Python for Data Analysis

Dr Malu G

Python for Data Analysis

- Overview of Python Libraries for Data Scientists
- Reading Data; Selecting and Filtering the Data; Data manipulation, sorting, grouping, rearranging
- Plotting the data
- Descriptive statistics
- Inferential statistics

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy

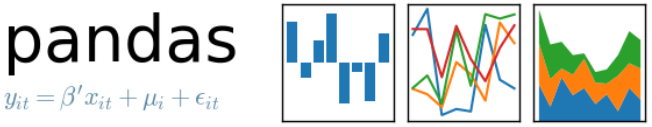
Link: http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link: https://www.scipy.org/scipylib/



Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.
- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Science

Seaborn:

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

Python – NumPy

Scientific Python?

- Extra features required:
  - fast, multidimensional arrays
  - libraries of reliable, tested scientific functions
  - plotting tools
- NumPy is at the core of nearly every scientific Python application or module since it provides a fast N-d array datatype that can be manipulated in a vectorized form.

NumPy documentation

- Official documentation
  - http://docs.scipy.org/doc/
- The NumPy book
  - http://web.mit.edu/dvp/Public/numpybook.pdf
- Example list
  - https://docs.scipy.org/doc/numpy/reference/routines.html

Arrays – Numerical Python (Numpy)

- Lists ok for storing small amounts of one-dimensional data

```
>>> a = [1,3,5,7,9]
>>> print(a[2:4])
[5, 7]
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

```
>>> a = [1,3,5,7,9]
>>> b = [3,5,6,7,9]
>>> c = a + b
>>> print c
[1, 3, 5, 7, 9, 3, 5, 6, 7, 9]
```

- But, can't use directly with arithmetical operators (+, -, *, /, …)
- Need efficient arrays with arithmetic and better multidimensional tools
- **Numpy**

```
>>> import numpy
```

- Similar to lists, but much more capable, except fixed size

Numpy – N-dimensional Array manpulations

The fundamental library needed for scientific computing with Python is called NumPy.

This Open Source library contains:

- a powerful N-dimensional array object
- advanced array slicing methods (to select array elements)
- convenient array reshaping methods

and it even contains 3 libraries with numerical routines:

- basic linear algebra functions
- basic Fourier transforms
- sophisticated random number capabilities

NumPy can be extended with C-code for functions where performance is highly time critical. In addition, tools are provided for integrating existing Fortran code.

NumPy is a hybrid of the older NumArray and Numeric packages, and is meant to replace them both.

Numpy – Creating arrays

- There are a number of ways to initialize new numpy arrays, for example from
  - a Python list or tuples
  - using functions that are dedicated to generating numpy arrays, such as arange, linspace, etc.
  - reading data from files

Numpy – Creating arrays

NumPy Array

- NumPy arrays are used to store lists of numerical data, vectors and matrices.
- The Numpy array is officially called ndarray but commonly known as array

Creation of NumPy Arrays from List

- To create an array and to use its methods, first we need to import the NumPy library
- >>> import numpy as np
- The NumPy's array() function converts a given list into an array.
- For example, Create an array called array1 from the given list.
- >>> array1 = np.array([10,20,30])
- #Display the contents of the array
- >>> array1
- array([10, 20, 30])

Creating a 1-D Array

An array with only single row of elements is called 1-D array.

```
>>> import numpy as np
>>> array1 = np.array([10,20,30])
>>> array1
array([10, 20, 30])
>>> array1[0]
10
>>> array1[1]
20
>>> array2=np.array([1,2,3,4])
>>> array2
array([1, 2, 3, 4])
>>> array3=np.array([5,-7.2,'s',8.3])
>>> array3
array(['5', '-7.2', 's', '8.3'], dtype='<U32')
>>> array4=np.array([5,-7.2,8.3])
>>> array4
array([ 5. , -7.2,  8.3])
```

Creating a 2-D Array

We can create a two dimensional (2-D) arrays by passing nested lists to the array() function.

```
>>> array6=np.array([[1,2],[3,4],[5,6]])
>>> array6
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
>>> array3 = np.array([[2.4,3],
                       [4.91,7],[0,-1]])
>>> array3
array([[ 2.4 ,  3.  ],
       [ 4.91,  7.  ],
       [ 0.  , -1.  ]])
```

Attributes of NumPy Array

i) ndarray.ndim: gives the number of dimensions of the array as an integer value.

Arrays can be 1-D, 2-D or n-D. NumPy calls the dimensions as axes (plural of axis). Thus, a 2-D array has two axes. The row-axis is called axis-0 and the column-axis is called axis-1. The number of axes is also called the array's rank.

```
>>> array1.ndim
1
>>> array6.ndim
2
```

ii) ndarray.shape: It gives the sequence of integers indicating the size of the array for each dimension.

```
>>> array1.shape
(3,)
>>> array6.shape
(3, 2)
>>> array2.shape
(4,)
>>> array3.shape
(4,)
```

Attributes of NumPy Array

iii) ndarray.size: It gives the total number of elements of the array. This is equal to the product of the elements of shape.

```
>>> array6.size
6
>>> array1.size
3
```

iv) ndarray.dtype: is the data type of the elements of the array. All the elements of an array are of same data type. Common data types are int32, int64, float32, float64, U32, etc

```
>>> array1.dtype
dtype('int32')
```

v) ndarray.itemsize: It specifies the size in bytes of each element of the array.

```
>>> array1.itemsize
4
```

Other Ways of Creating NumPy Arrays

1. We can specify data type (integer, float, etc.) while creating array using dtype as an argument to array(). This will convert the data automatically to the mentioned type.

```
>>> array4 = np.array( [ [1,2], [3,4] ],
             dtype=float)
>>> array4
array([[1., 2.],
       [3., 4.]])
```

2. We can create an array with all elements initialised to 0 using the function zeros(). By default, the data type of the array created by zeros() is float.

```
>>> array5 = np.zeros((3,4))
>>> array5
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

3. We can create an array with all elements initialised to 1 using the function ones(). By default, the data type of the array created by ones() is float.

```
>>> array6 = np.ones((3,2))
>>> array6
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

Other Ways of Creating NumPy Arrays

4. We can create an array with numbers in a given range and sequence using the arange() function.

```
>>> array7 = np.arange(6)
# an array of 6 elements is created with
start value 5 and step size 1
>>> array7
array([0, 1, 2, 3, 4, 5])
# Creating an array with start value -2, end
# value 24 and step size 4
>>> array8 = np.arange( -2, 24, 4 )
>>> array8
array([-2,  2,  6, 10, 14, 18, 22])
```

Indexing

- For 2-D arrays indexing for both dimensions starts from 0, and each element is referenced through two indexes i and j, where i represents the row number and j represents the column number.

Name	Maths	English	Science
Ramesh	78	67	56
Vedika	76	75	47
Harun	84	59	60
Prasad	67	72	54

- Consider Table showing marks obtained by students in three different subjects.
- Let us create an array called marks to store marks given in three subjects for four students given in this table. As there are 4 students (i.e. 4 rows) and 3 subjects (i.e. 3 columns), the array will be called marks[4][3]. This array can store 4*3 = 12 elements.
- Here, marks[i,j] refers to the element at (i+1)th row and (j+1)th column because the index values start at 0. Thus marks[3,1] is the element in 4th row and second column which is 72

```
# accesses the element in the 1st row in
# the 3rd column
>>> marks[0,2]
56
>>> marks [0,4]
index Out of Bound "Index Error". Index 4
is out of bounds for axis with size 3
```

Slicing

- Sometimes we need to extract part of an array. This is done through slicing.
- We can define which part of the array to be sliced by specifying the start and end index values using [start : end] along with the array name.

```
>>> array8
array([-2,  2,  6, 10, 14, 18, 22])

# excludes the value at the end index
>>> array8[3:5]
array([10, 14])

# reverse the array
>>> array8[ : : -1]
array([22, 18, 14, 10,  6,  2, -2])
```

Slicing for 2-D arrays

Consider a 2-D array called array9 having 3 rows and 4 columns.

```
>>> array9 = np.array([[ -7,  0,  10,  20],
                       [ -5,  1,  40, 200],
                       [ -1,  1,   4,  30]])

# access all the elements in the 3rd column
>>> array9[0:3,2]
array([10, 40,  4])
```

Note that we are specifying rows in the range 0:3 because the end value of the range is excluded

```
# access elements of 2nd and 3rd row from 1st
# and 2nd column
>>> array9[1:3,0:2]
array([[-5,  1],
       [-1,  1]])
```

If row indices are not specified, it means all the rows are to be considered. Likewise, if column indices are not specified, all the columns are to be considered. Thus, the statement to access all the elements in the 3rd column can also be written as:

```
>>>array9[:,2]
array([10, 40, 4])
```

Operations on Arrays

Arithmetic Operations

```
>>> array1 = np.array([[3,6],[4,2]])
>>> array2 = np.array([[10,20],[15,12]])
```

```
#Element-wise addition of two matrices.
>>> array1 + array2
array([[13, 26],
       [19, 14]])

#Subtraction
>>> array1 - array2
array([[ -7, -14],
       [-11, -10]])

#Multiplication
>>> array1 * array2
array([[ 30, 120],
       [ 60,  24]])

#Matrix Multiplication
>>> array1 @ array2
array([[120, 132],
       [ 70, 104]])
```

```
#Exponentiation
>>> array1 ** 3
array([[ 27, 216],
       [ 64,   8]], dtype=int32)

#Division
>>> array2 / array1
array([[3.33333333, 3.33333333],
       [3.75      , 6.        ]])

#Element wise Remainder of Division
#(Modulo)
>>> array2 % array1
array([[1, 2],
       [3, 0]], dtype=int32)
```


It is important to note that for element-wise operations, size of both arrays must be same. That is, array1.shape must be equal to array2.shape.

Operations on Arrays

Transpose

Transposing an array turns its rows into columns and columns into rows just like matrices in mathematics.

```
#Transpose
>>> array3 = np.array([[10,-7,0, 20],
                       [-5,1,200,40],[30,1,-1,4]])
>>> array3
array([[ 10,  -7,   0,  20],
       [ -5,   1, 200,  40],
       [ 30,   1,  -1,   4]])

# the original array does not change
>>> array3.transpose()
array([[ 10,  -5,  30],
       [ -7,   1,   1],
       [  0, 200,  -1],
       [ 20,  40,   4]])
```

Operations on Arrays

Sorting

- Sorting is to arrange the elements of an array in hierarchical order either ascending or descending.
- By default, numpy does sorting in ascending order.

```
>>> array4 = np.array([1,0,2,-3,6,8,4,7])
>>> array4.sort()
>>> array4
array([-3,  0,  1,  2,  4,  6,  7,  8])
```

- In 2-D array, sorting can be done along either of the axes i.e., row-wise or column-wise.
- By default, sorting is done row-wise (i.e., on axis = 1). It means to arrange elements in each row in ascending order.
- When axis=0, sorting is done column-wise, which means each column is sorted in ascending order.

```
>>> array4 = np.array([[10,-7,0, 20],
                       [-5,1,200,40],[30,1,-1,4]])
>>> array4
array([[ 10,  -7,   0,  20],
       [ -5,   1, 200,  40],
       [ 30,   1,  -1,   4]])

#default is row-wise sorting
>>> array4.sort()
>>> array4
array([[ -7,   0,  10,  20],
       [ -5,   1,  40, 200],
       [ -1,   1,   4,  30]])
```

```
>>> array5 = np.array([[10,-7,0, 20],
                       [-5,1,200,40],[30,1,-1,4]])

#axis =0 means column-wise sorting
>>> array5.sort(axis=0)
>>> array5
array([[ -5,  -7,  -1,   4],
       [ 10,   1,   0,  20],
       [ 30,   1, 200,  40]])
```

Operations on Arrays

```
>>> array5=np.array([[10,-7,0,20],[-5,1,200,40],[30,1,-1,4]])
>>> array5.sort()
>>> array5
array([[ -7,   0,  10,  20],
       [ -5,   1,  40, 200],
       [ -1,   1,   4,  30]])
>>> array5.sort(axis=0)
>>> array5
array([[ -7,   0,   4,  20],
       [ -5,   1,  10,  30],
       [ -1,   1,  40, 200]])
```

DESCENDING order

```
>>> array5[::-1].sort(axis=0)
>>> array5
array([[ -1,   1,  40, 200],
       [ -5,   1,  10,  30],
       [ -7,   0,   4,  20]])
```

Operations on Arrays

Concatenating Arrays

- Concatenation means joining two or more arrays.
- NumPy.concatenate() function can be used to concatenate two or more 2-D arrays either row-wise or column-wise.
- All the dimensions of the arrays to be concatenated must match exactly except for the dimension or axis along which they need to be joined.
- Any mismatch in the dimensions results in an error.
- By default, the concatenation of the arrays happens along axis=0.

```
>>> array1 = np.array([[10, 20], [-30,40]])
>>> array2 = np.zeros((2, 3), dtype=array1.
             dtype)

>>> array1
array([[ 10,  20],
       [-30,  40]])

>>> array2
array([[0, 0, 0],
       [0, 0, 0]])

>>> array1.shape
(2, 2)
>>> array2.shape
(2, 3)
```

```
>>> np.concatenate((array1,array2), axis=1)
array([[ 10,  20,   0,   0,   0],
       [-30,  40,   0,   0,   0]])

>>> np.concatenate((array1,array2), axis=0)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    np.concatenate((array1,array2))
ValueError: all the input array dimensions
except for the concatenation axis must
match exactly
```

Operations on Arrays

```
>>> ar3=np.array([[1,2],[3,4],[5,6]])

>>> ar4=np.array([[7,8,9],[10,11,12]])
>>> ar3
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> ar4
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
>>> np.concatenate((ar3,ar4.transpose()),axis=0)
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7, 10],
       [ 8, 11],
       [ 9, 12]])
```

```
>>> np.concatenate((ar3,ar4),axis=0)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    np.concatenate((ar3,ar4),axis=0)
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at index 0 has size 2 and the array at index 1 has size 3
>>> np.concatenate((ar3,ar4),axis=1)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    np.concatenate((ar3,ar4),axis=1)
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 3 and the array at index 1 has size 2
```

Operations on Arrays

Reshaping Arrays

- We can modify the shape of an array using the reshape() function.
- Reshaping an array cannot be used to change the total number of elements in the array.
- Attempting to change the number of elements in the array using reshape() results in an error

```
>>> array3 = np.arange(10,22)
>>> array3
array([10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21])


>>> array3.reshape(3,4)
array([[10, 11, 12, 13],
       [14, 15, 16, 17],
       [18, 19, 20, 21]])

>>> array3.reshape(2,6)
array([[10, 11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20, 21]])
```

Operations on Arrays

Splitting Arrays

- numpy.split() splits an array along the specified axis.
- By default, NumPy.split() splits along axis = 0.
- Consider the array given below:

```
>>> array4
array([[ 10,  -7,   0,  20],
       [ -5,   1, 200,  40],
       [ 30,   1,  -1,   4],
       [  1,   2,   0,   4],
       [  0,   1,   0,   2]])

# [1,3] indicate the row indices on which
# to split the array
>>> first, second, third = numpy split(array4,
    [1, 3])

# array4 is split on the first row and
# stored on the sub-array first
>>> first
array([[10, -7,  0, 20]])

# array4 is split after the first row and
# upto the third row and stored on the
# sub-array second
>>> second
array([[ -5,   1, 200,  40],
       [ 30,   1,  -1,   4]])

# the remaining rows of array4 are stored
# on the sub-array third
>>> third
array([[1, 2, 0, 4],
       [0, 1, 0, 2]])
```

Operations on Arrays

Splitting Arrays ..... cont

```
#[1, 2], axis=1 give the columns indices
#along which to split
>>> firstc, secondc, thirdc =numpy split(array4,
[1, 2], axis=1)
>>> firstc
array([[10],
       [-5],
       [30],
       [ 1],
       [ 0]])

>>> secondc
array([[-7],
       [ 1],
       [ 1],
       [ 2],
       [ 1]])

>>> thirdc
array([[  0,  20],
       [200,  40],
       [ -1,   4],
       [  0,   4],
       [  0,   2]])
```

```
# 2nd parameter 2 implies array is to be
# split in 2 equal parts axis=1 along the
# column axis
>>> firsthalf, secondhalf =np.split(array4,2,
axis=1)
>>> firsthalf
array([[10, -7],
       [-5,  1],
       [30,  1],
       [ 1,  2],
       [ 0,  1]])

>>> secondhalf
array([[  0,  20],
       [200,  40],
       [ -1,   4],
       [  0,   4],
       [  0,   2]])
```

Operations on Arrays

Statistical Operations on Arrays

Let us consider two arrays:

```
>>> arrayA = np.array([1,0,2,-3,6,8,4,7])
>>> arrayB = np.array([[3,6],[4,2]])
```

1. The max() function finds the maximum element from an array.

```
# max element form the whole 1-D array
>>> arrayA.max()
8
# max element form the whole 2-D array
>>> arrayB.max()
6
# if axis=1, it gives column wise maximum
>>> arrayB.max(axis=1)
array([6, 4])
# if axis=0, it gives row wise maximum
>>> arrayB.max(axis=0)
array([4, 6])
```

2. The min() function finds the minimum element from an array.

```
>>> arrayA.min()
-3
>>> arrayB.min()
2
>>> arrayB.min(axis=0)
array([3, 2])
```

Operations on Arrays

3. The sum() function finds the sum of all elements of an array.

```
>>> arrayA.sum()
25
>>> arrayB.sum()
15
#axis is used to specify the dimension
#on which sum is to be made. Here axis = 1
#means the sum of elements on the first row
>>> arrayB.sum(axis=1)
array([9, 6])
```

4. The mean() function finds the average of elements of the array.

```
>>> arrayA.mean()
3.125
>>> arrayB.mean()
3.75
>>> arrayB.mean(axis=0)
array([3.5, 4. ])

>>> arrayB.mean(axis=1)
array([4.5, 3. ])
```

Operations on Arrays

Statistical Operations on Arrays

5. The std() function is used to find standard deviation of an array of elements.

```
>>> arrayA.std()
3.550968177835448

>>> arrayB.std()
1.479019945774904

>>> arrayB.std(axis=0)
array([0.5, 2. ])

>>> arrayB.std(axis=1)
array([1.5, 1. ])
```

numpy.loadtxt() and numpy.genfromtxt() are functions used to load data from files. The savetxt() function is used to save a NumPy array to a text file.

Some ndarray methods

- ndarray. tolist ()
  - The contents of self as a nested list
- ndarray. copy ()
  - Return a copy of the array
- ndarray. fill (scalar)
  - Fill an array with the scalar value

Some NumPy functions

abs()
add()
binomial()
cumprod()
cumsum()
floor()
histogram()

- min()
- max()

multipy()

polyfit()

randint()

shuffle()

transpose()

Numpy – Creating vectors

- From lists
  - numpy.array

```
# as vectors from lists
>>> a = numpy.array([1,3,5,7,9])
>>> b = numpy.array([3,5,6,7,9])
>>> c = a + b
>>> print(c)
[4, 8, 11, 14, 18]

>>> type(c)
(<type 'numpy.ndarray'>)

>>> c.shape
(5,)
```

Numpy – Creating matrices

```
>>> l = [[1, 2, 3], [3, 6, 9], [2, 4, 6]]  # create a list
>>> a = numpy.array(l)  # convert a list to an array
>>>print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
>>> a.shape
(3, 3)
>>> print(a.dtype)  # get type o
int64

# or directly as matrix
>>> M = array([[1, 2], [3, 4]])
>>> M.shape
(2,2)
>>> M.dtype
dtype('int64')
```

```
#only one type
>>> M[0,0] = "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for long() with base 10: 'h ello'

>>> M = numpy.array([[1, 2], [3, 4]], dtype=complex)
>>> M
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Numpy – Matrices use

```
>>> print(a)
[[1 2 3]
 [3 6 9]
 [2 4 6]]
>>> print(a[0])  # this is just like a list of lists
[1 2 3]
>>> print(a[1, 2])  # arrays can be given comma separated indices
9
>>> print(a[1, 1:3])  # and slices
[6 9]
>>> print(a[:,1])
[2 6 4]
>>> a[1, 2] = 7
>>> print(a)
[[1 2 3]
 [3 6 7]
 [2 4 6]]
>>> a[:, 0] = [0, 9, 8]
>>> print(a)
[[0 2 3]
 [9 6 7]
 [8 4 6]]
```

Numpy – Creating arrays

- Generation functions

```
>>> x = arange(0, 10, 1) # arguments: start, stop, step
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> numpy.linspace(0, 10, 25)
array([  0.        ,   0.41666667,   0.83333333,   1.25      ,
         1.66666667,   2.08333333,   2.5       ,   2.91666667,
         3.33333333,   3.75      ,   4.16666667,   4.58333333,
         5.        ,   5.41666667,   5.83333333,   6.25      ,
         6.66666667,   7.08333333,   7.5       ,   7.91666667,
         8.33333333,   8.75      ,   9.16666667,   9.58333333,  10.        ])
>>> numpy.logspace(0, 10, 10, base=numpy.e)
array([  1.00000000e+00,   3.03773178e+00,   9.22781435e+00,
         2.80316249e+01,   8.51525577e+01,   2.58670631e+02,
         7.85771994e+02,   2.38696456e+03,   7.25095809e+03,
         2.20264658e+04])
```

Numpy – Creating arrays

```
# a diagonal matrix
>>> numpy.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])

>>> b = numpy.zeros(5)
>>> print(b)
[ 0.  0.  0.  0.  0.]
>>> b.dtype
dtype('float64')
>>> n = 1000
>>> my_int_array = numpy.zeros(n, dtype=numpy.int)
>>> my_int_array.dtype
dtype('int32')

>>> c = numpy.ones((3,3))
>>> c
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Numpy – array creation and use

```
>>> d = numpy.arange(5)  # just like range()
>>> print(d)
[0 1 2 3 4]

>>> d[1] = 9.7
>>> print(d)  # arrays keep their type even if elements changed
[0 9 2 3 4]

>>> print(d*0.4)  # operations create a new array, with new type
[ 0.   3.6  0.8  1.2  1.6]

>>> d = numpy.arange(5, dtype=numpy.float)
>>> print(d)
[ 0.  1.  2.  3.  4.]

>>> numpy.arange(3, 7, 0.5)  # arbitrary start, stop and step
array([ 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

Numpy – array creation and use

```
>>> x, y = numpy.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
>>> x
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])
# random data
>>> numpy.random.rand(5,5)
array([[ 0.51531133,  0.74085206,  0.99570623,  0.97064334,  0.5819413 ],
       [ 0.2105685 ,  0.86289893,  0.13404438,  0.77967281,  0.78480563],
       [ 0.62687607,  0.51112285,  0.18374991,  0.2582663 ,  0.58475672],
       [ 0.72768256,  0.08885194,  0.69519174,  0.16049876,  0.34557215],
       [ 0.93724333,  0.17407127,  0.1237831 ,  0.96840203,  0.52790012]])
```

Numpy – Creating arrays

- File I/O

```
>>> os.system('head DeBilt.txt')
"Stn", "Datum", "Tg", "qTg", "Tn", "qTn", "Tx", "qTx"
001, 19010101,   -49, 00,   -68, 00,   -22, 40
```

```
>>> numpy.savetxt('datasaved.txt', data)
>>> os.system('head datasaved.txt')
1.0000000000000000e+00 1.9010101000000000e+07 -4.9000000000000000e+01
0.0000000000000000e+00 -6.8000000000000000e+01 0.0000000000000000e+00
-2.2000000000000000e+01   4.0000000000000000e+01
1.0000000000000000e+00 1.9010102000000000e+07 -2.1000000000000000e+01
0.0000000000000000e+00 -3.6000000000000000e+01 3.0000000000000000e+01
-1.3000000000000000e+01   3.0000000000000000e+01
1.0000000000000000e+00 1.9010103000000000e+07 -2.8000000000000000e+01
0.0000000000000000e+00 -7.9000000000000000e+01 3.0000000000000000e+01
-5.0000000000000000e+00 2.0000000000000000e+01

(25568, 8)
```

Numpy – Creating arrays

```
>>> M = numpy.random.rand(3,3)
>>> M
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464  ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> numpy.save('saved-matrix.npy', M)
>>> numpy.load('saved-matrix.npy')
array([[ 0.84188778,  0.70928643,  0.87321035],
       [ 0.81885553,  0.92208501,  0.873464  ],
       [ 0.27111984,  0.82213106,  0.55987325]])
>>>
>>> os.system('head saved-matrix.npy')
NUMPYF{'descr': '<f8', 'fortran_order': False, 'shape': (3, 3), }
Ï<
£¾ðê?sy²æ?$÷ÒVñë?Ù4ê?%dn¸í?Ã[Äjóë?Ä,ZÑ?Ç
ÎåNê?ó7L{êá?0
>>>
```

Numpy – array creation and use

Two ndarrays are mutable and may be views to the same memory:

```
>>> x = np.array([1,2,3,4])
>>> y = x
>>> x is y
True
>>> id(x), id(y)
(139814289111920, 139814289111920)
>>> x[0] = 9
>>> y
array([9, 2, 3, 4])

>>> x[0] = 1
>>> z = x[:]
>>> x is z
False
>>> id(x), id(z)
(139814289111920, 139814289112080)
>>> x[0] = 8
>>> z
array([8, 2, 3, 4])
```

```
>>> x = np.array([1,2,3,4])
>>> y = x.copy()
>>> x is y
False
>>> id(x), id(y)
(139814289111920, 139814289111840)
>>> x[0] = 9
>>> x
array([9, 2, 3, 4])
>>> y
array([1, 2, 3, 4])
```

Numpy – array creation and use

```
>>> a = numpy.arange(4.0)
>>> b = a * 23.4
>>> c = b/(a+1)
>>> c += 10
>>> print c
[ 10.   21.7  25.6  27.55]

>>> arr = numpy.arange(100, 200)
>>> select = [5, 25, 50, 75, -5]
>>> print(arr[select])  # can use integer lists as indices
[105, 125, 150, 175, 195]

>>> arr = numpy.arange(10, 20 )
>>> div_by_3 = arr%3 == 0  # comparison produces boolean array
>>> print(div_by_3)
[ False False  True False False  True False False  True False]
>>> print(arr[div_by_3])  # can use boolean lists as indices
[12 15 18]

>>> arr = numpy.arange(10, 20) . reshape((2,5))
[[10 11 12 13 14]
 [15 16 17 18 19]]
```

Numpy – array functions

- Most array methods have equivalent functions

```
>>> arr.sum()
45
>>> numpy.sum(arr)
45
```

- Ufuncs provide many element-by-element math, trig., etc. operations
  - e.g., add(x1, x2), absolute(x), log10(x), sin(x), logical_and(x1, x2)

- See http://numpy.scipy.org