Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2023/24, I semester

Lecture XVI: NumPy arrays

NumPy

NumPy is a third-party library very popular for scientific/numerical programming (https://numpy.org/).

- Features familiar to matlab, R, Julia programmers
- The key data structure is the array
  - 1-dimension arrays: vectors
  - 2-dimension arrays: matrices
  - n-dimension arrays

In some languages array is more or less synonym of list: Python distinguishes: lists (mutable, arbitrary elements), arrays (mutable, all elements have the same type), tuples (immutable, fixed length, arbitrary elements).

NumPy arrays

The most important data structure in `NumPy` is `ndarray`: a (usually fixed-size) sequence of same type elements, organized in one or more dimensions.

https://numpy.org/doc/stable/reference/arrays.ndarray.html

Implementation is based on byte arrays: accessing an element (all of the same byte-size) is virtually just the computation of an 'address'.

Why?

- using NumPy arrays is often more compact, especially when there's more than one dimension
- faster than lists when the operation can be vectorized
- (slower than lists when you append elements to the end)
- can be used with element of different types but this is less efficient

A `ndarray` has a `dtype` (the type of elements) and a `shape` (the length of the array on each dimensional axis). (Note the jargon: slightly different from linear algebra)

- Since appending is costly, normally they are pre-allocated (`zeros`, `ones`, `arange`, `linspace`, ...)
- vectorized operations can simplify code (no need for loops) and they are faster with big arrays
- vector indexing syntax (similar to R): very convenient (but you need to learn something new)

All the elements must have the same size

This is actually a big limitation: the faster access comes with a price in flexibility.

```
>>> np.array(['','',''])
array(['', '', ''], dtype='<U1')
>>> np.array(['a','bb','ccc'])
array(['a', 'bb', 'ccc'], dtype='<U3')
>>> np.array(['a','bb','cccxxxxxxxxxxxxxxxxxx'])
array(['a', 'bb', 'cccxxxxxxxxxxxxxxxxxx'], dtype='<U21')
```

Usually the length is not changed

The best use of arrays is to avoid a change in their length, that can be costly. Thus, they are normally preallocated at creation:

- `np.array([1,2,3])`
- `np.zeros(2)`, `np.zeros(2, float)`, `np.ones(2)`
- `np.empty((2,3))` six not meaningful float values
- `np.arange(1, 5)` be careful with floats:

  ```
  >>> np.arange(0.4, 0.8, 0.1)
  array([0.4, 0.5, 0.6, 0.7])
  >>> np.arange(0.5, 0.8, 0.1)
  array([0.5, 0.6, 0.7, 0.8])
  ```

- `np.linspace(0.5, 0.8, 3)` with this the length is easier to predict

You can concatenate arrays with `np.concatenate` (be careful with the shapes!)