# NumPy Arrays

Pouria Hadjibagheri and Gerold Baier

Experimental

2018-19

# Contents

# 0.1 NumPy Arrays

You should be familiar with two types of arrays in Python, lists and dictionaries. Here we will look at *NumPy arrays* which are frequently used to handle data in Machine Learning. NumPy is a fundamental package for scientific computing with Python. See for example this quickstart tutorial.

## 0.1.1 Creating NumPy arrays

Let us practice creating different kinds of NumPy arrays. In each case, the corresponding functions need to be imported from the package.

A Python *list* can directly be converted to a NumPy array:

In [1]:

```
from numpy import asarray


mylist = [1, 3, 5, 7, 9]

na_mylist = asarray(mylist)

print(na_mylist)
```

Out [1]:

```
[1 3 5 7 9]
```

You can check its type by using `type`:

In [2]:

```
print('Type:', type(na_mylist))
```

Out [2]:

```
Type: <class 'numpy.ndarray'>
```

There are some convenient predefined types of arrays, for instance arrays of zeroes and ones:

In [3]:

```
from numpy import zeros, ones


na_zeros = zeros(5)
na_ones = ones(4)

print(na_zeros, na_ones)
```

Out [3]:

```
[0. 0. 0. 0. 0.] [1. 1. 1. 1.]
```

Here is one way to generate a sequence of consecutive numbers:

In [4]:

```
from numpy import arange


# Sequence of integers from 1 to 10
na_range = arange(1, 11)

# Array of integers from 1 to 10 in steps of 2
na_range_2 = arange(1, 11, 2)

print(na_range, na_range_2, sep='\n')
```

Out [4]:

```
[ 1  2  3  4  5  6  7  8  9 10]
[1 3 5 7 9]
```

And here is another possibility: we can divide a given interval into a set of numbers:

In [5]:
```
from numpy import linspace


# Array of 6 numbers from 1 to 10

na_linspace = linspace(1, 10, 6)

print(na_linspace)
```

Out [5]:
```
[ 1.   2.8  4.6  6.4  8.2 10. ]
```

Then there is the possibility for a variety of arrays containing random numbers. The functions are contained in `numpy.random`. As an example, here is how to create an array of normally distributed random numbers:

In [6]:
```
from numpy.random import normal

MEAN = 0
STD = 1

na_random = normal(MEAN, STD, size=5)

print(na_random)
```

Out [6]:
```
[ 1.34466714 -0.26935832  0.62343156  0.35500307  1.22220159  2.42802815]
```

So far we have only seen linear, one-dimensional arrays. However, NumPy arrays can be more than one-dimensional:

In [7]:
```
na_random = normal(MEAN, STD, size=(6,5))
print(na_random)
```

Out [7]:
```
[[-0.03375752 -1.46393965  1.14079763  0.55099799 -1.03035426]
 [-0.34692165 -0.21415599 -1.18715302 -1.33351754 -0.44242791]
 [ 1.30445973 -1.13354672 -1.48283949 -0.82756462 -1.26541849]
 [ 0.53111819 -0.59019693 -1.53054258  0.9659866   0.75995194]
 [-0.51780085  0.04335344  0.2644038  -1.83251049 -0.00800095]
 [-1.04727405  0.84087167 -0.92577748  0.43677273  0.67695771]]
```

To find out the dimension, the shape, and the number of elements of an array we use the following *methods*:

In [8]:
```
print(na_random.ndim)
```

Out [8]:
```
2
```

In [9]:
```
print(na_random.shape)
```

Out [9]:
```
(6, 5)
```

In [10]:
```
print(na_random.size)
```

Out [10]:
```
30
```

## 0.1.2 Boolean operations

An important feature of these arrays is that we can perform Boolean operations on them. E.g. we can label an array according to whether a number is larger than zero or not:

In [1]:
```
mask = na_random > 0

print(mask)
```

Out [1]:
```
[[False, False,  True,  True, False],
 [False, False, False, False, False],
 [ True, False, False, False, False],
 [ True, False, False,  True,  True],
 [False,  True,  True, False, False],
 [False,  True, False,  True,  True]]
```

## 0.1.3 Reshaping

To alter the shape of a *NumPy* array, we may use the `.reshape()` *method*. The *method* takes numeric arguments that define the new shape of an array. For instance, `.reshape(3, 4, 2)` will, if possible, attempt to reshape the array onto a three-dimensional array with 3 rows, 4 columns, and 2 planes.

Let's us create a simple one-dimensional array:

In [1]:
```
one_dim_array = asarray([1, 2, 3, 4])

print('Shape:', one_dim_array.shape)
print(one_dim_array)
```

Out [1]:
```
Shape: (4,)
[1 2 3 4]
```

To convert this array to two-dimensions, we do as follows:

In [2]:
```
two_dim_array = one_dim_array.reshape(-1, 1)

print('Shape:', two_dim_array.shape)
print(two_dim_array)
```

Out [2]:
```
Shape: (4, 1)
[[1]
 [2]
 [3]
 [4]]
```

In this example, we used `.reshape(-1, 1)` to reshape our array, which in essence means the follow: Reshape the array to *as many rows as needed* (*i.e.* $-1$) with 1 column.

It is important that the members in an array exactly fit the new shape. To that end, instead of using $-1$ in the above example, we can use the number of rows that we require:

In [3]:
```
alternative_two_dim = one_dim_array.reshape(4, 1)

print('Shape:', alternative_two_dim.shape)
print(alternative_two_dim)
```

Out [3]:
```
Shape: (4, 1)
[[1]
 [2]
 [3]
 [4]]
```

If, however, we attempt to reshape an array, and the number of members do not fit the target shape, a `ValueError` will be raised:

In [4]:
```
one_dim_array.reshape(5, 1)
```

Out [4]:
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
----> 1 one_dim_array.reshape(5, 1)

    ValueError: cannot reshape array of size 4 into shape (5,1)
```

Conversely, we can convert a two-dimensional array to a one-dimensional, too:

In [5]:
```
back_to_one_dim = two_dim_array.reshape(-1)

print('Shape:', back_to_one_dim.shape)
print(back_to_one_dim)
```

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

Alternatively, we can use the `.ravel()` method to convert an array into one-dimension:

In [6]:
```
one_dim_ravel = two_dim_array.ravel()

print('Shape:', one_dim_ravel.shape)
print(one_dim_ravel)
```

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

## 0.1.4 Mesh grid

Suppose that we need a matrix of numbers that represent every possible combination of two numeric vectors (*i.e.* one-dimensional arrays). This matrix is referred to as a *mesh grid*. In other a words, a *mesh grid* of 2 vectors `a` and `b` is defined as 2 matrices, such that `A` is a matrix where each row is a copy of vector `a`, and `B` is a matrix where each column is a copy of vector `b`.

In mathematical terms, a mesh grid is defined as:

Given vectors `a` and `b`:

$$a = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

$$b = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

The grids are defined as:

$$A = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} -2 & -2 & -2 & -2 & -2 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

There are two options to implement such grids in Python: - Using the `meshgrid()` function, we can create a grid from 2 existing vectors (*i.e.* one-dimensional array); - Using the `mgrid` function, which allows us to define our grid *ab initio*.

Using the `meshgrid()` function:

In [1]:

```
from numpy import arange

vector_a = arange(5)
vector_b = arange(4)

print('Vector A, shape:', vector_a.shape)
print(vector_a)

print('Vector B, shape:', vector_b.shape)
print(vector_b)
```

Out [1]:

```
Vector A, shape: (5,)
[0 1 2 3 4]
Vector B, shape: (4,)
[0 1 2 3]
```

In [2]:

```
from numpy import meshgrid


grid_a, grid_b = meshgrid(vector_a, vector_b)

print('Grid A, shape:', grid_a.shape)
print(grid_a)

print('Grid B, shape:', grid_b.shape)
print(grid_b)
```

Out [2]:

```
Grid A, shape: (4, 5)
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
Grid B, shape: (4, 5)
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
```

```
 [3 3 3 3 3]]
```

An important application of such a grid is that it can be used for the display of functions of two variables:

In [3]:

```python
from matplotlib.pyplot import subplots
from numpy import linspace

vector_a = linspace(-1, 1, 100)
vector_b = linspace(-1, 1, 100)

grid_a, grid_b = meshgrid(vector_a, vector_b)

z = grid_a ** 2 + grid_b ** 2


fig, ax = subplots(figsize=(5, 5))

ax.contourf(grid_a, grid_b, z, cmap='gray_r');
```

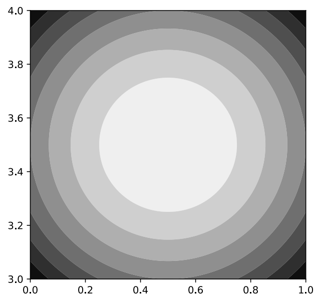


**Figure 0.1.1** Contour plot

> **Note**
> MatPlotLib supplies a range of colour maps (referred to as cmap), see their official documentation to find out more. Note that the reverse of each colour map can be obtained by adding _r to the end of the colour map's name.

Alternatively, we can use the `mgrid` function to simplify the process. The application of `mgrid` is slightly different to other Python functions, in that it uses square brackets (`[...]`) instead of parentheses (`(...)`) to receive arguments. The arguments are defined as follows:

Out [3]:

```
#       dimension 1       dimension 2      ...
# ------------------------------------------
mgrid[start:stop:step, start:stop:step, ...]
```

In [4]:

```python
from numpy import mgrid


grid_a, grid_b = mgrid[-2:3:1, -2:3:1]

print('Grid A, shape:', grid_a.shape)
print(grid_a)

print('Grid B, shape:', grid_b.shape)
print(grid_b)
```

Out [4]:

```
Grid A, shape: (5, 5)
[[-2 -2 -2 -2 -2]
 [-1 -1 -1 -1 -1]
 [ 0  0  0  0  0]
 [ 1  1  1  1  1]
 [ 2  2  2  2  2]]
Grid B, shape: (5, 5)
[[-2 -1  0  1  2]
 [-2 -1  0  1  2]
```

```
 [-2 -1  0  1  2]
 [-2 -1  0  1  2]
 [-2 -1  0  1  2]]
```

Let us now convert these grids into a one-dimensional array to represent the following vectors:

> **Note**
> As you may have noticed, there is a slight difference between the outputs of `meshgrid` and `mgrid` in that the first and the second matrices are returned in a different order. That is, when we use `meshgrid` the first matrix represent each member of the the vector it is given in a different columns, and the second matrix represents them in different rows. Conversely, `mgrid` returns the first and the second matrices to represent the members in different rows and columns respectively. This is usually not important, but it is something that a programmer / data analyst is expected to be aware of.

$$G'_A = \begin{bmatrix} -2 & -2 & -2 & -2 & -2 & -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

$$G'_B = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 & -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 & -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

In [5]:

```python
grid_a_flat = grid_a.ravel()
grid_b_flat = grid_b.ravel()

print('Grid A:', grid_a_flat)
print('Grid B:', grid_b_flat)
```

Out [5]:

```
Grid A: [-2 -2 -2 -2 -2 -1 -1 -1 -1 -1  0  0  0  0  0  1  1  1  1  1  2  2  2
     ↪  2
  2]
Grid B: [-2 -1  0  1  2 -2 -1  0  1  2 -2 -1  0  1  2 -2 -1  0  1  2 -2 -1  0
     ↪  1
  2]
```

We now have 2 one-dimensional arrays whose members, when taken together, represents every possible combination of the members of the original vectors:

$$f'(x) = \begin{bmatrix} -2 & -2 \\ -1 & -2 \\ 0 & -2 \\ 1 & -2 \\ 2 & -2 \\ \vdots & \vdots \\ 2 & 2 \end{bmatrix}$$

To create $f'(x)$ in Python, we need to concatenate `grid_a_flat` and `grid_b_flat`, and create a two-dimensional matrix. Concatenation of arrays in Python may be achieved using the `c_` function in `NumPy`. Similar to the `mgrid` function, the `c_` function requires its arguments to be given using square bracket (`[...]`):

In [6]:

```python
from numpy import c_


combinations = c_[grid_a_flat, grid_b_flat]

print(combinations)
```

Out [6]:

```
[[-2 -2]
 [-2 -1]
 [-2  0]
 [-2  1]
 [-2  2]
 [-1 -2]
 [-1 -1]
 [-1  0]
 [-1  1]
 [-1  2]
 [ 0 -2]
 [ 0 -1]
 [ 0  0]
 [ 0  1]
 [ 0  2]
 [ 1 -2]
 [ 1 -1]
 [ 1  0]
 [ 1  1]
 [ 1  2]
 [ 2 -2]
 [ 2 -1]
 [ 2  0]
 [ 2  1]
 [ 2  2]]
```

Let us now go ahead and plot `combinations`:

In [7]:

```
fig, ax = subplots(figsize=(5, 5))

ax.scatter(combinations[:, 0], combinations[:, 1])

ax.set(xticks=[-2, 0, 2], yticks=[-2, 0, 2]);
```

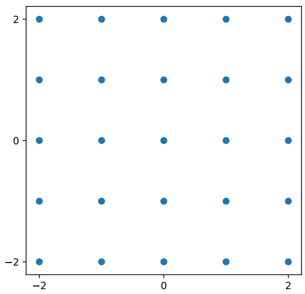


**Figure 0.1.2**

Two dimensional histograms are constructed using a grid of two one-dimensional arrays (vectors):

In [8]:

```
from numpy.random import normal


fig, ax = subplots(figsize=(10, 8))

x = normal(0, .5, 100000)
y = normal(0, .5, 100000)

_, _, _, cax = ax.hist2d(x, y, bins=100, cmap='magma')

fig.colorbar(cax);
```

For more on the creation of arrays see the Array creation tutorial from SciPy.

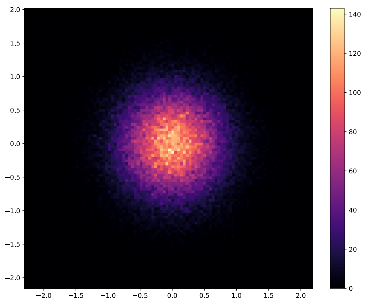
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
140
120
100
80
60
40
20
0

Figure 0.1.3