BF550: Fall 2021
Project 1 is due by 12 pm on Monday, November 1

Submission instructions
Please submit this project as a single .py file that contains the "array" class in the assignment on blackboard.

Writing an array package
Write a package that contains an array class. The array class will consist of the data contained in the array, and several methods and attributes. The arrays should be implemented in pure python, without the use of numpy, scipy, or any other additional libraries. The package should support arrays up to 2D, as well as row and column vectors. We will use the notation $A[i, j]$ below to denote an array A indexed to the i-th row and j-th column.

Objects of this class should have several attributes:

- size: The size attribute should return the total number of entries in the array.
- shape: The shape attribute should be a tuple consisting of the number of rows followed by the number of columns, as in the NumPy specification.
- data: Contains the array structured as a nested list.

The class should also have several methods. These can be divided into "linear algebra" methods and "statistics" methods, in addition to an "`__init__`" method that takes in a 2D array from a list in the same way as NumPy does: It should take a nested list, with the elements of the outer list being rows and the elements of the inner list being the elements of those rows.

The "linear algebra" methods are the following:

- transpose: The transpose method should return a new array object with the rows and columns reversed. If the original array has rows indexed by i and columns indexed by j. so A.transpose()$[i, j] = A[j, i]$
- sum: A sum of elements in the array. If no argument is given, it should return the sum of all elements in the array. It should optionally take arguments either 0 or 1, indicating a sum along the rows or columns, respectively. In particular if array A has shape (m, n) then A.sum(0) should return an n dimensional column vector and A.sum(1) should return an m dimensional row vector.
- __add__: Defining the __add__method will overload the addition operator allowing these arrays to be added element-wise togther using the usual + operation. If $C = A + B$ then we should have $C[i, j] = A[i, j] + B[i, j]$ for all (i, j). Should support addition by scalars (single floats or ints) as well as arrays of the same shape as the given array. Other shapes or broadcasting methods are not required.

- __mul__: Same as __add__but for element-wise multiplication.
- __sub__: Same as __add__but for element-wise subtraction.
- __truediv__: Same as __add__but for element-wise division.
- __neg__: Same as __add__but for negation. Allows $-A$ to be used.
- __pow__: Element-wise exponentiation, only single number arguments are necessary to consider, but should support floats in addition to ints.
- __getitem__: This will implement object indexing of the form $A[i]$ or $A[i, j]$. The argument should be a tuple of length 2. The first argument should be taken to be the the row index and the second should be the column index. You should assume that the arrays will always be called with two indices.
- dot: Matrix dot product. If we take $C = A.dot(B)$ we should have $C[i, j] = \sum_k A[i, k]B[k, j]$. If A has shape (x, y) and B has shape (y, z) the shape of C should be (x, z).

The "statistics" methods are the following:

- mean: The mean method should return the average value of an array along a provided axis, in the same way as the sum method does, and it should call the sum method. As such, it should take an optional axis argument.
- var: The var method calculates a covariance matrix. It assumes that the different features of the data (e.g. different time points, generally different components of a single experiment) will be aligned along columns and that different experiments will be aligned along rows. So each row represents a different experiment, consisting of columns corresponding to the different features of a single experiment. Given a data array X, covariance matrix C of X is defined in the following way: $C = \frac{1}{n} X_c^T \cdot X_c$, where $X_c = X - \bar{X}$ and $\bar{X}$ is the average of X taken over different *experiments*, and has the appropriate shape.

You should be able to code the "statistics" methods using the "linear algebra" methods without directly accessing the underlying data. Many of the elementwise operations will likely reuse some amount of code.