listSum( lst ):
    if lst == []:
        return 0
    else:
        return lst[0] + listSum( lst[1:] )
```

Running Our Code

```
>>> from recursionExamples import *
>>> listSum([])
0
>>> listSum([1, 3, 5, 8, -10])
7
```

Three Laws of Recursion

1. A recursive function must have one or more *base cases*—calculated directly without a recursive call.
2. It must have one or more *recursive calls*; without that it's not recursive.
3. Each recursive call must move the computation closer to a base case (usually by making the argument "smaller" in the recursive call).

Some Faulty Examples

```
def factBad( n ):
    return n * factBad( n - 1 )

def isEven( n ):
    if n == 0:
        return True
    else:
        return isEven(n - 2)
```

What's wrong and how would you fix these?

Recursive Thinking: Some Examples

Example: how many items are in a list L?

Base case: If L is empty, length is 0.

Recursive case: If I knew the length of L[1:], then I could compute the length of L. How?

```
def countItemsInList( L ):
    """ Recursively count the number of items in list. """
    if L == []:                  # not L: also works.  Why?
        return 0
    else:
        return 1 + countItemsInList( L[1:] )
```

```
>>> l1 = [ 1, 2, 3, "red", 2.9 ]
>>> countItemsInList( l1 )
5
```

Does this work for any list?

Recursive Thinking: Some Examples

Example: how can you count the occurrences of key in list L?

Base case: If L is empty, the count is 0.

Recursive case: If L starts with key, then it's 1 plus the count in the rest of the list; otherwise, it's just the count in the rest of the list.

```
def countOccurrencesInList( key, L ):
    """ Recursively count the occurrences of key in L. """
    if L == []:
        return 0
    elif key == L[0]:
        return 1 + countOccurrencesInList( key, L[1:] )
    else:
        return countOccurrencesInList( key, L[1:] )
```

```
>>> lst = [ 5, 6, 14, -3, 0, -70, 6 ]
>>> countOccurrencesInList( 3, lst )
0
>>> countOccurrencesInList( 6, lst )
2
```

Recursive Thinking: Some Examples

Example: how can you reverse a list L?

Base case: If L is empty, the reverse is [].

Recursive case: If L is not empty, remove the first element and append it to end of the reverse of the rest.

```
def reverseList( L ):
    """ Recursively reverse a list. """
    if L == []:
        return []
    else:
        return reverseList( L[1:] ) + [ L[0] ]
```

```
>>> lst = [ 1, 5, "red", 2.3, 17 ]
>>> print( reverseList( lst ))
[17, 2.3, 'red', 5, 1]
```

Recursive Thinking: Some Examples

An algorithm that dates from Euclid finds the greatest common divisor of two positive integers:

$$gcd(a, b) = a, \text{ if } a = b$$
$$gcd(a, b) = gcd(a - b, b), \text{ if } a > b$$
$$gcd(a, b) = gcd(a, b - a), \text{ if } b > a$$

```
def gcd( a, b ):
    """ Euclid's algorithm for GCD. """
    print( "Computing gcd(", a, ",", b, ")" )
    if a < b:
        return gcd( a, b-a )
    elif b < a:
        return gcd( a-b, b )
    else:
        print( "Found gcd:", a )
        return a

print("gcd( 68, 119 ) =", gcd( 68, 119 ))
```

What is assumed about a and b? What is the base case? The recursive cases?

Running GCD

```
> python gcd.py
Computing gcd( 68 , 119 )
Computing gcd( 68 , 51 )
Computing gcd( 17 , 51 )
Computing gcd( 17 , 34 )
Computing gcd( 17 , 17 )
gcd( 68, 119 ) = 17
```

Some Exercises for You to Try

1. Write a recursive function to append two lists.
2. Write a recursive version of linear search in a list.
3. Write a recursive function to sum the digits in a decimal number.
4. Write a recursive function to check whether a string is a palindrome.

It's probably occurred to you that many of these problems were already solved with built in Python methods or could be solved with loops.

That's true, but our goal is to teach you to think recursively!