Recursion

Vivek K. S., Deepak G.

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2017

Recursion

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller sub-problems until we get to a state where we cannot break it down any further and it becomes small enough that it can be solved trivially.

- Recursion usually involves calling the same function repeatedly.
- Recursion allows us to come up with elegant solution to problems that would otherwise require complex programming.

Example of Recursion

Let's consider a simple example of calculating the sum of a list of numbers. We would usually solve this iteratively using loops as follows:

```
def list_sum(num_list):
    sum_list = 0
    for i in num_list:
        sum_list = sum_list + i
    return sum_list
```

We simply loop over the list and on each iteration, we keep adding each element of the list to the sum variable.

Breaking Down the Problem

- If we were to break down this problem, we could see that the addition operation looks something like this:
  $((((2 + 3) + 4) + 5) + 6)$
- The sum operation adds two elements in one iteration.
- When that little addition operation is completed, the result is added to the following item in the list and it goes on.
- The same can also be seen as :
  $(2 + (3 + (4 + (5 + 6))))$

Implementation Using Recursion

This is our clue to a simple, more elegant way of computing the sum it if we consider that the sum is simply the addition of the first element in the list with the sum of the rest of the list elements in each iteration.

- Two scenarios rise out this understanding.
- If there is only element in the list, the sum is just that element.
- If there is more than one element, the sum is the addition of the first element with the sum of the rest of the list.

Python Implementation

We'll have a function call itself recursively over this logic.

```
def list_sum(num_list):
    if len(num_list) == 1:
        return num_list[0]
    else:
        return num_list[0] +
         list_sum(num_list[1:])
```

- In the function above, with each iteration, the first element in the list is added to the rest of the list, which is done by calling the function again with the rest of the list.
- The series of function calls breaks down the problem until it becomes absolutely small after which it cannot get any smaller.

The Three Laws of Recursion

The following are the three fundamental principles of Recursion.

- A recursive algorithm must have a base case.
- A recursive algorithm must move toward the base case by changing its state.
- A recursive algorithm must call itself recursively.

Comparing this to the problem we just solved with recursion could help us more in understanding the laws.

The Three Laws of Recursion Contd..

- A base case is a condition that helps the algorithm from recursing again.
- In our example, it is the first condition where the length of the list is checked to be 1.
- The base case is the problem broken down into its smallest form and in our case, it is a list containing only one element whose sum is simply the element itself.
- The base case will vary for different problems.
- The second law requires a change of state with each recursion, inching towards the base case.
- This is accomplished by making the data that the algorithm uses smaller and smaller. In our example, it is the list of numbers.
- The base case is a list containing only a single element as we already discussed.

The Three Laws of Recursion Contd..

- The final law says that the algorithm must call itself, which is the definition of recursion.
- In our example, with each iteration, the state of our data (the input list) is being changed and recursed again.
- Lets look at another example to help us understand this better.
- Lets calculate factorial of a number using recursion.

```
def recursive_factorial(n):
    if n == 1:
        return n
    else:
        return n*recursive_factorial(n-1)
recursive_factorial(5)
120
```

Palindrome Checker Using Recursion

- For a string to be a palindrome, the string and its reverse must be the same.
- The problem could be broken down by progressively comparing the first character with the last and then deciding whether or not to proceed ahead with recursion.
- If the characters are the same, change the state of the data, that is consider only the character besides the characters we just compared and recurse again.
- We repeat this until the point the string cannot be broken down any further.

```
def palindrome_checker(str):
    if len(str) == 1:
        return True
    if str[0] != str[-1]:
        return False
    return palindrome_checker(str[1:-1])
```

Summary

- Recursion is a programming paradigm that can help us solve complex and long-winding problems by simpler means.
- We understood the three laws that govern recursion.
  - The requirement of a base case.
  - The The change of state with each recursion, and
  - The ability and need to call itself which achieves recursion.