The `numpy` Package

Python and Machine Learning

Machine Learning applications typically involve

- Dealing with large amounts of data
- Compuational-heavy algorithms
- Complex pipelines of preprocessing/learning/inference

Is it a good idea to use Python for that?

- Combining operations in complex pipelines is easy in Python
- ...But the language itself is rather slow

And yet, Python is the nowadays the mainstream language for ML

Packages to the Rescue

The trick to get high-performance in Python is using external packages

Dedicated packages can provide:

- Data structures to handle large amount of data
- Efficient algorithms for frequently occurring problems

Both can be implemented in high-performance languages, like C, C++, or Fortran

A fundamental package in this group is called `numpy` and provides:

- A data structure to handle data in tensor format
- Algorithms for a number of common numerical operators

In a nutshell, `numpy` makes Python behave a bit like Matlab

The `numpy.array` Class

The main data structure provided by `numpy` is called `array`

From a math standpoit, it corresponds to a tensor

- A tensor is an n-dimensional collection of elements of a uniform type
- Intuitively, it's a generalization of a vector to n dimensions
- 1 dimension = vector, 2 dimensions = matrix, > 3 dimensions = tensor

From an implementation standpoint:

- Data in an `array` is memorized in a single sequence
- ...But the array also stores a "shape", listing the size of each dimension
- This shape is used to determine how to access the elements

The `numpy.array` Class

Let's see as an example a $2x3$ matrix:

The actual matrix is:

$$\begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} \\ x_{1,0} & x_{1,1} & x_{1,2} \end{pmatrix}$$

...Which is memorized as a sequence by rows, i.e.:

$$\begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{1,0} & x_{1,1} & x_{1,2} \end{pmatrix}$$

- The shape is in this acse $(2, 3)$
- The two-dimension index (i, j) corresponds to the linear index $3i + j$

Using `numpy`

`numpy` is not part of a minimal Python installation

- You can find it pre-installed in scientific Python distributions (e.g. Anaconda)
- ...Or you can install it using a package managed
  - E.g. `pip install numpy`

You can import `numpy` as any other package

...Except that it has a canonical alias, i.e. `np`

```
In [2]: import numpy as np
```

- Extensive documentation can be found online
- ...Or accessed with `help(numpy)` or `help('numpy')`

Building an Array

Any iterable can be converted to an array via the `np.array` constructor:

In [3]:
```
x = [1, 2, 3]
a = np.array(x)
print('Original collection:', x)
print('Array:', a)
```

```
Original collection: [1, 2, 3]
Array: [1 2 3]
```

- The array shape can be access through the `shape` attribute:

In [52]:
```
a.shape
```

Out[52]:
```
(3,)
```

- `shape` is always a tuple (with a single element for one-dimensional arrays)

Building an Array

Using nested iterables leads to multi-dimensional arrays

E.g. a list of list becomes a two-dimensional array

```
In [4]: x = [[1, 2, 3],
             [4, 5, 6]]
        a = np.array(x)
        print('Original collection:', x)
        print('Array:')
        print(a)
        print('Shape:', a.shape)
```

```
Original collection: [[1, 2, 3], [4, 5, 6]]
Array:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)
```

- In this case, the `shape` tuple has two elements
- ...Respectively the number of rows and columns

Building an Array

Ad-hoc functions can be used to build notable arrays

For an all-zero array you can use `zeros`

In [5]:
```
shape = (2, 3) # number of rows and columns
print(np.zeros(shape))
```

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

For an all-one array you can use `ones`:

In [6]:
```
print(np.ones(shape))
```

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

Building an Array

Ad-hoc functions can be used to build notable arrays

For an array filled with a single, user-chosen, values you can use `full`:

```
In [7]: shape = (2, 3) # number of rows and columns
        val = np.NaN
        print(np.full(shape, np.NaN))
```

```
[[nan nan nan]
 [nan nan nan]]
```

- `NaN` stands for Not a Number
- It's the equivalent of a missing/ill defined value

For the identity matrix, you can use `eye`:

```
In [8]: n = 3
        print(np.eye(n))
```

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

Building an Array

There are two ways to build arrays with uniformly spaced values

If you know the spacing, you can use `arange`:

In [14]:
```
x = np.arange(1, 10, step=0.5)
print(x)
```

```
[1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5 9.  9.5]
```

If you know how many numbers you need, you can use `linspace`

In [19]:
```
x = np.linspace(start=0, stop=8, num=6)
print(x)
```

```
[0.  1.6 3.2 4.8 6.4 8. ]
```

- The default value for `step` is 1
- The default value for `num` is 50

Type of an Array

All elements in an array must be of the same type

...Which can be accessed via the `dtype` attribute

```
In [20]: x = np.zeros(3)
         x.dtype

Out[20]: dtype('float64')
```

- If you try converting a collection with non-uniform types
- ... `numpy` tries to cast them to the same, most-general, type

```
In [21]: x = np.array([1, 2.3, True])
         print(x, x.dtype)

         [1.  2.3 1. ] float64
```

Array Operators

Most basic operators are redefined for arrays

In particular, the apply element wise to the involved arrays

- Some example with arithmetic operators

```
In [22]: x = np.array([1, 2, 3])
         y = np.array([4, 5, 6])
         print('x + y:', x + y)
         print('x * y:', x * y)
         print('x - y:', x - y)
         print('x / y:', x / y)
         print('y % 2:', y % 2)

x + y: [5 7 9]
x * y: [ 4 10 18]
x - y: [-3 -3 -3]
x / y: [0.25 0.4  0.5 ]
y % 2: [0 1 0]
```

Array Operators

Most basic operators are redefined for arrays

In particular, the apply element wise to the involved arrays

- Some example with comparison operators

```
In [23]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         print('x <= y:', x <= y)
         print('x == y:', x == y)

         x <= y: [ True  True False]
         x == y: [False  True False]
```

- The results is an array with a logical type

Array Operators

Most basic operators are redefined for arrays

In particular, the apply element wise to the involved arrays

- The `&`, `|`, and `~` operatore no longer apply bit-wise
- ...But element-wise (the same as all the others)

In [24]:
```python
print('~(x <= y):', ~(x <= y))
print('(x <= y) | (x >= y):', (x <= y) | (x >= y))
print('(x <= y) & (x >= y):', (x <= y) & (x >= y))
```

```
~(x <= y): [False False  True]
(x <= y) | (x >= y): [ True  True  True]
(x <= y) & (x >= y): [False  True False]
```

- Priorities are a bit tricky with these operators
- E.g. `&` and `|` has higher priority than comparison operators
- When in doubt, add brackets ;-)

Accessing Arrays

Arrays can be accesses via the indexing operator, i.e. `[]`

In particular, we use tuple indices to access single elements

```
In [26]: x = np.array([[1, 2, 3], [4, 5, 6]])
         print(x[0, 2]) # row 0, column 2

3
```

Slice indexing is also possible:

```
In [27]: print(x[0, :]) # The whole row 0
         print(x[:, 1]) # The whole column 1
         print(x[:2, :2]) # First two rows and columns

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

- Basically, you can specify one slide per dimension

Accessing Arrays

It's possible to access arrays via a collection of indices

This is most often employed with single-dimenension arrays:

```
In [28]: x = np.array([2, 4, 6, 8, 10, 12])
         idx = [0, 2, 4]
         print(x[idx]) # accesso agli indici 0, 2 e 4

[ 2  6 10]
```

- First, we build an iterable with the desired indexes
- ...Then we pass it as the argument for the indexing operator
- I.e. between the square brackets `[]`

The results is an array containing the elements at the specified indices

Accessing Arrays

Arrays can be access via a "logic mask"

- The mask is second array, having the same shape
- ...But filled with boolean values

In [29]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
mask = np.array([[True, True, False], [False, False, True]])
print(mask)
```

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

- Using such a mask an index always returns a one-dimensional arrays
- ...Containig the elements whose value is `True` in the mask

In [30]:
```
print(x[mask])
```

```
[1 2 6]
```

Accessing Arrays

Arrays can be access via a "logic mask"

It's often used to retrieve elements that satisfy a given condition:

```
In [31]: x = np.array([[1, 2, 3], [4, 5, 6]])
         x[x % 2 == 0]

Out[31]: array([2, 4, 6])
```

In this example:

- The expression `x % 2 == 0` returns a Boolean-type array
- ...Which is then used as mask to access the `x` array

The results contained all elements in `x` having an even value

Assigning Arrays

It's possible to assign individual elements within an array

...Just like it is done for lists:

In [32]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[1, 1] = -1
print(x)
```

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

Assigning Arrays

It's possible to assign entire sub-parts of an array

E.g. we can assign a whole column:

In [33]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[:, 1] = [-1, -1]
print(x)
```

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

...Or a row:

In [34]:
```
x[0, :] = [-1, -1, -1]
print(x)
```

```
[[-1 -1 -1]
 [ 4 -1  6]]
```

Assigning Arrays

It's possible to assign entire sub-parts of an array

- If the shape of the selected sub-part
- ...Is different from the shape of the assigned object
- `numpy` trys to bridge the gap by repeating the assigned object

The typical case is that of assigning a scalar to an tensor:

In [37]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[:2, :2] = -1
print(x)
```

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

- In this case, all the selected elements
- ...Are replaced with the specified scalar

Functions and Methods in `numpy`

`numpy` provides a number of functions to work with arrays

Here are a few arithmetic functions:

In [38]:
```
x = np.array([1, 2, 3, 4])
print(np.square(x)) # quadrato elemento per elemento
print(np.sqrt(x)) # radice quadrata elemento per elemento
print(np.exp(x)) # esponenziale elemento per elemento
print(np.log(x)) # logaritmo elemento per elemento
print(np.sin(x)) # seno elemento per elemento
print(np.cos(x)) # coseno elemento per elemento
```

```
[ 1  4  9 16]
[1.         1.41421356 1.73205081 2.        ]
[ 2.71828183  7.3890561  20.08553692 54.59815003]
[0.         0.69314718 1.09861229 1.38629436]
[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
[ 0.54030231 -0.41614684 -0.9899925  -0.65364362]
```

Functions and Methods in `numpy`

`numpy` provides a number of functions to work with arrays

...And here we have some aggregation functions:

In [39]:
```python
x = np.array([1, 2, 3, 4])
print(np.prod(x)) # prodotto degli elementi
print(np.sum(x)) # somma degli elementi
print(np.mean(x)) # media
print(np.std(x)) # deviazione standard
```

```
24
10
2.5
1.118033988749895
```

Functions and Methods in `numpy`

`numpy` provides a number of functions to work with arrays

Here are some functions to work with pseudo-random numbers:

```
In [40]: np.random.seed(42) # scelta del "seed"
         shape = (4,)
         print(np.random.random(shape)) # generazione di numeri casuali in [0,1)
         print(np.random.randint(low=0, high=4, size=shape)) # generazione di numeri casuali interi
         vals = [2, 4, 6, 8]
         print(np.random.choice(vals, size=shape)) # elementi casuali da una co
```

```
[0.37454012 0.95071431 0.73199394 0.59865848]
[2 1 2 2]
[6 6 8 2]
```

- Le funzioni in questa categoria sono nel modulo `np.random`
- Vi sono altri moduli utili (al solito: vedere la documentazione!)

Functions and Methods in `numpy`

Some functions are also available as methods

Benefits of `numpy`

We can use `numpy` to write more readable and efficient code

E.g. let's assume we need to sum two sequences of numbers

- First, we see a native Python solution:

In [42]:
```
%%time
n = 20000000
a = [i for i in range(n)]
b = [i for i in range(n)]
c = [v1 + v2 for v1, v2 in zip(a, b)]
```

```
CPU times: user 1.67 s, sys: 439 ms, total: 2.11 s
Wall time: 2.12 s
```

- The `%%time` directive measures and prints the run-time of a cell

Benefits of `numpy`

We can use `numpy` to write more readable and efficient code

E.g. let's assume we need to sum two sequences of numbers

- Then, we solve it using `numpy`

In [43]:
```
%%time
n = 20000000
a = np.arange(n)
b = np.arange(n)
c = a + b
```

```
CPU times: user 309 ms, sys: 154 ms, total: 462 ms
Wall time: 462 ms
```

- The `numpy` version us more readable
- ...And also much faster!