1 Exercise : List transformation

In this exercise you will learn how to transform python lists and filter specific elements from them.

1.1 Transform an interval of numbers

1.1.1 Square elements of a list

Create two lists, one is called interval and has all integer numbers from 0 to 20 (inclusive) in it, the other is called sq_interval and is empty. Use a for loop to iterate over the elements of interval and fill sq_interval with the squared value of each element.

Expected output:
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121,

144, 169, 196, 225, 256, 289, 324, 361, 400]
Hint: make use of range() for creating interval and use the ** operator for squaring.

1.1.2 Remove odd numbers from the squared list
Create another empty list even_sq_interval. Fill it with only the even numbers from the

list sq_interval.

Expected output:
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324, 400]

Hint: remember the if statement and combine it with the modulus operator % 2 in order to estimate oddity/evenness of a number.

1.2 Transform a corrupted string

You are given the badly formatted DNA sequence

some_dna = "aACTa TtCcC acCtc\tcaTCC CGGCc\nTaTaT CTGaa"

and want to convert it into a more readable version.

1.2.1 Remove white space and convert sequences to capital letters

Copy the given string from this exercise sheet and assign it to a variable named some_dna within your python script. Create a list dna_pieces that includes all pieces of DNA from within some_dna, without white spaces (' ', \n and \t).

Expected output:
["aACTa", "TtCcC", "acCtc", "caTCC", "CGGCc", "TaTaT", "CTGaa"]

Hint: use str.split() (see python cheat sheet on page 1, string) to split the string into a list of pieces.

1.2.2 Convert the pieces into capital letters and combine them into a multi-line string

Now do a for loop over the previously created list dna_pieces, convert all letters in each piece into capital letters, and add them to the new list upper_dna_pieces. Then convert this new list to a string in which all pieces are connected by newline characters (\n).

Expected output:

AACTA
TTCCC
ACCTC
CATCC
CGGCC
TATAT
CTGAA

Hint: the string functions str.upper() and "\n".join() are useful for this task. Refer to the cheat sheet to see how they are used.

2 Exercise : Nucleotide counting

Your professor has a .txt file with a DNA sequence (strange_dna.txt) and would like to know which of the nucleotides A, T, G, C can be found how often in that sequence. Unfortunately he doesn't have time for counting and would like you to write the function count_nucleotides to make his life easier.

2.1 Create the function

Create the function count_nucleotides. Make sure it can be called from another cell with the name of the .txt file as its first argument. To begin with, try to make the function print the argument it was called with.

Expected output: "files/exercise_2/strange_dna.txt"

2.2 Read the .txt file

Add the open() command and a for loop to the function in order to load the text from the file into the variable DNA within python. print DNA to show its contents and make sure it doesn't contain newline characters (\n).

Expected output: "TACGGGGCCGACATCTGCGGGGGGGGCGGGGCGGCCCCCGGTCAGACC

AGAAAGGAGGCACCCCGCTTCCGCCGTGACTGGCATGTCGGGAGCGGGC

CCCGAGCCAGGCCGCTGAGGGGCCGCCCTGCGCGCGACCCGCAGGGGC
CCGCGGCGCCGACCAGCCGCCCCGCCCAGCCCCCTCGCGCTTCGACGGT
CCGCAACGCGCCAAGCCTCGCGAGCGCCGGCGCCTCCGACCGCGGGTGA
GACCGCGGCGCGCGTAACGCGCTCCAGTGCCAGGCGCGTCCCCTTACGCC
CGCGGGCAGTCCCCGGGCAGACACACGGAGGACCGGGATCAAATTGAACC
CAGAGCATCACGACACCACGGCCGGGGCCCCGCAACCCGGGGTGGGAGC
GGGACAGA"

Hint: open the file in read mode ('r'). Then read it line by line and add the lines to DNA using the += operator. Check the python cheat sheet (page 1, string) for a function to remove newline characters from strings.

2.3 Scan the sequence and count nucleotides

Now add another loop that visits each nucleotide in the variable DNA and counts for all types of nucleotides separately, how often they have been seen. If you use a dictionary for this, print it to the screen.

Expected output: {"A": 62, "T": 31, "C": 159, "G": 148}

Hint: use either 4 separate variables or a dict to keep track of the counts. You can use if/elif/else statements to distinquish between the different nucleotides in each step of the loop (but there is an easier way if you use a dict).

2.4 Report the results

Show the final nucleotides on the screen in an easily readable way. Expected output:

Count of A : 62
Count of T : 31
Count of G : 148
Count of C : 159

Hint: You can get all key-value pairs from a dictionary with the function dict.items() and then nicely display them with another for loop and print statements.

3 Exercise : Pandas DataFrames

optional challenge

This exercise does not depend on the previous exercise, but we will use some of the microbe data later.

What is pandas?

Oftentimes, experimental data is stored in Microsoft Excel files, and we want to take this data and analyze it with python. Biological data from public databases is often stored as a tab separated (tsv) or comma separated (csv) text file. What these different formats represent is called simply a table, a two-dimensional matrix or data frame. Since this is such a common format, there is a python module called pandas which makes it easy to read, write, filter, merge, perform arithmetic operations on, and plot such data frames.

Pandas is built on top of the numpy module, so almost all operations that can be performed on numpy arrays can also be performed on pandas dataframes.

3.1 Creating a data frame

3.1.1 Import modules

We'll start by importing the os, pandas, and numpy modules:

```
import os
import pandas as pd
import numpy as np
```

3.1.2 Create a pandas series

Create a simple list:

```
animals = [0.2, 4.5, 2.4]
```

We can turn this simple list into a pandas Series, which is a one-dimensional, named array:

```
animalSeries = pd.Series(animals)
```

Assign names to the Series values:

```
animalSeries.index = ['mouse', 'duck', 'snake']
animalSeries
Out[1]:
mouse    0.2
duck     4.5
snake    2.4
dtype: float64
```

3.1.3 Create a pandas data frame from two Series

May be you have more than one value per animal. This would now be a table, so we can create a pandas data frame from two Series:

```
animalSeries2 = animalSeries**2
pd.DataFrame([animalSeries, animalSeries2])
Out[2]:
  mouse   duck  snake
0  0.20   4.50   2.40
1  0.04  20.25   5.76
```

3.1.4 Create a pandas data frame from a matrix

Assume you have a matrix of values that you want to format as a data frame:

```
animalMeasurements = [['alpha', False, np.nan, 4],
     ['beta', False, 0.1, 2],
     ['beta', True, 0.333, 0],
     ['beta', True, -14, 30]]
animalDF = pd.DataFrame(animalMeasurements)
animalDF
Out[3]:

0123

    1.  0  alpha  False    NaN  4
    2.
    3.  1   beta  False  0.100  2
    4.

2  beta  True   0.333  0
3  beta   True -14.000  30
```

Name the columns and the index :

```
animalDF.columns = ['class', 'venomous', 'score', 'number_of_legs']
animalDF.index = ['mouse', 'duck', 'snake', 'centipede']
animalDF
Out[4]:
mouse      alpha
duck       beta
snake      beta
centipede   beta
False
False
 True
  NaN            4
0.100            2
0.333            0
animalDF.dtypes
Out[5]:
class
venomous
score
number_of_legs
dtype: object
 object
   bool
float64
  int64
class  venomous   score  number_of_legs
True -14.000           30
```

3.1.5 Pandas column types

You can see that pandas automatically assigns types to your columns:

3.1.6 Create a pandas data frame from a dictionary

Another possibility of creating a data frame is to provide the data as a dictionary, and to give the index already during data frame creation:

```
animalDF = pd.DataFrame({'class': ['alpha', 'beta', 'beta', 'beta'],
      'venomous': ['False', 'False', 'True', 'True'],
      'score': ['nan', '0.1', '0.333', '-14'],
      'number_of_legs': ['4', '2', '0', '30']},
index=['mouse', 'duck', 'snake', 'centipede'])
```

4.1.7 Indexing

Each column of the dataframe is a series:

animalDF.score

Another way of selecting a column, which is more robust:

```
animalDF.loc[:,'score']
Out[6]:
mouse
duck
snake
centipede
Name: score, dtype: float64
```

Select by column and index names:

```
animalDF.loc[['mouse', 'duck'],['score', 'venomous']]
Out[7]:
      score  venomous
mouse    NaN     False
duck     0.1     False
```

Select by column and index numbers:

```
animalDF.iloc[0:2,2:4]
Out[8]:
      score  number_of_legs
mouse    NaN             4
duck     0.1             2
```

3.2 Manipulating a real data file

```
   NaN
 0.100
 0.333
-14.000
```

Navigate to the folder with the files for Python exercise 4 (use os.chdir()), and use pandas to read the provided Excel file. It contains the microbial abundance data from the Human Microbiome Project from exercise 3.

```
samp = pd.read_excel('filtered_samples.xlsx')
```

3.2.1 Inspect a pandas object

Use the functions given below to get a first impression of the data in hand

```
samp.head()
samp.shape
samp.columns
samp.index
samp.dtypes
samp.describe()
samp.info()
```

Print the mean abundance for each microbe using the command below. What happens if you set axis=1?

```
samp.mean(axis=0)
```

3.2.2 Indexing

Use the .loc indexer as you learned above to create a new data frame called "selection" containing only the microbes with identifiers S97-100, S97-13505, and S97-8603. Expected output:

```
selection
Out[9]:
     S97-100  S97-13505  S97-8603
Oral     107          0         0
Oral      78          0         0
Oral     150          0         0
Oral     166          0         0
...
```

3.2.3 Column assignment

Currently, the body sites are used as an index. We want to create a new column called bodysites that contains this information instead, and just use a simple line count as the index instead. You can add a new column to an existing data frame using:

```
selection = selection.assign(bodysite = ...)
```

Expected output after assigning a new column and a new index:

```
selection.head()
```

Out[10]:

	S97-100	S97-13505	S97-8603	bodysite
0	107	0	0	Oral
1	78	0	0	Oral
2	150	0	0	Oral
3	166	0	0	Oral
4	0	0	0	Oral

What is the dtype of your new column?

The body site column does not contain lots of unique values, it instead represents a sample category. We can store this column as a categorical column in pandas, which is a lot more memory-efficient for large data frames, and allows plotting by category. You can change the body site column like so:

```
selection.loc[:,'bodysite'] = pd.Categorical(selection.bodysite)
```

Now check the dtypes again to see the change.

3.2.4 Sorting

We can sort a dataframe by any column or even by several columns, using the .sort_values() method. It can take either a single column name or a list of column names as input. With the ascending= parameter, you can select in which order to sort. Most pandas methods do not change the data frame itself, but return a copy of the data frame which is modified in the way you specified. If you want to change the data frame, you need to assign this modified copy to the original variable name.

Sort the 'selection' data frame by the microbe S97-100, with the largest value appearing at the top of the table. The dataframe should now look like this:

```
selection.head()
```

Out[11]:

	S97-100	S97-13505	S97-8603	bodysite
18	538	0	0	Oral
19	459	0	0	Oral
24	412	0	0	Oral
32	386	0	0	Oral
6	369	0	0	Oral

Did you notice how the row indexes stay with their rows?

3.2.5 Boolean indexing

You can filter for certain rows or columns of the data frame by specific criteria. This is extremely useful for large data frames. In order to do this, you need to create a boolean (True/False) data frame or series first, for example like this:

```
selection.loc[:,'bodysite'] == 'Skin'
Out[12]:

    18. 18  False
    19. 19  False

24    False
32    False
...
78    False
77    False
75    False
Name: bodysite, Length: 150, dtype: bool
```

Now we can use this Boolean pandas Series as an indexer for our data frame like this: 12

```
selection.loc[selection.loc[:,'bodysite'] == 'Skin',:]
Out[13]:
   S97-100  S97-13505  S97-8603 bodysite
138      79         0        0    Skin
118      28         0       20     Skin
148      26         0        0    Skin
147      23         0        0    Skin
124      22         0        0    Skin
139      13         0        0    Skin
```

Which part of the data frame is being selected here?

```
selection.loc[(selection.loc[:,'bodysite'] == 'Skin') &
  (selection.loc[:,'S97-8603'] >= 1), :]
```

Select all samples (i.e. rows) from the 'Feces' body site where at least one of the three microbes is present and assign this selection to a new data frame called feces_samples.

The .sum() method may come in handy for figuring out which samples are not all zeros. It works very similarly to the .mean() method.

3.2.6 Write to file

Now write the feces samples to a tab-separated file called feces_samples.tsv using the .to_csv('path\to\myfile.tsv', sep='\t') method. Can you find the parameter to use in order to not write the index to file?

3.2.7 Plot a pandas data frame

As you may know already, IPython has a magic command to enable displaying graphic output, %matplotlib. Turn on the graphic output using this command, and try a few simple plots of the data, for example:

```
selection.loc[selection.bodysite == 'Oral',:].plot.hist(subplots=True)
selection.plot.box(logy=True)
```

Pandas only offers a relatively basic plotting function. If for example we would want to plot the microbial abundances by bodysite, we would have to use the matplotlib module directly. A more elegant alternative would be to use the seaborn module, which provides flexible and beautiful plotting directly from pandas dataframe objects.

3.2.8 Further reading

Pandas offers a wide range of other highy useful methods for data analysis, such as the .groupby() method, the .merge() method, or the .apply() method. If you would like to

dive deeper into Pandas (and maybe also start using Jupyter notebooks), have a look at this beautiful and comprehensive Pandas introduction by our colleague David Lyon: https://github.com/dblyon/pandasintro

4 Exercise : Pipelines

optional challenge

You are working with two collaborators on a project to understand the human microbiome found in the body sites skin, oral tract and gut (the latter approximated by feces). Collaborator A is a microbiologist who has selected interesting samples from the Human Microbiome Project and provided them to you, while Collaborator B, who is a statistician, developed a method that infers ecological interactions between microbes from raw frequencies, which he implemented in the script compute_microbial_interactions.py. It is your task to make a pipeline script analyse_microbial_samples.py that converts the samples into a format suitable for interaction analysis, while also reporting general statistics about the data set like the most microbe-rich samples and the most abundant microbes, and then runs the statistical inference script to discover ecological interactions.

4.1 Create the pipeline script

Create the script analyse_microbial_samples.py with the input parameters body site, dataset path and output path in the following way:

ipython analyse_microbial_samples.py <body site> <dataset path>
<output path>

body site should be one out of Skin, Oral and Feces, every other input should result in an error message. You also want to make sure that a folder with path dataset path exists and that the folder output path is being created if it doesn't exist yet. The os module with its functions os.path.exists() and os.makedirs() can make this happen (use help() to see how these methods are used).

4.2 Filter the samples

A sample file has the following format:

Oral 107 0 29 0 0 86 325 0 0 0 0 0 0 0 0 0 3 550 0 42 ...

where items are delimited by single spaces and the body site from which the sample was taken is indicated in the first field (Oral in this case), followed by abundance counts for all microbes measured within the sample. Here, the nth count in each sample shows the abudance for the same microbe and identifiers for each organism (i.e. the ID of the nth microbe) can be found in file microbial_identifiers.txt.

You agreed with your collaborators to analyse only one body site at a time, starting with Oral. Unfortunately, all body sites are mixed within the sample files and need to be filtered prior to the analysis. Also, ideally you want to collect all filtered samples into one file filtered_samples.txt located in the folder at output path. In the case of oral:

Oral 107 0 29 0 0 86 325 0 0 0 ...
Oral 78 0 5 0 0 36 11 0 0 0 ...
Oral 150 0 3 0 0 5 3 0 0 0 ...
Oral 166 0 9 0 0 7 46 1 0 0 ...

As you learned in the morning, Unix provides a number of powerful tools to achieve this, in particular the piping operator |. We do not want to enter the unix command each time we get a new data set or investigate a new body site, which is why we will call the termi- nal command automatically from within python using the subprocess module and let the script do the work. With this module, you can call any software from within python, wait for the results and process them further from within the same script, which is the essence of a python pipeline. For starters, it is easiest to use the function subprocess.run() with the parameter shell=True. This allows you to pass a string with the unix command to the shell, execute it, and then proceed with the python script by reading the output file created by that command.

Hint: cat and grep come in handy. Make sure to append the name of the output file to the output folder using os.path.join().

4.3 General data set statistics

After you have assembled your oral data set, you want to see which samples harbor the most microbes in total (sum of all abundances) and which microbes are on average the most abundant in the oral microbiome (mean across all samples). You can do this by reading the data set file previously created by the unix command and then applying appropriate sum and means calculations to its rows and columns.

Other than in the previous exercises, we will use the numpy module to read the data from the text file into a numpy array (a fast numerical version of a traditional python llist) and use numpy math functions to compute the desired statistics. The standard convention for importing numpy is:

import numpy as np
Usenp.<tab>ortheofficialnumpy referenceathttps://docs.scipy.org/doc/numpy/

reference/routines.html to find the appropriate file reading and statistics functions.

Hint: While there are different ways of reading data in numpy , the easiest solution here is the genfromtxt function. Remember to remove the first column of the matrix through

slicing, since it will be filled with NaNs (elements that were "Oral" in the original text file and cannot be interpreted as numbers)

Finally, print the sorted list of all abundance sums (i.e. for each sample): [89233, 47277, 29332, 28917, 28143, ...]

For the microbe abundances, you would additionally like to report the microbe identifiers which can be read from the file microbe_identifiers.txt. While there are multiple ways of iterating over two lists at the same time (one with microbe identifiers and one with abundances), can you figure out how to do this with the function zip()? The output should look like this:

S97-100 98.9
S97-10339 0.2
S97-105 139.3
S97-11531 0.14
S97-12192 0.0
...

(Bonus: report only the 10 microbes with the highest average abundance, in descending sorting order. If you are unsure on how to sort a list of pairs, read the documentation of the sort commands and check online resources like www.stackoverflow.com)

Hint: The data set is basically a matrix of counts, which can be represented as a list of lists. Python features multiple ways of sorting lists, all of which have an argument that specifies in which direction the list should be sorted.

4.4 Microbial interactions

Now that you learned a bit about the oral microbiome, you want to go a step further and look at which microbes usually occur together, which can indicate a benefitial relationship between the species, and which microbes tend to avoid each other, hinting at a negative ecological interaction or distinct habitat preferences. As mentioned earlier, the method is al- ready implemented in compute_microbial_interactions.py with the following signature:

ipython compute_microbial_interactions.py <filtered dataset path>

Luckily, your pipeline already produces files of the right input format for this script (filtered_samples.txt). As with the Unix command, you want to call the interaction computation script from within your own pipeline such that you don’t have to do it manually for future data sets.

The script creates a file pairwise_interaction_strengths.txt within the same folder as the input file, which gives for each pair of microbes a number that indicates, how strong the interaction between these microbes is and also, whether it is positive (symbiosis) or

negative (avoidance), as indicated by the value's sign. 1.0 means perfect positive interaction, while -1.0 indicates a total avoidance relationship. Microbial identifiers for the rows and columns of this matrix are the same as for the input dataset. You want to read this output file and report for each microbe the strongest positive and the strongest negative interaction partner by printing interaction partner IDs and exact interaction strength values. The output should look as follows:

S97-100 : S97-15892 (0.8061); S97-39454 (-0.3676)
S97-10339 : S97-30317 (0.5249); S97-45618 (-0.1705)
S97-105 : S97-13107 (0.6); S97-20690 (-0.2652)
S97-11531 : S97-13620 (0.9209); S97-40435 (-0.1275)
...

(Bonus: look at the script compute_microbial_interactions.py. Can you figure out, which statistical measure it uses to approximate interaction strength? Is there a numpy func- tion that can do the same more conveniently and faster? How much faster is it exactly (you can use the time module or the more convenient %timeit magic in ipython)?

Hint: For technical reasons the script also returns the interaction strength of a microbe with itself, which is always a perfect positive interaction. Make sure to not report this self-relationship as the strongest one.

END