Programming Fundamentals

Recursion

João Correia Lopes

INESC TEC, FEUP

27 November 2018

GOALS

By the end of this class, the student should be able to:

- Describe recursive algorithms
- Describe how to process recursive data structures
- Describe infinite recursion and mutual recursion
- Describe significant case studies that are recursive by nature

BIBLIOGRAPHY

- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3, 2018 (Chapter 10) [PDF]
- Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist — Learning with Python 3 (RLE), 2012 (Chapter 18) [HTML]
- Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures using Python (Chapter 15) [HTML]

TIPS

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CONTENTS

RECURSION

- **Recursion** means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective
- Recursion is a method of solving problems that involves **breaking a problem down into smaller and smaller sub-problems** until you get to a small enough problem that it can be solved trivially
- Programming languages generally support recursion, which means that, in order to solve a problem, **functions can call themselves** to solve smaller sub-problems
- Recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program [1]

[1] Any problem that can be solved iteratively (with a for or while loop) can also be solved recursively. However, recursion takes a while wrap your head around, and because of this, it is generally only used in specific cases, where either your problem is recursive in nature, or your data is recursive

FACTORIAL

```
n! = n*(n-1)*(n-2)*(n-3)* ... * 1
```

- **Base case**: we know the factorial of 1[2]

```
if n == 1:
    return 1
```

- **Recursive step**: Rewrite in terms of something simpler to reach base case

```
else:
    return n * factorial(n-1)
```

[2]Without a base case, you'll have **infinite recursion**, and your program will not work.

FACTORIAL

```
n! = n*(n-1)*(n-2)*(n-3)* ... * 1
```

- **Base case**: we know the factorial of 1[2]

```
if n == 1:
    return 1
```

- **Recursive step**: Rewrite in terms of something simpler to reach base case

```
else:
    return n * factorial(n-1)
```

[2]Without a base case, you'll have **infinite recursion**, and your program will not work.

FACTORIAL

```
n! = n*(n-1)*(n-2)*(n-3)* ... * 1
```

- **Base case**: we know the factorial of 1[2]

```
if n == 1:
    return 1
```

- **Recursive step**: Rewrite in terms of something simpler to reach base case

```
else:
    return n * factorial(n-1)
```

[2]Without a base case, you'll have **infinite recursion**, and your program will not work.

Scope of a recursive function

- See the scope in Python Tutor

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

print(fact(4))
```

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

SCOPE OF A RECURSIVE FUNCTION

- See the scope in Python Tutor

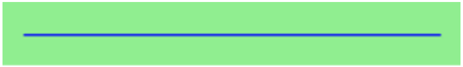
```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)

print(fact(4))
```

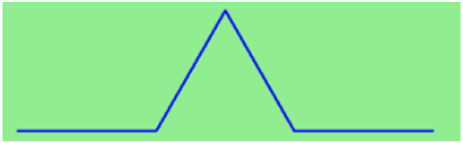
- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

KOCH FRACTAL

- A **fractal** is a drawing which also has self-similar structure, where it can be defined in terms of itself
- This is a typical example of a problem which is recursive in nature

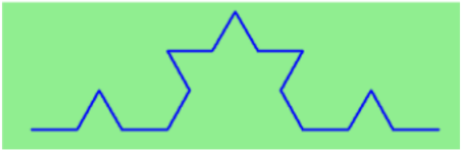


An order 0 Koch fractal

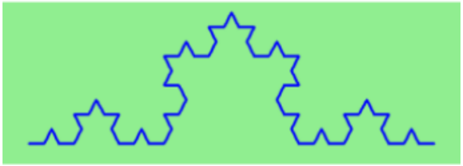


An order 1 Koch fractal

KOCH FRACTAL



An order 2 Koch fractal



An order 3 Koch fractal

⇒ `https://github.com/fpro-admin/lectures/blob/master/17/koch.py`

RECURSION, THE HIGH-LEVEL VIEW

- The function works correctly when you call it for an order 0 fractal
- Focus on is how to draw an order 1 fractal *if I can assume the order 0 one is already working*
- You're practicing **mental abstraction** — ignoring the subproblem while you solve the big problem
- See that it will work when called for order 2 *under the assumption that it is already working for level 1*
- And, in general, if I can assume the order n-1 case works, can I just solve the level n problem?
- Students of mathematics who have played with proofs of **induction** should see some very strong similarities here

RECURSION, THE LOW-LEVEL OPERATIONAL VIEW

- The trick of "unrolling" the recursion gives us an operational view of what happens
- You can trace the program into `koch_3`, and from there, into `koch_2`, and then into `koch_1`, etc., all the way down the different layers of the recursion.

⇒ `https://github.com/fpro-admin/lectures/blob/master/17/low_level.py`

RECURSIVE DATA STRUCTURES

- The organization of data for the purpose of making it easier to use is called a **data structure**
- Most of the Python data types we have seen can be grouped inside lists and tuples in a variety of ways
- Lists and tuples can also be nested, providing many possibilities for organizing data
- Example: A *nested number list* is a list whose elements are either:
  1. numbers
  2. nested number lists
- Notice that the term, *nested number list* is used in its own definition
- Recursive definitions like this provide a concise and powerful way to describe **recursive data structures**

PROCESSING RECURSIVE NUMBER LISTS

```
>>> sum([1, 2, 8])
11

>>> sum([1, 2, [11, 13], 8])
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>>
```

⇒ https://github.com/fpro-admin/lectures/blob/master/17/rec_sum.py

INFINITE RECURSION

```
    def recursion_depth(number):
        print("{0}, ".format(number), end="")
        recursion_depth(number + 1)

    recursion_depth(0)

...

    RuntimeError: maximum recursion depth exceeded ...
```

⇒ https://github.com/fpro-admin/lectures/blob/master/17/infinite_recursion.py

CASE STUDY: FIBONACCI NUMBERS

- Fibonacci sequence `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134,...`
- was devised by Fibonacci (1170-1250) who used this to model the breeding of (pairs) of rabbits
  - If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have 13+21=34, of which 21 are adults.

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
for n >= 2
```

⇒ https://github.com/fpro-admin/lectures/blob/master/17/fib.py

RECURSIVE DIRECTORIES AND FILES

- Let’s do a program that lists the contents of a directory and all its sub-directories
- First we need `get_dirlist(path)`

```
import os

def get_dirlist(path):
    """
    Return a sorted list of all entries in path.
    This returns just the names, not the full path to the names.
    """
    dirlist = os.listdir(path)
    dirlist.sort()

    return dirlist
```

⇒ https://github.com/fpro-admin/lectures/blob/master/17/dirs.py

MUTUAL RECURSION

- In addition to a function calling just itself, it is also possible to make multiple functions that call each other
- This is rarely really usefull, but it can be used to make state machines
- In mathematics, a Hofstadter sequence is a member of a family of related integer sequences defined by non-linear recurrence relations
- The *Hofstadter Female and Male sequences*:

```
F ( 0 ) = 1
M ( 0 ) = 0
F ( n ) = n - M ( F ( n - 1 ) ), n > 0
M ( n ) = n - F ( M ( n - 1 ) ), n > 0
```

⇒ https://github.com/fpro-admin/lectures/blob/master/17/hofstadter.py

EXERCISES

- Moodle activity at: LE17: Recursion