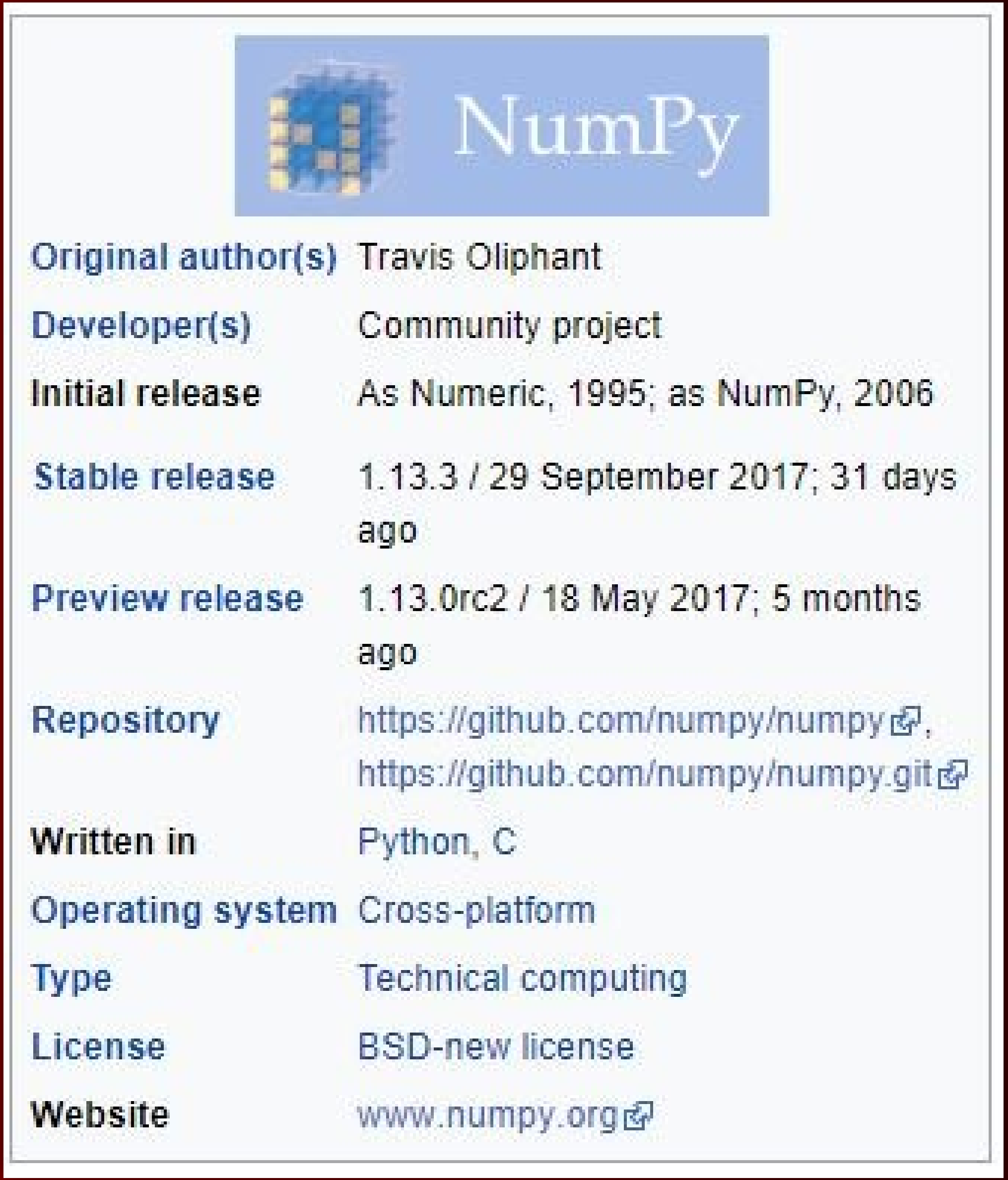
# NUMPY

Module VI, Chapter II

# INTRODUCTION TO NUMPY

- NumPy is the fundamental package for scientific computing in Python.
- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays.



NumPy

| | |
|---|---|
| Original author(s) | Travis Oliphant |
| Developer(s) | Community project |
| Initial release | As Numeric, 1995; as NumPy, 2006 |
| Stable release | 1.13.3 / 29 September 2017; 31 days ago |
| Preview release | 1.13.0rc2 / 18 May 2017; 5 months ago |
| Repository | https://github.com/numpy/numpy, https://github.com/numpy/numpy.git |
| Written in | Python, C |
| Operating system | Cross-platform |
| Type | Technical computing |
| License | BSD-new license |
| Website | www.numpy.org |

- Operations including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
- At the core of the NumPy package, is the *ndarray* object
- This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

## THERE ARE SEVERAL IMPORTANT DIFFERENCES BETWEEN NUMPY ARRAYS AND THE STANDARD PYTHON SEQUENCES:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.

## TRAITS

- NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring rewriting some code, mostly inner loops using NumPy.
- Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted, and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars.
- Python bindings of the widely used computer vision library OpenCV utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, slicing or masking with other arrays are very efficient ways to access specific pixels of an image. The NumPy array as universal data structure in OpenCV for images, extracted feature points, filter kernels and many more vastly simplifies the programming workflow and debugging.

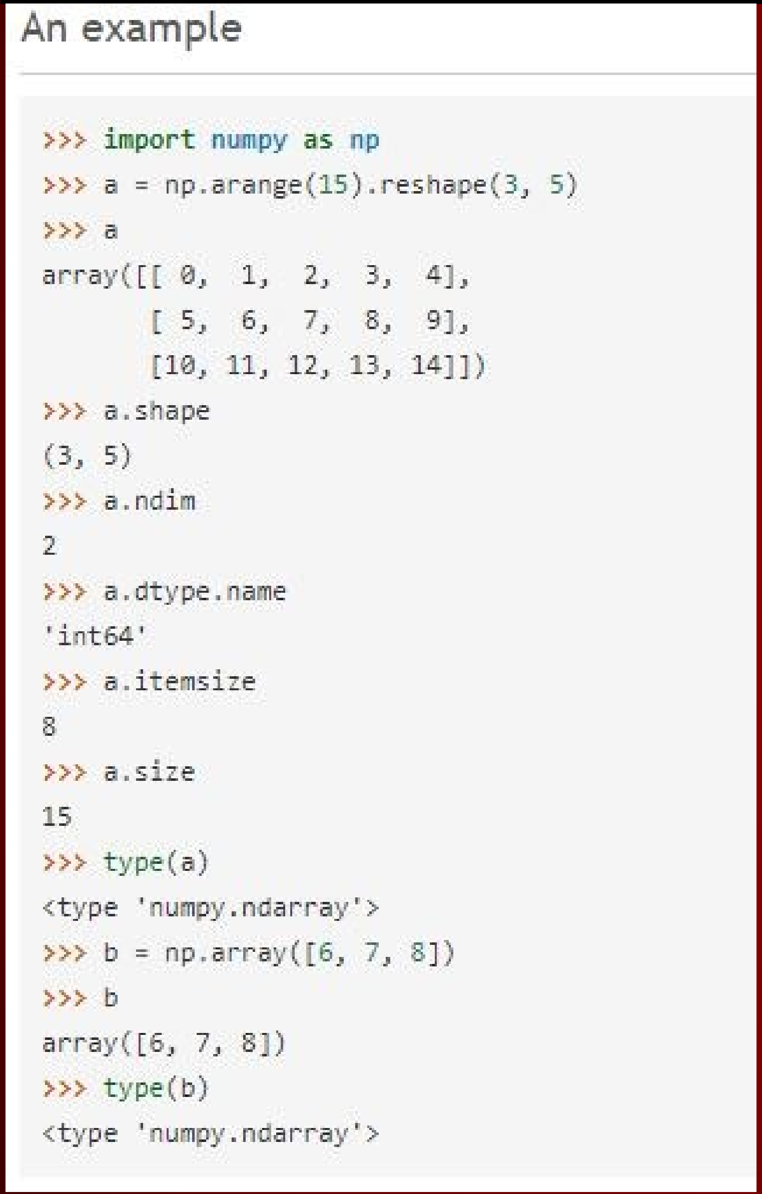
## THE BASICS

- NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called *axes*. The number of axes is *rank*.
- For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3. In the example pictured below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.
- [[ 1., 0., 0.],
- [ 0., 1., 2.]]

## THE MORE IMPORTANT ATTRIBUTES OF AN ndarray OBJECT ARE:

- **ndarray.ndim**: The number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as rank.
- **ndarray.shape**:The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.
- **ndarray.size:** The total number of elements of the array. This is equal to the product of the elements of shape.
- **ndarray.dtype**: An object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- **ndarray.itemsize**: The size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.
- **ndarray.data**: The buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

# EXAMPLE



An example

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

## THE ndarray DATA STRUCTURE

- The core functionality of numpy is its "ndarray", for *n*-dimensional array, data structure.
- These arrays are strided views on memory. In contrast to Python's built-in list data structure, these arrays are homogeneously typed: all elements of a single array must be of the same type.

## ARRAY CREATION

Create an array from a regular Python list or tuple using the array() function

- **import numpy as np**
- a = np.array([2,3,4])
- b = np.array([1.2, 3.5, 5.1])
- c = np.array([(1.5,2,3), (4,5,6)])
- d = np.array( [ [1,2], [3,4] ], dtype=complex )
- e=np.zeros( (3,4) )
- f=np.ones( (2,3,4), dtype=np.int16 )
- g=np.empty( (2,3) )
- h=np.arange( 10, 30, 5 )
- i=np.linspace( 0, 2, 9 )

## PRINTING ARRAYS

- When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout: the last axis is printed from left to right, the second-to-last is printed from top to bottom, the rest are also printed from top to bottom, with each slice separated from the next by an empty line.
- One-dimensional arrays are then printed as rows, bi dimensionals as matrices and tri dimensionals as lists of matrices.
- >>> a = np.arange(6) *# 1d array*
- >>> print(a)
- [0 1 2 3 4 5]

## PRINTING ARRAYS

- >>> b = np.arange(12).reshape(4,3) *# 2d array*
- >>> print(b)

```
[[ 0 1 2]
[ 3 4 5]
[ 6 7 8]
[ 9 10 11]]
```

## BASIC OPERATIONS

- Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.
- >>> a = np.array( [20,30,40,50] )
- >>> b = np.arange( 4 )
- >>> b

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

- >>> c = a-b
- >>> c
- array([20, 29, 38, 47])

## UNIVERSAL FUNCTIONS

- NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
• >>> B = np.arange(3)
• >>> B
array([0, 1, 2])
• >>> np.exp(B)
array([ 1. , 2.71828183, 7.3890561 ])
• >>> np.sqrt(B)
array([ 0. , 1. , 1.41421356])
• >>> C = np.array([2., -1., 4.])
• >>> np.add(B, C)
array([ 2., 0., 6.])
```

## LIMITATIONS

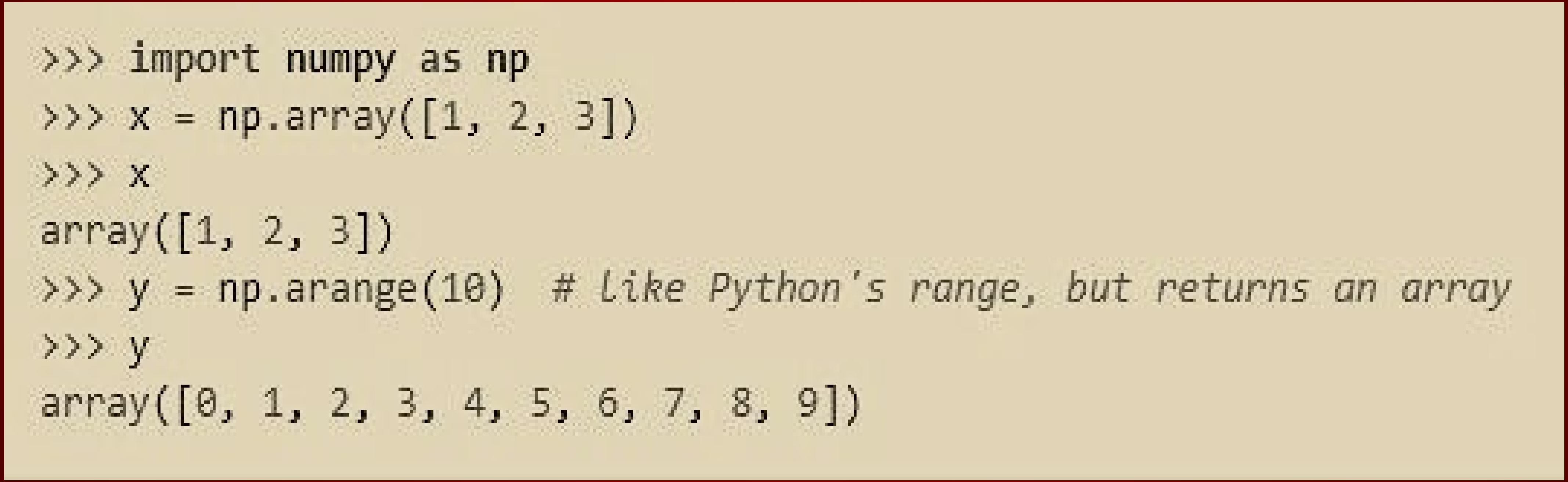
- Inserting or appending entries to an array is not as trivially possible as it is with Python's lists.
- The np.pad(...) routine to extend arrays actually creates new arrays of the desired shape and padding values, copies the given array into the new one and returns it.
- NumPy's np.concatenate([a1,a2]) operation does not actually link the two arrays but returns a new one, filled with the entries from both given arrays in sequence.

## LIMITATIONS

- Reshaping the dimensionality of an array with np.reshape(...) is only possible as long as the number of elements in the array does not change.
- These circumstances originate from the fact that NumPy's arrays must be views on contiguous memory buffers. A replacement package called Blaze attempts to overcome this limitation.

## EXAMPLES

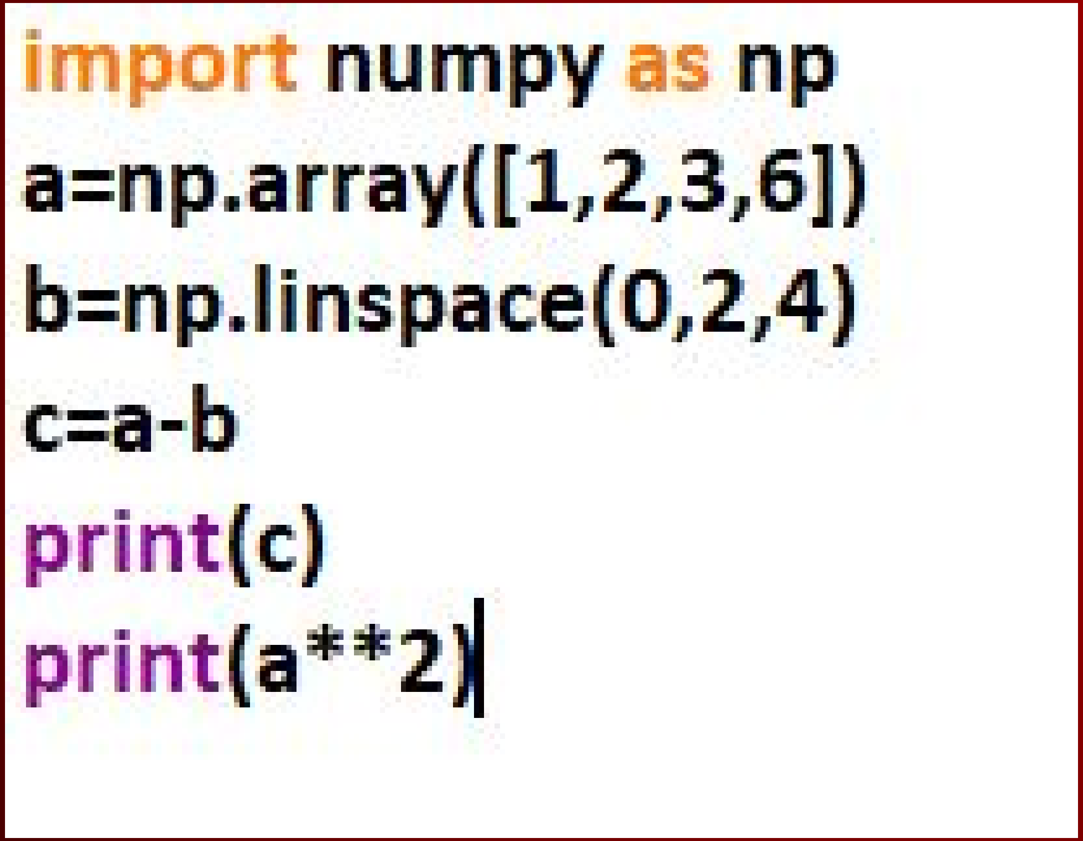
- **Array creation**



```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> x
array([1, 2, 3])
>>> y = np.arange(10)  # like Python's range, but returns an array
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# EXAMPLES

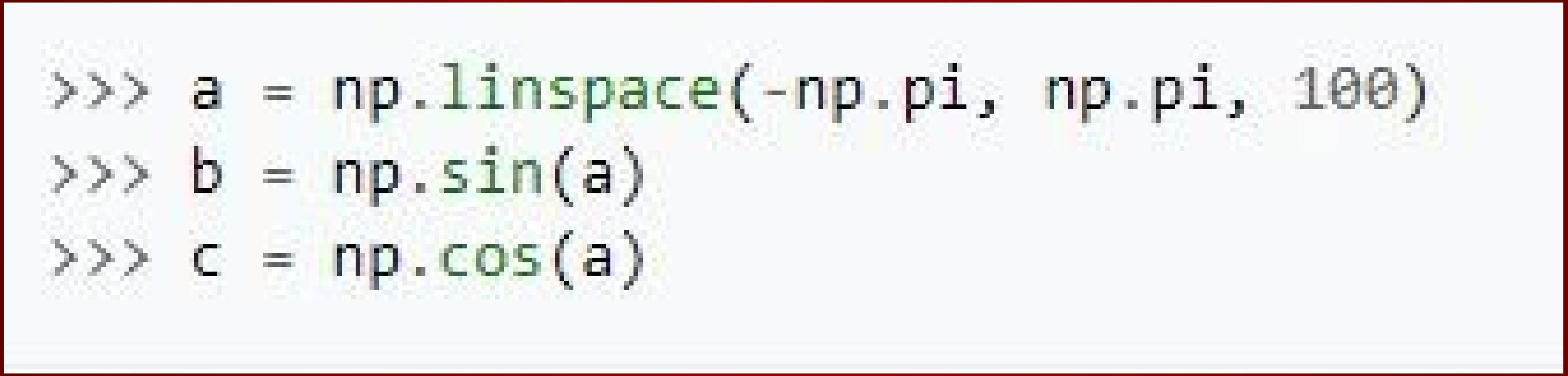
- **Basic operations**



```
import numpy as np
a=np.array([1,2,3,6])
b=np.linspace(0,2,4)
c=a-b
print(c)
print(a**2)
```

# EXAMPLES

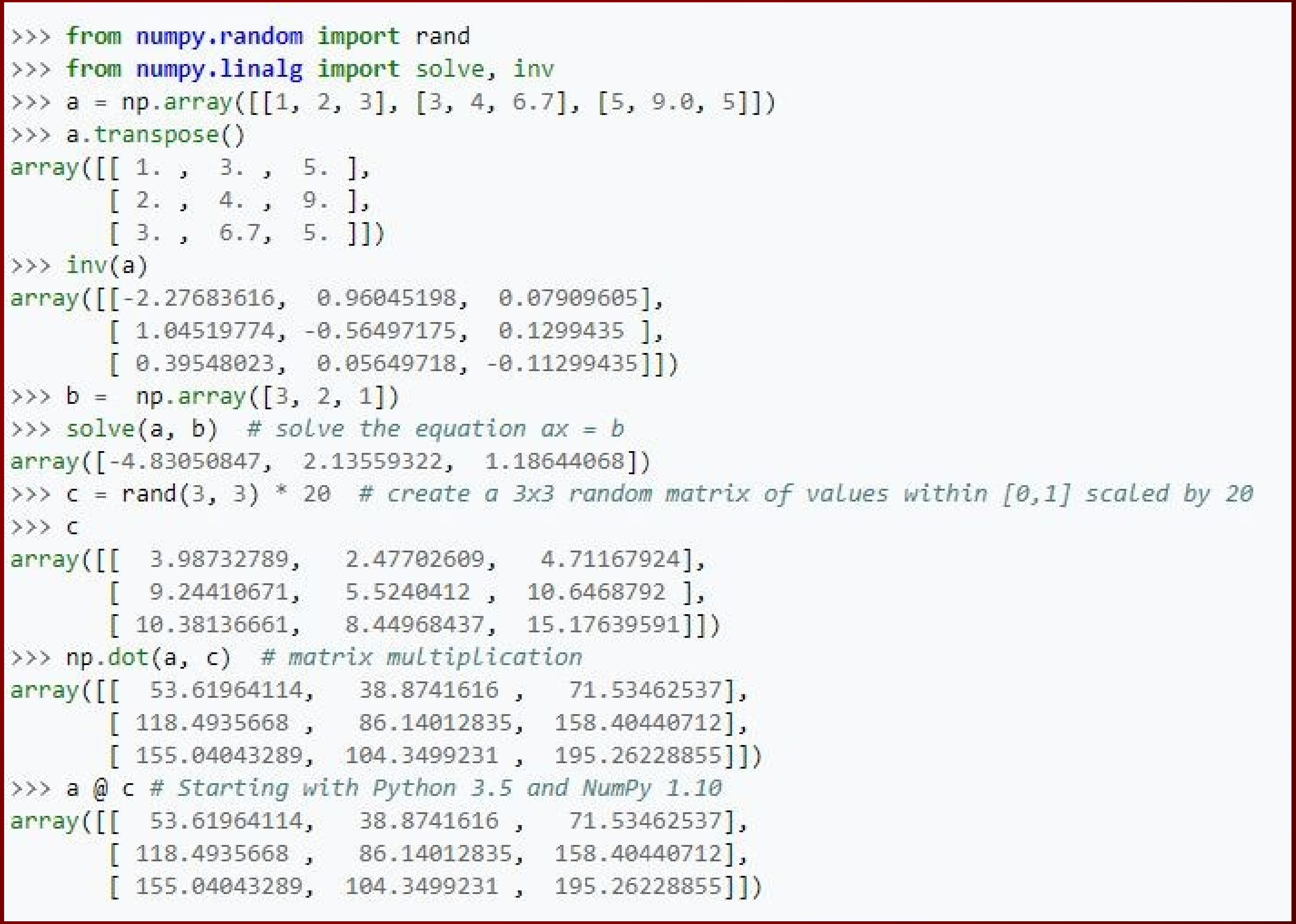
- **Universal functions**



```
>>> a = np.linspace(-np.pi, np.pi, 100)
>>> b = np.sin(a)
>>> c = np.cos(a)
```

## EXAMPLES

- **Linear algebra**



```
>>> from numpy.random import rand
>>> from numpy.linalg import solve, inv
>>> a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]])
>>> a.transpose()
array([[ 1. ,  3. ,  5. ],
       [ 2. ,  4. ,  9. ],
       [ 3. ,  6.7,  5. ]])
>>> inv(a)
array([[-2.27683616,  0.96045198,  0.07909605],
       [ 1.04519774, -0.56497175,  0.1299435 ],
       [ 0.39548023,  0.05649718, -0.11299435]])
>>> b =  np.array([3, 2, 1])
>>> solve(a, b)  # solve the equation ax = b
array([-4.83050847,  2.13559322,  1.18644068])
>>> c = rand(3, 3) * 20  # create a 3x3 random matrix of values within [0,1] scaled by 20
>>> c
array([[  3.98732789,   2.47702609,   4.71167924],
       [  9.24410671,   5.5240412 ,  10.6468792 ],
       [ 10.38136661,   8.44968437,  15.17639591]])
>>> np.dot(a, c)  # matrix multiplication
array([[  53.61964114,   38.8741616 ,   71.53462537],
       [ 118.4935668 ,   86.14012835,  158.40440712],
       [ 155.04043289,  104.3499231 ,  195.26228855]])
>>> a @ c # Starting with Python 3.5 and NumPy 1.10
array([[  53.61964114,   38.8741616 ,   71.53462537],
       [ 118.4935668 ,   86.14012835,  158.40440712],
       [ 155.04043289,  104.3499231 ,  195.26228855]])
```