TD08_solutions

December 9, 2014

1 TD08 - Exercises around fasta-formatted sequences files

1.1 Reading a fasta file

Exercise Write a function that takes as input the name of a fasta file, reads its content, display the number of genes anr returns a discionary associating each gene ID to its sequence.

1.1.1 Defining a function to read a sequence file

Fasta files are structured as follows:

- Rows starting with a '>' symbol are called "header" rows. The header includes the identifier of the next sequence (defined as all the text following the '>' until the first spacing or the newline character '\n'), and may optionally contain additional comments (any text located after the first space of the header is considered as comment text). There can be only one sequence header per sequence.
- The other rows correspond to the sequences themselves. A same sequence can come on several successive rows.

A fasta file can contain one or several sequences: each time a new header line is found, we will thus need to initialize a new sequence. We will then concatenate all the following rows to get the sequence, until we encounter a new header line or the end of the file.

A slight difficulty will be to extract sequence identifiers from the header rows. We will need to strip out the leading symbol ('>'), the newline character at the end of the row ('\n'), and to ignore the comments. Thus, we need to select all the text from the leading '>' up to the first spacing character.

1.1.2 Solution 1: fiddling around to select the ID

```
In [15]: def read_fasta_file_1(file_name):
             """
             This function reads a sequence file in fasta format, and returns a
             dictionnary with one entry per sequence (key=ID, value = sequence).

             Args:
                 file_name: name of the input file. This file is supposed to be formated in fasta.

             Return:
                 seq_dict: a dictionary with sequence IDs as keys, and sequences as values

             Examples:
                 plasmoCodingGenes = read_fasta_file_1("PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa")
             """
             myFile = open(file_name,"r")
             print("Reading sequences from file\n", "\t", file_name)
             geneName = "" ## initialize an embty geneName variable
```

```
    seq_dict = {} ## initialize an empty dictionary to store all sequences
    for line in myFile: ## Successively read each line of the input file
        if line.startswith(">"): ## Test if the current line is a fasta header line
            if not " " in line : ## If there is no space in the header line, everything that f
                geneName = line[1:-1] ## Ignore the leading ">", and the trailing letter, whic
            else:
                geneName = line[1:line.index(" ")] # Ignore all text following the space (fast
            seq_dict[geneName] = ""; ## Create an empty entry in the dictionary for the curren
        else: ## Treat other lines than the fasta header lines, i.e. sequences
            seq_dict[geneName]+=line[:-1].upper() ## convert sequence to uppercases, and appen
    myFile.close() ## Close the input file after having read all lines
    print("Number of sequences\t", len(seq_dict.keys())) ## Report the number of sequences rea
    return seq_dict ## Return the result (dictionary of sequences)
```

Note that the function starts with some comments explaining what it does, and describing the arguments and return type. This documentation is quite simple to formulate, and is extremely useful to help users.

As soon as the function is documented in this way, it is equipped with an on-line help message, as shown below.

```
In [16]: help(read_fasta_file_1)
```

```
Help on function read_fasta_file_1 in module __main__:

read_fasta_file_1(file_name)
    This function reads a sequence file in fasta format, and returns a
    dictionnary with one entry per sequence (key=ID, value = sequence).

    Args:
        file_name: name of the input file. This file is supposed to be formated in fasta.

    Return:
        seq_dict: a dictionary with sequence IDs as keys, and sequences as values

    Examples:
        plasmoCodingGenes = read_fasta_file("PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa")
```

1.1.3 Handling directories and listing files

Before testing our method to read a sequence file, we must know where this file is stored on our computer. The code below shows some useful functions fro the *os* library (*os* stands for *operating system*), which will allow us to specify directories, and list the files they contain. The result should vary from user to user, according to yourlocal configuration.

For this exercise, we will assume that you already created in the root of your accound: - a directory named "jgb53d-bd-prog", for the content of this course - a sub-directory "jgb53d-bd-prog/data" for all the data - a sub-directory "jgb53d-bd-prog/data/sequences" do store sequences

You should also have downloaded the two sequence files used in this tutorial, containing all the nucleotidic coding sequences for the two following organisms: - the enterobacteria *Escherichia coli* (fasta file Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa) - the vector of malaria, *Plasmodium falciparum* (fasta file PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa)

```
In [17]: ## Before doing the analysis, we will check our current working directory
         ## The result should differ for each person, depending on the configuration of your computer.

         import os
```

```
## Print the current working directory
print("Current working directory = " + os.getcwd())

## Get home directory
home_dir = os.path.expanduser('~')
print("Home directory = " + home_dir)

## Specify the data directory
course_dir = os.path.join(home_dir, "jgb53d-bd-prog")
seq_dir = os.path.join(course_dir, "data", "sequences")
print("Sequence directory = " + seq_dir)

## List all files in the data directory
seq_files = os.listdir(seq_dir)
print(seq_files)

## Specify the full path of the organism-specific sequence files
ecoli_seq_file = os.path.join(seq_dir, "Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa")
print("E.coli CDS file : " +ecoli_seq_file)
plasmodium_seq_file = os.path.join(seq_dir, "PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa")
print("P.falciparum CDS file: " + plasmodium_seq_file)
```

```
Current working directory = /Users/jvanheld/Documents/enseignement/bioinformatics_courses/jgb53d-bd-prog
Home directory = /Users/jvanheld
Sequence directory = /Users/jvanheld/jgb53d-bd-prog/data/sequences
['Escherichia_coli_aa.fa', 'Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa', 'PlasmoDB-12.0_Pfalciparum
E.coli CDS file : /Users/jvanheld/jgb53d-bd-prog/data/sequences/Escherichia_coli_str_K-12_substr_MG1655.P
P.falciparum CDS file: /Users/jvanheld/jgb53d-bd-prog/data/sequences/PlasmoDB-12.0_Pfalciparum3D7_Annota
```

1.1.4 Test: reading CDS files

We can now use the function **read_fasta_file_1()** defined above to read all coding sequences from *Escherichia coli*.

In [18]:
```
## Read all sequences from the Escherichia coli file
print("\nReading all CDS for Escherichia coli")
coli_cds = read_fasta_file_1(ecoli_seq_file)

print("\nReading all CDS for Plasmodium falciparum\n")
plasmodium_cds = read_fasta_file_1(plasmodium_seq_file)
```

```
Reading all CDS for Escherichia coli
Reading sequences from file
     /Users/jvanheld/jgb53d-bd-prog/data/sequences/Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa
Number of sequences         4549

Reading all CDS for Plasmodium falciparum

Reading sequences from file
     /Users/jvanheld/jgb53d-bd-prog/data/sequences/PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa
Number of sequences         5542
```

1.1.5 Second solution: using string methods to extrat sequence IDs

We will now develop a cleaner solution , where we use the string methods lstrip(), rstrip(), split() to extract the gene ID from the sequence header.

```
In [19]: def read_fasta_file_2(file_name):
             """
             This function reads a sequence file in fasta format, and returns a
             dictionnary with one entry per sequence (key=ID, value = sequence).

             Args:
                 file_name: name of the input file. This file is supposed to be formated in fasta.

             Return:
                 seq_dict: a dictionary with sequence IDs as keys, and sequences as values

             Examples:
                 plasmoCodingGenes = read_fasta_file_2("PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa")
             """
             print("Reading sequences from file\n", "\t", file_name)
             my_file = open(file_name,"r") ## Open the sequence file in read mode
             seq_dict = dict()          ## create the dictionnary that will contain the result
             for line in my_file:           ## Iterate over the lines of the sequence file
                 if  line.startswith(">") :    ## Detect sequence headers (lines starting with ">")
                   line = line.lstrip(">") ## Remove the leading ">"
                   line = line.lstrip(" ") ## Remove everything that follows the first space (comments)
                   tab = line.split("|")   ## Split the sequence descriptor
                   gene_id = tab[0]        ## The first element of the splitted list contains the gene
                   line = line.rstrip(" ")
                   seq_dict[gene_id] = ""   ## Initialize the nucleotidic sequence
                 else:                         ## Treatment for lines that do not correspond to a heade
                   line = line.rstrip("\n")  ## Remove the carriage return at the end of the line
                   seq_dict[gene_id] += line.upper() ## Convert the sequence to uppercase, and append i

             ## Close the input file
             my_file.close()

             print("Number of sequences\t" , len(seq_dict.keys()))
             return seq_dict
```

We can check that this second solution gives the same result as the previous one.

```
In [20]: ## Read all sequences from the Escherichia coli file
         print("\nReading all CDS for Escherichia coli")
         coli_cds = read_fasta_file_2(ecoli_seq_file)

         print("\nReading all CDS for Plasmodium falciparum\n")
         plasmodium_cds = read_fasta_file_2(plasmodium_seq_file)
```

```
Reading all CDS for Escherichia coli
Reading sequences from file
          /Users/jvanheld/jgb53d-bd-prog/data/sequences/Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa
Number of sequences         4549

Reading all CDS for Plasmodium falciparum

Reading sequences from file
          /Users/jvanheld/jgb53d-bd-prog/data/sequences/PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa
Number of sequences         5542
```

1.1.6 Solution 3: Reading a fasta file with regular expression

In [20]:
```
import re ## Import the regular expression (re) library

def read_fasta_file(file_name):
    """
    This function reads a sequence file in fasta format, and returns a
    dictionnary with one entry per sequence (key=ID, value = sequence).

    Args:
        file_name: name of the input file. This file is supposed to be formated in fasta.

    Return:
        seq_dict: a dictionary with sequence IDs as keys, and sequences as values

    Examples:
        plasmoCodingGenes = read_fasta_file("PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa")

    """
    print("Reading sequences from file\n", "\t", file_name)
    my_file = open(file_name,"r") ## Open the fasta sequence file in read mode
    seq_dict = dict()         ## Initialize the dictionary which will contain the sequences

    for line in my_file:
        # The following regular expression searches strings
        # - starting ("^") by the character ">",
        # - potentially followed by a space " "
        # - followed by at least one non-space characters ("[^ ]+")
        # The parentheses allow to capture the relevant part of the motif.
        re_result = re.search("^> *([^ \|]+)", line) # This regExp can still be improved
        if   re_result:                       ## We identified a new sequence header
              gene_name = re_result.group(1)  ## Extract the gene name
              seq_dict[gene_name] = ""    ## Initialize the dictionary entry for this gene nam
        else:
              line = line.rstrip("\n")        ## Strip the trailing newline character
              seq_dict[gene_name] += line.upper()  ## Append the sequence to the previous one
    my_file.close()

    print("Number of sequences\t" , len(seq_dict.keys()))
    return seq_dict
```

In [25]:
```
## Read all sequences from the Escherichia coli file
print("\nReading all CDS for Escherichia coli")
coli_cds = read_fasta_file(ecoli_seq_file)

print("\nReading all CDS for Plasmodium falciparum\n")
plasmodium_cds = read_fasta_file(plasmodium_seq_file)
```

```
Reading all CDS for Escherichia coli
Reading sequences from file
          /Users/jvanheld/jgb53d-bd-prog/data/sequences/Escherichia_coli_str_K-12_substr_MG1655.PATRIC.fa
Number of sequences         4549

Reading all CDS for Plasmodium falciparum
```

```
Reading sequences from file
          /Users/jvanheld/jgb53d-bd-prog/data/sequences/PlasmoDB-12.0_Pfalciparum3D7_AnnotatedCDSs.fa
Number of sequences         5542
```

1.2 Reverse complementary sequence

Write a function that computes the reverse complement of a sequence. Check that, after applying it to some sequence, the original sequence has not been modified (we don't want to loose the forward sequence).

In [21]:
```
def reverseComplement(seq):
    """Compute the reverse complement of a nucleotiduc sequence.
    The current version only supports non-degenerated nucleotidic sequence.
    This function could easily be extended to support IUPAC code.

    Args:
        seq -- input sequence (string)
    Returns:
        rc -- a string containing the reverse complementary sequence
    """

    # The simplest way to reverse a sequence: read it with an iteration of -1
    rev = seq[::-1]

    # Compute a "translation table" in the python sense.
    # Note: "translation" has a more general meaning here than in biology.
    transtab = str.maketrans("aAcCgGtT","tTgGcCaA")
    rev_cpl = rev.translate(transtab)
    return rev_cpl
```

In [22]:
```
# Create a test sequence.
# Note the presence of lower-and upper-cases
testSeq = "aaaaTTttGCGCgcg"
print("Original sequence           ", "\t", testSeq)

## Compute the reverse complement
rc = reverseComplement(testSeq) ## Compute the reverse complement
print("Forward sequence (unchanged)", "\t", testSeq) ## Check that the function has no side ef
print("Reverse complementary seq.  ", "\t", rc)      ## Check the result

## Yet another control: compute the reverse
## complement of the reverse complement.
rc_and_back = reverseComplement(rc)
print("Forward sequence (unchanged)", "\t", testSeq) ## Check that the function has no side ef
print("Rev.compl. of rev.compl.  ", "\t", rc_and_back)      ## Check the result

# We can check formally that the reverse complement of the reverse
# complement equals the original sequence
print ("rc(rc(seq)) == seq ?\t", rc_and_back == testSeq)
```

```
Original sequence                    aaaaTTttGCGCgcg
Forward sequence (unchanged)         aaaaTTttGCGCgcg
Reverse complementary seq.           cgcGCGCaaAAtttt
Forward sequence (unchanged)         aaaaTTttGCGCgcg
Rev.compl. of rev.compl.            aaaaTTttGCGCgcg
rc(rc(seq)) == seq ?        True
```

1.3 Translating nucleotidic into peptidic sequences

1.3.1 Exercise

Write a function that translates nucleotidic sequences into peptidic sequences, using the codon dictionary (from codons to aminoacids) defined below. Use then your function to generate a fasta file with all the proteins of the studied organisms.

1.3.2 Approach

We will create a dictionary containing the genetic code (i.e. the correspondence between codons and aminoacids). Once this dictionary is defined, translation can be done simply by iterating over the sequence bysteps of 3, and getting the aminoacid (value of the dictionary) associated to each trinucleotide (key of the dictionary).

```
In [25]: def translate_seq(nt_seq):
             """
             Translates one nucleic sequence into the corresponding peptidic
             sequence, using the canonical code of living organisms.

             Args:
                 nt_seq -- a nucleic sequence (string)

             Returns:
                 aa_seq -- a peptidic sequence (string)

             Example:
                 pep_seq = translate_seq("ATGTTTTATGTACATTAA")
             """



             ## Before anything else, check that the input sequence has a correct length (must be a mul
             if (len(nt_seq) %3):
                 raise Exception('translate_seq() : the length of input seq must be a multiple of 3.')

             ####################################
             ##  Translation dictionary from codons (triplets of nucleotides) to aminoacids
             ####################################
             codon2AA = {}
             codon2AA["ATT"]="I";codon2AA["ATC"]="I";codon2AA["ATA"]="I"
             codon2AA["CTT"]="L";codon2AA["CTC"]="L";codon2AA["CTA"]="L";codon2AA["CTG"]="L";codon2AA["
             codon2AA["GTT"]="V";codon2AA["GTC"]="V";codon2AA["GTA"]="V";codon2AA["GTG"]="V"
             codon2AA["TTT"]="F";codon2AA["TTC"]="F"
             codon2AA["ATG"]="M"
             codon2AA["TGT"]="C";codon2AA["TGC"]="C"
             codon2AA["GCT"]="A";codon2AA["GCC"]="A";codon2AA["GCA"]="A";codon2AA["GCG"]="A"
             codon2AA["GGT"]="G";codon2AA["GGC"]="G";codon2AA["GGA"]="G";codon2AA["GGG"]="G"
             codon2AA["CCT"]="P";codon2AA["CCC"]="P";codon2AA["CCA"]="P";codon2AA["CCG"]="P"
             codon2AA["ACT"]="T";codon2AA["ACC"]="T";codon2AA["ACA"]="T";codon2AA["ACG"]="T"
             codon2AA["TCT"]="S";codon2AA["TCC"]="S";codon2AA["TCA"]="S";codon2AA["TCG"]="S";codon2AA["
             codon2AA["TAT"]="Y";codon2AA["TAC"]="Y"
             codon2AA["TGG"]="W"
             codon2AA["CAA"]="Q";codon2AA["CAG"]="Q"
             codon2AA["AAT"]="N";codon2AA["AAC"]="N"
             codon2AA["CAT"]="H";codon2AA["CAC"]="H"
             codon2AA["GAA"]="E";codon2AA["GAG"]="E"
```

```
    codon2AA["GAT"]="D";codon2AA["GAC"]="D"
    codon2AA["AAA"]="K";codon2AA["AAG"]="K"
    codon2AA["CGT"]="R";codon2AA["CGC"]="R";codon2AA["CGA"]="R";codon2AA["CGG"]="R";codon2AA["
    codon2AA["TAA"]="";codon2AA["TAG"]="";codon2AA["TGA"]=""

    aa_seq = ""
    for i in range(0,len(nt_seq),3):
        aa_seq += codon2AA[ nt_seq[i:i+3] ]
    return aa_seq
```

We can do a quick test of the method *translate_seq(nt_seq)* with a small sequence.

```
In [28]: translate_seq("ATGTTTTATGTACATTAA")
```

```
Out[28]: 'MFYVH'
```

We can now implement another method that will translate all the sequences of the sequence dictionary returned by our method *read_fasta_file()*.

```
In [29]: def translate_seq_dict (nt_seq_dict):
             """
             Translate a set of nucleic into the corresponding peptidic sequences,
             using the canonical code of living organisms.

             Args:
                 nt_seq_dict -- a dictionary of nucleic sequences (key=name or ID, value=sequence)
             Returns:
                 aa_seq_dict -- a dictionary of peptidic sequence obtained by translating each individu
             """
             aa_seq_dict = dict()
             for name,nt_seq in nt_seq_dict.items():
                 aa_seq_dict[name] = translate_seq(nt_seq)
             return(aa_seq_dict)
```

We can now test the method on a subset of the CDS of *Escherichia coli*. For this, we create a small dictionary with the 2 first items of the sequence dictionary defined above.

```
In [40]: coli_cds_subset = {}

         for key in (sorted(coli_cds.keys())[:2]):
             coli_cds_subset[key] = coli_cds[key]

         translate_seq_dict(coli_cds_subset)
```

```
Out[40]: {'32115101': 'MVKVYAPASSANMSVGFDVLGAAVTPVDGALLGDVVTVEAAETFSLNNLGRFADKLPSEPRENIVYQCWERFCQELGKQI
          '32115099': 'VLKFGGTSVANAERFLRVADILESNARQGQVATVLSAPAKITNHLVAMIEKTISGQDALPNISDAERIFAELLTGLAAAQ
```

We will now implement a method that writes a sequence dictionary into a fasta file.

```
In [41]: def write_seq(seq_dict, file_name):
             """
             Write a set of sequences in a fasta-formatted file.

             Args:
                 seq_dict -- a dictionary of sequences (key: ID or name, value=sequence)
                 file_name -- name of the output file
```

```
        Returns:
            no value
        """
        print ("Writing ",len(seq_dict)," sequences in file", file_name)
        outfile = open(file_name,"w") ## Open output file in write mode
        for name,sequence in seq_dict.items():
            outfile.write(">" + name +"\n")
            outfile.write(sequence+"\n")
        outfile.close()
```

In [45]:
```
coli_peptides = translate_seq_dict(coli_cds)
print("Number of peptidic sequences\t", len(coli_peptides))
write_seq(coli_peptides, os.path.join(seq_dir, "Escherichia_coli_aa.fa"))
```

```
Number of peptidic sequences         4544
Writing  4544  sequences in file /Users/jvanheld/jgb53d-bd-prog/data/sequences/Escherichia_coli_aa.fa
```

In [46]:
```
plasmodium_peptides = translate_seq_dict(plasmodium_cds)
print("Number of peptidic sequences\t", len(plasmodium_peptides))
write_seq(plasmodium_peptides, os.path.join(seq_dir, "Plasmodium_falciparum_aa.fa"))
```

```
Number of peptidic sequences         5542
Writing  5542  sequences in file /Users/jvanheld/jgb53d-bd-prog/data/sequences/Plasmodium_falciparum_aa.
```

In [47]:
```
def seq_stats(seq_dict):
    """
    Compute summary statistics on a set of sequences (can be nucleic or peptidic).

    Args:
        seq_dict -- a dictionary of sequences (key=ID or name, value=sequence)
    Returns:
        stats -- a dictionary with statistics computed on the sequences
    """
    stats = dict()
    stats["nb"] = 0
    stats["len_sum"] = 0
    stats["len_mean"] = 0
    for name,sequence in seq_dict.items():
        stats["nb"] += 1
        stats["len_sum"] += len(sequence)
    if stats["nb"] > 0:
        stats["len_mean"] = stats["len_sum"] / stats["nb"]
    return stats
```

In [56]:
```
print( "E.coli CDS stats")
print(seq_stats(coli_cds))

print( "E.coli Protein stats")
print(seq_stats(coli_peptides))

print( "P.falciparum CDS stats")
print(seq_stats(plasmodium_cds))

print( "P.falcimarum Protein stats")
print(seq_stats(plasmodium_peptides))
```

```
E.coli CDS stats
{'len_sum': 4075029, 'len_mean': 895.8076500329743, 'nb': 4549}
E.coli Protein stats
{'len_sum': 1353791, 'len_mean': 297.6018905253902, 'nb': 4549}
P.falciparum CDS stats
{'len_sum': 12583182, 'len_mean': 2270.512811259473, 'nb': 5542}
P.falcimarum Protein stats
{'len_sum': 4188154, 'len_mean': 755.7116564417178, 'nb': 5542}
```

1.4 Computation of nucleotide frequencies

```
In [48]: # -*- coding: utf8 -*-
         def nt_composition(codingGenes):
             nbA, nbT, nbG, nbC, seq_len = 0, 0, 0, 0, 0 ## Initialize the counters
             for gene,sequence in codingGenes.items():
                 nbA += sequence.count("A") ## Increment the counter for A
                 nbT += sequence.count("G") ## Increment the counter for C
                 nbG += sequence.count("G") ## Increment the counter for G
                 nbC += sequence.count("T") ## Increment the counter for T
                 seq_len += len(sequence)

             tot = nbA + nbT + nbG + nbC

             if( tot == seq_len ):
                 print("le compte est bon!")
             else:
                 print("Problème! Le total (",tot,") ne correspond pas à la somme des longueurs de la s
             return {"A" : nbA/tot, "T" : nbT/tot, "G" : nbG/tot, "C" : nbC/tot}

         plasmoCompoNT = nt_composition(plasmodium_cds)
         print("Plasmo compo ", plasmoCompoNT)

         ecoliCompoNT = nt_composition(coli_cds)
         print("E.coli compo ", ecoliCompoNT)
```

```
Problème! Le total ( 13110179 ) ne correspond pas à la somme des longueurs de la sequence ( 12583182 )
Plasmo compo  {'T': 0.13419107397389463, 'A': 0.43315487912102496, 'C': 0.29846297293118573, 'G': 0.134
Problème! Le total (

4182337 ) ne correspond pas à la somme des longueurs de la sequence ( 4067802 )
E.coli compo  {'T': 0.2658800570111878, 'A': 0.2341853370495969, 'C': 0.23405454892802757, 'G': 0.26588
```

```
In [49]: import random
         basesNT = ("A", "T", "G", "C")
         def seqAlea(compoNT, length):
             retSeq = ""
             for i in range(length):
                 r = random.random()
                 for base in basesNT:
                     if r < compoNT[base] :
                         retSeq += base
                         break;
                     else:
                          r -= compoNT[base]
             return retSeq
```

```
In [50]: print("Target composition", plasmoCompoNT)

         rand_seq = dict() ## Instantiate a dictionary to compute random sequences
         rand_seq[1] = seqAlea(plasmoCompoNT,1000)
         print(rand_seq)

         ## Check that the nucleotide composition of the random seq fits our expectation
         print("Random seq compos", nt_composition(rand_seq))
```

```
Target composition {'T': 0.13419107397389463, 'A': 0.43315487912102496, 'C': 0.29846297293118573, 'G': 0
{1: 'TACCTCCCCAGCGAACAAAACTACGAAGGAACATGAACGGAACTCAACAAATACACGCAACCCACCAAAATATTCAGCTACTCCAGACATGGGGACAA
Problème! Le total ( 828 ) ne correspond pas à la somme des longueurs de la sequence ( 1000 )
Random seq compos {'T': 0.15821256038647344, 'A': 0.5120772946859904, 'C': 0.17149758454106281, 'G': 0.1
```

2 Tests réalisés en séance avec le groupe 2

```
In [56]: seq = "ATTGCGGG"
         print(seq)
```

```
ATTGCGGG
```

```
In [57]: ## Instantiate an empty sequence for the reverse
         rev = ""
         print("Original sequence = ", seq)
         for i in range(len(seq)):
             rev = rev + seq[-1-i]
             print(i, seq[-1-i], rev)
         print("Reverse sequence = ", rev)
```

```
Original sequence =  ATTGCGGG
0 G G
1 G GG
2 G GGG
3 C GGGC
4 G GGGCG
5 T GGGCGT
6 T GGGCGTT
7 A GGGCGTTA
Reverse sequence =  GGGCGTTA
```

```
In [58]: ## Reminder about slice handling
         ## Rappel: manipulation de tranches dans des chaînes de caractères
         print(seq)
         print(seq[3:5])
```

```
ATTGCGGG
GC
```

```
In [59]: print(seq[0:6:2])
```

```
ATC
```

```
In [60]: print(seq[0::2])
```

```
ATCG
```

```
In [61]: print(seq[::2])
```

```
ATCG
```

```
In [62]: seq[::-1]
```

```
Out[62]: 'GGGCGTTA'
```

```
In [98]: seq[::]
```

```
Out[98]: 'ATTGCGGG'
```

2.1 Une méthode particulièrement lourde pour calculer le complément

```
In [63]: cpl = "" ## initialiser la variable qui contiendra la séq complémentaire

         cpl_dict = {"A":"T", "T":"A", "G":"C", "C":"G"}
         print(cpl_dict)

         for i in range(len(seq)):
             nt = seq[i]
             cpl_nt = cpl_dict[nt]
             cpl = cpl + cpl_nt
             print(i, nt, cpl, cpl_nt)
```

```
{'T': 'A', 'A': 'T', 'C': 'G', 'G': 'C'}
0 A T T
1 T TA A
2 T TAA A
3 G TAAC C
4 C TAACG G
5 G TAACGC C
6 G TAACGCC C
7 G TAACGCCC C
```

```
In [65]: seq = "ATTGCGGG"
         print(seq)

         cpl = "" ## initialiser la variable qui contiendra la séq complémentaire

         cpl_dict = {"A":"T", "T":"A", "G":"C", "C":"G"}
         print(cpl_dict)

         ## In Python, strings can directly be iterated
         for nt in seq:
             cpl += cpl_dict[nt]
             print(nt, cpl)
         print("Original sequence: ", seq)
         print("Complementary seq: ",cpl)
```

```
ATTGCGGG
{'T': 'A', 'A': 'T', 'C': 'G', 'G': 'C'}
A T
T TA
T TAA
G TAAC
```

```
C TAACG
G TAACGC
G TAACGCC
G TAACGCCC
Original sequence:  ATTGCGGG
Complementary seq:  TAACGCCC
```