Homework 5 - Numpy arrays and data structures

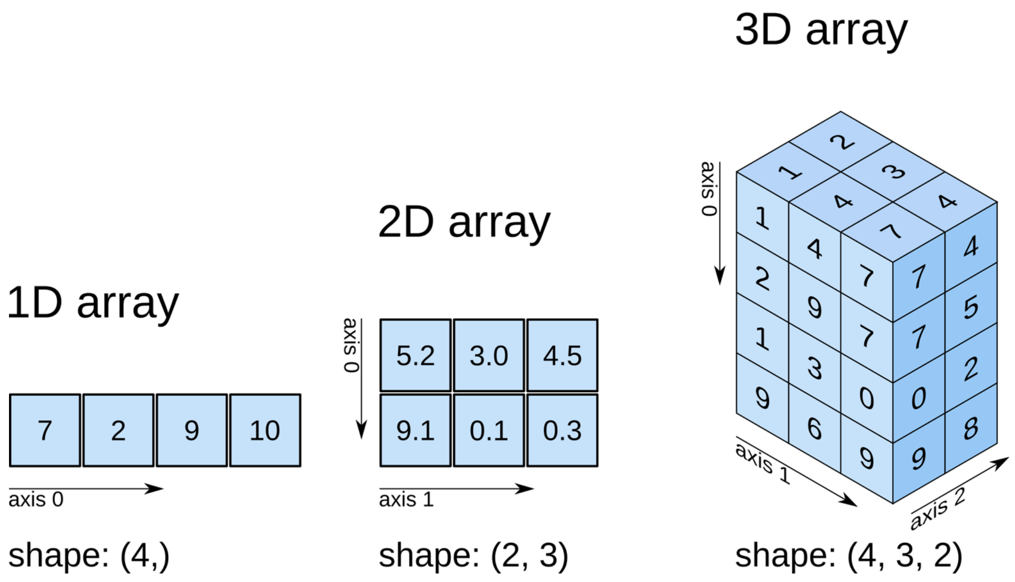
BIOINF 575 - Fall 2021

Total 60 points

For each each problem part provide the solution in Code cells after the description of the problem part. Answers to questions should be written either as comments together with the code or in Markdown cell(s) for each part of the problem.

This homework will require the use of numpy arrays.

Arrays of different dimensions (`shape` gives the number of elements on each dimension):



Basic array attributes:

- shape: array dimension
- size: Number of elements in array
- ndim: Number of array dimension (len(arr.shape))
- dtype: Data-type of the array
- slicing: [row_slice,col_slice]

Pandas Series a named array:

```
pd.Series(data, index)
```

Problem 1 - Exploring time course gene expression data

Identify the time interval with the most changed genes

20 points

The file **GSE22955_small_gene_table.txt** contains gene expression data for about 10000 genes

during treatment of breast cancer cells with a HER2 inhibitor data was collected at every three hours for 45 hours.
The file contains tab-separated data, has a header that contains the time points and row labels on the first column that are gene symbols.
More details at: https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE22955

Below is code to get the data loaded into numpy array that you can use to solve the problem.

In [10]:
```
# This code loads the data into a 2D array

# genes on the rows time points (h) on the columns every three hours starting at 0h end
# total of 16 time points, 17 columns with the gene symbols

import numpy as np
import pandas as pd

time_course_filename = "GSE22955_small_gene_table.txt"

# the following line uses the loadtxt function form the np library to read data from a 
# it skips the first two rows one. comment row and a header rowL skiprows = 2
# it takes only columns 2 to 17 skipping the first column where we have gene symbols: u
# help(np.loadtxt) # see function documentation
expr_matrix = np.loadtxt(time_course_filename, skiprows = 2, usecols = range(1,17))
print("Expression matrix dimensions: ", expr_matrix.shape)

time_points = 3 * np.arange(16)
print("Time points in hours: ", time_points)


# you will not need the genes symbols for this problem
# it is here to use if you want to explore the data further
# the following line of code creates an array with the gene symbols
# takes them from the first column in the file: usecols = 0
# it makes the data str, default is numeric
gene_symbols = np.loadtxt(time_course_filename, skiprows = 2, usecols = 0, dtype = str)
print("Gene symbols: ", gene_symbols)
```

```
Expression matrix dimensions:  (1175, 16)
Time points in hours:  [ 0  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45]
Gene symbols:  ['ABCA1' 'ABCC11' 'ABCC3' ... 'ZNF83' 'ZNFX1' 'ZWINT']
```

....

Part 1 (6 points) - Count the number of genes that change between two time points

Count the number of genes that have a change (up or down) of at least a 25% between two time points 0h and 12h (the time points are two numbers that represent hours)

- (3 points) Compute the log fold change for each gene/row
  - the log fold change is the log 2 of the ratio between the gene expression at the two time points (expr at 12h/expr at 0h)
  - the np library has a log2 function youcan make use of

- vectorized operations allow you to apply a funtion or an to the array or operations between arrays and it will be applied to each element of the array `python`
- (3 points) Count the rows that have at least a 25% change (ratio > 1.25, up or down) in average expression between the time points
  - the log is a positive value for numbers >1 (increase in expression) and negative for numbers <1 (decrease in expression) so you can use the absolute value to capture both
  - use sum to count values for which a contition is true, for arrays the condition is applied to each value yielding a logical array (of True, which is 1 and False, which is 0) and adding up all values from the logical array will count the number of True values

In [33]:
```
#Write your solution here, feel free to add new cells.

#compute the log fold change for each gene/row. Approaching this solution from
#a general viewpoint.

time_0 = 0
time_1 = 12
np.where(time_points == time_0)
index_0 = np.where(time_points == time_0)[0][0]
index_1 = np.where(time_points == time_1)[0][0]

#compute the ratio between the gene expression at the two time points.
gene_ratio = (expr_matrix[:,index_1]/expr_matrix[:,index_0])
#use the log2 function from the np library.
gene_log = np.log2(gene_ratio)
#use absolute value here to capture both positive and negative values.
gene_abs = abs(gene_log)
#use panda series to include the row names. This was to help me visualize the
#information better.
gene_fold = pd.Series(gene_abs, index = gene_symbols)
#take the log of the percentage and then count the rows that have at least a 25%
#change in the average expression between the two time points.
log_var = np.log2(1.25)
get_25 = gene_fold > log_var
value_sum = get_25.sum()
print(value_sum)
```
```
5
```

In [ ]:
```
```

....

Part 2 (14 points) - Identify the time interval with the most changed genes

Apply the code/approach used at Part 1 to compute the count of genes for every pair of time points and identify the pair of time points with the highest number of changed genes.

- (6 points) Create the combination of every two (different) time points
  - You can use two for loops to create the combination of timepoints (recommended) or you can find functionality already available in python that does this
- (6 points) Find the pair with the maximum number of genes (if they are multiple you can just find the last one)

    - You can use the find maximum algorithm shown in one of the first sessions in the class
- (2 points) Display the result in a nice format. You can use the following f-string:
  f"The time interval with the most, {x}, changes genes is {start_time}h - {stop_time}h"

In [25]:
```
#Write your solution here, feel free to add new cells.

#create a "global" tuple because it is outside of the for loop.
#this will contain what I consider for now a max number of genes
#along with its corresponding time points. It will be used later
#to iterate through all the gene counts to find the real max.
max_tuple = (time_0, time_1, value_sum)

#beginning of my for loops.
for time_start in time_points:
    for time_end in time_points:
        if time_start < time_end:
            index_0 = np.where(time_points == time_start)[0][0]
            index_1 = np.where(time_points == time_end)[0][0]
            gene_ratio = (expr_matrix[:,index_1]/expr_matrix[:,index_0])
            gene_log = np.log2(gene_ratio)
            gene_abs = abs(gene_log)
            gene_fold = pd.Series(gene_abs, index = gene_symbols)
            log_var = np.log2(1.25)
            get_25 = gene_fold > log_var
            value = get_25.sum()

            if value > max_tuple[2]:
                max_tuple = (time_start, time_end, value)
print(max_tuple)
```
```
(0, 45, 78)
```

In [86]:
```
f"The time interval with the most changed genes, {max_tuple[2]}, is from {max_tuple[0]}
```

Out[86]:
```
'The time interval with the most changed genes, 78, is from 0 hours - 45 hours.'
```

Problem 2 - Understand the code

10 points

Transcription networks contain a small set of recurring regulation patterns, called network motifs.

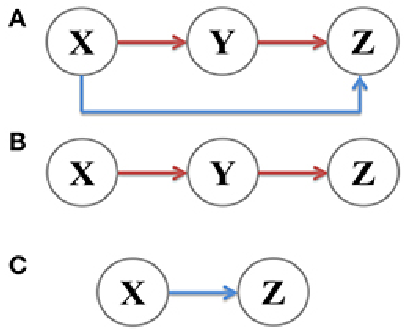
Some references about network motifs in biological networks.

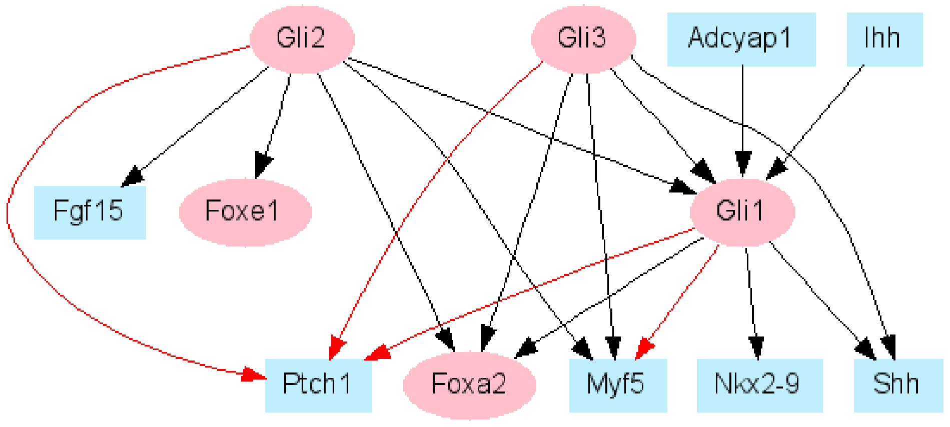
https://www.nature.com/articles/nrg2102

https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-9863-7_463

https://media.springernature.com/lw785/springer-static/image/prt%3A978-1-4419-9863-7%2F6/MediaObjects/978-1-4419-9863-7_6_Part_Fig1-463_HTML.gif

Feed forward loop (A) - small recurring pattern (B and C show the two parts of the FFL):





https://www.frontiersin.org/files/Articles/222083/fphys-07-00600-HTML/image_m/fphys-07-00600-g001.jpg http://rulai.cshl.edu/TRED/GRN/Gli.htm

....

Explain what the following code does and how it computes the result it displays.

Score breakdown:

- **(10 points) Explaining each statement is worth 1 point and 1 point for explaining what the function does overall**

In [96]:
```
biological_network = {
    "Adcyap1": {"Gli1"},
    "Gli1": {"Ptch1", "Foxa2", "Myf5", "Nkx2-9", "Shh"},
    "Gli2": {"Ptch1", "Fgf15", "Foxe1", "Foxa2", "Myf5", "Gli1"},
    "Gli3": {"Ptch1", "Foxa2", "Myf5", "Gli1", "Shh"},
    "Ihh": {"Gli1"}
}
```

In [97]:
```
def compute_FFL(network):
    FFL_set = set()
    for gene in network:
        reg_genes = network[gene]
        for regg in reg_genes:
            for iregg in set(reg_genes).intersection(set(network.get(regg,()))):
                FFL_set.add((gene,regg,iregg))
    return FFL_set
```

```
compute_FFL(biological_network)
```

Out[97]:
```
{('Gli2', 'Gli1', 'Foxa2'),
 ('Gli2', 'Gli1', 'Myf5'),
 ('Gli2', 'Gli1', 'Ptch1'),
 ('Gli3', 'Gli1', 'Foxa2'),
 ('Gli3', 'Gli1', 'Myf5'),
 ('Gli3', 'Gli1', 'Ptch1'),
 ('Gli3', 'Gli1', 'Shh')}
```

In [ ]:
```

```

Write your solution here and then run the cell (Markdown)

Cell #1:

Line 1: You are setting a dictionary equal to the variable biological_network where the keys are strings and the values are sets of string values.

Cell #2:

Line 1: You are defining a function that is called compute_FFL with one parameter called network, which will be a dictionary.

Line 2: You are creating an empty set equal to the variable FFL_set.

Line 3: You are beginning a for loop where the variable called gene is iterated through your parameter network. So it is being iterated through the dictionary keys.

Line 4: You are setting a new variable called reg_genes equal the set of values from the specific dictionary key, variable gene, in the dictionary called network.

Line 5: You are beginning a second for loop where your variable regg is iterated through reg_genes, which is your set of values for your specific dictionary key "gene".

Line 6: You are beginning your third for loop where your variable iregg is iterated through the value set of reg_genes, and is looked for in the value set for another dictionary key which is represented by the variable regg. If irreg is found in both value sets of reg_genes and regg, you are to get that value for iregg.

Line 7: You are adding to your empty set (FFL_set) your variables gene, regg, and iregg in that order.

Line 8: You end your function by returning your FFL_set set.

Line 9: You call the function with your parameter being your biological_network dictionary.

Overall Function Goal: Take a gene, find that gene in the dictionary's keys, and look at that value set. See if you find another dictionary key in that particular value set. If you do, compare that new key's value set with your starting key's value set and look for intersection. In other words, look for any values in the value sets that are the same. For values that are the

same in both value sets, print each value with both keys in separate tuples. This will be the returned output.

double click here to edit the cell _

```
In [ ]:
```

Problem 3 - Covid-19 data analysis - try to find out what makes Dexamethasone more dangerous for some patients

20 points

"COVID-19 ICU patients with the IFIH1 rs1990760 TT variant show an ameliorated inflammatory response that results in better outcomes than CC/CT variants. Dexamethasone may reverse this anti-inflammatory phenotype."
The above is the conclusion from a recent (July 2021) study.
The data and more details from the study can be found at the following urls:
https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE168400
https://www.medrxiv.org/content/10.1101/2021.07.03.21259946v1

...

We took the read counts data from the study mentioned above and want to detect the genes that show a highchange in expression between the patients with the TT variant and the other patients.
The counts data is in the file covid_counts.txt.
The file has genes on the rows and patients/samples on the columns, the last column is the gene length in number of bases/nucleotides.

```
In [30]:
import numpy as np
import pandas as pd

covid_file_name = "covid_counts.txt"

# thegrouplabelsfor the data
group_labels = pd.Series(["TT", "CT", "CC", "CT", "CC", "TT", "CT",
                          "CT", "CC", "TT", "CC", "TT", "TT", "CC",
                          "CC", "CT", "TT", "CT", "CC", "TT", "CC",
                          "CT", "CT", "CC", "CT", "CT", "CT", "CT",
                          "CT", "CT", "CC", "CT", "CC", "TT", "CC",
                          "TT", "TT", "TT", "CT", "CT", "CT", "CT"],
                         index= ['Patient1', 'Patient9', 'Patient10', 'Patient11', 'Patie
                                 'Patient13', 'Patient14', 'Patient15', 'Patient16', 'Pat
                                 'Patient18', 'Patient2', 'Patient19', 'Patient20', 'Pati
                                 'Patient22', 'Patient23', 'Patient24', 'Patient25', 'Pat
                                 'Patient27', 'Patient28', 'Patient3', 'Patient29', 'Pati
                                 'Patient31', 'Patient32', 'Patient33', 'Patient34', 'Pat
                                 'Patient36', 'Patient37', 'Patient4', 'Patient38', 'Pati
                                 'Patient40', 'Patient41', 'Patient42', 'Patient5', 'Pati
                                 'Patient7', 'Patient8'])

#  load the counts data into an array for all 42 patients
```

```
counts_matrix = np.loadtxt(covid_file_name, skiprows = 1, usecols = range(1,43))
print("Counts matrix dimensions (N genes by M patients): ", counts_matrix.shape)

# gene length (median length of all its mature, mRNA, transcripts) in number of base pa
# we transform the data in kilo base by dividing by 1000
gene_length_array = np.loadtxt(covid_file_name, skiprows = 1, usecols = 43)/1000
gene_symbol_array = np.loadtxt(covid_file_name, skiprows = 1, usecols = 0, dtype = str)

gene_data = pd.Series(gene_length_array, index = gene_symbol_array)
print("Gene length data (in kb): \n", gene_data)
```

```
Counts matrix dimensions (N genes by M patients):  (17624, 42)
Gene length data (in kb):
 "A1BG"       1.7720
"A1CF"       2.1480
"A2M"        4.6890
"A2ML1"      5.2220
"A4GALT"     2.0940
              ...
"ZYG11A"     4.2850
"ZYG11B"     8.1540
"ZYX"        2.3125
"ZZEF1"     11.4490
"ZZZ3"       4.3370
Length: 17624, dtype: float64
```

....

Part 1 (5 points) - Compute TPM

Before we can compute the difference between the TT and the other phenotype we should quatify the expression value, we will use the TPM (Transcripts Per Kilobase Million) measure to do that.

To compute the TPM you do:

- **(2 points) Divide the read counts by the length of each gene in kilobases. This gives you reads per kilobase (RPK).**
  - **Should make the gene_length_array a column (reshape) so you can do operations (division) at the column level**
- **(2 points) Count up all the RPK values in a sample and divide this number by 1,000,000. This is your “per million” scaling factor.**
- **(1 point) Divide the RPK values by the “per million” scaling factor. This gives you TPM.**

In [38]:
```
#Write your solution here, feel free to add new cells.

#gene_length_array is a 1D array which is one row with 17624 elements.
#We want one column with those many elements. So we'll have to reshape.
gene_length_array
gene_length_array.size
```

Out[38]:
```
17624
```

In [88]:
```
#reshaping the gene_length_array so that we can divide counts_matrix by it.
#we want it to be a one column array.
gene_length_array_reshaped = gene_length_array.reshape(gene_length_array.size,1)
```

```
#now that gene_length_array is reshaped, we can divide counts_matrix by this reshaped a
RPK_values = counts_matrix/gene_length_array_reshaped
RPK_values
```

Out[88]:
```
array([[1.69300226e+00, 1.69300226e+00, 7.33634312e+00, ...,
        1.12866817e+00, 2.25733634e+00, 4.51467269e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 1.27959053e+00, 1.27959053e+00, ...,
        0.00000000e+00, 2.77244615e+00, 6.39795266e-01],
       ...,
       [1.58486486e+03, 2.94529730e+03, 1.83005405e+03, ...,
        8.98594595e+02, 3.88756757e+03, 1.01664865e+03],
       [3.68591143e+01, 6.55952485e+01, 3.44134859e+01, ...,
        2.82994148e+01, 5.50266399e+01, 4.11389641e+01],
       [9.45353931e+00, 1.77542080e+01, 1.59096149e+01, ...,
        2.69771732e+01, 1.77542080e+01, 5.21097533e+01]])
```

In [90]:
```
#I want to sum all the RPK values for each sample, meaning each column.
#So I should be able to see 42 separate sums. To visualize this better for myself,
#I used a Dataframe to see it, plus it's good practice to use Dataframes from class :)

RPK_values_DF = pd.DataFrame(data = RPK_values)
RPK_values_DF
```

Out[90]:

	0	1	2	3	4	5	6
0	1.693002	1.693002	7.336343	5.643341	4.514673	6.207675	12.415350
1	0.000000	0.000000	0.000000	0.000000	0.465549	0.465549	0.000000
2	0.000000	1.279591	1.279591	0.853060	0.000000	0.426530	0.000000
3	0.382995	0.000000	0.191498	0.574493	0.574493	0.382995	0.191498
4	0.000000	0.000000	0.000000	1.432665	0.477555	3.820439	3.820439
...	...	...	...	...	...	...	...
17619	1.866978	3.267211	3.500583	2.333722	4.900817	5.600933	9.334889
17620	20.358106	25.876870	16.188374	18.273240	13.490312	16.556291	15.084621
17621	1584.864865	2945.297297	1830.054054	3260.108108	1241.945946	1166.270270	1693.405405
17622	36.859114	65.595248	34.413486	46.991004	25.067692	25.941130	35.024893
17623	9.453539	17.754208	15.909615	20.982246	11.298132	19.137653	18.445930

17624 rows × 42 columns

In [55]:
```
#Now I want to take the sum of each sample or column in this dataframe.
#the result should give me a total of 42 individual sums.
RPK_sum_DF = RPK_values_DF.sum()
RPK_sum_DF
```

Out[55]:
```
0     3.737677e+06
1     3.663875e+06
```

```
2     3.656148e+06
3     4.622665e+06
4     4.341615e+06
5     5.217481e+06
6     5.243704e+06
7     5.395328e+06
8     5.467460e+06
9     4.590804e+06
10    5.030903e+06
11    7.797163e+06
12    5.177077e+06
13    6.776149e+06
14    5.733708e+06
15    5.754934e+06
16    4.192604e+06
17    2.709181e+06
18    2.637149e+06
19    5.098205e+06
20    6.191517e+06
21    5.479956e+06
22    5.008046e+06
23    5.990210e+06
24    4.360611e+06
25    4.055595e+06
26    3.739412e+06
27    3.643908e+06
28    5.247223e+06
29    7.680379e+06
30    5.256261e+06
31    2.628549e+06
32    4.335472e+06
33    2.745889e+06
34    4.808920e+06
35    5.703400e+06
36    5.133438e+06
37    4.397216e+06
38    5.267491e+06
39    5.813046e+06
40    5.389168e+06
41    4.605905e+06
dtype: float64
```

In [91]:

```
#Now to calculate the scaling factor, I need to dividie each sum by 1000000.
RPK_scal_fac = RPK_sum_DF/1000000
RPK_scal_fac
```

Out[91]:

```
0     3.737677
1     3.663875
2     3.656148
3     4.622665
4     4.341615
5     5.217481
6     5.243704
7     5.395328
8     5.467460
9     4.590804
10    5.030903
11    7.797163
12    5.177077
13    6.776149
14    5.733708
15    5.754934
16    4.192604
```

```
17    2.709181
18    2.637149
19    5.098205
20    6.191517
21    5.479956
22    5.008046
23    5.990210
24    4.360611
25    4.055595
26    3.739412
27    3.643908
28    5.247223
29    7.680379
30    5.256261
31    2.628549
32    4.335472
33    2.745889
34    4.808920
35    5.703400
36    5.133438
37    4.397216
38    5.267491
39    5.813046
40    5.389168
41    4.605905
dtype: float64
```

In [59]:

```
#Now to calculate the TPM I simply take my values dataframe and divide it by my scaling
#This is what I want to use for part 2 now.
TPM_DF = RPK_values_DF/RPK_scal_fac
TPM_DF
```

Out[59]:

	0	1	2	3	4	5	6	
0	0.452956	0.462080	2.006577	1.220798	1.039860	1.189784	2.367668	0.00
1	0.000000	0.000000	0.000000	0.000000	0.107230	0.089229	0.000000	0.00
2	0.000000	0.349245	0.349983	0.184539	0.000000	0.081750	0.000000	1.14
3	0.102469	0.000000	0.052377	0.124277	0.132322	0.073406	0.036520	0.00
4	0.000000	0.000000	0.000000	0.309922	0.109995	0.732238	0.728576	0.08
...	...	...	...	...	...	...	...	
17619	0.499502	0.891737	0.957451	0.504843	1.128800	1.073494	1.780209	0.25
17620	5.446727	7.062706	4.427713	3.952966	3.107211	3.173234	2.876711	3.02
17621	424.024032	803.875035	500.541570	705.244275	286.056226	223.531275	322.940700	530.02
17622	9.861503	17.903246	9.412498	10.165349	5.773817	4.971964	6.679419	13.22
17623	2.529255	4.845747	4.351469	4.538993	2.602288	3.667987	3.517729	12.82

17624 rows × 42 columns

In [ ]:

....

Part 2 (15 points) - Identify changed genes

Identify the genes with the highest difference (>2 log 2 fold change) between the TT and the other phenotypes.

To identify these genes:

- **(3 points) Separate the TPM matrix in two matrices select the columns for the TT group in one matrix and the columns for the other groups in another matrix**
  - **Use the `group_labels` Series, select from the TPM matrix only columns that have `group_labels == "TT"` for one group and `group_labels != "TT"` for the other group**
- **(3 points) Compute the mean value for each in gene in the two groups**
  - **Apply the mean method for each row in the two matrices to result in two one dimensional arrays that have the average TPM for each gene**
- **(3 points) Take the log 2 of the ratio between the mean in the TT and non TT groups**
  - **Call the log2 method from the np library on the ratio between the two arrays computed at the previous step**
  - **You will get a warning because of division by 0 and na.inf values will be created**
- **(3 points) Create a named array (Series) with the log2FC and the gene symbols**
  - **Create a pandas Series using the array created at the previous step (lo2FC) as the values and the `gene_symbol_array` as the index**
- **(3 points) Select the genes with the highest difference (>3 log 2 fold change) between the TT and the other phenotypes**
  - **From the above Series select only the elements with an absolute value >2 and that are not na.inf**

In [101…

```
#Write your solution here, feel free to add new cells.

#I wanted to see if I could convert group_labels first to an array to
#make things easier for myself at this time. I did research and found
#this to_numpy() method and tried it out.
labels_array = group_labels.to_numpy()
labels_array

#I then converted my TPM dataframe to an array too. So now I'm just working
#with arrays.
TPM_array = TPM_DF.to_numpy()

#Then use conditiionals to separate the TPM_array into two groups where
#samples either equaled "TT" or did not equal "TT".
group_TT = TPM_array[:, labels_array == "TT"]
group_non_TT = TPM_array[:, labels_array != "TT"]

#computd the mean value for each gene in the two groups.
#I used mean(1) because I am taking the mean of each row, not column.
mean_group_TT = group_TT.mean(1)
mean_group_non_TT = group_non_TT.mean(1)

#take the log 2 of the ratio between the mean in the TT and non TT groups.
```

```
group_ratio = mean_group_TT/mean_group_non_TT
group_log2FC = np.log2(group_ratio)
group_log2FC
```

```
<ipython-input-101-602cf45ae1c6>:24: RuntimeWarning: divide by zero encountered in true_
divide
  group_ratio = mean_group_TT/mean_group_non_TT
<ipython-input-101-602cf45ae1c6>:24: RuntimeWarning: invalid value encountered in true_d
ivide
  group_ratio = mean_group_TT/mean_group_non_TT
<ipython-input-101-602cf45ae1c6>:25: RuntimeWarning: divide by zero encountered in log2
  group_log2FC = np.log2(group_ratio)
```

Out[101…

```
array([-0.0284585 , -0.11315733, -0.72139435, ...,  0.04934111,
        0.05639543, -0.22147326])
```

In [95]:

```
#take the absolute value of group_log2FC.
group_log2FC_abs = abs(group_log2FC)

#create a named array (panda Series) with the log2FC and the gene symbols.
log2FC_series = pd.Series(data = group_log2FC_abs, index = gene_symbol_array)

#select the genes with the highest difference between the TT and the other phenotypes.
#want only the elements with an absolute value >2 and that are not np.inf
#which means I'm going to use conditionals.
log2FC_FINAL = log2FC_series[(log2FC_series > 2) & (log2FC_series != np.inf)]
log2FC_FINAL
```

Out[95]:

```
"ABCC6P1"     2.400189
"ABRA"        2.268561
"ACSBG2"      2.321102
"ADAMTS19"    2.021450
"ADARB2"      2.410965
                ...
"ZBBX"        2.396594
"ZNF48"       2.080803
"ZNF648"      2.882104
"ZNF705D"     2.165953
"ZP1"         2.174154
Length: 251, dtype: float64
```

In [ ]:

Problem 4 - Understand the code

10 points

Explain what the following code does and how it computes the result it displays.

Score breakdown:

- **(4 points) Explaining the function: each statement is worth 0.5 points**
- **(6 points) Explaining the code in the second cell: each statement is worth 1 point**

In [98]:

```
def update_network(gene1, gene2, network):
    if gene1 in network:
        if gene2 in network[gene1]:
            print(gene1, "->", gene2, "link already there.")
        else:
            network[gene1].add(gene2)
            print("Added ", gene1, "->", gene2, "link.")
    else:
        network[gene1] = {gene2}
        print("Added ", gene1, "->", gene2, "link.")
    return network # this is not necessary, the dictionary will be updated no need to r
```

In [99]:

```
# network - might want to use this to reset the network while you test the code

test_network = {
    "Adcyap1": {"Gli1"},
    "Gli1": {"Ptch1", "Foxa2", "Myf5", "Nkx2-9", "Shh"},
    "Gli2": {"Ptch1", "Fgf15", "Foxe1", "Foxa2", "Myf5", "Gli1"},
    "Gli3": {"Ptch1", "Foxa2", "Myf5", "Gli1", "Shh"},
    "Ihh": {"Gli1"}
}

ntw_edges = (("Ihh", "Shh"), ("Gli3", "Shh"), ("Ext1", "Ihh"))
for link in ntw_edges:
    gene1, gene2 = link
    update_network(gene1, gene2, test_network)
test_network
```

```
Added  Ihh -> Shh link.
Gli3 -> Shh link already there.
Added  Ext1 -> Ihh link.
```

Out[99]:

```
{'Adcyap1': {'Gli1'},
 'Gli1': {'Foxa2', 'Myf5', 'Nkx2-9', 'Ptch1', 'Shh'},
 'Gli2': {'Fgf15', 'Foxa2', 'Foxe1', 'Gli1', 'Myf5', 'Ptch1'},
 'Gli3': {'Foxa2', 'Gli1', 'Myf5', 'Ptch1', 'Shh'},
 'Ihh': {'Gli1', 'Shh'},
 'Ext1': {'Ihh'}}
```

In [ ]:

Write your solution here and then run the cell (Markdown)

Code in first cell:

Line 1: You are defining the function update_network with three parameters: gene1, gene2, and network.

Line 2: You are writing an if statement saying if your variable gene1 is in the network, which is a dictionary's keys.

Line 3: You are writing a second if statement saying if gene2 is in the value set for the gene1 key in the dictionary.

Line 4: Then print in this order the variable gene1, "->", variable gene2, and the string "link already here."

line 5: The else portion of the second if statement.

Line 6: Take the value set of the gene1 key and add the value for gene2 to the gene1 key's value set in the dictionary.

Line 7: Then print in this order "Added ", gene1, "->", gene2, "link."

Line 8: The else part of the first if statement in the function.

Line 9: Take gene1 and create a new dictionary key for gene1 along with its corresponding value set which will contain gene2. Add this to the exisiting dictionary.

Line 10: Then print in this order "Added ", gene1, "->", gene2, "link."

Line 11: Return your parameter network, which will be your updated dictionary.

Code in second cell:

Line 1: Setting your dictionary equal to the variable test_network, where keys are strings and values are sets of strings.

Line 2: Setting a new variable called ntw_edges equal to a list of tuples, each containing two string values.

Line 3: Beginning a for loop where your variable link is iterated through your list of tuples called ntw_edges.

Line 4: Have your two parameters in your function gene1 and gene2 equal your variable link which is iterated through your list of tuples. So you are seeing if your gene1 and gene2 variables are present in any of these tuples.

Line 5: Call your function update_network and give it your three parameters: gene1, gene2, and test_network.

Line 6: Return/print your test_network dictionary.

double click here to edit the cell _

```
In [ ]:
```