Practical 3: Sequence manipulation with Python control flow

Contents

The aim of this practical is to apply Python conditional statements and loops in bioinformatic context. This practical is based on the Lecture 4.

Task 1: Counting bases in a DNA

You already know how to generate a DNA string by concatenating two DNA fragments or by using the `random` Python module. Now we will count how many times each DNA base (`'A', 'T', 'G', 'C'`) occurs in a sequence.

Make a new project, call it, for instance, ***practical_03***. Add a new Python script or Jupyter notebook to the project, call it ***count_nucleotides.py / ipynb*** .

Start file with some brief information about the code purpose and the author, as required for the good programming practice, e.g.:

```
# Counting the occurrence of nucleotides in a DNA
# Author, date
```

Use code from the ***random_dna.py / ipynb*** to make a DNA *string* named **dna**.

Then, initialise counters for each individual base:

```
# Initialise counters for each base
A = 0
T = 0
G = 0
C = 0
unknown = 0 # in case of something outside A, T, G, C
```

We will loop through each element of the sequence and check whether it is either `'A'` or `'T'` or `'G'` or `'C'`; if not, then the base is not known. If we find the base, then we increment the related counter.

```
# Loop over DNA sequence
for base in dna3:
     if(base == 'A'):
          A = A + 1
     elif(base == 'T'):
          T = T + 1
     elif(base == 'G'):
          G = G + 1
     elif(base == 'C'):
          C = C + 1
     else:
          unknown = unknown + 1
```

Lastly, we will print the results to show the numbers of bases:

```
# Printing the result
print ("Adenine: ", A)
print ("Thymine: ", T)
print ("Guanine: ", G)
print ("Cytosine: ", C)
print ("Unknown bases:", unknown)
```

Q: Can you do it another way?

Exercise 3.1. *Write a new script that will calculate the percentage of each nucleotide in the sequence. The output should be a float number and needs to be presented for each base in the following format:*

The percentage of A = 24.700%

Do you see any problem with such output? If yes, then format your output to two decimal places. [Hint: use any of the string formatting methods that you learnt in Lecture 3]

Task 2: Calculating GC content

After you know how to count the bases, let's calculate the *GC content* of a sequence. GC content is important for thermodynamic properties of DNA (such as annealing). It is calculated using sliding window: as the count of G's and C's over some window (e.g. 100 nucleotides), which moves along DNA sequence, with a certain sliding step at a time.

For example:

AGTGCTGCTGCTACGTACGGGCA
AGTGCTGCTGCTACGTACGGGCA
AGTGCTGCTGCTACGTACGGGCA
AGTGCTGCTGCTACGTACGGGCA

$$GC\ content\ per\ each\ window\ = \frac{N_C + N_G}{N_A + N_G + N_C + N_T}$$

In this example the window width is 8 (it includes 8 nucleotides) and the sliding step is 4.

In the first window there are 3 G's and 2 C's, and so the GC content = 5/8.

The next window has moved by 4 bases to the right; the computation is repeated, and it also produces GC content = 5/8. In the third step GC content = 4/8, and in the last step GC content = 5/8.

Of course, in real bioinformatics task the window and the step would be much larger.

Now let's calculate GC content for the random DNA string, which you generated earlier.

Make a new Python file, call it ***gc_content.py / ipynb*** .

Start the file with making a random DNA sequence of 1,000bp:

```
dna = ?????
```

Here, replace "?????" with your random DNA sequence.

Let's set window = 100 and step = 5

```
window_size = 100
step = 5
```

Make an empty ***list*** for GC contents, wich you will calculate per each window:

```
gc_contents = []
```

Loop over the DNA string, using the given step:

```
for i in range(0, len(dna), step ):
```

Get the sliding window of required size (pay attention to the indentation !) :

```
    window = dna[i : i + window_size]
```

Count 'A', 'T', 'G', 'C' in the window:

```
    a = window.count('A')
    g = window.count('G')
    c = window.count('C')
    t = window.count('T')
```

Calculate GC content in the window:

```
    gc = g + c
    total = a + g + c + t

    if total > 0: # Make sure that the window is not empty
        ratio = gc / total
        gc_contents.append(ratio)
```

After all the windows have been processed, print the result:

```
print(gc_contents)
```

Q: Why the condition `total > 0` **has been used to calculate the ratio? What type of error might occur if the condition has not been used?**

Q: Do the size of sliding window and step size have any effect on GC contents?

Exercise 3.2. *Use the following DNA sequence (500 nucleotides) to calculate GC content with sliding window = 100 and step size = 50*

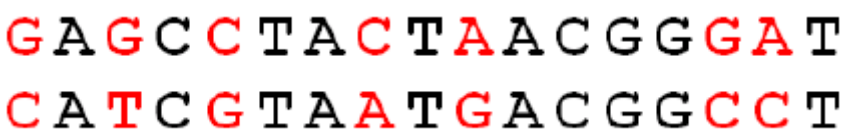
'CAAAAAAGTAAAGCCTATAACTACAAAGAGCGGTTCAAAGTCCTAACTATTAGATGCAC
GATGTGTCACGCGCCACTCCGGATCCAACCGTGATGTGCAACTAGATAGGCAGACTA
CTCTGTCTTGAGTCCGTTATTAGTAAATTACCGTCGGCGTAGGCCGACGTGCAGCTCT
CGAACCGCCTCCTGGGTTCCACAAGCATGGGGGGTGATGGGCGAATGGTACAATTGG
CAGGCTGAAGTTCCTATTTCGTCAGAGCAAGTTGCCAACAGCGTCTAGGCTAATTTTG
ACGGGCCGTGAGGAATGTGGGGTCTACTTCGTGCCCCCGACTACCATGTAGCGGCTC
CGCACAATTGTCGTGATGAAAGCGGGCTCCAGGGTTTCCCGGCCGCGTGTTCTATGT
CCACAAATCCCTGGCAGATTTCCTTGACGTGCCGGTACTCTAGTGTCTTTTGCAGTTAA
GCTACCCCGTCGCAAGACTCCTACCACTTAGTTACATG'

*The **coding** regions of this DNA sequence are: [1,150] and [401,500]. Calculate GC content in coding and **non-coding** regions separately.*

Do you observe any differences between GC contents in coding and non-coding regions?

Task 3: Hamming distance between two sequences

The ***Hamming distance*** between two strings of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other. For example:



The Hamming distance between these two strings is 7. Mismatched symbols are coloured **RED**.

Make a new file, call it ***hamming_distance.py / ipynb*** .

Calculating Hamming distance is really easy.

Let's use the two sequences shown above:

```
sequence1 = "GAGCCTACTAACGGGAT"
sequence2 = "CATCGTAATGACGGCCT"
```

Now we will start with a distance of zero, and count the mismatched positions.

```
distance = 0
```

Loop over each position of the strings:

```
for i in range(len(sequence1)):
```

The `range(len(sequence1))` means range from 0 to the length of Sequence 1.

Range is a commonly used "***iterable***" object in Python. Previously, you have seen the syntax `range(start,end)` or `range(start,end,step)`. However, if the ***start*** is omitted, it is assumed to be zero.

Add 1 to the ***distance*** if the bases in two strings are not equal (pay attention to indentation):

```
    if(sequence1[i] != sequence2[i]):
        distance += 1
```

Print the result

```
print ("The hamming distance: ",distance)
```

Exercise 3.3. *Input two sequences from command line, calculate the Hamming distance and print the alignment as follows (this is a standard output of many alignment tools) :*

```
GAGCCTACTAACGGGAT
 | | ||  | ||||  |
CATCGTAATGACGGCCT
```

Additionally, provide a check for unequal sequence lengths.

Well done! You have finished Practical 3.