Introduction

An *array* is a numpy object (whose type is `numpy.ndarray`). It is similar to Python lists but suited for mathematical operations.

In [1]:
```
import numpy as np
var = [1, 2, 3]
print(type(var))
arr = np.array([1, 2, 3])
print(type(arr))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

Create an array

One can create an array from an iterable (ex: list, see above) or use dedicated `numpy` functions:

```
In [2]:
print(np.ones(5))
print(np.arange(2, 42, 5))      # similar to `range`
print(np.zeros(5))
```

```
[1. 1. 1. 1. 1.]
[ 2  7 12 17 22 27 32 37]
[0. 0. 0. 0. 0.]
```

Functions `linspace` and `logspace` define regularly spaced values in a linear space or logarthmic space.

Using `linspace`: the difference between two consecutive elements is constant:

In [3]:
```
arr = np.linspace(1, 10, 5)
print(arr)
print(arr[1]-arr[0])
print(arr[2]-arr[1])
```

```
[ 1.    3.25  5.5   7.75 10.  ]
2.25
2.25
```

Using `logspace`: the ratio of two consecutive elements is constant:

In [4]:
```
arr = np.logspace(1, 10, 5)
print(arr)
print(arr[1]/arr[0])
print(arr[2]/arr[1])
```

```
[1.00000000e+01 1.77827941e+03 3.16227766e+05 5.62341325e+07
 1.00000000e+10]
177.82794100389228
177.82794100389225
```

Important ideas: axis and dimension

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Introduction

Usually, the length of an iterable is it's number of elements, given by the length function `len`. For an array, there may be **more than one dimension**. Below is a 2-dimensionnal array:

- 2 lines
- 3 columns

In [5]:
```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
print(arr.ndim)
```

```
[[1 2 3]
 [4 5 6]]
2
```

The shape of this array is (2, 3) because:

- it has 2 elements along dimension 1
- it has 3 elements along dimension 2

In [6]:
```
print(arr.shape)
```

```
(2, 3)
```

Instead of "dimension", `numpy` uses the term **"axis"**.

Modify the shape

`shape` gives the actual size of an `array`. But one can change this shape using `reshape`:

In [7]:
```
arr = arr.reshape((3, 2))   # 3 rows, 2 columns: still 6 elements,
                            # hence reshape is possible
print(arr)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

If you don't want several dimensions, the `flatten` method will return all elements along one dimension.

```
In [8]: arr2 = arr.flatten()
        print(arr2)                # all values along a single dimension
```

```
[1 2 3 4 5 6]
```

Note that `flatten` returns a copy, hence if `arr2` is modified `arr` will remain the same.

```
In [9]: arr2[3] = 100              # resulting array is modified
        print(arr)                 # this does not change original array
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Conversely, `ravel` performs the same than `flatten` but uses the same underlying data:

In [10]:
```
arr2 = arr.ravel()
print(arr2)                  # all values along a single dimension
arr2[3] = 100                # resulting array is modified
print(arr)                   # this changes the original array because data is shared in m
```

```
[1 2 3 4 5 6]
[[  1   2]
 [  3 100]
 [  5   6]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Handle axes

Axes of a multidimensionnal array start from 0 (and goes to `ndim-1`). One can index an array following a specific axis the same way it is done for lists.

In [11]:
```
arr = np.zeros((4, 4))
print(arr)
arr[1:3, :2] = 99              # same value for some indexes
print(arr)
arr[2:, 3:4] = [[34], [35]]  # an iterable of the same shape as the modified subarray
print(arr)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[ 0.  0.  0.  0.]
 [99. 99.  0.  0.]
 [99. 99.  0.  0.]
 [ 0.  0.  0.  0.]]
[[ 0.  0.  0.  0.]
 [99. 99.  0.  0.]
 [99. 99.  0. 34.]
 [ 0.  0.  0. 35.]]
```

Let's create a 3-dimensionnal array:

- `axis` 0 has 4 éléments
- `axis` 1 has 3 éléments
- `axis` 2 has 2 éléments

In [12]:
```
arr = np.arange(4 * 3 * 2).reshape((4, 3, 2))
print(arr)
```

```
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Let's extract the elements whose coordinates match:

- 0, 1 or 3 on first axis
- 2 on second axis
- whatever on third axis

In [13]:
```
arr[[0, 1, 3], 2, :]
```

Out[13]:
```
array([[ 4,  5],
       [10, 11],
       [22, 23]])
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Many `numpy` methods take an optional argument `axis` to specify where the mathematical operation must be performed.

For instance, let's compute a mean over axis 1.

In [14]:
```
result = arr.mean(axis=1)
result
```

Out[14]:
```
array([[ 2.,  3.],
       [ 8.,  9.],
       [14., 15.],
       [20., 21.]])
```

Above, **using `axis=1`, `numpy` takes the mean of all elements whose coordinates are all equal, except for axis 1**.

Hence, `result[2, 1]` (3rd row, 1st column) is the mean of:

- `arr[2, 0, 1]` (13)
- `arr[2, 1, 1]` (15)
- `arr[2, 2, 1]` (17)

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Concatenate some arrays

`np.concatenate` gather different arrays into a single instance. Their dimensions must be compatible:

In [15]:
```
arr1 = np.arange(16).reshape((2, 8))
arr2 = np.arange(16, 32).reshape((2, 8))
print(arr1)
print(arr2)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
[[16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
In [16]: print(np.concatenate([arr1, arr2], axis=0))    # shape of axis 0 is increased (i.e: m
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]]
```

```
In [17]: print(np.concatenate([arr1, arr2], axis=1))    # shape of axis 1 is increased (i.e: m
```

```
[[ 0  1  2  3  4  5  6  7 16 17 18 19 20 21 22 23]
 [ 8  9 10 11 12 13 14 15 24 25 26 27 28 29 30 31]]
```

One can also gather arrays along a new dimension, using `np.stack`.

In [18]:
```
arr1 = np.arange(16)
arr2 = np.arange(16, 32)
print(arr1)
print(arr2)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
[16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]
```

In [19]:
```
print(np.stack([arr1, arr2], axis=0))        # `arr1` and `arr2` can be accessed alon
```

```
[[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]]
```

In [20]:
```
print(np.stack([arr1, arr2], axis=1))        # `arr1` and `arr2` can be accessed alon
```

```
[[ 0 16]
 [ 1 17]
 [ 2 18]
 [ 3 19]
 [ 4 20]
 [ 5 21]
 [ 6 22]
 [ 7 23]
 [ 8 24]
 [ 9 25]
 [10 26]
 [11 27]
 [12 28]
 [13 29]
 [14 30]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Split some arrays

You want to define different variables for som parts of an array? Use `np.split`.

In [21]:

```
arr = np.arange(16).reshape((2, 8))
print(arr)
```

```
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]
```

Let's split `arr` along axis 1. `np.split` takes a list of indexes to specify where to split. By giving `[2, 4, 5]`, 4 sub arrays are defined, having these coordinates along axis 1:

- [0, 2[
- [2, 4[
- [4, 5[
- [5, 8[

In [22]:

```
sub1, sub2, sub3, sub4  = np.split(arr, [2, 4, 5], axis=1)
print(sub1, sub2, sub3, sub4, sep='\n')
```

```
[[0 1]
 [8 9]]
[[ 2  3]
 [10 11]]
[[ 4]
 [12]]
[[ 5  6  7]
 [13 14 15]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

A different way is to ask for a fixed number of sub arrays, for instance 4.

```
In [23]: sub1, sub2, sub3, sub4  = np.split(arr, 4, axis=1)
         print(sub1, sub2, sub3, sub4, sep='\n')
```

```
[[0 1]
 [8 9]]
[[ 2  3]
 [10 11]]
[[ 4  5]
 [12 13]]
[[ 6  7]
 [14 15]]
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Modify dimensions

It may happen that an array has only one element along a specific axis. One can delete this dimensions using `np.squeeze`.

In [24]:
```
arr = np.arange(24).reshape((6, 1, 4))
print(arr)
print(arr.shape)
```

```
[[[ 0  1  2  3]]

 [[ 4  5  6  7]]

 [[ 8  9 10 11]]

 [[12 13 14 15]]

 [[16 17 18 19]]

 [[20 21 22 23]]]
(6, 1, 4)
```

```
In [25]: arr = np.squeeze(arr, axis=1)
         print(arr)
         print(arr.shape)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
(6, 4)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Conversely, one can add dimensions using `np.expand_dims`.

```
In [26]:
arr = np.arange(24).reshape((6, 4))
print(arr)
print(arr.shape)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
(6, 4)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

In [27]:
```python
arr = np.expand_dims(arr, axis=1)
print(arr)
print(arr.shape)
```

```
[[[ 0  1  2  3]]

 [[ 4  5  6  7]]

 [[ 8  9 10 11]]

 [[12 13 14 15]]

 [[16 17 18 19]]

 [[20 21 22 23]]]
(6, 1, 4)
```

Common operations

Contrary to lists, numpy arrays handle mathematical operations in a simple way.

```
In [28]: arr = np.arange(5)
```

```
In [29]: arr + arr
```

```
Out[29]:   array([0, 2, 4, 6, 8])
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Operations on arrays are done ***element wise***.

```
In [30]: print(arr * arr)         # product
         print(arr ** 2)          # power
         print(np.exp(arr))       # some function
         print((5*arr+9) % 14)    # modulo
```

```
[ 0  1  4  9 16]
[ 0  1  4  9 16]
[ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
[ 9  0  5 10  1]
```

Note that you can perform matric multiplication using `@`

In [31]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])                         # shape is (2, 3)
arr2 = np.array([[5, 6, 7, 8], [7, 8, 9, 10], [8, 9, 10, 11]])   # shape is (3, 4)
arr @ arr2                                                     # shape is (2, 4)
```

Out[31]:
```
array([[ 43,  49,  55,  61],
       [103, 118, 133, 148]])
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js