Introduction to Python
Python II

1. Getting started

Before you start with the following assignments, please create a directory PythonLab2 in your home directory (~/). In this directory save all the files you create during this assignment.

2. Basic exercises

Remember to load the module for Python 3 with the command module load python3/3.6.0 if you have reconnected to UPPMAX since before the lunch break.

2.1 Similarity of sequences

Write in an editor the program, which calculates the distance between two sequences,

```python
# define sequences
seq1 = "ACGT"
seq2 = "AGGT"

# initiate counter
distance_score = 0

# for each letter in the sequences
for i in range(len(seq1)):
    # if they don't match, add a distance point
    if (seq1[i] != seq2[i]):
        distance_score += 1

# print result to the user
print("Distance between A and B: ", distance_score)
```
1. Calculate the distance between the following sequences and print out the result. Since the following sequences are already aligned, we can calculate the distance between them. Change your program so that it can read two aligned sequences from the command line. Test your program with the following sequences.

a) ACGT and A-GT
b) AC-GT and AGT--
c) AC-CGT and AGT---
d) ACCGT and TGCCA
e) GATT-ACA and TACCATA
f) --GA--TT--AC and TA--CC--AT--CA

```python
import sys

# read sequences from command line arguments
seq1 = sys.argv[1]
seq2 = sys.argv[2]

# initiate counter
distance_score = 0

# for each letter in the sequences
for i in range(len(seq1)):
    # if they don't match, add a distance point
    if (seq1[i] != seq2[i]):
        distance_score += 1

# print result to user
print("Distance between seq1 and seq2: ", distance_score)
```

Distance between seq1 and seq2: 1
Distance between seq1 and seq2: 4
Distance between seq1 and seq2: 5
Distance between seq1 and seq2: 4
Distance between seq1 and seq2: 7
Distance between seq1 and seq2: 13
2. Extend the program that the aligned sequences are printed out additionally to their distance.

```python
print("Sequence seq1: ", seq1)
print("Sequence seq2: ", seq2)
print("Distance between seq1 and seq2: ", distance_score)
```

Sequence seq1: ACGT
Sequence seq2: A-GT
Distance between seq1 and seq2: 1

Sequence seq1: AC-GT
Sequence seq2: AGT--
Distance between seq1 and seq2: 4

# and so on

3. Extend the program that the distance between two sequences is only calculated when both sequences have the same length. Test your program with the input sequences:

a) ACGT and AGT
b) ACCGT and TGCCA

```python
import sys

# read sequences from command line arguments
seq1 = sys.argv[1]
seq2 = sys.argv[2]

# check if the sequence lengths match
if (len(seq1) == len(seq2)):
    # initiate the counter
distance_score = 0

    # for each letter in the sequences
    for i in range(len(seq1)):
        # if they don't match, add a distance point
        if (seq1[i] != seq2[i]):
            distance_score += 1

    # print the result
    print("Distance between seq1 and seq2: ", distance_score)
```

Sequences seq1 and seq2 are of different length.

Sequence seq1:  ACCGT
Sequence seq2:  TGCCA
Distance between seq1 and seq2:  4

4. Extend the program so that the second sequence is inverted and assigned to a third sequence. Please, read the first and second sequence from the command line. Calculate the distances between the first and the second and between the first and the third sequence.

Compare the distance between the first and the second and the first and the third sequence and print the alignment with the smaller distance. If the distances are equal, then print the alignment of the first and second sequence.

Test your program with the following sequences:

a) ACGT and A-GT
b) AC-GT and AGT--
c) ACCGT and TGCCA
d) GATT-ACA and TACCATA

```python
import sys

# get sequences from the command line arguments
seq1 = sys.argv[1]
seq2 = sys.argv[2]

### reverse the seq2 string and save as seq2_rev

# initiate variable
seq2_rev = ""

# for each letter in seq2
for i in range(len(seq2)):
    # add the next character to the reversed seq2 string
    seq2_rev += seq2[len(seq2) - i - 1]
```
# only run the calculation if the sequences have the same length
if (len(seq1) == len(seq2)):
    # initiate counters
dist_1_2 = 0
dist_1_2rev = 0

    # for each letter in the sequences
    for i in range(len(seq1)):
        # if they don't match, add a distance point
        if (seq1[i] != seq2[i]):
            dist_1_2 += 1

        # if they don't match, add a distance point
        if (seq1[i] != seq2_rev[i]):
            dist_1_2rev += 1

    # if the distance seq1seq2 is less or eq to distance seq1seq2_rev
    if (dist_1_2 <= dist_1_2rev):
        # print the seq1seq2 sequences and distance score
        print("Sequence seq1: ", seq1)
        print("Sequence seq2: ", seq2)
        print("Distance between seq1 and seq2: ", dist_1_2)
    else:
        # print the seq1seq2_rev sequences and distance score
        print("Sequence seq1: ", seq1)
        print("Sequence seq2: ", seq2_rev)
        print("Distance between seq1 and seq2_rev: ", dist_1_2rev)

    # tell the user the lengths differ
else:
    print("Sequences seq1 and seq2 are of different length.")

Sequence seq1: ACGT
Sequence seq2: A-GT
Distance between seq1 and seq2: 1

Sequence seq1: AC-GT
Sequence seq2: AGT--
Distance between seq1 and seq2: 4

# and so on
3. Bonus exercises

3.1 Functions

Open an editor and save your new program. In this program we will create a few functions.

1. Define the two functions similarity and distance that will calculate the similarity and distance for a single pair of nucleotides:

\[
similarity(a, b) = \begin{cases} 
1, & \text{if } a = b \\
0.5, & \text{if } a \neq b, a \text{ and } b \text{ are the same kind, either purines or pyrimidines} \\
0, & \text{if } a \neq b, a \text{ and } b \text{ are not the same kind}
\end{cases}
\]

\[
distance(a, b) = \begin{cases} 
0, & \text{if } a = b \\
0.5, & \text{if } a \neq b, a \text{ and } b \text{ are the same kind, either purines or pyrimidines} \\
1, & \text{if } a \neq b, a \text{ and } b \text{ are not the same kind}
\end{cases}
\]

Note: Purines are A and G, pyrimidines are C and T.

```python
# define which bases are purines and pyrimidines
pur = ["A", "G"]
pyr = ["C", "T"]

# define the similarity function for two single bases
def similarity(base1, base2):
    # if they match, return 1
    if (base1 == base2):
        return 1

    # else, if they dont match but are of the same kind
    elif (((base1 in pur) and (base2 in pur)) or ((base1 in pyr) and (base2 in pyr))):
        return 0.5

    # if they neither matches or are of the same kind, return 0
    else:
        return 0
```
# define the distance function for two single bases

def distance(base1, base2):
    # if they match, return 0
    if (base1 == base2):
        return 0
    # else, if they don't match but are of the same kind
    elif (((base1 in pur) and (base2 in pur)) or ((base1 in pyr) and (base2 in pyr))):
        return 0.5
    # if they don't match and are of different kind, return 1
    else:
        return 1

2. Write two functions sequence_similarity and sequence_distance, which calculates the similarity and distance of two whole sequences.

# define the similarity function for whole sequences

def sequence_similarity(seq1, seq2):
    # initiate counter
    similarity_score = 0.0
    # go through all bases in seq1
    for i in range(len(seq1)):
        # calculate their similarity and add to the score
        similarity_score = similarity_score + similarity(seq1[i], seq2[i])
    # return the final score
    return similarity_score

# define the distance function for whole sequences

def sequence_distance(seq1, seq2):
    # initiate counter
    distance_score = 0.0
    # go through all bases in seq1
    for i in range(len(seq1)):
        # calculate the distance and add to the score
        distance_score = distance_score + distance(seq1[i], seq2[i])
3. Calculate the similarity and distance for the following sequences. Read these sequences from the command line and print out their similarity and distance.

a) ACGT and TGCA
b) ATAG and ACAC
c) ACGC and ATTT
d) AGTT and ACTT
e) TCGC and AGAG

```python
import sys

# read the sequences from command line arguments
seq1 = sys.argv[1]
seq2 = sys.argv[2]

# print the similarity and distance
print("Similarity: ", sequence_similarity(seq1, seq2))
print("Distance: ", sequence_distance(seq1, seq2))
```

Similarity: 0.0
Distance: 4.0

Similarity: 2.5
Distance: 1.5

# and so on

3.2 Modules

Modules are just a way to write functions in one file, which you then can reuse in programs you are writing in other files. That way, if you need to change a part of your function you only
have to change it in a single file. The alternative is to copy/paste the function into each program that needs to use it, but if you do it this way, you will have to change each copy of the function in each of the files.

As an example, instead of having a single file with this code:

```python
def my_function(a):
    b = a + 5
    return b
print( myfunction(37) )
```

you could put the function in its own file and then just import it in any program that needs to use that function.

In a file named `some_functions.py`:

```python
def my_function(a):
    b = a + 5
    return b
```

In an interactive python3 prompt, or another file you are writing your program in:

```python
import some_functions
print( some_functions.my_function(37) )
```

which will make all the functions inside `some_functions.py` available in your script. If there was another function defined inside `some_functions.py` you would simply call them by typing `some_functions.whateverNameTheyHave()` As you can see, there is nothing special about a python file that we use as a module. It contains the same kind of function definition as you use in any python file you write. The only special thing is that you will have to specify the name of the module when you call the function (i.e. `some_functions.my_function()`).

If you prefer to shorten the names a bit,

```python
from some_functions import my_function
print( my_function(37) )
```

If there are multiple functions inside `some_functions.py` you can import all using the shorter name as in the example just above of them by writing

```python
from some_functions import *
```
and call them using just their name (without the `some_functions.` in front of the function name).

In this case it’s a really short function we have defined, but imagine if it was a 500+ lines function that we need to reuse all over the place. If we were to just include a copy of the huge function in each of those files we would be in for a world of trouble (been there, done that). You will end up finding bugs in the function when you are writing in one file and update that copy of the function. Later on you will find another bug in another place in the function while you are writing in another file and then update that part of the function in that file. If you are lucky you will remember to update all the copies of the function, but eventually you will end up having 5 different versions of the function and none of them contain all the bug fixes. That day you will decide to create a module for the function and never try to have multiple copies of a function again.

As with most of the time with computers, you can’t just say the name of a file (some_functions.py) and hope the computer will magically know where that file is located. The file has to be present in the same directory as the file you are trying to do the import in. If the file is not present in that directory python will just die and say it can’t find the module you are trying to import.

As you get more advanced and start using python more regularly you might eventually start learning about how to get around the limitation of having everything in the same folder, and learn to change the PYTHONPATH environment variable in linux or even start distributing your own modules through pip or conda, but that is out of scope for this course. (It took me ~5+ years of daily python usage before i started looking at that)

1. Write a new Python file (module) called `sequence_tools.py` which contain both the two functions similarity and distance as defined previously.

```python
# define which bases are purines and pyrimidines
pur = ["A", "G"]
pyr = ["C", "T"]

# define the similarity function for two single bases
def similarity(base1, base2):
    # if they match, return 1
    if (base1 == base2):
        return 1
    # else, if they dont match but are of the same kind
    # return 0.5
    elif (((base1 in pur) and (base2 in pur)) or ((base1 in pyr) and (base2 in pyr))):
```
return 0.5
    # if they neither matches or are of the same kind, return 0
else:
    return 0

# define the distance function for two single bases
def distance(base1, base2):
    # if they match, return 0
    if (base1 == base2):
        return 0
    # else, if they don't match but are of the same kind
    # return 0.5
    elif (((base1 in pur) and (base2 in pur)) or ((base1 in pyr) and (base2 in pyr))):
        return 0.5
    # if they don't match and are of different kind, return 1
else:
    return 1

# define the similarity function for whole sequences
def sequence_similarity(seq1, seq2):
    # initiate counter
    similarity_score = 0.0
    # go through all bases in seq1
    for i in range(len(seq1)):
        # calculate their similarity and add to the score
        similarity_score = similarity_score + similarity(seq1[i], seq2[i])
    # return the final score
    return similarity_score

# define the distance function for whole sequences
def sequence_distance(seq1, seq2):
    # initiate counter
    distance_score = 0.0
    # go through all bases in seq1
    for i in range(len(seq1)):
        # calculate the distance and add to the score
        distance_score = distance_score + distance(seq1[i], seq2[i])
    # return the final score
    return distance_score
2. Write another Python file that calculates for each combination of two sequences stored in list \texttt{seq_list} the similarity and distance using the module defined previously.

```python
seq_list = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]

from sequence_tools import *

# define sequences
seq_list = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]

# loop over each sequence in seq_list
for i in range(len(seq_list)):
    # loop over the remaining sequences in seq_list
    for j in range(i+1, len(seq_list)):
        # calculate the similarity and distance
        similarity_score = sequence_similarity(seq_list[i], seq_list[j])
        distance_score = sequence_distance(seq_list[i], seq_list[j])

        # print the result for this comparison
        print(seq_list[i], seq_list[j], " Similarity: ", similarity_score, " Distance: ", distance_score)
```

```
ATCCGGT  GCGTTAC  Similarity:  2.5  Distance:  4.5
ATCCGGT  CTACTGC  Similarity:  3.5  Distance:  3.5
ATCCGGT  TTGCAGT  Similarity:  4.5  Distance:  2.5
ATCCGGT  AGTCACC  Similarity:  3.5  Distance:  3.5
GCGTTAC  CTACTGC  Similarity:  4.0  Distance:  3.0
GCGTTAC  TTGCAGT  Similarity:  3.0  Distance:  4.0
GCGTTAC  AGTCACC  Similarity:  2.0  Distance:  5.0
CTACTGC  TTGCAGT  Similarity:  4.5  Distance:  2.5
CTACTGC  AGTCACC  Similarity:  2.0  Distance:  5.0
TTGCAGT  AGTCACC  Similarity:  2.5  Distance:  4.5
```

3. Extend your program. Determine the combination of sequences with the highest similarity of all sequences stored in list \texttt{l}. Write these two sequences and the alignment into a new file, called similar_sequences.txt.
For example for two given sequences: "ATC" and "ACC" The alignment would be:

```
ATC  
| |  
ACC
```

And this alignment should be written to a new output file.

**Hint:** A line-break in Python can be made by adding `'n'` to the end of the line.

```python
from sequence_tools import *

# define sequences
seq_list = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]

# define variables
similarity_highscore = 0
best_seq1 = ""
best_seq2 = ""

# loop over each sequence in seq_list
for i in range(len(seq_list)):
    # compare the sequence to all remaining sequences in seq_list
    for j in range(i+1, len(seq_list)):
        # calculate the similarity
        similarity_score = sequence_similarity(seq_list[i], seq_list[j])

        # check if it's a new similarity highscore
        if (similarity_score > similarity_highscore):
            # if it is, save this as the new highscore
            similarity_highscore = similarity_score
            best_seq1 = seq_list[i]
            best_seq2 = seq_list[j]

    # create an empty string to add the alignment to
    alignment_matches = ""

    # go through each letter the best aligned pair
    for i in range(len(best_seq1)):
        # find places where they match
        if (best_seq1[i] == best_seq2[i]):
            alignment_matches = alignment_matches + "|"

        # and places they don't match
```
else:
    alignment_matches = alignment_matches + " "

# write the sequences and the match symbols to file
outfile = open("similar_sequences.txt", "w")
outfile.write(best_seq1 + "\n")
outfile.write(alignment_matches + "\n")
outfile.write(best_seq2 + "\n")