Numerical Computing - Numpy Arrays

Gabriele Pompa
gabriele.pompa@unisi.com

March 23, 2020

Contents

Resources:

- *Python for Finance (2nd ed.)*: Sec. 4.Numpy Arrays, 4.Basic Vectorization
- *Numpy Quickstart Tutorial - The Basics* (An Example; Array Creation; Printing Arrays; Basic Operations; Universal Functions; Indexing, Slicing and Iterating), *Numpy Quickstart Tutorial - Shape Manipulation* (Changing the shape of an array; Stacking together different arrays), *Numpy Quickstart Tutorial - Indexing with Boolean Arrays*.

Executive Summary

The following sections are organized as follows:

- In Sec. 1 we provide the motivation for a more specialized data-structure to model sequences of numbers w.r.t. lists.
- In Sec. 2 we introduce 1-dimensional NumPy's arrays, describing their similarities with lists (indexing, slicing, ...) and their peculiarities (element-wise operations).
- In Sec. 3 we generalize to multi-dimensional NumPy's arrays and their smart methods and universal functions that make them a great asset for numerical analysis.
- In Sec. 4 we conclude comparing Lists and NumPy's arrays in terms of speed of code and efficiency.

1 Introduction: from Lists to Arrays

We'll keep this discussion as intuitive as possible and as informal as possible too. The concept of *array* belongs to two knowledge domains (at least):

- **Mathematics**: an array is a sequence of numbers of the same type (Naturals, Rationals, Reals,...). It can be:
  - a 1-dimensional vector v. That is, the sequence of elements indexed by the one integer i;
  - a 2-dimensional matrix M. That is, the sequence of elements indexed by the couple of integers (i, j);

  - a N>2-dimensional tensor T. That is, the sequence of elements indexed by n-tuple of n integers $(i_1, \cdots, i_n)$. (*)

(*) Feel lost? Ok, no problem. Let's make an example with a 3-dimensional Tensor. So, if $N = 3$, a 3-dimensional sequence of numbers can be visualized as a *cube* of numbers indexed by the 3 indexes (i, j, k) where indexes i and j run along *height* (rows axes) and *width* (columns axes) of the cube, respectively. While index k runs along the depth (say the *pages* axes) of the cube. Therefore, each page is distinguished by the value of index $k = 0, 1, 2, ...$, whereas numbers on the same page differ by the values of (i, j) indexes (yes, you can think each page as a matrix of numbers). For example, the red 1 in the front page has indexes $(i, j, k) = (3, 2, 0)$, wherease the three green 2 on the bottom right corner have share same $i = 4$ and $j = 4$ indexes and differ by the value of $k = 0$ (front-page), $k = 1$ (second page) and $k = 2$ (back page). See picture.

- **Informatics**: an array is a sequence of data of the same data-type. The fact that all data stored in an array are of the same data-type is important because it allows to allocate the same amount of memory (bits) for each item in the array. Moreover, being a *sequence,* translates into the fact that consecutive items are stored in consecutive portions of memory, which are thus easily to be indexed and therefore quicker to be accessed.

We have already seen a great example of sequence-like data-structure in basic Python: the `list`. In particular Lists feature the following key facts:

a: *Lists are sequences.* Therefore, consecutive elements of the lists can be allocated in consecutive slots of memory.

b: *Lists can store simultaneously data of heterogeneous data-type.* Therefore, it's not known *a priori* whether we can reserve the same amount of memory to each element of the list.

c: *Lists are mutable (e.g. think to `.append()` method).* Therefore, the totale amount of memory to be reserved for the allocation of the whole list is not known *a priori* or, at most, is not fixed.

Points **b** and **c**, though they make lists very flexible, they also represent bottlenecks in terms of memory usage and performance. Lists are somehow too *general* to be excell excel in performance too.

There is the need of a more *specialized* data-structure, sharing with lists the sequentiality of data, but compromising on some flexibility in the name of performance. That's why we have NumPy and its data-structure `numpy.ndarray` has been created.

Key-facts of Numpy's arrays:

a: arrays extend the sequentiality of lists, introducing a built-in notion of dimensions (called *axes*);

b: array's length (*size*) is constrained to be immutable;

c: array's items are constrained to have the same data-type;

The built-in notion of dimensions allows to easily map the mathematical concepts of vectors, matrices and N-dimensional tensors into 1-dim, 2-dim and N-dim Numpy's arrays, respectively. Moreover, the constraints on array size (**b**) and same data-type **c** allow several speed improvements and *vectorization* of code. That is, those allow to have fast(er) memory access and to write functions that work on all the elements of an array "at once".

1.1 `numpy.ndarray` $\mu\epsilon\tau\alpha$-informations

These key-facts translates into the following meta-informations that can be accessed as attributes of any array:

Attribute	Meaning	Constraints (if any)
`.ndim`	The number of axes (dimensions) of an array: 1 for a vector, 2 for a matrix….	-
`.shape`	The dimensions of the array: a Tuple `(n,m)` for a matrix of `n` rows and `m` cols	-
`.size`	The number of elements of the array: `n` $\times$ `m` for a matrix of shape `(n,m)`	fixed (*)
`.dtype`	The data-type of array's elements	fixed for all elements

We'll use these attributes to explore arrays that we'll introduce.

(*) the `.resize()` method allows to actually re-size an array, but creating a new array. See section Section 3.4.3.

The function `type()` returns `numpy.ndarray` for NumPy's arrays.

As preliminary import we import `numpy` modulus and give to it the `np` alias

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

Now we have access to all the contents of NumPy module. Let's start!

2 1-dim arrays

We start with one-dimensional arrays (i.e. vectors). That is, a sequence of elements (usually numbers), all of the same data-type. As said, in NumPy, dimensions are called *axes* and 1-dim arrays have 1 dimension.

2.1 Array Creation

Array can be created: - from Lists or Tuples; - from sequences of numbers; - using placeholder functions.

2.1.1 From Lists or Tuples

We define a list of the first 0,…,9 integers squared

[2]:
```python
lis = [i**2 for i in range(10)]
lis
```

```
[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[3]: type(lis)
```

```
[3]: list
```

and we can define a 1-dim array `vec` accordingly

```
[4]: vec = np.array(lis)
     vec
```

```
[4]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
[5]: type(vec)
```

```
[5]: numpy.ndarray
```

Notice that I defined separately the list `lis` just for clarity. The above definition is equivalent to

```
[6]: vec = np.array([i**2 for i in range(10)])
     vec
```

```
[6]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Let's take a look at `vec` meta-informations:

`vec` is 1-dimensional:

```
[7]: vec.ndim
```

```
[7]: 1
```

altough not very significant in the 1-dim case, let's take a look at its shape: it has all the 10 values arranged along its unique dimension

```
[8]: vec.shape
```

```
[8]: (10,)
```

The number of elements:

```
[9]: vec.size
```

```
[9]: 10
```

observe that in the 1-dim case, you can retrieve the number of elements also using the `len()` function...

```
[10]: len(vec)
```

[10]: 10

…but we'll see this is not the case in the N-dimensional case, so please use `vec.size` if you want to know how many numbers your array holds.

Finally, having defined our vector as the array of the first 10 integers squared, it is created with elements of integer data-type and its `.dtype` is inferred accordingly

[11]:
```
vec.dtype
```

[11]:
```
dtype('int32')
```

Don't be scared by the fact that what is returned is `dtype('int32')` and not simply `int`, it's just that NumPy that has chosen to have it's own data-types. Anyway, you can read `dtype('int32')` as `int` peacefully.

Notice, that we could also have chosen explicitly to define our array as an array of Floats, instead of integers, using the `dtype` parameter of `np.array()` function

[12]:
```
vec_float = np.array(lis, dtype='float')
vec_float
```

[12]:
```
array([ 0.,  1.,  4.,  9., 16., 25., 36., 49., 64., 81.])
```

[13]:
```
vec_float.dtype
```

[13]:
```
dtype('float64')
```

and as you can see `vec_float` is the same of `vec` but its elements are all casted as decimal numbers and its data-type is then `dtype('float64')` (NumPy's version for `float`).

A possible *signature* for the creational function `np.array()` would be

`np.array(sequence[, dtype])`

where `sequence` could be a list and `dtype` - if not provided - is inferred by the data-type of `sequence`'s elements, as we have just seen. The syntax `[, optionalArgument]` is conventional. Get familiar with it.

What if we mix data-types? Recall that all the elements of a NumPy's array must share the same data-type, therefore if we mix (intentionally or unintentionally) types, it's `np.array` internals that take care of the required homogeneization of elements data-types. It's of course your duty to be aware of it.

We can distinguish a couple of relevant cases:

- `int` and `float`: NumPy promotes integers to floats and define the array as float `dtype`

[14]:
```
lis = [1, 2.5, 5, 6, 7.5]
print("lis: ", lis)

vec = np.array(lis)
```

```
print("dtype: ", vec.dtype)
vec
```

```
lis:  [1, 2.5, 5, 6, 7.5]
dtype:  float64
```

[14]:
```
array([1. , 2.5, 5. , 6. , 7.5])
```

- numbers and `str`: NumPy casts all the numbers as Strings and define the array as Unicode-encoded characters `dtype` (the `U` stands for Unicode), that is an array of string-like characters.

[15]:
```
lis = [1, 2.5, "EUR"]
print("lis: ", lis)

vec = np.array(lis)
print("dtype: ", vec.dtype)
vec
```

```
lis:  [1, 2.5, 'EUR']
dtype:  <U32
```

[15]:
```
array(['1', '2.5', 'EUR'], dtype='<U32')
```

Again, in these examples, lists are separately defined just for clarity. You could equivalently do:

[16]:
```
vec = np.array(lis)
print("dtype: ", vec.dtype)
vec
```

```
dtype:  <U32
```

[16]:
```
array(['1', '2.5', 'EUR'], dtype='<U32')
```

Take-home message: do not mix data-types in `np.array()` (unless you have a strong reason to do it).

2.1.2 From sequences of numbers: `np.arange()`

If you want to create an array from a sequence of numbers, you can use

`np.arange([start,] stop[, step])`

which creates a 1-dim array of numbers from `start` to `stop-1`, each `step` numbers.

As suggested by the use of `[]` conventional syntax in `arange`'s signature, the parameters `start` and `step` are optional and - if not specified - default values are `start=0` and `step=1`.

[17]:
```
vec = np.arange(10)
vec
```

```
[17]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

which is equal to

```
[18]: vec = np.arange(0,11,1)
```

In general we can write

```
[19]: vec = np.arange(1, 7, 0.25)
      vec
```

```
[19]: array([1.  , 1.25, 1.5 , 1.75, 2.  , 2.25, 2.5 , 2.75, 3.  , 3.25, 3.5 ,
             3.75, 4.  , 4.25, 4.5 , 4.75, 5.  , 5.25, 5.5 , 5.75, 6.  , 6.25,
             6.5 , 6.75])
```

When working with floating point values, the finite-precision may cause impredictability of the number of elements returned by `np.arange()`. Therefore, where the size is something we want to control for, it's better to use

`np.linspace([start,] stop[, num])`

which returns a 1-dimensional array of `num` numbers from `start`(included) to `stop` included

```
[20]: np.linspace(0,3,30)
```

```
[20]: array([0.        , 0.10344828, 0.20689655, 0.31034483, 0.4137931 ,
             0.51724138, 0.62068966, 0.72413793, 0.82758621, 0.93103448,
             1.03448276, 1.13793103, 1.24137931, 1.34482759, 1.44827586,
             1.55172414, 1.65517241, 1.75862069, 1.86206897, 1.96551724,
             2.06896552, 2.17241379, 2.27586207, 2.37931034, 2.48275862,
             2.5862069 , 2.68965517, 2.79310345, 2.89655172, 3.        ])
```

This is typically used when one wants to evaluate a function at a lot of points

```
[21]: from math import pi # importing pi constant

      x = np.linspace(0,2*pi,1000)
      y = np.sin(x) # this is another example of universal function, see later␣
       ↪dedicated section
```

let's plot it

```
[22]: import matplotlib.pylab as plt

      plt.plot(x,y)
      plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

2.1.3 Using placeholder content: `np.zeros()`, `np.ones()`, `np.empty()`

When you know how long your vector should be but you don't know which values to input in advance, you can use one of these initialized functions:

- `np.ones(shape[, dtype])`: creates an array of ones;
- `np.zeros(shape[, dtype])`: creates an array of zeros;
- `np.empty(shape[, dtype])`: creates an empty array, where the actual values depend on the state of memory.

In each of these,

- `dtype` is optional and, default set to `float64` (NumPy's '`float` type);
- `shape`, in the 1-dim case, is an `int` number representing the number of values the vector will holds.

These functions are used in their full generality in the multi-dimensional case. We'll see this later.

```
[23]: vec = np.ones(10)
      print("dtype: ", vec.dtype)
      vec
```

```
dtype:  float64
```

```
[23]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
[24]: vec = np.ones(10, dtype="int")
      print("dtype: ", vec.dtype)
      vec
```

```
dtype:  int32
```

```
[24]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
[25]: vec = np.zeros(10)
      print("dtype: ", vec.dtype)
      vec
```

```
dtype:  float64
```

```
[25]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
[26]: vec = np.zeros(10, dtype="int")
      print("dtype: ", vec.dtype)
      vec
```

```
dtype:  int32
```

```
[26]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[27]:
```
vec = np.empty(10)
print("dtype: ", vec.dtype)
vec
```

```
dtype:  float64
```

[27]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

these values filled by `.empty()` could have been whatever depending on the current state of the memory

2.2 2.2. Indexing, Slicing, Assigning and Iterating

One-dimensional arrays can be *zero-based* indexed, sliced and iterated over, much like lists and other Python sequences.

2.2.1 Indexing

Let's re-define our `vec` friend

[28]:
```
vec = np.array([i**2 for i in range(10)])
vec
```

[28]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

The one-dimensional array `vec` can be indexed as

`vec[i]`

where `i` is the index of the $(i+1)$-th element of `vec`. Notice that:

- positive indexing starts from `i == 0` up to `i == len(vec) - 1`
- negative indexing starts from `i == -1` down to `i == -len(vec)`

[29]:
```
# 0 is the index of the first element of the array (positive indexing)
print(vec[0])
type(vec[0])
```

```
0
```

[29]:
```
numpy.int32
```

[30]:
```
# len(vec) -1 is the index of the last element of the array (positive indexing)
print(vec[len(vec) - 1])
```

```
81
```

[31]:
```
# -1 is the index of the last element of the array (negative indexing)
print(vec[-1])
```

```
81
```

[32]:
```
# -len(vec) is the index of the last element of the array (negative indexing)
print(vec[-len(vec)])
```

```
0
```

2.2.2 Slicing

The one-dimensional array `vec` can be sliced as

`vec[i:j:k]`

where:

- `i` is the starting index (included) of the slice. It is optional: if not provided is set to `i == 0` (slice from the start)
- `j` is the starting index (excluded) of the slice, It is optional: if not provided is set to `j == len(vec)` (slice to the end)
- `k` is the step of the slice. It is optional: if not provided is set to `k == 1` (each element). If negative, reads the array from the last element.

That is, `vec[i:j:k]` slice `vec` from `vec[i]` to `vec[j-1]`, each `k` elements.

The role of the `:` (colon) is that of range selector. Here is the full range `vec[:]`

[33]:
```
vec
```

[33]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

[34]:
```
vec[:] # equivalent to simply vec
```

[34]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

[35]:
```
vec_slice = vec[0:2] # elements from position 0 (included) to 2 (excluded)
vec_slice
```

[35]:
```
array([0, 1])
```

[36]:
```
vec[2:5] # elements from position 2 (included) to 5 (excluded)
```

[36]:
```
array([ 4,  9, 16])
```

[37]:
```
vec[:2]   # elements from the beginning to position 2 (excluded) --- equivalent␣
 ↪to vec[0:2]
```

[37]:
```
array([0, 1])
```

[38]:
```
vec[-2:]  # elements from the second-last (included) to the end
```

[38]:
```
array([64, 81])
```

The role of the step:

```
[39]: vec[1:7:2] # from position 1 (included) to 7 (excluded), each 2 elements
```

```
[39]: array([ 1,  9, 25])
```

```
[40]: vec[::2] # from the beginning to the end of vec, each 2 elements
```

```
[40]: array([ 0,  4, 16, 36, 64])
```

and this is how to revert the vector:

```
[41]: vec
```

```
[41]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
[42]: vec[::-1] # literally from the beginning to the end of vec, each element␣
       ↪starting from the last one
```

```
[42]: array([81, 64, 49, 36, 25, 16,  9,  4,  1,  0])
```

2.2.3 Assigning new values

```
[43]: vec
```

```
[43]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

To change a single value:

```
[44]: vec[2]
```

```
[44]: 4
```

```
[45]: vec[2] = -17
      vec
```

```
[45]: array([  0,   1, -17,   9,  16,  25,  36,  49,  64,  81])
```

To change a whole slice:

```
[46]: vec[4:7]
```

```
[46]: array([16, 25, 36])
```

```
[47]: vec[4:7] = 1000
      vec
```

```
[47]: array([   0,    1,  -17,    9, 1000, 1000, 1000,   49,   64,   81])
```

2.2.4 Iterating over arrays

Iteration happen as in Python lists:

- over the array;
- counter-based (using `np.arange`)
- using the `enumerate()` function

let's quickly review the three methods

[48]:
```
vec = np.array([i**2 for i in range(10)])
vec
```

[48]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Iteration over the array: no access to the indexes

[49]:
```
from math import sqrt

for square in vec:
    print(sqrt(square))
```

```
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
```

Counter-based iteration using `np.arange()`:

[50]:
```
for i in np.arange(vec.size):
    print(vec[i])
```

```
0
1
4
9
16
25
36
49
64
81
```

Enumeration of indexes and values using `enumerate()`: access to both indexes and values

```
[51]: for i, vec_i in enumerate(vec):
          print("i = {}, vec_i = {}".format(i, vec_i)) # vec_i is equivalent to vec[i]
```

```
i = 0, vec_i = 0
i = 1, vec_i = 1
i = 2, vec_i = 4
i = 3, vec_i = 9
i = 4, vec_i = 16
i = 5, vec_i = 25
i = 6, vec_i = 36
i = 7, vec_i = 49
i = 8, vec_i = 64
i = 9, vec_i = 81
```

2.3 Basic operations (are *element-wise*)

Basic array operations are computed element-wise. We can distinguish a couple of relevant cases:

```
[52]: vec = np.array([i**2 for i in range(10)])
      vec
```

```
[52]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

- array + number: number is added to each element of array (same for -, *, /, ** operators)

```
[53]: vec + 100 # equivalent to 100 + vec
```

```
[53]: array([100, 101, 104, 109, 116, 125, 136, 149, 164, 181])
```

```
[54]: vec - 1
```

```
[54]: array([-1,  0,  3,  8, 15, 24, 35, 48, 63, 80])
```

```
[55]: vec * 2 # equivalent to 2 * vec
```

```
[55]: array([  0,   2,   8,  18,  32,  50,  72,  98, 128, 162])
```

```
[56]: vec_new = vec / 3 # note the casting to float
      print("dtype: ", vec_new.dtype)
      vec_new
```

```
dtype:  float64
```

```
[56]: array([ 0.        ,  0.33333333,  1.33333333,  3.        ,  5.33333333,
              8.33333333, 12.        , 16.33333333, 21.33333333, 27.        ])
```

```
[57]: vec ** 2
```

```
[57]: array([   0,    1,   16,   81,  256,  625, 1296, 2401, 4096, 6561],
            dtype=int32)
```

- array + array: elements are added element-wise (same for -, *, / operators)

```
[58]: vec_rev = vec[::-1]
      vec_rev
```

```
[58]: array([81, 64, 49, 36, 25, 16,  9,  4,  1,  0])
```

```
[59]: vec + vec_rev # equivalent to vec_rev + vec
```

```
[59]: array([81, 65, 53, 45, 41, 41, 45, 53, 65, 81])
```

```
[60]: vec - vec_rev
```

```
[60]: array([-81, -63, -45, -27,  -9,   9,  27,  45,  63,  81])
```

```
[61]: vec * vec_rev # equivalent to vec_rev * vec
```

```
[61]: array([  0,  64, 196, 324, 400, 400, 324, 196,  64,   0])
```

```
[62]: vec / vec_rev  # notice the 'inf' and the warning when we divide by zero some␣
      ↪elements
```

```
C:\Users\gabri\Anaconda3\envs\ITForBusAndFin2020_env\lib\site-
packages\ipykernel_launcher.py:1: RuntimeWarning: divide by zero encountered in
true_divide
  """Entry point for launching an IPython kernel.
```

```
[62]: array([0.00000000e+00, 1.56250000e-02, 8.16326531e-02, 2.50000000e-01,
             6.40000000e-01, 1.56250000e+00, 4.00000000e+00, 1.22500000e+01,
             6.40000000e+01,            inf])
```

```
[63]: vec_rev
```

```
[63]: array([81, 64, 49, 36, 25, 16,  9,  4,  1,  0])
```

```
[64]: vec ** vec_rev # each element of vec is exponentiated to the corresponding␣
      ↪element of vec_rev
```

```
[64]: array([          0,           1,           0,  -919996767,           0,
             -2052264063,  1159987200,     5764801,          64,           1],
            dtype=int32)
```

2.3.1 *Focus on:* * operator on lists

On Lists, operator * have completely different behavior: it repeats (*) the list

```
[65]: lis = [i**2 for i in range(10)]
      lis
```

```
[65]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[66]: lis * 2
```

```
[66]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2.3.2 Built-in methods: `.min()`, `.max()`, `.sum()` and more

```
[67]: vec
```

```
[67]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

the sum of its elements is

```
[68]: vec.sum()
```

```
[68]: 285
```

its minimum and maximum elements are

```
[69]: print(vec.min())
      print(vec.max())
```

```
0
81
```

its mean and standard deviation (useful when working with arrays made from random values)

```
[70]: vec.mean()
```

```
[70]: 28.5
```

```
[71]: vec.std()
```

```
[71]: 26.852374196707448
```

2.3.3 Universal functions

Universal functions are functions which are general enough to work both on NumPy's arrays (element-wise) and basic Python data types. We can just name a few, but there are many (many) more. Take a look here

```
[72]: vec
```

```
[72]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
[73]: np.exp(vec) # exponential of each element of vec
```

```
[73]: array([1.00000000e+00, 2.71828183e+00, 5.45981500e+01, 8.10308393e+03,
             8.88611052e+06, 7.20048993e+10, 4.31123155e+15, 1.90734657e+21,
             6.23514908e+27, 1.50609731e+35])
```

```
[74]: np.exp(3) # e^3
```

```
[74]: 20.085536923187668
```

```
[75]: np.sqrt(vec)
```

```
[75]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

```
[76]: np.sqrt(3)
```

```
[76]: 1.7320508075688772
```

3 N-dim arrays

Numpy arrays are born to be multi-dimensional. So we now talk about the NumPy's main object: the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called axes.

We'll focus on the case $N = 2$, which is the case of matrices. But what follows extends to any dimension $N > 2$ too.

3.1 3.1. Array Creation

In my experience, N-dim arrays are most easily created using placeholder functions like `np.ones()`, `np.zeros()` and `np.empty()` and then change the initiliazed values with our data.

But, let's'quickly review the other methods too. As in the 1-dim case, they can be created:

- from Lists or Tuples;
- using placeholder functions.

3.1.1 From Lists or Tuples

Nested lists can be used to define arrays. In general, funciton `np.array()` transforms: - sequences of sequences into two-dimensional arrays, - sequences of sequences of sequences into three-dimensional arrays,

and so on.

Let's make an example with a list of lists, first five even numbers in the first list and first five odds numbers in the second.

```
[77]: lis_of_lis = [[i for i in range(10) if i%2 == 0], [i for i in range(10) if i%2 !
      ↪= 0]]
```

```
lis_of_lis
```

```
[77]: [[0, 2, 4, 6, 8], [1, 3, 5, 7, 9]]
```

In passing, notice the flexibility of the lst comprehension, which allows to define a list from a loop with an `if` condition.

Let's now define the corresponding array:

```
[78]: mat = np.array(lis_of_lis)
      mat
```

```
[78]: array([[0, 2, 4, 6, 8],
             [1, 3, 5, 7, 9]])
```

which we have called `mat` as it can indeed represent a 2×5 matrix of integers

```
[79]: type(mat)
```

```
[79]: numpy.ndarray
```

As said for the 1-dim case, I defined separately the list `lis_of_lis` just for clarity. The above definition is equivalent to

```
[80]: mat = np.array([[i for i in range(10) if i%2 == 0], [i for i in range(10) if␣
       ↪i%2 != 0]])
      mat
```

```
[80]: array([[0, 2, 4, 6, 8],
             [1, 3, 5, 7, 9]])
```

Let's have a look at `mat` meta-informations. First of all, `mat` is 2-dimensional:

```
[81]: mat.ndim
```

```
[81]: 2
```

now its shape: it has 2 rows with 5 values each

```
[82]: mat.shape
```

```
[82]: (2, 5)
```

The number of elements:

```
[83]: mat.size
```

```
[83]: 10
```

Now observe the output of `len()` function

[84]:
```
len(mat)
```

[84]:
```
2
```

`len()` applied to a multi-dimensional array simply returns the *length* of the first axes which - for a 2-dim array - simply is the number of rows. It's not what we expected, that's why I reccomended to use `.size` instead.

`.dtype` considerations are as in the 1-dim case:

[85]:
```
mat.dtype
```

[85]:
```
dtype('int32')
```

3.1.2 *Focus on:* printing arrays

When you print an array, NumPy displays it in this way: - 1-dim arrays are printed as *rows* (as we have seen before); - 2-dim arrays are printed as matrices (*rows* and *cols*); - 3-dim arrays are printed as several matrices, one for each *page*

[86]:
```
l1 = [i for i in np.arange(3)]
l2 = [i**2 for i in np.arange(3)]
l3 = [i**3 for i in np.arange(3)]

print("l1: ", l1)
print("l2: ", l2)
print("l3: ", l3)
```

```
l1:  [0, 1, 2]
l2:  [0, 1, 4]
l3:  [0, 1, 8]
```

1-dim: vector of length 3

[87]:
```
vec = np.array(l1)
print(vec.ndim)
vec
```

```
1
```

[87]:
```
array([0, 1, 2])
```

2-dim: (2,3) matrix

[88]:
```
mat = np.array([l1,l2])

print(mat.ndim)
print(mat.shape)
mat
```

```
2
(2, 3)
```

[88]:
```
array([[0, 1, 2],
       [0, 1, 4]])
```

3-dim case: 4 nested lists of 2 lists of length 3 each, which become 4 distinct 2×3 matrices of numbers

[89]:
```
tensor = np.array([[l1,l1], [l2,l2], [l3,l3], [l1,l2]])

print(tensor.ndim)
print(tensor.shape)
tensor
```

```
3
(4, 2, 3)
```

[89]:
```
array([[[0, 1, 2],
        [0, 1, 2]],

       [[0, 1, 4],
        [0, 1, 4]],

       [[0, 1, 8],
        [0, 1, 8]],

       [[0, 1, 2],
        [0, 1, 4]]])
```

3.1.3 Using placeholder content: role of the `shape` parameter

As in 1-dim case, when you know how many values you need to store, how you want to arrange them, but you don't know which values to input in advance, you can use one of these initialized functions:

- `np.ones(shape[, dtype])`: creates an array of ones;
- `np.zeros(shape[, dtype])`: creates an array of zeros;
- `np.empty(shape[, dtype])`: creates an empty array, where the actual values depend on the state of memory.

In each of these,

- `dtype` is optional and, default set to `float64` (NumPy's `float` type);
- `shape` is a Tuple number corresponding to the `.shape` attribute of the newly created array.

[90]:
```
mat = np.ones((2,5))
print("dtype: ", mat.dtype)
mat
```

```
dtype:  float64
```

```
[90]: array([[1., 1., 1., 1., 1.],
             [1., 1., 1., 1., 1.]])
```

```
[91]: mat = np.ones((2,5), dtype="int")
      print("dtype: ", mat.dtype)
      mat
```

```
dtype:  int32
```

```
[91]: array([[1, 1, 1, 1, 1],
             [1, 1, 1, 1, 1]])
```

```
[92]: mat = np.zeros((2,5))
      print("dtype: ", mat.dtype)
      mat
```

```
dtype:  float64
```

```
[92]: array([[0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.]])
```

```
[93]: mat = np.zeros((2,5), dtype="int")
      print("dtype: ", mat.dtype)
      mat
```

```
dtype:  int32
```

```
[93]: array([[0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0]])
```

```
[94]: mat = np.empty((2,5))
      print("dtype: ", mat.dtype)
      mat
```

```
dtype:  float64
```

```
[94]: array([[0., 0., 0., 0., 0.],
             [0., 0., 0., 0., 0.]])
```

these values filled by `.empty()` could have been whatever depending on the current state of the memory

3.2 3.2. Indexing, Slicing, Assigning and Iterating

Multi-dimensional arrays have one index per axis. These indices are given as a comma-separated sequence. Each index follows the rules of 1-dim arrays. We'll focus here on the 2-dim case. Generalization to the $N > 2$-dim case should be easy.

3.2.1 Indexing

Let's re-define our `mat` matrix

[95]:
```
mat = np.array([[i for i in range(10) if i%2 == 0], [i for i in range(10) if␣
 ↪i%2 != 0]])
mat
```

[95]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

The 2-dimensional array `mat` can be indexed as

`mat[i,j]`

where `i` is the index of the rows and `j` the one of columns. Each index behaves as in the 1-dim case

[96]:
```
print(mat[0,1])
type(mat[0,1])
```

```
2
```

[96]:
```
numpy.int32
```

[97]:
```
print(mat[1,-1])
```

```
9
```

3.2.2 Slicing

The 2-dimensional array `mat` can be sliced as

`mat[i:j:k, m:n:q]`

where for each dimension, indexes have start, stop, step meanings as in the 1-dim case

The role of the `:` (colon) is that of range selector. Here is the full range `mat[:]`

[98]:
```
mat
```

[98]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

[99]:
```
mat[:,:] # equivalent to simply mat
```

[99]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

[100]:
```
mat[:, 0:2] # elements from column 0 (included) to column 2 (excluded), all rows
```

[100]:
```
array([[0, 2],
       [1, 3]])
```

```
[101]: mat[1, :] # the second row
```

```
[101]: array([1, 3, 5, 7, 9])
```

The role of the step is the same as in the 1-dim case for each dim

```
[102]: mat[0, ::2] # elements from the first row, each 2 columns
```

```
[102]: array([0, 4, 8])
```

```
[103]: mat[::-1, 2] # column 3, reading bottom-up the rows
```

```
[103]: array([5, 4])
```

3.2.3 Assigning new values

```
[104]: mat
```

```
[104]: array([[0, 2, 4, 6, 8],
               [1, 3, 5, 7, 9]])
```

To change a single value:

```
[105]: mat[1,2]
```

```
[105]: 5
```

```
[106]: mat[1,2] = -17
       mat
```

```
[106]: array([[  0,   2,   4,   6,   8],
               [  1,   3, -17,   7,   9]])
```

To change a whole slice:

```
[107]: mat[0, 2:5]
```

```
[107]: array([4, 6, 8])
```

```
[108]: mat[0, 2:5] = 1000
       mat
```

```
[108]: array([[   0,    2, 1000, 1000, 1000],
               [   1,    3,  -17,    7,    9]])
```

3.2.4 Iterating over N-dim arrays

Because of celebrated *vectorization* and Universal functions which apply element-wise to each array, `for` loops over multi-dim arrays are rarely used in practice.

For your knowledge, you can loop as you would do in the 1-dim case, but consider that the loops will be done over the first axis (rows in 2-dim case):

[109]:
```
mat = np.array([[i for i in range(10) if i%2 == 0], [i for i in range(10) if␣
 ↪i%2 != 0]])
mat
```

[109]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

[110]:
```
for row in mat:
    print(row)
```

```
[0 2 4 6 8]
[1 3 5 7 9]
```

To loop over each element you can do a nested loop

[111]:
```
for row in mat:
    for element in row:
        print(element)
```

```
0
2
4
6
8
1
3
5
7
9
```

which in real-life situation is highly inefficient and I discourage you to use it.

3.3 Basic operations (are *element-wise*)

As in the 1-dim case, basic array operations are computed element-wise.

[112]:
```
mat = np.array([[i for i in range(10) if i%2 == 0], [i for i in range(10) if␣
 ↪i%2 != 0]])
mat
```

[112]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

[113]:
```
mat * 2
```

[113]:
```
array([[ 0,  4,  8, 12, 16],
       [ 2,  6, 10, 14, 18]])
```

and similarly for other mathematical operators.

3.3.1 *Focus on:* Matrix Operations

The product operators works element-wise between matrices too:

[114]:
```
A = mat[:,:2]
print("Shape of A: (n,m) = ", A.shape)
A
```

```
Shape of A: (n,m) =  (2, 2)
```

[114]:
```
array([[0, 2],
       [1, 3]])
```

[115]:
```
B = mat[:,-2:]
print("Shape of B: (n,m) = ", B.shape)
B
```

```
Shape of B: (n,m) =  (2, 2)
```

[115]:
```
array([[6, 8],
       [7, 9]])
```

such that the (i, j) element of matrix `A*B` is

$$(A * B)_{ij} = A_{ij} \times B_{ij}$$

which is not the element of the product-matrix of `A` and `B`...

[116]:
```
C_elementwise = A*B
C_elementwise
```

[116]:
```
array([[ 0, 16],
       [ 7, 27]])
```

The matrix product can be implemented using the `@` operator, such that the (i, j) element of matrix `A @ B` is

$$(A@B)_{ij} = \sum_{k=0}^{1} A_{ik} \times B_{kj}$$

given the common shape $(n, m) = (2, 2)$, which makes `A @ B` the real product-matrix of `A` and `B`.

```
[117]: AB = A @ B
       AB
```

```
[117]: array([[14, 18],
              [27, 35]])
```

From linear algebra, recall that two matrices A - of shape (n, m) - and D - of shape (p, q) - an be multiplied together only if $m = q$.

```
[118]: D = mat[:,:3]
       print("Shape of D: (p,q) = ", D.shape)
       D
```

```
Shape of D: (p,q) =  (2, 3)
```

```
[118]: array([[0, 2, 4],
              [1, 3, 5]])
```

therefore if you try to do `A @ D`, then no problem because $m = q = 2$ and the product matrix of the shapes

$$(n, m) = (2, 2) \times (2, 3) = (p, q)$$

will be a matrix of shape $(m, p) = (2, 3)$

```
[119]: AD = A @ D
       print("Shape of A: (n,m) = ", A.shape)
       print("Shape of D: (p,q) = ", D.shape)
       print("Shape of AD: (m,p) = ", AD.shape)
       AD
```

```
Shape of A: (n,m) =  (2, 2)
Shape of D: (p,q) =  (2, 3)
Shape of AD: (m,p) =  (2, 3)
```

```
[119]: array([[ 2,  6, 10],
              [ 3, 11, 19]])
```

whereas if you try to multiply `D @ A`, this would be a product of matrices of shapes

$$(p, q) = (2, 3) \times (2, 2) = (n, m)$$

which is *forbidden* mathematically because of mismatch of the inner shapes: $q = 3 \neq 2 = n$.

This consistently results in the raise of the error:

```
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signatu
```

This (rather verbose) error message is simply warning you that of the mentioned mismatch between q and n, which makes impossible to do `D @ A`

[120]:
```
# This raises: ValueError: matmul: Input operand 1 has a mismatch in its core␣
↪dimension 0,
#                            with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2␣
↪is different from 3)

# D @ A
```

Finally, notice that the element-wise products are (in general) defined only if the two matrices have the same shape.

This is to say that `A*D`, which would try to do the element-wise product of matrices of shapes $(n, m) = (2, 2)$ and $(p, q) = (2, 3)$, raises:

`ValueError: operands could not be broadcast together with shapes (2,2) (2,3)`

[121]:
```
# This raises: ValueError: operands could not be broadcast together with shapes␣
↪(2,2) (2,3)

# A*D
```

and, symmetrically, `D*A`, which would try to do the element-wise product of matrices of shapes $(p, q) = (2, 3)$ and $(n, m) = (2, 2)$, raises:

`ValueError: operands could not be broadcast together with shapes (2,3) (2,2)`

[122]:
```
# This raises: ValueError: operands could not be broadcast together with shapes␣
↪(2,3) (2,2)

# D*A
```

3.3.2 Built-in methods: role of the `axis` parameter

These built-in methods work the same as in the 1-dim case, but allow to specify the axis too:

[123]:
```
mat
```

[123]:
```
array([[0, 2, 4, 6, 8],
       [1, 3, 5, 7, 9]])
```

Sum of all elements

[124]:
```
mat.sum()
```

[124]:
```
45
```

Sum row-wise (that is, over rows for each column)

```
[125]: mat.sum(axis=0)
```

```
[125]: array([ 1,  5,  9, 13, 17])
```

Sum column-wise (that is, over columns for each row)

```
[126]: mat.sum(axis=1)
```

```
[126]: array([20, 25])
```

3.3.3 Universal functions

Same as 1-dim case

```
[127]: mat
```

```
[127]: array([[0, 2, 4, 6, 8],
              [1, 3, 5, 7, 9]])
```

```
[128]: np.exp(mat)
```

```
[128]: array([[1.00000000e+00, 7.38905610e+00, 5.45981500e+01, 4.03428793e+02,
               2.98095799e+03],
              [2.71828183e+00, 2.00855369e+01, 1.48413159e+02, 1.09663316e+03,
               8.10308393e+03]])
```

3.4 3.4. Shape Manipulation

As we have seen, the shape of an array is the number of elements along each axis (read: dimensions)

```
[129]: A = np.array([[i for i in range(4)],
                     [i**2 for i in range(4)],
                     [i**3 for i in range(4)]])
       A
```

```
[129]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[130]: A.shape
```

```
[130]: (3, 4)
```

```
[131]: A.size
```

```
[131]: 12
```

so matrix `A`, which is for NumPy a 2-dim `numpy.ndarray`, has its 12 elements arranged in 3 rows and 4 columns.

[132]:
```
print("Rows of A: ", A.shape[0])
print("Columns of A: ", A.shape[1])
```

```
Rows of A:  3
Columns of A:  4
```

Notice that it's always true that the product of the numbers in the `.shape` Tuple is the `.size` of the array.

In our example: `A.shape[0]` $\times$ `A.shape[1]` $= 3 \times 4 = 12 =$ `A.size`

3.4.1 Changing the shape: `.reshape()`

We can change the shape of an array. That is, keeping the total number of elements constant, we can change the way in which they are arranged. There is an `numpy.ndarray` method to do this:

`array.reshape(shape)`

where `array` is the original array and `shape` is the new shape Tuple. The product of the `shape` Tuple must match the original `array.size`.

For example, instead of arranging the $3 \times 4 = 12$ elements of `A` in a $(3, 4)$ shape, we could arrange them as $(2, 6)$ shape because $2 \times 6 = 12$

[133]:
```
A_26 = A.reshape((2,6))
A_26
```

[133]:
```
array([[ 0,  1,  2,  3,  0,  1],
       [ 4,  9,  0,  1,  8, 27]])
```

which has a $(2, 6)$ shape

[134]:
```
A_26.shape
```

[134]:
```
(2, 6)
```

but still the same number of elements of `A`

[135]:
```
A_26.size
```

[135]:
```
12
```

Under the hoods: `.reshape()` is basically simply changing the *view* of our array, without creating a new array. This is a cheap operation.

Notice that if you (intentionally or unintentionally) try to reshape an array with a new shape that wouldn't keep constant the number of items, an error is raised.

For example, instead of arranging the $3 \times 4 = 12$ elements of `A` in a $(3, 4)$ shape, we try to arrange them in a $(2, 8)$ shape… this raises the error

```
ValueError: cannot reshape array of size 12 into shape (2,8)
```

because `A_28.size` $= 2 \times 8 = 16 \neq 12 =$ `A.size`.

```
[136]: A
```

```
[136]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[137]: # This raises: ValueError: cannot reshape array of size 12 into shape (2,8)

       # A_28 = A.reshape((2,8))
```

3.4.2 *Focus on:* Matrix Transpose `.T`

A particular kind of reshape is the exchange of rows for columns, which is the matematical operation of *transposition* of a matrix.

```
[138]: A
```

```
[138]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

The transpose of A is simply calculated using the `.T` attribute of any array:

```
[139]: A_t = A.T
       A_t
```

```
[139]: array([[ 0,  0,  0],
              [ 1,  1,  1],
              [ 2,  4,  8],
              [ 3,  9, 27]])
```

```
[140]: A_t.shape
```

```
[140]: (4, 3)
```

```
[141]: A_t.size
```

```
[141]: 12
```

3.4.3 Changing the size: `.resize()`

While, as we said, `.reshape()` simply change the view of an array, there is another method which can change the shape as well, but relaxing the constraint of having the `.size` constant. The function to do this is:

```
np.resize(array, shape)
```

Notice that `.resize()` will, in general, create a new array. This is then an expensive operation, which should be avoided as much as you can.

```
[142]: A = np.array([[i for i in range(4)],
                     [i**2 for i in range(4)],
                     [i**3 for i in range(4)]])
       A
```

```
[142]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

With `.resize()` we can do the same things we did with `.reshape()`, in particular simply rearranging the elements in a different shape

```
[143]: A_26 = np.resize(A, (2,6))
       A_26
```

```
[143]: array([[ 0,  1,  2,  3,  0,  1],
              [ 4,  9,  0,  1,  8, 27]])
```

```
[144]: A_26.shape
```

```
[144]: (2, 6)
```

```
[145]: A_26.size
```

```
[145]: 12
```

or you can down-size (that is, select few numbers)

```
[146]: A_23 = np.resize(A, (2,3))
       A_23
```

```
[146]: array([[0, 1, 2],
              [3, 0, 1]])
```

```
[147]: A_23.shape
```

```
[147]: (2, 3)
```

```
[148]: A_23.size
```

```
[148]: 6
```

or you can up-size the array

```
[149]: A_64 = np.resize(A, (6,4))
       A_64
```

```
[149]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27],
              [ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[150]: A_64.shape
```

```
[150]: (6, 4)
```

```
[151]: A_64.size
```

```
[151]: 24
```

Notice that while a a down-sizing will select a few of the elements of the original array, an up-sizing will fill the additional slots repeating elements of the original array.

3.4.4 From N-dim to 1-dim: `.flatten()`

If you want to reshape your multi-dimensional array rearranging its elements as a 1-dim array (useful to iterate over all its elements), you can use the `.flatten()` method

```
[152]: A
```

```
[152]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[153]: A.size
```

```
[153]: 12
```

```
[154]: A_1dim = A.flatten()
       A_1dim
```

```
[154]: array([ 0,  1,  2,  3,  0,  1,  4,  9,  0,  1,  8, 27])
```

```
[155]: A_1dim.size
```

```
[155]: 12
```

3.5 Stacking Arrays together: `.hstack()` and `.vstack()`

You can stack together arrays, either vertically:

[156]:
```
A
```

[156]:
```
array([[ 0,  1,  2,  3],
       [ 0,  1,  4,  9],
       [ 0,  1,  8, 27]])
```

[157]:
```
B = 10 * A
B
```

[157]:
```
array([[  0,  10,  20,  30],
       [  0,  10,  40,  90],
       [  0,  10,  80, 270]])
```

[158]:
```
AB_vstack = np.vstack((A,B))
AB_vstack
```

[158]:
```
array([[  0,   1,   2,   3],
       [  0,   1,   4,   9],
       [  0,   1,   8,  27],
       [  0,  10,  20,  30],
       [  0,  10,  40,  90],
       [  0,  10,  80, 270]])
```

or horizontally

[159]:
```
AB_hstack = np.hstack((A,B))
AB_hstack
```

[159]:
```
array([[  0,   1,   2,   3,   0,  10,  20,  30],
       [  0,   1,   4,   9,   0,  10,  40,  90],
       [  0,   1,   8,  27,   0,  10,  80, 270]])
```

3.6 Indexing with Boolean Arrays

So far, to select elements or slices of an array we have provided directly the integer indexes `i` or range of indexes `i:j:k` to each dimension of the array

[160]:
```
A
```

[160]:
```
array([[ 0,  1,  2,  3],
       [ 0,  1,  4,  9],
       [ 0,  1,  8, 27]])
```

[178]:
```
A[2,3]
```

[178]:
```
27
```

[179]:
```
A[1,1:3]
```

```
[179]: array([1, 4])
```

Another typical situation that you can encounter in practice is that you want to select only portions of the array that satisfy some logical condition(s). This is achieved quite easily in NumPy using Boolean arrays.

3.6.1 Boolean Arrays

Boolean arrays are arrays with a boolean `.dtype` and are the result of the action of *comparison operators* `>`, `<`, `>=`, `<=`, `==` and `!=`. For example

```
[184]: A
```

```
[184]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[185]: A > 2.5
```

```
[185]: array([[False, False, False,  True],
              [False, False,  True,  True],
              [False, False,  True,  True]])
```

what is returned by the operation `A > 2.5` is actually a booalean array

```
[186]: cond = (A > 2.5)
       cond
```

```
[186]: array([[False, False, False,  True],
              [False, False,  True,  True],
              [False, False,  True,  True]])
```

```
[187]: cond.dtype
```

```
[187]: dtype('bool')
```

As expected, boolean arrays can be combined using the *logical operators* `&` (logical and), `|` (logical or) and `!` (logical not)

```
[190]: cond = (A > 2.5) & (A < 10)
       cond
```

```
[190]: array([[False, False, False,  True],
              [False, False,  True,  True],
              [False, False,  True, False]])
```

```
[191]: cond.dtype
```

```
[191]: dtype('bool')
```

3.6.2 Conditional Selection

We can use a boolean array to select only portions of an array for which the corresponding boolean array has `True` value.

```
[192]: A
```

```
[192]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[193]: cond
```

```
[193]: array([[False, False, False,  True],
              [False, False,  True,  True],
              [False, False,  True, False]])
```

this is done in an obvious way:

```
[195]: A_c = A[cond]
       A_c
```

```
[195]: array([3, 4, 9, 8])
```

notice that that this conditional selection returns a flattened array

```
[196]: print(A.ndim)
       print(A.shape)
```

```
2
(3, 4)
```

```
[197]: print(A_c.ndim)
       print(A_c.shape)
```

```
1
(4,)
```

If your goal is to change some values according to a condition, you can simply use the boolean array for indexing and then assign the value to the selection

```
[198]: A
```

```
[198]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[199]: cond
```

[199]:
```
array([[False, False, False,  True],
       [False, False,  True,  True],
       [False, False,  True, False]])
```

[200]:
```
A[cond] = -1
A
```

[200]:
```
array([[ 0,  1,  2, -1],
       [ 0,  1, -1, -1],
       [ 0,  1, -1, 27]])
```

as you can see, value `-1` is assigned to all (and only) elements of `A` for which `cond` was `True`. The shape is not affected by this assignment.

4 Lists Vs Arrays

4.1 Speed-comparison: Lists 0 - 1 Arrays

Let's compare the execution time of some simple function first over a list and then over a (comparable) array.

Let's consider a nested list made of $m = 5000$ lists, each made of $n = 1000$ elements which are sum of numbers $i + k$ defined as follow

[161]:
```
n = 1000
m = 5000
```

[162]:
```
l = [[i+k for i in range(n)] for k in range(m)]
```

To visualize the structure of the list try with smaller values like

[163]:
```
l_small = [[i+k for i in range(3)] for k in range(5)]
l_small
```

[163]:
```
[[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
```

Let's now compute and time the creation of new list which is made out of `l`, taking the square `**2` of each of its elements

[164]:
```
%time l_squared = [[i**2 for i in lis] for lis in l]
```

```
Wall time: 1.91 s
```

so it took more than 1.5 seconds to compute the list of squares and assign it to `l_squared`

let's define the corresponding array (notice that the printing of NumPy arrays is also smart enough to work with huge arrays too)

[165]:
```
arr = np.array(l)
```

[166]:
```
arr
```

[166]:
```
array([[   0,    1,    2, …,  997,  998,  999],
       [   1,    2,    3, …,  998,  999, 1000],
       [   2,    3,    4, …,  999, 1000, 1001],
       …,
       [4997, 4998, 4999, …, 5994, 5995, 5996],
       [4998, 4999, 5000, …, 5995, 5996, 5997],
       [4999, 5000, 5001, …, 5996, 5997, 5998]])
```

[167]:
```
arr.shape
```

[167]:
```
(5000, 1000)
```

and let's compute the squared array: it is just `arr**2`

[168]:
```
%time arr_squared = arr ** 2
```

```
Wall time: 14 ms
```

and this took order of milli-seconds!!!

[169]:
```
arr_squared
```

[169]:
```
array([[       0,        1,        4, …,   994009,   996004,   998001],
       [       1,        4,        9, …,   996004,   998001,  1000000],
       [       4,        9,       16, …,   998001,  1000000,  1002001],
       …,
       [24970009, 24980004, 24990001, …, 35928036, 35940025, 35952016],
       [24980004, 24990001, 25000000, …, 35940025, 35952016, 35964009],
       [24990001, 25000000, 25010001, …, 35952016, 35964009, 35976004]],
      dtype=int32)
```

4.2 *Vectorization* of code: Lists 0 - 2 Arrays

The idea of vectorization is that of applying an operation or a function on an array object "at once", instead of looping over the array and applying it on each element. Needless, to say that there is considerable speed gain.

We have already seen examples of vectorizations: array + numbers and array + array operations are all example of it beacause we delegate to the operator `+` (but also works for other operators of course) the looping over the array:

[170]:
```
A
```

[170]:
```
array([[ 0,  1,  2,  3],
       [ 0,  1,  4,  9],
       [ 0,  1,  8, 27]])
```

```
[171]: A + 2
```

```
[171]: array([[ 2,  3,  4,  5],
              [ 2,  3,  6, 11],
              [ 2,  3, 10, 29]])
```

is equivalent to

```
[172]: B = np.zeros(A.shape, dtype="int")
       print("B initialized: ")
       print(B)

       for i, A_row in enumerate(A):
           #print("i = {}; A_row = {}".format(i, A_row))
           for j, A_ij in enumerate(A_row):
               #print("j = {}; A_ij = {}".format(j, A_ij))
               B[i,j] = A_ij + 2

       print("B = A+2:")
       print(B)
```

```
B initialized:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
B = A+2:
[[ 2  3  4  5]
 [ 2  3  6 11]
 [ 2  3 10 29]]
```

which is way more complicated, error-prone and inefficient (slower).

The same when you combine arrays element-wise

```
[173]: A
```

```
[173]: array([[ 0,  1,  2,  3],
              [ 0,  1,  4,  9],
              [ 0,  1,  8, 27]])
```

```
[174]: B
```

```
[174]: array([[ 2,  3,  4,  5],
              [ 2,  3,  6, 11],
              [ 2,  3, 10, 29]])
```

```
[175]: C = A*B
       C
```

[175]:
```
array([[  0,   3,   8,  15],
       [  0,   3,  24,  99],
       [  0,   3,  80, 783]])
```

this is equivalent to

[176]:
```
C = np.zeros(A.shape, dtype="int")
print("C initialized: ")
print(C)

for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        C[i,j] = A[i,j] * B[i,j]

print("C = A*B: ")
print(C)
```

```
C initialized:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
C = A*B:
[[  0   3   8  15]
 [  0   3  24  99]
 [  0   3  80 783]]
```

Basically the idea is that, when you want to do an operation or applying a function to each element of one or more arrays, simply treat the array as if it is a single number. NumPy - in most of the cases - will recognize that that operation or function has to be distributed over each element of the array. NumPy will take care of the distribution of the application of the operation or function over the array, without the need for you to explicitly code the loops…

Basic Python lists have limited functionalities in this regard. Worth to mention are `map()` and `filter()` methods for lists, which to be honest I barely never used. 2-0.