LAB # Numpy,

NumPy Arrays

python objects:

high-level number objects: integers, floating point

containers: lists (costless insertion and append), dictionaries (fast lookup)

Numpy provides:

extension package to Python for multi-dimensional arrays

closer to hardware (efficiency)

designed for scientific computation (convenience)

Also known as array oriented computing

```
import numpy as np
a = np.array([0, 1, 2, 3])
print(a)

print(np.arange(10))
[0 1 2 3]
[0 1 2 3 4 5 6 7 8 9]
```

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
#python lists
L = range(1000)
%timeit [i**2 for i in L]
299 µs ± 4.78 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

a = np.arange(1000)
%timeit a**2
 1.87 µs ± 20.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
 each)
```

Creating arrays

Manual Construction of arrays

```
#1-D
a = np.array([0, 1, 2, 3])
a

#print dimensions
a.ndim
```

```
#shape
a.shape

len(a)

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

b

b.ndim

b.shape

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

c = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])

c

c.ndim

c.shape
```

Functions for creating arrays

using arrange function

```
# arange is an array-valued version of the built-in Python range
function

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

b = np.arange(1, 10, 2) #start, end (exclusive), step

b
```

Using Linspace

```
#using linspace

a = np.linspace(0, 1, 6) #start, end, number of points

a
```

```
#common arrays

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

a

b = np.zeros((3, 3))

b

c = np.eye(3)  #Return a 2-D array with ones on the diagonal and zeros elsewhere.

c

d = np.eye(3, 2) #3 is number of rows, 2 is number of columns, index of diagonal start with 0

d

#create array using diag function

a = np.diag([1, 2, 3, 4]) #construct a diagonal array.

a

np.diag(a)   #Extract diagonal


#create array using random

#Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).
a = np.random.rand(4)

a
Out[ ]:

a = np.random.randn(4)#Return a sample (or samples) from the "standard normal" distribution.  ***Gausian***

a
```

Note:

For random samples from N (\mu, \sigma^2), use:

sigma * np.random.randn(...) + mu

Basic DataTypes

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.arange(10)

a.dtype

#You can explicitly specify which data-type you want:

a = np.arange(10, dtype='float64')
a

#The default data type is float for zeros and ones function

a = np.zeros((3, 3))

print(a)

a.dtype
```

other datatypes

```
d = np.array([1+2j, 2+4j])   #Complex datatype

print(d.dtype)

b = np.array([True, False, True, False])  #Boolean datatype

print(b.dtype)

s = np.array(['Ram', 'Robert', 'Rahim'])

s.dtype
```

Each built-in data type has a character code that uniquely identifies it.

'b' − boolean

'i' − (signed) integer

'u' − unsigned integer

'f' − floating-point

'c' − complex-floating point

'm' − timedelta

'M' − datetime

'O' − (Python) objects

'S', 'a' – (byte-)string

'U' – Unicode

'V' – raw data (void)

For more details

https://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html

Indexing and Slicing

Indexing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

```
a = np.arange(10)

print(a[5])  #indices begin at 0, like other Python sequences (and C/C++)

# For multidimensional arrays, indexes are tuples of integers:
a = np.diag([1, 2, 3])
print(a[2, 2])

a[2, 1] = 5 #assigning value

a
```

Slicing

```
a = np.arange(10)

a

a[1:8:2] # [startindex: endindex(exclusive) : step]

#we can also combine assignment and slicing:
a = np.arange(10)
a[5:] = 10
a

b = np.arange(5)
a[5:] = b[::-1]  #assigning

a
```

Copies and Views

A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. 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

b = a[::2]
b

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

np.shares_memory(a, b)

b[0] = 10
b

a  #eventhough we modified b,  it updated 'a' because both shares same
memory

a = np.arange(10)

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

np.shares_memory(a, c)

c[0] = 10

a
```

Fancy Indexing

NumPy arrays can be indexed with slices, but also with boolean or integer arrays **(masks)**. This method is called **fancy indexing**. It creates copies not views.

Using Boolean Mask

```
a = np.random.randint(0, 20, 15)
a

mask = (a % 2 == 0)

extract_from_a = a[mask]

extract_from_a
```

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

```
a[mask] = -1
a
```

Indexing with an array of integers

```
a = np.arange(0, 100, 10)

a

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

a[[2, 3, 2, 4, 2]]

# New values can be assigned

a[[9, 7]] = -200

a
```