P2_Numpy_basics_professor

September 19, 2019

1 Exercises about Numpy

```
Notebook version:
   * 1.0 (Mar 15, 2016) - First version - UTAD version
   * 1.1 (Sep 12, 2017) - Python3 compatible
   * 1.2 (Sep 3, 2018) - Adapted to TMDE (only numpy exercises)
   * 1.3 (Sep 4, 2019) - Spelling and structure revision.

Authors: Jerónimo Arenas García (jeronimo.arenas@uc3m.es),
         Jesús Cid Sueiro (jcid@tsc.uc3m.es),
         Vanessa Gómez Verdejo (vanessa@tsc.uc3m.es),
         Óscar García Hinde (oghinnde@tsc.uc3m.es),
         Simón Roca Sotelo (sroca@tsc.uc3m.es)
```

This notebook is an introduction to the Numpy library. Numpy adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. Here we will learn the basics on how to work with numpy arrays, and some of the most common operations which are needed when working with these data structures.

2 1. Importing numpy

[0]:
```
# Import numpy library

import numpy as np
```

3 2. Numpy exercises

3.1 2.1. Create numpy arrays

The following code fragment defines variable x as a list of 4 integers, you can check that by printing the type of any element of x.

[0]:
```
x = [5, 4, 3, 4]
print(type(x[0]))
```

If you want to apply a transformation over each element of this list you have to build a loop and operate over each element.

Exercise 1: *Complete the following code to create a new list with the same elements as x, but where each element of the list is a float (you can use the `float()` function).*

[0]:
```
# Create a list of floats containing the same elements as in x

x_f = []
for element in x:
  # <FILL IN>
  x_f.append(float(element))

print(x_f)
print(type(x_f[0]))
```

The output should be:

```
[5.0, 4.0, 3.0, 4.0]
<class 'float'>
```

Numpy is a python library that lets you work with data vectors and matrices (we will call them numpy arrays) and directly apply operations over these arrays without the need to operate element by element.

Numpy arrays can be defined directly using methods such as `np.arange()`, `np.ones()`, `np.zeros()`, `np.eye()`, as well as random number generators. Alternatively, you can easily generate them from python lists (or lists of lists) containing elements of a numerical type by using `np.array(my_list)`.

You can easily check the shape of any numpy array with the property `.shape`.

[0]:
```
# Numpy arrays can be created from numeric lists or using different numpy methods
y = np.arange(8) + 1
x = np.array(x_f)
z = np.array([[1, 2], [2, 1]])

# The arange() function generates vectors of equally spaced numbers. We can
# specify start and stop positions as well as the step length (the steps don't
# need to be integers!):
print('A vector that goes from 2 to 8 in steps of 2: ', np.arange(2, 9, 2))

# Numpy also has a linspace() function that works exactly like its Matlab
# counterpart:
print('\nA vector of length 5 that spans from 0 to 1 in constant increments:\n',
      np.linspace(0, 1, 5))

# Check the different data types involved
print('\nThe type of variable x_f is ', type(x_f))
print('The type of variable x is ', type(x))

# Print the shapes of the numpy arrays
print('\nThe variable x has shape ', x.shape)
```

```
print('The variable y has shape ', y.shape)
print('The variable z has shape ', z.shape)
```

Note: Compare the shape of x and y with the shape of z and note the difference between 1-D and N-D numpy arrays (ndarrays). We will later review this issue in detail.

We can also convert a numpy array or matrix into a python list with the method `np.tolist()`.

[0]:
```
my_array = np.arange(9).reshape((3, 3))
print(my_array)
print('the type is: ', type(my_array))

# Convert my_array to list

my_array_list =  my_array.tolist()
print('\n', my_array_list)
print('the type is: ',type(my_array_list))
```

Exercise 2: *Complete the following exercises:*

[0]:
```
# 1. Define a new 3x2 array named my_array with [1, 2, 3] in the first row and
# [4, 5, 6] in the second. Check the dimension of the array.
# my_array = <FILL IN>
my_array = np.array([[1, 2, 3],[4, 5, 6]])
print(my_array)
print('Its shape is: ', np.shape(my_array))

#2. Define a new 3x4 array named my_zeros with all its elements to zero
# my_zeros = <FILL IN>
my_zeros = np.zeros((3,4))
print('A 3x4 vector of zeros:')
print(my_zeros)

#3. Define a new 4x2 array named my_ones with all its elements to one
# my_ones = <FILL IN>
my_ones = np.ones((4,2))
print('A 4x2 vector of ones:')
print(my_ones)

#4. Modify the dimensions of my_ones to a 2x4 array using command np.reshape()
# my_ones2 = <FILL IN>
my_ones2 = my_ones.reshape((2,4))
print('A 2x4 vector of ones:')
print(my_ones2)

#5. Define a new 4x4 identity array named my_eye
# my_ones2 = <FILL IN>
my_ones2 = np.eye((4))
print('A 4x4 identity vector:')
print(my_ones2)
```

The output should be:

```
[[1 2 3]
 [4 5 6]]
Its shape is:  (2, 3)

A 3x4 vector of zeros:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
A 4x2 vector of ones:
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
A 2x4 vector of ones:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

3.2 2.2 Numpy operations

We can perform all the usual numerical and matrix operations with numpy. In the case of matrix addition and subtraction, we can use the common "+" or "-" operators:

[0]:
```
x1 = np.arange(9).reshape((3, 3))
x2 = np.ones((3, 3))
result = x1 + x2
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nAddition of x1 and x2 using the + operator:\n', result)
```

However, numpy provides us with built-in functions that guarantee that any errors and exceptions are handled properly:

[0]:
```
# We can add two arrays:
x1 = np.arange(9).reshape((3, 3))
x2 = np.ones((3, 3))
result = np.add(x1, x2)
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nAddition of x1 and x2 using built-in functions:\n', result)

# Or compute the difference:
result = np.subtract(x1, x2)
print('\nSubtraction of x1 and x2 using built-in functions:\n', result)
```

Whether you use the basic operators or the built-in functions will depend on the situation.

We can also add or subtract column or row vectors from arrays. Again, both the basic operators and the built in functions will perform the same operations. Unlike in Matlab, where this operation will raise an error, Python will automatically execute it row by row or column by column as appropriate:

[0]:
```
# We can add or subtract row and column vectors:
row_vect = np.ones((1, 3))
col_vect = np.ones((3, 1))
result = np.add(x1, row_vect)
print('x1:\n', x1, '\n\nrow_vect:\n', row_vect, '\n\ncol_vect:\n', col_vect)
print('\nAddition of a row vector:\n', result)
result = np.add(x1, col_vect)
print('\nAddition of a column vector:\n', result)
```

Another key difference with Matlab is that the "*" operator won't give us matrix multiplication. It will instead compute an element-wise multiplication. Again, numpy has a built-in function for this purpose that will guarantee propper handling of errors:

[0]:
```
# We can perform element-wise multiplication by using the * operator:
x1 = np.arange(9).reshape((3, 3))
x2 = np.ones((3, 3)) * 2 # a 3x3 array with 2s in every cell
result = x1 * x2
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nElement-wise multiplication of x1 and x2 using the * operator:\n',␣
 ↪result)

# or by using the built-in numpy function:
result = np.multiply(x1, x2)
print('\nElement-wise multiplication of x1 and x2 using built-in functions:\n',␣
 ↪result)
```

Numpy also gives us funtions to perform matrix multiplications and dot products.

[0]:
```
# We can perform matrix multiplication:
x1 = np.arange(9).reshape((3, 3))
x2 = np.ones((3, 3)) * 2 # a 3x3 array with 2s in every cell
result = np.matmul(x1, x2)
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nProduct of x1 and x2:\n', result)

# Or the dot product between vectors:
v1 = np.arange(4)
v2 = np.arange(3, 7)
result = np.dot(v1, v2)
print('\nv1:\n', v1, '\n\nv2:\n', v2)
print('\nDot product of v1 and v2:\n', result)
```

Note that the `np.dot()` function is very powerful and can perform a number of different operations depending on the nature of the input arguments. For example, if we give it a pair of matrices of adequate dimensions, it will perform the same operation as `np.matmul()`.

[0]:
```
x1 = np.arange(9).reshape((3, 3))
x2 = np.ones((3, 3)) * 2 # a 3x3 array with 2s in every cell
result = np.matmul(x1, x2)
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nProduct of x1 and x2 using np.matmul():\n', result)
```

```
result = np.dot(x1, x2)
print('\nProduct of x1 and x2 using np.dot():\n', result, '\n')

# Read the np.dot() documentation for more information:
help(np.dot)
```

We can also compute typical numerical operations, which will be applied element-wise:

[0]:
```
# Examples of element-wise numerical operations:
x1 = np.arange(9).reshape((3, 3)) + 1
print('x1:\n', x1)
print('\nExponentiation of x1:\n', np.exp(x1))
print('\nLogarithm of x1:\n', np.log(x1))
print('\nSquare root of x1:\n', np.sqrt(x1))
```

Element-wise division between matrices is performed using the "/" operator or the `divide()` built-in function:

[0]:
```
# Element-wise division of two matrices:
x1 = np.arange(9).reshape((3, 3)) * 3
x2 = np.ones((3, 3)) * 3
result = np.divide(x1, x2)
print('x1:\n', x1, '\n\nx2:\n', x2)
print('\nElement-wise division of x1 and x2:\n', result)
```

We can use the "**" operator or the `power()` built-in function to raise elements from a matrix to a given power, or to raise elements of one matrix to positionally-corresponding powers in another matrix:

[0]:
```
# Performing power operations with the ** operator:
x1 = np.arange(9).reshape((3, 3))
result = x1**2
print('x1:\n', x1)
print('\nRaising all elements in x1 to the power of 2 using the ** operator:\n',␣
 ↪result)

result = x1**x1
print('\nRaising all elements in x1 to themselves using the ** operator:\n',␣
 ↪result)

# Performing power operations with the power() function:
result = np.power(x1, 2)
print('\nRaising all elements in x1 to the power of 2 using the power() function:
 ↪\n', result)

result = np.power(x1, x1)
print('\nRaising all elements in x1 to themselves using the power() function:
 ↪\n', result)
```

Finally, we can transpose a matrix by using the `numpy.transpose()` function, the `ndarray.transpose()` method or its abbreviated version, `ndarray.T`. We usually use the abbrevi-

ated version, but the other forms have their place in certain contexts. Check their documentations to see what options they offer:

[0]:
```
# Three different ways of transposing a matrix:
x1 = np.arange(9).reshape((3, 3))
print('x1:\n', x1)
print('\nTranspose of x1 using the numpy function:\n', np.transpose(x1))
print('\nTranspose of x1 using the ndarray method:\n', x1.transpose())
print('\nTranspose of x1 using the abbreviated form:\n', x1.T)
print('\nOddly enough, the three methods produce the same result!')
```

Exercise 3: *In the next cell you'll find a few exercises for you to practice these operations.*

[0]:
```
# Complete the following exercises. Print the partial results to visualize them.

# Create a 3x4 array called `y`. It's up to you to decide what it contains.
#y = <FILL IN>
y = np.arange(12).reshape((3, 4))

# Create a column vector of length 3 called `x_col`.
#x_col = <FILL IN>
x_col = np.ones((3, 1)) * 2

# Multiply the 2-D array `y` by 2
#y_by2 = <FILL IN>
y_by2 = y * 2

# Multiply each of the columns in `y` by the column vector `x_col`
#z = <FILL IN>
z = x_col * y

# Obtain the matrix product of the transpose of x_col and y
#x_by_y = <FILL IN>
x_by_y = x_col.T.dot(y)

# Compute the sine of a vector that spans from -5 to 5 in increments of 0.5
#x = <FILL IN>
x = np.arange(-5, 5.5, 0.5)
#x_sin = <FILL IN>
x_sin = np.sin(x)
```

3.3 2.2. N-D numpy arrays

To correctly operate with numpy arrays we have to be aware of their dimensions. Are these two arrays equal?

[0]:
```
array1 = np.array([1,1,1])
print('array1:\n', array1)

array2 = np.ones((3,1))
```

```
print('\narray2:\n', array2)
```

The answer is **no**. We can easily check this by examining their shapes and dimensions:

[0]:
```
print('Shape of array1 :',array1.shape)
print('Number of dimensions of array1 :',array1.ndim)
print('Shape of array2 :',array2.shape)
print('Number of dimensions of array2 :',array2.ndim)
```

Effectively, `array1` is a 1D array, whereas `array2` is a 2D array. There are some methods that will let you modify the dimensions of an array. To go from a 2-D to 1-D array we have the methods `flatten()`, `ravel()` and `reshape()`. Check the result of the following code (you can use the help function to check the funcionalities of each method).

[0]:
```
x1 = np.arange(9).reshape((3, 3))
print('x1:\n', x1)
print('Its shape is: ', x1.shape)

print('\n Use the method flaten:')
print('x1.flatten(): ', x1.flatten())
print('Its shape is: ', x1.flatten().shape)

print('\n Use the method ravel:')
print('x1.ravel(): ', x1.ravel())
print('Its shape is:', x1.ravel().shape)

print('\n Use the method shape:')
print('x1.reshape(-1): ', x1.reshape(-1))
print('Its shape is: ', x1.reshape(-1).shape)


# Note that here the method shape is used to reorganize the array into a 1-D
# array. A more common use of reshape() is to simply redimension an array from
# shape (i, j) to shape (i', j') satisfying the condition i*j = i'*j'.
# For example:
print('\n A more common use of reshape():')
x1 = np.arange(12).reshape((4, 3))
print('x1:\n', x1)
print('Its shape is: ', x1.shape)

print('\nx1.reshape((2,6)):\n', x1.reshape((2,6)))
print('Its shape is: ', x1.reshape((2,6)).shape)
```

Note: `flatten()` always returns a copy of the original vector, whereas `ravel()` and `shape()` returns a view of the original array whenever possible.

Sometimes we need to add a new dimension to an array, for example to turn a 1-D array into a 2-D column vector. For this we use `np.newaxis`.

[0]:
```
# Let's start with a 1-D array:
array1 = np.array([1,1,1])
print('1D array:\n',array1)
```

```
print('Its shape is: ', array1.shape)

# Let's turn it into a column vector (2-D array with dimension 1x3):
array2 = array1[:,np.newaxis]
print('\n2D array:\n',array2)
print('Its shape is: ', array2.shape)

# Let's turn it into a row vector (2-D array with dimension 3x1):
array3 = array1[np.newaxis,:]
print('\n2D array:\n',array3)
print('Its shape is: ', array3.shape)
```

We might also need to remove empty or unused dimensions. For this we have `np.squeeze()`:

[0]:
```
array1_1D = np.squeeze(array1)
print('1D array:\n',array1_1D)
print('Its shape is: ', array1_1D.shape)

array2_1D = np.squeeze(array2)
print('\n1D array:\n',array2_1D)
print('Its shape is: ', array2_1D.shape)

array3_1D = np.squeeze(array3)
print('\n1D array:\n',array3_1D)
print('Its shape is: ', array3_1D.shape)
```

Exercise 4: *Complete the following excercise:*

[0]:
```
# Given the following matrix and vector:
vect = np.arange(3)[:, np.newaxis]
mat = np.arange(9).reshape((3, 3))

# Apply the necessary transformation to vect so that you can perform the matrix
# multiplication np.matmul(vect, mat)

# vect = <FILL IN>
vect = np.squeeze(vect)

print(np.matmul(vect, mat))
```

The output should be:
`'[15 18 21]`

3.4 2.3. Numpy methods that can be carried out along different dimensions

Compare the result of the following commands:

[0]:
```
x1 = np.arange(24).reshape((8, 3))

print(x1.shape)
print(np.mean(x1))
```

```
print(np.mean(x1,axis=0))
print(np.mean(x1,axis=1))
```

Other numpy methods where you can specify the axis along with a certain operation should be carried out are:

- `np.median()`
- `np.std()`
- `np.var()`
- `np.percentile()`
- `np.sort()`
- `np.argsort()`

If the axis argument is not provided, the array is flattened before carriying out the corresponding operation.

Exercise 5: *Complete the following exercises:*

[0]:
```
# Given the following list of heights:
heights = [1.60, 1.85, 1.68, 1.90, 1.78, 1.58, 1.62, 1.60, 1.70, 1.56]

# 1. Obtain a 2x5 array, called `h_array`, using the methods you learnt above.

# h_array = <FILL IN>
h_array = np.array(heights).reshape((2,5))
print('h_array: \n',h_array)
print('Its shape is: \n', h_array.shape)

# 2. Use method mean() to get the mean of each column, and the mean of each row.
# Store them in two vectors, named `mean_column` and `mean_row` respectively.

#mean_column = <FILL IN>
mean_column = np.mean(h_array,axis=0)
#mean_row = <FILL IN>
mean_row = np.mean(h_array,axis=1)

print('\nMean of each column: \n',mean_column)
print('Its shape is (it must coincide with number of columns):\n', mean_column.
 ↪shape)

print('\nMean of each row: \n',mean_row)
print('Its shape is (it must coincide with number of rows):\n', mean_row.shape)

# 3. Obtain a 5x2 array by multiplying the mean vectors. You may need to create␣
 ↪a
# new axis. The array name should be `new_array`

#new_array = <FILL IN>
new_array = mean_column[:,np.newaxis].dot(mean_row[:,np.newaxis].T)
print('\nNew array: \n',new_array)
```

```
print('New array shape: \n', new_array.shape)
```

The output should be: "' h_array: [[1.6 1.85 1.68 1.9 1.78][1.58 1.62 1.6 1.7 1.56]] Its shape is: (2, 5)

Mean of each column: [1.59 1.735 1.64 1.8 1.67] Its shape is (it must coincide with number of columns): (5,)

Mean of each row: [1.762 1.612] Its shape is (it must coincide with number of rows): (2,)

New array: [[2.80158 2.56308][3.05707 2.79682] [2.88968 2.64368][3.1716 2.9016] [2.94254 2.69204]] New array shape: (5, 2)"'

3.5 2.4. Concatenating arrays

Provided that the corresponding dimensions fit, horizontal and vertical stacking of matrices can be carried out with methods `np.hstack()` and `np.vstack()`.

Exercise 6: *Complete the following exercises to practice matrix concatenation:*

[0]:
```
my_array = np.array([[1, -1, 3, 3],[2, 2, 4, 6]])
print('Array 1:')
print(my_array)
print(my_array.shape)

my_array2 = np.ones((2,3))
print('Array 2:')
print(my_array2)
print(my_array2.shape)


# Vertically stack matrix my_array with itself
#ex1_res = <FILL IN>
ex1_res = np.vstack((my_array,my_array))
print('Vertically stack:')
print(ex1_res)

# Horizontally stack matrix my_array and my_array2
#ex2_res = <FILL IN>
ex2_res = np.hstack((my_array,my_array2))
print('Horizontally stack:')
print(ex2_res)

# Transpose the vector `my_array`, and then stack a ones vector
#ăas the first column. Alternatively, you can stack a row, and then transpose.
# Just make sure that the final shape is (4,3). Name it `expanded`:

#ones_v = <FILL_IN>
ones_v = np.ones((4,1))
#expanded = <FILL IN>
expanded = np.hstack((ones_v,my_array.T))
```

```
print('Expanded array: \n',expanded)
print('Its shape is: \n', expanded.shape)
```

The output should be:

```
Array 1: [[ 1 -1  3  3]  [ 2  2  4  6]] (2, 4) Array 2: [[1. 1. 1.]  [1. 1. 1.]]
(2, 3) Vertically stack: [[ 1 -1  3  3]  [ 2  2  4  6]  [ 1 -1  3  3]  [ 2  2  4
6]] Horizontally stack: [[ 1. -1.  3.  3.  1.  1.  1.]  [ 2.  2.  4.  6.  1.  1.
1.]] Expanded array:   [[ 1.  1.  2.]  [ 1. -1.  2.]  [ 1.  3.  4.]  [ 1.  3.  6.]]
Its shape is:   (4, 3)
```

3.6 2.5. Slicing

In numpy, slicing means selecting and/or accessing specific array rows and columns.

Particular elements of numpy arrays (both unidimensional and multidimensional) can be accessed using standard python slicing. When working with multidimensional arrays, slicing can be carried out along several different dimensions at once.

Let's look at some examples:

[0]:
```
# Selecting specific elements from a vector:
vect = np.arange(10)+4
new_vect = vect[[1, 3, 7]] # remember that in python arrays start at 0
print('vect:\n', vect)
print('\nSelecting specific elements:\n', new_vect)

# Selecting a range of elements from a vector:
new_vect = vect[3:7]
print('\nSelecting a range of elements:\n', new_vect)

# Selecting a subarray from an array:
array = np.arange(12).reshape((3, 4))
new_array = array[2, 2:4]
print('\nArray:\n', array)
print('\nSelecting a subarray:\n', new_array)
```

Exercise 7: *Complete the following excersises:*

[0]:
```
X = np.arange(0,25).reshape((5,5))
print('X:\n',X)

# 1. Keep last row of matrix X
#X_sub1 = <FILL IN>
X_sub1 = X[-1,]
print('\nX_sub1: \n',X_sub1)


# 2. Keep first column of the three first rows of X
#X_sub2 = <FILL IN>
X_sub2 = X[:3,0]
print('\nX_sub2: \n',X_sub2)
```

```
# 3. Keep first two columns of the three first rows of X
#X_sub3 = <FILL IN>
X_sub3 = X[:3,:2]
print('\nX_sub3: \n',X_sub3)

# 4. Invert the order of the rows of X
#X_sub4 = <FILL IN>
X_sub4 = X[::-1,:]
print('\nX_sub4: \n',X_sub4)

# 5. Keep odd columns (first, third...) of X
#X_sub5 = <FILL IN>
X_sub5 = X[:,::2]
print('\nX_sub5: \n',X_sub5)
```

The output should be: "' X: [[0 1 2 3 4][5 6 7 8 9] [10 11 12 13 14][15 16 17 18 19] [20 21 22 23 24]]

X_sub1: [20 21 22 23 24]

X_sub2: [0 5 10]

X_sub3: [[0 1][5 6] [10 11]]

X_sub4: [[20 21 22 23 24][15 16 17 18 19] [10 11 12 13 14][5 6 7 8 9] [0 1 2 3 4]]

X_sub5: [[0 2 4][5 7 9] [10 12 14][15 17 19] [20 22 24]] "'

We have seen how slicing allows us to index over the different dimensions of a given array. In the previous examples we learned how to select the rows and columns we're interested in, but how can we select only the elements of an array that meet a specific condition?

Numpy provides us with the method `np.where(condition)`. A common way of using this function is by setting a condition involving an array. For example, the condition x > 5 will give us the indexes in which x contains numbers higher than 5.

[0]:
```
x = np.array([-3,-2,-1,0,1,2,3])

# Create a new vector `y` with the elements of x, but replacing by 0 each number␣
 ↪whose
# absolute value is 2 or less.

y = np.copy(x) # CAUTION! Doing y = x will create two pointers to the same array.
condition = np.abs(x)<=2
y[np.where(condition)[0]]=0
# Note that np.where() returns a tuple. In this case the second element of the
# tuple is empty. Read the np.where() docstring for more info.

print('Before conditioning: \n',x)
print('\nAfter conditioning: \n',y)
```

Exercise 8: *Given array* `x` *below, select the subarray formed by the columns of x that add more than 4.:*

[0]:
```
#
```

```python
x = np.arange(12).reshape((3, 4))

# Write your code here
# <SOL>
idx = np.where(np.sum(x, axis=0) > 4)[0]
x_new = x[:, idx]
# </SOL>

print(x_new)
```

The output should be: `[[ 2  3]  [ 6  7]  [10 11]]`