Numpy

**Numerical** Python, It is an open source extension module for Python, which provides fast precompiled functions for mathematical and numerical routines. Python with powerful data structures for efficient computation of multi-dimensional arrays and matrices. The implementation is even aiming at huge matrices and arrays. Besides that the module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.

NumPy is a Python extension module that provides efficient operation on arrays of homogeneous data. It allows python to serve as a high-level language for manipulating numerical data, much like IDL, MATLAB, or Yorick

**SciPy** (Scientific Python) is often mentioned in the same breath with NumPy. SciPy extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.

**The Python Alternative to Matlab**

Python in combination with **Numpy**, **Scipy** and **Matplotlib** can be used as a replacement for MATLAB. The combination of NumPy, SciPy and Matplotlib is a free alternative to MATLAB. Even though MATLAB has a huge number of additional toolboxes available, NumPy has the advantage that Python is a more modern and complete programming language and - as we have said already before - is open source. SciPy adds even more MATLAB-like functionalities to Python. Python is rounded out in the direction of MATLAB with the module Matplotlib, which provides MATLAB-like plotting functionality.

**Advantages of Numpy:**

1. NumPy is not just more efficient; it is also more convenient. You get a lot of vector and matrix operations for free, which sometimes allow one to avoid unnecessary work. And they are also efficiently implemented.
2. **Speed**: adding two lists in python will take more time that adding using numpy array.
3. **Memory**
4. The main benefits of using numpy arrays should be smaller memory consumption and better runtime behavior
5. Extension package to Python for **multi-dimensional arrays**
6. Closer to hardware (**efficiency**)
7. Designed for **scientific computation** (convenience)

* **Convert a python list to one dimensional numpy array :**

Cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]

**C = np.array (**Cvalues**)**

**print(C \* 9 / 5 + 32)**

**[68.18 69.44 71.42 72.5 72.86 72.14 71.24 70.16 69.62 68.18]**

* **Graphical representation:**

import matplotlib.pyplot as plt

plt.plot(C)

plt.show ()

Numpy Arrays:

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension

**1-D:**

a = np.array([0, 1, 2, 3])

**>>>** a

array([0, 1, 2, 3])

**>>>** a.ndim

1

**>>>** a.shape

(4,)

**>>>** len(a)

4

2-D,3-D:

b = np.array([[0, 1, 2], [3, 4, 5]]) *# 2 x 3 array*

**>>>** b

array([[0, 1, 2],

[3, 4, 5]])

**>>>** b.ndim

2

**>>>** b.shape

(2, 3)

**>>>** len(b) *# returns the size of the first dimension*

2

**>>>** c = np.array([[[1], [2]], [[3], [4]]])

**>>>** c

array([[[1],

[2]],

[[3],

[4]]])

**>>>** c.shape

(2, 2, 1)

Functions for creating arrays

* **Evenly spaced:**

>>>

**>>>** a = np.arange(10) *# 0 .. n-1 (!)*

**>>>** a

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

**>>>** b = np.arange(1, 9, 2) *# start, end (exclusive), step*

**>>>** b

array([1, 3, 5, 7])

* **By number of points:**

>>>

**>>>** c = np.linspace(0, 1, 6) *# start, end, num-points*

**>>>** c

array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. ])

**>>>** d = np.linspace(0, 1, 5, endpoint=False)

**>>>** d

array([ 0. , 0.2, 0.4, 0.6, 0.8])

* **Common arrays:**

>>>

**>>>** a = np.ones((3, 3)) *# reminder: (3, 3) is a tuple*

**>>>** a

array([[ 1., 1., 1.],

[ 1., 1., 1.],

[ 1., 1., 1.]])

**>>>** b = np.zeros((2, 2))

**>>>** b

array([[ 0., 0.],

[ 0., 0.]])

**>>>** c = np.eye(3)

**>>>** c

array([[ 1., 0., 0.],

[ 0., 1., 0.],

[ 0., 0., 1.]])

**>>>** d = np.diag(np.array([1, 2, 3, 4]))

**>>>** d

array([[1, 0, 0, 0],

[0, 2, 0, 0],

[0, 0, 3, 0],

[0, 0, 0, 4]])

* **np.random**: random numbers

**>>>** a = np.random.rand(4) *# uniform in [0, 1]*

**>>>** a

array([ 0.95799151, 0.14222247, 0.08777354, 0.51887998])

**>>>** b = np.random.randn(4) *# Gaussian*

**>>>** b

array([ 0.37544699, -0.11425369, -0.47616538, 1.79664113])

**>>>** np.random.seed(1234) *# Setting the random seed*

[Basic data types](http://www.scipy-lectures.org/intro/numpy/array_object.html#id3)

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. 2. vs 2). This is due to a difference in the data-type used:

>>>

**>>>** a = np.array([1, 2, 3])

**>>>** a.dtype

dtype('int64')

**>>>** b = np.array([1., 2., 3.])

**>>>** b.dtype

dtype('float64')

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

>>>

**>>>** c = np.array([1, 2, 3], dtype=float)

**>>>** c.dtype

dtype('float64')

The **default** data type is floating point:

>>>

**>>>** a = np.ones((3, 3))

**>>>** a.dtype

dtype('float64')

**There are also other types:**

|  |  |
| --- | --- |
| **Complex:** | >>>  **>>>** d = d = np.array([1+2j, 3+4j, 5+6\*1j])  **>>>** d.dype  Dty dtype(‘complex128') |
| **Bool:** | >>>  **>>>** e = np.array([True, False, False, True])   * **>>** e.dtype   dtype('bool') |
| **Strings:** | >>>  **>>>** f = np.array(['Bonjour', 'Hello', 'Hallo',])  **>>>** f.dtype *# <--- strings containing max. 7 letters*  dtype('S7') |
| **Much more:** | * int32 * int64 * uint32 * uint64 |

[Basic visualization](http://www.scipy-lectures.org/intro/numpy/array_object.html#id4)

>>> %matplotlib inline

The inline is important for the notebook, so that plots are displayed in the notebook and not in a new window

***Matplotlib*** is a 2D plotting package. We can import its functions as below:

**>>> import** **matplotlib.pyplot** **as** **plt** *# the tidy way*

>>> plt.plot(x, y) *# line plot*

**>>>** plt.show() *# <-- shows the plot (not needed with interactive plots)*

**1D plotting**:

**>>>** x = np.linspace(0, 3, 20)

**>>>** y = np.linspace(0, 9, 20)

**>>>** plt.plot(x, y) *# line plot*

[<matplotlib.lines.Line2D object at ...>]

**>>>** plt.plot(x, y, 'o') *# dot plot*

[<matplotlib.lines.Line2D object at ...>]

[Indexing and slicing](http://www.scipy-lectures.org/intro/numpy/array_object.html#id5)

>>> a = np.arange(10)

**>>>** a

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

**>>>** a[0], a[2], a[-1]

(0, 2, 9)

**The usual python idiom for reversing a sequence is supported:**

>>> a[::-1]

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

**For multidimensional arrays, indexes are tuples of integers:**

**>>>** a = np.diag(np.arange(3))

**>>>** a

array([[0, 0, 0],

[0, 1, 0],

[0, 0, 2]])

**>>>** a[1, 1]

1

**>>>** a[2, 1] = 10 *# third line, second column*

**>>>** a

array([[ 0, 0, 0],

[ 0, 1, 0],

[ 0, 10, 2]])

**>>>** a[1]

array([0, 1, 0])

**Slicing: Arrays, like other Python sequences can also be sliced:**

**>>>** a = np.arange(10)

**>>>** a

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

**>>>** a[2:9:3] *# [start:end:step]*

array([2, 5, 8])

**Note that the last index is not included! :**

**>>>** a[:4]

array([0, 1, 2, 3])

**All three slice components are not required: by default, *start* is 0, *end* is the last and *step* is 1:**

**>>>** a[1:3]

array([1, 2])

**>>>** a[::2]

array([0, 2, 4, 6, 8])

**>>>** a[3:]

array([3, 4, 5, 6, 7, 8, 9])

[Copies and views](http://www.scipy-lectures.org/intro/numpy/array_object.html#id6)

You can use np.may\_share\_memory()to check if two arrays share the same memory block.

**When modifying the view, the original array is modified as well**:

>>>

**>>>** a = np.arange(10)

**>>>** a

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

**>>>** b = a[::2]

**>>>** b

array([0, 2, 4, 6, 8])

**>>>** np.may\_share\_memory(a, b)

True

**>>>** b[0] = 12

**>>>** b

array([12, 2, 4, 6, 8])

**>>>** a *# (!)*

array([12, 1, 2, 3, 4, 5, 6, 7, 8, 9])

**>>>** a = np.arange(10)

**>>>** c = a[::2].copy() *# force a copy*

**>>>** c[0] = 12

**>>>** a

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

**>>>** np.may\_share\_memory(a, c)

False

* **Indexing with Mask:**

Indexing with a mask can be very useful to assign a new value to a sub-array:

np.random.seed(3)

**>>>** a = np.random.randint(0, 21, 15)

a[a % 3 == 0] = -1

**>>>** a

array([10, -1, 8, -1, 19, 10, 11, -1, 10, -1, -1, 20, -1, 7, 14])

* **Indexing with an array of integers :**

**>>>** a = np.arange(0, 100, 10)

**>>>** a

array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

**Indexing can be done with an array of integers, where the same index is repeated several time:**

**>>>** a[[2, 3, 2, 4, 2]] *# note: [2, 3, 2, 4, 2] is a Python list*

array([20, 30, 20, 40, 20])

References:

<http://www.scipy-lectures.org/intro/numpy/array_object.html#what-are-numpy-and-numpy-arrays>