Aims

- practice writing simple functions in python that perform numerical computation and string operations
- learn how to test python functions using the *nosetools* package

Problem 1 - Quadratic

Write the following functions in a file called **t1_sol.py** under folder task1.

- Write a function **quad**(a, b, c, x) that takes as input three floats a, b, c, a value x and returns the value of the quadratic function $f(x) = ax^2 + bx + c$. For example, $quad(1, 2, 0, 1) = 5$ and $quad(1, 0, 0, 1) = 1$. File **t1_tests.py** contains some unit tests for all the functions. After implementing the function, you can test it by typing the following command in the Linux terminal (open terminal with Ctrl+Alt+T):

```
nosetests t1_tests.py:test_quad
```

- Write a function **quadIsZero**(a, b, c, x) that takes similar arguments as **quad**, calls **quad** and returns True if the quadratic expression evaluates to zero, otherwise False (return type is boolean). For example, $quadIsZero(1, 2, 0, 1) = False$ and $quad(1, 0, -1, 1) = True$.
- Write a function **quadSolver**(a, b, c, x) that takes similar arguments as **quad** and returns the two roots of a quadratic equation with coefficients a,b,c. The roots value can be calculated using the following formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  There are two main cases:

  - if $\sqrt{b^2 - 4ac} \geq 0$ then the roots are real, in which case return them as a tuple (x_1, x_2) where $x_1 < x_2$.
  - if $\sqrt{b^2 - 4ac} < 0$ then the roots are imaginary, in which case return NaN instead of the tuple.

  HELPER: for sqrt, you need to import the numpy module and use **np.sqrt()**. This can be done as follows:

```
import numpy as np # add this once at beginning of the file

np.sqrt(2)
```

- Add 3 more unit tests for each function in **t1_tests.py** following the other examples.

Problem 2 - Strings

- write a function **toUpperCase(sentence)** which takes a lowercase string s and converts the letters at the beginning of every word to uppercase. For example:

```
toUpperCase('ana are mere si pere') = 'Ana Are Mere Si Pere'
```

  Perform this in two ways:

  - Perform a for loop ever every character:

```
for char in sentence:
  # check if letter is at the beginning of the word. This requires
    checking a boolean variable isAtBegWord at every loop
    # if letter is at beginning of word, then transform it to
    uppercase.

  # if the current character is a space ' ', set the boolean variable
    isAtBegWord to True, else False.
```

- Use the **str.split** function with delimiter ' ', which splits the string into words and returns the list of words (without spaces). Then update the first letter of each word and assemble the words back together using **str.join**(wordList). Check on google the documentation of these two functions. HINT: You can concatenate two strings with '+' (i.e. str1 + str2) and you can concatenate two lists of words also with '+' (list1 + list2). Call this function **toUpperCase2**
- Add 3 more unit tests in function **t1_tests.py:test_toUpperCase**. Test them using a similar sintax to problem 1.

Recursion

Recursion is a powerful tool in programming, which can provide simple implementations to some problems by avoiding complicated for/while loops. Our aim here is to practice implementing recursive functions.

Problem 3 - Prime numbers

Number factorisation plays a very important role in computer cryptography, as in the famous RSA algorithm. These cryptographic systems ensure the security of our online data and internet transactions. Here, we will implement some basic prime number factorisation functions using recursion.

- Write a function **isPrime(n)** which takes an integer n and returns True if n is prime, else False. You can assume the input $n \geq 2$. For example, isPrime(2) = True, isPrime(4) = False.
- Implement a function **nextPrime(n)** which returns the smallest prime higher than **n**. HINT: use recursion .. no for/while loops required.
- Write a function **primeFactors(n)** which for a given n returns the list of its prime factors, sorted in ascending order. For example, primeFactors(66556544) = [2 2 2 2 2 2 2 19 27367], primeFactors (3**3 * 2**2) = [2 2 3 3 3]. HINT: first write a helper function **primeFactorsHelper(n,k)** which checks if n is divisible by k, and if true then divides n by k and recursively calls itself, otherwise tries the next prime number (again with recursion).
- Add three more unit tests for each function above.

Problem 4 - Fibonnaci numbers

- Implement a recursive function $fib(n)$ which returns the n-th fibonnaci number. The Fibbonaci sequence is defined as:

$$\begin{cases} fib(1) = 1 \\ fib(2) = 1 \\ fib(n) = fib(n-1) + fib(n-2), \forall n \geq 3 \end{cases}$$

- Implementing the same function iteratively, i.e. using a for loop instead of recursion. Call this function $fib2(n)$
- Cross-check the output of the two functions. That is, write a unit_test function **test_fibCrossCheck()** in **t1_tests.py** that loops over a range of numbers $n \in [1..100]$ and checks if $fib(n) = fib2(n)$.

Problem 5 - Towers of Hanoi

There is an ancient legend that in Hanoi there is an elegantly-crafted cosmic puzzle which controls the birth and death of the world. It consists of three pegs, which we will call A, B and C, and (initially, at the beginning of time) a tower of 64 disks with holes in the middle, stacked up on peg A. The disks are all different sizes, and are stacked in order of size, largest at the bottom, smallest at the top. A team of untiring priests works endlessly in shifts, moving one disk at a time, from one peg to another, with the aim of transferring the entire tower from its original peg A to peg C. But they must at all times obey the sacred rule, which is that a larger disk can never be placed on top of a smaller disk. When they complete their task time will end and the world will crumble to dust. Fortunately finishing the puzzle will take a long time.

Try the Towers of Hanoi puzzle with some coins or scraps of paper to represent the disks. There can be any number of disks, but there must be exactly three pegs. Four disks is a good number to practise with.

Write a Python function **hanoi(n)** which takes a non-negative integer n, and then solves the Towers of Hanoi puzzle recursively, using the algorithm given below. For example, if there are two disks, hanoi(2), your program should return a list of strings (representing disk moves): ['Move disk 1 from peg A to peg B', 'Move disk 2 from peg A to peg C', 'Move disk 1 from peg B to peg C']

If there are zero disks to be moved, do nothing. To move $N > 0$ disks from peg X to peg Z via spare peg Y, then:

- move n-1 disks from X to Y, with Z playing the role of spare
- move disk n from X to Z
- move n-1 disks from Y to Z, with X playing the role of spare