# Dynamic Modelling Course - TEP4290: Warm-up 8

The point of this exercise is for you to practice what you have learned in the recommended video:

- https://www.youtube.com/watch?v=GB9ByFAIAH4 (especially between time codes 11:08 and 23:14)
- Numpy cheat sheet (also on blackboard): https://d20ohkaloyme4g.cloudfront.net/img/document_thumbnails/dc631e418f3609e00521

In the last task, you are required to use the **np.einsum()** method. This task is going a bit further but this method will prove very useful during the project. You can have a look at the documentation of this function here https://ajcr.net/Basic-guide-to-einsum/ but do not get scared of it yet, as we will have a look at it again later during the semester.

Good luck!

# About NumPy

NumPy stands for 'Numerical Python' and is a open source standard library for Python, and one of the most popular ones (together with Pandas, SciPy, Matplotlib).

It is centered around the *ndarray* data structure, and is the first choice for mathematical work on matrix (like) objects - which is why in industrial ecology it is quite essential for LCA, IOA and MFA.

#### Indexing in Python

Since you will work with indices in this assignment, it is important to remember that Python indexes arrays (and anything else really) starting with zero.

The list l = ['one', 'two', 'three'] therfore has 'one' at place 0 ( l[0] returns 'one'), 'two' at place 1 (l[1] returns 'two') and no entry exists for l[3].

# Tasks

Complete the tasks outlined below to achieve the same output as you find in the original file. Use what you learned in the video or find a common

## Import the package

To use the functionalities offered by the NumPy package, you must first load it. Usually, NumPy is imported under the alias **np**.

Import the NumPy library under the alias "**np**".

In [1]:
```
import numpy as np
```

## Mastering the basics

The main objects in the NumPy library are called **arrays**.

Create and display a one-dimension array containing three numbers of your choice.

In [3]:
```
a = np.array([0, 0, 7])
a
```

Out[3]:
```
array([0, 0, 7])
```

Create and display a two-dimension array containing six numbers of your choice.

In [4]:
```
b = np.array([[1, 2, 3], [4, 5, 6]])
b
```

Out[4]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Retrieve and display the size of the array you just created (number of rows and columns).

In [6]:
```
b.shape
```

Out[6]:
```
(2, 3)
```

Convert the following list into an array.

In [9]:
```
sw = [4,5,6,1,2,3,7,3.5,8,9]
sw = np.array(sw)
sw
```

Out[9]:
```
array([4. , 5. , 6. , 1. , 2. , 3. , 7. , 3.5, 8. , 9. ])
```

During your project, you will sometimes work with arrays having above two dimensions.

Define and display an array of dimension three, with each dimension having a size of two. The choice of coefficients is up to you.

In [10]:
```
c = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
c
```

Out[10]:
```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

## Accessing/changing specific elements, rows, columns...

For this section, the tasks will be performed using an two-dimensions array made of random numbers.

Note: we manually set a seed for the random module here to ensure everyone gets the same (pseudo) random numbers. This is simply a convenience for this exercise, if we don't set a seed, the current time will be taken automatically.

In [23]:
```
np.random.seed( seed = 100)
arr = np.random.rand(5,5)
print(arr)
```

```
[[0.54340494 0.27836939 0.42451759 0.84477613 0.00471886]
 [0.12156912 0.67074908 0.82585276 0.13670659 0.57509333]
 [0.89132195 0.20920212 0.18532822 0.10837689 0.21969749]
 [0.97862378 0.81168315 0.17194101 0.81622475 0.27407375]
 [0.43170418 0.94002982 0.81764938 0.33611195 0.17541045]]
```

Retrieve and display the 2nd coefficient in the 3rd row of the array. Watch out that indexes start at 0 for rows and columns.

In [24]:
```
a = arr[2,1]
print(a)
```

```
0.20920212211718958
```

Retrieve and display the last column of the array. Change it to only zeros and display the new array.

In [25]:
```
b = arr[:,-1]
print(b)

new_arr = arr.copy()
new_arr[:,-1] = 0
print(new_arr)
```

```
[0.00471886 0.57509333 0.21969749 0.27407375 0.17541045]
[[0.54340494 0.27836939 0.42451759 0.84477613 0.        ]
 [0.12156912 0.67074908 0.82585276 0.13670659 0.        ]
 [0.89132195 0.20920212 0.18532822 0.10837689 0.        ]
 [0.97862378 0.81168315 0.17194101 0.81622475 0.        ]
 [0.43170418 0.94002982 0.81764938 0.33611195 0.        ]]
```

In [26]:
```
arr[:,-1]
```

Out[26]:
```
array([0.00471886, 0.57509333, 0.21969749, 0.27407375, 0.17541045])
```

Notice that depending on how you make the changes (if you work on the initial array or on a copy), you can change permanently the coefficients of the array. You can observe that if you run the last cell only, a second time: your last column displayed will now be all zeros, as according to your modifications. This shows that you may have to run the whole Jupyter notebook to reinitialize your array, and watch out in the future when you make modifications like this.

Change the first coefficient in the array to pi. Make sure that you change it to Python's pi number and not just a manual change to "3.14159" or similar.

In [27]:
```
import math

arr[0] = math.pi
print(arr)
```

```
[[3.14159265 3.14159265 3.14159265 3.14159265 3.14159265]
 [0.12156912 0.67074908 0.82585276 0.13670659 0.57509333]
 [0.89132195 0.20920212 0.18532822 0.10837689 0.21969749]
 [0.97862378 0.81168315 0.17194101 0.81622475 0.27407375]
 [0.43170418 0.94002982 0.81764938 0.33611195 0.17541045]]
```

It will also be important for your project to understand how to work with multi-dimensional arrays (above 2). Create a random array of size (10,10,3) and, *considering that time and cohort start being accounted for in 1975*, access the coefficient corresponding to year t = 1976, cohort c = 1980, type j = 2 (with the three types being 0, 1 and 2).

In [30]:
```
np.random.seed(100)
stock_tcj = 1e5 * np.random.rand(10,10,3) # random numbers
print(stock_tcj[1,5,2])
```

```
25069.52291383959
```

## Basic operations

Add the following two arrays and display the result.

In [31]:
```
a = np.array([3,2,1])
b = np.sort(a)

a+b
```

Out[31]:
```
array([4, 4, 4])
```

Multiply (element-wise) your result with the following array.

In [32]:
```
x = np.array([2,8,3])

(a+b) * x
```

Out[32]:
```
array([ 8, 32, 12])
```

Sort the following array and display the result.

In [33]:
```
sw = np.array([4, 5, 6, 1, 2, 3, 7, 3.5, 8, 9])

np.sort(sw)
```

Out[33]:
```
array([1. , 2. , 3. , 3.5, 4. , 5. , 6. , 7. , 8. , 9. ])
```

Perform an algebraic multiplication of the following two arrays using the **np.dot()** method and the **np.einsum()** method, and display the results.

In [38]:
```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([10,20,30])

print("np.dot", a.dot(b))
print("np.einsum", np.einsum('ij,j->i', a, b))
```

```
np.dot [140 320]
np.einsum [140 320]
```