Bootcamp
Python

42 ARTIFICIAL INTELLIGENCE

Module03
Numpy

Module03 - Numpy

Today you will learn how to use the Python library that will allow you to manipulate multidimensional arrays (vectors, matrices, tensors...) and perform complex mathematical operations on them.

Notions of the module

Numpy array, slicing, stacking, dimensions, broadcasting, normalization, etc...

General rules

- Use the Numpy Library: use Numpy's built-in functions as much as possible. Here you will be given no credit for reinventing the wheel.
- The version of Python recommended to use is 3.7, you can check the version of Python with the following command: `python -V`
- The norm: during this bootcamp you will follow the PEP 8 standards. You can install pycodestyle which is a tool to check your Python code.
- The function eval is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask questions in the `#bootcamps` channel in the 42 AI Slack: 42-ai.slack.com.
- If you find any issue or mistakes in the subject please create an issue on our bootcamp python repository on Github.

Helper

For this module you will use the image provided in the `resources` folder

Ensure that you have the right Python version.

```
$> which python
/goinfre/miniconda/bin/python
$> python -V
Python 3.7.*
$> which pip
/goinfre/miniconda/bin/pip
```

Exercise 00 - NumpyCreator

Exercise 01 - ImageProcessor

Exercise 02 - ScrapBooker

Exercise 03 - ColorFilter

Exercise 04 - K-means Clustering

Exercise 00 - NumpyCreator

Turn-in directory:	ex00/
Files to turn in:	NumpyCreator.py
Allowed libraries:	Numpy
Remarks:	n/a

Objective:

Introduction to Numpy library.

Instructions:

Write a class named `NumpyCreator`, that implements all of the following methods.
Each method receives as an argument a different type of data structure and transforms it into a Numpy array:

- `from_list(lst)` : takes a list or nested lists and returns its corresponding Numpy array.
- `from_tuple(tpl)` : takes a tuple or nested tuples and returns its corresponding Numpy array.
- `from_iterable(itr)` : takes an iterable and returns an array which contains all its elements.
- `from_shape(shape, value)` : returns an array filled with the same value.
  The first argument is a tuple which specifies the shape of the array, and the second argument specifies the value of the elements. This value must be 0 by default.
- `random(shape)` : returns an array filled with random values.
  It takes as an argument a tuple which specifies the shape of the array.
- `identity(n)` : returns an array representing the identity matrix of size n.

BONUS: Add to those methods an optional argument which specifies the datatype (dtype) of the array (e.g. to represent its elements as integers, floats, ...)

NB: All those methods can be implemented in one line. You only need to find the right Numpy functions.

Examples:

```
from NumpyCreator import NumpyCreator
 npc = NumpyCreator()

npc.from_list([[1,2,3],[6,3,4]])
# Output
array([[1, 2, 3],
       [6, 3, 4]])

npc.from_list([[1,2,3],[6,4]])
# Output:
None

npc.from_list([[1,2,3],['a','b','c'],[6,4,7]])
# Output:
array([['1','2','3'],
       ['a','b','c'],
       ['6','4','7'], dtype='<U21'])

npc.from_list((1,2),(3,4))
# Output:
None
```

```
npc.from_tuple(("a", "b", "c"))
# Output
array(['a', 'b', 'c'])

npc.from_tuple(["a", "b", "c"])
# Output
None

npc.from_iterable(range(5))
# Output
array([0, 1, 2, 3, 4])

shape=(3,5)
npc.from_shape(shape)
# Output
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])

npc.random(shape)
# Output
array([[0.57055863, 0.23519999, 0.56209311, 0.79231567, 0.213768  ],
      [0.39608366, 0.18632147, 0.80054602, 0.44905766, 0.81313615],
      [0.79585328, 0.00660962, 0.92910958, 0.9905421 , 0.05244791]])

npc.identity(4)
# Output
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Exercise 01 - ImageProcessor

Turn-in directory:	ex01/
Files to turn in:	ImageProcessor.py
Forbidden functions:	None
Helpful libraries:	Matplotlib

Objective:

Basic manipulation of image via matplotlib library

Instructions:

Build a tool that will be helpful to load and display images in the upcoming exercises.

Write a class named `ImageProcessor` that implements the following methods:

- `load(path)` : opens the PNG file specified by the `path` argument and returns an array with the RGB values of the pixels image. It must display a message specifying the dimensions of the image (e.g. 340 x 500).
- `display(array)` : takes a numpy array as an argument and displays the corresponding RGB image.

NB: You can use the library of your choice for this exercise, but converting the image to a numpy array is mandatory. The goal of this exercise is to dispense with the technicality of loading and displaying images, so that you can focus on array manipulation in the upcoming exercises.

Examples:

```
from ImageProcessor import ImageProcessor
imp = ImageProcessor()
arr = imp.load("non_existing_file.png")
# Output
Exception: FileNotFoundError -- strerror: No such file a directory

print(arr)
# Output:
None

arr = imp.load("Eempty_file.png")
# Output:
Exception: OSError -- strerror: None

print(arr)
# Output
None

arr = imp.load("../resources/42AI.png")
Loading image of dimensions 200 x 200
arr
array([[[0.03529412, 0.12156863, 0.3137255 ],
        [0.03921569, 0.1254902 , 0.31764707],
        [0.04313726, 0.12941177, 0.3254902 ],
        ...,
        [0.02745098, 0.07450981, 0.22745098],
        [0.02745098, 0.07450981, 0.22745098],
        [0.02352941, 0.07058824, 0.22352941]],
```

```
        [[0.03921569, 0.11764706, 0.30588236],
         [0.03529412, 0.11764706, 0.30980393],
         [0.03921569, 0.12156863, 0.30980393],
         ...,
         [0.02352941, 0.07450981, 0.22745098],
         [0.02352941, 0.07450981, 0.22745098],
         [0.02352941, 0.07450981, 0.22745098]],

        [[0.03137255, 0.10980392, 0.2901961 ],
         [0.03137255, 0.11372549, 0.29803923],
         [0.03529412, 0.11764706, 0.30588236],
         ...,
         [0.02745098, 0.07450981, 0.23137255],
         [0.02352941, 0.07450981, 0.22745098],
         [0.02352941, 0.07450981, 0.22745098]],

        ...,

        [[0.03137255, 0.07450981, 0.21960784],
         [0.03137255, 0.07058824, 0.21568628],
         [0.03137255, 0.07058824, 0.21960784],
         ...,
         [0.03921569, 0.10980392, 0.2784314 ],
         [0.03921569, 0.10980392, 0.27450982],
         [0.03921569, 0.10980392, 0.27450982]],

        [[0.03137255, 0.07058824, 0.21960784],
         [0.03137255, 0.07058824, 0.21568628],
         [0.03137255, 0.07058824, 0.21568628],
         ...,
         [0.03921569, 0.10588235, 0.27058825],
         [0.03921569, 0.10588235, 0.27058825],
         [0.03921569, 0.10588235, 0.27058825]],

        [[0.03137255, 0.07058824, 0.21960784],
         [0.03137255, 0.07058824, 0.21176471],
         [0.03137255, 0.07058824, 0.21568628],
         ...,
         [0.03921569, 0.10588235, 0.26666668],
         [0.03921569, 0.10588235, 0.26666668],
         [0.03921569, 0.10588235, 0.26666668]]], dtype=float32)
imp.display(arr)
```

The image must to be displayed in a separate window when running in the console.

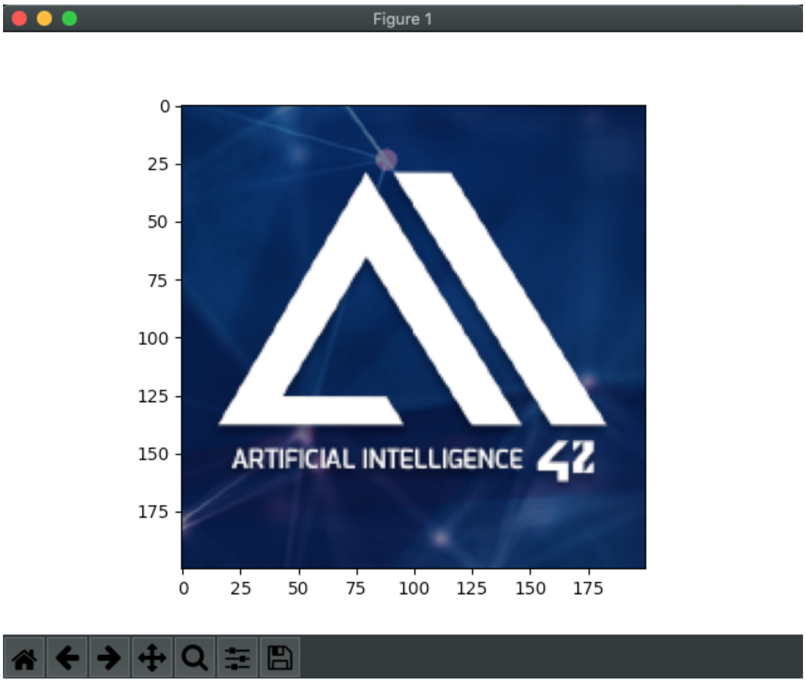

Figure 1: 42AIlogo

Exercise 02 - ScrapBooker

Turn-in directory:	ex02/
Files to turn in:	ScrapBooker.py
Allowed libraries:	NumPy
Notions:	Slicing

Objective:

Manipulation and initiation to slicing method on numpy arrays.

Instructions:

Implement a class named `ScrapBooker` with the following methods:

- `crop`,
- `thin`,
- `juxtapose`,
- `mosaic`.

```python
# within the class
def crop(self, array, dim, position=(0,0)):
    """
    Crops the image as a rectangle via dim arguments (being the new height
    and width of the image) from the coordinates given by position arguments.
    Args:
    -----
      array: numpy.ndarray
      dim: tuple of 2 integers.
      position: tuple of 2 integers.
    Return:
    -------
      new_arr: the cropped numpy.ndarray.
      None (if combinaison of parameters not compatible).
    Raise:
    ------
      This function should not raise any Exception.
    """
    ... your code ...

def thin(self, array, n, axis):
    """
    Deletes every n-th line pixels along the specified axis (0: vertical, 1: horizontal)
    Args:
    -----
      array: numpy.ndarray.
      n: non null positive integer lower than the number of row/column of the array
         (depending of axis value).
      axis: positive non null integer.
    Return:
    -------
      new_arr: thined numpy.ndarray.
      None (if combinaison of parameters not compatible).
    Raise:
    ------
      This function should not raise any Exception.
```

```
    """
    ... your code ...

def juxtapose(self, array, n, axis):
    """
    Juxtaposes n copies of the image along the specified axis.
    Args:
    -----
      array: numpy.ndarray.
      n: positive non null integer.
      axis: integer of value 0 or 1.
    Return:
    -------
      new_arr: juxtaposed numpy.ndarray.
      None (combinaison of parameters not compatible).
    Raises:
    -------
      This function should not raise any Exception.
    """
    ... your code ...

def mosaic(self, array, dim):
    """
    Makes a grid with multiple copies of the array. The dim argument specifies
    the number of repetition along each dimensions.
    Args:
    -----
      array: numpy.ndarray.
      dim: tuple of 2 integers.
    Return:
    -------
      new_arr: mosaic numpy.ndarray.
      None (combinaison of parameters not compatible).
    Raises:
    -------
      This function should not raise any Exception.
      """
      ... your code ...
```

In this exercise, when specifying positions or dimensions, we will assume that the first coordinate is counted along the vertical axis starting from the top, and that the second coordinate is counted along the horizontal axis starting from the left. Indexing starts from 0.

e.g.: (1,3) . . . . x. . . . ..

Examples:

```
import numpy as np
from Scrapbooker import ScrapBooker

spb = ScrapBooker()
arr1 = np.arange(0,25).reshape(5,5)
spb.crop(arr1, (3,1),(1,0))
#Output
array([[ 5],
       [10],
       [15]])

arr2 = np.array("A B C D E F G H I".split() * 6).reshape(-1,9)
spb.thin(arr2,3,0)
```

```
#Output
array([['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K'],
       ['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K'],
       ['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K'],
       ['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K'],
       ['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K'],
       ['A', 'B', 'D', 'E', 'G', 'H', 'J', 'K']], dtype='<U1')

arr3 = np.array([[1, 2, 3],[1, 2, 3],[1, 2, 3]])
spb.juxtapose(arr3, 3, 1)
#Output
array([[1, 2, 3, 1, 2, 3, 1, 2, 3],
       [1, 2, 3, 1, 2, 3, 1, 2, 3],
       [1, 2, 3, 1, 2, 3, 1, 2, 3]])
```

Exercise 03 - ColorFilter

Turn-in directory:	ex03/
Files to turn in:	ColorFilter.py
Forbidden functions:	See each method
Notions:	Broadcasting

Objective:

Manipulation of loaded image via numpy arrays and broadcasting.

Instructions:

You have to build a tool that can apply a variety of color filters on images. For this exercise, the authorized functions and operators are specified for each methods. You are not allowed to use anything else.

Write a class named `ColorFilter` with 6 methods with the following signatures:

```
def invert(array):
    """
    Inverts the color of the image received as a numpy array.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
    Raises:
    -------
        This function should not raise any Exception.
    """
```

```
def to_blue(array):
    """
    Applies a blue filter to the image received as a numpy array.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
    Raises:
    -------
        This function should not raise any Exception.
    """
```

```
def to_green(array):
    """
    Applies a green filter to the image received as a numpy array.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
```

```
    Raises:
    -------
        This function should not raise any Exception.
    """
```

```
def to_red(array):
    """
    Applies a red filter to the image received as a numpy array.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
    Raises:
    -------
        This function should not raise any Exception.
    """
```

```
def to_celluloid(array):
    """
    Applies a celluloid filter to the image received as a numpy array.
    Celluloid filter must display at least four thresholds of shades.
    Be careful! You are not asked to apply black contour on the object,
    you only have to work on the shades of your images.
    Remarks:
        celluloid filter is also known as cel-shading or toon-shading.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
    Raises:
    -------
        This function should not raise any Exception.
    """
```

```
def to_grayscale(array, filter, **kwargs):
    """
    Applies a grayscale filter to the image received as a numpy array.
    For filter = 'mean'/'m': performs the mean of RBG channels.
    For filter = 'weight'/'w': performs a weighted mean of RBG channels.
    Args:
    -----
        array: numpy.ndarray corresponding to the image.
        filter: string with accepted values in ['m','mean','w','weight']
        weights: [kwargs] list of 3 floats where the sum equals to 1,
                 corresponding to the weights of each RBG channels.
    Return:
    -------
        array: numpy.ndarray corresponding to the transformed image.
        None: otherwise.
    Raises:
    -------
        This function should not raise any Exception.
    """
```

You have some restrictions on the authorized methods and operators for each filter method in class `ColorFilter`:

- `invert`:
  - Authorized functions: `None`
  - Authorized operator: `+`, `-`
- `to_blue` :
  - Authorized functions: `.zeros`, `.shape`, `.dstack`
  - Authorized operator: `None`
- `to_green`:
  - Authorized functions: `copy,deepcopy`
  - Authorized operator: `*`
- `to_red`:
  - Authorized functions : `.to_green`, `.to_blue`
  - Authorized operator: `-`, `+`
- `to_celluloid(array)`:
  - Authorized functions: `.arange`, `.linspace`
- `to_grayscale`:
  - Authorized functions: `.sum`, `.shape`, `.reshape`, `.broadcast_to`, `.as_type`
  - Authorized operator: `*`, `/`

```
>>> from ImageProcessor import ImageProcessor
>>> imp = ImageProcessor()
>>> arr = imp.load("../ressources/42AI.png")
Loading image of dimensions 200 x 200
>>> from ColorFilter import ColorFilter
>>> cf = ColorFilter()
>>> cf.invert(arr)
>>>
>>> cf.to_green(arr)
>>>
>>> cf.to_red(arr)
>>>
>>> cf.to_blue(arr)
>>>
>>> cf.to_celluloid(arr)
>>>
>>> cf.to_grayscale(arr, 'm')
>>>
>>> cf.to_grayscale(arr, 'weighted', [0.2, 0.3, 0.5])
>>>
```

Examples:

Info:

“This image is a stylisation of elon musk that has been generated using a style transfer algorithm implemented in our lab. You can see the code in our repository 42-AI/StyleTransferMirror”

Figure 2: Elon Musk

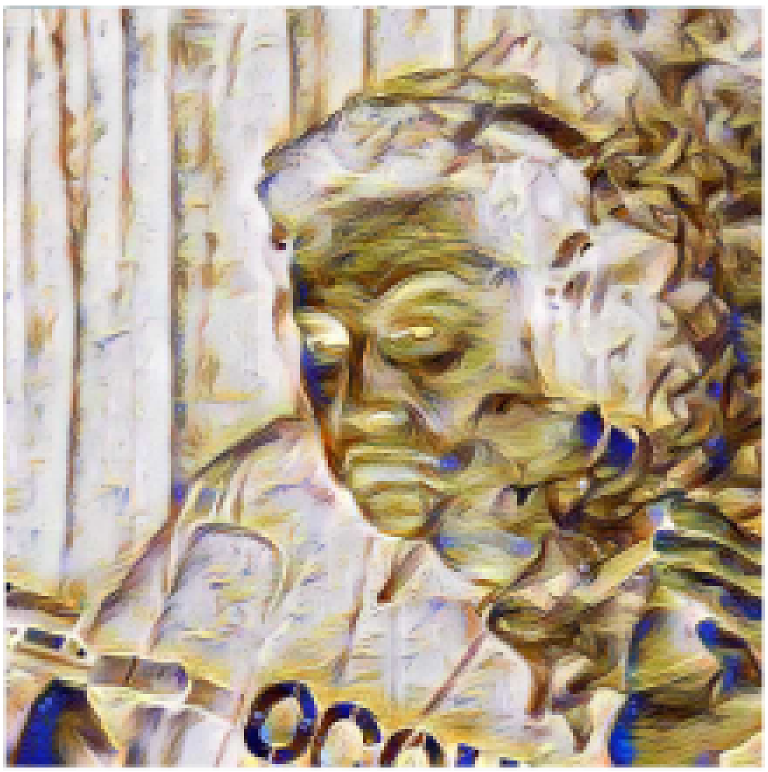

Figure 3: invert

Figure 4: to_blue

Figure 5: to_green

Figure 6: to_red

Figure 7: celluloid

Exercise 04 - K-means Clustering

Turn-in directory:	ex04/
Files to turn in:	Kmeans.py
Forbidden functions:	None
Remarks:	n/a

ALERT! DATA CORRUPTED

Objective:

Implementation of a basic Kmeans algorithm.

Instructions:

The solar system census dataset is corrupted! The citizens' homelands are missing!
You must implement the K-means clustering algorithm in order to recover the citizens' origins.

On this web-page you can find good explanations on how K-means is working:
Possibly the simplest way to explain K-Means algorithm

The missing part is how to compute the distance between 2 data points (cluster centroid or a row in the data). In our case the data we have to process is composed of 3 values (height, weight and bone_density). Thus, each data point is a vector of 3 values.
Now that we have mathematically defined our data points (vector of 3 values), it is then very easy to compute the distance between two points using vector properties. You can use L1 distance, L2 distance, cosine similarity, and so forth... Choosing the distance to use is called hyperparameter tuning. I would suggest you to try with the easiest setting (L1 distance) first.
What you will notice is that the final result of the "training"/"fitting" will depend a lot on the random initialization. Commonly, in machine-learning libraries, K-means is run multiple times (with different random initializations) and the best result is saved.

NB: To implement the fit function, keep in mind that a centroid can be considered as the gravity center of a set of points.

Your program `Kmeans.py` takes 3 parameters: `filepath`, `max_iter` and `ncentroid`:

```
python Kmeans.py filepath='../ressources/solar_system_census.csv' ncentroid=4 max_iter=30
```

it is expected by running your program to:

- parse the arguments,
- read the dataset,
- fit the dataset,
- display the coordinates of the different centroids and the associated region (for the case `ncentroid=4`),
- display the number of individuals associated to each centroid,
- (Optional) display on 3 differents plots, corresponding to 3 combinaisons of 2 parameters, the results. Use different colors to distinguish between Venus, Earth, Mars and Belt asteroids citizens.

Create the class `KmeansClustering` with the following methods:

```python
class KmeansClustering:
    def __init__(self, max_iter=20, ncentroid=5):
        self.ncentroid = ncentroid # number of centroids
        self.max_iter = max_iter # number of max iterations to update the centroids
        self.centroids = [] # values of the centroids

    def fit(self, X):
```

```
        """
        Run the K-means clustering algorithm.
        For the location of the initial centroids, random pick ncentroids from the dataset.
        Args:
        -----
          X: has to be an numpy.ndarray, a matrice of dimension m * n.
        Return:
        -------
          None.
        Raises:
        -------
          This function should not raise any Exception.
        """
        ... your code ...

    def predict(self, X):
        """
        Predict from wich cluster each datapoint belongs to.
        Args:
        -----
          X: has to be an numpy.ndarray, a matrice of dimension m * n.
        Return:
        -------
          the prediction has a numpy.ndarray, a vector of dimension m * 1.
        Raises:
        -------
          This function should not raise any Exception.
        """
        ... your code ...
```

Dataset:

The dataset, named **solar_system_census** can be found in the resources folder.
It is a part of the solar system census dataset, and contains biometric informations such as the height, weight, and bone density of solar system citizens.

As you should know solar citizens come from four registered areas: The flying cities of Venus, United Nations of Earth, Mars Republic, and the Asteroids' Belt colonies.

Unfortunately the data about the planets of origin was lost... Use your K-means algorithm to recover it! Once your clusters are found, try to find matches between clusters and the citizens' homelands.

Hints:

- People are slender on Venus than on Earth.
- People of the Martian Republic are taller than on Earth.
- Citizens of the Belt are the tallest of the solar system and have the lowest bone density due to the lack of gravity.

Example:

Here is an exemple of the algorithm in action: K-means animation