Python: Recursion

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts of Southern Switzerland

Content created in collaboration with: L. Grossi, F. Landoni

Subprograms: procedures and functions

In general a subprogram can be a procedure (do not return a result) or a function (returns a result).

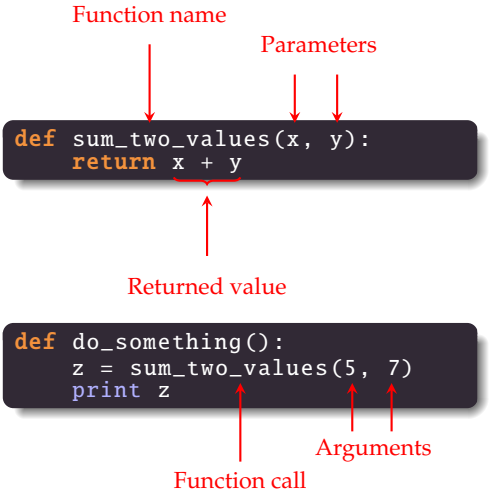
In Python, by default a function without a return value returns None. We therefore speak only of functions.

A function is a block of statements that has a name, and that can be called within any code expression.

The usage of functions allows to better structure and reuse the source code.

Refresher

Functions example



Refresher

Functions completion

A function completes when:

- all the internal statements are executed
- the `return` statement is executed
- an exception is thrown (we will discuss this later)

Refresher

The `return` statement

The behavior of the `return` statement is similar to the one of the `break` statement that we discussed before.

In the case of the `return` statement, the execution of the function terminates in that point.

The return statement is used to retrieve a value from the function.

The return statement is not mandatory in a Python function. If a function lacks a return statement, it will automatically return the value `None`.

Refresher

Recursion

Each procedure or function, inside its body, is able to call other procedures or functions, which in turn can call further ones.

A special case of these calls is when the procedure or function calls itself.

We speak of recursive function or procedure, and therefore of recursion.

Recursion

In recursion, an algorithm uses itself and performs the same operations "in loop".

Therefore, each call to a procedure or function of recursion is similar to an iteration of a repetition statement (`for`, `while`). However, the formulation of the algorithm is different.

The behavior also differs, in particular as regards the occupation of memory.

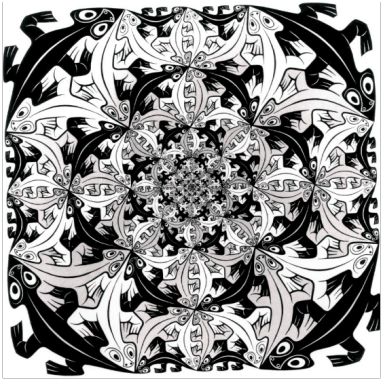
Typically, recursion produces progressions of results. Nearby invocations produce similar results.

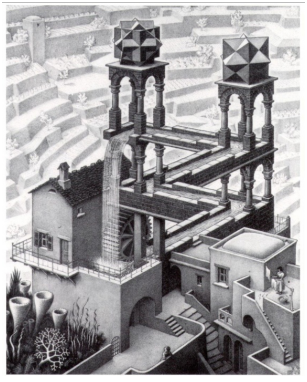
Recursion

We can observe the recursion in nature:

Recursion

Some artists have been inspired by recursive processes:

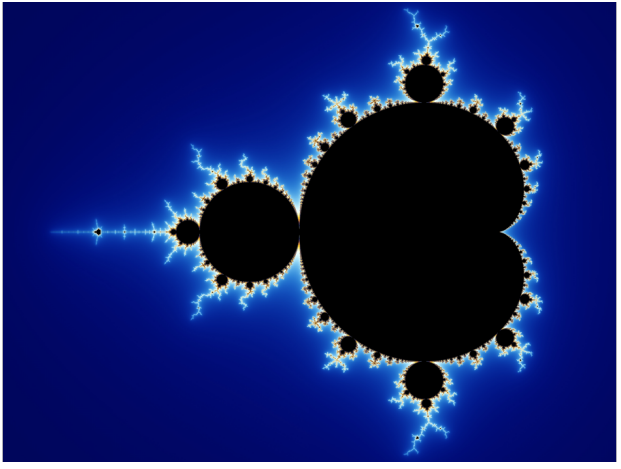




Maurits Cornelis Escher (1898 – 1972). Dutch graphic artist.
Often mathematically inspired. Impossible constructions and explorations of infinity.

The Mandelbrot set

Benoît Mandelbrot (1924 - 2010) was a Polish, naturalized French, mathematician known for his work on fractal geometry.



Fractals

"A fractal is a geometric object with internal homothetics: it is repeated in its form in the same way on different scales, and therefore by enlarging any part of it, a figure similar to the original is obtained."

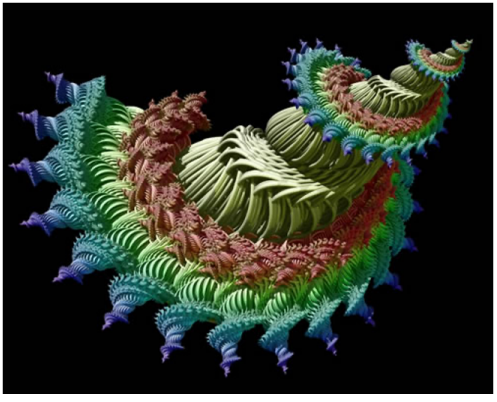
— *Wikipedia*

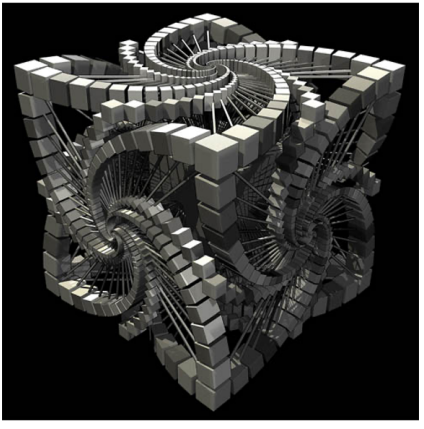
Fractals are self-similar patterns. By self-similar we mean that they are similar from near and far. Fractals can be exactly identical at any scale of size, or very similar at different scales. Fractals are self-similar to infinity, iteratively.

Fractals are not limited to geometric patterns, they can also describe processes over time. Fractal patterns of different levels and self-similarities have been observed and studied in images, sound, nature, technology, art, ...

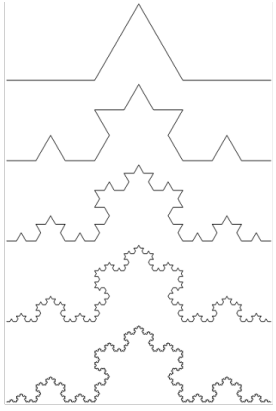
Fractals

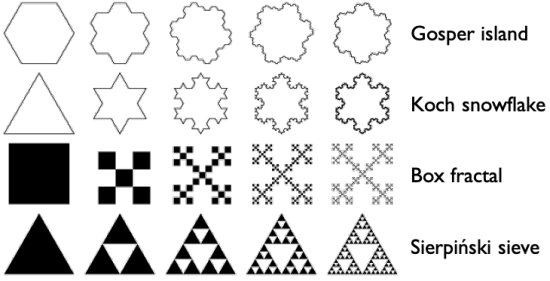
Fractals





How do recursion works?





How do we use it in engineering? The idea is to break down the problem in smaller and simpler parts, until you have to solve a very simple problem (divide et impera).

Recursion in Python

In Python it is possible to use the recursion by means of recursive calls to functions.

A recursive function is a function that calls itself.

Recursion in Python

The idea is to break down a problem into smaller parts until you have to solve a very simple problem.

Requirements:

- entry condition: the state of a function at the moment of a the call
- recursive call: within a function it is present one or more calls to itself
- exit condition: needed to stop the recursion. Essential to avoid infinite recursion.

Example: natural numbers

Recursive definition of a natural number:

$$\begin{cases} 0 & \in \mathbb{N} \\ n_k = n_{k-1} + 1 & n_k \in \mathbb{N} \text{ if } n_{k-1} \in \mathbb{N} \end{cases}$$

Example: exponetiation

Definition of exponentiation:

$$a^n = \underbrace{a * a * \cdots * a}_{\text{n times}}$$

Recursive definition of exponentiation:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a * a^{n-1} & \text{if } n > 0 \end{cases}$$

$2^0 = 1$

$2^1 = 2 * 2^0 = 2 * 1 = 2$

$2^2 = 2 * 2^1 = 2 * 2 * 1 = 4$

$2^3 = 2 * 2^2 = 2 * 2 * 2 * 1 = 8$

$a^n = a * a^{n-1} = a * a * a^{n-2} = \underbrace{a * \cdots * a}_{\text{n times}}$

Example: iterative exponetiation

```python
if __name__ == '__main__':

    def exponentiation(base, exponent):
        result = 1
        for i in range(exponent):
            result *= base

        return result


    base = 3
    exponent = 4
    print(f"{base}^{exponent} = {exponentiation(base, exponent)}")
```

Example: recursive exponetiation

```python
if __name__ == '__main__':

    def exponentiation(base, exponent):
        if exponent == 0:
            return 1

        return base * exponentiation(base, exponent-1)

    base = 3
    exponent = 4
    print(f"{base}^{exponent} = {exponentiation(base, exponent)}")
```

Example: factorial

Definition of factorial:

$$n! = \prod_{k=1}^{n} k = 1 * 2 * \cdots * (n-2) * (n-1) * n$$

Recursive definition of exponentiation:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

$0! = 1$

$1! = 1 * 0! = 1 * 1 = 1$

$2! = 2 * 1! = 2 * 1 * 1 = 2$

$3! = 3 * 2! = 3 * 2 * 1 * 1 = 6$

$n! = n * (n-1)! = n * (n-1) * (n-2) * \cdots * 2 * 1 * 1$

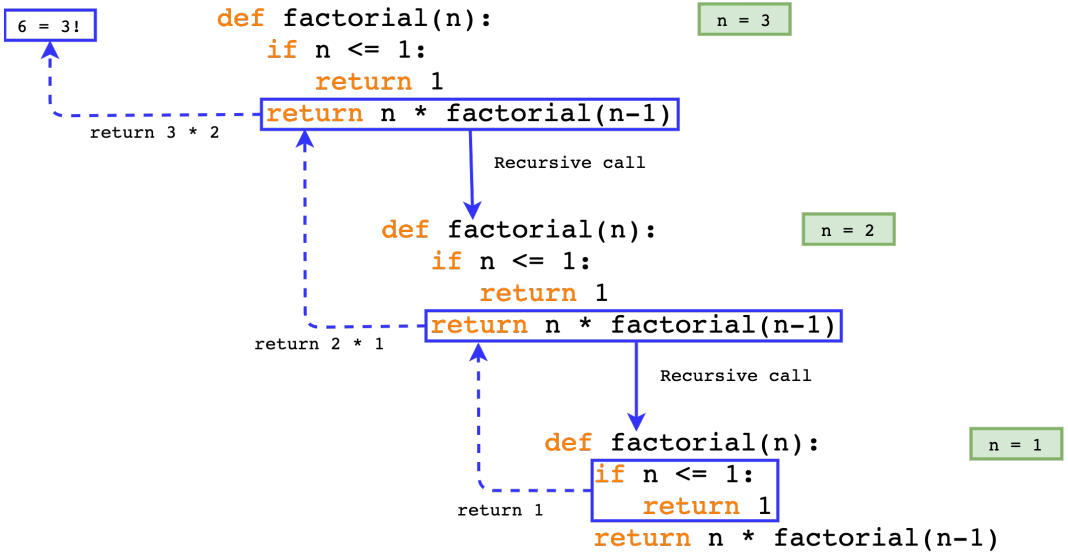
Example: iterative factorial

```python
if __name__ == '__main__':

    def factorial(n):
        factorial = 1
        for i in range(2, n+1):
            factorial *= i

        return factorial

    num = 3
    print(f"{num}! = {factorial(num)}")
```

Example: recursive factorial

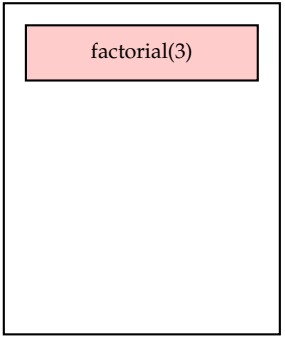
```python
if __name__ == '__main__':

    def factorial(n):
        if n <= 1:
            return 1

        return n * factorial(n-1)

    num = 5
    print(f"{num}! = {factorial(num)}")
```

Example: factorial

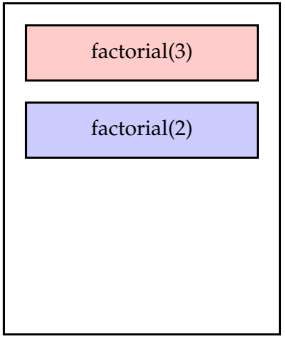


Recursion and stack

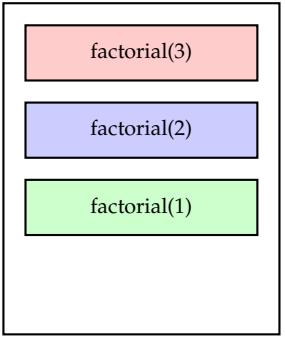
The main context executes `factorial(3)`



`factorial(3)` calls `factorial(2)`

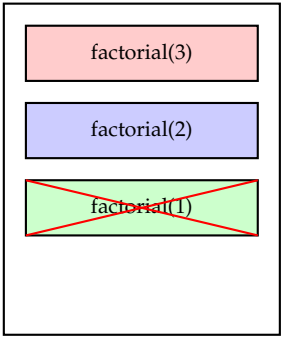


`factorial(2)` calls `factorial(1)`

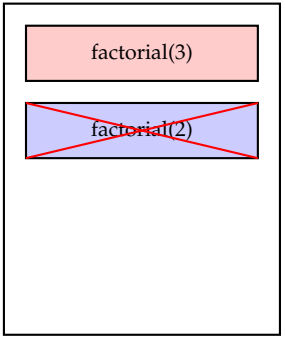


Recursion and stack

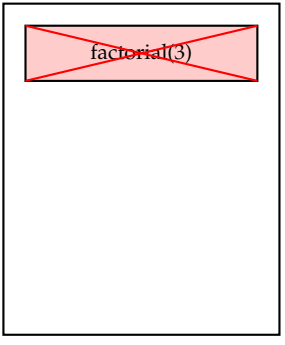
`factorial(1)` ends returning 1. The control goes back to `factorial(2)`



`factorial(2)` does the multiplication, terminates and returns 2. The control goes back to `factorial(3)`



`factorial(3)` does the multiplocation, terminates and returns 6.



Infinite recursion

If the exit condition is missing or if the solution is not simplified at each recursive step (incorrect handling of recursive calls), the procedure or function continues to call itself indefinitely, giving rise to an infinite recursion.

The program ends with a RecursionError error because it runs out of memory available to track calls.

Unfolding and folding

The unfolding phase is when the recursive procedure or function continues to call itself until it meets the termination condition.

Example:

```
factorial(4)
  4 * factorial(3)
    4 * 3 * factorial(2)
      4 * 3 * 2 * factorial(1)
```

Unfolding and folding

The folding phase is when the termination condition is met and the nested calls close.

Example:

```
4 * 3 * 2 * 1
    4 * 3 * 2
        4 * 6
            24
```

Unfolding and folding

It is possible to take advantage of the recursion both during the unfolding and rewinding phases.

```
if __name__ == '__main__':
    def recursive(counter):
        if counter == 0:
            return
        print(f"Before: {counter}")
        counter -= 1
        recursive(counter)
        print(f"After: {counter}")

    recursive(5)
```

Output:

```
Before:  5
Before:  4
Before:  3
Before:  2
Before:  1
After:  0
After:  1
After:  2
After:  3
After:  4
```

Recursion, peculiarities

If the procedure or function call occurs after all other operations have been performed, the recursion is called tail recursion.

A procedure or function with tail recursion can easily be transformed into an iterative procedure or function.

Instead, if the recursive call is the first of all operations, it is called head recursion.

In the other cases we speak of middle or multi recursion.

If function X calls function Y and, respectively, function Y calls the function X, we speak of mutual recursion or indirect recursion.

If the function calls itself multiple times, it is called multiple recursion. It should be used with extreme caution as it can lead to very inefficient programs.

Recursion or iteration

Everything you can solve with a recursion can be solved with an iteration.

Often recursion is more "elegant".
Often iteration is more efficient.

"To iterate is human, to recurse is divine"

— *Laurence Peter Deutsch*

Examples of problems solvable with recursion

The following problems can be solved using recursion:

- Recognition of palindromes: Madam I'm Adam; Sit on a potato pan, Otis.
- Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...
- Calculation of the greatest common divisor:
  1. $a = |a|, b = |b|$
  2. Sort a and b so that $a > b$
  3. If $b = 0$ then $MCD(a, b) = a$; otherwise $MCD(a, b) = MCD(b, a \bmod b)$
- Pascal or Tartaglia's triangle:

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & & 1 & & & & \\ & & & 1 & & 2 & & 1 & & & \\ & & 1 & & 3 & & 3 & & 1 & & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \end{array}$$

- Traversal of a dynamic data structure (lists, trees).

Thinking recursively: the palindromes

Problem: recognize if a string is palindrome.

Let's start with thinking how we can simplify the problem.

Which of these seems to be the most correct approach to simplify the problem?

- delete the first character,
- delete the last character,
- eliminate the central character,
- delete the first and last character,
- divide the string in half.

Palindromes: approach

Remember that we need to obtain a simpler problem, but identical to the more complex problem.

A string is palindrome if:

- The first and last characters are the same (ignoring the punctuation marks): RADAR
- The string obtained by deleting the first and last characters is still palindrome: ADA

Palindromes: exit condition

The simplest palindrome string is a string of two equal characters, but then the previous slide test (first and last equal letters) still applies.

Otherwise it is a one-character string: by definition it is palindrome. This case occurs if the string has an odd number of letters.

Otherwise it is an empty string. This is the case if the string has an even number of letters.

Example: Hanoi tower

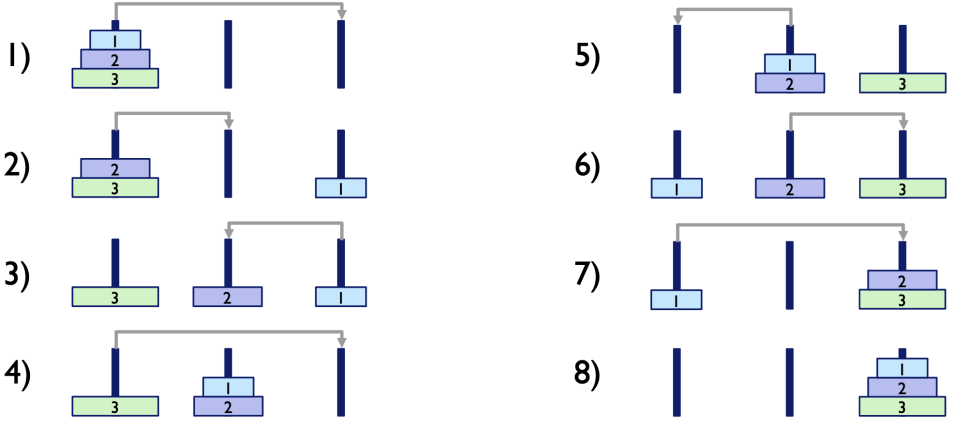
Mathematical puzzle consisting of three stakes and a certain number of discs of decreasing size, which can be inserted into any of the stakes.

The game begins with all the discs stacked on a stake in descending order to form a cone.

The object of the game is to bring all the discs to a different stake, being able to move only one disc at a time and being able to put one disc only on another larger disc, never on a smaller one.

Example: Hanoi tower - 3 discs

Solution for three discs:



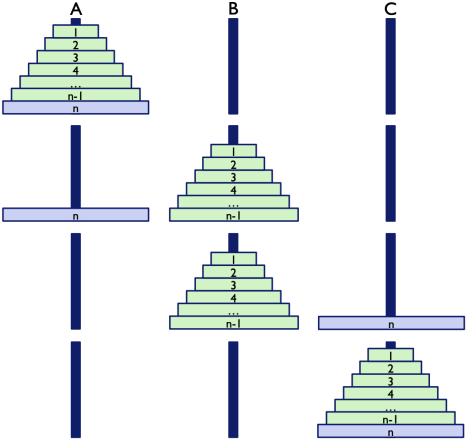
Minimum number of movements necessary: 7 ($= 2^3 - 1$)

Example: Hanoi tower - n discs

The general problem consists into moving n disks from a stake (A) to another stake (C), using the third stake (B) as a temporary deposit.

Recursive solution:

1. Move the first n-1 disks from A to B
2. Move the disk n from A to C
3. Move n-1 disks from B to C



Example: Hanoi tower - n discs

```
if __name__ == '__main__':

    def move(start, end):
        print(f"{start} -> {end}")

    def hanoi(n, start, aux, end):
        if n == 1:
            move(start, end)
        else:
            hanoi(n - 1, start, end, aux)
            hanoi(1, start, aux, end)
            hanoi(n - 1, aux, start, end)


    hanoi(3, "A", "B", "C")
```

Animation: http://www.towersofhanoi.info/Animate.aspx

Exanple: Hanoi tower - n discs

```
hanoi(3, A, B, C)
  hanoi(2, A, C, B)
    hanoi(1, A, B, C)
      move(A, C)
    end
    hanoi(1, A, C, B)
      move(A, B)
    end
    hanoi(1, C, A, B)
      move(C, B)
    end
  end
  hanoi(1, A, B, C)
    move(A, C)
  end
...
```

```
...
  hanoi(2, B, A, C)
    hanoi(1, B, C, A)
      move(B, A)
    end
    hanoi(1, B, A, C)
      move(B, C)
    end
    hanoi(1, A, B, C)
      move(A, C)
    end
  end
end
```

Summary

- Recursion
- Recursion examples (exponentiation, factorial, palindromes)
- Recursion and stack
- Infinite recursion
- Functions usage
- Unfolding and folding
- Head, middle and tail recursion
- Recursion vs iteration

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapter 19
- Python Official Documentation: https://docs.python.org/3/tutorial/
- LearnByExample: https://www.learnbyexample.org/python/