NUMPY

ARIHARASUDHAN

arihara-sudhan.github.io

NUMPY

NumPy is the fundamental package for scientific computing in Python. It provides a multidimensional array object, various derived objects and a set of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

NDARRAY

The ndarray object is at the core of the NumPy Package. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

Why NumPy is Fast ?

Vectorization describes the absence of any explicit looping, indexing, etc., in the code - these things are taking place, of course, just “behind the scenes” in optimized, pre-compiled C code.

AXES

NumPy's main object is the homogeneous multidimensional array. In NumPy, dimensions are called *axes*. For example, the array for the coordinates of a point in 3D space, [1, 2, 1], has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[  [0, 3, 4],
   [5, 6, 7]  ]
```

NDARRAY ATTRIBUTES

NumPy's array class is called ndarray. It is also known by the alias array.

ndarray.ndim

the number of axes (dimensions) of the array.

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 number of axes, ndim.

ndarray.size

the total number of elements of the array. This is equal to the product of the elements of shape.

ndarray.dtype

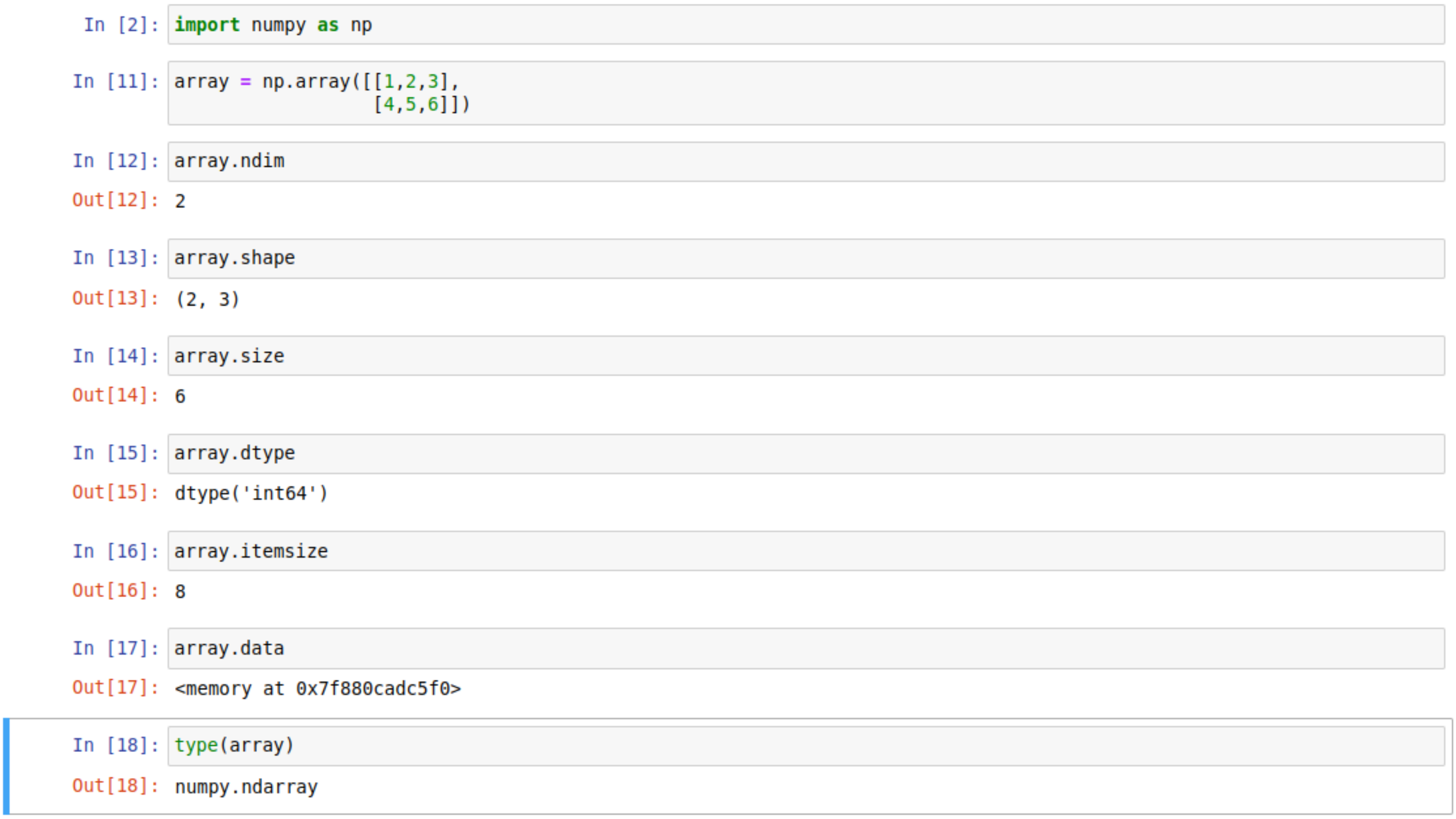
an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

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

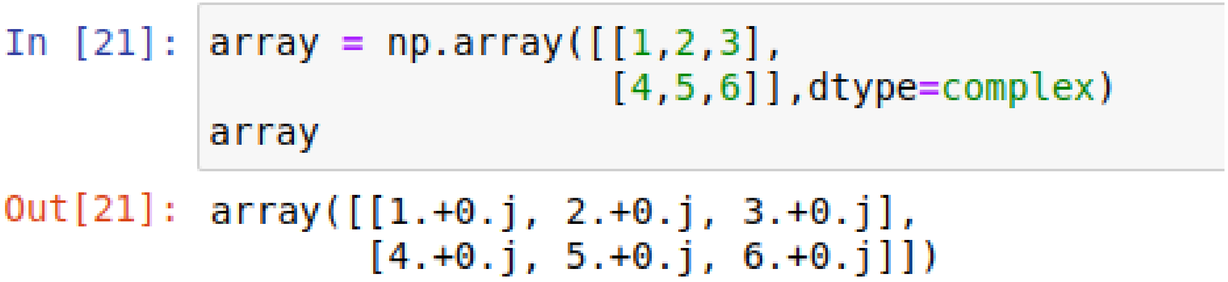
the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.



```
In [2]: import numpy as np

In [11]: array = np.array([[1,2,3],
                           [4,5,6]])

In [12]: array.ndim
Out[12]: 2

In [13]: array.shape
Out[13]: (2, 3)

In [14]: array.size
Out[14]: 6

In [15]: array.dtype
Out[15]: dtype('int64')

In [16]: array.itemsize
Out[16]: 8

In [17]: array.data
Out[17]: <memory at 0x7f880cadc5f0>

In [18]: type(array)
Out[18]: numpy.ndarray
```

SPECIFYING TYPE

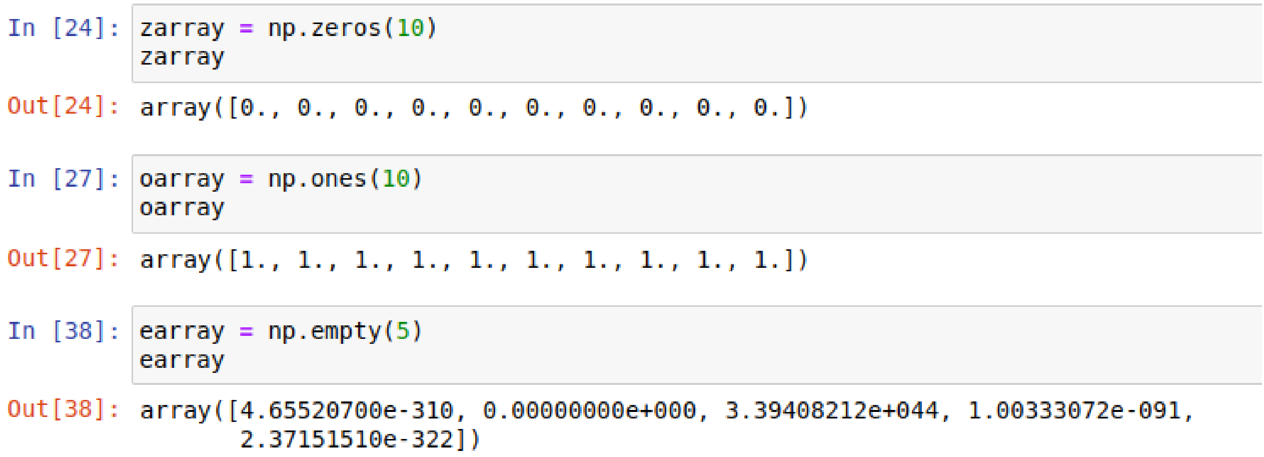
The Type can be specified explicitly.



```
In [21]: array = np.array([[1,2,3],
                           [4,5,6]],dtype=complex)
         array
Out[21]: array([[1.+0.j, 2.+0.j, 3.+0.j],
                [4.+0.j, 5.+0.j, 6.+0.j]])
```

ZEROS , ONES & EMPTY

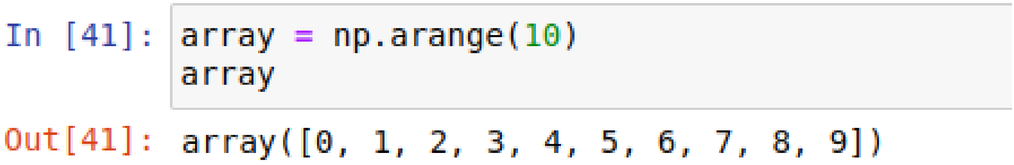
We can fill Zeros and Ones multiple times. The empty() creates an ndarray of random values.



```
In [24]:  zarray = np.zeros(10)
          zarray

Out[24]:  array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [27]:  oarray = np.ones(10)
          oarray

Out[27]:  array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

In [38]:  earray = np.empty(5)
          earray

Out[38]:  array([4.65520700e-310, 0.00000000e+000, 3.39408212e+044, 1.00333072e-091,
                 2.37151510e-322])
```

The arange( ) Method

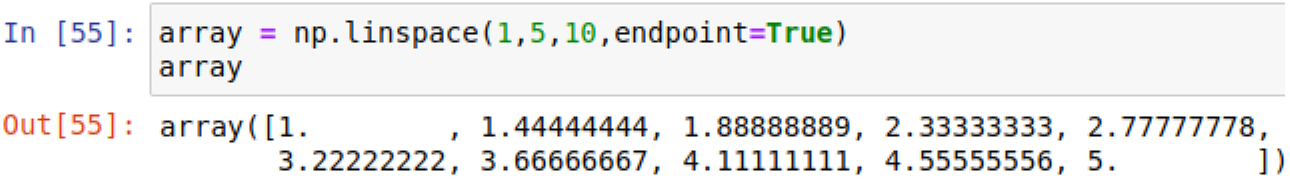
To create sequences of numbers, NumPy provides the `arange` function.



```
In [41]:  array = np.arange(10)
          array

Out[41]:  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The linspace( ) Method

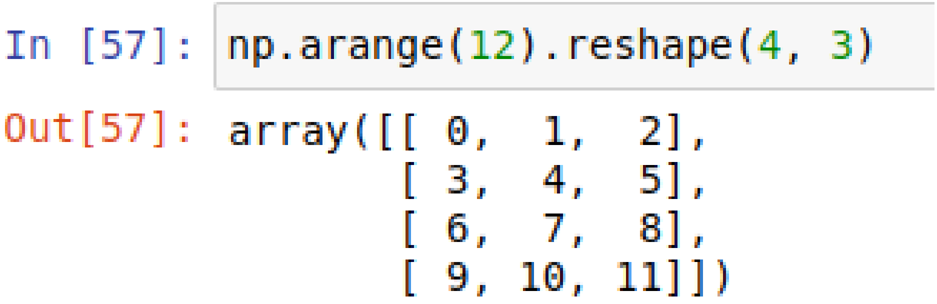
It is similar to the arrange function. However, it allows to specify the step size. It returns evenly separated values over the specified period.



```
In [55]: array = np.linspace(1,5,10,endpoint=True)
         array

Out[55]: array([1.        , 1.44444444, 1.88888889, 2.33333333, 2.77777778,
                3.22222222, 3.66666667, 4.11111111, 4.55555556, 5.        ])
```

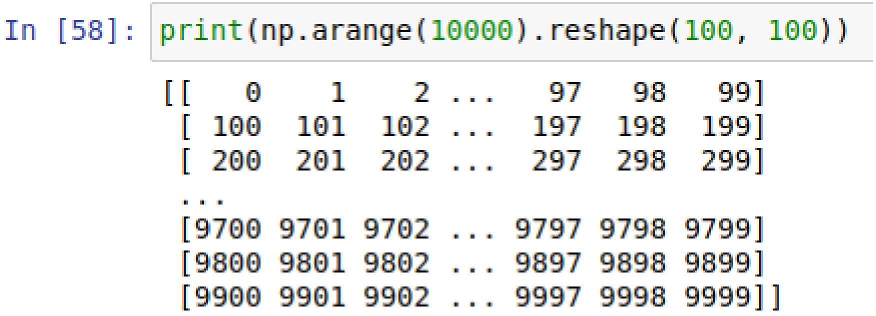
The reshape( ) Method

We can alter the dimensions of an ndarray.



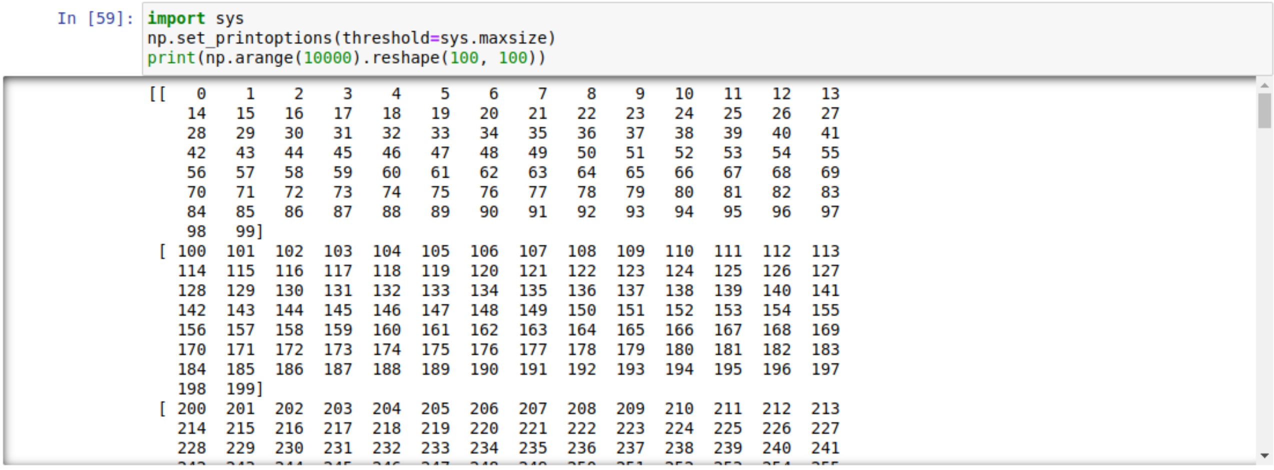
```
In [57]: np.arange(12).reshape(4, 3)

Out[57]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])
```

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners.



```
In [58]: print(np.arange(10000).reshape(100, 100))

         [[   0    1    2 ...   97   98   99]
          [ 100  101  102 ...  197  198  199]
          [ 200  201  202 ...  297  298  299]
          ...
          [9700 9701 9702 ... 9797 9798 9799]
          [9800 9801 9802 ... 9897 9898 9899]
          [9900 9901 9902 ... 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using `set_printoptions`.

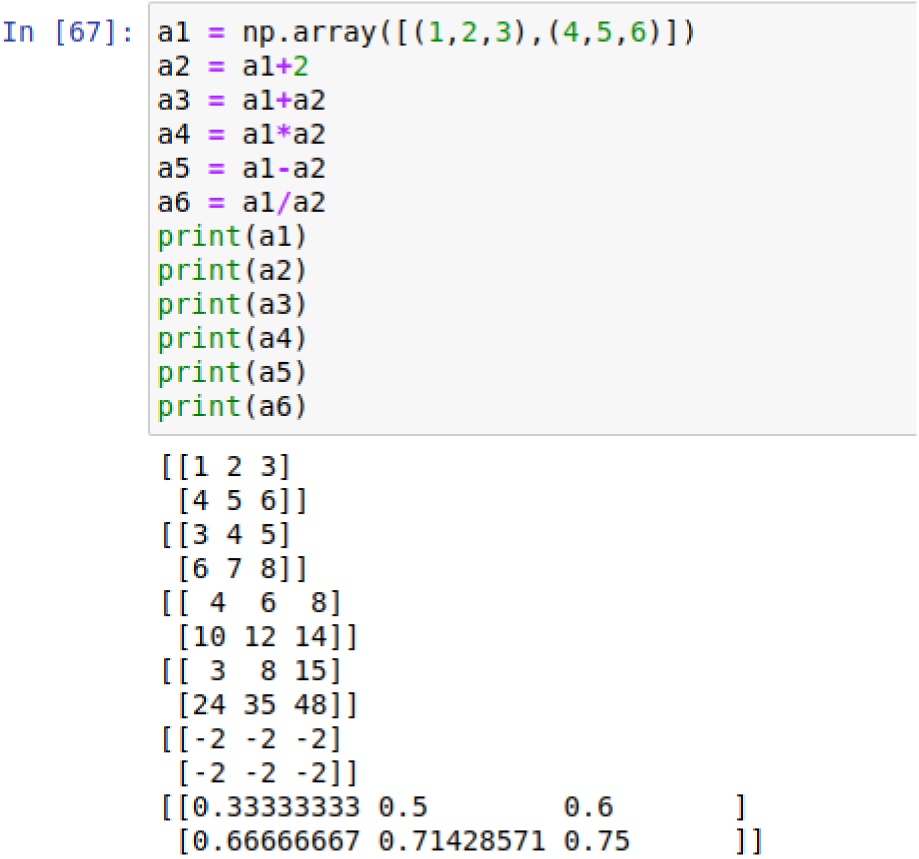


```
In [59]: import sys
         np.set_printoptions(threshold=sys.maxsize)
         print(np.arange(10000).reshape(100, 100))
```

```
[[   0    1    2    3    4    5    6    7    8    9   10   11   12   13
    14   15   16   17   18   19   20   21   22   23   24   25   26   27
    28   29   30   31   32   33   34   35   36   37   38   39   40   41
    42   43   44   45   46   47   48   49   50   51   52   53   54   55
    56   57   58   59   60   61   62   63   64   65   66   67   68   69
    70   71   72   73   74   75   76   77   78   79   80   81   82   83
    84   85   86   87   88   89   90   91   92   93   94   95   96   97
    98   99]
 [ 100  101  102  103  104  105  106  107  108  109  110  111  112  113
   114  115  116  117  118  119  120  121  122  123  124  125  126  127
   128  129  130  131  132  133  134  135  136  137  138  139  140  141
   142  143  144  145  146  147  148  149  150  151  152  153  154  155
   156  157  158  159  160  161  162  163  164  165  166  167  168  169
   170  171  172  173  174  175  176  177  178  179  180  181  182  183
   184  185  186  187  188  189  190  191  192  193  194  195  196  197
   198  199]
 [ 200  201  202  203  204  205  206  207  208  209  210  211  212  213
   214  215  216  217  218  219  220  221  222  223  224  225  226  227
   228  229  230  231  232  233  234  235  236  237  238  239  240  241
```

ARITHMETIC

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

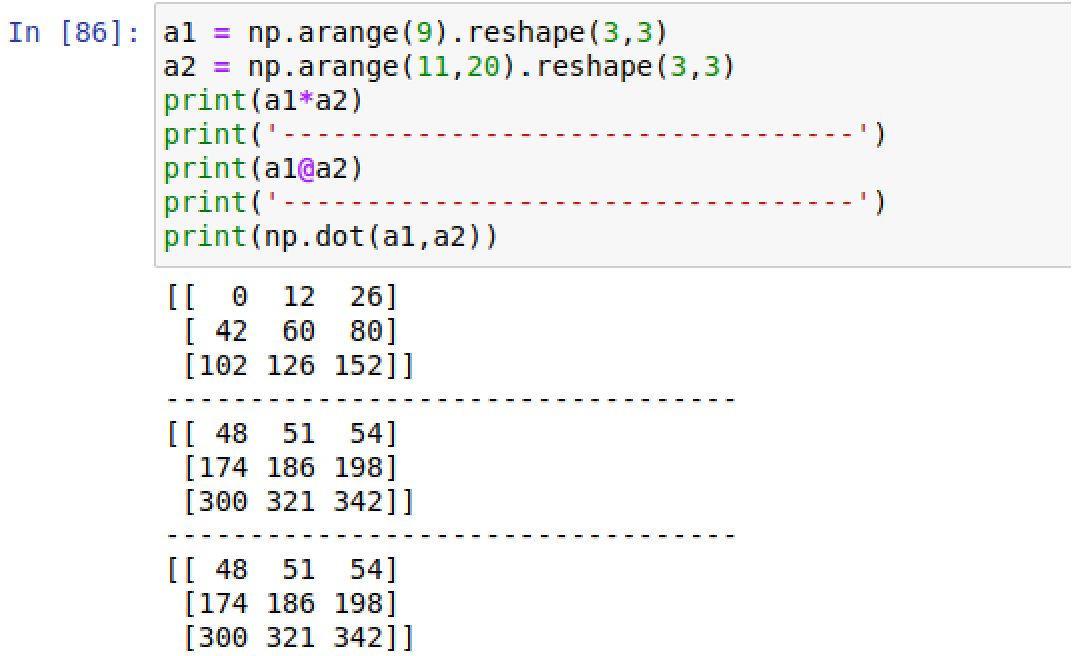


```
In [67]: a1 = np.array([(1,2,3),(4,5,6)])
         a2 = a1+2
         a3 = a1+a2
         a4 = a1*a2
         a5 = a1-a2
         a6 = a1/a2
         print(a1)
         print(a2)
         print(a3)
         print(a4)
         print(a5)
         print(a6)
```

```
[[1 2 3]
 [4 5 6]]
[[3 4 5]
 [6 7 8]]
[[ 4  6  8]
 [10 12 14]]
[[ 3  8 15]
 [24 35 48]]
[[-2 -2 -2]
 [-2 -2 -2]]
[[0.33333333 0.5        0.6       ]
 [0.66666667 0.71428571 0.75      ]]
```

The Matrix Product

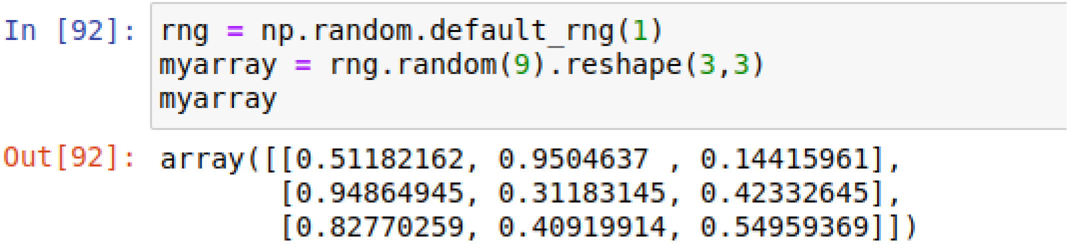
The product operator * operates elementwise. The matrix product can be performed using the @ operator (in python >=3.5) or the `dot` function or method.



```
In [86]: a1 = np.arange(9).reshape(3,3)
         a2 = np.arange(11,20).reshape(3,3)
         print(a1*a2)
         print('----------------------------------')
         print(a1@a2)
         print('----------------------------------')
         print(np.dot(a1,a2))

         [[  0  12  26]
          [ 42  60  80]
          [102 126 152]]
         ----------------------------------
         [[ 48  51  54]
          [174 186 198]
          [300 321 342]]
         ----------------------------------
         [[ 48  51  54]
          [174 186 198]
          [300 321 342]]
```

The Random Number Generator

It creates an instance of default random number generator which can be used for random number generation.



```
In [92]: rng = np.random.default_rng(1)
         myarray = rng.random(9).reshape(3,3)
         myarray

Out[92]: array([[0.51182162, 0.9504637 , 0.14415961],
                [0.94864945, 0.31183145, 0.42332645],
                [0.82770259, 0.40919914, 0.54959369]])
```

UNARY OPERATIONS

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```
> a.sum()
      3.1057109529998157

> a.min()
      0.027559113243068367

> a.max()
      0.8277025938204418
```

By specifying the axis parameter you can apply an operation along the specified axis of an array.

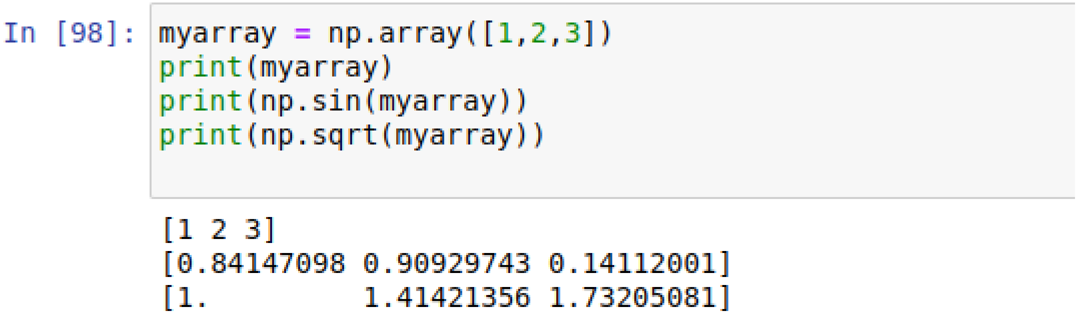
```
[  [ 0,   1,    2,    3   ],
   [ 4,   5,    6,    7   ],
   [ 8,   9,   10,   11   ]   ]
```

```
> b.sum(axis=0)                    # sum of each column
array([12, 15, 18, 21])

> b.min(axis=1)                    # min of each row
array([0, 4, 8])
```

```
> b.cumsum(axis=1)  # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

UNIVERSAL FUNCTIONS

NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions” (`ufunc`). These functions operate elementwise.

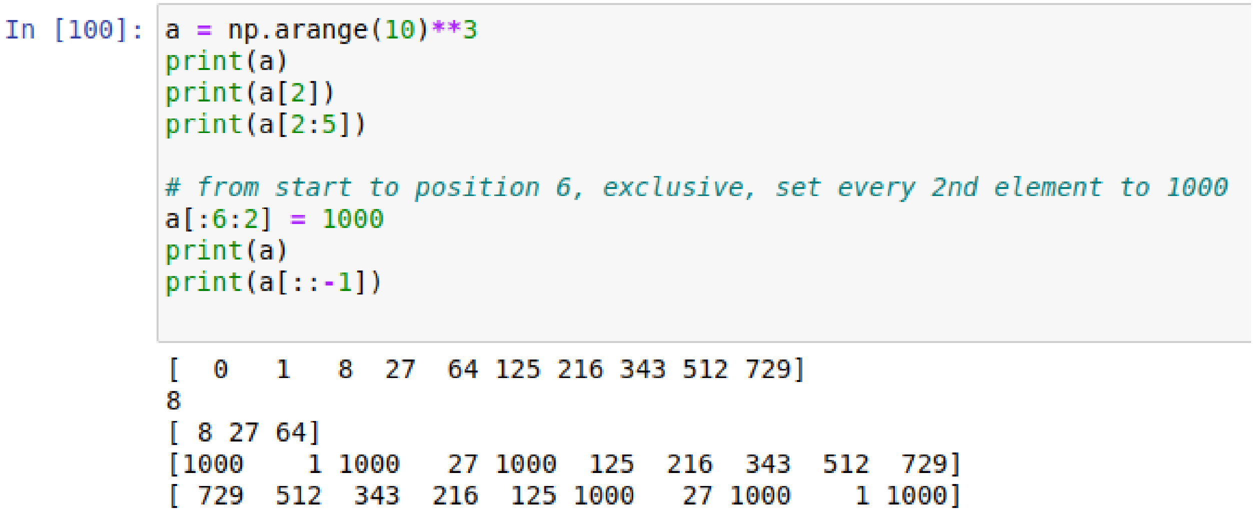


```
In [98]: myarray = np.array([1,2,3])
         print(myarray)
         print(np.sin(myarray))
         print(np.sqrt(myarray))
```

```
[1 2 3]
[0.84147098 0.90929743 0.14112001]
[1.         1.41421356 1.73205081]
```

INDEXING & SLICING

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

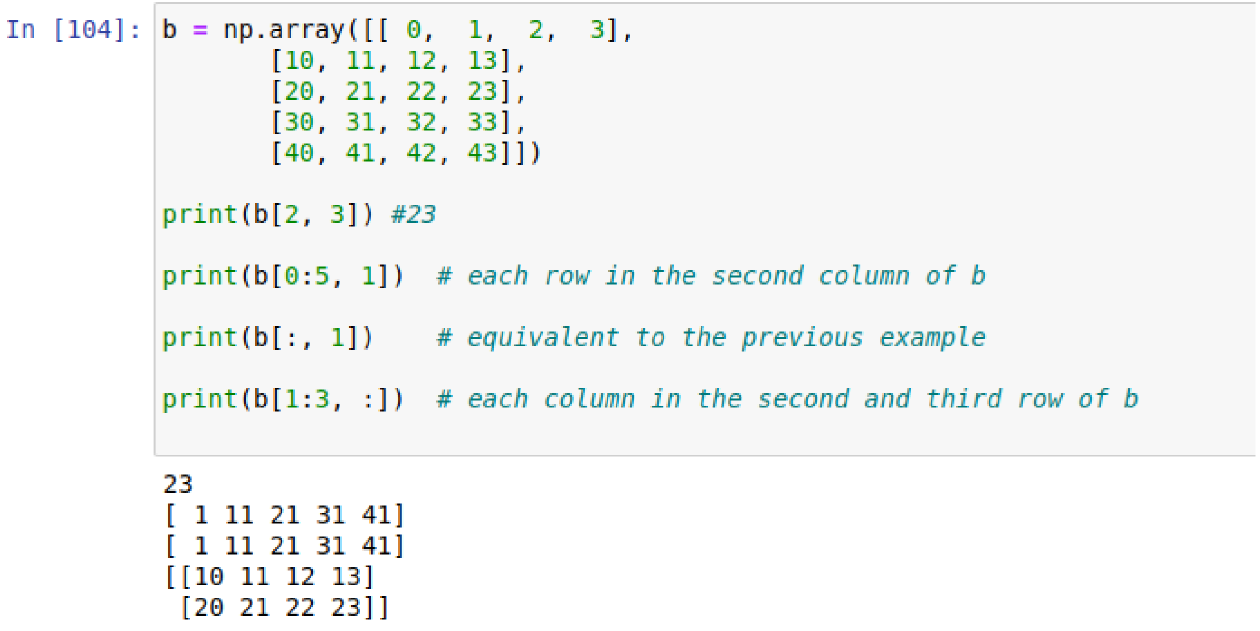


```
In [100]: a = np.arange(10)**3
          print(a)
          print(a[2])
          print(a[2:5])

          # from start to position 6, exclusive, set every 2nd element to 1000
          a[:6:2] = 1000
          print(a)
          print(a[::-1])
```

```
[  0   1   8  27  64 125 216 343 512 729]
8
[ 8 27 64]
[1000    1 1000   27 1000  125  216  343  512  729]
[ 729  512  343  216  125 1000   27 1000    1 1000]
```

INDEXING & SLICING ON MULTIDIMENSIONAL ARRAYS

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas.

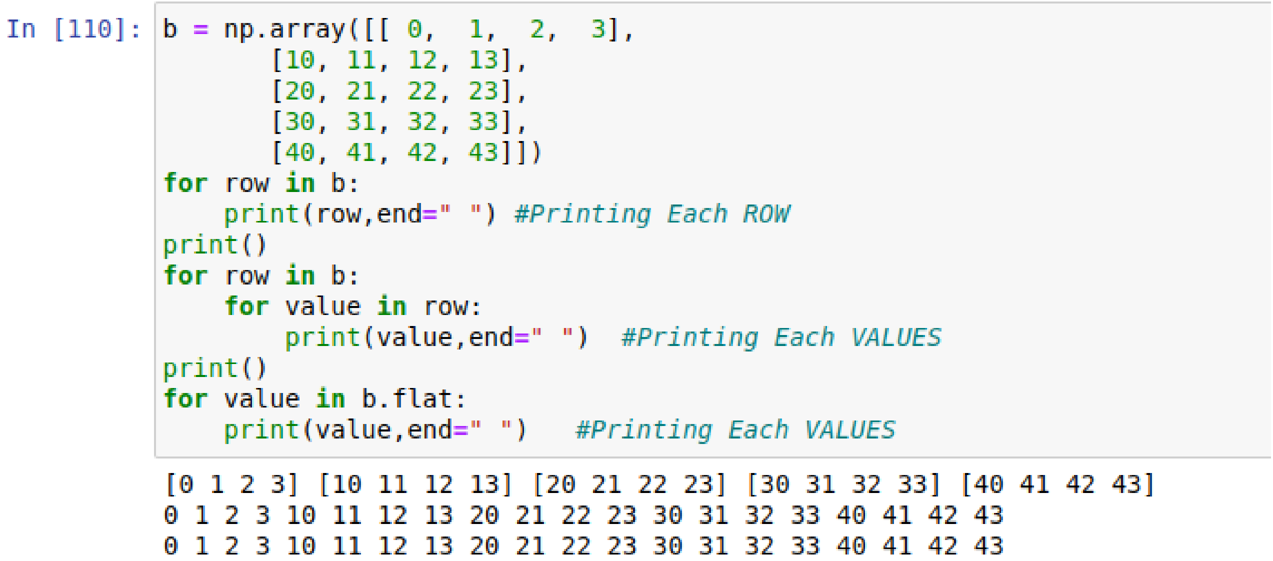


```
In [104]: b = np.array([[ 0,  1,  2,  3],
                        [10, 11, 12, 13],
                        [20, 21, 22, 23],
                        [30, 31, 32, 33],
                        [40, 41, 42, 43]])

          print(b[2, 3]) #23

          print(b[0:5, 1])  # each row in the second column of b

          print(b[:, 1])    # equivalent to the previous example

          print(b[1:3, :])  # each column in the second and third row of b
```

```
23
[ 1 11 21 31 41]
[ 1 11 21 31 41]
[[10 11 12 13]
 [20 21 22 23]]
```

ITERATION

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas.

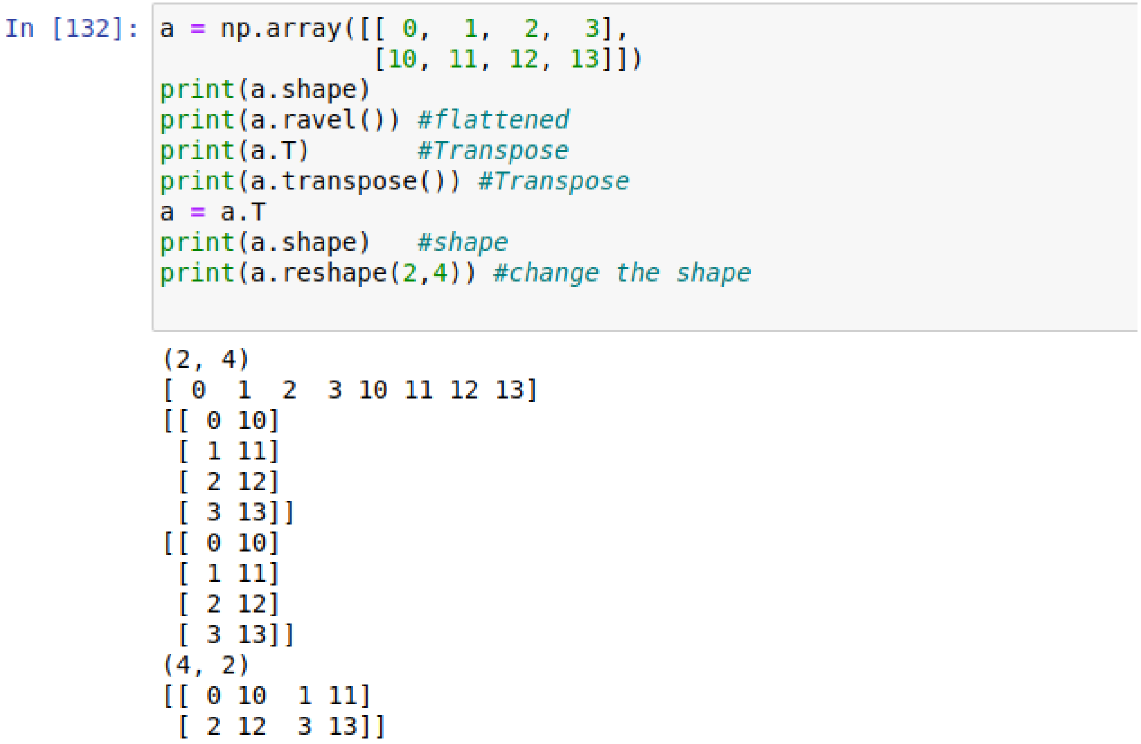


```
In [110]: b = np.array([[ 0,  1,  2,  3],
                 [10, 11, 12, 13],
                 [20, 21, 22, 23],
                 [30, 31, 32, 33],
                 [40, 41, 42, 43]])
          for row in b:
              print(row,end=" ") #Printing Each ROW
          print()
          for row in b:
              for value in row:
                  print(value,end=" ")  #Printing Each VALUES
          print()
          for value in b.flat:
              print(value,end=" ")   #Printing Each VALUES
```

```
[0 1 2 3] [10 11 12 13] [20 21 22 23] [30 31 32 33] [40 41 42 43]
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43
```

PLAYING WITH SHAPES

An array has a shape given by the number of elements along each axis.

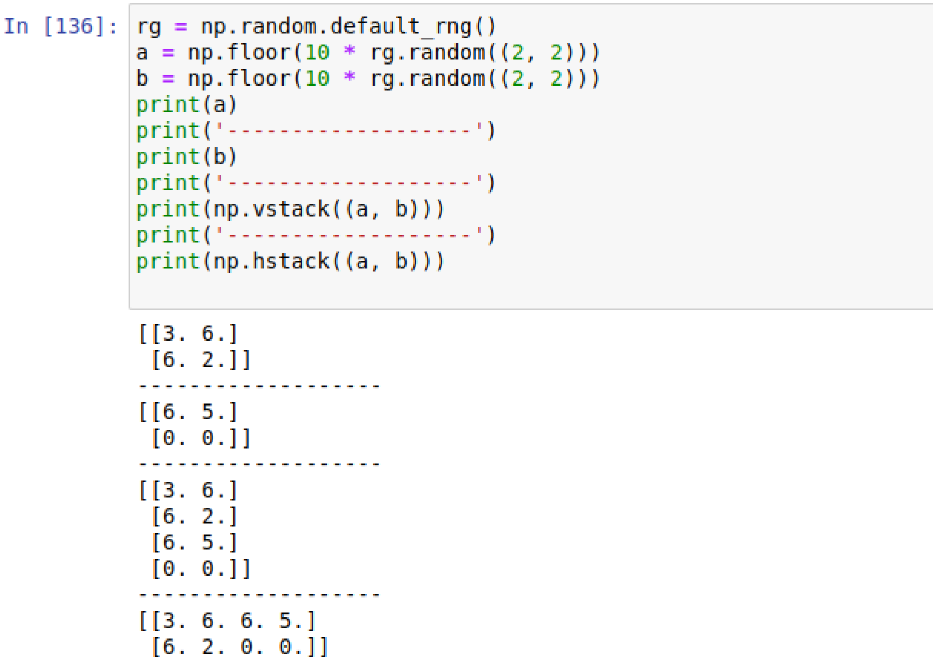


```
In [132]: a = np.array([[ 0,  1,  2,  3],
                        [10, 11, 12, 13]])
          print(a.shape)
          print(a.ravel()) #flattened
          print(a.T)       #Transpose
          print(a.transpose()) #Transpose
          a = a.T
          print(a.shape)   #shape
          print(a.reshape(2,4)) #change the shape
```

```
(2, 4)
[ 0  1  2  3 10 11 12 13]
[[ 0 10]
 [ 1 11]
 [ 2 12]
 [ 3 13]]
[[ 0 10]
 [ 1 11]
 [ 2 12]
 [ 3 13]]
(4, 2)
[[ 0 10  1 11]
 [ 2 12  3 13]]
```

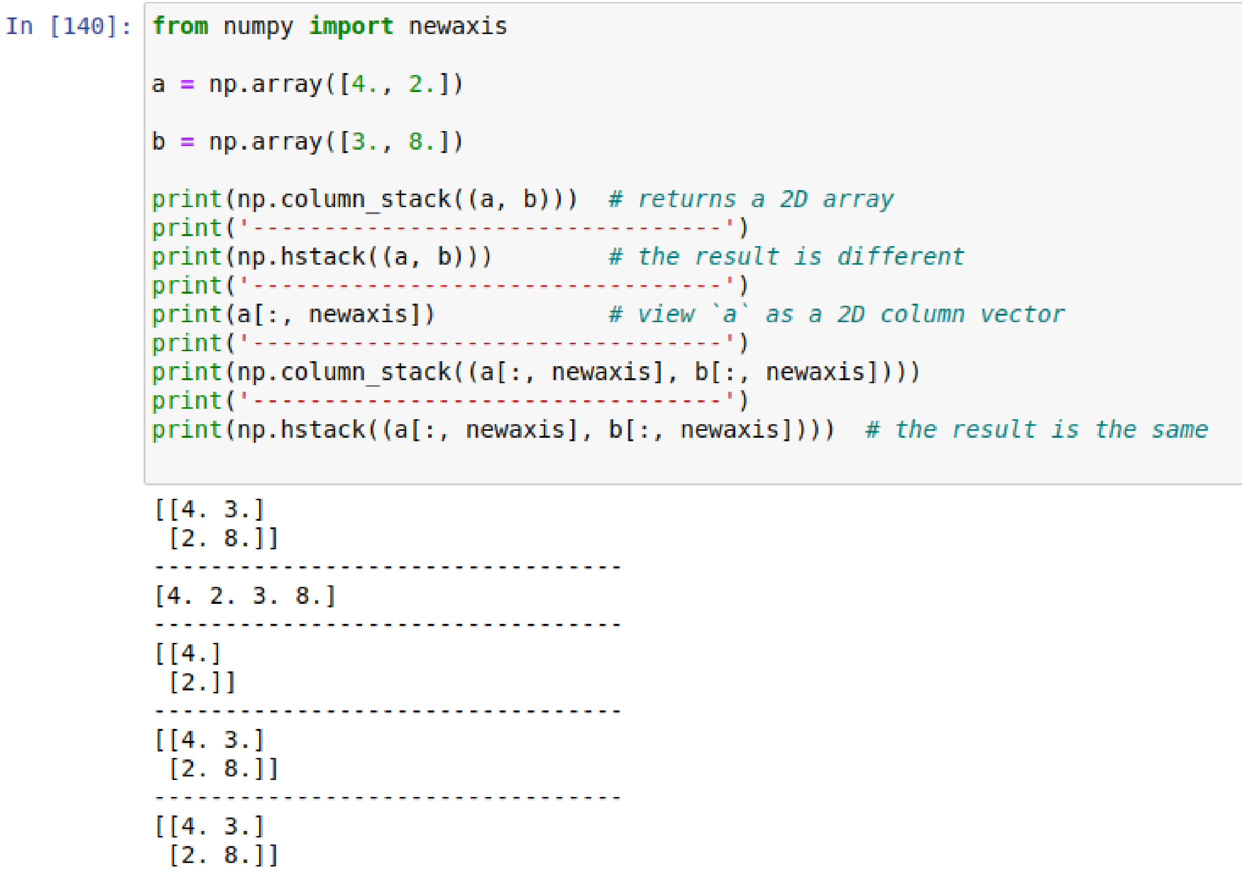
STACKING

Arrays can be stacked together along different axes.



```
In [136]: rg = np.random.default_rng()
          a = np.floor(10 * rg.random((2, 2)))
          b = np.floor(10 * rg.random((2, 2)))
          print(a)
          print('-------------------')
          print(b)
          print('-------------------')
          print(np.vstack((a, b)))
          print('-------------------')
          print(np.hstack((a, b)))
```

```
[[3. 6.]
 [6. 2.]]
-------------------
[[6. 5.]
 [0. 0.]]
-------------------
[[3. 6.]
 [6. 2.]
 [6. 5.]
 [0. 0.]]
-------------------
[[3. 6. 6. 5.]
 [6. 2. 0. 0.]]
```



```
In [140]: from numpy import newaxis

          a = np.array([4., 2.])

          b = np.array([3., 8.])

          print(np.column_stack((a, b)))  # returns a 2D array
          print('---------------------------------')
          print(np.hstack((a, b)))        # the result is different
          print('---------------------------------')
          print(a[:, newaxis])            # view `a` as a 2D column vector
          print('---------------------------------')
          print(np.column_stack((a[:, newaxis], b[:, newaxis])))
          print('---------------------------------')
          print(np.hstack((a[:, newaxis], b[:, newaxis])))  # the result is the same
```

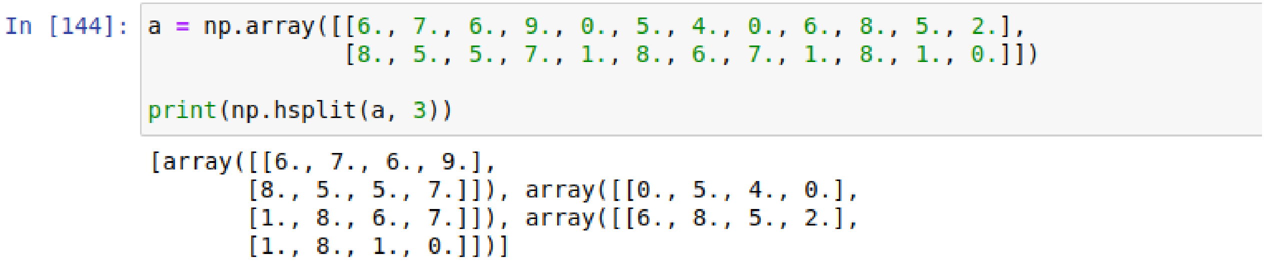
```
[[4. 3.]
 [2. 8.]]
---------------------------------
[4. 2. 3. 8.]
---------------------------------
[[4.]
 [2.]]
---------------------------------
[[4. 3.]
 [2. 8.]]
---------------------------------
[[4. 3.]
 [2. 8.]]
```

We also have row_stack( ) which is equal to the vstack( ) function.

SPLITTING

Arrays can be splitted along different axes.

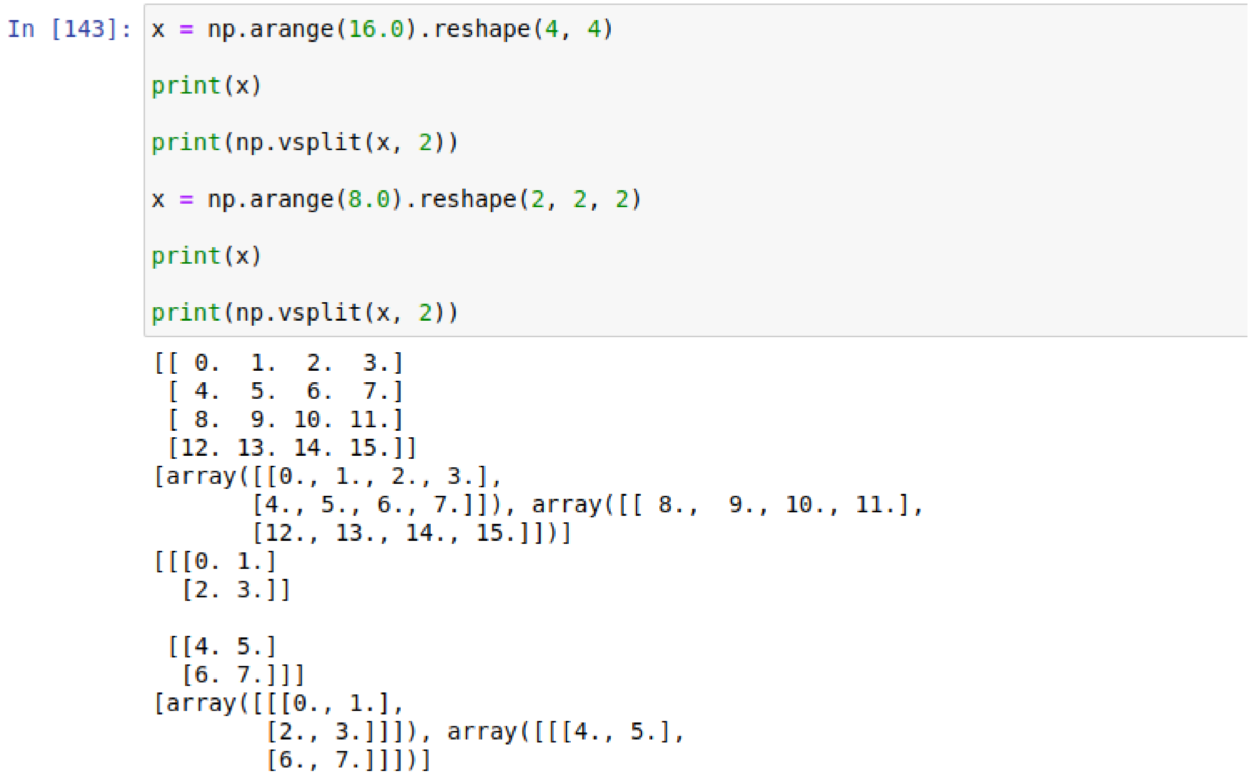
Splitting Horizontally...



```
In [144]: a = np.array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
                        [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])

          print(np.hsplit(a, 3))
```

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

Splitting Vertically...



```
In [143]: x = np.arange(16.0).reshape(4, 4)

          print(x)

          print(np.vsplit(x, 2))

          x = np.arange(8.0).reshape(2, 2, 2)

          print(x)

          print(np.vsplit(x, 2))
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]]
[array([[0., 1., 2., 3.],
       [4., 5., 6., 7.]]), array([[ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])]
[[[0. 1.]
  [2. 3.]]

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

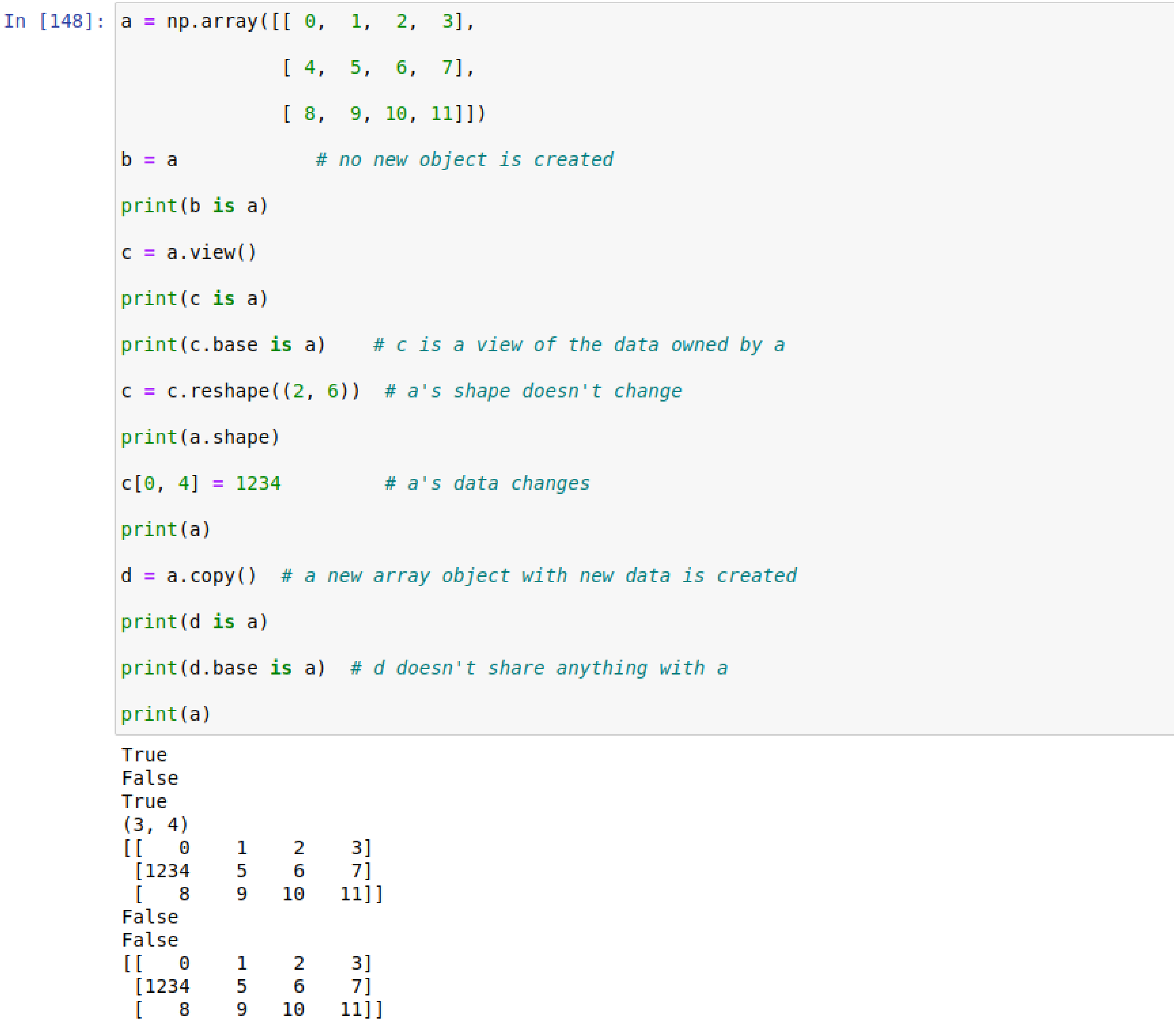
COPY & VIEWS

When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not.

No Copy at All : Simple assignments make no copy of objects or their data.

Shallow Copy / View : The `view` method creates a new array object that looks at the same data.

Deep Copy / View : The `copy` method makes a complete copy of the array and its data.

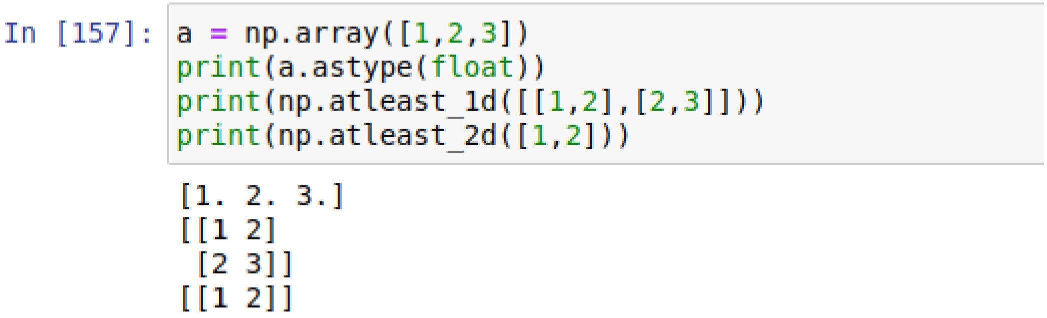


```
In [148]: a = np.array([[ 0,  1,  2,  3],
                        [ 4,  5,  6,  7],
                        [ 8,  9, 10, 11]])
          b = a           # no new object is created
          print(b is a)
          c = a.view()
          print(c is a)
          print(c.base is a)    # c is a view of the data owned by a
          c = c.reshape((2, 6))  # a's shape doesn't change
          print(a.shape)
          c[0, 4] = 1234         # a's data changes
          print(a)
          d = a.copy()  # a new array object with new data is created
          print(d is a)
          print(d.base is a)  # d doesn't share anything with a
          print(a)
```

```
True
False
True
(3, 4)
[[   0    1    2    3]
 [1234    5    6    7]
 [   8    9   10   11]]
False
False
[[   0    1    2    3]
 [1234    5    6    7]
 [   8    9   10   11]]
```

CONVERSIONS

To convert between different types, we have some methods available.

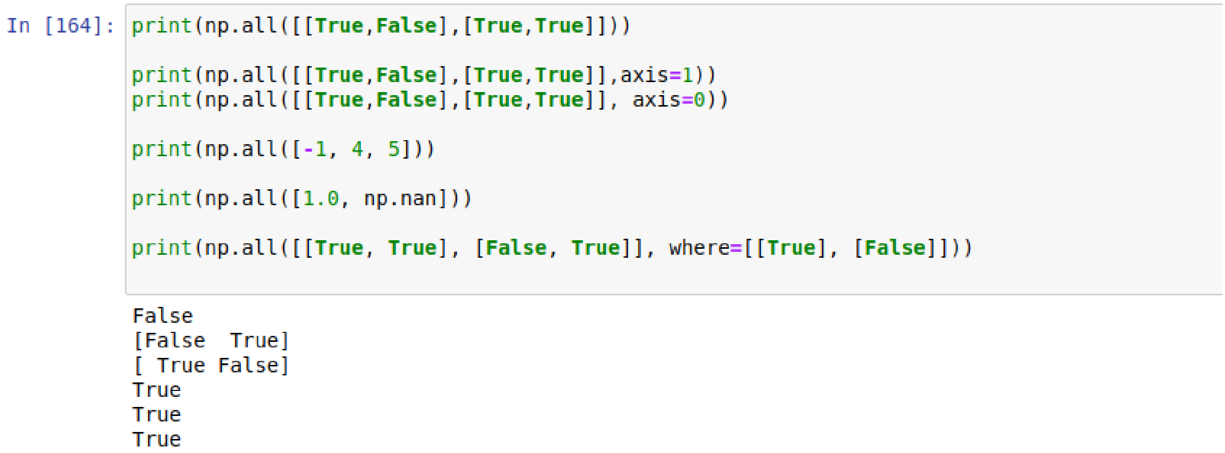


```
In [157]: a = np.array([1,2,3])
          print(a.astype(float))
          print(np.atleast_1d([[1,2],[2,3]]))
          print(np.atleast_2d([1,2]))
```

```
[1. 2. 3.]
[[1 2]
 [2 3]]
[[1 2]]
```

CONDITIONS

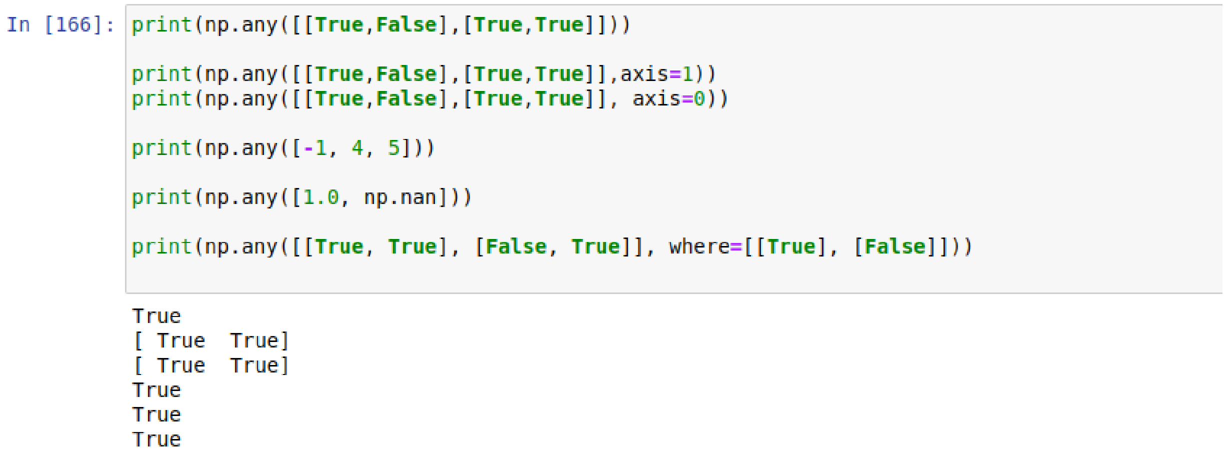
all( ) : All has to be True



```
In [164]: print(np.all([[True,False],[True,True]]))

          print(np.all([[True,False],[True,True]],axis=1))
          print(np.all([[True,False],[True,True]], axis=0))

          print(np.all([-1, 4, 5]))

          print(np.all([1.0, np.nan]))

          print(np.all([[True, True], [False, True]], where=[[True], [False]]))
```

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

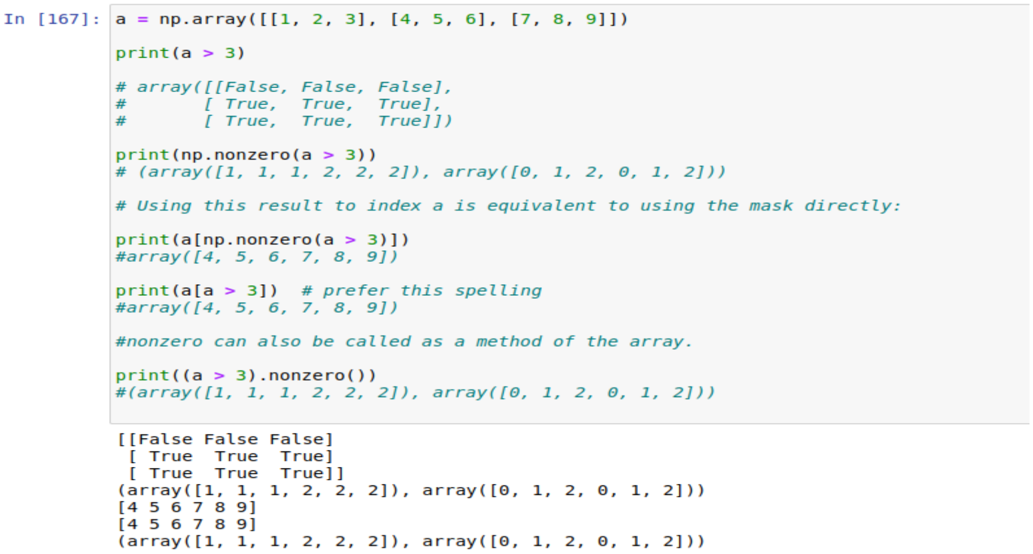
any( ) : Any one has to be True



```
In [166]: print(np.any([[True,False],[True,True]]))

          print(np.any([[True,False],[True,True]],axis=1))
          print(np.any([[True,False],[True,True]], axis=0))

          print(np.any([-1, 4, 5]))

          print(np.any([1.0, np.nan]))

          print(np.any([[True, True], [False, True]], where=[[True], [False]]))
```

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

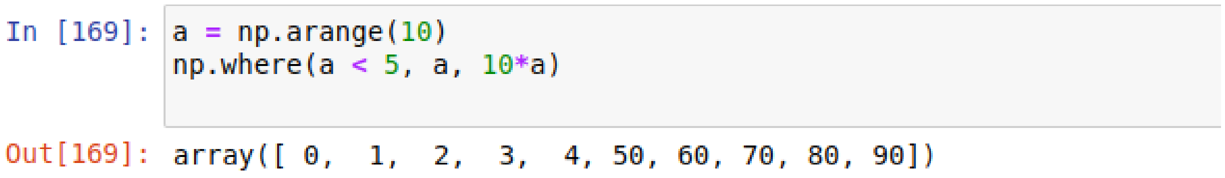
nonzero( ) : To return positions of non-zero elements



```
In [167]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

          print(a > 3)

          # array([[False, False, False],
          #        [ True,  True,  True],
          #        [ True,  True,  True]])

          print(np.nonzero(a > 3))
          # (array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))

          # Using this result to index a is equivalent to using the mask directly:

          print(a[np.nonzero(a > 3)])
          #array([4, 5, 6, 7, 8, 9])

          print(a[a > 3])  # prefer this spelling
          #array([4, 5, 6, 7, 8, 9])

          #nonzero can also be called as a method of the array.

          print((a > 3).nonzero())
          #(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

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

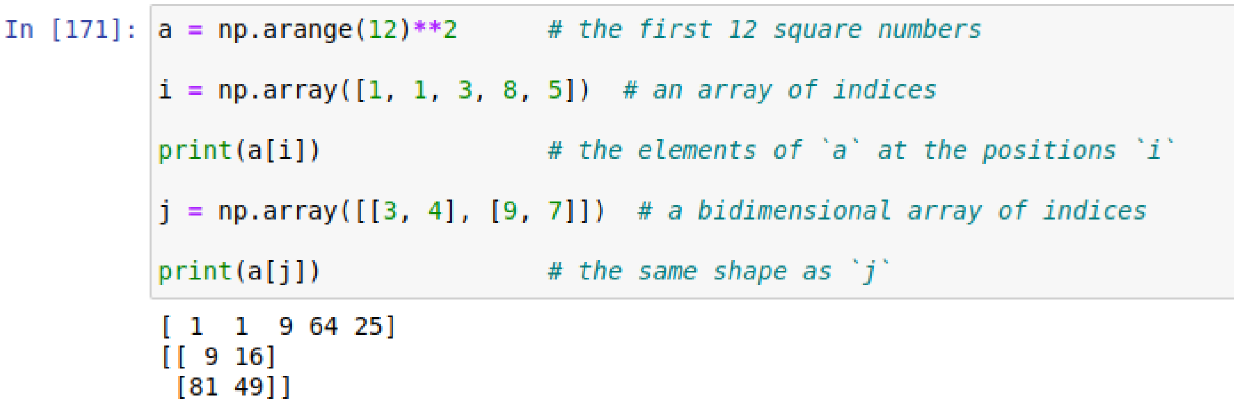
where( ) : Works as Ternary Operator



```
In [169]: a = np.arange(10)
          np.where(a < 5, a, 10*a)

Out[169]: array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

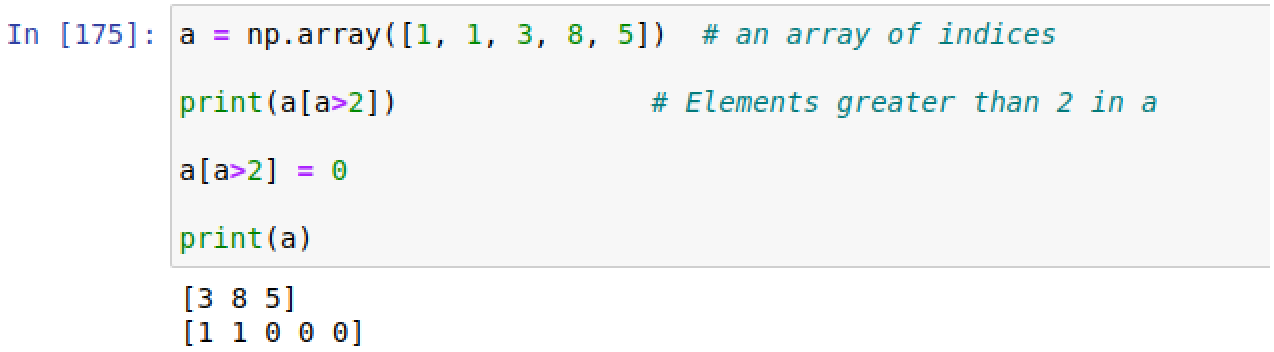
ADVANCED INDEXING

Indexing with Array of Indices :



```
In [171]: a = np.arange(12)**2      # the first 12 square numbers

          i = np.array([1, 1, 3, 8, 5])  # an array of indices

          print(a[i])              # the elements of `a` at the positions `i`

          j = np.array([[3, 4], [9, 7]])  # a bidimensional array of indices

          print(a[j])              # the same shape as `j`

          [ 1  1  9 64 25]
          [[ 9 16]
           [81 49]]
```

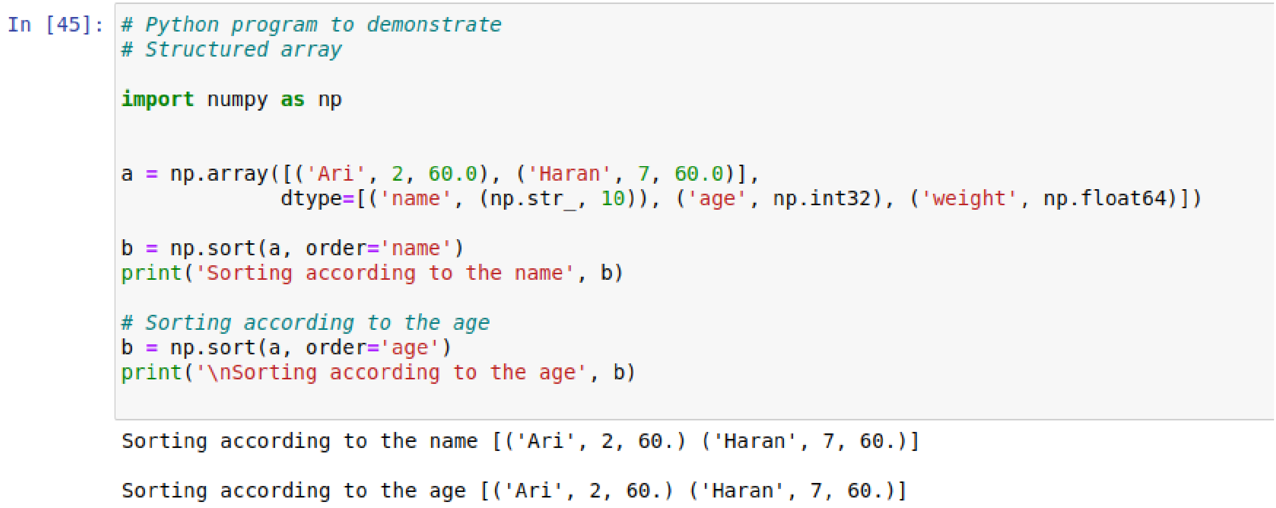
Indexing with Boolean Arrays :



```
In [175]: a = np.array([1, 1, 3, 8, 5])  # an array of indices

          print(a[a>2])              # Elements greater than 2 in a

          a[a>2] = 0

          print(a)

          [3 8 5]
          [1 1 0 0 0]
```

STRUCTURED ARRAY

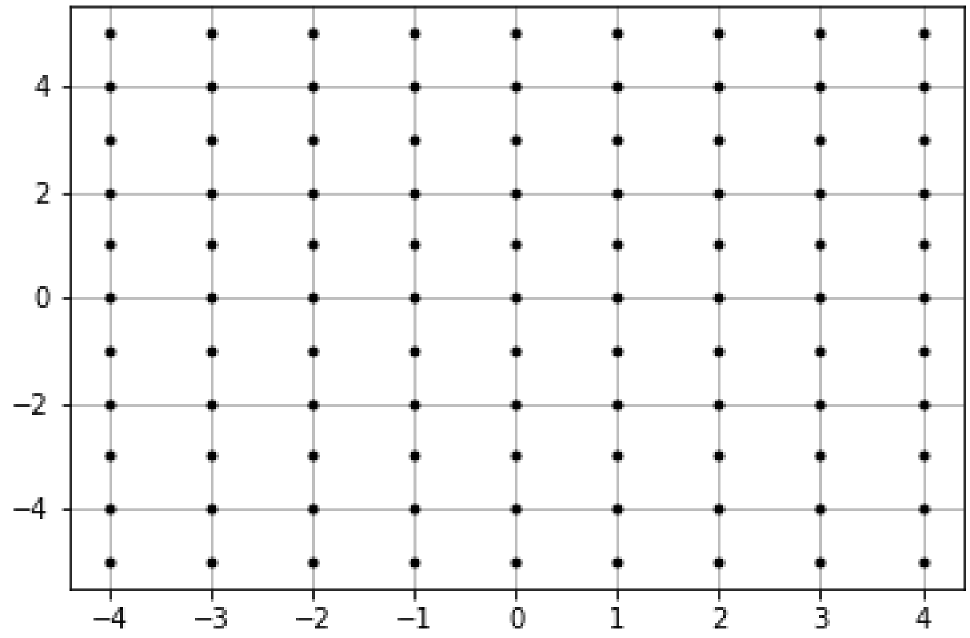
Numpy's Structured Array is similar to Struct in C. It is used for grouping data of different types and sizes. Structured array uses data containers called fields. Each data field can contain data of any type and size. Array elements can be accessed with the help of dot-notation.

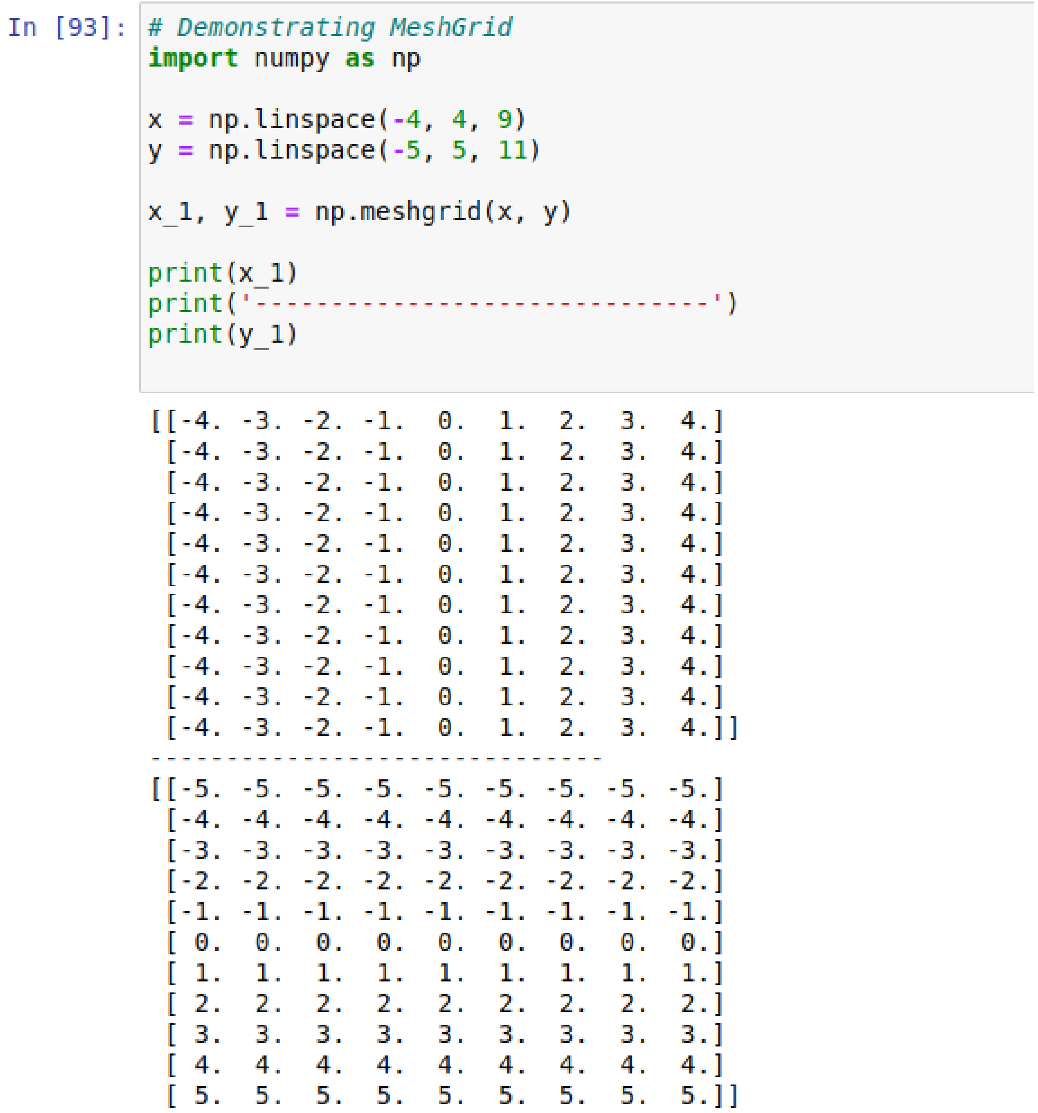


```
In [45]: # Python program to demonstrate
         # Structured array

         import numpy as np


         a = np.array([('Ari', 2, 60.0), ('Haran', 7, 60.0)],
                      dtype=[('name', (np.str_, 10)), ('age', np.int32), ('weight', np.float64)])

         b = np.sort(a, order='name')
         print('Sorting according to the name', b)

         # Sorting according to the age
         b = np.sort(a, order='age')
         print('\nSorting according to the age', b)

         Sorting according to the name [('Ari', 2, 60.) ('Haran', 7, 60.)]

         Sorting according to the age [('Ari', 2, 60.) ('Haran', 7, 60.)]
```

MESH-GRID

The numpy.meshgrid function is used to create a rectangular grid out of two given one-dimensional arrays representing the Cartesian indexing or Matrix indexing. Consider the figure with X-axis ranging from -4 to 4 and Y-axis ranging from -5 to 5. So there are a total of (9 * 11) = 99 points marked in the figure each with a X-coordinate and a Y-coordinate.



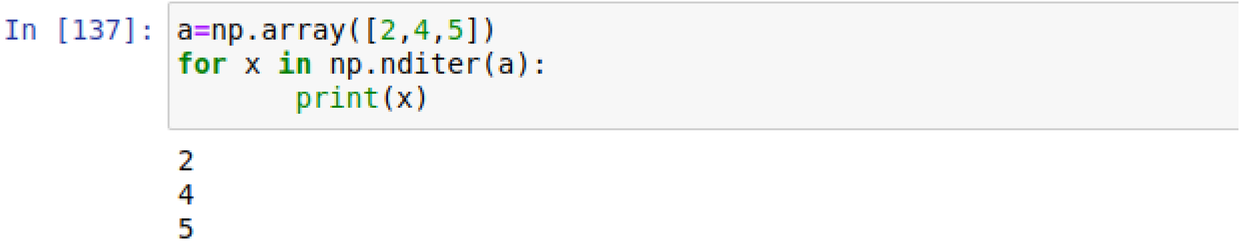


```
In [93]: # Demonstrating MeshGrid
         import numpy as np

         x = np.linspace(-4, 4, 9)
         y = np.linspace(-5, 5, 11)

         x_1, y_1 = np.meshgrid(x, y)

         print(x_1)
         print('-----------------------------')
         print(y_1)
```

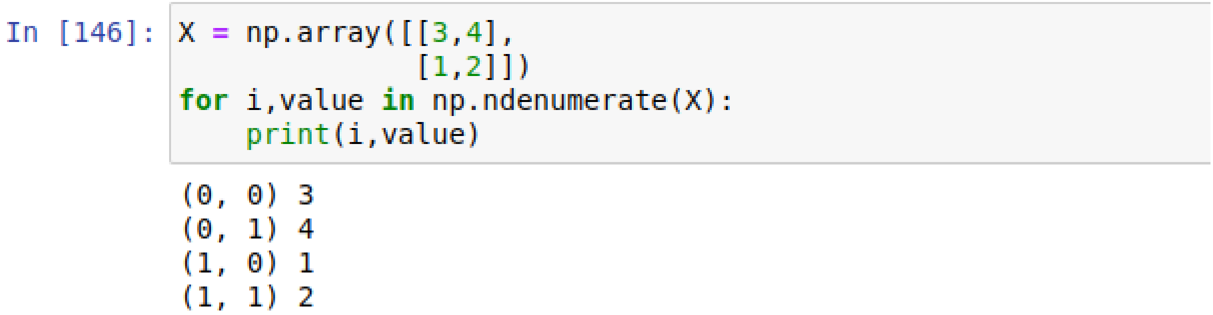
```
[[-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.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

Some Cool Iterations

Numpy contains a function nditer() that can be used for very basic iterations to advanced iterations.

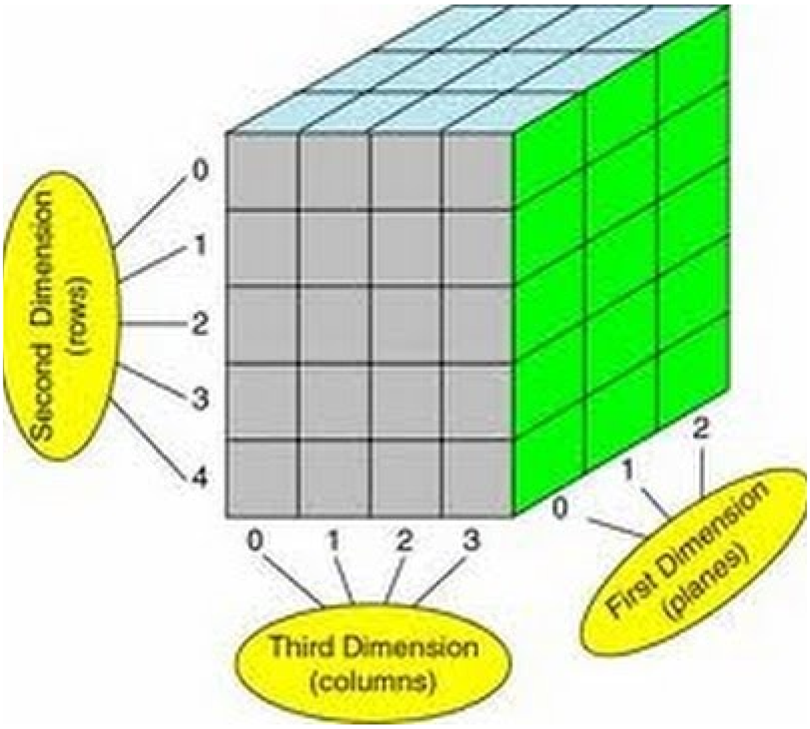


```
In [137]: a=np.array([2,4,5])
          for x in np.nditer(a):
                  print(x)

          2
          4
          5
```



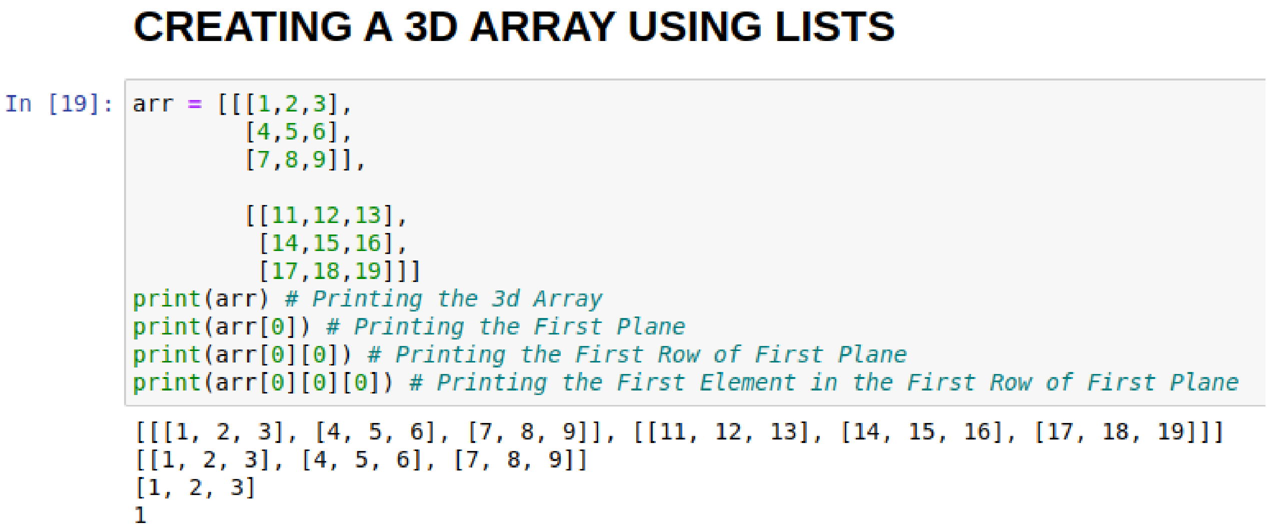
```
In [146]: X = np.array([[3,4],
                        [1,2]])
          for i,value in np.ndenumerate(X):
              print(i,value)

          (0, 0) 3
          (0, 1) 4
          (1, 0) 1
          (1, 1) 2
```

Working on 3D Arrays



★ Creating A 3D Array

We can create a 3D Array using the nested lists.

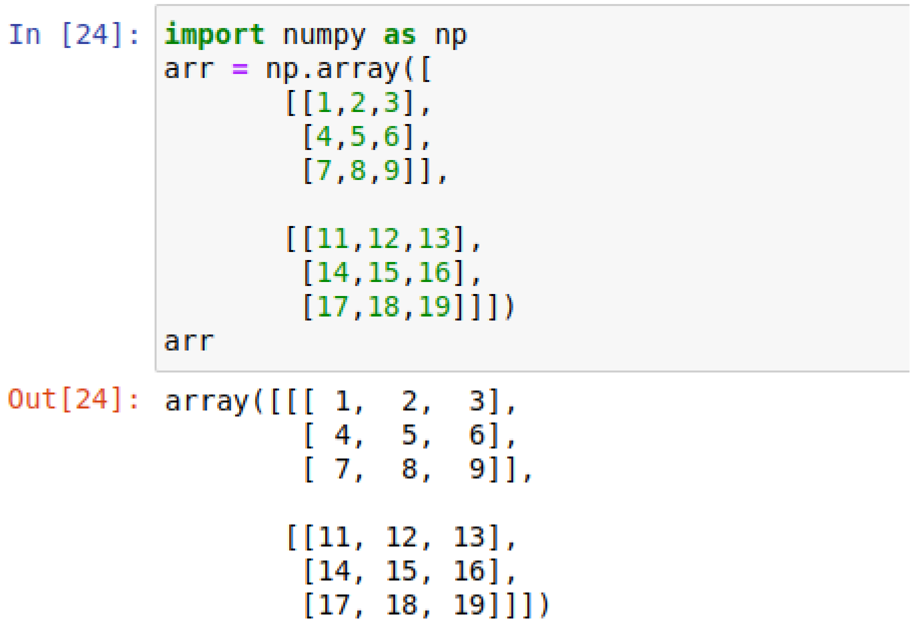


CREATING A 3D ARRAY USING LISTS

```
In [19]: arr = [[[1,2,3],
                 [4,5,6],
                 [7,8,9]],

                [[11,12,13],
                 [14,15,16],
                 [17,18,19]]]
         print(arr) # Printing the 3d Array
         print(arr[0]) # Printing the First Plane
         print(arr[0][0]) # Printing the First Row of First Plane
         print(arr[0][0][0]) # Printing the First Element in the First Row of First Plane
```

```
[[[1, 2, 3], [4, 5, 6], [7, 8, 9]], [[11, 12, 13], [14, 15, 16], [17, 18, 19]]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[1, 2, 3]
1
```

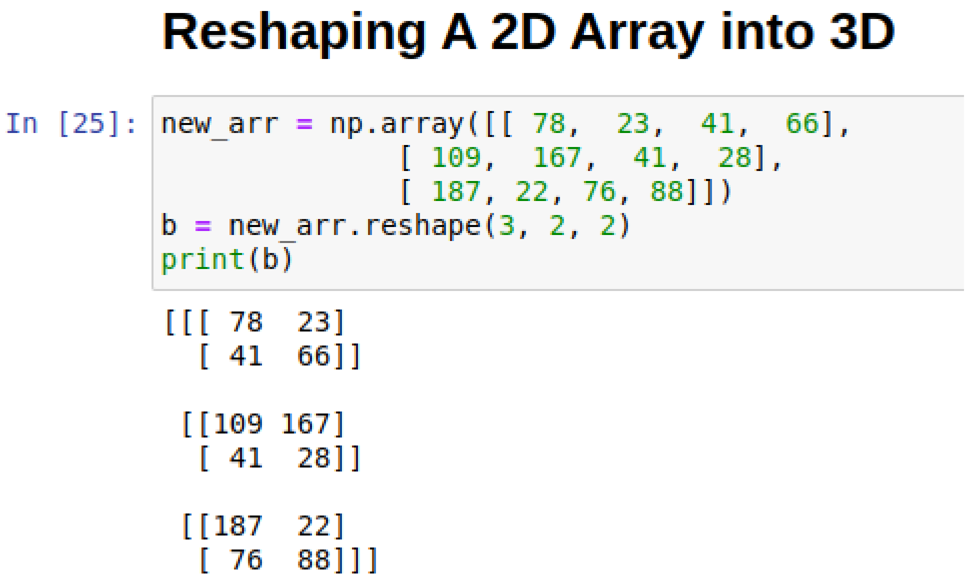
We can achieve the same using ndarrays.



```
In [24]: import numpy as np
         arr = np.array([
                [[1,2,3],
                 [4,5,6],
                 [7,8,9]],

                [[11,12,13],
                 [14,15,16],
                 [17,18,19]]])
         arr

Out[24]: array([[[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9]],

                [[11, 12, 13],
                 [14, 15, 16],
                 [17, 18, 19]]])
```

★ Reshaping

To convert a 2D Array into a 3D Array, we can use the reshape() method.

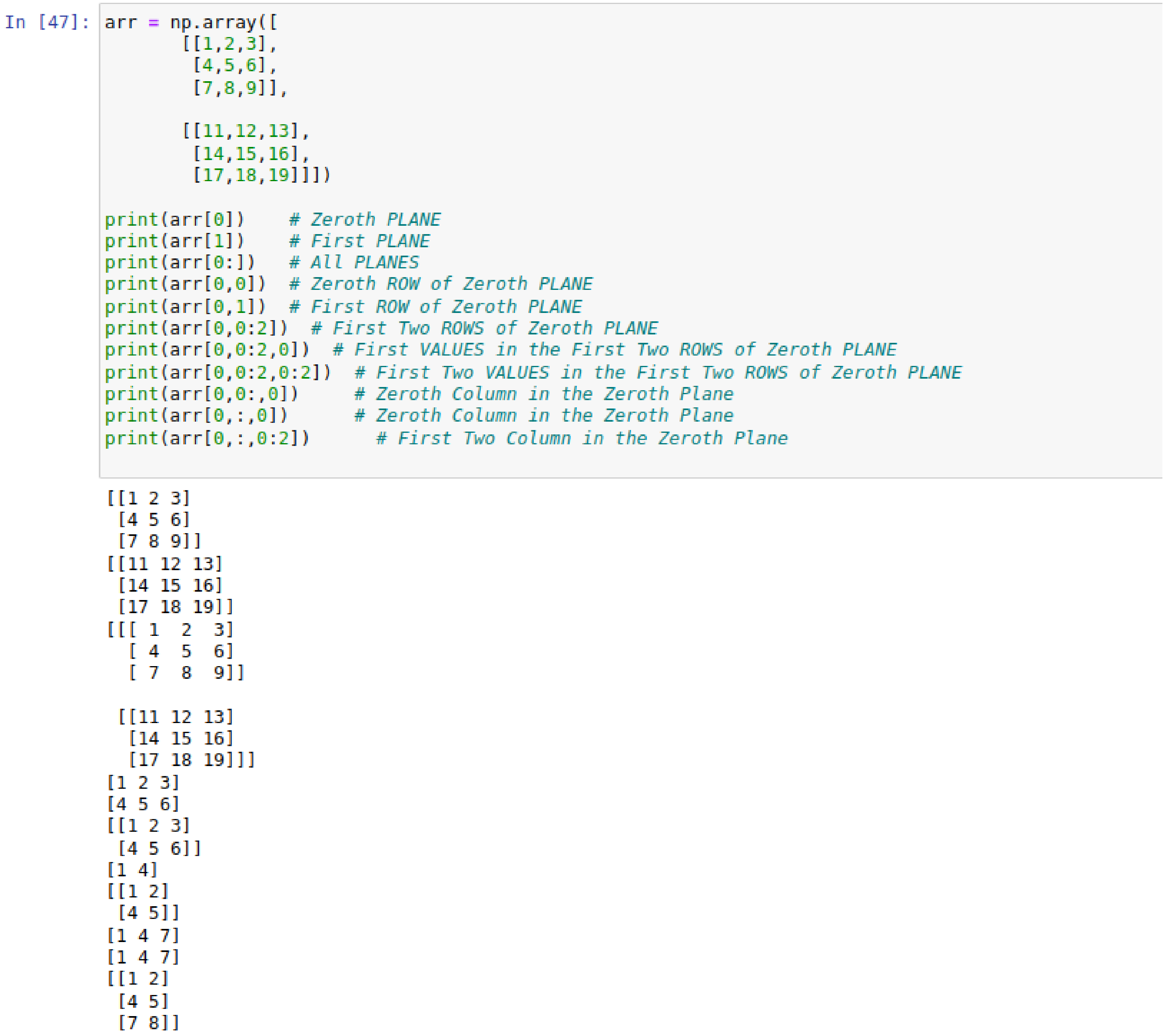


Reshaping A 2D Array into 3D

```
In [25]: new_arr = np.array([[ 78,  23,  41,  66],
                       [ 109,  167,  41,  28],
                       [ 187, 22, 76, 88]])
         b = new_arr.reshape(3, 2, 2)
         print(b)

         [[[ 78  23]
           [ 41  66]]

          [[109 167]
           [ 41  28]]

          [[187  22]
           [ 76  88]]]
```

★ Accessing & Slicing

As we do on two dimensional arrays, 3D Arrays provide slicing & accessing.

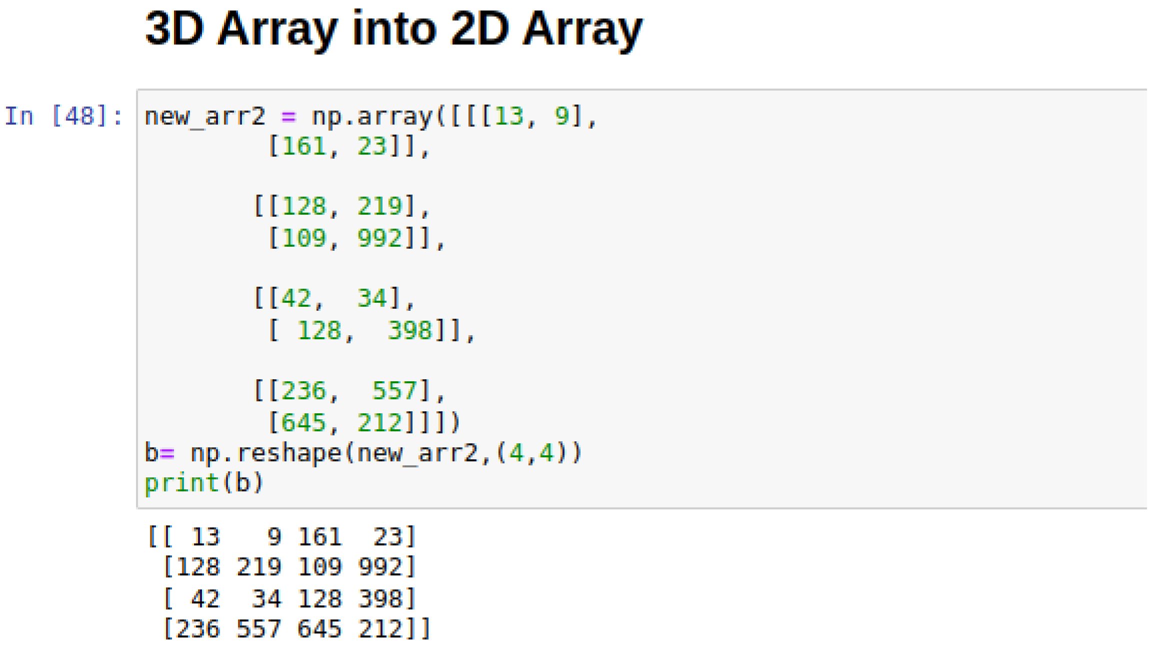


```
In [47]: arr = np.array([
                [[1,2,3],
                 [4,5,6],
                 [7,8,9]],

                [[11,12,13],
                 [14,15,16],
                 [17,18,19]]])

         print(arr[0])    # Zeroth PLANE
         print(arr[1])    # First PLANE
         print(arr[0:])   # All PLANES
         print(arr[0,0])  # Zeroth ROW of Zeroth PLANE
         print(arr[0,1])  # First ROW of Zeroth PLANE
         print(arr[0,0:2])  # First Two ROWS of Zeroth PLANE
         print(arr[0,0:2,0])  # First VALUES in the First Two ROWS of Zeroth PLANE
         print(arr[0,0:2,0:2])  # First Two VALUES in the First Two ROWS of Zeroth PLANE
         print(arr[0,0:,0])     # Zeroth Column in the Zeroth Plane
         print(arr[0,:,0])      # Zeroth Column in the Zeroth Plane
         print(arr[0,:,0:2])      # First Two Column in the Zeroth Plane
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[11 12 13]
 [14 15 16]
 [17 18 19]]
[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[11 12 13]
  [14 15 16]
  [17 18 19]]]
[1 2 3]
[4 5 6]
[[1 2 3]
 [4 5 6]]
[1 4]
[[1 2]
 [4 5]]
[1 4 7]
[1 4 7]
[[1 2]
 [4 5]
 [7 8]]
```

★ 3D Arrays into 2D Arras

Reshaping helps us here...



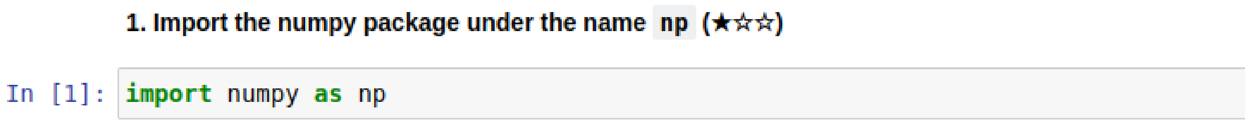
3D Array into 2D Array

```
In [48]: new_arr2 = np.array([[[13, 9],
                 [161, 23]],

                [[128, 219],
                 [109, 992]],

                [[42,  34],
                 [ 128,  398]],

                [[236,  557],
                 [645, 212]]])
         b= np.reshape(new_arr2,(4,4))
         print(b)
```

```
[[ 13   9 161  23]
 [128 219 109 992]
 [ 42  34 128 398]
 [236 557 645 212]]
```

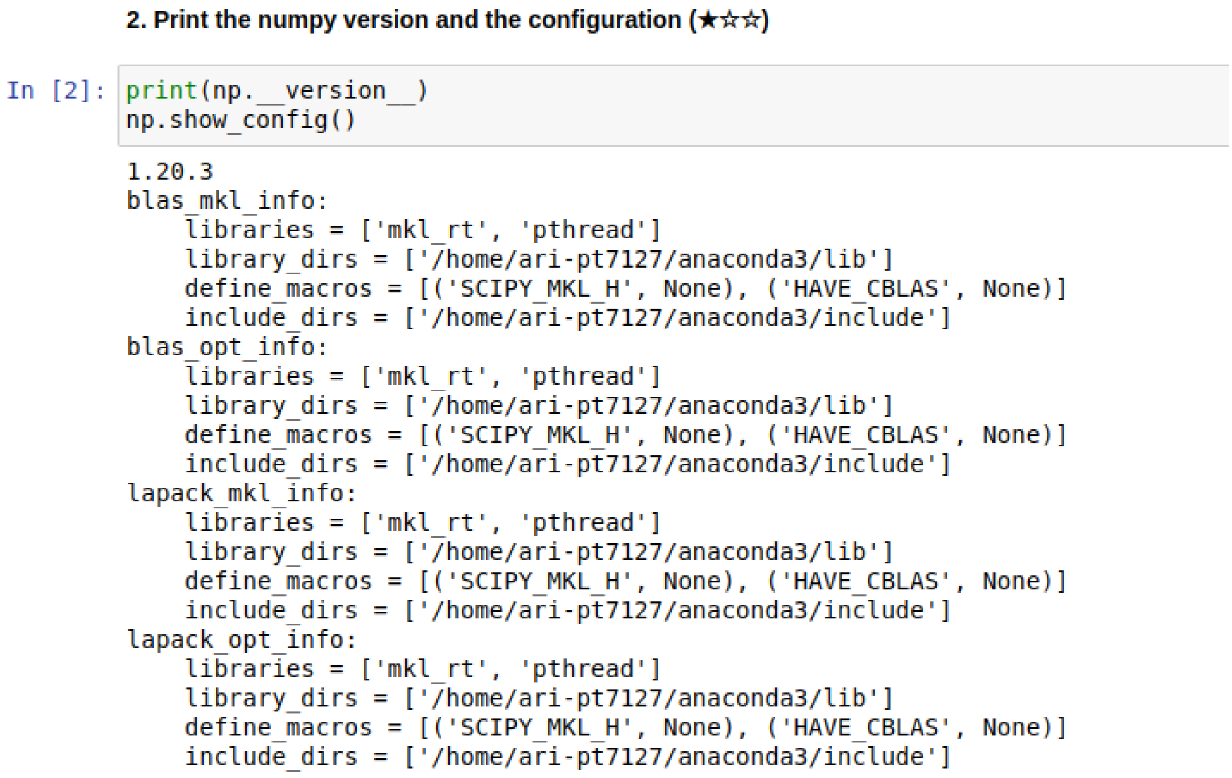
NumPy Exercises

Found at : https://github.com/rougier/numpy-10



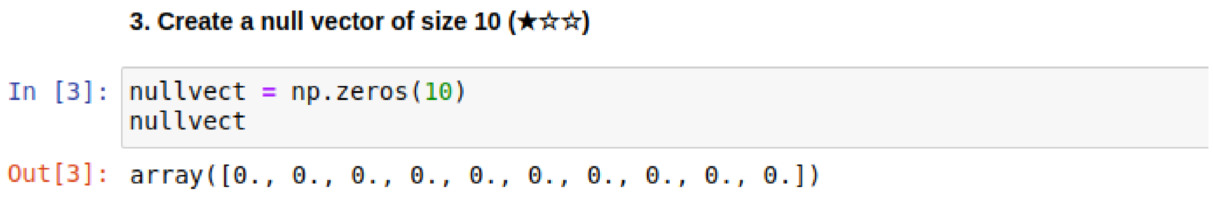
1. Import the numpy package under the name `np` (★☆☆)

```
In [1]: import numpy as np
```



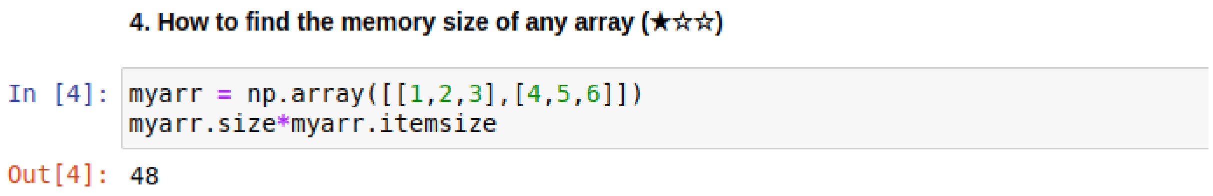
2. Print the numpy version and the configuration (★☆☆)

```
In [2]: print(np.__version__)
        np.show_config()

1.20.3
blas_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/ari-pt7127/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/ari-pt7127/anaconda3/include']
blas_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/ari-pt7127/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/ari-pt7127/anaconda3/include']
lapack_mkl_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/ari-pt7127/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/ari-pt7127/anaconda3/include']
lapack_opt_info:
    libraries = ['mkl_rt', 'pthread']
    library_dirs = ['/home/ari-pt7127/anaconda3/lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['/home/ari-pt7127/anaconda3/include']
```



3. Create a null vector of size 10 (★☆☆)

```
In [3]: nullvect = np.zeros(10)
        nullvect

Out[3]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```



4. How to find the memory size of any array (★☆☆)

```
In [4]: myarr = np.array([[1,2,3],[4,5,6]])
        myarr.size*myarr.itemsize

Out[4]: 48
```

5. How to get the documentation of the numpy add function from the command line? (★☆☆)

```
In [8]: np.info(np.add)
```

```
add(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extob
j])

Add arguments element-wise.

Parameters
----------
x1, x2 : array_like
    The arrays to be added.
    If ``x1.shape != x2.shape``, they must be broadcastable to a common
    shape (which becomes the shape of the output).
out : ndarray, None, or tuple of ndarray and None, optional
    A location into which the result is stored. If provided, it must have
    a shape that the inputs broadcast to. If not provided or None,
    a freshly-allocated array is returned. A tuple (possible only as a
    keyword argument) must have length equal to the number of outputs.
where : array_like, optional
    This condition is broadcast over the input. At locations where the
    condition is True, the `out` array will be set to the ufunc result.
    Elsewhere, the `out` array will retain its original value.
    Note that if an uninitialized `out` array is created via the default
    ``out=None``, locations within it where the condition is False will
    remain uninitialized.
**kwargs
    For other keyword-only arguments, see the
    :ref:`ufunc docs <ufuncs.kwargs>`.
```

6. Create a null vector of size 10 but the fifth value which is 1 (★☆☆)

```
In [9]: nullvect = np.zeros(10)
        nullvect[4] = 1
        nullvect

Out[9]: array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.])
```

7. Create a vector with values ranging from 10 to 49 (★☆☆)

```
In [8]: myvect = np.arange(10,50)
        myvect

Out[8]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
               27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
               44, 45, 46, 47, 48, 49])
```

8. Reverse a vector (first element becomes last) (★☆☆)

```
In [9]: myvect[::-1]

Out[9]: array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33,
               32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,
               15, 14, 13, 12, 11, 10])
```

9. Create a 3x3 matrix with values ranging from 0 to 8 (★☆☆)

```
In [10]: mymat = np.arange(0,9)
         mymat = mymat.reshape(3,3)
         mymat

Out[10]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

10. Find indices of non-zero elements from [1,2,0,0,4,0] (★☆☆)

```
In [11]: myarr = np.array([1,2,0,0,4,0])
         nonz = np.nonzero(myarr)
         nonz

Out[11]: (array([0, 1, 4]),)
```

11. Create a 3x3 identity matrix (★☆☆)

```
In [12]: idmat = np.eye(3)
         idmat

Out[12]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

12. Create a 3x3x3 array with random values (★☆☆)

```
In [10]: randarr = np.random.random((3,3,3))
         randarr

Out[10]: array([[[0.05578979, 0.32794572, 0.01761488],
                 [0.70742114, 0.24727322, 0.97844211],
                 [0.04934766, 0.0856386 , 0.136641  ]],

                [[0.77113415, 0.99966401, 0.5422891 ],
                 [0.85847872, 0.47312571, 0.55740461],
                 [0.42989752, 0.05003293, 0.98535541]],

                [[0.09107777, 0.22242871, 0.78129522],
                 [0.75217738, 0.52572714, 0.12412493],
                 [0.0295307 , 0.7999364 , 0.4161922 ]]])
```

13. Create a 10x10 array with random values and find the minimum and maximum values (★☆☆)

```
In [14]: myarr = np.random.random((10,10))
         print(myarr.min())
         print(myarr.max())

         0.0067507690905814766
         0.9809614051973846
```

14. Create a random vector of size 30 and find the mean value (★☆☆)

```
In [15]: ranvect = np.random.random(30)
         ranvect.mean()
```

```
Out[15]: 0.4073178961732454
```

15. Create a 2d array with 1 on the border and 0 inside (★☆☆)

```
In [16]: myarr = np.ones((5,5))
         myarr[1:-1,1:-1] = 0
         myarr
```

```
Out[16]: array([[1., 1., 1., 1., 1.],
                [1., 0., 0., 0., 1.],
                [1., 0., 0., 0., 1.],
                [1., 0., 0., 0., 1.],
                [1., 1., 1., 1., 1.]])
```

16. How to add a border (filled with 0's) around an existing array? (★☆☆)

```
In [17]: myarray = np.random.random((3,3))
         myarray = np.pad(myarray, pad_width=1, mode='constant', constant_values=0)
         myarray
```

```
Out[17]: array([[0.        , 0.        , 0.        , 0.        , 0.        ],
                [0.        , 0.09539762, 0.01429146, 0.72936927, 0.        ],
                [0.        , 0.6742889 , 0.84188886, 0.06824863, 0.        ],
                [0.        , 0.20348361, 0.09096531, 0.91253038, 0.        ],
                [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

17. What is the result of the following expression? (★☆☆)

```
0 * np.nan
np.nan == np.nan
np.inf > np.nan
np.nan - np.nan
np.nan in set([np.nan])
0.3 == 3 * 0.1
```

In [18]:

```
print(0*np.nan)
print(np.nan==np.nan)
print(np.inf>np.nan)
print(np.nan-np.nan)
print(np.nan in set([np.nan]))
print(0.3 == 0.1+0.2)
```

```
nan
False
False
nan
True
False
```

18. Create a 5x5 matrix with values 1,2,3,4 just below the diagonal (★☆☆)

```
In [19]: np.diag(1+np.arange(4),k=-1)

Out[19]: array([[0, 0, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [0, 2, 0, 0, 0],
                [0, 0, 3, 0, 0],
                [0, 0, 0, 4, 0]])
```

19. Create a 8x8 matrix and fill it with a checkerboard pattern (★☆☆)

```
In [20]: myarr = np.zeros((8,8),dtype=int)
         myarr[1::2,::2] = 1
         myarr[::2,1::2] = 1
         myarr

Out[20]: array([[0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0]])
```

20. Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th element? (★☆☆)

```
In [37]: np.unravel_index(99,(6,7,8))

Out[37]: (1, 5, 3)
```

21. Create a checkerboard 8x8 matrix using the tile function (★☆☆)

```
In [22]: myarr = np.array([[1,0],[0,1]])
         myarr = np.tile(myarr,(4,4))
         myarr

Out[22]: array([[1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1]])
```

22. Normalize a 5x5 random matrix (★☆☆)

```
In [23]: myarr = np.random.random((4,4))
         myarr = (myarr-np.mean(myarr))/np.std(myarr)
         myarr
```

```
Out[23]: array([[ 0.06995102,  0.05721187, -1.17331399, -0.06746793],
                [-0.04019315,  0.30403673,  1.17702591, -1.17283297],
                [-1.73134272,  1.5029312 , -1.51271432,  0.68307921],
                [ 0.37405071,  1.70767034,  0.49081313, -0.66890503]])
```

23. Create a custom dtype that describes a color as four unsigned bytes (RGBA) (★☆☆)

```
In [24]: color = np.dtype([("R", np.ubyte),("G", np.ubyte),("B", np.ubyte),("A", np.ubyte)])
         red = np.array([(255,0,0,0)],dtype=color)
         print(red)
```

```
         [(255, 0, 0, 0)]
```

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

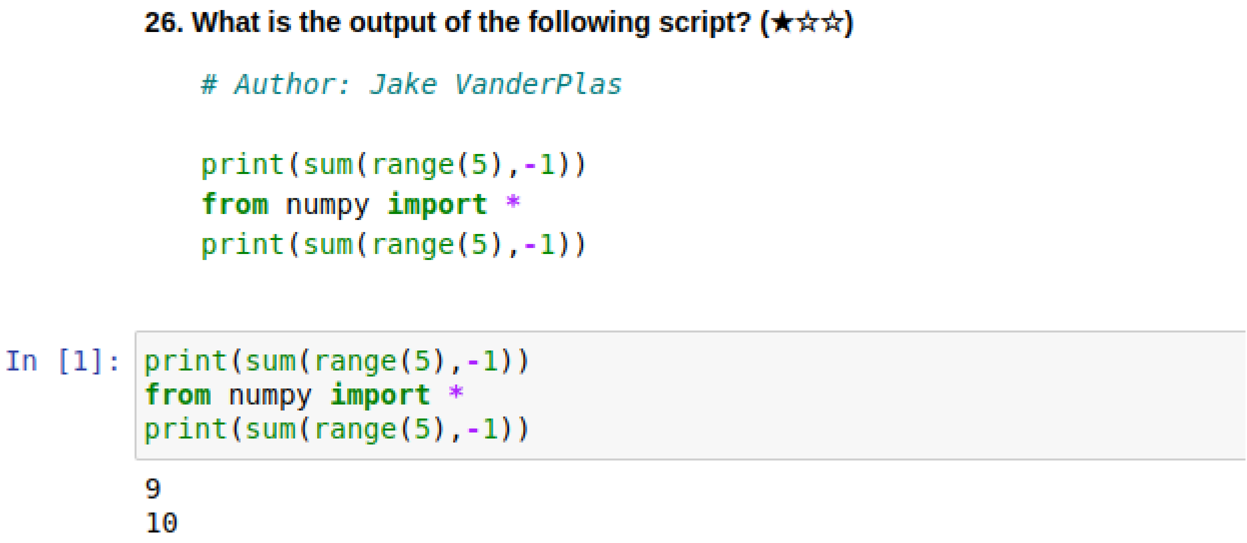
```
In [25]: mymat1 = np.ones((5,3))
         mymat2 = np.ones((3,2))
         np.dot(mymat1,mymat2)                                       # mymat1 @ mymat2 ( Above Python 3.5 )
```

```
Out[25]: array([[3., 3.],
                [3., 3.],
                [3., 3.],
                [3., 3.],
                [3., 3.]])
```

25. Given a 1D array, negate all elements which are between 3 and 8, in place. (★☆☆)

```
In [26]: myarr = np.arange(1,11,dtype=int)
         myarr[(myarr<8)&(myarr>3)]*=-1
         myarr
```

```
Out[26]: array([ 1,  2,  3, -4, -5, -6, -7,  8,  9, 10])
```

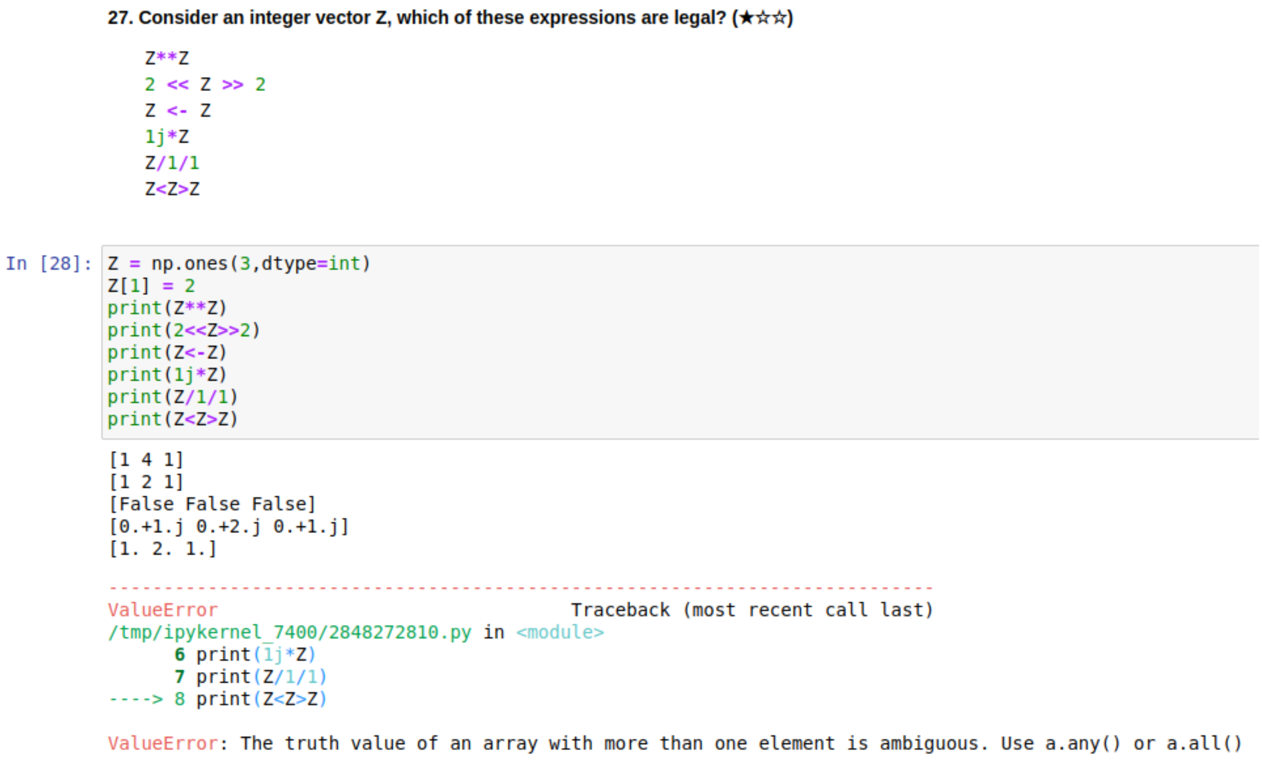


26. What is the output of the following script? (★☆☆)

```
# Author: Jake VanderPlas

print(sum(range(5),-1))
from numpy import *
print(sum(range(5),-1))
```

```
In [1]: print(sum(range(5),-1))
        from numpy import *
        print(sum(range(5),-1))
```

```
9
10
```



27. Consider an integer vector Z, which of these expressions are legal? (★☆☆)

```
Z**Z
2 << Z >> 2
Z <- Z
1j*Z
Z/1/1
Z<Z>Z
```

```
In [28]: Z = np.ones(3,dtype=int)
         Z[1] = 2
         print(Z**Z)
         print(2<<Z>>2)
         print(Z<-Z)
         print(1j*Z)
         print(Z/1/1)
         print(Z<Z>Z)
```

```
[1 4 1]
[1 2 1]
[False False False]
[0.+1.j 0.+2.j 0.+1.j]
[1. 2. 1.]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_7400/2848272810.py in <module>
      6 print(1j*Z)
      7 print(Z/1/1)
----> 8 print(Z<Z>Z)

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

28. What are the result of the following expressions? (★☆☆)

```
np.array(0) / np.array(0)
np.array(0) // np.array(0)
np.array([np.nan]).astype(int).astype(float)
```

```
In [29]: print(np.array(0) / np.array(0))
         print(np.array(0) // np.array(0))
         print(np.array([np.nan]).astype(int).astype(float))
```

```
nan
0
[-9.22337204e+18]
```

29. How to round away from zero a float array ? (★☆☆)

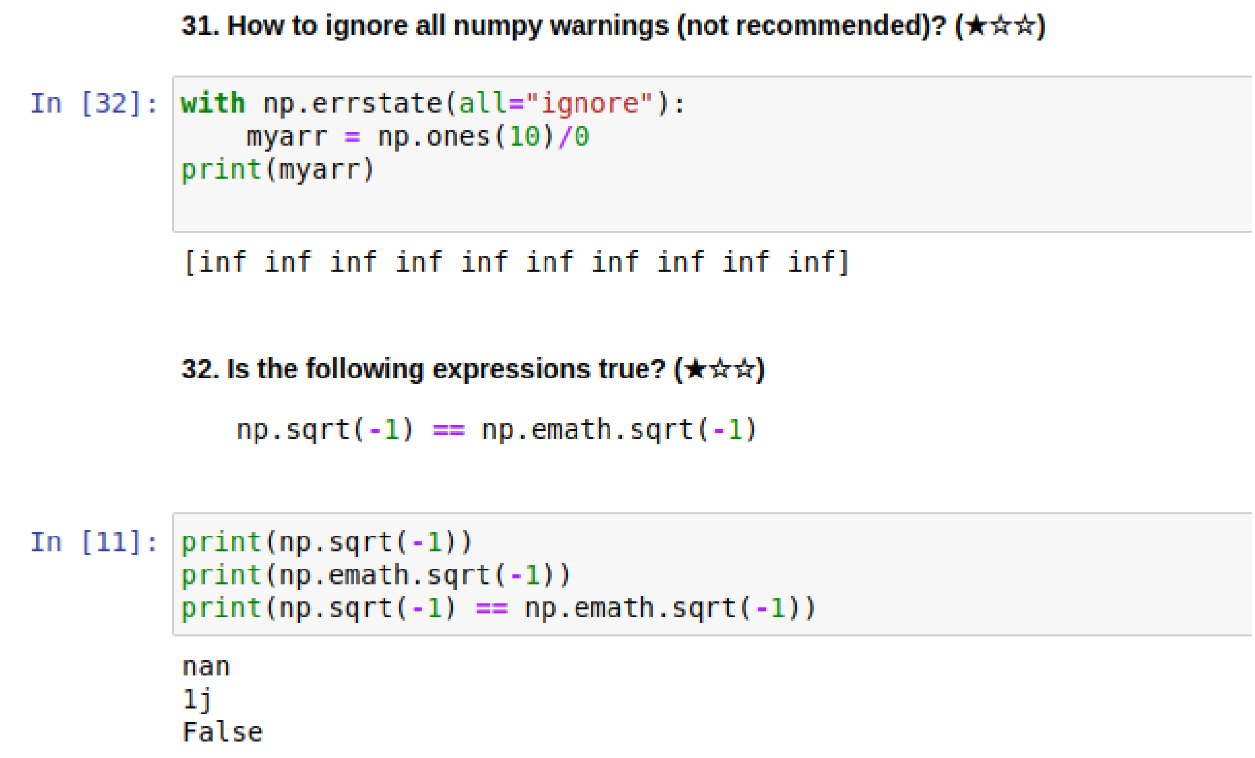
```
In [35]: array = np.array([1.01,2.34,-0.21,-0.234,0.45])
         print(np.where(array>0, np.ceil(array), np.floor(array)))
```

```
[ 2.  3. -1. -1.  1.]
```

30. How to find common values between two arrays? (★☆☆)

```
In [31]: arr1 = np.arange(1,10)
         arr2 = np.arange(5,15)
         np.intersect1d(arr1,arr2)
```

```
Out[31]: array([5, 6, 7, 8, 9])
```



31. How to ignore all numpy warnings (not recommended)? (★☆☆)

In [32]:
```
with np.errstate(all="ignore"):
    myarr = np.ones(10)/0
print(myarr)
```

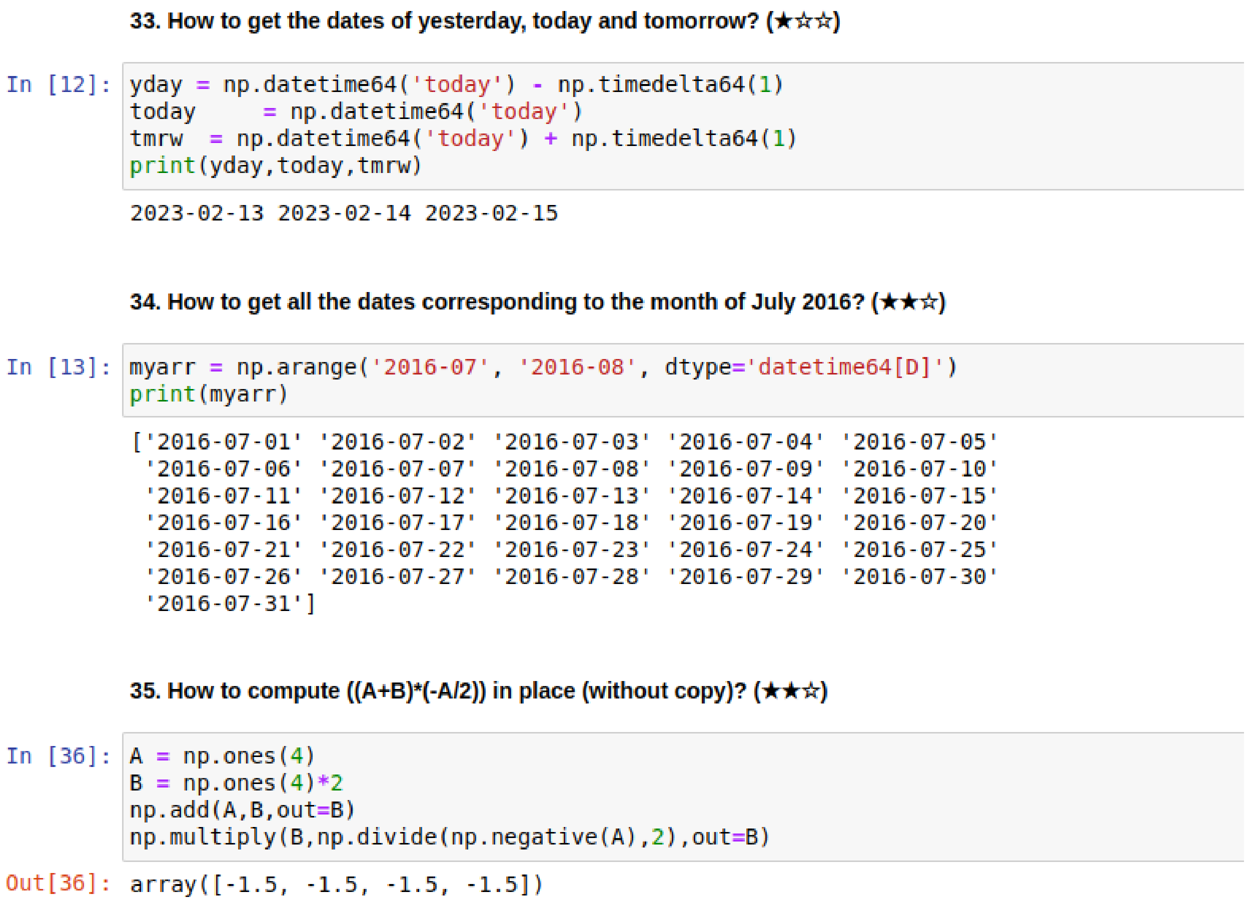
```
[inf inf inf inf inf inf inf inf inf inf]
```

32. Is the following expressions true? (★☆☆)

```
np.sqrt(-1) == np.emath.sqrt(-1)
```

In [11]:
```
print(np.sqrt(-1))
print(np.emath.sqrt(-1))
print(np.sqrt(-1) == np.emath.sqrt(-1))
```

```
nan
1j
False
```



33. How to get the dates of yesterday, today and tomorrow? (★☆☆)

In [12]:
```
yday = np.datetime64('today') - np.timedelta64(1)
today     = np.datetime64('today')
tmrw  = np.datetime64('today') + np.timedelta64(1)
print(yday,today,tmrw)
```

```
2023-02-13 2023-02-14 2023-02-15
```

34. How to get all the dates corresponding to the month of July 2016? (★★☆)

In [13]:
```
myarr = np.arange('2016-07', '2016-08', dtype='datetime64[D]')
print(myarr)
```

```
['2016-07-01' '2016-07-02' '2016-07-03' '2016-07-04' '2016-07-05'
 '2016-07-06' '2016-07-07' '2016-07-08' '2016-07-09' '2016-07-10'
 '2016-07-11' '2016-07-12' '2016-07-13' '2016-07-14' '2016-07-15'
 '2016-07-16' '2016-07-17' '2016-07-18' '2016-07-19' '2016-07-20'
 '2016-07-21' '2016-07-22' '2016-07-23' '2016-07-24' '2016-07-25'
 '2016-07-26' '2016-07-27' '2016-07-28' '2016-07-29' '2016-07-30'
 '2016-07-31']
```

35. How to compute ((A+B)*(-A/2)) in place (without copy)? (★★☆)

In [36]:
```
A = np.ones(4)
B = np.ones(4)*2
np.add(A,B,out=B)
np.multiply(B,np.divide(np.negative(A),2),out=B)
```

Out[36]:
```
array([-1.5, -1.5, -1.5, -1.5])
```

36. Extract the integer part of a random array of positive numbers using 4 different methods (★★☆)

In [14]:

```
myarr = np.random.random(5)
print(np.array(myarr,dtype=int))
print(np.floor(myarr))
print(myarr.astype(int))
print(np.trunc(myarr))
```

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

37. Create a 5x5 matrix with row values ranging from 0 to 4 (★★☆)

In [39]:

```
myarr = np.zeros((5,5))
myarr += np.arange(5)
print(myarr)
```

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

38. Consider a generator function that generates 10 integers and use it to build an array (★☆☆)

In [40]:

```
def genfunc():
    for x in range(10):
        yield x
myarr = np.fromiter(genfunc(),dtype=int)
print(myarr)
```

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

39. Create a vector of size 10 with values ranging from 0 to 1, both excluded (★★☆)

In [25]:

```
myarr = np.linspace(0,1,10,endpoint=False)
print(myarr)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

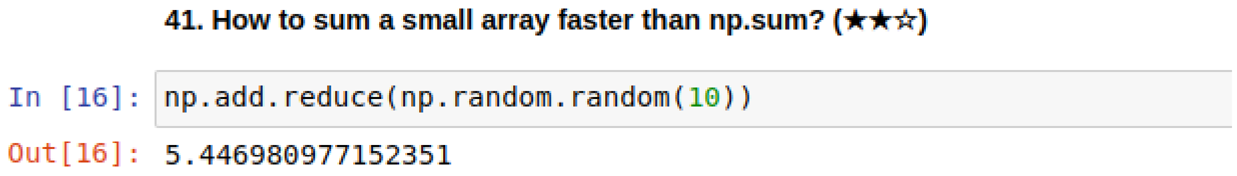
40. Create a random vector of size 10 and sort it (★★☆)

In [42]:

```
np.sort(np.random.random(10))
```

Out[42]:

```
array([0.05165707, 0.11506384, 0.307378  , 0.3622006 , 0.38890519,
       0.58033998, 0.67142848, 0.69136779, 0.93926682, 0.95727387])
```



41. How to sum a small array faster than np.sum? (★★☆)

In [16]:

```
np.add.reduce(np.random.random(10))
```

Out[16]:

```
5.446980977152351
```

42. Consider two random array A and B, check if they are equal (★★☆)

In [26]:

```
A = np.array(10)
B = np.random.random(10)
print(np.array_equal(A,B))
```

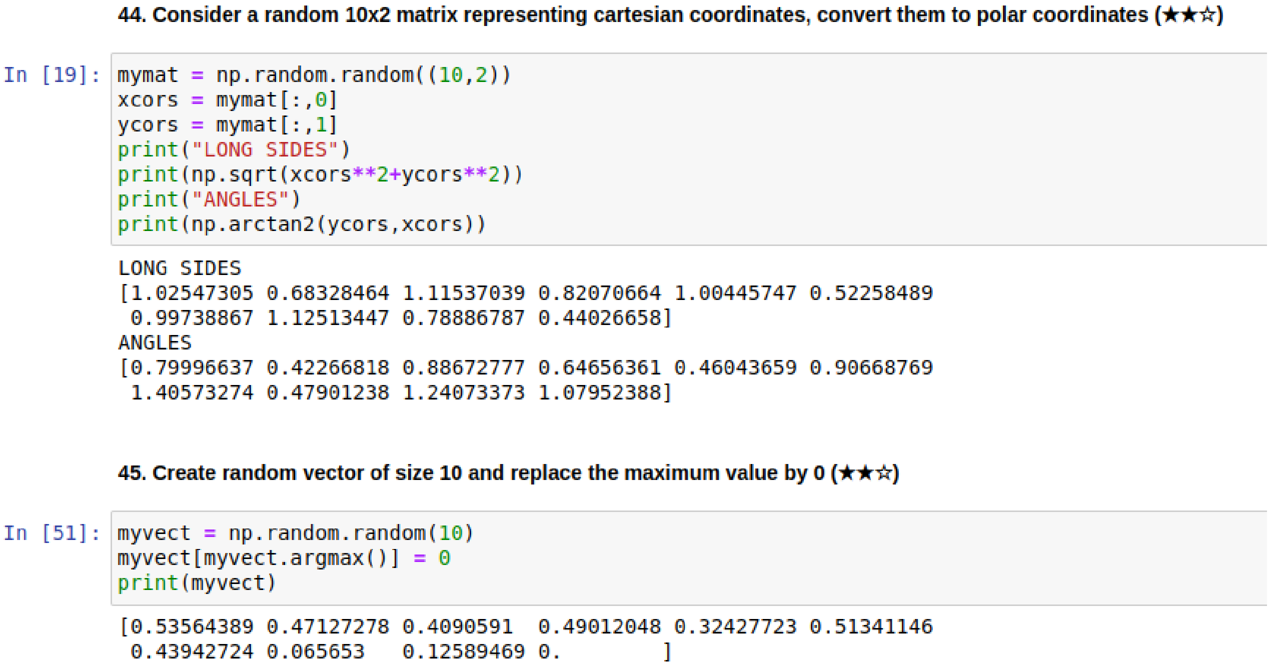
```
False
```

43. Make an array immutable (read-only) (★★☆)

In [18]:

```
A = np.array([1,2,3])
A[0]=1
print("DONE")
A.flags.writeable = False
A[0]=1
```

```
DONE

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_87602/202326665.py in <module>
      3 print("DONE")
      4 A.flags.writeable = False
----> 5 A[0]=1

ValueError: assignment destination is read-only
```



44. Consider a random 10x2 matrix representing cartesian coordinates, convert them to polar coordinates (★★☆)

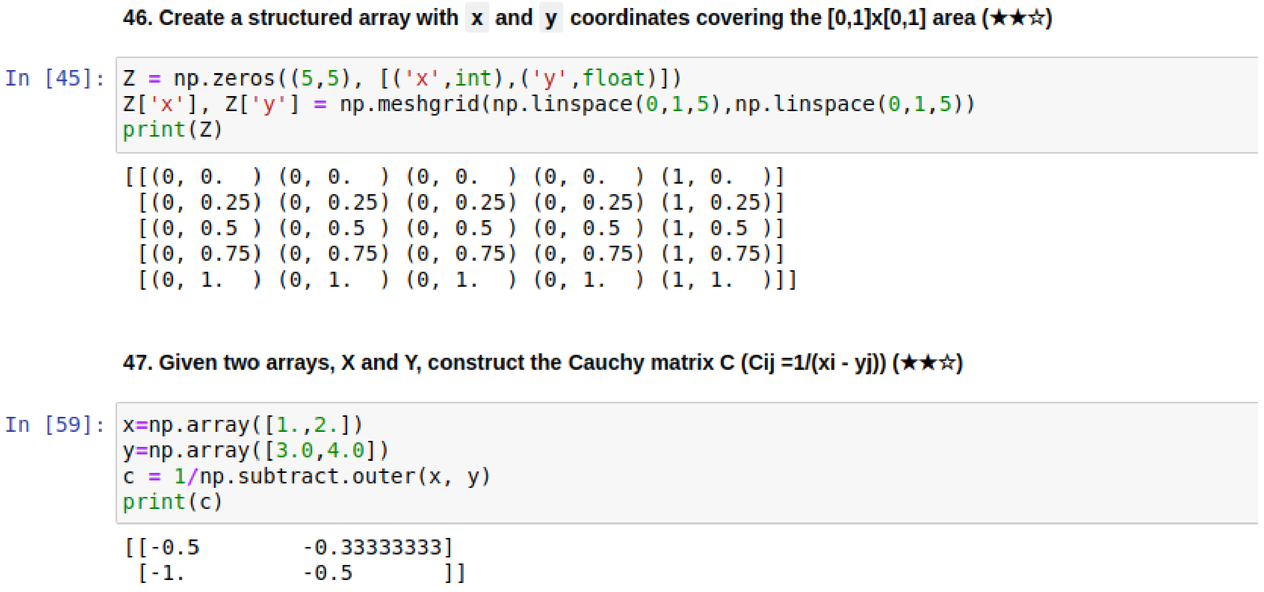
```
In [19]: mymat = np.random.random((10,2))
         xcors = mymat[:,0]
         ycors = mymat[:,1]
         print("LONG SIDES")
         print(np.sqrt(xcors**2+ycors**2))
         print("ANGLES")
         print(np.arctan2(ycors,xcors))
```

```
LONG SIDES
[1.02547305 0.68328464 1.11537039 0.82070664 1.00445747 0.52258489
 0.99738867 1.12513447 0.78886787 0.44026658]
ANGLES
[0.79996637 0.42266818 0.88672777 0.64656361 0.46043659 0.90668769
 1.40573274 0.47901238 1.24073373 1.07952388]
```

45. Create random vector of size 10 and replace the maximum value by 0 (★★☆)

```
In [51]: myvect = np.random.random(10)
         myvect[myvect.argmax()] = 0
         print(myvect)
```

```
[0.53564389 0.47127278 0.4090591  0.49012048 0.32427723 0.51341146
 0.43942724 0.065653   0.12589469 0.        ]
```



46. Create a structured array with `x` and `y` coordinates covering the [0,1]x[0,1] area (★★☆)

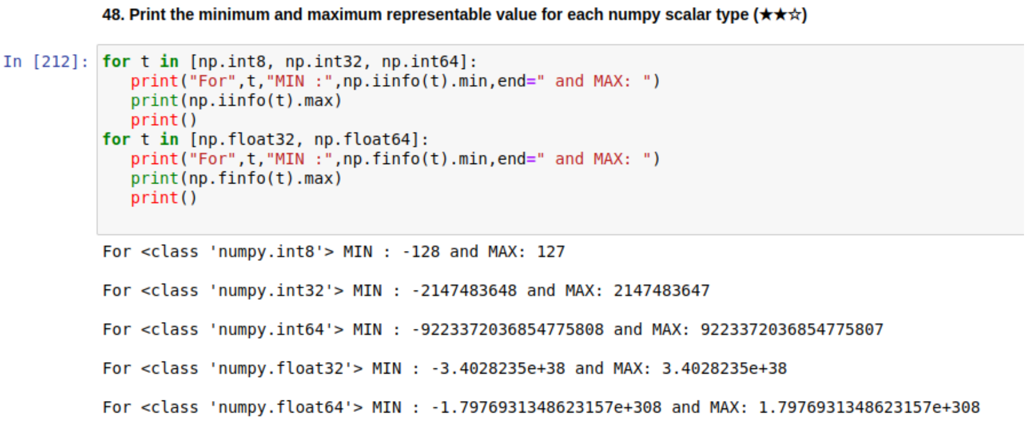
```
In [45]: Z = np.zeros((5,5), [('x',int),('y',float)])
         Z['x'], Z['y'] = np.meshgrid(np.linspace(0,1,5),np.linspace(0,1,5))
         print(Z)
```

```
[[(0, 0.  ) (0, 0.  ) (0, 0.  ) (0, 0.  ) (1, 0.  )]
 [(0, 0.25) (0, 0.25) (0, 0.25) (0, 0.25) (1, 0.25)]
 [(0, 0.5 ) (0, 0.5 ) (0, 0.5 ) (0, 0.5 ) (1, 0.5 )]
 [(0, 0.75) (0, 0.75) (0, 0.75) (0, 0.75) (1, 0.75)]
 [(0, 1.  ) (0, 1.  ) (0, 1.  ) (0, 1.  ) (1, 1.  )]]
```

47. Given two arrays, X and Y, construct the Cauchy matrix C (Cij =1/(xi - yj)) (★★☆)

```
In [59]: x=np.array([1.,2.])
         y=np.array([3.0,4.0])
         c = 1/np.subtract.outer(x, y)
         print(c)
```

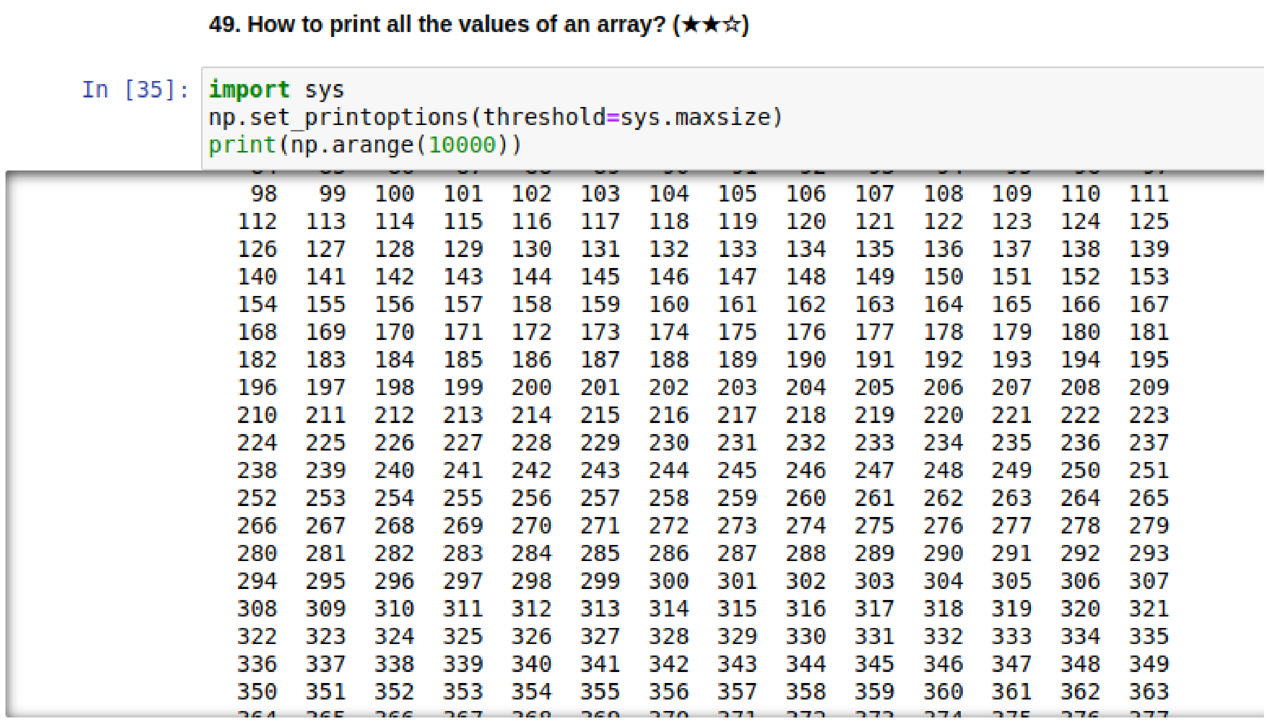
```
[[-0.5        -0.33333333]
 [-1.         -0.5       ]]
```



48. Print the minimum and maximum representable value for each numpy scalar type (★★☆)

```
In [212]: for t in [np.int8, np.int32, np.int64]:
              print("For",t,"MIN :",np.iinfo(t).min,end=" and MAX: ")
              print(np.iinfo(t).max)
              print()
          for t in [np.float32, np.float64]:
              print("For",t,"MIN :",np.finfo(t).min,end=" and MAX: ")
              print(np.finfo(t).max)
              print()
```

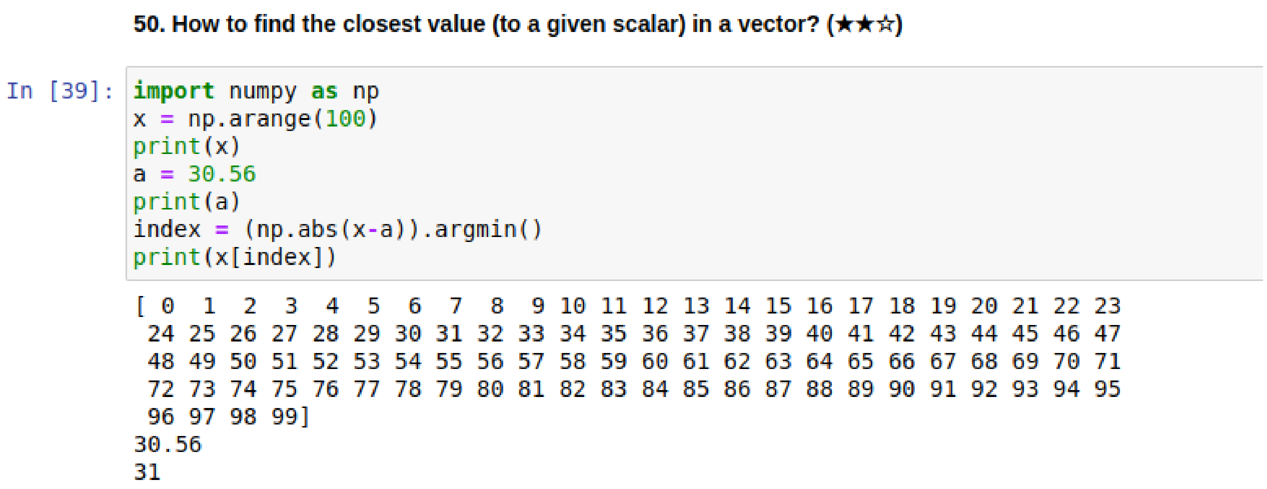
```
For <class 'numpy.int8'> MIN : -128 and MAX: 127

For <class 'numpy.int32'> MIN : -2147483648 and MAX: 2147483647

For <class 'numpy.int64'> MIN : -9223372036854775808 and MAX: 9223372036854775807

For <class 'numpy.float32'> MIN : -3.4028235e+38 and MAX: 3.4028235e+38

For <class 'numpy.float64'> MIN : -1.7976931348623157e+308 and MAX: 1.7976931348623157e+308
```



49. How to print all the values of an array? (★★☆)

In [35]:
```
import sys
np.set_printoptions(threshold=sys.maxsize)
print(np.arange(10000))
```

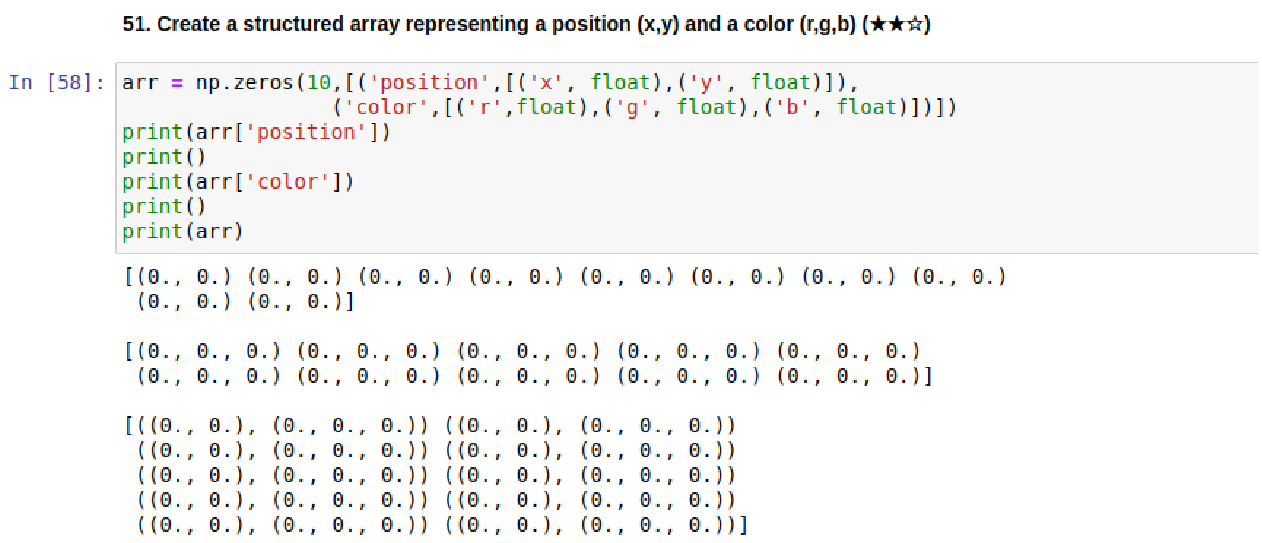
```
   98   99  100  101  102  103  104  105  106  107  108  109  110  111
  112  113  114  115  116  117  118  119  120  121  122  123  124  125
  126  127  128  129  130  131  132  133  134  135  136  137  138  139
  140  141  142  143  144  145  146  147  148  149  150  151  152  153
  154  155  156  157  158  159  160  161  162  163  164  165  166  167
  168  169  170  171  172  173  174  175  176  177  178  179  180  181
  182  183  184  185  186  187  188  189  190  191  192  193  194  195
  196  197  198  199  200  201  202  203  204  205  206  207  208  209
  210  211  212  213  214  215  216  217  218  219  220  221  222  223
  224  225  226  227  228  229  230  231  232  233  234  235  236  237
  238  239  240  241  242  243  244  245  246  247  248  249  250  251
  252  253  254  255  256  257  258  259  260  261  262  263  264  265
  266  267  268  269  270  271  272  273  274  275  276  277  278  279
  280  281  282  283  284  285  286  287  288  289  290  291  292  293
  294  295  296  297  298  299  300  301  302  303  304  305  306  307
  308  309  310  311  312  313  314  315  316  317  318  319  320  321
  322  323  324  325  326  327  328  329  330  331  332  333  334  335
  336  337  338  339  340  341  342  343  344  345  346  347  348  349
  350  351  352  353  354  355  356  357  358  359  360  361  362  363
```



50. How to find the closest value (to a given scalar) in a vector? (★★☆)

In [39]:
```
import numpy as np
x = np.arange(100)
print(x)
a = 30.56
print(a)
index = (np.abs(x-a)).argmin()
print(x[index])
```

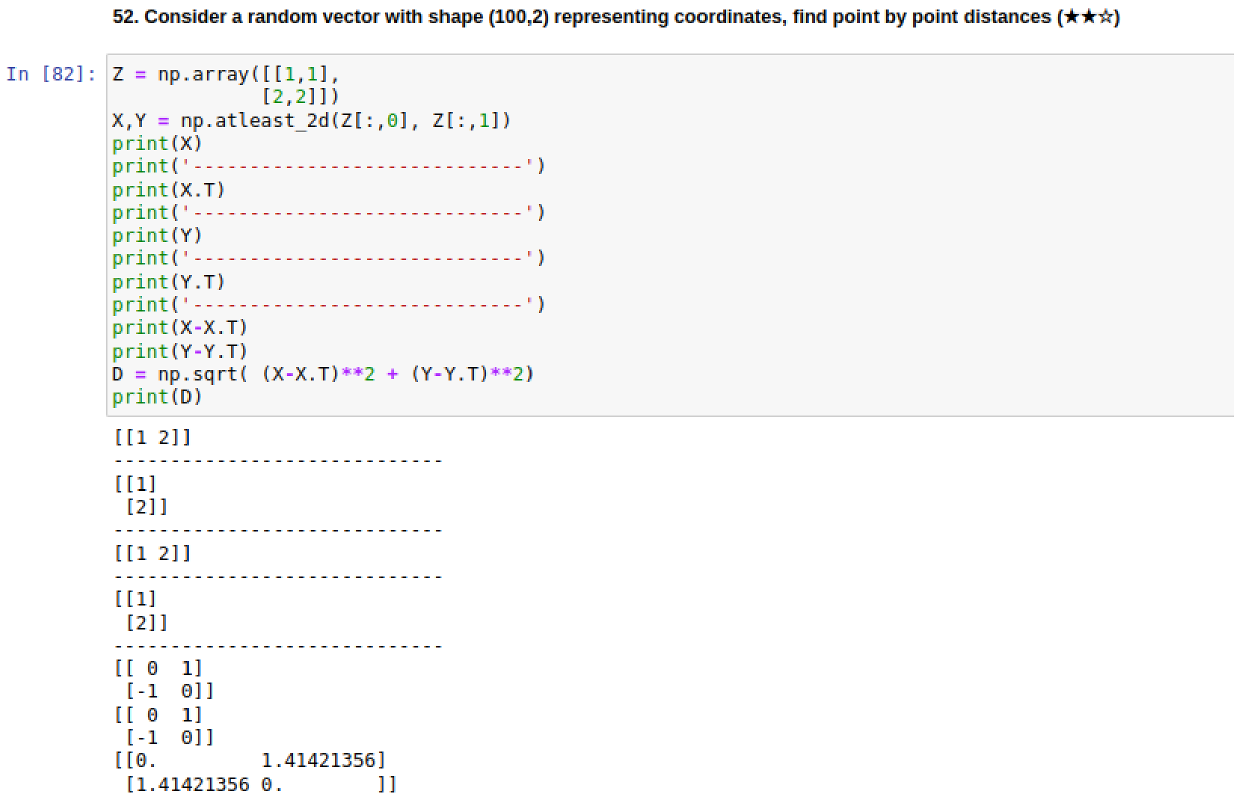
```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
30.56
31
```



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

In [58]:
```
arr = np.zeros(10,[('position',[('x', float),('y', float)]),
                  ('color',[('r',float),('g', float),('b', float)])])
print(arr['position'])
print()
print(arr['color'])
print()
print(arr)
```

```
[(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., 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.))
 ((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))]
```

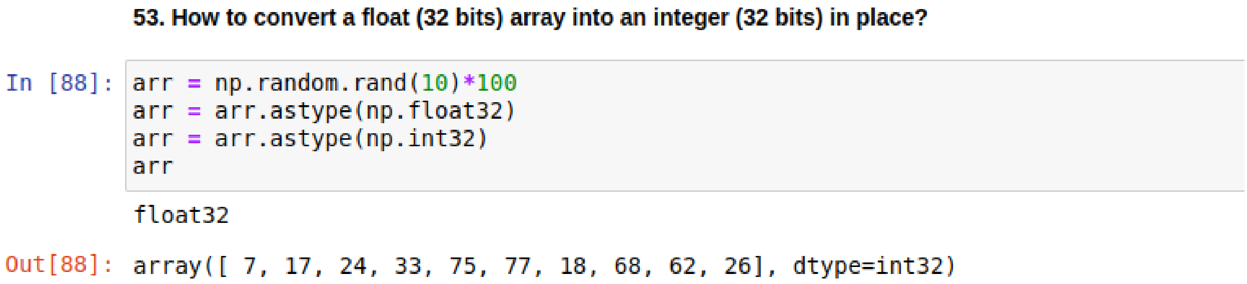


52. Consider a random vector with shape (100,2) representing coordinates, find point by point distances (★★☆)

In [82]:

```
Z = np.array([[1,1],
              [2,2]])
X,Y = np.atleast_2d(Z[:,0], Z[:,1])
print(X)
print('-----------------------------')
print(X.T)
print('-----------------------------')
print(Y)
print('-----------------------------')
print(Y.T)
print('-----------------------------')
print(X-X.T)
print(Y-Y.T)
D = np.sqrt( (X-X.T)**2 + (Y-Y.T)**2)
print(D)
```

```
[[1 2]]
-----------------------------
[[1]
 [2]]
-----------------------------
[[1 2]]
-----------------------------
[[1]
 [2]]
-----------------------------
[[ 0  1]
 [-1  0]]
[[ 0  1]
 [-1  0]]
[[0.         1.41421356]
 [1.41421356 0.        ]]
```



53. How to convert a float (32 bits) array into an integer (32 bits) in place?

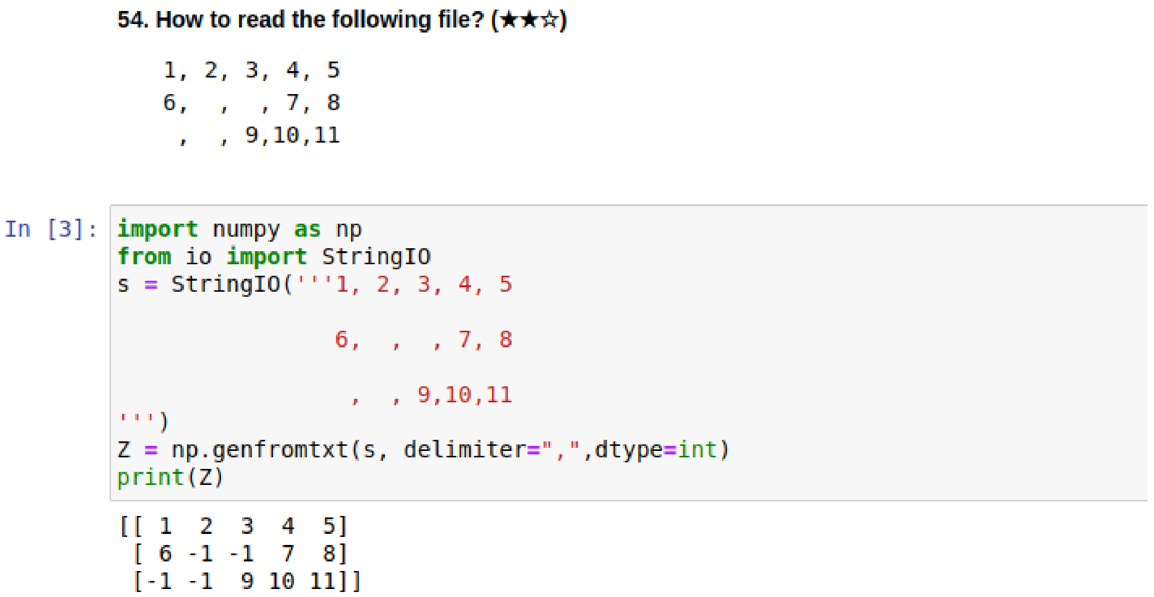
In [88]:

```
arr = np.random.rand(10)*100
arr = arr.astype(np.float32)
arr = arr.astype(np.int32)
arr
```

```
float32
```

Out[88]:

```
array([ 7, 17, 24, 33, 75, 77, 18, 68, 62, 26], dtype=int32)
```



54. How to read the following file? (★★☆)

```
1, 2, 3, 4, 5
6,  ,  , 7, 8
 ,  , 9,10,11
```

In [3]:

```
import numpy as np
from io import StringIO
s = StringIO('''1, 2, 3, 4, 5

                6,  ,  , 7, 8

                 ,  , 9,10,11
''')
Z = np.genfromtxt(s, delimiter=",",dtype=int)
print(Z)
```

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

55. What is the equivalent of enumerate for numpy arrays? (★★☆)

In [5]:

```
arr = np.arange(12).reshape(4,3)
for i, v in np.ndenumerate(arr):
    print(i, v)
```

```
(0, 0) 0
(0, 1) 1
(0, 2) 2
(1, 0) 3
(1, 1) 4
(1, 2) 5
(2, 0) 6
(2, 1) 7
(2, 2) 8
(3, 0) 9
(3, 1) 10
(3, 2) 11
```

56. Generate a generic 2D Gaussian-like array (★★☆)

In [89]:

```
x, y = np.meshgrid(np.linspace(-1,1,10), np.linspace(-1,1,10))
d = np.sqrt(x*x+y*y)
sigma, mu = 1.0, 0.0
g = np.exp(-( (d-mu)**2 / ( 2.0 * sigma**2 ) ) )
print(g)
```

```
[[0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
  0.57375342 0.51979489 0.44822088 0.36787944]
 [0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
  0.69905581 0.63331324 0.54610814 0.44822088]
 [0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
  0.81068432 0.73444367 0.63331324 0.51979489]
 [0.57375342 0.69905581 0.81068432 0.89483932 0.9401382  0.9401382
  0.89483932 0.81068432 0.69905581 0.57375342]
 [0.60279818 0.73444367 0.85172308 0.9401382  0.98773022 0.98773022
  0.9401382  0.85172308 0.73444367 0.60279818]
 [0.60279818 0.73444367 0.85172308 0.9401382  0.98773022 0.98773022
  0.9401382  0.85172308 0.73444367 0.60279818]
 [0.57375342 0.69905581 0.81068432 0.89483932 0.9401382  0.9401382
  0.89483932 0.81068432 0.69905581 0.57375342]
 [0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
  0.81068432 0.73444367 0.63331324 0.51979489]
 [0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
  0.69905581 0.63331324 0.54610814 0.44822088]
 [0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
  0.57375342 0.51979489 0.44822088 0.36787944]]
```

57. How to randomly place p elements in a 2D array? (★★☆)

```
In [176]: p = 3
          arr = np.zeros((5,5))
          np.put(arr, np.random.choice(range(5*5), p),[2,3,4])
          print(arr)
```

```
[[0. 0. 0. 0. 3.]
 [0. 0. 0. 0. 0.]
 [0. 0. 4. 0. 0.]
 [0. 0. 0. 0. 0.]
 [2. 0. 0. 0. 0.]]
```

58. Subtract the mean of each row of a matrix (★★☆)

```
In [102]: X = np.array([[3,4],
                        [1,2]])
          X - X.mean(axis=1)
```

```
Out[102]: array([[-0.5,  2.5],
                 [-2.5,  0.5]])
```

59. How to sort an array by the nth column? (★★☆)

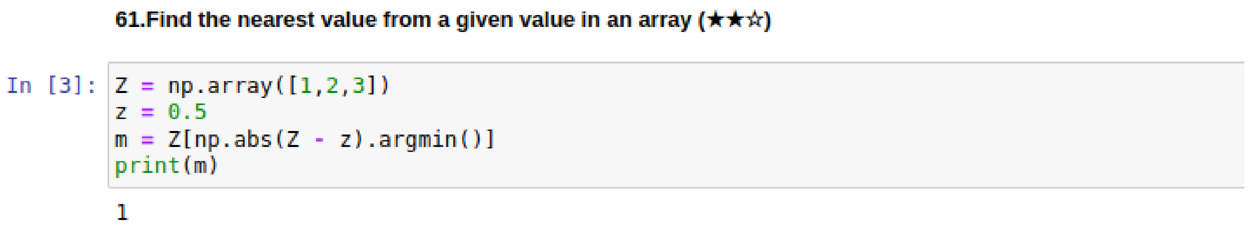
```
In [ ]: Z = np.random.randint(0,10,(3,3))
        print(Z)
        print('-----------------------------')
        col = int(input("ENTER : "))
        print(Z[(Z[:,col].argsort())])
```

```
[[0 2 2]
 [1 7 2]
 [3 3 3]]
-----------------------------
```

60. How to tell if a given 2D array has null columns? (★★☆)

```
In [112]: Z=np.array([
              [0,1,np.nan],
              [1,2,np.nan],
              [4,5,2]
          ])
          print(np.isnan(Z))
          print('-------------------------')
          print(np.isnan(Z).any(axis=0))
          print('-------------------------')
          print(np.isnan(Z).any(axis=0).any())
```

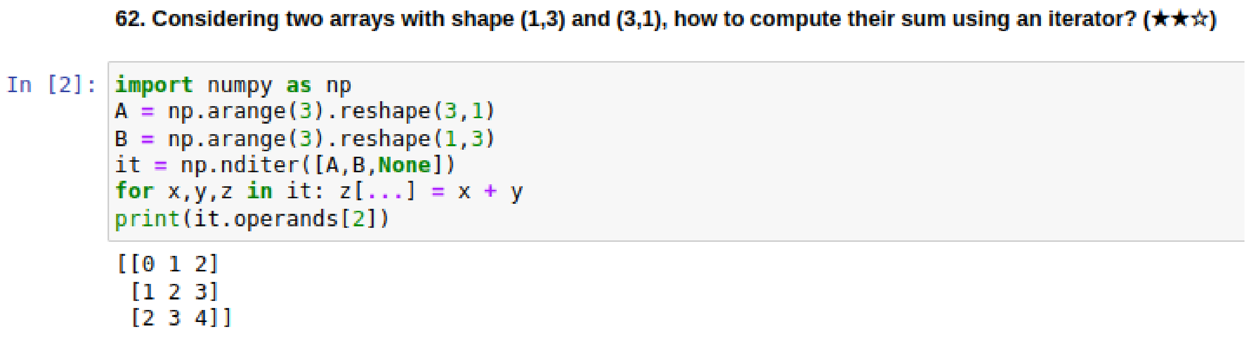
```
[[False False  True]
 [False False  True]
 [False False False]]
-------------------------
[False False  True]
-------------------------
True
```



61.Find the nearest value from a given value in an array (★★☆)

In [3]:
```
Z = np.array([1,2,3])
z = 0.5
m = Z[np.abs(Z - z).argmin()]
print(m)
```

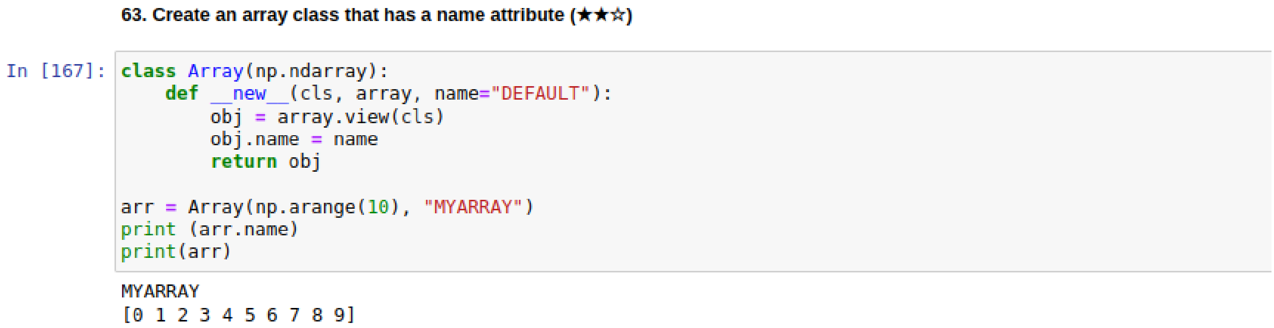
```
1
```



62. Considering two arrays with shape (1,3) and (3,1), how to compute their sum using an iterator? (★★☆)

In [2]:
```
import numpy as np
A = np.arange(3).reshape(3,1)
B = np.arange(3).reshape(1,3)
it = np.nditer([A,B,None])
for x,y,z in it: z[...] = x + y
print(it.operands[2])
```

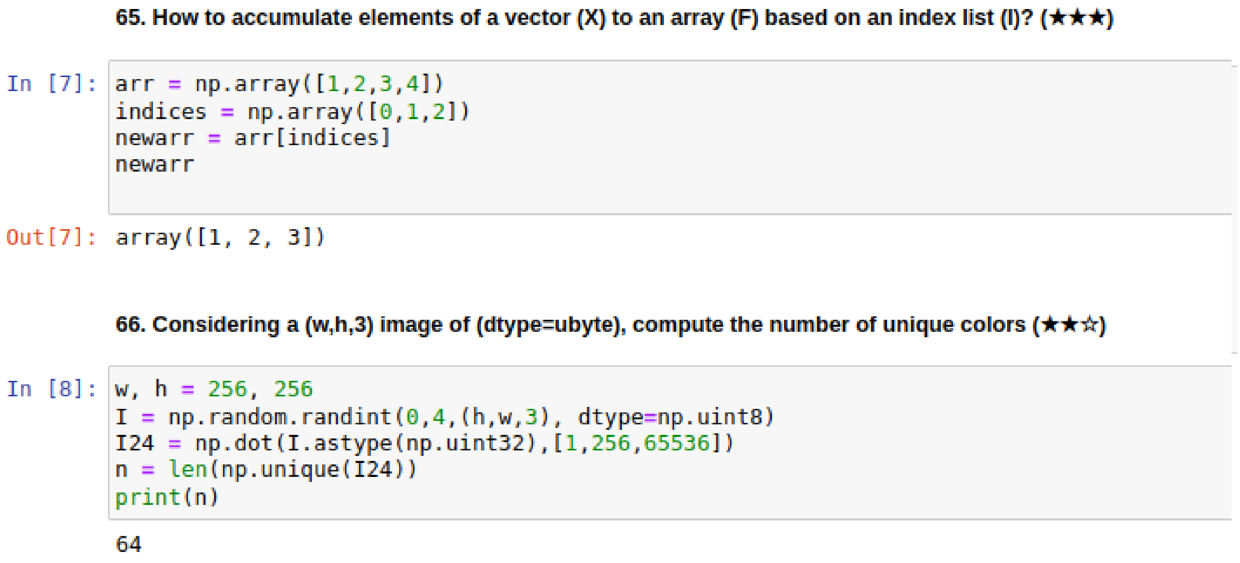
```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```



63. Create an array class that has a name attribute (★★☆)

In [167]:
```
class Array(np.ndarray):
    def __new__(cls, array, name="DEFAULT"):
        obj = array.view(cls)
        obj.name = name
        return obj

arr = Array(np.arange(10), "MYARRAY")
print (arr.name)
print(arr)
```

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



65. How to accumulate elements of a vector (X) to an array (F) based on an index list (I)? (★★★)

In [7]:
```
arr = np.array([1,2,3,4])
indices = np.array([0,1,2])
newarr = arr[indices]
newarr
```

Out[7]:
```
array([1, 2, 3])
```

66. Considering a (w,h,3) image of (dtype=ubyte), compute the number of unique colors (★★☆)

In [8]:
```
w, h = 256, 256
I = np.random.randint(0,4,(h,w,3), dtype=np.uint8)
I24 = np.dot(I.astype(np.uint32),[1,256,65536])
n = len(np.unique(I24))
print(n)
```

```
64
```

67. Considering a four dimensions array, how to get sum over the last two axis at once? (★★★)

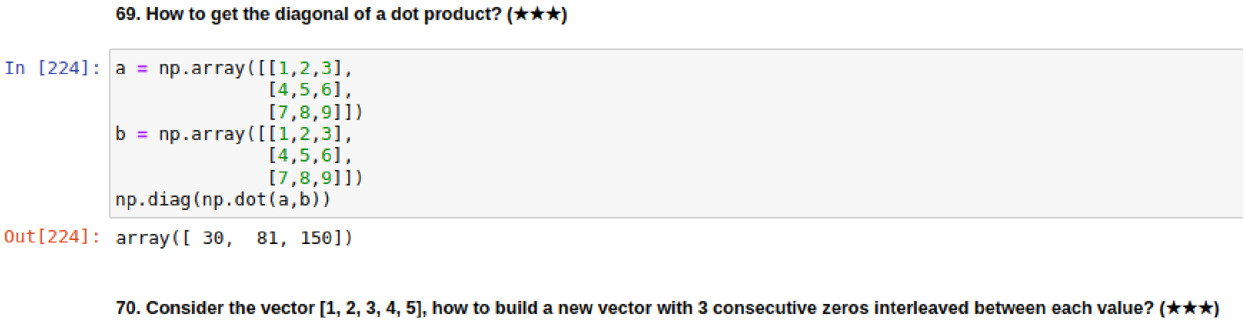
```
In [9]: A = np.random.randint(0,10,(3,4,3,4))
        # solution by passing a tuple of axes (introduced in numpy 1.7.0)
        sum = A.sum(axis=(-2,-1))
        print(sum)
```

```
[[59 56 50 40]
 [51 53 58 66]
 [55 66 47 65]]
```

68. Considering a one-dimensional vector D, how to compute means of subsets of D using a vector S of same size describing subset indices? (★★★)

```
In [22]: import numpy as np
         arr = np.array([1,2,3,4,5,6,7])
         indices = np.array([[1,0],[2,0]])
         for i in np.nditer(indices:
             print(np.sum(arr[i]))
```

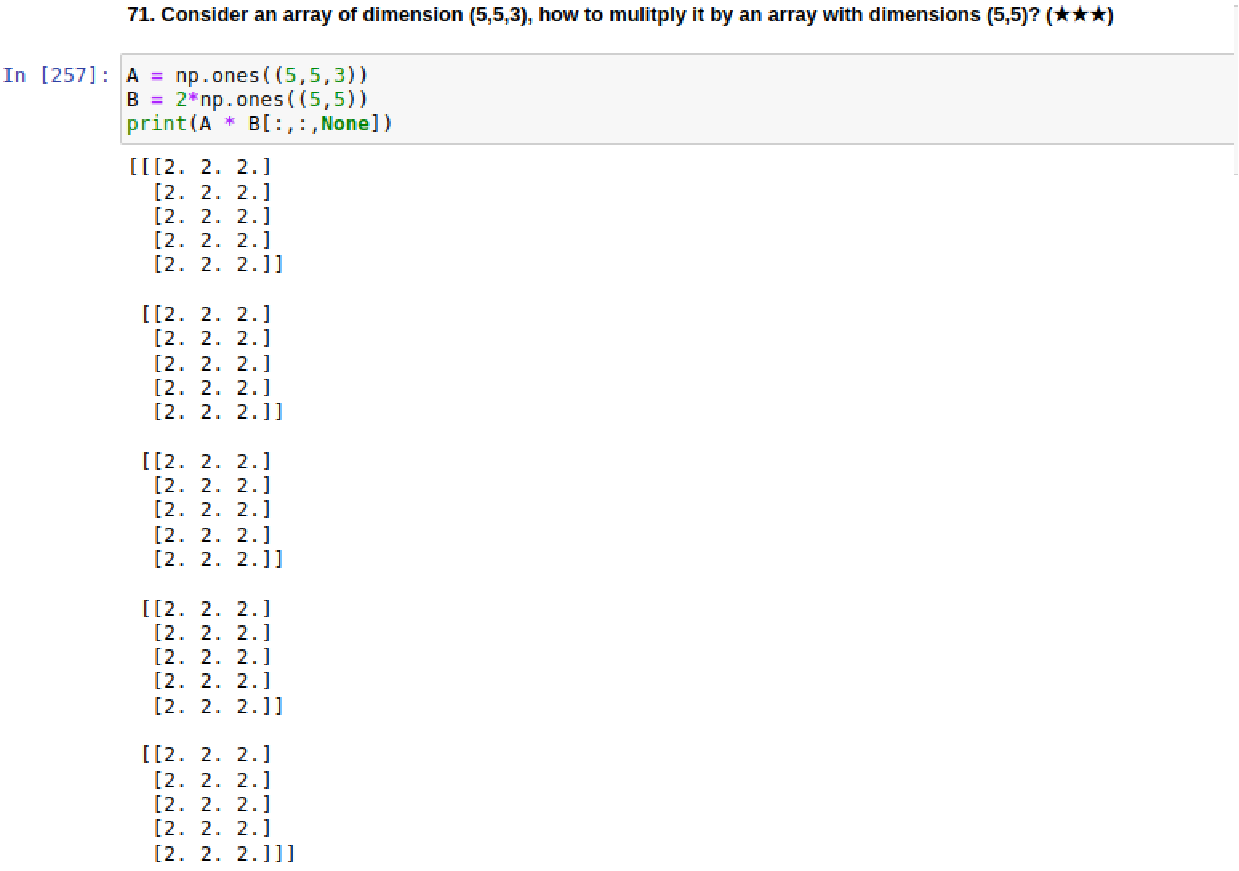
```
3
4
```



69. How to get the diagonal of a dot product? (★★★)

```
In [224]: a = np.array([[1,2,3],
                        [4,5,6],
                        [7,8,9]])
          b = np.array([[1,2,3],
                        [4,5,6],
                        [7,8,9]])
          np.diag(np.dot(a,b))

Out[224]: array([ 30,  81, 150])
```

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



71. Consider an array of dimension (5,5,3), how to mulitply it by an array with dimensions (5,5)? (★★★)

```
In [257]: A = np.ones((5,5,3))
          B = 2*np.ones((5,5))
          print(A * B[:,:,None])
```

```
[[[2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]]

 [[2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]]

 [[2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]]

 [[2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]]

 [[2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]
  [2. 2. 2.]]]
```

72. How to swap two rows of an array? (★★★)

In [275]:

```
arr = np.array([[1,2],
                [3,4]])
print('--------------------------')

print(arr[[0,1]])

print('--------------------------')

print(arr[[1,0]])

print('--------------------------')

arr[[0,1]] = arr[[1,0]]

print(arr)
```

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