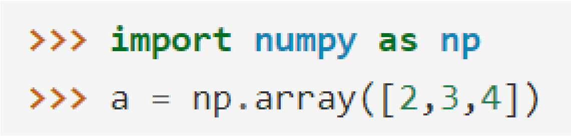
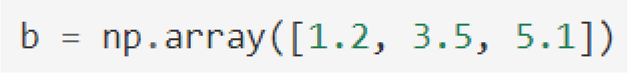
Numpy Quicker Start - Extracted from:

https://docs.scipy.org/doc/numpy/user/quickstart.html

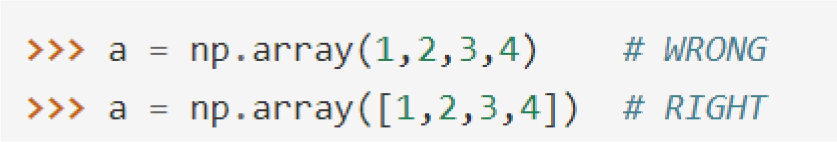
Array Creation



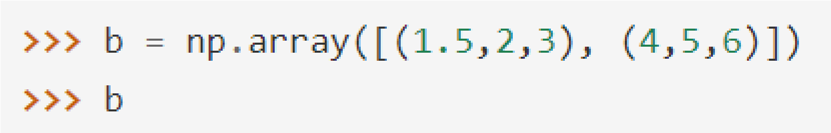
```
>>> import numpy as np
>>> a = np.array([2,3,4])
```



```
b = np.array([1.2, 3.5, 5.1])
```

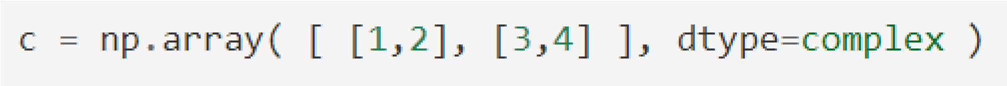


```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```



```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
```

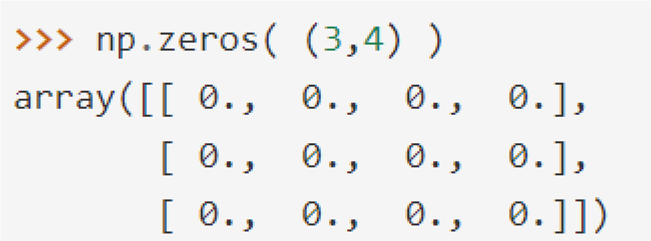
Notice ( )



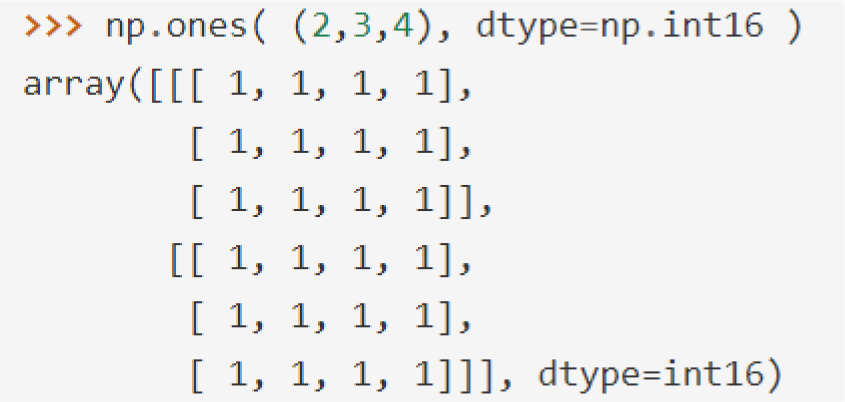
```
c = np.array( [ [1,2], [3,4] ], dtype=complex )
```

Notice [ ]

Notice dtype



```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```



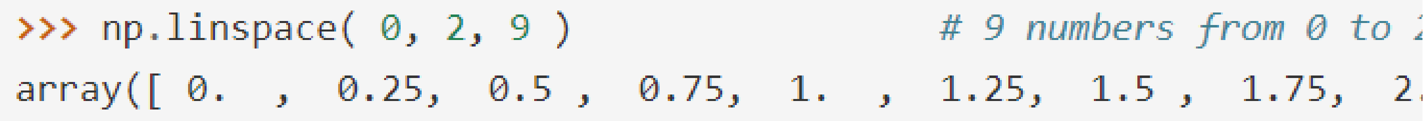
```
>>> np.ones( (2,3,4), dtype=np.int16 )
array([[[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]],
       [[ 1, 1, 1, 1],
        [ 1, 1, 1, 1],
        [ 1, 1, 1, 1]]], dtype=int16)
```

Notice dtype

dtype=np.float32

dtype=np.float64

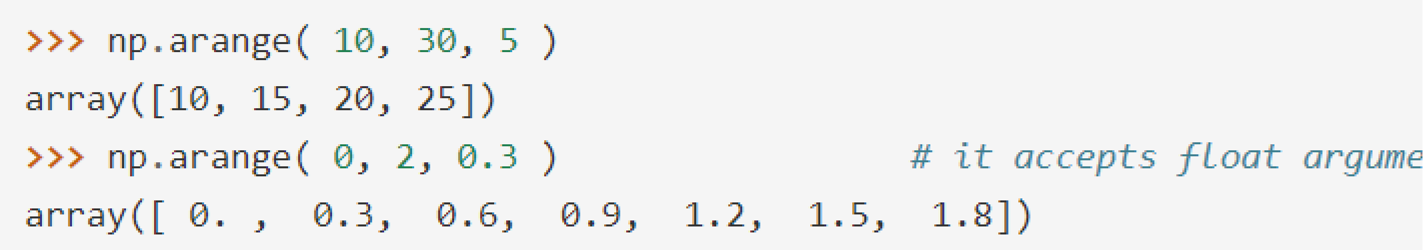
dtype=float



```
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2
```

A numpy function similar to range( ) that can use ***floats***

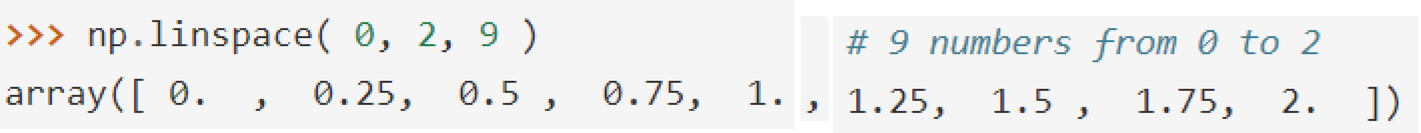
a - range .... not .... arrange!



```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float argume
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

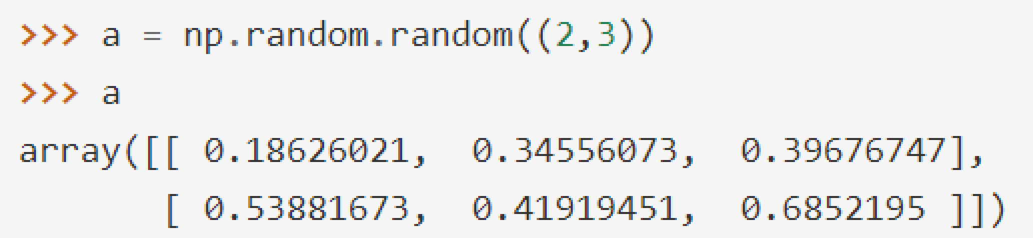
0 to 2 exclusive ... like range

A similar but different function: start, stop (inclusive) and ***count***



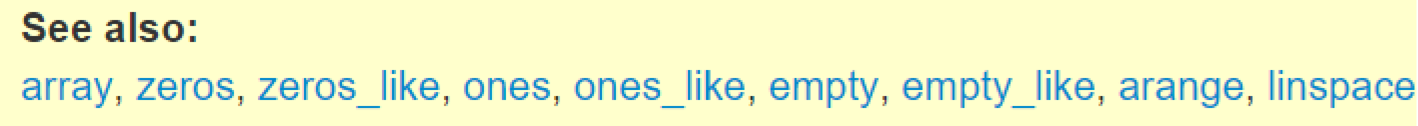
```
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

0 to 2 inclusive!!!!



```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
```

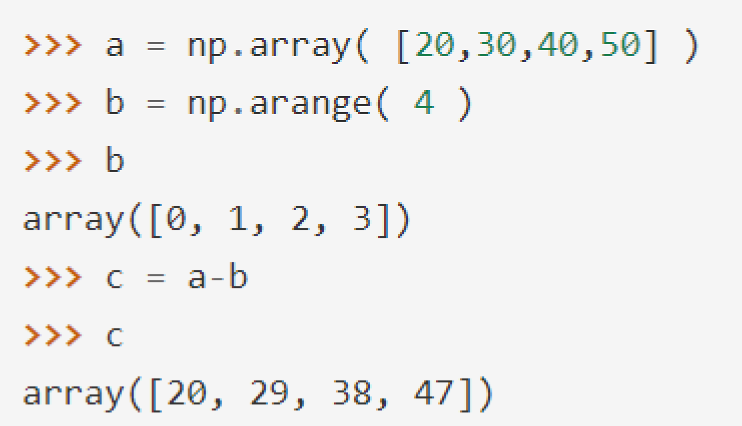
a random number generator



See also:

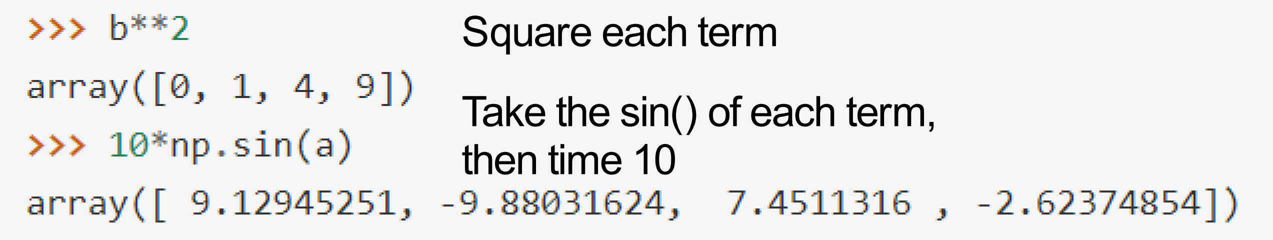
array, zeros, zeros_like, ones, ones_like, empty, empty_like, arange, linspace

Basic Operations



```
>>> 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])
```

element-wise addition and subtraction

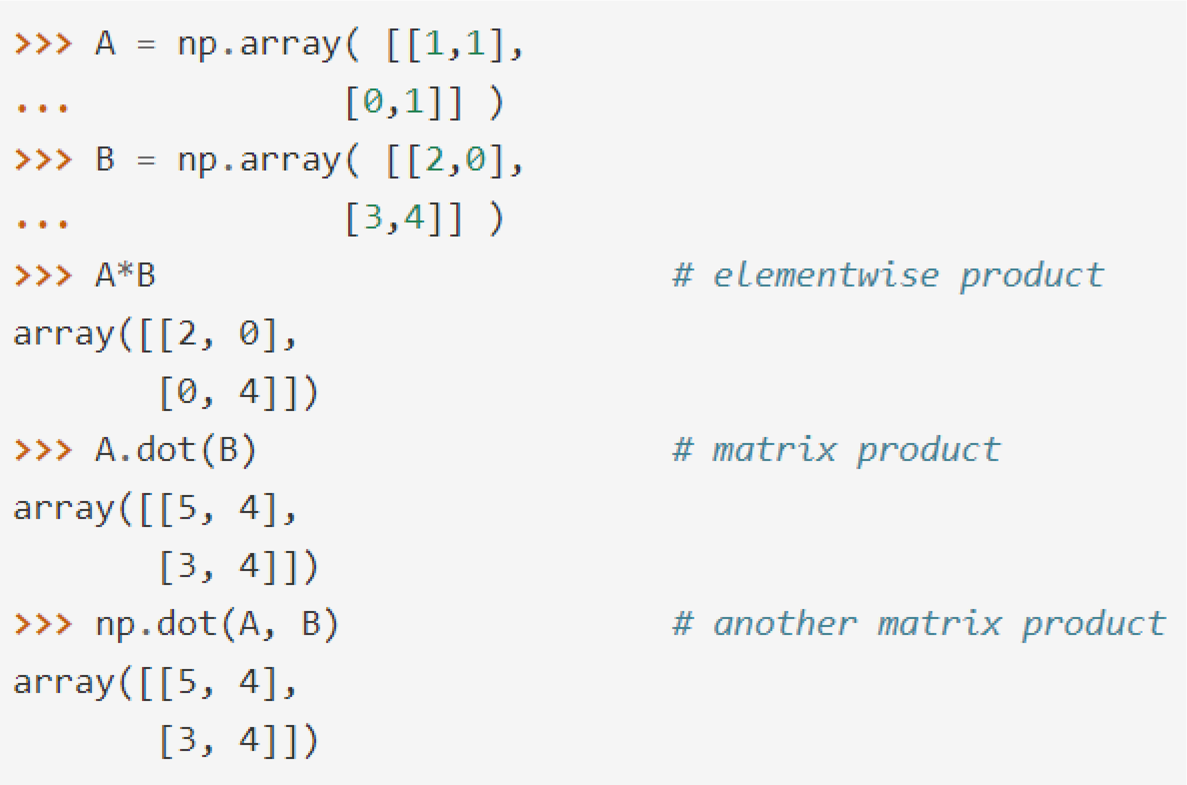


```
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

Square each term

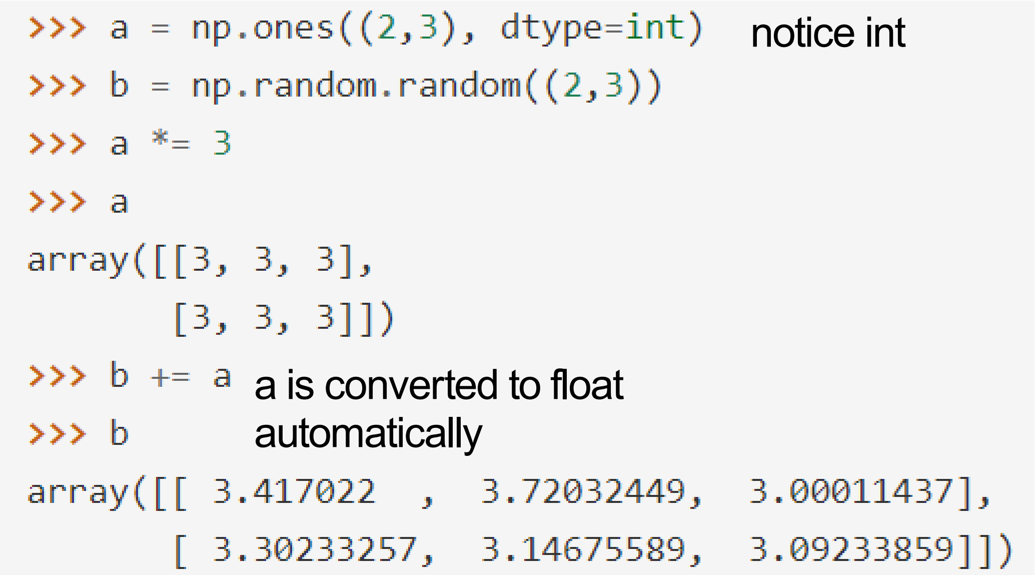
Take the sin() of each term,
then time 10

Term-wise and True Matrix Multiplication



```
>>> A = np.array( [[1,1],
...             [0,1]] )
>>> B = np.array( [[2,0],
...             [3,4]] )
>>> A*B                         # elementwise product
array([[2, 0],
       [0, 4]])
>>> A.dot(B)                    # matrix product
array([[5, 4],
       [3, 4]])
>>> np.dot(A, B)                # another matrix product
array([[5, 4],
       [3, 4]])
```

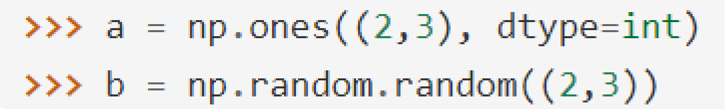
+=   -=   *=



```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
```

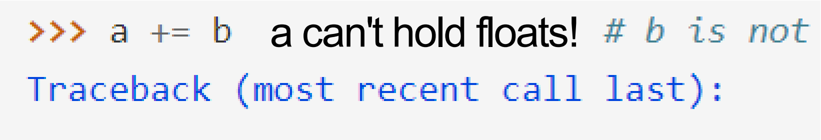
notice int

a is converted to float automatically



```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
```

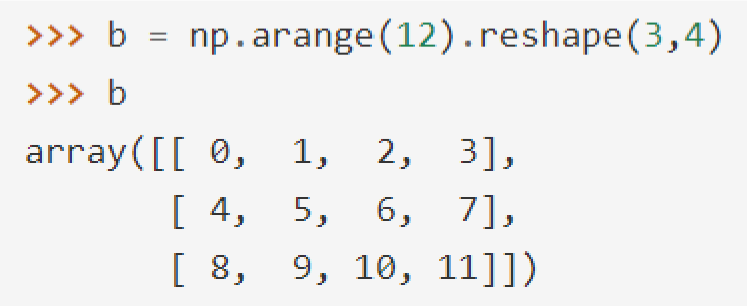
a is an array of integers
b is an array of floats



```
>>> a += b                  # b is not
Traceback (most recent call last):
```

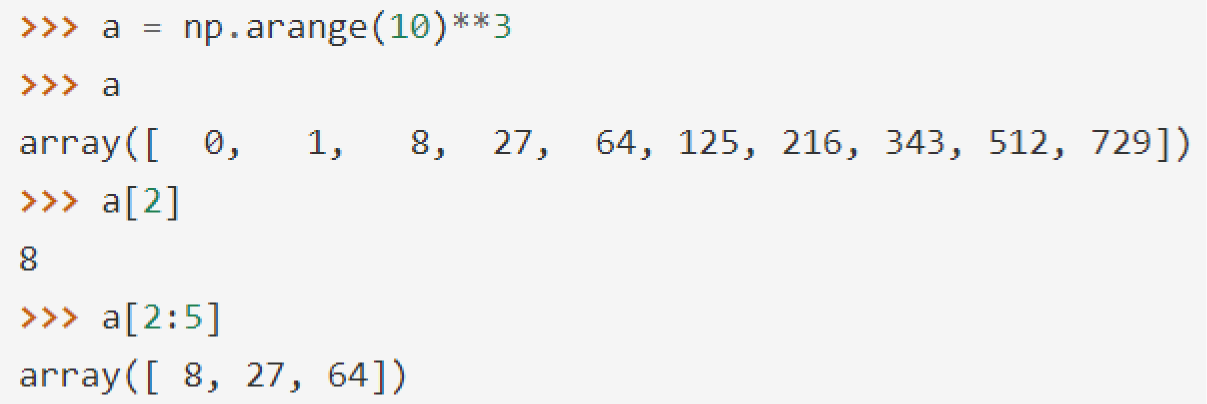
a can't hold floats!

The reshape method

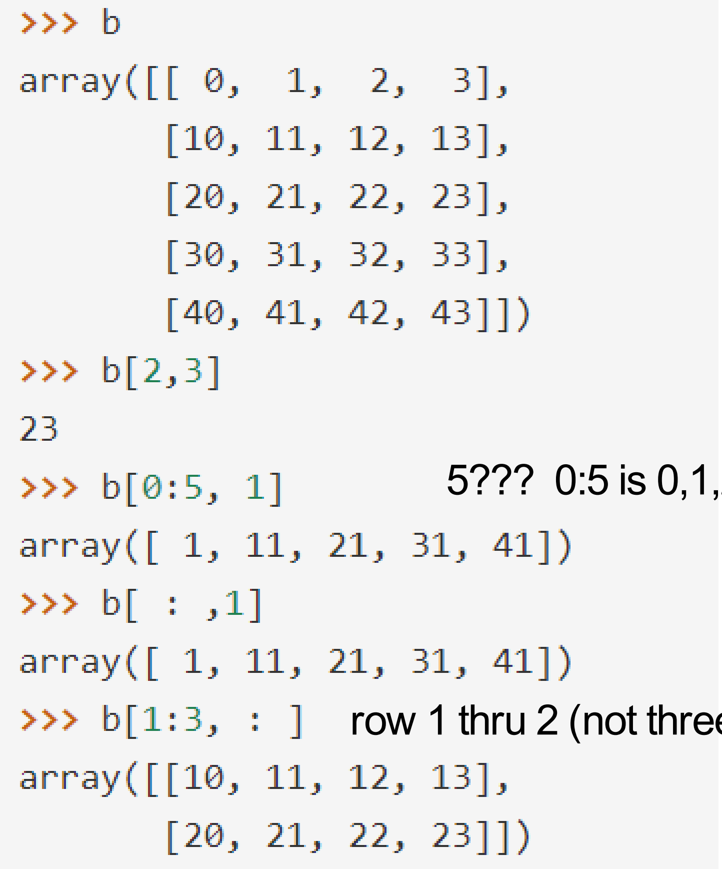


```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Indexing, Slicing and Iterating



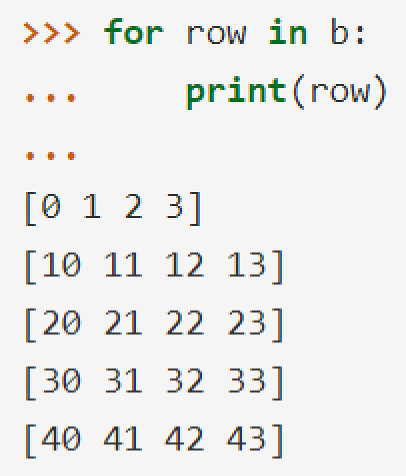
```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
```



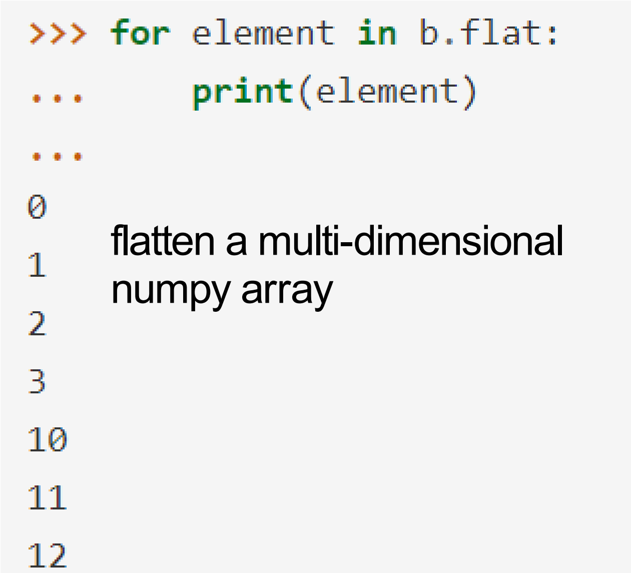
```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
>>> b[0:5, 1]
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

5??? 0:5 is 0,1,2,3,4

row 1 thru 2 (not three) and all columns

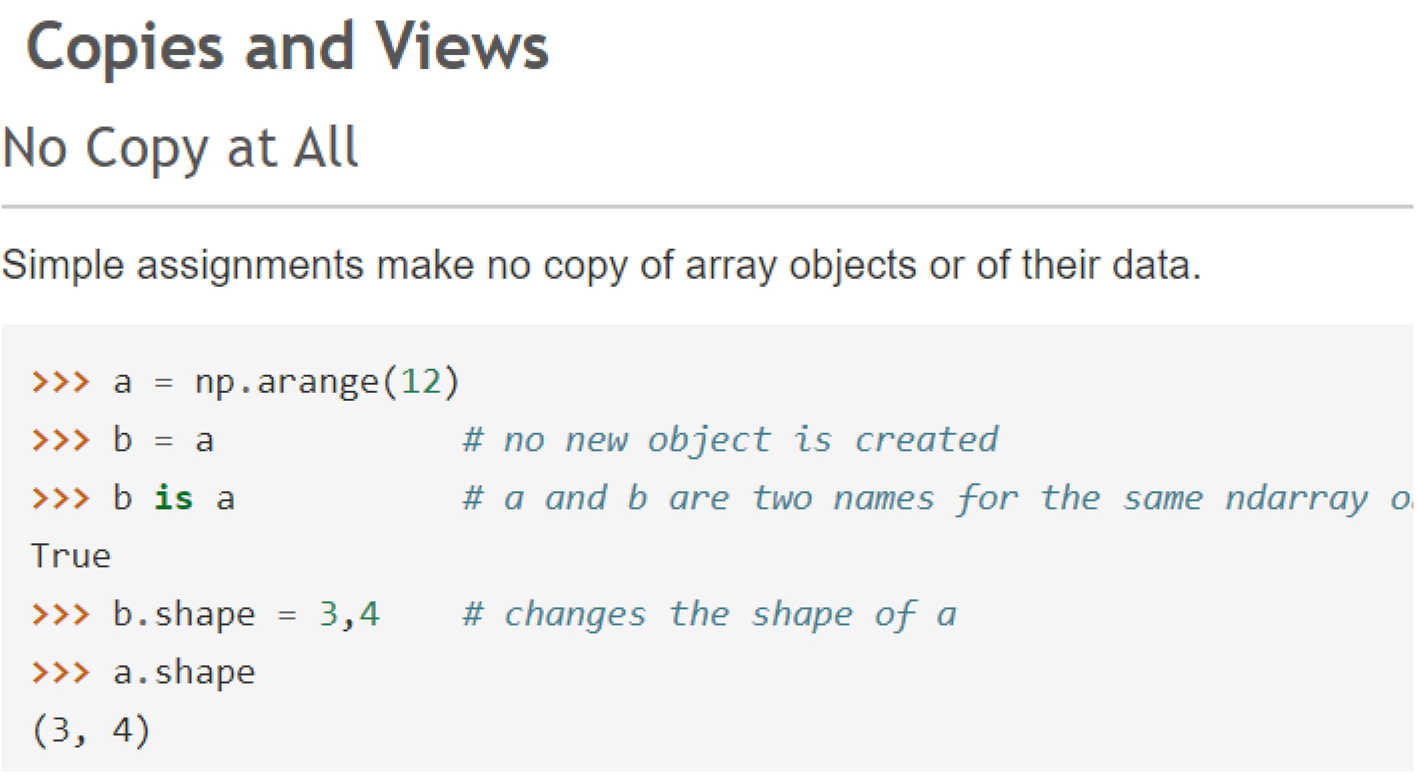


```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```



```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
```

flatten a multi-dimensional numpy array

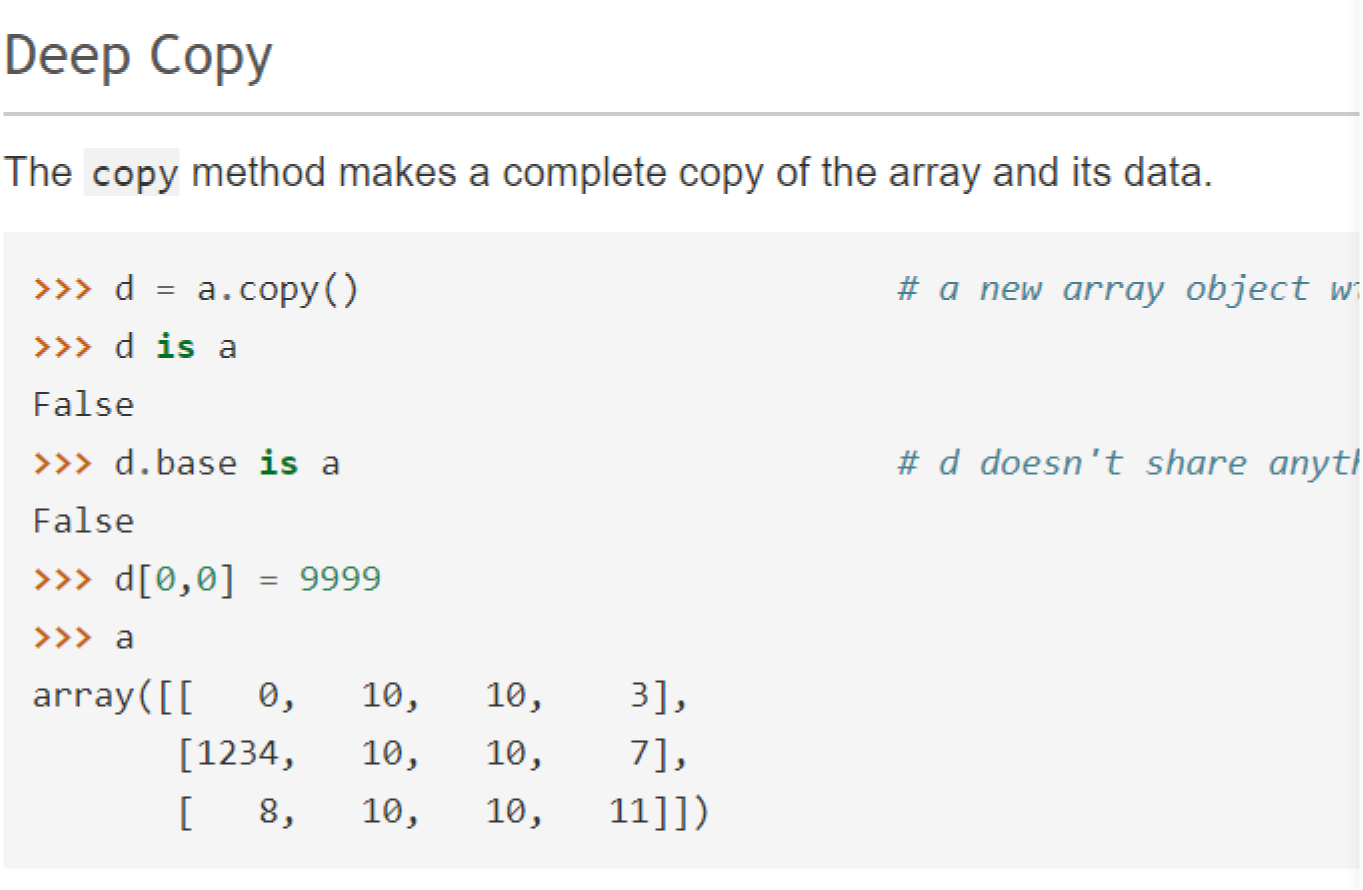


Copies and Views

No Copy at All

Simple assignments make no copy of array objects or of their data.

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray o
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

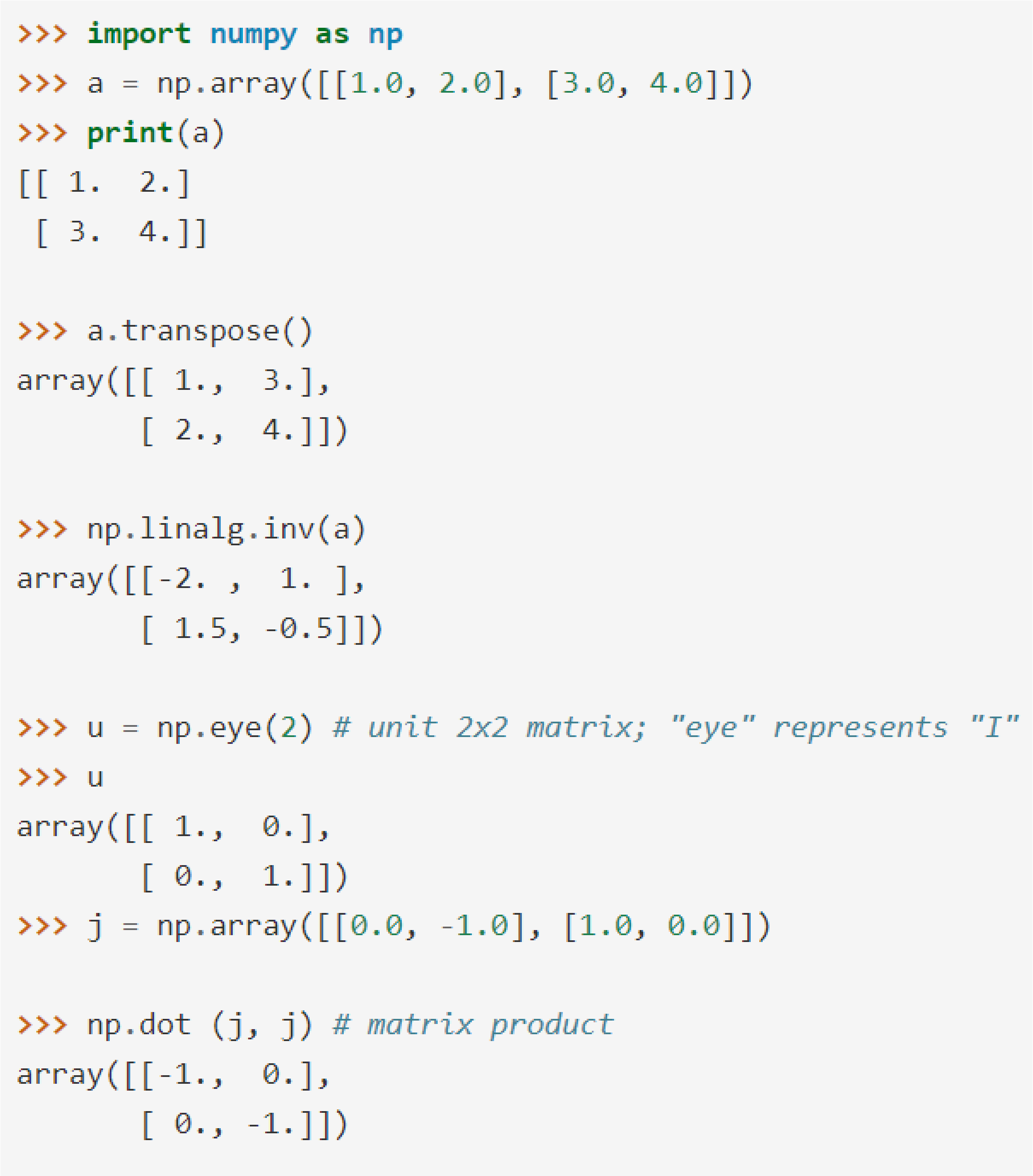


Deep Copy

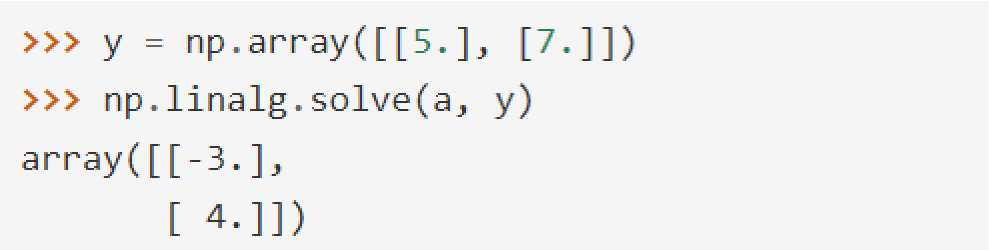
The `copy` method makes a complete copy of the array and its data.

```
>>> d = a.copy()                          # a new array object w
>>> d is a
False
>>> d.base is a                           # d doesn't share anyth
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

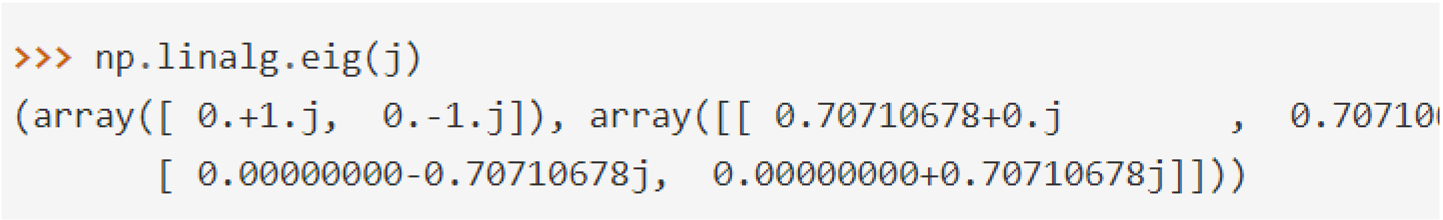
Linear Algebra



```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> np.dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])
```

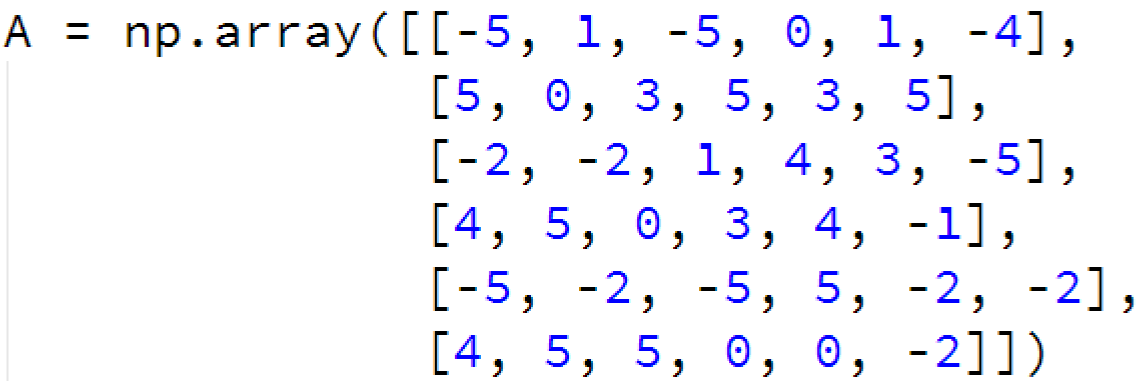


```
>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(a, y)
array([[-3.],
       [ 4.]])
```

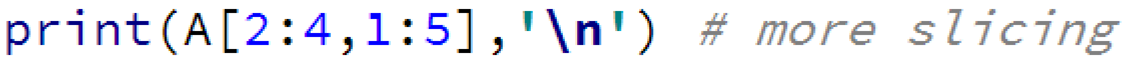


```
>>> np.linalg.eig(j)
(array([ 0.+1.j,  0.-1.j]), array([[ 0.70710678+0.j        ,  0.70710
       [ 0.00000000-0.70710678j,  0.00000000+0.70710678j]]))
```

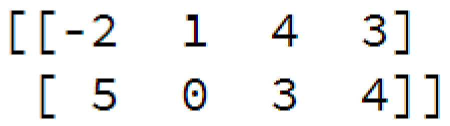
More Slicing



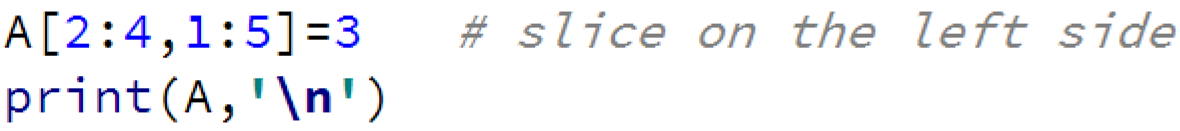
```
A = np.array([[-5, 1, -5, 0, 1, -4],
              [5, 0, 3, 5, 3, 5],
              [-2, -2, 1, 4, 3, -5],
              [4, 5, 0, 3, 4, -1],
              [-5, -2, -5, 5, -2, -2],
              [4, 5, 5, 0, 0, -2]])
```



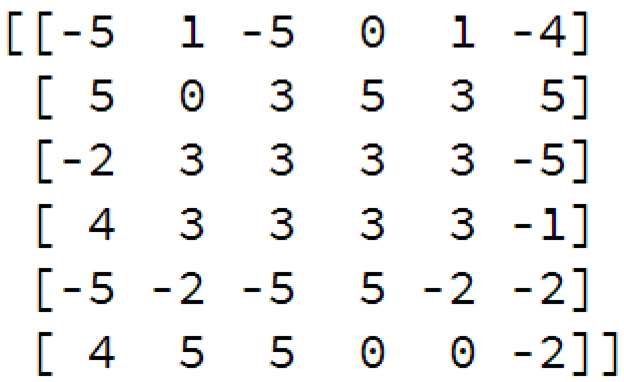
```
print(A[2:4,1:5],'\n') # more slicing
```



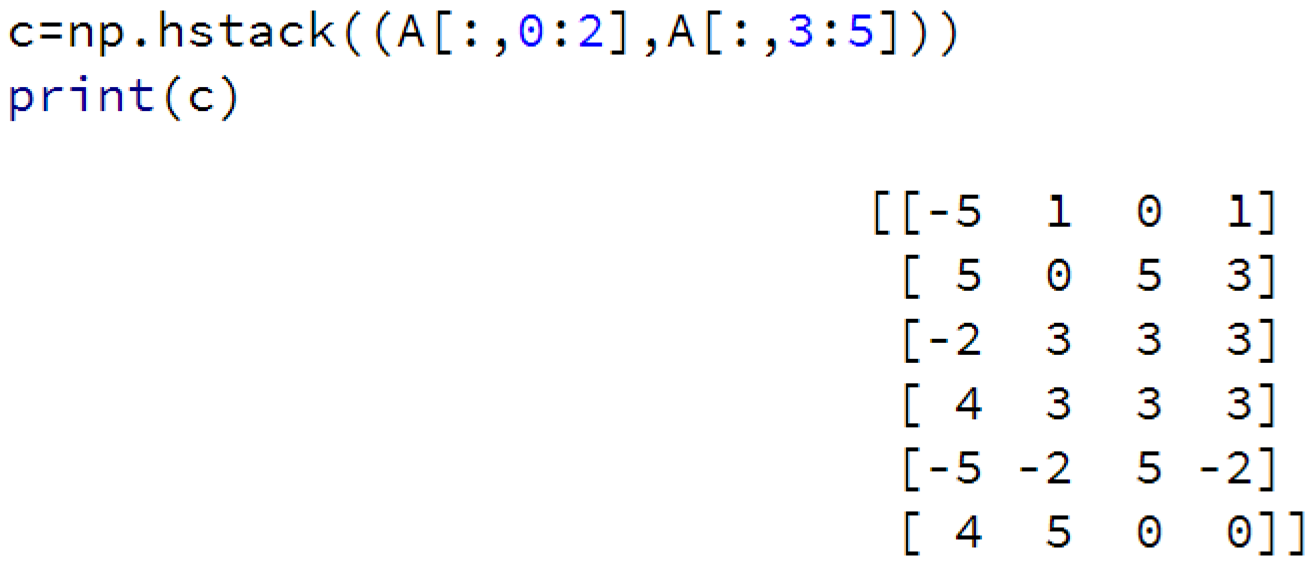
```
[[-2  1  4  3]
 [ 5  0  3  4]]
```



```
A[2:4,1:5]=3    # slice on the left side
print(A,'\n')
```



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



```
c=np.hstack((A[:,0:2],A[:,3:5]))
print(c)
```

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