Recursion in Python

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

Recursion Definition

From Webster on-line dictionary

<table>
<thead>
<tr>
<th>Lemma</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etymology</td>
<td>From French récursif, the act of recurring</td>
</tr>
</tbody>
</table>

| Definition | noun (mathematics) an expression such that each term is generated by repeating a particular mathematical operation |

Or a more pragmatic definition:

... recursive: adjective, see recursive ...

Recursion Intuitive Definition

Problem: to have to wash a heap of dishes.

- lazy approach, :-)

Algorithm:

- if the sink is empty everything is done,
- if it is not empty then:
  - to wash the first dish in the heap and
  - to ask to the closest friend to wash the remaining heap of dishes.

Note, every involved person should do less work than whose is involving him/her.
Recursion: Recursive Function

A function is called recursive when it is defined through itself.

Example: Factorial
- $5! = 5 	imes 4 	imes 3 	imes 2 	imes 1$
- Note that: $5! = 5 	imes 4!$, $4! = 4 	imes 3!$ and so on.

Potentially a recursive computation.

From the mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n=0, \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

When $n=0$ is the base of the recursive computation (axiom) whereas the second step is the inductive step.

Recursion: What in Python?

Still, a function is recursive when its execution implies another invocation to itself:
- directly, i.e. in the function body there is an explicit call to itself;
- indirectly, i.e. the function calls another function that calls the function itself.

```python
def fact(n):
    return 1 if n<=1 else n*fact(n-1)
```

```python
if __name__ == '__main__':
    for i in [5, 7, 15, 25, 30, 42, 100]:
        print('fact({0:3d}) :- {1}'.format(i, fact(i)))
```

Side Notes on the Execution.

At any invocation the run-time environment creates an activation record or frame used to store the current values of:
- local variables, parameters and the location for the return value.

To have a frame for any invocation permits to:
- trace the execution flow;
- store the current state and restore it after the execution;
- avoid interferences on the local variables.

Warning:
Without any stopping rule, the inductive step will be applied "forever".
- Actually, the inductive step is applied until the memory reserved by the virtual machine is full.
Recursion

Design: Requirements

Recursive solutions better apply to problems that can be inductively define.

To use the solution of a smaller/easier version of the problem to solve the problem:
- recursive call;

To recognize when the problem is so easy that a direct solution is feasible
- stopping/termination rule.

Recursive solutions are more elegant and intuitive than the iterative solutions.

Recursion

Design: Requirements

Recursive solutions better apply to problems that can be inductively define.

To use the solution of a smaller/easier version of the problem to solve the problem:
- recursive call;

To recognize when the problem is so easy that a direct solution is feasible
- stopping/termination rule.

Recursive solutions are more elegant and intuitive than the iterative solutions.

Recursion

Case Study: Fibonacci Numbers

Leonardo Pisano, known as Fibonacci, in 1202 in his book "Liber Abaci" faced the (quite unrealistic) problem of determining:

"how many pairs of rabbits can be produced from a single pair if each pair begets a new pair each month and every new pair becomes productive from the second month on, supposing that no pair dies"

to introduce a sequence whose i-th member is the sum of the 2 previous elements in the sequence. The sequence will be soon known as the Fibonacci numbers.

Recursion

Case Study: Fibonacci Numbers

Leonardo Pisano, known as Fibonacci, in 1202 in his book "Liber Abaci" faced the (quite unrealistic) problem of determining:

"how many pairs of rabbits can be produced from a single pair if each pair begets a new pair each month and every new pair becomes productive from the second month on, supposing that no pair dies"

to introduce a sequence whose i-th member is the sum of the 2 previous elements in the sequence. The sequence will be soon known as the Fibonacci numbers.

Recursion

Case Study: Fibonacci Numbers

Fibonacci numbers are recursively defined:

\[
f(n) = \begin{cases} 
0 & \text{if } n=0, \\ 
1 & \text{if } n=1 \text{ or } n=2, \\ 
\text{fibo}(n-1)+\text{fibo}(n-2) & \text{otherwise}. 
\end{cases}
\]

The implementation comes forth from the definition:

```python
def fibo(n):
    if n<=1:
        return n
    else:
        return fibo(n-1)+fibo(n-2)

if __name__ == '__main__':
    for i in [5, 7, 15, 25, 30]:
        print('fibo({0:3d}) :- {1}'.format(i, fibo(i)))
```

cazzola@ulik:~/esercizi-pa>python3 fibonacci.py
fibo( 5) :- 5
fibo( 7) :- 13
fibo( 15) :- 610
fibo( 25) :- 75025
fibo( 30) :- 832040

Recursion

Case Study: Fibonacci Numbers

Fibonacci numbers are recursively defined:

\[
f(n) = \begin{cases} 
0 & \text{if } n=0, \\ 
1 & \text{if } n=1 \text{ or } n=2, \\ 
\text{fibo}(n-1)+\text{fibo}(n-2) & \text{otherwise}. 
\end{cases}
\]

The implementation comes forth from the definition:

```python
def fibo(n):
    if n<=1:
        return n
    else:
        return fibo(n-1)+fibo(n-2)

if __name__ == '__main__':
    for i in [5, 7, 15, 25, 30]:
        print('fibo({0:3d}) :- {1}'.format(i, fibo(i)))
```

cazzola@ulik:~/esercizi-pa>python3 fibonacci.py
fibo( 5) :- 5
fibo( 7) :- 13
fibo( 15) :- 610
fibo( 25) :- 75025
fibo( 30) :- 832040

Recursion

Case Study: Fibonacci Numbers (The Execution Tree).

```
fibo(3)
fibo(1)
fibo(2)
fibo(0)
```
Recursion Easier & More Elegant

The recursive solution is more intuitive:

```python
def fibo(n):
    return n if n<=1 else fibo(n-1)+fibo(n-2)
```

The iterative solution is more cryptic:

```python
def fibo(n):
    Fib1, Fib2, FibN = 0,1,1
    if n<=1:
        return n
    else:
        for i in range(2, n+1):
            FibN=Fib1+Fib2
            Fib1=Fib2
            Fib2=FibN
        return FibN
```

But ...

Recursion vs Iteration

- **Iteration**: explicit loop
  - Termination: the loop condition fails

- **Recursion**: repeated calls to the function
  - Termination: the axiom is reached

Both can loop forever.

Balance
- The choice is between efficiency (iteration) and easiness/elegance (recursion)

The iterative implementation is more efficient:

```python
>>> from fibonacci import fibo
>>> Timer('fibo(10)', 'from fibonacci import fibo').timeit()
26.872473001480103
>>> Timer('fibo(10)', 'from ifibonacci import fibo').timeit()
657.5257818698883
```

The overhead is mainly due to the creation of the frame but this also affects the occupied memory.

As an example, the call `fibo(1000)`

- gives an answer if calculated by the iterative implementation;
- raises a `RuntimeError` exception in the recursive solution.

```python
>>> fibonacci.fibo(1000)
... File "fibonacci.py", line 2, in fibo
    return n if n<=1 else fibo(n-1)+fibo(n-2)
RuntimeError: maximum recursion depth exceeded in cmp
```

The Towers of Hanoi

**Definition (Édouard Lucas, 1883)**

**Problem Description**

There are 3 available pegs and several holed disks that should be stacked on the pegs. The diameter of the disks differs from disk to disk each disk can be stacked only on a larger disk.

The goal of the game is to move all the disks, one by one, from the first peg to the last one without ever violate the rules.
The Towers of Hanoi

The Recursive Algorithm

3-Disk Algorithm

1) 3
3
2) 1
1 2
5) 1 2 3
4)

n-Disk Algorithm

Base: n=1, move the disk from the source (S) to the target (T);
Step: move n-1 disks from S to the first free peg (F), move the last disk to the target peg (T), finally move the n-1 disks from F to T.

The Towers of Hanoi

Python Implementation

```python
def display(pegs):
    for j in range(len(pegs[0])):
        for i in range(3):
            print(" {0} ".format(pegs[i][j]), end="")
        print()
        print()

def move(pegs, source, target):
s = pegs[source].count(0)
t = pegs[target].count(0) - 1
pegs[target][t] = pegs[source][s]
pegs[source][s] = 0
def moveDisks(pegs, disks, source, target, free):
    if disks <= 1:
        print("moving from {0} to {1}".format(source, target))
        move(pegs, source, target)
    else:
        moveDisks(pegs, disks-1, source, free, target)
        print("moving from {0} to {1}".format(source, target));
        move(pegs, source, target);
        moveDisks(pegs, disks-1, free, target, source);

if __name__ == '__main__':
    pegs = [list(range(1,4)), [0]*3, [0]*3]
    print("Start!")
    display(pegs)
    moveDisks(pegs, 3, 0, 2, 1)
```

The Towers of Hanoi

3-Disk Run

```
[0:12] cazzola@ulik:/~esercizi-pa-python3 hanoi.py
Start!
moving from 0 to 1 moving from 0 to 2 moving from 1 to 2
0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 1 0 0 0 2
3 0 0 3 2 1 0 2 3 1 0 3
moving from 0 to 2 moving from 2 to 1 moving from 1 to 0 moving from 0 to 2
0 0 0 0 0 0 0 0 0 0 0 1
2 0 0 0 1 0 0 0 0 0 2
3 0 1 3 2 0 1 2 3 0 0 3
```

The Myth

The myth tells about some Buddhist monks devout to Brahma should engage in solving the problem with 64 golden disks and when solved the world will end.

Can we be quiet?

How many operations will be necessary to end the computation?

At every call of moveDisks() (at least) two recursive calls to itself are done. This can be proved very close to $2^n$.

If we could move one disk per second we need:

$$2^{64} = 18446744073709551616$$

that is about 586549 billions of years and the age of the whole universe is estimated of: 13.7 billions of years.