Numpy tutorial

Nicolas P. Rougier

DOI 10.5281/zenodo.28817

Table of Contents

Introduction
The Game of Life
Exercises
Beyond this tutorial
Quick references

Sources are available from github.

Tutorial can be read at
http://www.labri.fr/perso/nrougier/teaching/numpy/numpy.html

See also:

Matplotlib tutorial
100 Numpy exercices

Introduction

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

→ a powerful N-dimensional array object
→ sophisticated (broadcasting) functions
→ tools for integrating C/C++ and Fortran code
→ useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined and this allows NumPy to seamlessly and speedily integrate with a wide variety of projects. We are going to explore numpy through a simple example, implementing the Game of Life.

The Game of Life

Numpy is slanted toward scientific computing and we'll consider in this section the game of life by John Conway which is one of the earliest example of cellular automata (see figure below). Those cellular automaton can be conveniently considered as array of cells that are connected together through the notion of neighbours. We'll show in the following sections implementation of this game using pure python and numpy in order to illustrate main differences with python and numpy.

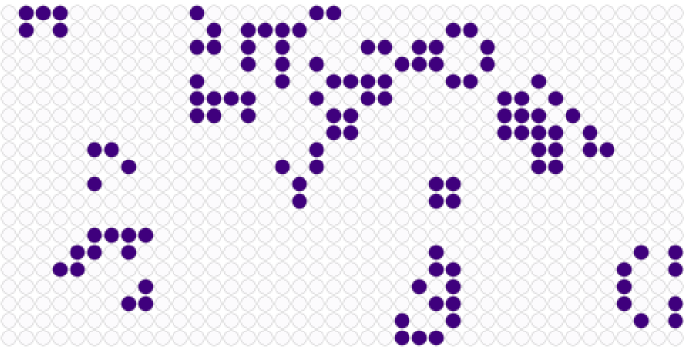


Figure 1 Simulation of the game of life.

Note
This is an excerpt from wikipedia entry on Cellular Automaton.

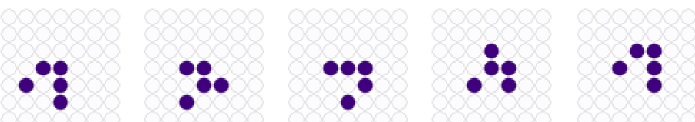
The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The "game" is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by needs caused by underpopulation.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbours becomes a live cell.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed – births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

We'll first use a very simple setup and more precisely, we'll use the glider pattern that is known to move one step diagonally in 4 iterations as illustrated below:



t = 0 t = 1 t = 2 t = 3 t = 4

This property will help us debug our scripts.

The way of python

Note
We could have used the more efficient python array interface but people may be more familiar with the list object.

In pure python, we can code the Game of Life using a list of lists representing the board where cells are supposed to evolve:

```
>>> Z = [[0,0,0,0,0,0],
         [0,0,0,1,0,0],
         [0,1,0,1,0,0],
         [0,0,1,1,0,0],
         [0,0,0,0,0,0],
         [0,0,0,0,0,0]]
```

This board possesses a `0` border that allows to accelerate things a bit by avoiding to have specific tests for borders when counting the number of neighbours. First step is to count neighbours:

```
def compute_neigbours(Z):
    shape = len(Z), len(Z[0])
    N  = [[0,]*(shape[0])  for i in range(shape[1])]
    for x in range(1,shape[0]-1):
        for y in range(1,shape[1]-1):
            N[x][y] = Z[x-1][y-1]+Z[x][y-1]+Z[x+1][y-1] \
                    + Z[x-1][y]            +Z[x+1][y]   \
                    + Z[x-1][y+1]+Z[x][y+1]+Z[x+1][y+1]
    return N
```

To iterate one step in time, we then simply count the number of neighbours for each internal cell and we update the whole board according to the 4 rules:

```
def iterate(Z):
    N = compute_neighbours(Z)
    for x in range(1,shape[0]-1):
        for y in range(1,shape[1]-1):
             if Z[x][y] == 1 and (N[x][y] < 2 or N[x][y] > 3):
                 Z[x][y] = 0
             elif Z[x][y] == 0 and N[x][y] == 3:
                 Z[x][y] = 1
    return Z
```

Note
The `show` command is supplied witht he script.

Using a dedicated display function, we can check the program's correct:

```
>>> show(Z)
[0, 0, 1, 0]
[1, 0, 1, 0]
[0, 1, 1, 0]
[0, 0, 0, 0]

>>> for i in range(4): iterate(Z)
>>> show(Z)
[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 1]
[0, 0, 1, 1]
```

You can download the full script here: game-of-life-python.py

The way of numpy

Note
There exists many more different ways to create a numpy array.

The first thing to do is to create the proper numpy array to hold the cells. This can be done very easily with:

```
>>> import numpy as np
>>> Z = np.array([[0,0,0,0,0,0],
                  [0,0,0,1,0,0],
                  [0,1,0,1,0,0],
                  [0,0,1,1,0,0],
                  [0,0,0,0,0,0],
                  [0,0,0,0,0,0]])
```

Note
For a complete review on numpy data types, check the documentation.

Note that we did not specify the data type of the array and thus, numpy will choose one for us. Since all elements are integers, numpy will then choose an integer data type. This can be easily checked using:

```
>>> print Z.dtype
int64
```

We can also check the shape of the array to make sure it is 6x6:

```
>>> print Z.shape
(6, 6)
```

Each element of `Z` can be accessed using a `row` and a `column` index (in that order):

```
>>> print Z[0,5]
0
```

Note
This element access is actually called indexing and this is very powerful tool for vectorized computation.

But even better, we can also access a subpart of the array using the slice notation:

```
>>> print Z[1:5,1:5]
[[0 0 1 0]
 [1 0 1 0]
 [0 1 1 0]
 [0 0 0 0]]
```

In the example above, we actually extract a subpart of `Z` ranging from rows 1 to 5 and columns 1 to 5. It is important to understand at this point that this is really a subpart of `Z` in the sense that any change to this subpart will have immediate impact on `Z`:

```
>>> A = Z[1:5,1:5]
>>> A[0,0] = 9
>>> print A
[[9 0 1 0]
 [1 0 1 0]
 [0 1 1 0]
 [0 0 0 0]]
```

```
>>> print Z
[[0 0 0 0 0 0]
 [0 9 0 1 0 0]
 [0 1 0 1 0 0]
 [0 0 1 1 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

We set the value of `A[0,0]` to 9 and we see immediate change in `Z[1,1]` because `A[0,0]` actually corresponds to `Z[1,1]`. This may seem trivial with such simple arrays, but things can become much more complex (we'll see that later). If in doubt, you can check easily if an array is part of another one:

```
>>> print Z.base is None
True
>>> print A.base is Z
True
```

Counting neighbours

Note
It is not always possible to vectorize computations and it requires generally some experience. You'll acquire this experience by using numpy (of course) but also by asking questions on the mailing list

We now need a function to count the neighbours. We could do it the same way as for the python version, but this would make things very slow because of the nested loops. We would prefer to act on the whole array whenever possible, this is called vectorization.

Ok, let's start then...

First, you need to know that you can manipulate `Z` as if (and only as if) it was a regular scalar:

```
>>> print 1+(2*Z+3)
[[4 4 4 4 4 4]
 [4 4 4 6 4 4]
 [4 6 4 6 4 4]
 [4 4 6 6 4 4]
 [4 4 4 4 4 4]
 [4 4 4 4 4 4]]
```

If you look carefully at the output, you may realize that the ouptut corresponds to the formula above applied individually to each element. Said differently, we have `(1+(2*Z+3))[i,j] == (1+(2*Z[i,j]+3))` for any i,j.

Ok, so far, so good. Now what happens if we add Z with one of its subpart, let's say `Z[-1:1,-1:1]` ?

```
>>> Z + Z[-1:1,-1:1]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (6,6) (4,4)
```

This raises a `Value Error`, but more interestingly, numpy complains about the impossibility of broadcasting the two arrays together. Broadcasting is a very powerful feature of numpy and most of the time, it saves you a lot of hassle. Let's consider for example the following code:

```
>>> print Z+1
[[1 1 1 1 1 1]
 [1 1 1 2 1 1]
 [1 2 1 2 1 1]
 [1 1 2 2 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
```

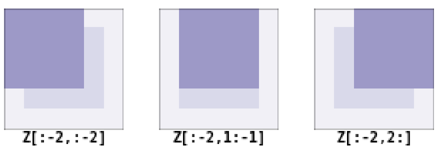
Note
See also the broadcasting section in the numpy documentation.

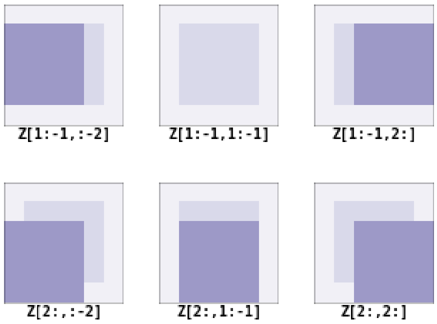
How can a matrix and a scalar be added together ? Well, they can't. But numpy is smart enough to guess that you actually want to add 1 to each of the element of `Z`. This concept of broadcasting is quite powerful and it will take you some time before masterizing it fully (if even possible).

However, in the present case (counting neighbours if you remember), we won't use broadcasting (uh ?). But we'll use vectorize computation using the following code:

```
>>> N = np.zeros(Z.shape, dtype=int)
>>> N[1:-1,1:-1] += (Z[ :-2, :-2] + Z[ :-2,1:-1] + Z[ :-2,2:] +
                     Z[1:-1, :-2]                + Z[1:-1,2:] +
                     Z[2:  , :-2] + Z[2:  ,1:-1] + Z[2:  ,2:])
```

To understand this code, have a look at the figure below:





What we actually did with the above code is to add all the darker blue squares together. Since they have been chosen carefully, the result will be exactly what we expected. If you want to convince yourself, consider a cell in the lighter blue area of the central sub-figure and check what will the result for a given cell.

Iterate

Note
Note the use of the ravel function that flatten an array. This is necessary since the argwhere function returns flattened indices.

In a first approach, we can write the iterate function using the argwhere method that will give us the indices where a given condition is True.

```
def iterate(Z):
    # Iterate the game of life : naive version
    # Count neighbours
    N = np.zeros(Z.shape, int)
    N[1:-1,1:-1] += (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
                     Z[1:-1,0:-2]                + Z[1:-1,2:] +
                     Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])
    N_ = N.ravel()
    Z_ = Z.ravel()

    # Apply rules
    R1 = np.argwhere( (Z_==1) & (N_ < 2) )
    R2 = np.argwhere( (Z_==1) & (N_ > 3) )
    R3 = np.argwhere( (Z_==1) & ((N_==2) | (N_==3)) )
    R4 = np.argwhere( (Z_==0) & (N_==3) )

    # Set new values
    Z_[R1] = 0
    Z_[R2] = 0
    Z_[R3] = Z_[R3]
    Z_[R4] = 1

    # Make sure borders stay null
    Z[0,:] = Z[-1,:] = Z[:,0] = Z[:,-1] = 0
```

Even if this first version does not use nested loops, it is far from optimal because of the use of the 4 argwhere calls that may be quite slow. We can instead take advantages of numpy features the following way.

```
def iterate_2(Z):
    # Count neighbours
    N = (Z[0:-2,0:-2] + Z[0:-2,1:-1] + Z[0:-2,2:] +
         Z[1:-1,0:-2]                + Z[1:-1,2:] +
         Z[2:  ,0:-2] + Z[2:  ,1:-1] + Z[2:  ,2:])

    # Apply rules
    birth = (N==3) & (Z[1:-1,1:-1]==0)
    survive = ((N==2) | (N==3)) & (Z[1:-1,1:-1]==1)
    Z[...] = 0
    Z[1:-1,1:-1][birth | survive] = 1
    return Z
```

If you look at the `birth` and `survive` lines, you'll see that these two variables are indeed arrays. The right-hand side of these two expressions are in fact logical expressions that will result in boolean arrays (just print them to check). We then set all `Z` values to 0 (all cells become dead) and we use the birth and survive arrays to conditionally set Z values to 1. And we're done ! Let's test this:

```
>>> print Z
[[0 0 0 0 0 0]
 [0 0 0 1 0 0]
 [0 1 0 1 0 0]
 [0 0 1 1 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
>>> for i in range(4): iterate_2(Z)
>>> print Z
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 1 0]
 [0 0 1 0 1 0]
 [0 0 0 1 1 0]
 [0 0 0 0 0 0]]
```

You can download the full script here: game-of-life-numpy.py

Getting bigger

While numpy works perfectly with very small arrays, you'll really benefit from numpy power with big to very big arrays. So let us reconsider the game of life with a bigger array. First, we won't initalize the array by hand but initalize it randomly:

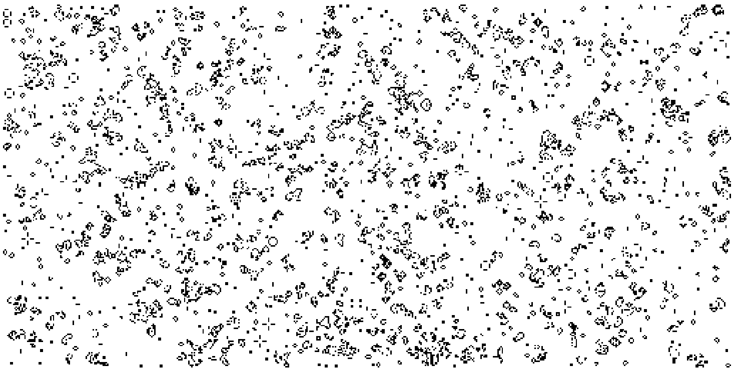
```
>>> Z = np.random.randint(0,2,(256,512))
```

and we simply iterate as previously:

```
>>> for i in range(100): iterate(Z)
```

and display result:

```
>>> size = np.array(Z.shape)
>>> dpi = 72.0
>>> figsize= size[1]/float(dpi),size[0]/float(dpi)
>>> fig = plt.figure(figsize=figsize, dpi=dpi, facecolor="white")
>>> fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=False)
>>> plt.imshow(Z,interpolation='nearest', cmap=plt.cm.gray_r)
>>> plt.xticks([]), plt.yticks([])
>>> plt.show()
```



Easy enough, no ?

A step further

We have reviewed the very basics of numpy so let's move on to more complex (and more fun) things.

Note
Description taken from the Gray-Scott homepage

Reaction and diffusion of chemical species can produce a variety of patterns, reminiscent of those often seen in nature. The Gray Scott equations model such a reaction. For more information on this chemical system see the article Complex Patterns in a Simple System, John E. Pearson, Science, Volume 261, 9 July 1993.

Let's consider two chemical species U and V with respective concentrations u and v and diffusion rates ru and rv. V is converted into P with a rate of conversion k. f represents the rate of the process that feeds U and drains U, V and P. We can now write:

Chemical reaction	Equations
$U + 2V \longrightarrow 3V$	$(\partial u)/(\partial t) = ru\nabla 2u - uv2 + f(1 - u)$
$V \longrightarrow P$	$(\partial v)/(\partial t) = rv\nabla 2v + uv2 - (f + k)v$

Examples
(click figure to see movie)

Obviously, you may think we need two arrays, one for `U` and for `V`. But since `U` and `V` are tighly linked, it may be indeed better to use a single array. Numpy allows to do that with the notion of structured array:

```
>>> n = 200
>>> Z = np.zeros((n+2,n+2), [('U', np.double),
                             ('V', np.double)])
>>> print Z.dtype
[('U', '<f8'), ('V', '<f8')]
```

The size of the array is (n+2,n+2) since we need the borders when computing the neighbours. However, we'll compute differential equation only in the center part, so we can already creates some useful views of this array:

```
>>> U,V = Z['U'], Z['V']
>>> u,v = U[1:-1,1:-1], V[1:-1,1:-1]
```

Next, we need to compute the Laplacian and we'll use a discrete approximation obtained via the finite difference method using the same vectorization as for the Game of Life:

```
def laplacian(Z):
    return (                 Z[0:-2,1:-1] +
            Z[1:-1,0:-2] - 4*Z[1:-1,1:-1] + Z[1:-1,2:] +
                             Z[2:  ,1:-1] )
```

Finally, we can iterate the computation after havong choosed some interesting parameters:

```
for i in xrange(25000):
    Lu = laplacian(U)
    Lv = laplacian(V)
    uvv = u*v*v
    u += (Du*Lu - uvv +  F   *(1-u))
```

```
v += (Dv*Lv + uvv - (F+k)*v    )
```

And we're done !

You can download the full script here: gray-scott.py

Exercises

Here are some exercises, try to do them without looking at the solution (just highligh the blank part to see it).

Neophyte

1. Import the numpy package under the name `np`

2. Print the numpy version and the configuration.

Hint
See np.zeros

3. Create a null vector of size 10

4. Create a null vector of size 10 but the fifth value which is 1

Hint
See np.arange

5. Create a vector with values ranging from 10 to 99

6. Create a 3x3 matrix with values ranging from 0 to 8

Hint
See np.nonzero

7. Find indices of non-zero elements from [1,2,0,0,4,0]

8. Declare a 3x3 identity matrix

9. Declare a 5x5 matrix with values 1,2,3,4 just below the diagonal

Hint
See Random sampling

10. Declare a 10x10x10 array with random values

Novice

1. Declare a 8x8 matrix and fill it with a checkerboard pattern

2. Declare a 10x10 array with random values and find the minimum and maximum values

3. Create a checkerboard 8x8 matrix using the tile function

4. Normalize a 5x5 random matrix (between 0 and 1)

Hint
See the linear algebra documentation

5. Multiply a 5x3 matrix by a 3x2 matrix (real matrix product)

6. Create a 10x10 matrix with row values ranging from 0 to 9

7. Create a vector of size 1000 with values ranging from 0 to 1, both excluded

8. Create a random vector of size 100 and sort it

9. Consider two random matrices A anb B, check if they are equal.

10. Create a random vector of size 1000 and find the mean value

Apprentice

1. Consider a random 100x2 matrix representing cartesian coordinates, convert them to polar coordinates

2. Create random vector of size 100 and replace the maximum value by 0

Hint
See the documentation on Structured arrays

3. Declare a structured array with x and y coordinates covering the [0,1]x[0,1] area.

Hint
Have a look at Data type routines

4. Print the minimum and maximum representable value for each numpy scalar type

5. Create a structured array representing a position (x,y) and a color (r,g,b)

6. Consider a random vector with shape (100,2) representing coordinates, find point by point distances

7. Generate a generic 2D Gaussian-like array

8. Consider the vector [1, 2, 3, 4, 5], how to build a new vector with 3 consecutive zeros interleaved between each value ?

Beyond this tutorial

Numpy benefits from extensive documentation as well as a large community of users and developpers. Here are some links of interest:

Other Tutorials

The SciPy Lecture Notes

The SciPy Lecture notes offers a teaching material on the scientific Python ecosystem as well as quick introduction to central tools and techniques. The different chapters each correspond to a 1 to 2 hours course with increasing level of expertise, from beginner to expert.

A Tentative numpy tutorial

- Prerequisites
- The Basics
- Shape Manipulation
- Copies and Views
- Less Basic
- Fancy indexing and index tricks
- Linear Algebra
- Tricks and Tips

Numpy MedKit

A first-aid kit for the numerically adventurous by Stéfan van der Walt.

An introduction to Numpy and Scipy

A short introduction to Numpy and Scipy by M. Scott Shell.

Numpy documentation

User guide

This guide is intended as an introductory overview of NumPy and explains how to install and make use of the most important features of NumPy.

Numpy reference

This reference manual details functions, modules, and objects included in Numpy, describing what they are and what they do.

FAQ

General questions about numpy
General questions about SciPy
Basic SciPy/numpy usage
Advanced NumPy/SciPy usage
NumPy/SciPy installation
Miscellaneous Issues

Code documentation

The code is fairly well documented and you can quickly access a specific command from within a python session:

```
>>> import numpy as np
>>> help(np.ones)
Help on function ones in module numpy.core.numeric:

ones(shape, dtype=None, order='C')
    Return a new array of given shape and type, filled with ones.

    Please refer to the documentation for `zeros` for further details.

    See Also
    --------
    zeros, ones_like

    Examples
    --------
    >>> np.ones(5)
    array([ 1.,  1.,  1.,  1.,  1.])

    >>> np.ones((5,), dtype=np.int)
    array([1, 1, 1, 1, 1])

    >>> np.ones((2, 1))
    array([[ 1.],
           [ 1.]])

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

Mailing lists

Finally, there is a mailing list where you can ask for help.

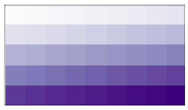
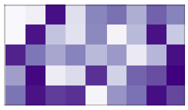
Quick references

Data type

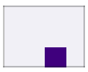
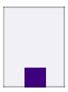
Data type	Description
bool	Boolean (True or False) stored as a byte
int	Platform integer (normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float	Shorthand for float64.

float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

Creation

Code	Result	Code	Result
`Z = zeros(9)`		`Z = zeros((5,9))`	
`Z = ones(9)`		`Z = ones((5,9))`	
`Z = array( [0,0,0,0,0,0,0,0,0] )`		`Z = array( [[0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0]])`	
`Z = arange(9)`		`Z = arange(5*9).reshape(5,9)`	
`Z = random.uniform(0,1,9)`		`Z = random.uniform(0,1,(5,9))`	

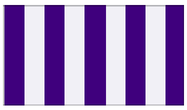
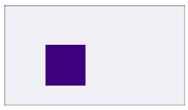
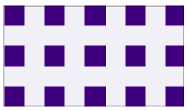
Reshaping

Code	Result	Code	Result
`Z[2,2] = 1`		`Z = Z.reshape(1,12)`	
`Z = Z.reshape(4,3)`			
		`Z = Z.reshape(12,1)`	

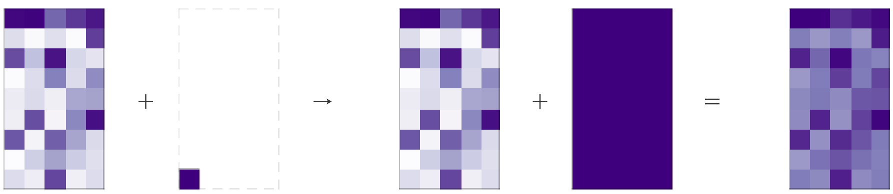
```
Z = Z.reshape(6,2)
```

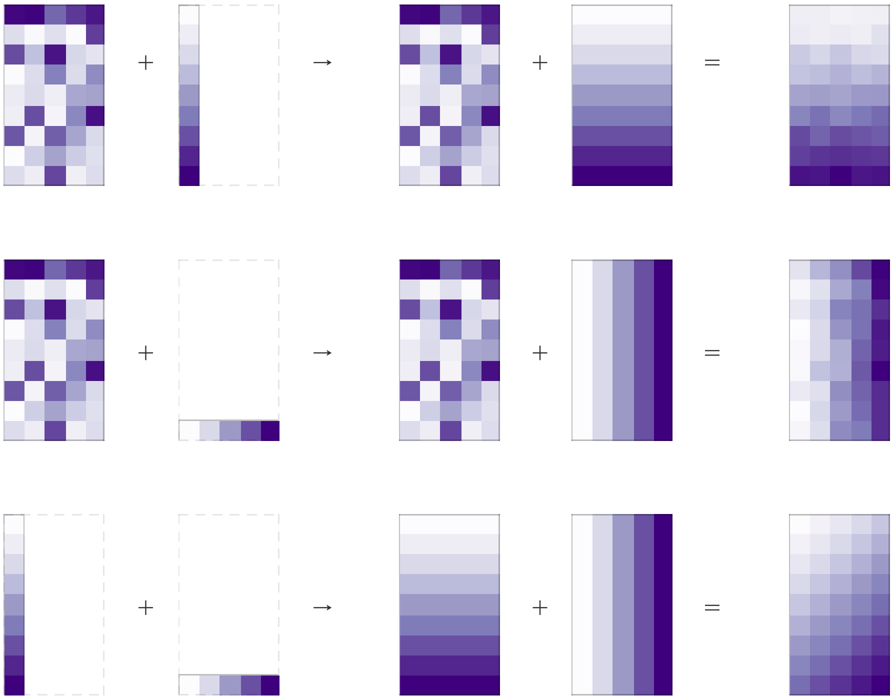
```
Z = Z.reshape(2,6)
```

Slicing

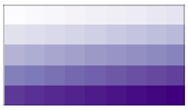
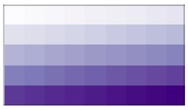
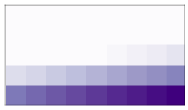
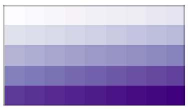
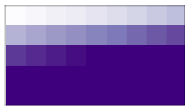
Code	Result	Code	Result
`Z`		`Z[...] = 1`	
`Z[1,1] = 1`		`Z[:,0] = 1`	
`Z[0,:] = 1`		`Z[2:,2:] = 1`	
`Z[:,::2] = 1`		`Z[::2,:] = 1`	
`Z[:-2,:-2] = 1`		`Z[2:4,2:4] = 1`	
`Z[::2,::2] = 1`		`Z[3::2,3::2] = 1`	

Broadcasting





Operations

Code	Before	After
`Z = np.where(Z > 0.5, 0, 1)`		
`Z = np.maximum(Z, 0.5)`		
`Z = np.minimum(Z, 0.5)`		
`Z = np.sum(Z, axis=0)`