Python: Recursive Functions

4th Lecture

1. Introduction

We mentioned the structure of the function in the previous lecture as well as we highlighted many types of function, which may be utilized in Python. We also briefly noted that Python supports many inbuilt functions. In some cases, you can decompose a complex problem into smaller problems of exactly the same form. In these cases, functions can call other functions and that sometimes this helps to make a complex problem more manageable in some way. It turns out that not only can functions call other functions but the sub-problems can all be solved by using the same function. This design strategy is called **recursive design**, and the resulting functions are called **recursive functions**.

2. Function Call Another Function

It is important to understand that each of the functions we write can be used and called from other functions in the same program. This is one of the most important ways that computer scientists take a large problem and break it down into a group of smaller problems. This process of breaking a problem into smaller sub-problems is called **functional decomposition**.

Example 1:

```
def square(x):
    y = x * x
    return y

def sum_of_squares(x, y, z):
    a = square(x)
    b = square(y)
```

```
    c = square(z)
    return a + b + c


a = -5
b = 2
c = 10
result = sum_of_squares(a, b, c)
print(result)
```

The output
129

Note: Building functions code according to calling it is very important.

3. Recursive Functions in Python

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfills the condition of recursion, we call this function a ***recursive function***. On the other words, A **recursive function** ***is a function that calls itself***. To prevent a function from repeating itself indefinitely, it must contain at **least one selection statement**. This statement examines a condition called a **base case** to determine whether to stop or to continue with another **recursive step**.

Example 2: Define a function displayRange that prints the numbers from a lower bound to an upper bound?

```
   "Outputs the numbers from lower to upper."
def displayRange(lower, upper):
      while lower <= upper:
       print(lower, end="  ")
       lower = lower + 1
displayRange(100, 200)
```

The output

```
100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118
119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137
138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156
157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175
176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194
195  196  197  198  199  200
```

The equivalent recursive function performs similar primitive operations, but the **loop** is replaced with a **selection statement**, and the assignment statement is placed with a recursive call of the function. Here is the code with these changes:

```
   "Outputs the numbers from lower to upper."
def displayRange(lower, upper):
 if lower <= upper:
      print(lower, end="  ")
      lower = lower + 1
      displayRange(lower + 1, upper)
displayRange(100, 200)
```

The output

```
100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118
119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137
138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156
157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175
176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194
195  196  197  198  199  200
```

Although the syntax and design of the two functions are different, the same algorithmic process is executed. Each call of the recursive function visits the next number in the sequence, just as the loop does in the iterative version of the function.

Example 3: This example is a recursive function that builds and returns a value. The sum function computes and returns the sum of the numbers between these two values. In the recursive version, sum returns 0 if lower exceeds upper (the base case). Otherwise, the function adds lower to the sum of lower + 1 and upper and returns this result. Here is the code for this function:

```
def sum(lower, upper):
      if lower > upper:
        return 0
      else:
        return lower + sum(lower + 1, upper)
print(sum(1, 100))
```

The output

```
5050
```

Example 4: Factorial can be written in Python much the same way it is defined in Mathematics. The if statement must come first and is the statement of the base case. The recursive case is always written last.

```
def factorial(n):
    if n==0:
        return 1
    return n * factorial(n - 1)
IntergerN=int(input("The enter value: "))
print("The result is: ",factorial(IntergerN))
```

The output
The enter value: 5
The result is: 120

4. Tracing a Recursive Function (run-time stack)

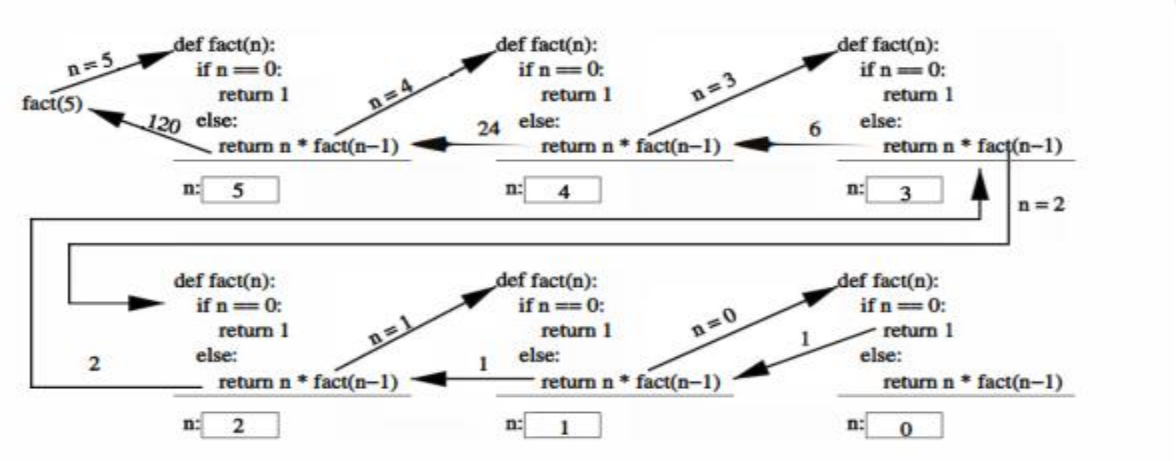
To get a better understanding of how recursion works, it is helpful to trace its calls. Let's do that for the recursive version of the sum function. You add an argument for a margin of indentation and print statements to trace the two arguments and the value returned on each call. The first statement on each call computes the indentation, which is then used in printing the two arguments. The value computed is also printed with this indentation just before each call returns. Here is the code, followed by a session showing its use:

```
    "Returns the sum of the numbers from lower to upper, and outputs a trace of the arguments and
return values on each call."
def sum(lower, upper, margin):
    blanks = " " * margin
    print(blanks, lower, upper)
if lower > upper:
        print(blanks, 0)
        return 0
else:
        result = lower + sum(lower + 1, upper, margin + 4)
        print(blanks, result)
        return result
sum(1, 4, 0)
```

The output

```
1 4
    2 4
        3 4
            4 4
                5 4
                0
            4
        7
    9
10
```

The another way to trace its calls. Here is what the stack diagram looks like for this sequence of function calls (factorial function in example):



5. Exercises

1. Draw a special flowchart that represents the Example 1?
2. The naive way to compute an for an integer n is simply to multiply a by itself n times a^n = a∗a∗a∗. . .∗a. We can easily implement this using a simple accumulator loop.

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
print (loopPower(3,4))
```

   Write a recursive function to get the same result?
3. Write a recursive function that computes the nth Fibonacci number. The Fibonacci numbers are defined as follows: Fib(0) = 1, Fib(1) = 1, Fib(n) = Fib(n − 1) + Fib(n − 2). Write this as a Python function and then write some code to find the tenth Fibonacci number?
4. Write algorithm and Python coding for a recursive function that computes the following equation:

$$C_k^n = \frac{n!}{k!\,(n-k)!}$$

<Best Regards>

Dr. Raaid Alubady