CHAPTER

V

Week 3 - Assignment 2A

In this chapter:
Solutions: Assignment 1
Warming up exercises
Mini project 3.
Submission instructions

Solutions: Assignment 1

Download the file **`mini_proj_1_solutions.py`** and **`mini_proj_2_solutions.py`** in Brightspace and check the **proposed solutions** to the mini projects in Assignment 1. The **individual feedback** for those problems were already provided in the form of the self tests. You can try to understand the questions that you didn't manage to solve.
Some **general feedback** regarding the most common mistakes made on each question will be posted later.

proposed solution: there are multiple ways of solving a problem. Maybe your solution is not identical but it's still correct. The proposed solution focus on solving the problems using only the tools that we have learned so far in the course.

general feedback: the general feedback is created using a random sample of the submitted assignments.

individual feedback: you can use the self testing scripts to get individual feedback on your solutions. They are not perfect, but might indicate that something is wrong. Use that together with the proposed solutions to understand what went wrong.

Warming up exercises

Use the **exercises** below to practice and consolidate your skills before going to the mini-project.

Warming up exercises help you to consolidate concepts separately, by working them in smaller, simpler problems.

Exercise 8. The code below creates an array x with values from 0 to 30 and steps of 0.1. Suppose you want to evaluate the mathematical function *f(x) = x2̂ + 2*x + 3* for every value of the vector x. Write a function that receives x and returns a vector with *f(x)*.

```
x = np.arange(0,30,0.1)
```

Use pyplot to plot f(x) and x.

Exercise 9. Implement a function swap_idx that receives an array x, and two integers i and j as parameters. Your function should swap the values of x at the indices i and j, if this operation is possible. Check if those are valid indices for x before doing the swap.

Exercise 10. Implement a function sum_array that receives an array of integers x, and returns the sum of all the even elements in x.
You can test if your function works with the following array:

```
x = np.arange(0,100,3) # similar to range(), but creates ↩
    an array
```

Exercise 11. Write a function that receives a 1D-array x and create a new array y, swap the first and second half of x. Ex.: the array [0, 1, 2, 3, 4, 5] becomes [3, 4, 5, 0, 1, 2]. If the number of elements is odd, the middle element is repeated.

Mini-Project 3 *Think-Pair-Share*

In this assignment, you will practice what you have learned about arrays. For simplicity, we will only use **numpy arrays** and all the information here refers to that specific type of array.

Download the template `mini_proj_3.py` in Brightspace to implement the following functions concerning arrays.

The **numpy** module is often imported with the alias `np`, which we also use in this document.

1-Dimensional arrays

In the first part of this assignment you will create and manipulate **1-dimensional numpy arrays**.

Implement the following functions to operate with 1D arrays:

1-dimensional numpy arrays: an array has 1 dimension if their shape has only one element. That is, *len(x.shape)* is 1, for the array x. Notice you can have a 2d array with only 1 row/column that looks like 1d. In that sense, *numpy.zeros((10,1))* is a 2d array, while *numpy.zeros((10,))* is a 1d array.

Remark. Using numpy and built-in functions.
Some of those functions have a very simple solution if you use numpy/built-in functions, which is not the goal of the exercise. I specify in each question things that you should not use in your solutions. You can always use: np.zeros, np.ones, np.where, slicing/indexing and the array shape.

- **random_array(a: int, b: int, n: int)**: That uses the function *randint* from module random to create an 1d-array of integers of size *n*. The values of the array should be randomly sampled between a and b (including both a and b).
  - Your function should deal with the case in which a and b are swapped, correcting so that a is the lower value.
  - The function should return the created array, and the type of the array should be int32.
- **element_mult(x: numpy.ndarray, y: numpy.ndarray)**: implement a function called that receive two 1-dimensional arrays *x* and *y* and returns a new array z with the element-wise multiplication of x and y (that is, the ith element of z is the multiplication of the ith elements of x and y).
  - If x and y have different lengths, your function should return None.
  - The array z should inherit the type of either x and/or y (ex.: If x and y are int, z is also int. If x is float and y is int, z is float).

  Remark. Compute each value of z individually, do not use the standard multiplication (ex: x*y) to solve this. Your function should loop over the individual values in the arrays.
- **find_max(x: numpy.ndarray)**: implement a function that receives an array x (of positive numbers) and returns an integer with the index of the maximum value in the array.

  Remark. Do not use any numpy or built-in **max** functions, you should use loops and conditions to find the maximum value.
  - In case of a tie, your function should return the index of the last occurring maximum value.
  - If there are negative values in the array the function should return None.

(Numpy) **Array operations and methods**:

- `my_array[i]`: access the element at index `i` at array `my_array` (indexing).
- `my_array.shape`: returns a tuple with the size of each dimension of the array.
- `np.zeros(t)` and `np.ones(t)`: return an array of zeros or ones (respectively) with the shape of the tuple `t`.
- `np.arange()`: array version of `range()`.
- `np.asarray(my_list)`: converts a list into an array.

(Check other array operations in the next page).

Testing your functions: the *main()* function of the script creates an array and plot it as a time-series, highlighting the maximum value. You can use that code to test your functions *random_array()* and *find_max()* once they are ready.

N-Dimensional arrays

Arrays can also have more than 1 dimension. A 2D array, often called a matrix, can be used, for example,to represent a system of equations and help us find it's solution. Also, some matrices have interesting properties. For example, square matrices (same number of rows and columns) always have an inverse matrix. Another example is when a 2D is a 'magic square'. That is, when the sum of every column, row and diagonal is a constant number.

Write the following functions that create and manipulate 2D matrices (and sometimes N-dimensional matrices).

- **transpose(x: numpy.ndarray)**: write a function that returns a new array y that is the transpose of the 2D array x. In other words, the value of x[i,j] should be equal to y[j,i] for every possible i and j. If the shape of x is (10,2), the shape of y should be (2,10).
  - The type of elements in y should be the same of elements in x.

  Remark. You should implement your own function, do not use x.T or numpy.transpose().

- **is_square(x: numpy.ndarray)**: receives an array x and checks if all the dimensions of this array have the same length. The function returns True if so, and False otherwise.
  - Your function should generalize for more than 2 dimensions. If all the dimensions of a N-dimensional array have the same size, the function should return True.
  - If the function receives a 1D array with only one element, it should also return True, and False if the 1D array have more than 1 element.

- **is_magic(x: numpy.ndarray)**: receives an array x and checks if this is a magic square. In a magic square, the sum of every column, row and the 2 diagonals are equal. If that is the case, the function should return True. It returns False otherwise. An example of a magic matrix is also given in *main()*.

  Remark. Compute the sums and diagonal manually, that is, without using numpy or built-in functions.

  - If the array has more or less than 2 dimensions the function should return None.
  - If the array is not a square matrix the function should return None.

Operations with more than one array:

- `mathematical operations (+, -, *, /)` and comparisons (>,<,==,!=) between arrays of the same shape: the operation/comparison is performed element by element and returned as an array (of the same shape).
- `np.matmult(a,b)`: compute the matrix multiplication between arrays a and b.
- `np.logical_and(a, b)` and `np.logical_or(a, b)`: performs a and and or logical operations element by element of the (numpy) arrays a and b. The result is a logical array (of the same shape of a and b).

Other **array manipulations and methods**:

`np.where(condition)`: returns the indices in which the array condition is True. (ex: `condition = my_array>10`)

`np.max(my_array, axis=i)`: returns an array with the maximum elements across the axis `i`.

- `np.argmax(my_array, axis=i)`: returns an array with the indices of the maximum elements across the axis `i`.
- `list(my_array)`: converts an array into a list.
- `my_array.sort()`: sort the array `my_array`.

Submission instructions

Extra documents:
If you have worked together with another student, create a .txt file called `programming_partner.txt` containing the S-number of both of you.

Assignment 2A is part of Assignment 2, which has its deadline after Week 5. Submit your final solutions in Brightspace.

Example. `programming_partner.txt`

```
Mini project 3:
s012345
s123456

```

Both students should submit their work individually on Brightspace.

General submission:
Create a zip file called `Assignment2` containing the files:

- `mini_proj_3.py`, with the following functions:
  - random_array()
  - element_mult()
  - find_max()
  - transpose()
  - is_magic()
  - is_square()
- `programming_partner.txt`

Submit your file to Assignment 2, in Brightspace.