# Repeat Exam BIO134: Programming in Biology HS2023

27.08.2024; 13:30-16:00

## Question 1

```
genotypes = [1, 4, 2, 1, 3, 4, 3, 4, 1, 4, 3, 2, 4, 4, 2]
```

The list *genotypes* contains integers between 1 and 4, representing different genotypes. Write a program that counts the number of elements in the list that are smaller than 4. Your program should then compute the fraction of these elements relative to the total number of elements in the list.
It should first print the count of numbers smaller than 4, and then the fraction as follows:

```
9
0.6
```

The program should be general enough such that it would still work according to the same principle if *genotypes* was replaced by another list of integers of different length.

## Question 2

```
codons = ['TCG', 'ATG', 'ATT', 'TCT', 'GAC', 'ATG', 'GCG',
'TCA', 'ATG', 'GTA', 'CTC', 'GCG', 'GAG']
```

Write a program that, based on the list *codons*, creates a dictionary, where the keys are the codons (3-letter strings) and the values are lists of numbers that indicate the positions (starting with 0) of the respective codons in the original list *codons* in the order they occur.

The program should then print the dictionary:

```
{'TCG': [0], 'ATG': [1, 5, 8], 'ATT': [2], 'TCT': [3], 'GAC':
[4], 'GCG': [6, 11], 'TCA': [7], 'GTA': [9], 'CTC': [10], 'GAG':
[12]}
```

Note that the order of the keys is irrelevant, it is only important that each key has the correct associated list of positions.
The program should be general enough that it would still work for a *codons* list of different content and length.

## Question 3

In biology, primers are short DNA sequences important for a process called PCR. One essential property of a primer is its melting temperature, Tm, which can be estimated based on the counts of the four nucleotides G, C, A and T:

$$Tm = 4 * (G_{count} + C_{count}) + 2 * (A_{count} + T_{count})$$

Write a function *meltingTemperature()* that takes a primer sequence as input and returns its estimated melting temperature.
Call the function as follows:

```
print(meltingTemperature('TCAGCTAGCTCGTAGCTACAGGC'))
print(meltingTemperature('TGAAGTGTGAATAGTACTCACGAG'))
```

The program should now print:

```
72
68
```

You may assume that the primer sequences contain only the four letters G, C, A, T.

## Question 4

The file *protoplasts.txt* contains length and width measurements [µm] of yeast cells collected at consecutive timepoints during an experimental time course. Each line in the file starts with the timepoint, followed by the length and width of a single cell, with the three values separated by commas.

The first lines from *'protoplasts.txt'* are:

```
0,9.3,4.1
0,7.5,4
…
```

Write a program that opens the file, reads the data, and calculates the length-to-width ratio for each cell.
Your program should then print the exact following lines with the deduced values:

```
time    length  width   ratio
-----------------------------
  0     9.3     4.1     2.3
  0     7.5     4.0     1.9
  0     8.8     4.5     2.0
-----------------------------
 30     7.3     4.1     1.8
 30     5.9     4.2     1.4
 30     6.0     4.5     1.3
 30     5.1     4.7     1.1
-----------------------------
120     5.7     5.2     1.1
120     6.3     4.6     1.4
120     5.6     5.4     1.0
```

Note the formatting and alignment of the different parts:

- All columns should be 8 characters wide.
- Timepoints should be right-aligned 3-digit integers.
- The length, width, and ratio for each cell should be specified with the precision of one decimal place (rounded).
- Separate the data for each timepoint with a line of 30 dashes (-).

Your program should still work if the file *protoplasts.txt* contained a different set of cells and/or more/less/different timepoints. You may assume that the current output format does not need to be adjusted in that case (e.g. no measurement 10µm or longer). You may also assume that the cell measurements in the input file are ordered chronologically.

## Question 5

```
encoded = 'rtrjsyx ufxx, zsstynhji, zsynq ymjd fwj ltsj.'
```

The message *encoded* has been encrypted using a simple substitution code where each letter in the original message was replaced by another letter according a specific pattern. This pattern is defined by two strings that contain the same letters but in a different order:

```
alphabet   = 'abcdefghijklmnopqrstuvwxyz'
key        = 'fghijklmnopqrstuvwxyzabcde'
```

For example, an 'a' in the original message would have been replaced by an 'f', a 'b' by a 'g', and so on. Any characters that are not letters of the provided alphabet (like spaces, punctuation, or numbers) were left unchanged.

Write a program to decode the string *encoded* by reversing the pattern, replacing each letter in the encoded message with its corresponding original letter from the alphabet. Ensure that your decoded message retains the original spacing and punctuation.
The program should print the decoded message:

```
moments pass, unnoticed, until they are gone.
```

Your program should be able to decode any message that was encoded using a similar letter replacement pattern. You may assume that all the letters are lower case, that the two strings *alphabet* and *key* contain the same set of characters and that only these characters present need to be decoded.

## Question 6

```
import numpy as np

binary = np.zeros(shape=(6,8),dtype=int)
binary[0,7] = 1
binary[5,3] = 1
binary[3,1:4] = 1
```

These lines create *binary*, a two-dimensional NumPy array representing a binary image where each pixel is either 0 or 1:

```
[[0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]]
```

Write a program that processes the array *binary* and creates a new array where each pixel contains the count of neighboring pixels that have a value of 1 in the original *binary* image.

Consider that:

- Non-boundary pixels have 8 neighbors (up, down, left, right and diagonals).
- Boundary pixels have fewer neighbors: corner pixels have 3 neighbors, edge pixels have 5 neighbors.

Make sure your program handles these cases correctly.

Finally, print the new array:

```
[[0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1 1]
 [1 2 3 2 1 0 0 0]
 [1 1 2 1 1 0 0 0]
 [1 2 4 3 2 0 0 0]
 [0 0 1 0 1 0 0 0]]
```

The program should be general enough to work with any binary image array of different length and/or width.

## Question 7

You are studying the diversity of the soil microbiome using a dataset containing counts of microbial species from different soil samples. Sparse matrices are often used to efficiently represent datasets where most elements are zero. One such representation is the Compressed Sparse Column (CSC) matrix, which uses three vectors to store data compactly.
Consider the following small example dataset:

| | Species 1 | Species 2 | Species 3 | Species 4 |
|---|---|---|---|---|
| Sample 1 | 1 | 15 | 1 | 0 |
| Sample 2 | 0 | 0 | 0 | 2 |
| Sample 3 | 4 | 0 | 3 | 6 |

The following three vectors represent a CSC matrix that corresponds to the example dataset (full matrix) from above:

```
data    = [1,4,15,1,3,2,6]
```
➢ contains the non-zero values of the matrix

```
indices = [0,2,0 ,0,2,1,2]
```
➢ contains row indices of non-zero elements

```
indptr  = [0,2,3,5,7]
```
(col 2, col 4 above the 2nd and 4th values; col 1, col 3 below the 1st and 3rd values)

➢ index pointer keeping track of start & end positions of slices in *data* and *indices*, for each column

➢ if there are n columns in the full matrix, *indptr* will have n+1 elements

### Task 1: Create a Full Matrix

You are given three vectors that represent a sparse matrix corresponding to a dataset with 10 samples (rows) and 9 species (columns).

```
data    = [1,4,6,1,2,2,2,1,5,1,11,3,11,2,9,7,5,11,75,1,6,103,…]
indices = [3,4,5,6,7,2,6,9,2,9,0,6,0,2,3,4,5,6,7,8,0,1,2,3,4,…]
indptr  = [0,5,8,10,12,20,30,31,33,36]
```

Write a program that creates a full matrix as a NumPy array using these vectors.
Then, print the resulting matrix:

```
[[  0   0   0  11  11   6   0   0   0]
 [  0   0   0   0   0 103   0   2   1]
 [  0   2   5   0   2   8   0   0   0]
 [  1   0   0   0   9  11   0   0   0]
 [  4   0   0   0   7  51   0   0   0]
 [  6   0   0   0   5  13   0   0   0]
 [  1   2   0   3  11  41   2   0   0]
 [  2   0   0   0  75  44   0   2   1]
 [  0   0   0   0   1  52   0   0   0]
 [  0   1   1   0   0  16   0   0   1]]
```

## Task 2: Calculate Diversity Metrics

Next, characterize the dataset using the matrix created in Task 1 by calculating the following two diversity metrics for each sample. Identify and print the highest and lowest values of these metrics and the corresponding samples. Note that the numbering of the samples should start with 1.

a. Calculate the **richness**: the total number of different species found per sample. The program should print the following:

```
highest: sample 7, richness 6
lowest: sample 9, richness 2
```

The precise formatting of the printed output is not relevant.

b. Calculate the **Shannon Diversity Index (H')** for each sample using the formula:

$$H' = -\sum_{i=0}^{R} p_i \ln(p_i)$$

Where:

- $R$ is the number of non-zero species in the sample
- $p_i$ is the relative abundance of species $i$, calculated as the count of species $i$ divided by the total count of all species in the sample
- you may use the function numpy.log() to calculate the natural logarithm $ln$

The program should print the following:

```
highest: sample 3, H' 1.2181950724180413
lowest: sample 9, H' 0.09359996324581574
```

Again, the precise formatting of the printed output is not relevant.

Your program for Tasks 1 and 2 should still work according to the same principle on a different sparse matrix of different size. You may assume that the sample size and the number of species are given. You may also assume that there is always one lowest and one highest diversity metric value.
You are not allowed to use any pre-existing modules/functions that convert sparse matrices to dense matrices.