Tutorial Brief

numpy is a powerful set of tools to perform mathematical operations of on lists of numbers. It works faster than normal python lists operations and can manupilate high dimentional arrays too.

Finding Help:

- http://wiki.scipy.org/Tentative_NumPy_Tutorial
- http://docs.scipy.org/doc/numpy/reference/

> SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering.
>
> *http://www.scipy.org/*

So NumPy is a part of a bigger ecosystem of libraries that build on the optimized performance of NumPy NDArray.

It contain these core packages:

	NumPy Base N-dimensional array package		**SciPy** Fundamental library for scientific computing		**Matplotlib** Comprehensive 2D Plotting
IP[y]: IPython	**IPython** Enhanced Interactive Console		**SymPy** Symbolic mathematics		**Pandas** Data structures & analysis

Importig the library

Import numpy library as np

This helps in writing code and it's almost a standard in scientific work

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

Working with ndarray

We will generate an ndarray with np.arange method.

np.arange([start,] stop[, step,], dtype=None)

In [2]:
```
np.arange(10)
```
Out[2]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [3]:
```
np.arange(1,10)
```
Out[3]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [4]:
```
np.arange(1,10, 0.5)
```
Out[4]:
```
array([ 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])
```

In [5]:
```
np.arange(1,10, 3)
```
Out[5]:
```
array([1, 4, 7])
```

In [6]:
```
np.arange(1,10, 2, dtype=np.float64)
```
Out[6]:
```
array([ 1.,  3.,  5.,  7.,  9.])
```

Examining ndrray

In [7]:
```
ds = np.arange(1,10,2)
ds.ndim
```
Out[7]:
```
1
```

In [8]:
```
ds.shape
```
Out[8]:
```
(5,)
```

In [9]:
```
ds.size
```
Out[9]:
```
5
```

In [10]:
```
ds.dtype
```
Out[10]:
```
dtype('int64')
```

In [11]:
```
ds.itemsize
```
Out[11]:
```
8
```

In [12]:
```
x=ds.data
list(x)
```
Out[12]:
```
['\x01',
```

```
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x03',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x05',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x07',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\t',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00',
 '\x00']
```

In [13]:
```
ds
```

Out[13]:
```
array([1, 3, 5, 7, 9])
```

In [14]:
```
# Memory Usage
ds.size * ds.itemsize
```

Out[14]:
```
40
```

Why to use numpy?

We will compare the time it takes to create two lists and do some basic operations on them.

Generate a list

In [15]:
```
%%capture timeit_results
# Regular Python
%timeit python_list_1 = range(1,1000)
python_list_1 = range(1,1000)
python_list_2 = range(1,1000)

#Numpy
%timeit numpy_list_1 = np.arange(1,1000)
numpy_list_1 = np.arange(1,1000)
numpy_list_2 = np.arange(1,1000)
```

In [16]:
```
print timeit_results
```

```
100000 loops, best of 3: 13.9 us per loop
100000 loops, best of 3: 2.14 us per loop
```

In [17]:
```
# Function to calculate time in seconds
def return_time(timeit_result):
    temp_time = float(timeit_result.split(" ")[5])
    temp_unit = timeit_result.split(" ")[6]
    if temp_unit == "ms":
        temp_time = temp_time * 1e-3
    elif temp_unit == "us":
        temp_time = temp_time * 1e-6
    elif temp_unit == "ns":
        temp_time = temp_time * 1e-9
    return temp_time
```

In [18]:
```
python_time = return_time(timeit_results.stdout.split("\n")[0])
numpy_time = return_time(timeit_results.stdout.split("\n")[1])

print "Python/NumPy: %.1f" % (python_time/numpy_time)
```

```
Python/NumPy: 6.5
```

Basic Operation

In [19]:
```
%%capture timeit_python
%%timeit
# Regular Python
[(x + y) for x, y in zip(python_list_1, python_list_2)]
[(x - y) for x, y in zip(python_list_1, python_list_2)]
[(x * y) for x, y in zip(python_list_1, python_list_2)]
[(x / y) for x, y in zip(python_list_1, python_list_2)];
```

In [20]:
```
print timeit_python
```

```
1000 loops, best of 3: 626 us per loop
```

In [21]:
```
%%capture timeit_numpy
%%timeit
#Numpy
numpy_list_1 + numpy_list_2
numpy_list_1 - numpy_list_2
numpy_list_1 * numpy_list_2
numpy_list_1 / numpy_list_2;
```

In [22]:
```
print timeit_numpy
```

```
10000 loops, best of 3: 34.6 us per loop
```

In [23]:
```
python_time = return_time(timeit_python.stdout)
numpy_time = return_time(timeit_numpy.stdout)

print "Python/NumPy: %.1f" % (python_time/numpy_time)
```

```
Python/NumPy: 18.1
```

Most Common Functions

List Creation

array(object, dtype=None, copy=True, order=None, subok=False, ndmin=0)

```
Parameters
----------
object : array_like
    An array, any object exposing the array interface, an
    object whose __array__ method returns an array, or any
    (nested) sequence.
dtype : data-type, optional
    The desired data-type for the array.  If not given, then
    the type will be determined as the minimum type required
    to hold the objects in the sequence.  This argument can only
    be used to 'upcast' the array.  For downcasting, use the
    .astype(t) method.
copy : bool, optional
    If true (default), then the object is copied.  Otherwise, a copy
    will only be made if __array__ returns a copy, if obj is a
    nested sequence, or if a copy is needed to satisfy any of the other
    requirements (`dtype`, `order`, etc.).
order : {'C', 'F', 'A'}, optional
    Specify the order of the array.  If order is 'C' (default), then the
    array will be in C-contiguous order (last-index varies the
    fastest).  If order is 'F', then the returned array
    will be in Fortran-contiguous order (first-index varies the
    fastest).  If order is 'A', then the returned array may
    be in any order (either C-, Fortran-contiguous, or even
    discontiguous).
subok : bool, optional
    If True, then sub-classes will be passed-through, otherwise
    the returned array will be forced to be a base-class array (default).
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be pre-pended to the shape as
    needed to meet this requirement.
```

In [24]:
```
np.array([1,2,3,4,5])
```

Out[24]:
```
array([1, 2, 3, 4, 5])
```

Multi Dimentional Array

In [25]:
```
np.array([[1,2],[3,4],[5,6]])
```

Out[25]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

zeros(shape, dtype=float, order='C')

```
Parameters
----------
shape : int or sequence of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
dtype : data-type, optional
    The desired data-type for the array, e.g., `numpy.int8`.  Default is
    `numpy.float64`.
order : {'C', 'F'}, optional
    Whether to store multidimensional data in C- or Fortran-contiguous
    (row- or column-wise) order in memory.
```

In [26]:
```
np.zeros((3,4))
```

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

In [27]:
```
np.zeros((3,4), dtype=np.int64)
```

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

np.linspace(start, stop, num=50, endpoint=True, retstep=False)

```
Parameters
----------
start : scalar
    The starting value of the sequence.
stop : scalar
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded.  Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50.
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
```

In [28]:
```
np.linspace(1,5)
```

Out[28]:
```
array([ 1.        ,  1.08163265,  1.16326531,  1.24489796,  1.32653061,
        1.40816327,  1.48979592,  1.57142857,  1.65306122,  1.73469388,
        1.81632653,  1.89795918,  1.97959184,  2.06122449,  2.14285714,
        2.2244898 ,  2.30612245,  2.3877551 ,  2.46938776,  2.55102041,
        2.63265306,  2.71428571,  2.79591837,  2.87755102,  2.95918367,
        3.04081633,  3.12244898,  3.20408163,  3.28571429,  3.36734694,
        3.44897959,  3.53061224,  3.6122449 ,  3.69387755,  3.7755102 ,
        3.85714286,  3.93877551,  4.02040816,  4.10204082,  4.18367347,
        4.26530612,  4.34693878,  4.42857143,  4.51020408,  4.59183673,
        4.67346939,  4.75510204,  4.83673469,  4.91836735,  5.        ])
```

In [29]:
```
np.linspace(0,2,num=4)
```

Out[29]:
```
array([ 0.        ,  0.66666667,  1.33333333,  2.        ])
```

In [30]:
```
np.linspace(0,2,num=4,endpoint=False)
```

Out[30]:
```
array([ 0. ,  0.5,  1. ,  1.5])
```

random_sample(size=None)

```
Parameters
----------
size : int or tuple of ints, optional
    Defines the shape of the returned array of random floats. If None
    (the default), returns a single float.
```

In [31]:
```
np.random.random((2,3))
```

Out[31]:
```
array([[ 0.60383905,  0.84632409,  0.18122863],
       [ 0.73495109,  0.36127266,  0.27401845]])
```

In [32]:
```
np.random.random_sample((2,3))
```

Out[32]:
```
array([[ 0.57143353,  0.23111517,  0.90913961],
       [ 0.03532667,  0.44399022,  0.15470195]])
```

Statistical Analysis

In [33]:
```
data_set = np.random.random((2,3))
data_set
```

Out[33]:
```
array([[ 0.99101817,  0.91362334,  0.37546237],
       [ 0.66962595,  0.27485988,  0.45081456]])
```

np.max(a, axis=None, out=None, keepdims=False)

```
Parameters
----------
a : array_like
    Input data.
axis : int, optional
    Axis along which to operate.  By default, flattened input is used.
out : ndarray, optional
    Alternative output array in which to place the result.  Must
    be of the same shape and buffer length as the expected output.
    See `doc.ufuncs` (Section "Output arguments") for more details.
keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the original `arr`.
```

In [34]:
```
np.max(data_set)
```

Out[34]: 0.99101817417900118

In [35]:
```
np.max(data_set, axis=0)
```
Out[35]:
```
array([ 0.99101817,  0.91362334,  0.45081456])
```

In [36]:
```
np.max(data_set, axis=1)
```
Out[36]:
```
array([ 0.99101817,  0.66962595])
```

np.min(a, axis=None, out=None, keepdims=False)

In [37]:
```
np.min(data_set)
```
Out[37]: 0.2748598782158802

np.mean(a, axis=None, dtype=None, out=None, keepdims=False)

In [38]:
```
np.mean(data_set)
```
Out[38]: 0.6125673792901426

np.median(a, axis=None, out=None, overwrite_input=False)

In [39]:
```
np.median(data_set)
```
Out[39]: 0.56022025506863438

np.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)

In [40]:
```
np.std(data_set)
```
Out[40]: 0.2688073942829961

np.sum(a, axis=None, dtype=None, out=None, keepdims=False)

In [41]:
```
np.sum(data_set)
```
Out[41]: 3.6754042757408558

Reshaping

np.reshape(a, newshape, order='C')

In [42]:
```
np.reshape(data_set, (3,2))
```
Out[42]:
```
array([[ 0.99101817,  0.91362334],
       [ 0.37546237,  0.66962595],
       [ 0.27485988,  0.45081456]])
```

In [43]:
```
np.reshape(data_set, (6,1))
```
Out[43]:
```
array([[ 0.99101817],
       [ 0.91362334],
       [ 0.37546237],
       [ 0.66962595],
       [ 0.27485988],
       [ 0.45081456]])
```

In [44]:
```
np.reshape(data_set, (6))
```
Out[44]:
```
array([ 0.99101817,  0.91362334,  0.37546237,  0.66962595,  0.27485988,
        0.45081456])
```

np.ravel(a, order='C')

In [45]:
```
np.ravel(data_set)
```
Out[45]:
```
array([ 0.99101817,  0.91362334,  0.37546237,  0.66962595,  0.27485988,
        0.45081456])
```

Slicing

In [46]:
```
data_set = np.random.random((5,10))
data_set
```
Out[46]:
```
array([[ 0.2187482 ,  0.87071416,  0.73663416,  0.27910705,  0.78239476,
         0.53835918,  0.51234398,  0.72563682,  0.7497531 ,  0.61090375],
       [ 0.46166143,  0.84292073,  0.19234863,  0.31204936,  0.64249925,
         0.23149184,  0.45047676,  0.79576087,  0.84369549,  0.09006852],
       [ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.74813493,  0.5738713 ,  0.40863753,  0.44157988,  0.32909602,
         0.51802248,  0.33975736,  0.36404317,  0.70869127,  0.50686958],
       [ 0.48861471,  0.16930154,  0.03239842,  0.0835669 ,  0.44708358,
         0.8001063 ,  0.39644714,  0.83747988,  0.71102625,  0.44535013]])
```

In [47]:
```
data_set[1]
```
Out[47]:
```
array([ 0.46166143,  0.84292073,  0.19234863,  0.31204936,  0.64249925,
        0.23149184,  0.45047676,  0.79576087,  0.84369549,  0.09006852])
```

In [48]:
```
data_set[1][0]
```
Out[48]: 0.46166143118294922

In [49]:
```
data_set[1,0]
```

Out[49]: 0.46166143118294922

Slicing a range

In [50]:
```
data_set[2:4]
```

Out[50]:
```
array([[ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.74813493,  0.5738713 ,  0.40863753,  0.44157988,  0.32909602,
         0.51802248,  0.33975736,  0.36404317,  0.70869127,  0.50686958]])
```

In [51]:
```
data_set[2:4,0]
```

Out[51]:
```
array([ 0.74299397,  0.74813493])
```

In [52]:
```
data_set[2:4,0:2]
```

Out[52]:
```
array([[ 0.74299397,  0.91711184],
       [ 0.74813493,  0.5738713 ]])
```

In [53]:
```
data_set[:,0]
```

Out[53]:
```
array([ 0.2187482 ,  0.46166143,  0.74299397,  0.74813493,  0.48861471])
```

Stepping

In [54]:
```
data_set[2:4:1]
```

Out[54]:
```
array([[ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.74813493,  0.5738713 ,  0.40863753,  0.44157988,  0.32909602,
         0.51802248,  0.33975736,  0.36404317,  0.70869127,  0.50686958]])
```



```
       [ 0.46166143,  0.84292073,  0.19234863,  0.31204936,  0.64249925,
         0.23149184,  0.45047676,  0.79576087,  0.84369549,  0.09006852],
       [ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.74813493,  0.5738713 ,  0.40863753,  0.44157988,  0.32909602,
         0.51802248,  0.33975736,  0.36404317,  0.70869127,  0.50686958],
       [ 0.48861471,  0.16930154,  0.03239842,  0.0835669 ,  0.44708358,
         0.8001063 ,  0.39644714,  0.83747988,  0.71102625,  0.44535013]])
```

In [56]:
```
data_set[::2]
```

Out[56]:
```
array([[ 0.2187482 ,  0.87071416,  0.73663416,  0.27910705,  0.78239476,
         0.53835918,  0.51234398,  0.72563682,  0.7497531 ,  0.61090375],
       [ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.48861471,  0.16930154,  0.03239842,  0.0835669 ,  0.44708358,
         0.8001063 ,  0.39644714,  0.83747988,  0.71102625,  0.44535013]])
```

In [57]:
```
data_set[2:4]
```

Out[57]:
```
array([[ 0.74299397,  0.91711184,  0.76535827,  0.16743916,  0.33435712,
         0.50974527,  0.82367946,  0.03806086,  0.70315627,  0.58959405],
       [ 0.74813493,  0.5738713 ,  0.40863753,  0.44157988,  0.32909602,
         0.51802248,  0.33975736,  0.36404317,  0.70869127,  0.50686958]])
```

In [58]:
```
data_set[2:4,::2]
```

Out[58]:
```
array([[ 0.74299397,  0.76535827,  0.33435712,  0.82367946,  0.70315627],
       [ 0.74813493,  0.40863753,  0.32909602,  0.33975736,  0.70869127]])
```