ME 2450 Lab 01: Array 1

In this lab you will practice storing and accessing data in 1D and 2D arrays to refresh your memory on programming in Matlab or Python.

File Management and Gradescope submission:

For this class you should practice good organizational skills in file management to organize programs and subroutines you create. This will facilitate reuse of code in subsequent labs when appropriate. Additionally, remember to give all script and function files meaningful names so you can quickly identify their purpose at a later date.

It is recommended that you setup a local file system on your computer or a school computer you access to complete homework that is similar to:

Desktop >> MEEN2450 >> Lab01>>Lab01_script.py

Additional labs should be stored in additional folders.

To submit your assignments to Gradescope, you will need to create a zip folder of the lab assignment folder using the appropriate command on your OS (if you haven't done this ask for help). Your zipped folder should have a name that follows the following convention:

Lastname_FirstInitial_Lab#_MMDD_ProgrammingLanguage.zip

Example: Stoll_R_Lab01_0831_Matlab.zip

MMDD should be the month and date that your lab is due which will always be the day of the next lab.

What are Arrays:

Arrays are an ordered collection of numbers that are indexed by position and the values are the same data type (e.g. all floats). For example, exam scores could be floating-point values stored in an array named *quiz_scores*. The first student's quiz would be found in the first index of the array, the second student's score in the second index, etc.

To understand array usage and advantages, consider the following example. Suppose we want to store and average the scores for a midterm exam of 100 students. We could, for example, do that by declaring 100 variables to store the 100 floating-point values (scores):

Var_1= 95 Var_2=89
:
:
Var_100=87

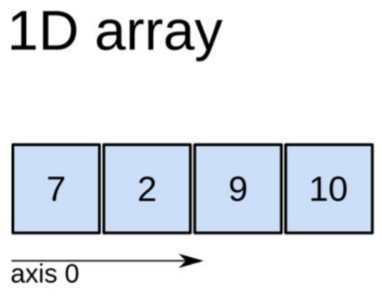
We would then need to add all these together with something like *total_sum = Var_1 + Var_2 + ... V_100*, and finally dividing *total_sum* by 100. However, this would be a very limited implementation because the code would need to be customized for other class sizes, not to mention that manually entering and summing the 100 variables would be quite time-consuming and prone to typing errors.

Instead, the implementation of the average computation should be made by using an array variable. For other class sizes (not equal to 100), the array length can be made shorter or longer as needed. After storing the exam scores, the *total_sum* can be computed using a for loop over the array's length, adding the value from each array index (at each iteration of the loop). Finally, the average would be computed by dividing *total_sum* by the array's length (number of exams).

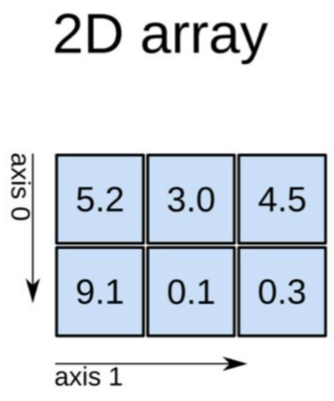
Types of Arrays:

In this class, we will focus mainly on two array dimensions:

1. 1D Array (aka Vectors)
2. 2D Array (aka Matrices)



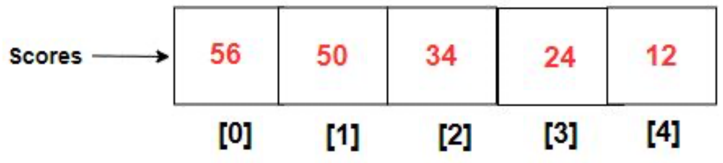
shape: (4,)

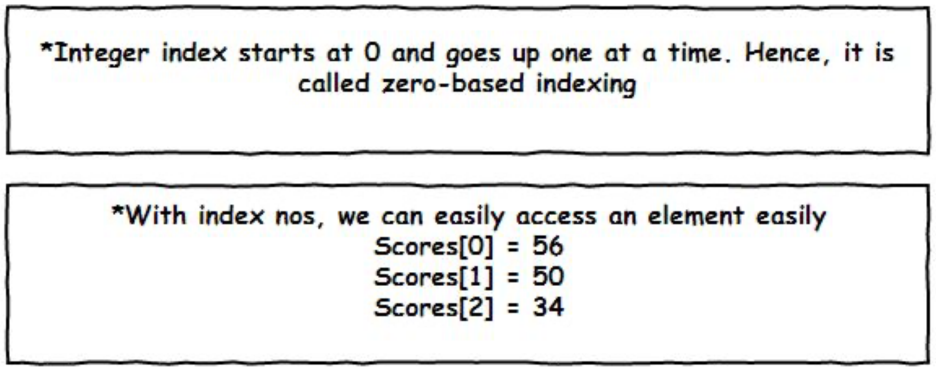


shape: (2,3)

Indexing in Arrays:

Each value in an array is stored by index (its position) in the array. Array indices can start at 0 (as with Python) or 1 (as with MatLab). From that starting position, each subsequent index is increased by a value of 1 until the end of the array. This concept is illustrated in the following images:

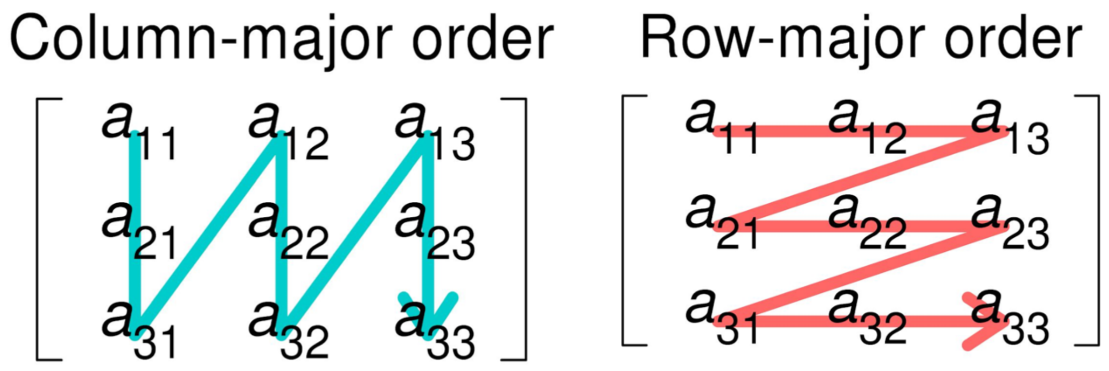



*Integer index starts at 0 and goes up one at a time. Hence, it is called zero-based indexing

*With index nos, we can easily access an element easily
Scores[0] = 56
Scores[1] = 50
Scores[2] = 34

Row-oriented vs Column-oriented Sequencing in Array:

In a 2D array, the positions are indexed as illustrated below (using indices starting at 1, as with MatLab), where the rows and columns are indexed separately. a_{23} would be written as *a(2,3)* in MatLab and *a[1,2]* in Python. (*Note: in Python indices start at zero so each value below would be 1 less.)*


Column-major order
Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Conditional operators and control statements: *if*, *else if*, *else*:

The if/else statement executes a block of code if a specified condition is true. If the condition is false, another block of code can be executed.

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

The **else** statement specifies a block of code to be executed if the condition is false:

```
if (condition) {
 // block of code to be executed if the condition is true
} else {
 // block of code to be executed if the condition is false
}
```

The **else if** statement specifies a new condition if the first condition is false:

```
if (condition1) {
 // block of code to be executed if condition1 is true
} else if (condition2) {
 // block of code to be executed if condition1 is false and condition2 is true
} else {
 // block of code to be executed if condition1 is false and condition2 is false
}
```

for loops:

A **for** loop is used to repeat a block of code a known number of times. For example, if we want to check the grade of every student in the class, a **for** loop over the indices of the array of grades would be used. When the number of times is not known beforehand, a **while** loop is used instead.

```
for (iterations) {
 // block of code to be executed at each iteration
}
```

Arrays and *for* loops (used together):

Defining, accessing, and operating on values in arrays is often done using **for** or **while** loops. For example, a 1D array can be initialized to zeros using the following pseudocode:

```
zeros = []
for index in number of values:
  zeros[index] = 0
```

Similarly, a 2D array can be initialized to zeros using the following pseudocode:

```
zeros = []
  for row_index in number of rows:
    for col_index in number of cols:
      zeros[row_index][col_index] = 0
```

Lab Exercises:

Write Python (using the numpy module) or MatLab (or other syntax, if desired) codes to complete the following tasks:

❏ (2 Points) Create an N x 1 (1D) array of integers that counts in sequence from 1 to N.
- ❏ Input: N=9
- ❏ Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

❏ (3 Points) Create an N x M (2D) array of integers that counts in sequence from 1 to N*M, row-oriented.
- ❏ Input: N=3, M=3
- ❏ Output: [1, 2, 3; 4, 5, 6; 7, 8, 9] (Note, semicolons indicate a new row)

❏ (3 Points) Create an N x M (2D) array of integers that counts in sequence from 1 to N*M, column-oriented.
- ❏ Input: N=3, M=3
- ❏ Output: [1, 4, 7; 2, 5, 8; 3, 6, 9] (Note, semicolons indicate a new row)

❏ (4 Points) Include the above code in a separate function that can be called. Inputs are: N, M, and a boolean option for row or column orientation.
- ❏ For example, *create_array(N, M, row_orient=True/False)*

❏ (2 Points) Utilize the *create_array* f unction to define and access data in a 1D array
- ❏ Input: N=3, M=1
- ❏ Outputs:
  - ❏ value at first index (i.e. 1)
  - ❏ value at last index (i.e. 3)

- ❑ (5 Points) Utilize the *create_array* f unction to define and access data in a 2D array
    - ❑ Input: N=3, M=3
    - ❑ Outputs:
        - ❑ value at the first row, first column (i.e. 1)
        - ❑ value at the last row, last column (i.e. 9)
        - ❑ values from the entire first row (i.e. 1, 2, 3)
        - ❑ values from the entire last column (i.e. 3, 6, 9)
        - ❑ values from the diagonal (i.e. 1, 5, 9)

References:

1. https://www.mathworks.com/help/matlab/matrices-and-arrays.html 2.
http://cs231n.github.io/python-numpy-tutorial/#numpy