Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Jan. 11 – 15, 2021

Python for Data Analytics

NumPy I

Outline

- What is NumPy?
- Creating Arrays
- Manipulating Arrays
- Array Broadcasting
- Statistical Operations
- Matrix Operations

What is "NumPy" Module?

- The "NumPy" (Numeric Python) package provides basic routines for manipulating large arrays and matrices of numeric data
- The "SciPy" (Scientific Python) package extends the functionality of NumPy with a substantial collection of useful algorithms
  - Minimization, Fourier transformation, regression, and other applied math techniques
- NumPy and SciPy are open source add-on modules (not Python Standard Library)
- More than functionalities of commercial packages like MatLab
- To catch up functionalities of R
- `>>> import numpy as np`

NumPy History

- Numeric (ancestor of Numpy)
  - Released in 1995 by Jim Hugunin by generalizing Jim Fulton's matrix package
- Numarray
  - A more flexible replacement for Numeric
  - Faster for large arrays, slower than Numeric on small arrays
- SciPy module
  - Created by Travis Oliphant et al. in 2001
  - Provides scientific and technical operations
  - NumPy 1.0 released (as part of SciPy) in 2006 by porting Numarray's features to Numeric
- NumPy module separated from SciPy as a stand-alone package

numpy.ndarray: The N-dimensional Array

- A multidimensional container of items of the same type and size
  - shape: a tuple of N non-negative integers that specify the sizes of each dimension
  - data-type object (dtype): the type of items in the array

- 1D numpy.ndarray object
  - Example: `array([3,6])` `array([3.5,6.4,7.2])`

- 2D numpy.ndarray object
  - Example: `array([[1,0,2], [3,5,2]])`

- 3D numpy.ndarray object
  - Example: `array([[[0,0,1],[1,2,3]], [[1,0,2],[2,3,4]], [[3,5,2],[1,1,1]]])`

Array Example

```
>>> import numpy as np
>>> x = np.array([[1, 2, 3], [4, 5, 6]], int)
>>> print(x)
[[1 2 3]
 [4 5 6]]
>>> type(x)
<class 'numpy.ndarray'>
>>> x.shape
(2, 3)                # (number of rows, number of columns)
>>> x.dtype
dtype('int64')
```

List vs. Array.array vs. Numpy.ndarray

- **Lists**
  - Simple
  - Can't constrain the type of elements stored in a list

```
>>> a = [[0]*3 for i in range(3)]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> b = [1, 3.5, 'hello']
```

- **Array.array**
  - All elements of the array must be of the same numeric type
  - May be used to interface with C code

```
>>> import array
>>> a = array.array('i', [1, 2, 3])
>>> a
array('i', [1, 2, 3])
```

- **Numpy.ndarray**
  - Supports various computations on arrays and matrices

```
>>> import numpy
>>> a = numpy.array([1, 2, 3], float)
>>> a
array([1., 2., 3.])
```

Creating Arrays

array()

- *np*.array(*object*, *dtype=None*, ...)
  - *object*: usually a list
  - *dtype*: the desired data type
    - If omitted, the type will be determined as the minimum type required to hold the objects

```
>>> l = [1, 2, 3, 4, 5]
>>> np.array(l)
>>> np.array(l, int)
>>> np.array(l, dtype='i')
>>> np.array(l, dtype=np.uint8)
>>> np.array(l, dtype='f')
>>> np.array(l, float)
```

Data types		Type code
Boolean	`bool`	`?`

Integers	`int8`
	`int16`
	`int32`
	`int64 (int)`
		`i1, b`
		`i2, h`
		`i4, i`
		`i8, l, q`

Unsigned integers	`uint8`
	`uint16`
	`uint32`
	`uint64`
		`u1, B`
		`u2, H`
		`u4, I`
		`u8, L, Q,`

Floating points	`float16`
	`float32`
	`float64 (float)`
	`float128`
		`f2`
		`f4, f`
		`f8, d`
		`f16, g`

Complex	`complex64`
	`complex128 (complex)`
	`complex256`
		`c8, F`
		`c16, D`
		`c32, G`

Unicode string	`unicode`	`U`

np.inf and np.nan

- *np*.`inf` (infinity)
  - Too large to be represented
  - e.g., n / 0, `np.inf * np.inf`, `np.inf + np.inf`, ...

```
>>> np.inf
inf
>>> a = array([3, 2, 5])
>>> a / 0
array([inf, inf, inf])
```

- *np*.`nan` (not-a-number)
  - A value that is undefined or unrepresentable
  - e.g., 0 / 0, `np.inf` / `np.inf`, `np.inf * 0`, `np.inf` - `np.inf`, ...

```
>>> np.nan
nan
>>> np.log(-1)
nan
>>> np.log([-1, 1, 2])
array([    nan, 0.    , 0.69314718])
```

full() and empty()

- *np*.`full(`*shape*, *value*[, *dtype*][, *order*]`)`
  - Return a new array of given shape and type, filled with *value*
- *np*.`full_like(`*a*, *value*, …`)`

```
>>> np.full(5, 2)
array([2, 2, 2, 2, 2])
>>> np.full((2,3), -1, float)
array([[-1., -1., -1.],
       [-1., -1., -1.]])
```

- *np*.`empty(`*shape*[, *dtype*][, *order*]`)`
  - Return a new array of given shape and type, without initializing entries
  - Faster than others
- *np*.`empty_like(`*a*, …`)`

```
>>> np.empty((2,4))
array([[6.91542951e-310,
6.91542951e-310, 1.24120911e-316,
        1.24120911e-316],
       [6.91542951e-310,
6.91542951e-310, 0.00000000e+000,
        0.00000000e+000]])
```

zeros() and ones()

- *np*.`zeros`(*shape*[, *dtype*][, *order*])
  - Return a new array of given shape and type, filled with zeros
  - *order*: 'C' = row-major (C-style),
'F' = column-major (Fortran-style)

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- *np*.`ones`(*shape*, [, *dtype*][, *order*])
  - Return a new array of given shape and type, filled with ones

```
>>> np.ones(4)
array([1., 1., 1., 1.])
>>> np.ones((3,3))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

zeros_like() and ones_like()

- *np*.`zeros_like(`*a*[, *dtype*], … `)`
  - Return an array of zeros with the same shape and type as a given array

```
>>> a = np.full((2,3), -1, float)
array([[-1., -1., -1.],
       [-1., -1., -1.]])
>>> np.zeros_like(a)
array([[0, 0, 0],
       [0, 0, 0]])
```

- *np*.`ones_like(`*a*[, *dtype*], … `)`
  - Return an array of ones with the same shape and type as a given array

```
>>> a = np.full((3,4), 10)
>>> np.ones_like(a)
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
```

identity() and eye()

- *np*.`identity(`*n*[, *dtype*]`)`
  - Return the identity array (a square array with ones on the main diagonal)
  - *n:* number of rows (and columns)

```
>>> np.identity(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

- *np*.`eye(`*N*[, *M*][, *k*][, *dtype*][, *order*]`)`
  - Return a 2D array with ones on the diagonal and zeros elsewhere
  - *M*: number of columns (default *N*)
  - *k*: index of the diagonal (default *0*)

```
>>> np.eye(2)
array([[1., 0.],
       [0., 1.]])
>>> np.eye(3,4,1)
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

arange()

- *np*.`arange(`[*start*,]*stop*[, *step*][, *dtype*]`)`
  - Return an array with evenly spaced values within a given interval: [*start*, *stop*)
  - When using a non-integer step, it may produce unexpected results
    → Use `np.linspace()` instead

```
>>> np.arange(10, 30, 5)
array([10 15 20 25])
>>> np.arange(0, 2, 0.3)
array([0.  0.3 0.6 0.9 1.2 1.5 1.8])
>>> np.arange(0, -1, -0.1)
array([ 0. , -0.1, -0.2, -0.3, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9])
>>> np.arange(8.0, 8.4, 0.05)
array([8.  , 8.05, 8.1 , 8.15, 8.2 , 8.25, 8.3 , 8.35, 8.4 ])
```

linspace()

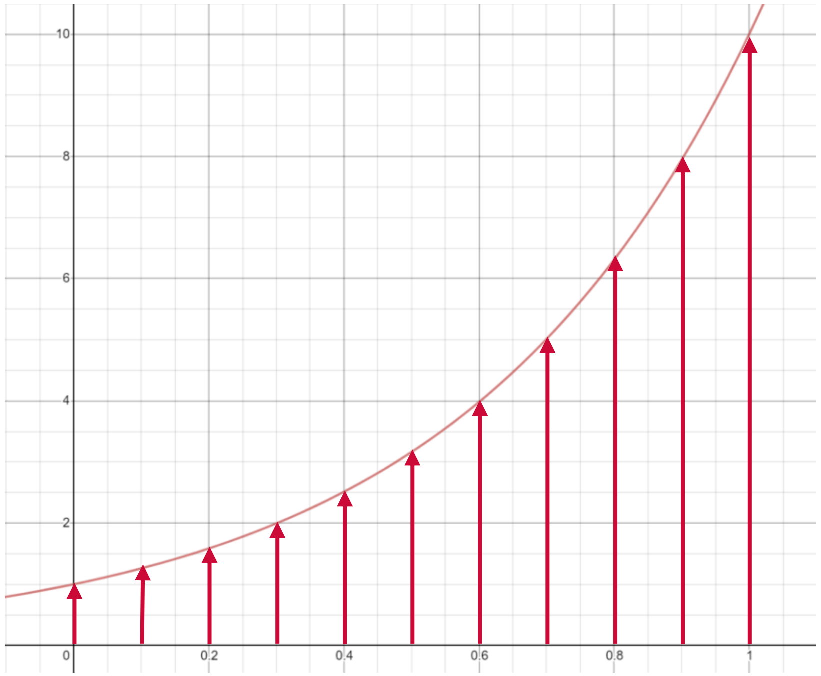
- *np*.`linspace(`*start*, *stop*[, *num*][, *endpoint*]…`)`
  - Return an array with evenly spaced numbers over a specified interval: [*start*, *stop*]
  - *num*: the number of evenly spaced samples (default 50)
  - *endpoint*: if False, the endpoint of the interval is excluded (default True)

```
>>> np.linspace(0, 100, 5)
array([  0.,   25.,   50.,   75., 100.])
>>> np.linspace(0, 100, 5, endpoint=False)
array([ 0., 20., 40., 60., 80.])
>>> np.linspace(0, 4, 4)
array([0.        , 1.33333333, 2.66666667, 4.        ])
>>> np.linspace(8.0, 8.4, 8, False)
array([8.  , 8.05, 8.1 , 8.15, 8.2 , 8.25, 8.3 , 8.35])
```

logspace()

- *np*.`logspace(`*start*, *stop*[, *num*][, *endpoint*][, *base*], … `)`
  - Return *num* numbers spaced evenly on a log scale
  - In linear space, the sequence starts at $base^{start}$ and ends with $base^{stop}$
  - *base*: the base of the log space (default: 10.0)

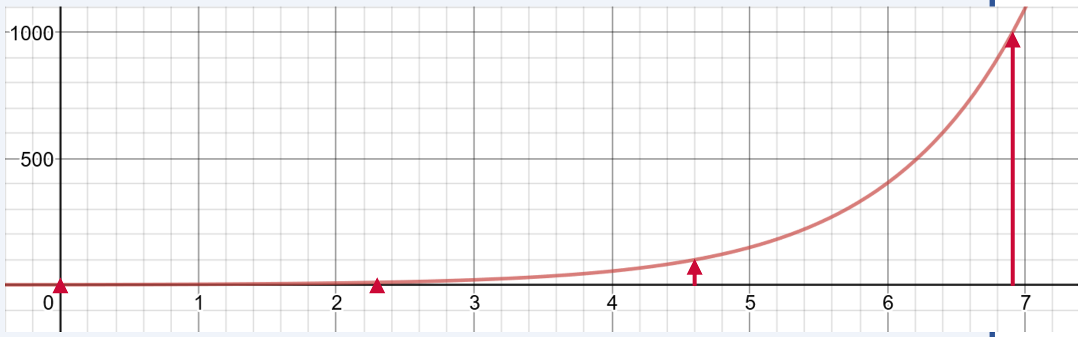
```
>>> np.logspace(0, 1, 11)
array([ 1.        ,  1.25892541,  1.58489319,
        1.99526231,  2.51188643,  3.16227766,
        3.98107171,  5.01187234,  6.30957344,
        7.94328235,  10.        ])

>>> [ 10**x for x in np.linspace(0, 1 ,11) ]
```



geomspace()

- *np*.geomspace(*start*, *stop*[, *num*][, *endpoint*], …)
  - Return *num* numbers spaced evenly on a log scale (a geometric progression)
  - Each output sample is a constant multiple of the previous

```
>>> import math
>>> [ math.exp(i) for i in np.linspace(math.log(1), math.log(1000), 4) ]
[1.0, 9.999999999999998, 99.99999999999996, 999.9999999999998]
>>> np.geomspace(1, 1000, 4)
array([   1.,   10.,  100., 1000.])
>>> np.geomspace(-1000, -1, num=4)
array([-1000.,  -100.,   -10.,    -1.])
>>> np.geomspace(1, 256, 9)
array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128., 256.])
```



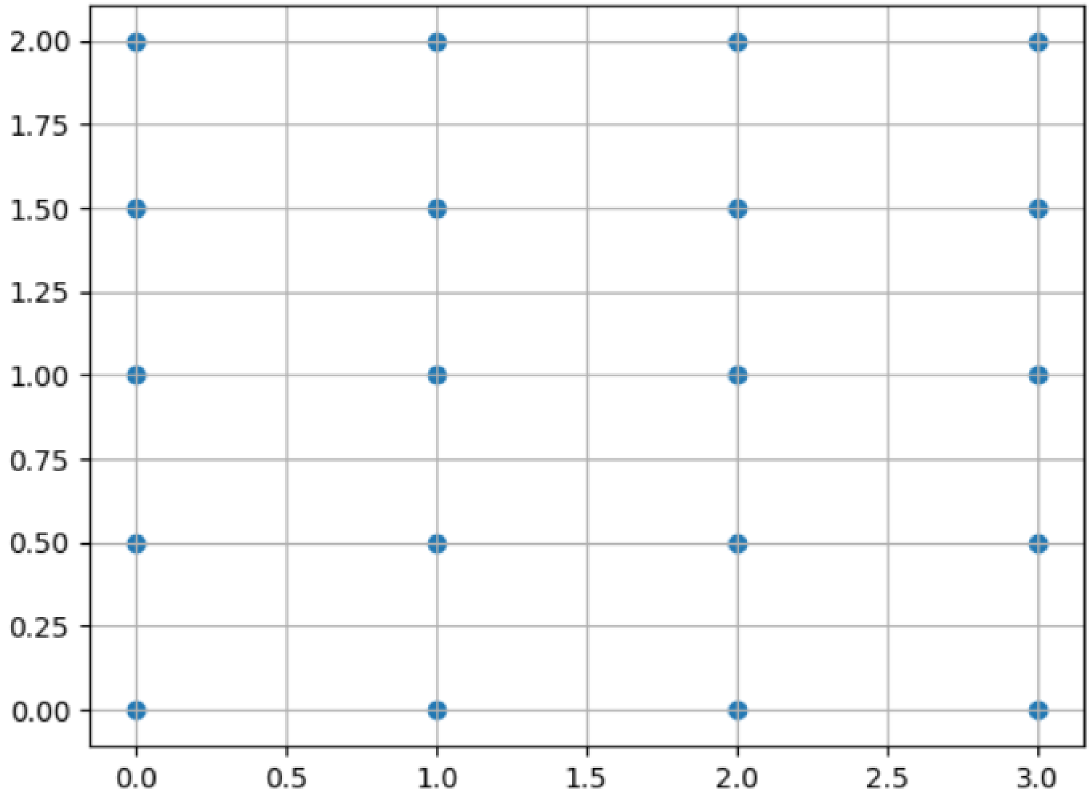
meshgrid()

- *np*.`meshgrid`(*x1*, *x2*, ..., *xn*, ...)
  - Return coordinate matrices from coordinate vectors

```
>>> x = np.linspace(0, 2, 3)
>>> x
array([0., 1., 2.])
>>> y = np.linspace(0, 1, 3)
>>> y
array([0. , 0.5, 1. ])
>>> xv, yv = np.meshgrid(x, y)
```

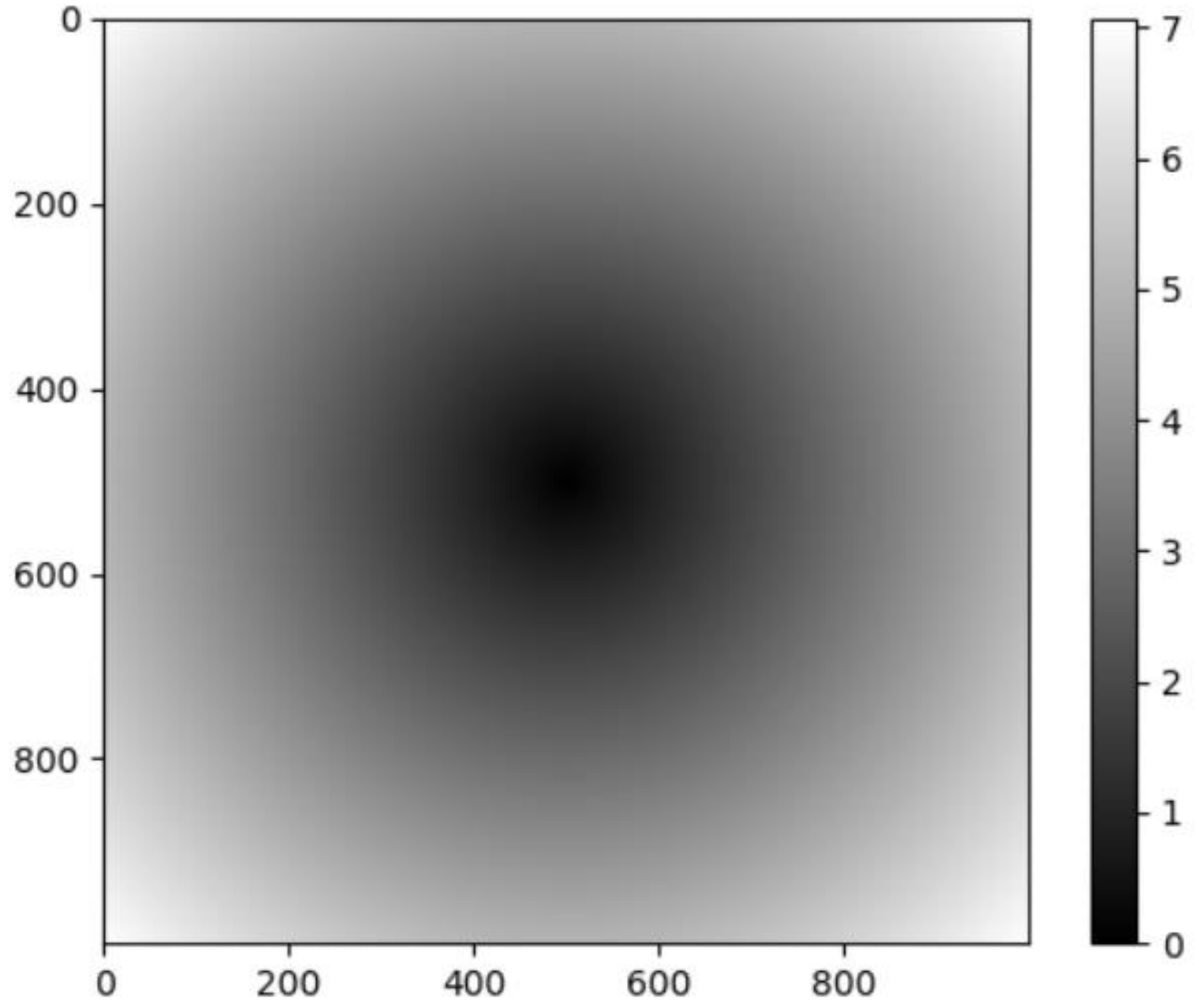
```
>>> xv
array([[0., 1., 2.],
       [0., 1., 2.],
       [0., 1., 2.]])
>>> yv
array([[0. , 0. , 0. ],
       [0.5, 0.5, 0.5],
       [1. , 1. , 1. ]])
```

meshgrid()




```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(xv, yv)
>>> plt.grid(True)
>>> plt.show()

>>> mg = [list(zip(x,y)) for x, y \
          in zip(xv, yv)]
>>> mg
[[(0.0, 0.0), (1.0, 0.0), (2.0, 0.0)],
 [(0.0, 0.5), (1.0, 0.5), (2.0, 0.5)],
 [(0.0, 1.0), (1.0, 1.0), (2.0, 1.0)]]
```

meshgrid() Example




```python
import numpy as np
import matplotlib.pyplot as plt

pts = np.arange(-5, 5, 0.01)
x, y = np.meshgrid(pts, pts)
z = np.sqrt(x**2 + y**2)
plt.imshow(z, cmap=plt.cm.gray)
plt.colorbar()
plt.show()
```

random.random() and random.randint()

- numpy.random submodule provides various random number generators
- *np*.`random.random(`*size*`)`
  - Return random floats in the interval [0.0, 1.0)
  - *size*: integer or tuple of integers for output array shape

```
>>> np.random.random((3,2))
array([[0.44325748, 0.61687924],
       [0.68575248, 0.60672728],
       [0.82738475, 0.38333312]])
```

- *np*.`random.randint(`*low*[, *high*] [, size],...`)`
  - Return random integers in the interval [*low*, *high*)
  - If *high* is omitted, results are from [0, *low*)

```
>>> np.random.randint(100)
91
>>> np.random.randint(10, size=5)
array([6, 5, 9, 5, 1])
```

random.rand() and random.randn()

- *np*.`random.rand(`*d0, d1, …, dn*`)`
  - Return random floats in the interval [0.0, 1.0)
  - *d0*, *d1*, …, *dn*: the dimensions of the output array (not tuple)

```
>>> np.random.rand(3,2)
array([[0.01250554, 0.43358273],
       [0.3730851 , 0.66585267],
       [0.03250322, 0.6765952 ]])
```

- *np*.`random.randn(`*d0, d1, …, dn*`)`
  - Return samples from the "standard normal" distribution $N(0, 1)$
  - For random samples from $N(\mu, \sigma^2)$, use σ * `np.random.randn() +` μ

```
>>> np.random.randn()
0.5288740495934477
>>> np.random.randn(3,2)
array([[-0.64964129,  1.26874147],
       [ 0.88909375,  0.51902268],
       [ 2.1553506 , -1.73896019]])
```

random.uniform()

- *np*.`random.uniform(`[*low=0.0*], [*high=1.0*], [*size*]`)`
  - Draw samples from a uniform distribution
  - Samples are uniformly distributed over the interval [*low*, *high*)

```
>>> np.random.uniform(1.0, 2.0)
1.6937903416817646
>>> np.random.uniform(1.0, 2.0, 5)
array([1.1922301 , 1.72618062, 1.82763685, 1.32765954, 1.45356649])
>>> import math
>>> np.random.uniform(0, math.pi, (3,4))
array([[0.53309063, 1.56158409, 2.34588318, 2.40615273],
       [0.28675017, 0.37922173, 1.24792002, 0.05974539],
       [2.02903176, 0.98991193, 0.61068395, 2.40537881]])
```

random.normal()

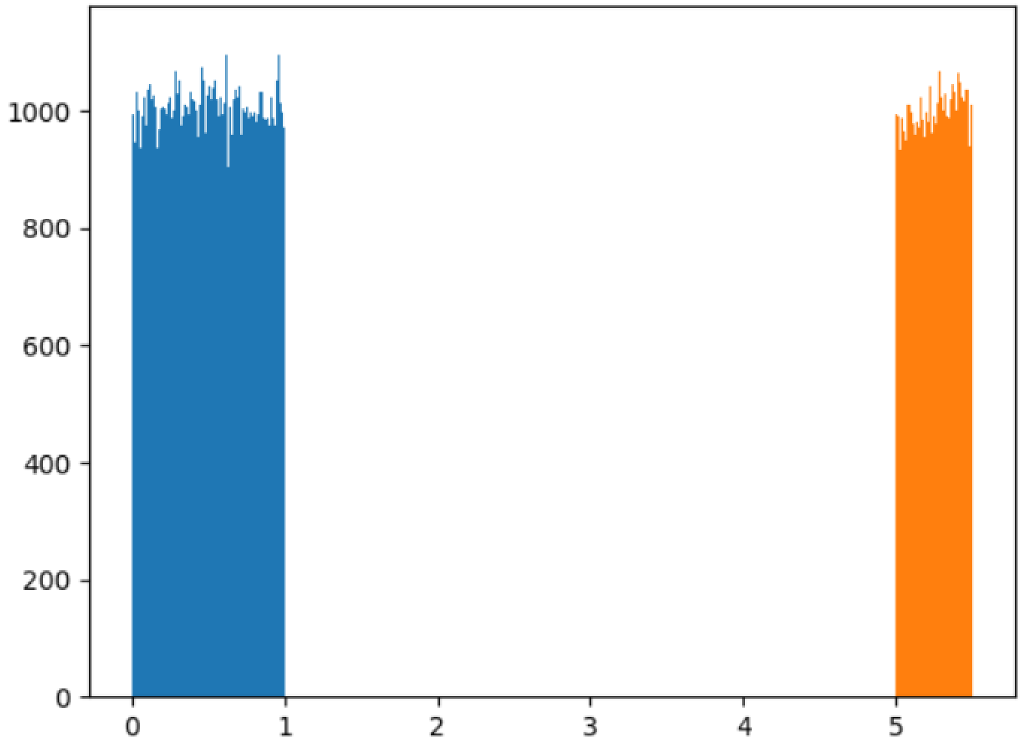
- *np*.`random.normal(`[*loc=0.0*], [*scale=1.0*], [*size*]`)`
  - Draw samples from a normal (Gaussian) distribution
  - *loc*: mean of the distribution, *scale*: standard deviation of the distribution
  - `random.normal(loc, scale) == loc + scale*random.normal()`

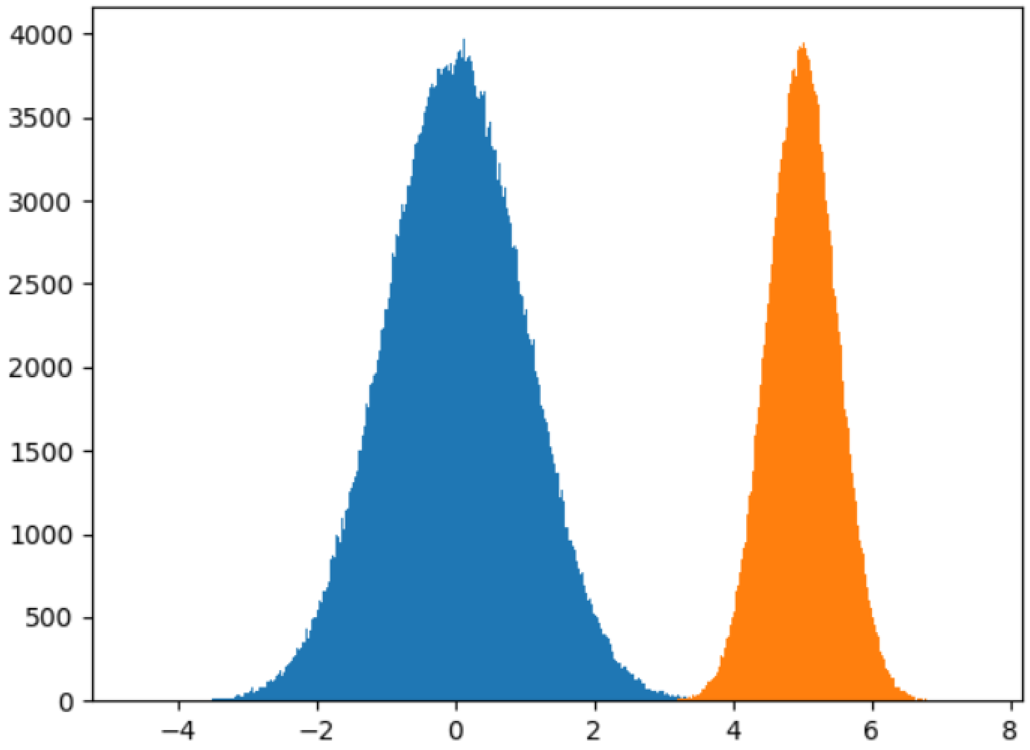
```
>>> np.random.normal()
-0.6901865834995796
>>> np.random.normal(size=5)
array([-0.1249383 ,  1.10623467, -0.38662773, -0.60593157,  0.71653932])
>>> np.random.normal(size=(2,3))
array([[ 0.2093123 ,  0.40961339,  0.64944229],
       [ 1.06565541,  1.0441453 , -0.81924546]])
```

Uniform vs. Normal

```
values = np.random.uniform(size=1000000)
plt.hist(values, 1000)
plt.hist(5+0.5*values, 1000)
plt.show()
```

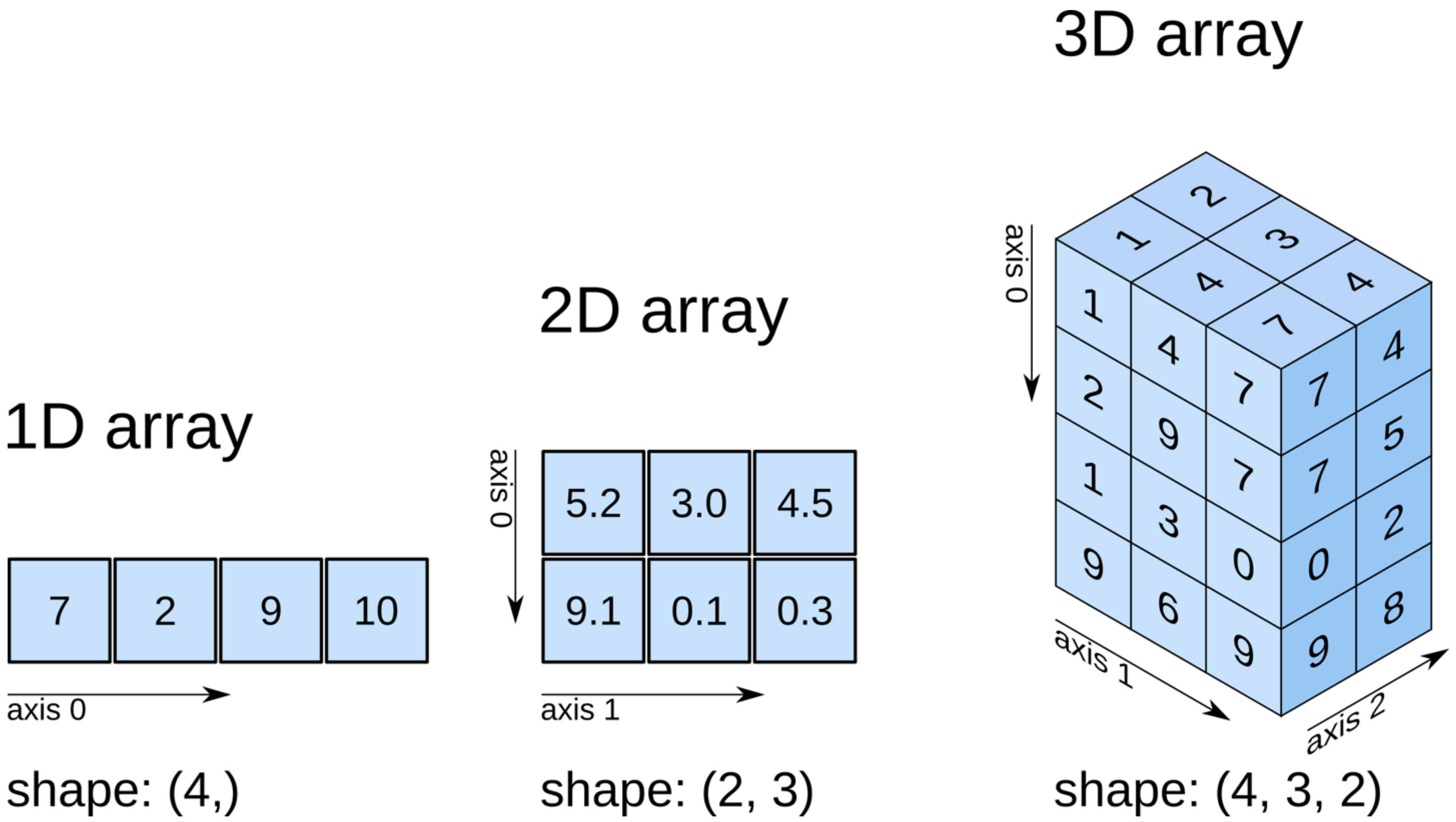
```
values = np.random.normal(size=1000000)
plt.hist(values, 1000)
plt.hist(5+0.5*values, 1000)
plt.show()
```





Manipulating Arrays

Array Shape



https://www.oreilly.com/library/view/elegant-scipy/9781491922927/ch01.html

Iteration

- Use the `for` loop

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> for i in a:
...     print(i)

0
1
2
3
4
```

```
>>> b = np.arange(6).reshape(2,3)
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> for x in b:
...     print(x)

[0 1 2]
[3 4 5]
```

Iteration (cont'd)

- One "for" loop for each dimension

```
>>> a = np.arange(18).reshape(3,2,-1)
>>> a
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]],

       [[12, 13, 14],
        [15, 16, 17]]])
```

```
>>> for x in a:
...   print(x)
...   print('-'*10)
[[0 1 2]
 [3 4 5]]
----------
[[ 6  7  8]
 [ 9 10 11]]
----------
[[12 13 14]
 [15 16 17]]
----------
```

```
>>> for x in a:
...   for y in x:
...     print(y)
...     print('-'*10)
[0 1 2]
----------
[3 4 5]
----------
[6 7 8]
----------
[ 9 10 11]
----------
[12 13 14]
----------
[15 16 17]
----------
```

Reshaping

- *a*.`reshape`(*shape*)
  - Return an array containing the same data with a new shape
- *a*.`resize`(*shape*)
  - Change shape and size of array in-place
  - Same as:
    a.`shape` = *shape*
- *a*.`flatten`()
  - Return a flattened array

```
>>> a=np.arange(6)
>>> a.shape
(6,)
>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.resize(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.flatten()
array([0, 1, 2, 3, 4, 5])
>>> a.shape = (1, 6)    # ???
```

Reshaping (cont'd)

- One dimension can be -1 in *a*.`reshape()`
  - The value is inferred from the length of the array and remaining dimensions

```
>>> a = np.arange(12)
>>> a.reshape(3, -1)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.reshape(-1, 3)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
>>> a.reshape(2, 2, -1)
array([[[ 0,  1,  2],
        [ 3,  4,  5]],
       [[ 6,  7,  8],
        [ 9, 10, 11]]])
>>> a.reshape(-1, 2, 3)
array([[[ 0,  1,  2],
        [ 3,  4,  5]],
       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

Transposing

- *a*.`transpose`(*axes*)
  - Return a view of the array with axes transposed
  - For a 1-D array, no effect
  - For a 2-D array, this is a standard matrix transpose
  - For an n-D array and *axes* are given, their order indicates how the axes are permuted. Otherwise, shapes are reversed
    - x.shape = (1, 2, 3)
      → x.transpose(1, 2, 0).shape = (2, 3, 1)

```
>>> a = np.arange(6)
>>> a.transpose()
array([0, 1, 2, 3, 4, 5])
>>> b = a.reshape(3,2)
>>> b
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b.transpose()
array([[0, 2, 4],
       [1, 3, 5]])
>>> c = np.arange(6).reshape(1,2,3)
>>> c.transpose().shape
(3, 2, 1)
```

Indexing

- **Single element indexing**
  - Similar to Python lists
  - Negative indices for indexing from the end of the array

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

- **Multidimensional indexing**
  - Used for multidimensional arrays
  - If you use fewer indices than dimensions, you get a subdimensional array
  - `x[0,2]` == `x[0][2]`: `x[0][2]` is more inefficient as a new temporary array is created

```
>>> x.shape = (2, 5)
>>> x[0]
array([0, 1, 2, 3, 4])
>>> x[1, 3]
8
>>> x[0][2]
2
```

Slicing

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> x[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
>>> y = np.arange(35).reshape(5,7)
>>> y[2,:]
array([14, 15, 16, 17, 18, 19, 20])
>>> y[:,2]
array([ 2,  9, 16, 23, 30])
>>> y[1:5:2,::3]
array([[ 7, 10, 13],
       [21, 24, 27]])
>>> y[-1:,-2:]
array([[33, 34]])
```

Views

- Slices of arrays do not copy the internal array data, but only produce new "views" of the original data

```
>>> x = np.arange(6).reshape(2, 3)
>>> y = x[:,1]
>>> y
array([1, 4])
>>> y[0] = 9
>>> y
array([9, 4])
>>> x
array([[0, 9, 2],
       [3, 4, 5]])
```

Index Arrays

- NumPy arrays can be indexed with other arrays or lists

```
>>> x = np.arange(8, 0, -1)
>>> x
array([8, 7, 6, 5, 4, 3, 2, 1])
>>> x[np.array([3, 3, 1, 6])]
array([5, 5, 7, 2])
>>> x[[3, 3, -1, 6]]
array([5, 5, 1, 2])
>>> x[np.array([[1,1],[2,3]])]
array([[7, 7],
       [6, 5]])
```

```
>>> y = np.arange(35).reshape(5,7)
>>> y[np.array([0,2,4]),np.array([0,1,2])]
array([ 0, 15, 30])
>>> y[np.array([0,2,4]), 1]
array([ 1, 15, 29])
>>> y[np.array([0,2,4])]
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])
>>> y[:, np.array([0, 2])]   # ???
```

Boolean Index Arrays

- Only choose the elements that satisfy the Boolean expression

```
>>> a = np.arange(1,7)
>>> a
array([1, 2, 3, 4, 5, 6])
>>> b = [True, True, False, False,
True, False]
>>> a[b]
array([1, 2, 5])
>>> c = [1, 1, 0, 0, 1, 0]
>>> a[c]
array([2, 2, 1, 1, 2, 1])
```

```
>>> x = np.arange(9).reshape(3,3)
>>> y = (x % 2 == 0)
>>> y
array([[ True, False,  True],
       [False,  True, False],
       [ True, False,  True]])
>>> x[y]
array([0, 2, 4, 6, 8])
>>> x[x % 2 == 0]
array([0, 2, 4, 6, 8])
```

Arithmetic Operations

- Shape of both operands must be same!
- One operand can be a constant

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([4, 5, 6], float)
>>> a + b
array([5., 7., 9.])
>>> a - b
array([-3., -3., -3.])
>>> a * b
array([ 4., 10., 18.])
>>> b / a
array([4. , 2.5, 2. ])
```

```
>>> b % a
array([0., 1., 0.])
>>> b**a
array([  4.,  25., 216.])
>>> a * 0.5
array([0.5, 1. , 1.5])
>>> b > 5
array([False, False,  True])
>>> a + b == 5
array([ True, False, False])
```

Operations: Python List vs. NumPy Array

- Operator *
  - List * n: repetition of the whole list
  - Array * n: multiply n to every element in the array

```
>>> L = [1, 2, 3]
>>> A = np.array([1, 2, 3])
>>> L * 2
[1, 2, 3, 1, 2, 3]
>>> A * 2
array([2, 4, 6])
```

- Operator +
  - List1 + List2: concatenation of two lists
  - Array1 + Array2: Element-wise addition

```
>>> L + L
[1, 2, 3, 1, 2, 3]
>>> A + A
array([2, 4, 6])
```

Concatenating

- *np*.concatenate(*(a1, a2, …, an), axis*)
  - Join a sequence of arrays along an existing axis
  - *a1, a2, …, an*: sequence of arrays
  - *axis*: the axis along which the arrays will be joined. If axis is None, arrays are flattened before use. (default 0)

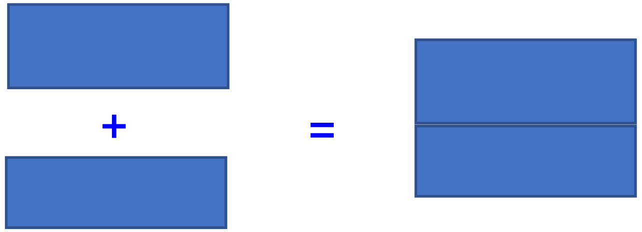
```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6])
```

```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = np.array([[5, 6]])
>>> np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Stacking

- *np*.hstack(*tup*)
  - Stack arrays in sequence horizontally (column wise)
  - *tup*: tuple of arrays

- *np*.vstack(*tup*)
  - Stack arrays in sequence vertically (row wise)
  - *tup*: tuple of arrays




```
>>> a=np.arange(6).reshape(2,3)
>>> b=np.arange(6,12).reshape(2,3)
>>> np.hstack((a,b))
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
>>> np.vstack((a,b))
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

np.r_[ ]

- *np*.r_[*axes*, ...]
  - Stack arrays along their first axis in the string

```
>>> a= np.array([0,1,2])
>>> b= np.array([3,4,5])

>>> np.r_[a, b]
array([0, 1, 2, 3, 4, 5])
>>> np.r_['0', [a], [b]]
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.r_['1', [a], [b]]
array([[0, 1, 2, 4, 5, 6]])
```

axis to stack

force to 2D shape (1, 3) if necessary

```
>>> np.r_['0,2', a, b]
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.r_['1,2', a, b]
array([[0, 1, 2, 3, 4, 5]])
>>> np.r_['1,2,0', a, b]
array([[0, 3],
       [1, 4],
       [2, 5]]), 1, 2, 3, 4, 5]])
# np.r_['1', a.reshape(3,1), b.reshape(3,1)]
```

force to 2D shape (3, 1)

np.c_[ ]

- *np*.c_[...]
  - Short-hand for `np.r_['-1,2,0', ...]`

stack on last axis

```
>>> np.c_[[0,1,2], [3,4,5]]
array([[0, 3],
       [1, 4],
       [2, 5]])

>>> np.c_[[[0,1,2]], [[3,4,5]]]
array([[0, 1, 2, 3, 4, 5]])
```

```
# Already in 3D
# Just stack on last axis

>>> np.c_[[[[0,1,2]]], [[[3,4,5]]]]
array([[[0, 1, 2, 3, 4, 5]]])
```

Summary: For 2D Arrays

- Stacking horizontally

```
>>> a = np.arange(6).reshape(2,3)
>>> b = np.arange(10,16).reshape(2,3)

>>> np.concatenate((a, b), axis=1)

>>> np.hstack((a, b))

array([[ 0,  1,  2, 10, 11, 12],
       [ 3,  4,  5, 13, 14, 15]])
>>> np.c_[a, b]

>>> np.r_['1', a, b]
```

- Stacking vertically

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [10, 11, 12],
       [13, 14, 15]])

>>> np.concatenate((a, b))

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

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

>>> np.r_[a, b]

>>> np.r_['0', a, b]
```

Tiling

- *np*.`tile`(*A*, *reps*)
  - Construct an array by repeating *A* the number of times given by *reps*

```
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (3,1,2))
array([[[0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2]]])
```

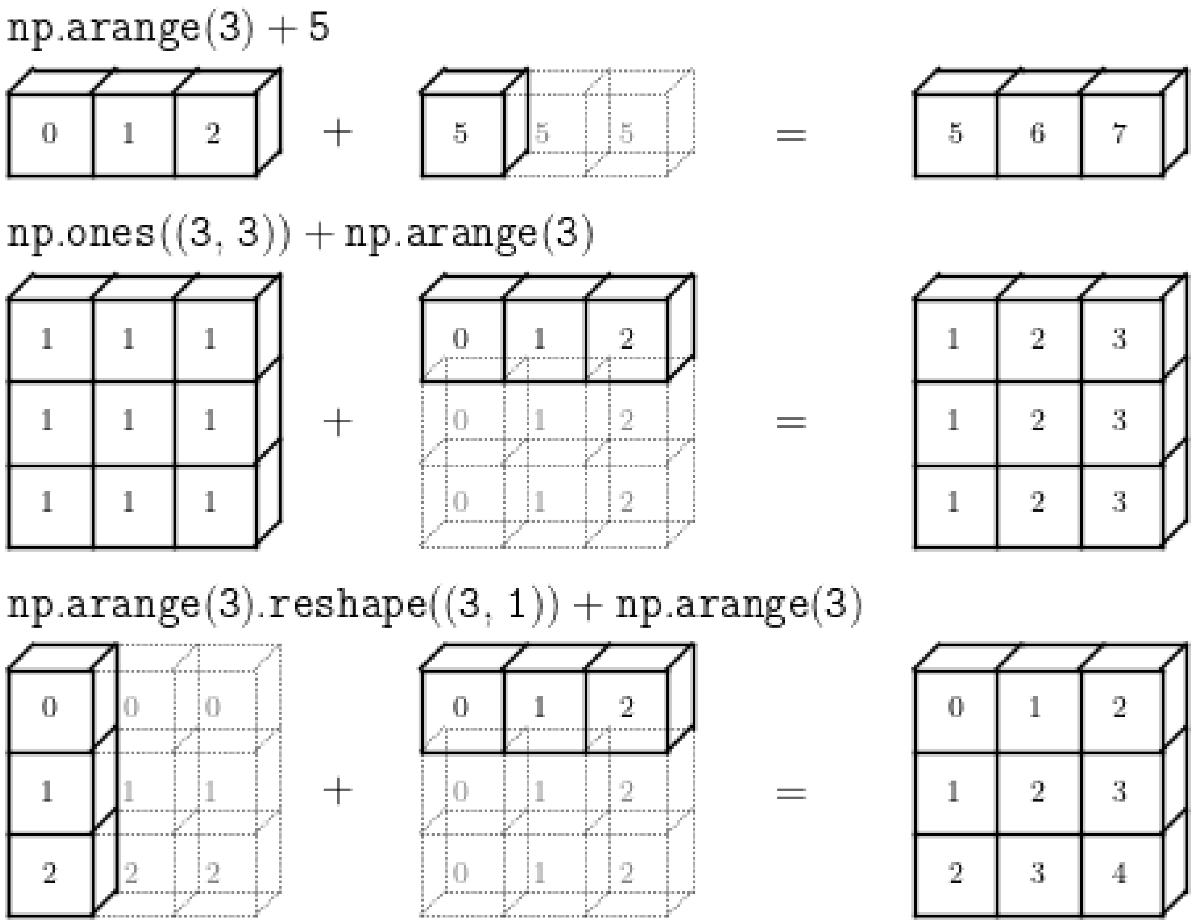
```
>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> np.tile(b, (2, 1))
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

Array Broadcasting

Broadcasting

- Allows arithmetic operations on arrays with different shapes

- The smaller array is "broadcast" across the larger array so that they have compatible shapes



http://astroml.org/book_figures/appendix/fig_broadcast_visual.html

Broadcasting Rule

- The size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one

Image	(3d array)	256 x	256 x	3
Scale	(1d array)			3
Result	(3d array)	256 x	256 x	3

A	(4d array)	8 x	1 x	6 x	1
B	(3d array)		7 x	1 x	5
Result	(4d array)	8 x	7 x	6 x	5

Broadcasting Example (1)

```
>>> a = np.array([1, 2, 3])
>>> b = 2
>>> a * b
array([2, 4, 6])
>>> a = np.array([[ 0.0,  0.0,  0.0],
...                [10.0, 10.0, 10.0],
...                [20.0, 20.0, 20.0],
...                [30.0, 30.0, 30.0]])
>>> b = array([1.0, 2.0, 3.0])
>>> a + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```

```
a:              3
b:              1
result:         3

a:      4   x   3
b:              3
result: 4   x   3
```

Broadcasting Example (2)

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
>>> a = np.arange(12).reshape(4, 3)
>>> b = np.array([1, 2, 3, 4])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast
together with shapes (4,3) (4,)
```

Increase a dimension:
(4,) → (4, 1)

```
a:      4  x  1
b:            3
result: 4  x  3
```

```
a:      4  x  3
b:            4
result:
```