Python and Bioinformatics

Pierre Parutto

October 10, 2016

Contents

1 Recursivity **2**

1.1 Recursive Thinking 2

1.2 Recursive Functions 2

1.3 Recursivity On Integer Sequences 3

1.3.1 Example - Factorial Function 3

1.4 Recursivity On Sequences 4

1.4.1 Example - Number Of Elements On Lists 4

1.4.1.1 Unrolling The Functions Calls 5

1.4.2 Example - Number Of Elements On Strings 5

Chapter 1

Recursivity

In computer-science recursivity provides another way for solving problems. Recursively solving a problem consists in solving sub-problems and combining their solutions to get the solution of the initial problem. Recursive thinking is not fit for all types of problems but is very natural when thinking about integer sequences (u_n given as a function of u_{n-1}), trees (process a node and apply the same computation to its sons) and graphs (process a node and apply the same computation to its successors).

1.1 Recursive Thinking

Definition 1 *Recursively solving a problem consists in two steps:*

1. **Breaking down** *the problem in sub-problems (easier to solve);*
2. **Merging** *the solutions of the sub-problems to obtain the solution of the initial problem.*

This is a way of thinking on how to solve a problem, it is done outside of the computer on a sheet of paper. You specify the problem and think of how to break it down into sub-problems and to merge their solutions.

1.2 Recursive Functions

Definition 2 *A recursive function is a function that calls itself.*

A recursive function allows to implement the recursive thinking in a computer program. A recursive function must distinguish between 2 types of input values:

Definition 3 *Recursive function cases:*

1. *The base case: the input argument corresponds to a minimal sub-problem that can be solved without calling the function.*
2. *The recursive case: the input argument is not minimal, the problem is broken down into one or multiple sub-problem(s). The function is called on each sub-problem and their solutions merged together.*

In the base case, the function must simply return the value associated to a trivial sub-problem, for example the empty list or the value 0. On the other hand, the recursive case is more involved as it requires to think how to break down the problem and how to merge the solutions of the sub-problems.

We call these two steps the recursive formulation of a problem.

Warning

Note that when you have the recursive formulation of a problem it is, in most cases, straightforward to transform it into Python code.

Remark

We also call the breakdown phase the descending and the merging phase the ascending phase.

1.3 Recursivity On Integer Sequences

For an integer sequences u given in a recursive form:

$$u_n = \begin{cases} f(u_{n-1}) & n = 0 \\ u_0 & \text{otherwise} \end{cases}$$

where f is some function and $n \geq 0$ is an integer.

The Python formulation of this kind of problem is straightforward.

1.3.1 Example - Factorial Function

The factorial function $n!$ (for $n > 0$ an integer) is defined as:

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

We are now going to implement the recursive function `facto(n: int) -> int` that returns the value $n!$:

```
def facto(n):
    if n == 0:
        return 1
    return n * facto(n-1)
```

Remark

Note that it is not necessary in the previous code to put an `else` case because the `return` statement already quits the function. Hence in the case where `n` is `0`, the function quits with the return value `1`.

1.4 Recursivity On Sequences

A sequence can be seen recursively as a first element and the remaining sequence. The base case is when the sequence is empty.

In Python for a sequence `s`, the first element is accessed using the syntax `s[0]` and the remaining elements can be accessed by `s[1:]`.

The empty sequence is `""` for strings and `[]` for lists.

1.4.1 Example - Number Of Elements On Lists

Let us write a recursive function `my_len(l: list) -> int` that returns the number of elements in the list `l`.

To solve this question we have to think about the two cases:

- The base case is when the list is empty, then its length is 0.
- In the recursive case, we can decompose the list in its first element `l[0]` and the remaining elements `l[1:]`. In this case the size is 1 plus the size of the list `l[1:]`.

This reasoning is translated in Python as:

```
def my_len(l):
    if l == []:
        return 0
    return 1 + my_len(l[1:])
```

Remark

Note that we do not care about the value of `l[0]`.

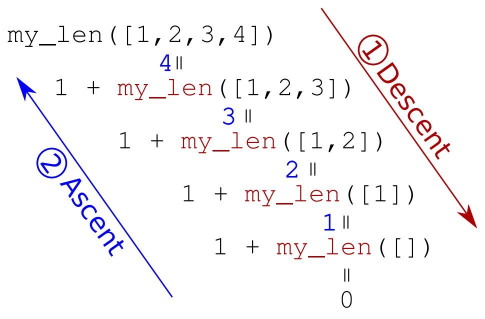

Figure 1.1: The recursive calls (red) and computed values (blue) for computing the length of a four elements list.

1.4.1.1 Unrolling The Functions Calls

Behind the scene, recursivity makes use of the function stack implemented in Python that stores the successive function calls.

In order to debug your code, it is thus important to understand what happens in Python when you make the call `my_len([1,2,3,4])`. Figure 1.1 presents the different functions calls and the returned values.

To compute the value `my_len([1,2,3,4])` Python needs to compute the value `my_len([1,2,3])`. Once again to compute `my_len([1,2,3])` Python needs to compute `my_len([1,2])`. These calls goes on until the call `my_len([])` for which the value is directly computed and is 0. From there, the values of the previous calls are computed until reaching the original call.

1.4.2 Example - Number Of Elements On Strings

The reasoning for strings is the same as the one for lists, except that the empty string has the value `""`:

```
def my_len(s):
    if s == "":
        return 0
    return 1 + my_len(l[1:])
```