(https://www.datalorelabs.com)

Numpy

- Numpy is the core library for scientific computing in Python.
- Numpy provides a high-performance multidimensional array object, and tools for working with these arrays.
- Numpy makes programming mathematical functions more akin to writing the mathematical functions!

import numpy module

```
import numpy as np
```

Arrays

A numpy array is a analogous to python list but the elements of the array should be of same type.

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

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

The `type` of the array is called `numpy.ndarray` . (numpy n-dimensional array)

```
type(a)
```

```
numpy.ndarray
```

We associate numpy arrays with two properties **shape** and **rank**, which describe the array about the *dimension* and *shape* it is of.

- `a` is one dimensional array with 3 elements

```
print("Rank of a: ", a.ndim)
print("Shape of a: ", a.shape)
print("Total number of elements in the array: ", a.size)
print("Data type of the elements of a:", a.dtype)
```

```
Rank of a:  1
Shape of a:  (3,)
Total number of elements in the array:  3
Data type of the elements of a: int64
```

```
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)
```

```
1 2 3
[5 2 3]
```

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

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
print("Rank of b: ", b.ndim)
print("Shape of b: ", b.shape)
print("Total number of elements in b: ", b.size)
print("Data type of the elements of b:", b.dtype)
```

```
Rank of b:  2
Shape of b:  (2, 3)
Total number of elements in b:  6
Data type of the elements of b: float64
```

Array b:

```
       0            1            2       < 3 columns (axis=1)
 _____________ _____________ _____________
|             |             |             |
```

0 | 1. | 2. | 3. | |_____|_____|_____|_____|_____|_____| | | | | 1 | 4. | 5. | 6. | |_____|_____|_____|_____|_____|_____|

^

2 rows (axis=0) shape=(2, 3)

Selecting along axis-0:

b[0] –> b[0, :] –> [1., 2., 3.] b[1] –> b[1, :] –> [4., 5., 6.]

Selecting along axis-1:

b[:, 0] –> [1., 4.] b[:, 1] –> [2., 5.] b[:, 2] –> [3., 6.]

```
# accessing elements
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
1.0 2.0 4.0
```

Array Creation

Numpy provides lots of ways to create a numpy array.

- numpy `zeros` and `ones`

```
a = np.zeros((3,3))   # Create an array of all zeros
print(a)
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

```
a = np.ones((2,5))   # Create an array of all ones
print(a)
```

```
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]
```

- numpy `zeros_like` and `ones_like`

```
c = np.zeros_like(a) # Create an array of all zeros like a's shape
print(c)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

```
c = np.ones_like(b) # Create an array of all ones like b's shape
print(c)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

- numpy `full`

```
d = np.full((2,2), 10)  # Create a constant array
print(d)                # Prints "[[ 7.  7.]
                        #          [ 7.  7.]]"
```

```
[[10 10]
 [10 10]]
```

- numpy `eye` (Identity)

```
e = np.eye(3)         # Create a 3x3 identity matrix
print(e)
```

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

- numpy `random`

```
f = np.random.random((2,2))  # Create an array filled with random values
print(f)
```

```
[[ 0.36212922  0.24199098]
 [ 0.77907491  0.75820274]]
```

- numpy `arange`

```
np.arange(0, 10, 1) # arguments: start, stop, step
```

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

- numpy `linspace`

```
np.linspace(0, 10, 5)
```

```
array([  0. ,   2.5,   5. ,   7.5,  10. ])
```

- loading from file

```
%%file mydata.dat

1 11
2 92
3 81
4 52
5 14
6 23
7 22
8 11
9 0
10 1
```

```
Writing mydata.dat
```

```
np.genfromtxt("mydata.dat",)
```

```
array([[  1.,  11.],
       [  2.,  92.],
       [  3.,  81.],
       [  4.,  52.],
       [  5.,  14.],
       [  6.,  23.],
       [  7.,  22.],
       [  8.,  11.],
       [  9.,   0.],
       [ 10.,   1.]])
```

```
?np.genfromtxt
```

You can read more about array creation in the documentation (http://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation).

Array Indexing

Numpy offers several ways to index into arrays.

Slicing

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
a = np.linspace(0, 500, 6)
print(a)
```

```
[   0.  100.  200.  300.  400.  500.]
```

```
# Elements at the middle of the array
a[2:4]
```

```
array([ 200.,  300.])
```

```
# Last element
a[-1]
```

```
500.0
```

```
# Last two elements
a[-2:]
```

```
array([ 400.,  500.])
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas. When accessing multidimensional arrays, we must specify a slice for each dimension of the array

```
a =np.array([
    np.linspace(1, 3, 3),
    np.linspace(4, 6, 3),
    np.linspace(7, 9, 3)
])
print(a)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

```
print("First element: ", a[0,0])
```

```
First element:  1.0
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices

```
print("First row:\n", a[0])
```

```
First row:
 [ 1.  2.  3.]
```

```
print("Upper Right 2x2 matrix:\n", a[0:2, 1:])
print("Lower Right 2x2 matrix:\n", a[1:, 1:])
```

```
Upper Right 2x2 matrix:
 [[ 2.  3.]
 [ 5.  6.]]
Lower Right 2x2 matrix:
 [[ 5.  6.]
 [ 8.  9.]]
```

```
print("Upper Left 2x2 matrix:\n", a[:2, :2])
print("Lower Left 2x2 matrix:\n", a[1:, :2])
```

```
Upper Left 2x2 matrix:
 [[ 1.  2.]
 [ 4.  5.]]
Lower Left 2x2 matrix:
 [[ 4.  5.]
 [ 7.  8.]]
```

Boolean array indexing

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

bool_idx = (a > 2)

print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

```
print(a[bool_idx])
```

```
[3 4 5 6]
```

We can do the above operation in single line:

```
print(a[a > 2])
```

```
[3 4 5 6]
```

Read More on indexing here (https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html)

Other function to subset an array

`where`

Convert conditional indices to position index using the `where` function

```
print(a, "\n")
m, n = np.where(a > 2)
print("Axis-0: ", m)
print("Axis-1: ",n)
```

```
[[1 2]
 [3 4]
 [5 6]]

Axis-0:  [1 1 2 2]
Axis-1:  [0 1 0 1]
```

```
a[m,n]
```

```
array([3, 4, 5, 6])
```

`diag`

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
print("A:\n", A)
np.diag(A)
```

```
A:
 [[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]




array([ 0, 11, 22, 33, 44])
```

reverse diagonal

```
A[:, ::-1]
```

```
array([[ 4,  3,  2,  1,  0],
       [14, 13, 12, 11, 10],
       [24, 23, 22, 21, 20],
       [34, 33, 32, 31, 30],
       [44, 43, 42, 41, 40]])
```

```
np.diag(A[:, ::-1])
```

```
array([ 4, 13, 22, 31, 40])
```

take

```
v = np.arange(-3,3)
v
```

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

indexing via a list

```
row_indices = [1, 3, 5]
v[row_indices]
```

```
array([-2,  0,  2])
```

Doesn’t work with List

```
[-3, -2, -1,  0,  1,  2][row_indices]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-40-752504d3dd6d> in <module>()
----> 1 [-3, -2, -1,  0,  1,  2][row_indices]


TypeError: list indices must be integers or slices, not list
```

Works like a charm!

```
np.take([-3, -2, -1,  0,  1,  2], row_indices)
```

```
array([-2,  0,  2])
```

Linear Algebra

Elementwise-array operations

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

Elementwise addition; both produce the array

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

print(x + y, "\n")

print(np.add(x, y))
```

```
[[  6.   8.]
 [ 10.  12.]]

[[  6.   8.]
 [ 10.  12.]]
```

Elementwise difference; both produce the array

```
print(x - y, "\n")
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]

[[-4. -4.]
 [-4. -4.]]
```

Elementwise product; both produce the array

```
print(x * y, "\n")
print(np.multiply(x, y))
```

```
[[  5.  12.]
 [ 21.  32.]]

[[  5.  12.]
 [ 21.  32.]]
```

Elementwise division; both produce the array

```
print(x / y, "\n")
print(np.divide(x, y))
```

```
[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]

[[ 0.2         0.33333333]
 [ 0.42857143  0.5       ]]
```

Other Useful Elementwise Operations

a^2

```
print("Squaring...\n")
print("a: \n", a)
print("\na**2: \n", a**2)
print("\nnp.square(a): \n", np.square(a))
```

```
Squaring...

a:
 [[1 2]
 [3 4]
 [5 6]]

a**2:
 [[ 1  4]
 [ 9 16]
 [25 36]]

np.square(a):
 [[ 1  4]
 [ 9 16]
 [25 36]]
```

Same operation on python list raises an error.

```
list_a = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
list_a**2
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-47-7d51e112cdad> in <module>()
      1 list_a = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
----> 2 list_a**2


TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

e^{a}

```
print("exp(a):\n", np.exp(a))
```

```
exp(a):
 [[   2.71828183    7.3890561 ]
 [  20.08553692   54.59815003]
 [ 148.4131591   403.42879349]]
```

$a^{1.2}$

```
print("a**1.2:\n", np.power(a, 1.2))
```

```
a**1.2:
 [[ 1.          2.29739671]
 [ 3.73719282  5.27803164]
 [ 6.89864831  8.58581449]]
```

$ln\ (a), log_{10}\ (a)$ and $log_2\ (a)$

```
print("Natural logarithm: \n", np.log(a))
print("\nBase10 logarithm: \n", np.log10(a))
print("\nBase2 logarithm: \n", np.log2(a))
```

```
Natural logarithm:
 [[ 0.          0.69314718]
 [ 1.09861229  1.38629436]
 [ 1.60943791  1.79175947]]

Base10 logarithm:
 [[ 0.          0.30103   ]
 [ 0.47712125  0.60205999]
 [ 0.69897     0.77815125]]

Base2 logarithm:
 [[ 0.          1.        ]
 [ 1.5849625   2.        ]
 [ 2.32192809  2.5849625 ]]
```

Vector Operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide vectors with scalar numbers.

```
v1 = np.arange(0, 5)
v1
```

```
array([0, 1, 2, 3, 4])
```

```
print(v1 * 2)
print(v1 / 2)
print(v1 ** 2)
print(v1 * v1)
```

```
[0 2 4 6 8]
[ 0.   0.5  1.   1.5  2. ]
[ 0  1  4  9 16]
[ 0  1  4  9 16]
```

Inner Product

```
v2 = np.arange(5, 10)
v2
```

```
array([5, 6, 7, 8, 9])
```

$v_1 = [0, 1, 2, 3, 4]$

$v_2 = [5, 6, 7, 8, 9]$

$v_1 . v_2 = 0 * 5 + 1 * 6 + 2 * 7 + 3 * 8 + 4 * 9$

```
np.dot(v1, v2)
```

```
80
```

Vector Magnitude (self inner product)

```
sum = 0
for each element in vector:
    sum += element * element
```

```
np.sum([element*element for element in v1])
```

```
30
```

```
print(v1 @ v1)
```

```
30
```

Matrix Algebra

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

Transpose

```
A.T
```

```
array([[ 0, 10, 20, 30, 40],
       [ 1, 11, 21, 31, 41],
       [ 2, 12, 22, 32, 42],
       [ 3, 13, 23, 33, 43],
       [ 4, 14, 24, 34, 44]])
```

Matrix-Vector Multiplication

```
v1
```

```
array([0, 1, 2, 3, 4])
```

`v1` is multiplied to each row

```
A * v1
```

```
array([[  0,   1,   4,   9,  16],
       [  0,  11,  24,  39,  56],
       [  0,  21,  44,  69,  96],
       [  0,  31,  64,  99, 136],
       [  0,  41,  84, 129, 176]])
```

Elementwise Matrix Multiplication

```
A * A
```

```
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
       [ 400,  441,  484,  529,  576],
       [ 900,  961, 1024, 1089, 1156],
       [1600, 1681, 1764, 1849, 1936]])
```

Matrix Multiplication

```
A.dot(A)
```

```
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

```
A @ A
```

```
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

Alternatively we can cast the array to `Matrix`, which enables normal arithmatic opertions to perform matrix algebra.

```
A_mat = np.matrix(A)
v = np.matrix(v1).T # make it a column vector

print("Matrix A:\n", A_mat)
print("\nVector v:\n",v)
```

```
Matrix A:
 [[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]

Vector v:
 [[0]
 [1]
 [2]
 [3]
 [4]]
```

```
type(A_mat)
```

```
numpy.matrixlib.defmatrix.matrix
```

```
A_mat * A_mat
```

```
matrix([[ 300,  310,  320,  330,  340],
        [1300, 1360, 1420, 1480, 1540],
        [2300, 2410, 2520, 2630, 2740],
        [3300, 3460, 3620, 3780, 3940],
        [4300, 4510, 4720, 4930, 5140]])
```

```
v.T * A_mat
```

```
matrix([[300, 310, 320, 330, 340]])
```

```
A_mat * v
```

```
matrix([[ 30],
        [130],
        [230],
        [330],
        [430]])
```

If we try to add, subtract or multiply objects with incomplatible shapes we get an error:

```
v = np.matrix([1,2,3,4]).T
```

```
A_mat.shape, v.shape
```

```
((5, 5), (4, 1))
```

```
A_mat * v
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-71-c5c8686b9891> in <module>()
----> 1 A_mat * v


/Users/vikramkalabi/anaconda3/envs/carnd-term1/lib/python3.5/site-packages/numpy/m
atrixlib/defmatrix.py in __mul__(self, other)
    307         if isinstance(other, (N.ndarray, list, tuple)) :
    308             # This promotes 1-D vectors to row vectors
--> 309             return N.dot(self, asmatrix(other))
    310         if isscalar(other) or not hasattr(other, '__rmul__') :
    311             return N.dot(self, other)


ValueError: shapes (5,5) and (4,1) not aligned: 5 (dim 1) != 4 (dim 0)
```

Other Useful Functions

Sum

```
A
```

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

```
A.sum()
```

```
550
```

Column-wise sum or Reduce by row

```
A.sum(axis=0)
```

```
array([100, 105, 110, 115, 120])
```

Row-wise sum or Reduce by column

```
A.sum(axis=1)
```

```
array([ 10,  60, 110, 160, 210])
```

Statistics

Mean

```
print("Mean of A: \n", A.mean())
print("Column-wise mean of A: \n", A.mean(axis=0))
print("Row-wise mean of A: \n", A.mean(axis=1))
```

```
Mean of A:
 22.0
Column-wise mean of A:
 [ 20.  21.  22.  23.  24.]
Row-wise mean of A:
 [  2.  12.  22.  32.  42.]
```

Variance

```
print("Variance of A: \n", A.var())
print("Column-wise variance of A: \n", A.var(axis=0))
print("Row-wise variance of A: \n", A.var(axis=1))
```

```
Variance of A:
 202.0
Column-wise variance of A:
 [ 200.  200.  200.  200.  200.]
Row-wise variance of A:
 [ 2.  2.  2.  2.  2.]
```

Standard deviation

```
print("Standard Deviation of A: \n", A.std())
print("Column-wise Standard Deviation of A: \n", A.std(axis=0))
print("Row-wise Standard Deviation of A: \n", A.std(axis=1))
```

```
Standard Deviation of A:
 14.2126704036
Column-wise Standard Deviation of A:
 [ 14.14213562  14.14213562  14.14213562  14.14213562  14.14213562]
Row-wise Standard Deviation of A:
 [ 1.41421356  1.41421356  1.41421356  1.41421356  1.41421356]
```

Min and Max

```
print("Minimum of A: \n", A.min())
print("Column-wise Minimum of A: \n", A.min(axis=0))
print("Row-wise Minimum of A: \n", A.min(axis=1))
```

```
Minimum of A:
 0
Column-wise Minimum of A:
 [0 1 2 3 4]
Row-wise Minimum of A:
 [ 0 10 20 30 40]
```

```
print("Maximum of A: \n", A.max())
print("Column-wise Maximum of A: \n", A.max(axis=0))
print("Row-wise Maximum of A: \n", A.max(axis=1))
```

```
Maximum of A:
 44
Column-wise Maximum of A:
 [40 41 42 43 44]
Row-wise Maximum of A:
 [ 4 14 24 34 44]
```

Broadcasting

source: Justin Johnson (http://cs.stanford.edu/people/jcjohns/)

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
X = np.array([[1,2],[3, 4] ,[5,6], [7,8]])
v = np.array([1, 2])

print("X: \n", X)
print("\nv:\n", v)
```

```
X:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]

v:
 [1 2]
```

We can add the vector v to each row of the matrix x, storing the result in the matrix y

```
Y = np.zeros_like(X)

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    Y[i, :] = X[i, :] + v


print(Y)
```

```
[[ 2  4]
 [ 4  6]
 [ 6  8]
 [ 8 10]]
```

Adding `v` to every row of matrix `X` is equivalent to form a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `X` and `vv`

```
vv = np.tile(v, (4, 1)) # stack four rows of v
print("Stacked vectors: \n", vv)

Y = X + vv  # Add x and vv elementwise
print("\nResult: \n", Y)
```

```
Stacked vectors:
 [[1 2]
 [1 2]
 [1 2]
 [1 2]]

Result:
 [[ 2  4]
 [ 4  6]
 [ 6  8]
 [ 8 10]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`. Subject to certain constraints, the smaller array is *broadcast* across the larger array so that they have compatible shapes.

```
Y = X + v  # Add v to each row of x using broadcasting
print(Y)
```

```
[[ 2  4]
 [ 4  6]
 [ 6  8]
 [ 8 10]]
```

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both

shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

For more on broadcasting please read Eric's Broadcasting Docs (http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc) or the documentation (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)

```
v = np.array([1])
print("Rank of v: ", v.ndim)
X + v
```

```
Rank of v:  1




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

Compute outer product of vectors

To compute an outer product, we first reshape `v` to be a column vector of shape `(3, 1)`. We can then broadcast it against `w` to yield an output of shape `(3, 2)`, which is the outer product of `v` and `w`:

```
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

Array Reshape, Concatenation, Stacking and Copy

Reshape

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A
```

```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

```
m, n = A.shape
```

```
B = A.reshape((1, m*n))
print(B.shape)
print(B)
```

```
(1, 25)
[[ 0  1  2  3  4 10 11 12 13 14 20 21 22 23 24 30 31 32 33 34 40 41 42 43
  44]]
```

```
B[0, 0:5] = -1
```

```
A
```

```
array([[-1, -1, -1, -1, -1],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

```
B = A.flatten()
print(B.shape)
print(B)
```

```
(25,)
[-1 -1 -1 -1 -1 10 11 12 13 14 20 21 22 23 24 30 31 32 33 34 40 41 42 43 44]
```

```
B[0:5] = 10
B
```

```
array([10, 10, 10, 10, 10, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24, 30, 31,
       32, 33, 34, 40, 41, 42, 43, 44])
```

```
A
```

```
array([[-1, -1, -1, -1, -1],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

Concatenation and Stacking

Join a sequence of arrays along an existing axis.

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

np.concatenate((a, b), axis=0)
```

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

```
np.concatenate((a, b.T), axis=1)
```

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

```
np.vstack((a,b))
```

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

```
np.hstack((a,b.T))
```

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

Copy

To achieve high performance, assignments in Python usually do not copy the underlaying objects. This is important for example when objects are passed between functions, to avoid an excessive amount of memory copying when it is not necessary (technical term: pass by reference).

```
A = np.array([[1, 2], [3, 4]])

A
```

```
array([[1, 2],
       [3, 4]])
```

```
# now B is referring to the same array data as A
B = A
```

```
# changing B affects A
B[0,0] = 10

B
```

```
array([[10,  2],
       [ 3,  4]])
```

```
A
```

```
array([[10,  2],
       [ 3,  4]])
```

If we want to avoid this behavior, so that when we get a new completely independent object `B` copied from `A`, then we need to do a so-called "deep copy" using the function `copy`:

```
B = A.copy()
```

```
# now, if we modify B, A is not affected
B[0,0] = -5

B
```

```
array([[-5,  2],
       [ 3,  4]])
```

```
A
```

```
array([[10,  2],
       [ 3,  4]])
```

Further Reading

- http://numpy.scipy.org (http://numpy.scipy.org)
- http://scipy.org/Tentative_NumPy_Tutorial (http://scipy.org/Tentative_NumPy_Tutorial)
- http://scipy.org/NumPy_for_Matlab_Users (http://scipy.org/NumPy_for_Matlab_Users) - A Numpy guide for MATLAB users.