Data Structures and Algorithms with Python - Tutorial 3

September 19, 2016

0.1 Reminder: recursion (factorial)

We maybe rememeber the first tutorial, we wrote an iterative version of the factorial: the exercise was to write a function fact(n) that returns factorial of n = $n! = n \times (n-1) \times \cdots 2 \times 1$. (Remember that factorial of 0 is 1, $0! = 1! = 1$).

We had something like this as a solution:

```
def fact(n):
    accumulator = 1 # fact(0) = 1
    for i in range(1, n+1):
        accumulator*= i
    return accumulator
```

```
>>> for i in range(6):
...     print(fact(i))
1
1
2
6
24
120
```

Now you have studied recursion in class, you know that there is a much more elegant way to write this function:

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

You also now know how to use list comprehension to write the small loop more elegantly:

```
>>> [fact(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
```

Remember, this is equivalent to writing:

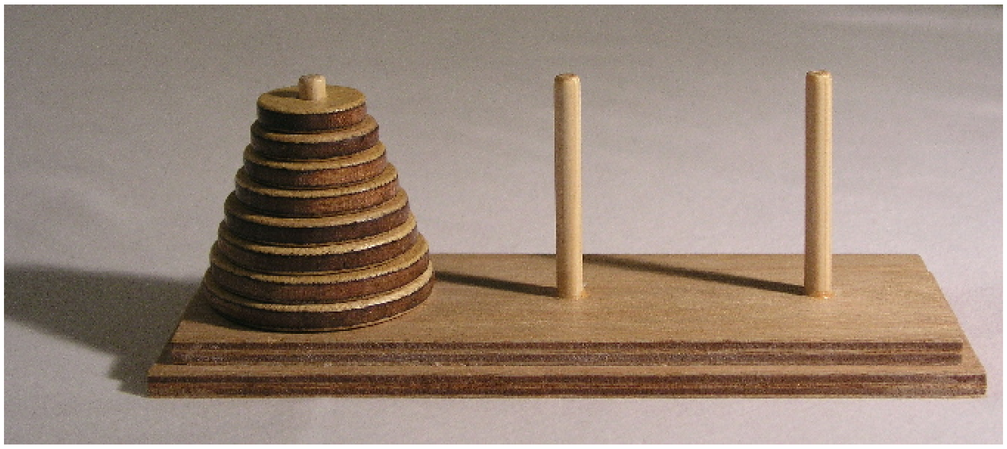
```
l = []
for i in range(6):
    l.append(fact(i))
```

1 Exercise: Towers of Hanoï

1.1 The game

The **Tower of Hanoï** is a game consiting of three rods (or towers): the *start* one, the *middle* one and the *end* one and a stack of disks of increasing size. Hanoï(n) has n disks of increasing size, with n any integer greater than 0.

The initial state of the game is the following: All n disks are on the *start* tower.



towers_of_hanoi

`Source:Wikipedia`

Then goal is to move all of the disks from the start tower to the end one while respecting the three following constraints: * Only one disk can be moved at a time. * You can only move the top disk of a tower onto the top disk of another tower. * You can only place a disk onto a larger disk.

1.2 Write a function to solve the Towers of Hanoï

First write a function move(start, end) were start and end or integers or strings that displays moving the top ring from tower start to tower end (just print something, you do not need to do anything fancy!).

Then write a **recursive** function hanoi(start, middle, end, n_rings) that solves this configuration of the problem. To do that, you have to try to find a way to break down the initial problem hanoi(n) into smaller problems hanoi(n-1)...

For this exercise, we do not even ask you to keep the state of each towers, just display the needed moves to solve the problem!

2 The Fibonacci sequence

2.1 Introduction

The famous Fibonacci sequence is the sequence $(U_n)_{n\in\mathbb{N}}$ defined by: * $U_0 = 0$
* $U_1 = 1$
* $\forall k \in \mathbb{N}, k > 1, U_k = U_{k-1} + U_{k-2}$

As an illustration, the first twenty numbers of the sequence are: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]

2.2 Testing function

Using the previous definition, write a simple function that takes as a parameter any function that given k returns U_k and tests that function for k in range(20). Use it to test all of the function you will be writing.

To test your testing function, this function should pass the test:

```python
def correct_fibo(n):
    """A horrible function that hard-codes
        the first 20 values of the Fibonacci sequence

    Paramters
    ---------
    n: int
        element of the sequence to compute

    Returns
    -------
    float: the n-th element of the Fibonacci sequence
    """
    answer = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
              144, 233, 377, 610, 987, 1597, 2584, 4181]
    return answer[n]
```

While the following should not:

```python
def wrong_fibo(n):
    """A horrible function that hard-codes
        the first 20 values of the Fibonacci sequence
        with one of them wrong
    """
    answer = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
              144, 233, 377, 610, 987, 10, 2584, 4181]
    return answer[n]
```

This testing function will be useful for you to quickly test your answers to the next questions.

2.3 Naïve implementation

Write a simple recursive function that given k, returns U_k.

2.4 Matrix version

In this part, we will use the matrix closed form formula:

$$\begin{pmatrix} U_{k+1} & U_k \\ U_k & U_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n.$$

For that we will use numpy.

2.4.1 Matrix power

First define a simple function that given a squared matrix and an integer `i`, returns the power of `i` of that matrix. You probably want to write a recursive function.

Reminder: for a squared matrix A of size $m * m$, A^0 is the identity matrix of size $m * m$ with ones on the diagonal and zeroes everywhere else, and $\forall n \in \mathbb{N}, n > 0, A^n = A\ A^{n-1}$.

Note: in numpy, the dot product is between two matrix A and B with appropriate sizes is defined as numpy.dot(A, B).

Can you make this function faster?

2.4.2 Fast exponentiation

Remembering that $a^n = (a^2)^{n/2}$ if a is even, $a * (a^2)^{n//2}$ otherwise, write a faster version of the previous function.

2.4.3 Back to Fibonacci

Now use you previously defined function to write a function that computes U_k using the matrix form.

2.5 Closed form solution

Write a function using the closed form solution:

$$U_k = \frac{\varphi^k - \psi^k}{\sqrt{5}}$$

with

$$\varphi = \frac{1+\sqrt{5}}{2}$$

and

$$\psi = \frac{1-\sqrt{5}}{2} = 1 - \varphi$$

Does your function past the test?

If not, remember that this formula gives you a close approximation, while your test function expects integers!

2.6 A Fibonacci class

Can you write a version that takes a constant time to compute elements in order? You could write a class that keeps track of the values already computed.

3 Sorting and duplicate removal

3.1 Implement the merge sort funciton

The merge-sort algorithm uses a divide and conquer strategy to sort a list: first, the list is divided in two parts of approximately equal size (divide step), then the merge-sort is applied (recursion) on each of the two sub-lists. Finally, the two resulting (sorted) lists are merged together into one sorted list.

You will first need a `merge` function to merge two **sorted** lists into one. Use the fact that both lists are sorted to not write code needlessly.

Then you need to write the actual `merge_sort` which will be a recursive function that given a list, breaks it down in two, applies `merge_sort` to the two sublists and merges them.

3.2 Duplicates removal

Modify your merge function to also remove duplicates. Do do this you can write an auxiliary function `add_if_not_duplicated` that only add an element in a list if it isn't a duplicate. Take advantage of the properties of the list and the element to not write code you do not need!!

All you need to do is then to update your merge function to use `add_if_not_duplicated`.

```
In [ ]:
```