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,

```
A = "ACGT"
B = "AGGT"
```

A simple program (without function and modules) is sufficient.

```
A = "ACGT"
B = "AGGT"

d = 0
for i in range(len(A)):
    if (A[i] != B[i]):
        d = d + 1
    else:
        d = d + 0

print("Distance between A and B: ", d)
```

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 `TACCATAC`
f) `--GA--TT--AC-A` and `TA--CC--AT--CA`

```
import sys

A = sys.argv[1]
B = sys.argv[2]

d = 0
for i in range(len(A)):
    if (A[i] != B[i]):
        d = d + 1
    else:
        d = d + 0

print("Distance between A and B: ", d)
```

2. Extend the program that the aligned sequences are printed out additionally to their distance.

```
print("Sequence A: ", A)
print("Sequence B: ", B)
print("Distance between A and B: ", d)
```

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`

```
import sys

A = sys.argv[1]
B = sys.argv[2]

if (len(A) == len(B)):
    d = 0
    for i in range(len(A)):
        if (A[i] != B[i]):
            d = d + 1
        else:
            d = d + 0
    print("Sequence A: ", A)
    print("Sequence B: ", B)
    print("Distance between A and B: ", d)

else:
    print("Sequences A and B are of different length.")
```

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 `TACCATAC`

```
import sys

A = sys.argv[1]
B = sys.argv[2]

# reverse the B string and save as C
C = ""
for i in range(len(B)):
    C = C + B[len(B) - i - 1]

if (len(A) == len(B)):
    distAB = 0
    distAC = 0
    for i in range(len(A)):
        if (A[i] != B[i]):
            distAB = distAB + 1
        if (A[i] != C[i]):
            distAC = distAC + 0

    if (distAB <= distAC):
        print("Sequence A: ", A)
        print("Sequence B: ", B)
        print("Distance between A and B: ", distAB)
    else:
        print("Sequence A: ", A)
        print("Sequence C: ", C)
        print("Distance between A and C: ", distAC)

else:
    print("Sequences A and B are of different length.")
```

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:

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

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

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

```
pur = ["A", "G"]
pyr = ["C", "T"]

def similarity(a, b):
    if (a == b):
        return 1
    elif (((a in pur) and (b in pur)) or ((a in pyr) and (b in
pyr))):
        return 0.5
    else:
        return 0

def distance(a, b):
    if (a == b):
        return 0
    elif (((a in pur) and (b in pur)) or ((a in pyr) and (b in
pyr))):
        return 0.5
    else:
        return 1
```

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

```
def sequence_similarity (A, B):
    sim = 0.0
    for i in range(len(A)):
        sim = sim + similarity(A[i], B[i])
    return sim

def sequence_distance(A, B):
    dist = 0.0
    for i in range(len(A)):
        dist = dist + distance(A[i], B[i])
    return dist
```

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**

```
import sys
seq1 = sys.argv[1]
seq2 = sys.argv[2]
print "Similarity: ", sequence_similarity(seq1, seq2)
print "Distance: ", sequence_distance(seq1, seq2)
```

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:

```
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`:

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

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

```
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,

```
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

```
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 troubles (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 let's not get into that for now. (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.

```
#########################
### sequence_tools.py ###
#########################
pur = ["A", "G"]
pyr = ["C", "T"]

def similarity(a, b):
    if (a == b):
        return 1
    elif (((a in pur) and (b in pur)) or ((a in pyr) and (b in
pyr))):
        return 0.5
    else:
        return 0

def distance(a, b):
    if (a == b):
        return 0
    elif (((a in pur) and (b in pur)) or ((a in pyr) and (b in
pyr))):
        return 0.5
    else:
        return 1

def sequence_similarity (A, B):
    sim = 0.0
    for i in range(len(A)):
        sim = sim + similarity(A[i], B[i])
    return sim

def sequence_distance(A, B):
    dist = 0.0
    for i in range(len(A)):
        dist = dist + distance(A[i], B[i])
    return dist
```

2. Write another Python file that calculates for each combination of two sequences stored in list `l` the similarity and distance using the module defined previously.

```
l = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]
```

```
from sequence_tools import *
l = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]
s = 0
d = 0
for i in range(len(l)):
    for j in range(i+1, len(l)):
        s = s + sequence_similarity(l[i], l[j])
        d = d + sequence_distance(l[i], l[j])
        print(l[i], l[j], " Similarity: ", s, " Distance: ", d)
```

3. Extend your program. Determine the combination of sequences with the highest similarity of all sequences stored in list 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.

```
from sequence_tools import *

l = ["ATCCGGT", "GCGTTAC", "CTACTGC", "TTGCAGT", "AGTCACC"]
bestSim = 0
bestA = ""
bestB = ""
s = 0

for i in range(len(l)):
    for j in range(i+1, len(l)):
        s = s + sequence_similarity(l[i], l[j])
        if (s > bestSim):
            bestSim = s
            bestA = l[i]
            bestB = l[j]

matches = ""
for i in range(len(bestA)):
    if (bestA[i] == bestB[i]):
        matches = matches + "|"
    else:
        matches = matches + " "

outfile = open("similar_sequences.txt", "w")
outfile.write(bestA + "\n")
outfile.write(matches + "\n")
outfile.write(bestB + "\n")
```