Recursion

CSE/IT 107L

NMT Department of Computer Science and Engineering

"A mirror mirroring a mirror."

"In the end, we self-perceiving, self-inventing, locked-in mirages are little miracles of self-reference."

— Douglas R. Hofstadter, I Am a Strange Loop

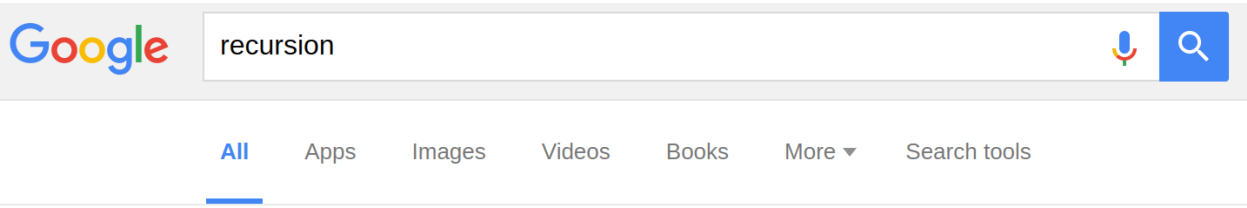


Figure 1: Google search for recursion (2016).

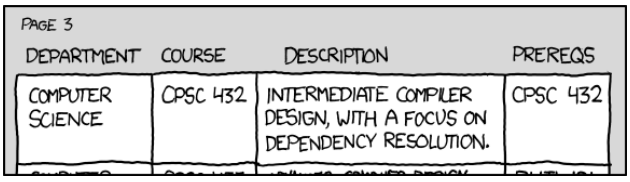


Figure 2: Dependencies `http://xkcd.com/754/`

Introduction

You have learned about while loops, Python lists and how to manipulate the elements in a list using for-loops and list comprehensions. In this lab you will learn about recursion. Recursion can be used to iterate through complicated data structures and to replicate the functionality offered by while loops and for loops. We will look at some examples of recursive functions and the type of problems recursion can help solve.

Contents

1 A Simple Recursive Function

How does Python evaluate this function?

```
1 def count(n):
2     print(n)
3     count(n + 1)
4 count(0)
```

This function never stops running. It follows this procedure:

1. To count from `n` to infinity, print `n`.
2. Continue counting from `n + 1` to infinity.

Notice the second step is self-referential. This instruction was translated into line 3 of the `count()` function. This line is an example of a recursive call where the function calls itself. In this lab, we will explore these *recursive functions*.

2 Termination

How do you make a recursive function stop running? Don't allow it to call itself every time. For example, this function only calls itself when its argument `problems` is greater than 1.

```
1 def how_to_do_homework(problems):
2     print("How to do {} homework problems".format(problems))
3     if problems <= 1:
4         print("Do the problem and then you're done.")
5     else:
6         print("Do the first problem and then do {} problems".format(problems))
7         problems -= 1  # get rid of one problem
8         how_to_do_homework(problems) # then do the rest of the problems
```

If we call the function with the argument `problems` set to 5, Python will print this output:

```
1 How do you do 5 homework problems?
2 Do the first one and then do 4 problems.
3 How do you do 4 homework problems?
4 Do the first one and then do 3 problems.
5 How do you do 3 homework problems?
6 Do the first one and then do 2 problems.
7 How do you do 2 homework problems?
8 Do the first one and then do 1 problems.
9 How do you do 1 homework problems?
10 Do the problem and then you're done.
```

3 Examples with One Recursive Call

3.1 Recursion in Mathematics

Recursive calls are not just for iterating through data, they can also be used for computations of arbitrary length. A common example of a recursive function is the factorial. The factorial of a number is the product of all numbers from 1 to that number, for example

$$\begin{aligned}
\text{factorial of } 1 &= 1 \\
\text{factorial of } 2 &= 1 \cdot 2 = 2 \\
\text{factorial of } 3 &= 1 \cdot 2 \cdot 3 = 6 \\
\text{factorial of } 4 &= 1 \cdot 2 \cdot 3 \cdot 4 = 24 \\
\text{factorial of } 50 &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \cdots \cdot 49 \cdot 50 \\
&= 30414093201713378043612608166064768844377641568960512000000000000
\end{aligned}$$

The code is very close to a mathematical definition of the factorial:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

In order to compute the factorial of an integer, you must compute the factorial of a smaller integer. Some examples. Note, unlike many other programming languages Python supports huge integers.

```
>>> factorial(1)
1
>>> factorial(4)
24
>>> factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
```

3.2 Iterating through Lists

Here is an example function that adds 1 to every element in a list. The function first checks if the list is non-empty. If so, it adds one to the first element and appends that element to the rest of the list. The rest of the list is passed as an argument to the recursive call.

This code is an *unrealistic* way of traversing a list. It would be better to use a list comprehension or a for-loop.

```
def add(elements):
    if len(elements) == 0:
        return []
    first_element = elements[0] + 1  # add 1 to the first element
    rest_of_list = elements[1:]  # shrink the list

```

```
    # work on the rest of the list
    return [first_element] + add(rest_of_list)
```

What happens when we call this function? Here is a step-by-step example of running `add([1,2,3])`.

Input of the first function call: `[1, 2, 3]`

First the function checks if the list is non-empty, then it adds 1 the first element and calls itself with a shorter input.

Input of recursive call 1: `[2, 3]`

This next function call also checks that the list is non-empty, it adds 1 to the first element and calls itself with an even shorter list.

Input of recursive call 2: `[3]`

Same thing happens with this input.

Input of recursive call 3: `[]`

There are no elements to add 1 to.

Output of recursive call 3: `[]`

When this function receives the empty list, it returns an empty list.

Output of recursive call 2: `[3 + 1] + []`

Function call 3 gets a list and adds an element to it.

Output of recursive call 1: `[2 + 1] + [3 + 1] + []`

Function call 2 also gets a list and adds an element to it.

Output of the first function call: `[3 + 1] + [2 + 1] + [3 + 1] + [] = [4, 3, 2]`

Finally, function call 1 gets the last two elements, adds the very first element after adding 1 and then returns final result.

When a function makes a recursive call by calling itself with a smaller input, it is requesting that the same thing be done to that smaller input. In this case, the first function call to `add` added 1 to the first value of the list and then passed off the rest of the work to function call 2. Function call 2 added 1 to the second element of the list, and passed the rest of the work on to function call 3. This happened until there were no more numbers in the list.

4 Examples with Multiple Recursive Calls

4.1 Iterating through Nested Lists

Recursion is especially useful when dealing with deeply nested data structures. These include trees, graphs, and other data structures. We will focus on nested lists. Consider this list:

```
[1, 2, [3, 4, [5, 6]], [7]]
```

What if we want to add 1 to every number in this list?

A useful code fragment when dealing with such a list is `type(element) == list`. This checks if the value of `element` is a Python list. We can use it to decide whether to add 1 to an element of a nested list or to add 1 to all of its sub-elements.

```
def add1(nested_list):
    new_list = []
    for element in nested_list:
        if type(element) == list:
            # if the element is a list
            # add 1 to each of its elements
            new_list.append(add1(element))
        else:
            # otherwise, just add 1 to the element
            new_list.append(element + 1)
    return new_list
```

We get these results:

```
>>> add1([1, 2, 3])
[2, 3, 4]
>>> add1([1, 2, [3, 4, [5, 6]], [7]])
[2, 3, [4, 5, [6, 7]], [8]]
```

Here a description of what the function `add1` did in the second example.

1. The function starts iterating through the given list.
2. `element = 1`

   The first element is an integer, so

   `new_list = [1 + 1]`
3. `element = 2`

   `new_list = [2, 3]`
4. `element = [3, 4, [5]]`

   We need to make a recursive call to add 1 to every element of this sub-list. Essentially, the list looks like this now:

   `new_list = [2, 3, add1([3, 4, [5]])]`

   The recursive call to `add1` makes its own recursive call in order to compute the last element of its return value:

   `[4, 5, add1([5])]`.

   The end result is `[4, 5, [6]]`. So after two recursive calls, we have:

   `new_list = [2, 3, [4, 5, [6]]]`

5. `element = [7]`

   This is a list, so a recursive call is made. A for loop iterates through the one-element list. In this recursive call `element = 7`, so 8 is added to the new list and the recursive call to the function returns `[8]`.

   `new_list = [2, 3, [4, 5, [6]], [8]]`

6. `add1` has finished iterating through the list.

We can also think of this function like this: add one to every number in the list, add one to every element in all sub-lists.

First function call `add1([1, 2, [3, 4, [5, 6]], [7]])`

Recursive call 1 `[2, 3, add1([3, 4, [5, 6]]), add1([7])]`

Recursive call 2 `[2, 3, [4, 5, add1([5, 6])], [8]]`

End result `[2, 3, [4, 5, [6, 7]], [8]]`

Here is another function that works on nested lists. It counts the number of integers or other "non-lists" in a nested list. For example,

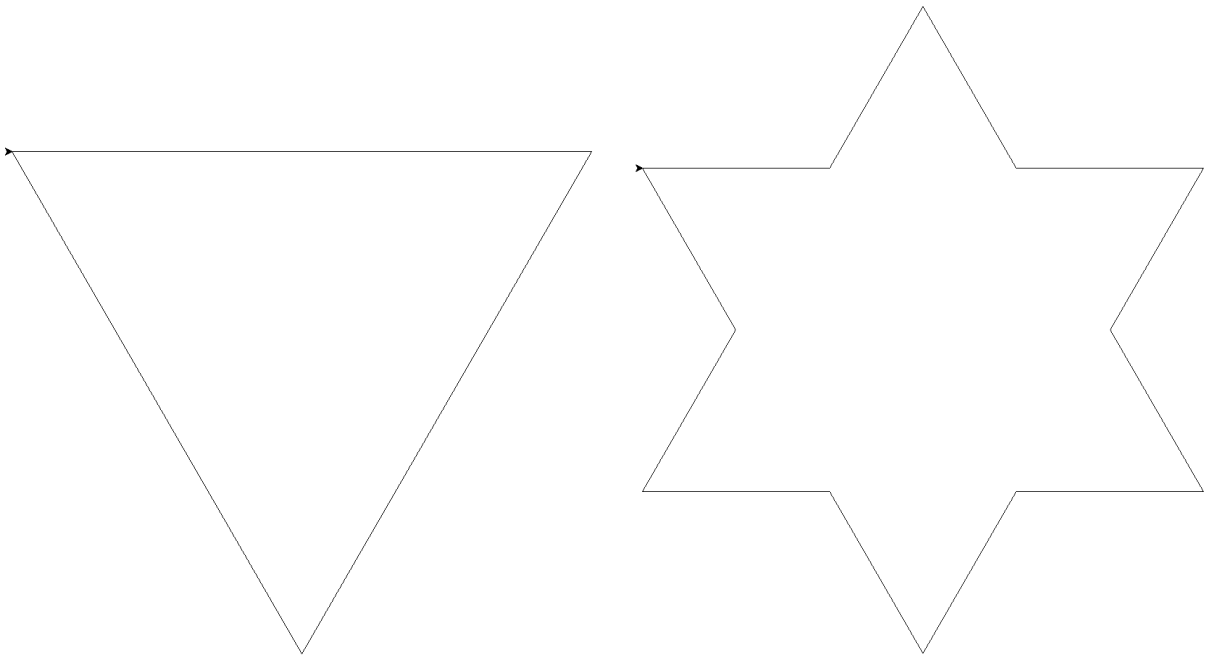
```
>>> elements_in([])
0
>>> elements_in([[[[]]]])
0
>>> elements_in([100, 200])
2
>>> elements_in([1, 2, [3, 4, [5, 6]], [7]])
7
```

This is the code for this recursive function. Recursive calls are made to count the number of elements in each sub-list. Try to run this code.

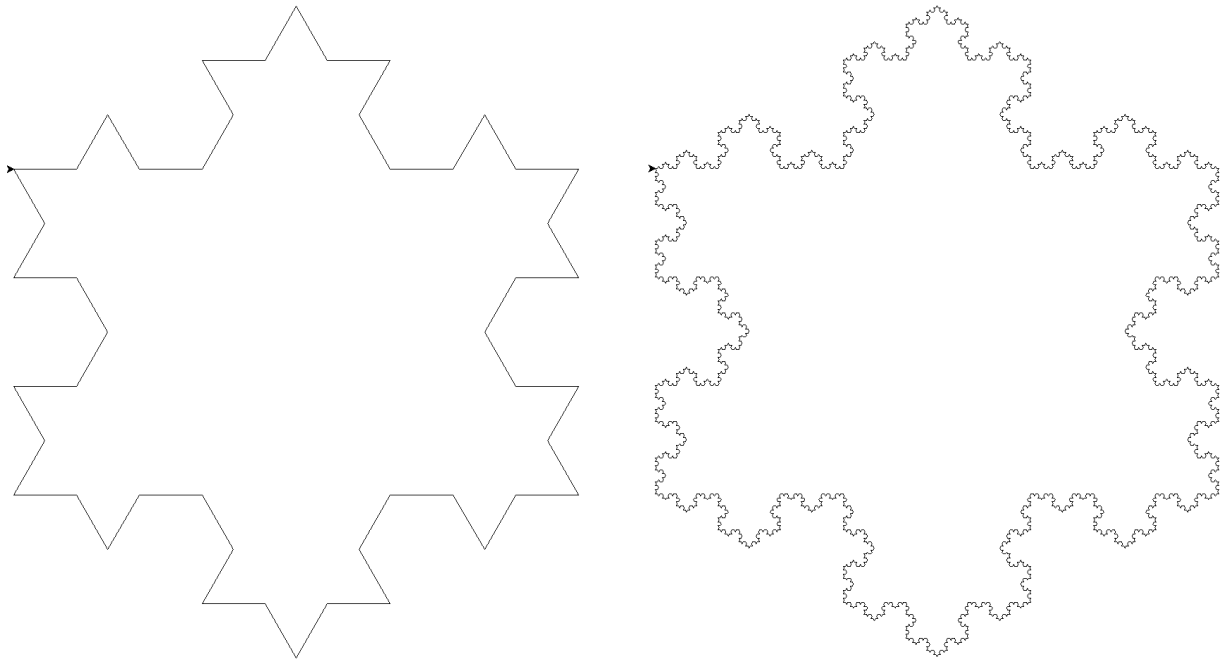
```
def elements_in(nested_list):
    count = 0
    for element in nested_list:
        if type(element) == list:
            # count the number of elements in a sub-list
            count += elements_in(element)
        else:
            # otherwise, just add 1 to the count
            count += 1
    return count
```

4.2 Drawing A Koch Snowflake

A Koch Snowflake of depth 0 is just an equilateral triangle. At depth 1, the straight lines are replaced by spiked lines which we will call Koch lines of depth 1. Here is a comparison of the two:



Instead of drawing Koch lines of depth 1 when drawing the depth 1 Snowflake, draw Koch lines of depth 2. These are drawn by replacing straight lines with Koch lines of depth 1. This gives the Koch curve at depth 2. This process can continue. The Snowflake at depths 2 and 5 are below.



To draw a Koch snowflake of depth n we can use the Turtle module to:

1. Draw an equilateral triangle with Koch lines of depth n instead of straight lines
2. To draw a Koch line of depth 0, draw a straight line.
3. To draw a Koch line of depth n of length L, we will need to draw four shorter Koch lines of length $L/3$ and depth $n-1$. This is how they are arranged:
   (a) Draw a shorter Koch line along the first third of the Koch line.
   (b) Turn 60° left and draw a shorter Koch line.
   (c) Turn 120° right and draw a shorter Koch line.
   (d) Turn 60° left and draw a shorter Koch line along the last third of the Koch line.

The self-referential instructions that ask to draw a shorter Koch line while drawing a Koch line will be translated into four recursive calls.

```
import turtle

def koch_line(width, depth=0):
    if depth <= 0:
        turtle.forward(width)
    else:
        koch_line(width / 3, depth - 1)
        turtle.left(60)

        koch_line(width / 3, depth - 1)
        turtle.right(2 * 60)

        koch_line(width / 3, depth - 1)
        turtle.left(60)

        koch_line(width / 3, depth - 1)

def koch_snowflake(width=100, depth=1):
    for _ in range(3):
        koch_line(width, depth)
        turtle.right(180 - 60)

if __name__ == "__main__":
    # move fast
    turtle.speed('fastest')

    koch_snowflake(200, 3)

    # display drawing
    turtle.done()
```

5 Overview

- A typical recursive function calls itself one or more times. (A function be recursive without calling itself. Can you think of an example?)
- A recursive function that finishes has a condition that causes it to stop calling itself.
- Whenever a problem needs small parts of its own solution in order to be solved, a recursive function may make a good solution.

Recursion is used to process nested data structures, sort lists of data, to read and run computer programs, solve mathematical problems, and to solve many other types of problems. Anything you can do with a while loop or a for loop can be done using a recursive function.

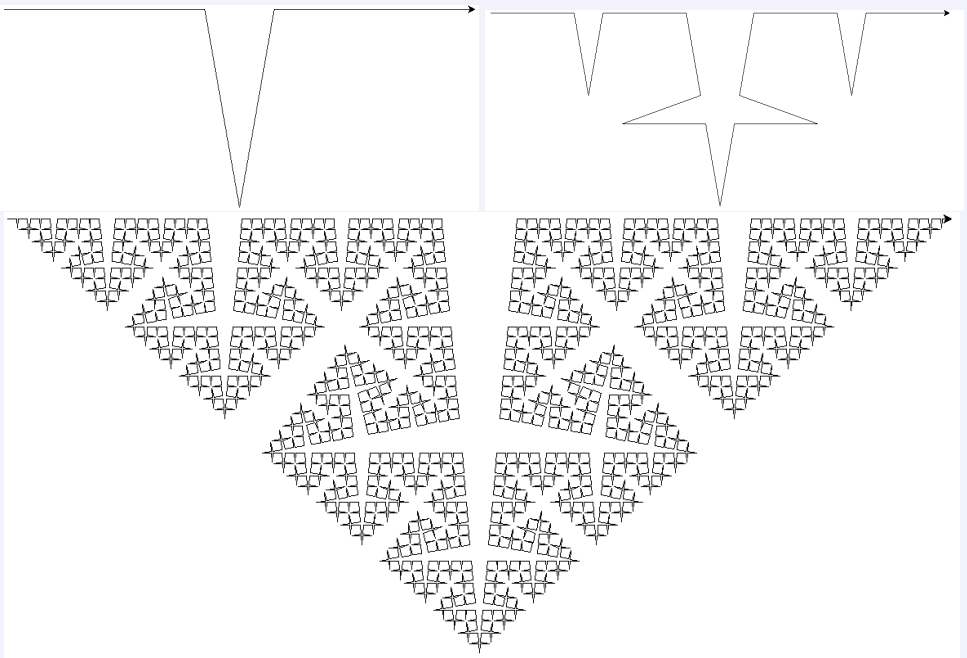
6 Exercises

Exercise 6.1 (palindrome.py).
Write a recursive function that determines whether a string is a palindrome. A palindrome is a word that is the same forwards and backwards, for example "abba", "racecar", or "amanaplanpanama". The function should take in only one argument (the string to check) and it should return either True or False.

There are easier ways to check if a string is a palindrome, but your function must be recursive.

Exercise 6.2 (cesaro.py).
Write a recursive function that draws the Cesaro torn line fractal to arbitrary depth. It is recommended to call `turtle.speed('fastest')` in order to speed up drawing.

Here are the Cesaro torn line fractals at depths 1, 2, and 6. The tear angle should be about 10°.



Hints:

- Using the Turtle module, the depth 0 Cesaro fractal of a width `w` is just `turtle.forward(w)`.
- Start by drawing the Cesaro fractal at depth one. This should not need any recursion.
- Replace all calls to the `turtle.forward` function with recursive calls to draw smaller Cesaro fractals of a lower depth.

Exercise 6.3 (nested.py).
A nested list is a list that contains one or more lists as elements. For example, the list `[1,2,3,[50,60,70],[[8888],999],10]` contains 6 elements.

- The first three elements are integers.
- **The fourth element is a list:** `[50,60,70]`.
- The fifth element is also a list. This list has two elements: another list (`[[8888],999]`) and 10.

The following functions should have recursive definitions.

1. Write a function named `element_of` that returns `True` if the first argument is within any of the sub-lists of the nested list and `False` otherwise.
2. Write a function named `filter_by_depth` that takes two arguments: an integer representing depth and a nested list. It should remove all sub-lists that are more than `depth` deep.

```
$ ls
nested.py
$ python3
>>> import nested
>>> nested.element_of(5, [1,2,3,4,5,6,7])
True
>>> nested.element_of(7, [1,2,[3,4,[5,6]],[7]])
True
>>> nested.element_of(77, [1,2,[3,4,[5,6]],[7]])
False
>>> nested.filter_by_depth(0, [1,2,3])
[]
>>> nested.filter_by_depth(1, [1,2,3])
[1,2,3]
>>> nested.filter_by_depth(5, [1,2,3])
[1,2,3]
>>> nested.filter_by_depth(2, [1,2,[3,4,[5,6]],[7]])
[1,2,[3,4],[7]]
```

Hint: use the `type` function to check if an element is a list.

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab8.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

Exercises start on page 9.