The number of axes is rank.

For example, the coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis.

Numpy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

ndarray.ndim = num of axes ndarray.shape = dim of ndarray ndarray.size = total number of elements ndarray.itemsize = size in bytes of each array element ndarray.data = buffer containing the actual elements

```
In [4]: from numpy import *
```

```
In [5]: arange(15)
```

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

```
In [6]: arange(15).reshape(3,5)
```

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

```
In [8]: a = arange(15).reshape(3,5)
        a
```

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

```
In [9]: a.ndim
```

```
Out[9]: 2
```

```
In [10]: a.dtype
```

```
Out[10]: dtype('int64')
```

```
In [11]: a.dtype.name
```

```
Out[11]: 'int64'
```

```
In [12]: type(a)
```

```
Out[12]: numpy.ndarray
```

In [13]:
```
# A frequent error consists in calling array with multiple numeric arguments
, rather than providing
# A single list of numbers as an argument.


# Wrong
a = array(1,2,3,4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-037d3b7121ad> in <module>()
      3 
      4 # Wrong
----> 5 a = array(1,2,3,4)

ValueError: only 2 non-keyword arguments accepted
```

In [14]:
```
# Right
a = array([1,2,3,4])
```

Array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

In [16]:
```
# Type of array can be explicitly specified during creation
c = array ([(1,2),(3,4)], dtype=complex)
```

In [17]:
```
c
```

Out[17]:
```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

In [19]:
```
# Examples of array creation
# By default all arrays are float64
zeros ( (3,4))
```

Out[19]:
```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

In [20]:
```
ones( (2,3))
```

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

```
In [21]: # 3-D Array
         # 2 sets, each array = 3 X 4
         ones ( (2,3,4))
```

```
Out[21]: 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.]]])
```

```
In [23]: # Empty = initial state is random
         empty ( (2,3))
```

```
Out[23]: array([[  1.72723371e-077,  -2.00389399e+000,   2.96439388e-323],
                [  0.00000000e+000,   0.00000000e+000,   0.00000000e+000]])
```

```
In [24]: # Create Sequences of Numbers
         # The functions returns arrays instead of lists

         arange (10, 35, 5)
```

```
Out[24]: array([10, 15, 20, 25, 30])
```

```
In [25]: arange (1, 2, .2) # Works also with floats
```

```
Out[25]: array([ 1. ,  1.2,  1.4,  1.6,  1.8])
```

```
In [26]: # But
         # When arrays are created with float, it is generally not possible
         # To determine the number of elements due to the finite
         # Floating Point Precision
         # Hence
         # It is better to create them with linspace
```

```
In [27]: linspace (0, 2, 9) # 9 numbers from 0 to 2
```

```
Out[27]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

```
In [29]: # If array is too large, numpy prints ...
         # This can be disabled with set_printoptions(threshold='nan')
         arange (0, 100000)
```

```
Out[29]: array([    0,     1,     2, ..., 99997, 99998, 99999])
```

Basic Operations

```
In [49]: # Arithmetic Operations are element-wise
         a = array([1,2,3,4])
         b = arange(4)
```

```
In [50]: a

Out[50]: array([1, 2, 3, 4])

In [51]: b

Out[51]: array([0, 1, 2, 3])

In [52]: a - b

Out[52]: array([1, 1, 1, 1])

In [53]: a ** 2

Out[53]: array([ 1,  4,  9, 16])

In [54]: a < 3

Out[54]: array([ True,  True, False, False], dtype=bool)

In [55]: # The * operator applies element-wise
         a = array ( [[1,2], [2,3]])
         b = array ( [[2,3], [4,5]])

In [56]: a * b

Out[56]: array([[ 2,  6],
                [ 8, 15]])

In [57]: # Dot Product
         dot (a,b)

Out[57]: array([[10, 13],
                [16, 21]])

In [58]: # Operators such as += and *= modifies array in place

In [59]: a

Out[59]: array([[1, 2],
                [2, 3]])

In [60]: a += 10

In [61]: a

Out[61]: array([[11, 12],
                [12, 13]])

In [62]: b

Out[62]: array([[2, 3],
                [4, 5]])
```

```
In [63]: a += b

In [64]: a

Out[64]: array([[13, 15],
                [16, 18]])

In [65]: b

Out[65]: array([[2, 3],
                [4, 5]])

In [70]: # When different array types are operated, they are upcasted
         a = ones (3, dtype=int32)
         b = linspace (0, pi, 3)
         b.dtype.name

Out[70]: 'float64'

In [71]: c = a + b

In [72]: c

Out[72]: array([ 1.        ,  2.57079633,  4.14159265])

In [73]: c.dtype.name

Out[73]: 'float64'

In [74]: # Many unary methoda are implemented in numpy
         a = random.random ((2,3))

In [75]: a

Out[75]: array([[ 0.5517013 ,  0.02944191,  0.40126392],
                [ 0.47745967,  0.30205265,  0.29804652]])

In [76]: a.sum()

Out[76]: 2.0599659581764205

In [77]: a.min()

Out[77]: 0.029441914825537818

In [78]: a.max()

Out[78]: 0.55170129732296391
```

```
In [82]: # These operates across the entire array, as if it were a
         # List of Numbers
         # You can specify the axis parameter to change this default behaviour


         b = arange(15).reshape(3,5)
```

```
In [83]: b
```

```
Out[83]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])
```

```
In [84]: b.sum()
```

```
Out[84]: 105
```

```
In [85]: b.sum(axis=0) # Sum of each column
```

```
Out[85]: array([15, 18, 21, 24, 27])
```

```
In [86]: b.min(axis=1) # Min of each row
```

```
Out[86]: array([ 0,  5, 10])
```

```
In [87]: b.cumsum(axis=1) # Cumulative Sum Across Row
```

```
Out[87]: array([[ 0,  1,  3,  6, 10],
                [ 5, 11, 18, 26, 35],
                [10, 21, 33, 46, 60]])
```

Indexing, Slicing and Iterating

```
In [88]: a = arange (10) ** 2
```

```
In [89]: a
```

```
Out[89]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
In [90]: a[2]
```

```
Out[90]: 4
```

```
In [91]: a[2:5]
```

```
Out[91]: array([ 4,  9, 16])
```

```
In [92]: a[:6:2] # Equivalent to a[0:6:2]
```

```
Out[92]: array([ 0,  4, 16])
```

```
In [93]: # a[:6:2] == a[0:6:2] == From start (0) to 6 in steps of 2

In [94]: a[0:6:2] = - 10

In [95]: a

Out[95]: array([-10,   1, -10,   9, -10,  25,  36,  49,  64,  81])

In [96]: a[::-1] # Start to End in -1 Steps = Reverse the Array

Out[96]: array([ 81,  64,  49,  36,  25, -10,   9, -10,   1, -10])

In [97]: a[:]

Out[97]: array([-10,   1, -10,   9, -10,  25,  36,  49,  64,  81])

In [99]: a[:1]

Out[99]: array([-10])

In [100]: # Multidimensional Arrays

In [101]: def f(x,y):
              return 10 * x + y

          b = fromfunction(f,(5,4), dtype=int)
          b

Out[101]: array([[ 0,  1,  2,  3],
                 [10, 11, 12, 13],
                 [20, 21, 22, 23],
                 [30, 31, 32, 33],
                 [40, 41, 42, 43]])

In [102]: b[2,3]

Out[102]: 23

In [103]: b[0:5,1] # Rows 0-5, Column 1

Out[103]: array([ 1, 11, 21, 31, 41])

In [104]: b[:1] # First 1 ELEMENT of B

Out[104]: array([[0, 1, 2, 3]])

In [107]: b[:2] # First 2 ELEMENTS of B

Out[107]: array([[ 0,  1,  2,  3],
                 [10, 11, 12, 13]])
```

```
In [112]: # But
          b[:,1] # The 1st Column in All Elements of b

Out[112]: array([ 1, 11, 21, 31, 41])

In [113]: b[:,2] # The 2nd Column in All Elements of b

Out[113]: array([ 2, 12, 22, 32, 42])

In [116]: b[:,2:4] # The 2nd - 3rd Column in All Elements of b

Out[116]: array([[ 2,  3],
                 [12, 13],
                 [22, 23],
                 [32, 33],
                 [42, 43]])

In [117]: b[:0] # No ELEMENTS of B

Out[117]: array([], shape=(0, 4), dtype=int64)

In [118]: b[1] # First Element

Out[118]: array([10, 11, 12, 13])

In [119]: b[:] # All ELEMENTS of B

Out[119]: array([[ 0,  1,  2,  3],
                 [10, 11, 12, 13],
                 [20, 21, 22, 23],
                 [30, 31, 32, 33],
                 [40, 41, 42, 43]])

In [120]: # When indices are missing, the missing indices are considered complete slic
          es

          b[-1] # == b[-1,:] == last row in all columns

Out[120]: array([40, 41, 42, 43])
```

The expression within brackets in b[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes. NumPy also allows you to write this using dots as b[i,...]. The dots (...) represent as many colons as needed to produce a complete indexing tuple. For example, if x is a rank 5 array (i.e., it has 5 axes), then x[1,2,...] is equivalent to x[1,2,:,:,:], x[...,3] to x[:,:,:,:,3] and x[4,...,5,:] to x[4,:,:,5,:].

```
In [129]: c = array( [
                  [[1,2],[2,3]],
                  [[10,20],[30,40]]
                       ])

In [130]: c.shape

Out[130]: (2, 2, 2)
```

In [131]:
```
c[1,]
```
Out[131]:
```
array([[10, 20],
       [30, 40]])
```
In [132]:
```
c[1,...
```
Out[132]:
```
array([[10, 20],
       [30, 40]])
```
In [134]:
```
c[...,1]  # The 1st column in each element in each element of c
```
Out[134]:
```
array([[ 2,  3],
       [20, 40]])
```
In [135]:
```
# Iterating over array
b
```
Out[135]:
```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```
In [136]:
```
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]
```
In [139]:
```
for row in b[0:1]:
    for element in row:
        print element
```
```
0
1
2
3
```
In [142]:
```
# Alternatively one can "FLATTEN" the array
for element in b[0:1].flat:
    print element
```
```
0
1
2
3
```

Shape Manipulation

```
In [162]: # The shape of an array can be changed with various commands
          a = floor (10*random.random((3,4)))
```

```
In [163]: a
```

```
Out[163]: array([[ 7.,  6.,  5.,  3.],
                 [ 8.,  8.,  8.,  3.],
                 [ 6.,  1.,  6.,  3.]])
```

```
In [164]: a.shape
```

```
Out[164]: (3, 4)
```

```
In [165]: a.flat # This is an object (iterator)
```

```
Out[165]: <numpy.flatiter at 0x7f9932182800>
```

```
In [167]: a.ravel() # This razes to 1st level
```

```
Out[167]: array([ 7.,  6.,  5.,  3.,  8.,  8.,  8.,  3.,  6.,  1.,  6.,  3.])
```

```
In [168]: # Now you can change the shape
```

```
In [169]: a.shape = (4,3)  # Essentially I am transposing
          a
```

```
Out[169]: array([[ 7.,  6.,  5.],
                 [ 3.,  8.,  8.],
                 [ 8.,  3.,  6.],
                 [ 1.,  6.,  3.]])
```

```
In [161]: a.transpose() # This will bring us back the original array
```

```
Out[161]: array([[ 3.,  1.,  4.,  2.],
                 [ 7.,  9.,  4.,  5.],
                 [ 3.,  3.,  0.,  9.]])
```

```
In [171]: a
```

```
Out[171]: array([[ 7.,  6.,  5.],
                 [ 3.,  8.,  8.],
                 [ 8.,  3.,  6.],
                 [ 1.,  6.,  3.]])
```

```
In [174]: a.resize((3,4))
In [175]: a

Out[175]: array([[ 7.,  6.,  5.,  3.],
                 [ 8.,  8.,  8.,  3.],

                 [ 6.,  1.,  6.,  3.]])

In [176]: # If -1 is given as a reshaping option, the other
          # Dimensions are automatically calculated
          a.reshape(3,-1)

Out[176]: array([[ 7.,  6.,  5.,  3.],
                 [ 8.,  8.,  8.,  3.],
                 [ 6.,  1.,  6.,  3.]])

In [197]: # Stacking Arrays
          a = floor(10*random.random((2,2)))
          b = floor(10*random.random((2,2)))

In [198]: a

Out[198]: array([[ 5.,  0.],
                 [ 8.,  5.]])

In [199]: b

Out[199]: array([[ 5.,  2.],
                 [ 6.,  2.]])

In [200]: vstack((a,b))

Out[200]: array([[ 5.,  0.],
                 [ 8.,  5.],
                 [ 5.,  2.],
                 [ 6.,  2.]])

In [201]: hstack((a,b))

Out[201]: array([[ 5.,  0.,  5.,  2.],
                 [ 8.,  5.,  6.,  2.]])

In [202]: # Column stack stacks 1D arrays as columns into a 2D array
          column_stack((a,b))

Out[202]: array([[ 5.,  0.,  5.,  2.],
                 [ 8.,  5.,  6.,  2.]])

In [203]: row_stack((a,b))

Out[203]: array([[ 5.,  0.],
                 [ 8.,  5.],
                 [ 5.,  2.],
                 [ 6.,  2.]])
```

```
In [204]: a = array([1,2])
          b = array([10,20])

          a

Out[204]: array([1, 2])
```

```
In [205]: b

Out[205]: array([10, 20])
```

```
In [206]: a[:,newaxis] # This allows 2D column vectors

Out[206]: array([[1],
                 [2]])
```

```
In [207]: b[:,newaxis]

Out[207]: array([[10],
                 [20]])
```

```
In [208]: column_stack((a[:,newaxis], b[:,newaxis]))

Out[208]: array([[ 1, 10],
                 [ 2, 20]])
```

```
In [209]: vstack((a[:,newaxis], b[:,newaxis]))

Out[209]: array([[ 1],
                 [ 2],
                 [10],
                 [20]])
```

```
In [210]: # Row Stack
          row_stack((a[:,newaxis], b[:,newaxis]))

Out[210]: array([[ 1],
                 [ 2],
                 [10],
                 [20]])
```

```
In [211]: # One can also use r_ and c_ to create arrays
          r_[1:4,0,4]

Out[211]: array([1, 2, 3, 0, 4])
```

```
In [233]: # Splitting an array into several smaller ones
          random.seed(seed=1)
          a = floor (10 * random.random((2,10)))
```

```
In [234]: a

Out[234]: array([[ 4.,  7.,  0.,  3.,  1.,  0.,  1.,  3.,  3.,  5.],
                 [ 4.,  6.,  2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.]])
```

```
In [235]: hsplit(a,5) # Split into 5 arrays

Out[235]: [array([[ 4.,  7.],
                 [ 4.,  6.]]), array([[ 0.,  3.],
                 [ 2.,  8.]]), array([[ 1.,  0.],
                 [ 0.,  6.]]), array([[ 1.,  3.],
                 [ 4.,  5.]]), array([[ 3.,  5.],
                 [ 1.,  1.]])]
```

```
In [236]: hsplit(a,(3,4)) # Splits after the 3rd and 4th columns

Out[236]: [array([[ 4.,  7.,  0.],
                 [ 4.,  6.,  2.]]), array([[ 3.],
                 [ 8.]]), array([[ 1.,  0.,  1.,  3.,  3.,  5.],
                 [ 0.,  6.,  4.,  5.,  1.,  1.]])]
```

Copies and Views

```
In [237]: # No Copy
          a = arange(12)
          b = a
          b is a

Out[237]: True
```

```
In [238]: b.shape = (3,4)
```

```
In [240]: # Shape of a also changes
          a.shape

Out[240]: (3, 4)
```

```
In [241]: # Python passes mutable objects as references
          # So, function calls make no copy

          def f(x):
              return id(x)

          id(a)

Out[241]: 140295982423248
```

```
In [242]: print f(a)

          140295982423248
```

```
In [243]: id(a) == f(a)

Out[243]: True
```

In [244]:
```
# Shallow Copy
# Different Array Objects can share the same data
# The 'view' method creates a new array obj that looks at the same data

# c is "looking at a"
# If you change c values, a's values will also change

c = a.view()
c is a
```

Out[244]:
```
False
```

In [245]:
```
c.base
```

Out[245]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [246]:
```
c.base is a
```

Out[246]:
```
True
```

In [247]:
```
c.flags.owndata
```

Out[247]:
```
False
```

In [248]:
```
c.shape = 2,6
```

In [249]:
```
a.shape # This does not change
```

Out[249]:
```
(3, 4)
```

In [250]:
```
c.shape # But this is still "looking at a"
```

Out[250]:
```
(2, 6)
```

In [256]:
```
# So, if we change some value in c, it will also change in a
c[1,1] = 1000
```

In [257]:
```
a
```

Out[257]:
```
array([[   0,    1,    2,    3],
       [   4,    5,    6, 1000],
       [   8,    9,   10,   11]])
```

In [263]:
```
# Slicing an Array also returns a view of it
s = a[:,1:3] # All Rows, Columns 1-2
s
```

Out[263]:
```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

```
In [264]: # s is a "view" of a
          # So, if we change s
          # It will also change a

          s[:] = 99
```

```
In [265]: a
```

```
Out[265]: array([[   0,   99,   99,    3],
                 [   4,   99,   99, 1000],
                 [   8,   99,   99,   11]])
```

```
In [266]: # Deep Copy
          # Makes a complete copy of the object

          d = a.copy()
```

```
In [267]: d is a
```

```
Out[267]: False
```

Fancy Indexing with Arrays

```
In [273]: a = arange(12)**2
          a
```

```
Out[273]: array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])
```

```
In [274]: i = array( [ 0, 1, 2])
```

```
In [275]: a[i] # Elements of a at positions 0, 1 and 2
```

```
Out[275]: array([0, 1, 4])
```

```
In [277]: # Bi-dimensional indexing
          j = array([[0,1],[1,2]])
          a[j]
```

```
Out[277]: array([[0, 1],
                 [1, 4]])
```

```
In [278]: palette = array( [
          [0,0,0],                # black
          [255,0,0],              # red
          [0,255,0],              # green
          [0,0,255],              # blue
          [255,255,255] ] )
```

```
In [279]: image = array( [ [ 0, 1, 2, 0 ], [ 0, 3, 4, 0 ]  ] )
```

In [281]:
```
palette[image] # image = array( [ [ 0, 1, 2, 0 ], [ 0, 3, 4, 0 ]  ] )

# 0 0 0 corresponds to palette[0]
# 255 0 0 corresponds to palette[1]
# 0 255 0 corresponds to palette[2]
# 0 0 0 corresponds to palette[0]
# ... and so on
```

Out[281]:
```
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],

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

In [300]:
```
# We can also give indices for more than one dimension
s = arange(12).reshape(2,2,3)
s
```

Out[300]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

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

In [301]:
```
i = array([0,1]) # Indices for the 1st dimension
```

In [304]:
```
j = array([1,0]) # Indices for the 2nd dimension
```

In [306]:
```
# i = 0,1 as 1st arg == 1st element of the 0th element of a
# j = 1,0 as 2nd arg == 0th element of the 1st element of a

# Hence, from above
# s[i,j] == 3,4,5 and 6,7,8

s[i,j]
```

Out[306]:
```
array([[3, 4, 5],
       [6, 7, 8]])
```

In [307]:
```
# You can put i,j in a list and index with the list
l = [i,j]
s[l]
```

Out[307]:
```
array([[3, 4, 5],
       [6, 7, 8]])
```

In [311]:
```
t = array([i,j])
t
```

Out[311]:
```
array([[0, 1],
       [1, 0]])
```

```
In [312]: tuple(t) # Two separate arrays, like i and j in a tuple

Out[312]: (array([0, 1]), array([1, 0]))

In [310]: s[tuple(t)]

Out[310]: array([[3, 4, 5],
                 [6, 7, 8]])

In [313]: # Another common use is the search for max of time series
          time = linspace (20, 145, 5)
          data = sin (arange(20)). reshape(5,4)
          time

Out[313]: array([  20.  ,   51.25,   82.5 ,  113.75,  145.  ])

In [314]: data

Out[314]: array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
                 [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
                 [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
                 [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
                 [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])

In [316]: ind = data.argmax(axis=0) # Find the "index" of the maxima of each series
          ind

Out[316]: array([2, 0, 3, 1])

In [317]: time_max = time[ind] # Times corresponding to data max
          time_max

Out[317]: array([  82.5 ,   20.  ,  113.75,   51.25])

In [319]: data.shape

Out[319]: (5, 4)

In [320]: data.shape[1]

Out[320]: 4

In [321]: xrange(data.shape[1]) # Similar to range, more efficient for large objects

Out[321]: xrange(4)

In [318]: # data[ind, xrange(data.shape[1])] == data[ind[0],0], data[ind[1],1]...
          data_max = data[ind, xrange(data.shape[1])]
          data_max

Out[318]: array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])
```

```
In [323]: # Much easier
          data.max(axis=0)

Out[323]: array([ 0.98935825,  0.84147098,  0.99060736,  0.6569866 ])

In [336]: a = arange(5) + 1
          a

Out[336]: array([1, 2, 3, 4, 5])

In [337]: a[[1,3,4]]

Out[337]: array([2, 4, 5])

In [338]: a[[1,3,4]] = [100,200,300]
          a

Out[338]: array([  1, 100,   3, 200, 300])

In [339]: a[[0,0,2]]

Out[339]: array([1, 1, 3])

In [340]: # However, you have to be careful when you use this method
          # With +=,... it will only modify the elements representing the indices
          # Hence, a[[0,0,2]] += 1 adds 1 to a[0] and a[2] and prints indices 0,0,2
          a[[0,0,2]] + 1

Out[340]: array([2, 2, 4])

In [341]: # But
          a[[0,0,2]] += 1
          a

Out[341]: array([  2, 100,   4, 200, 300])
```

Indexing with Boolean Arrays

```
In [348]: a = arange(12).reshape(3,4)
          a

Out[348]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])

In [349]: b = a > 4
          b

Out[349]: array([[False, False, False, False],
                 [False,  True,  True,  True],
                 [ True,  True,  True,  True]], dtype=bool)
```

```
In [350]: a[b]

Out[350]: array([ 5,  6,  7,  8,  9, 10, 11])

In [351]: a[b] = 0 # All elements > 4 set to 0
          a

Out[351]: array([[0, 1, 2, 3],
                 [4, 0, 0, 0],
                 [0, 0, 0, 0]])

In [352]: a = arange(12).reshape(3,4)
          a

Out[352]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])

In [354]: b = array([False,True,True])
          a[b,:] # Show only rows 1-2, and all columns

Out[354]: array([[ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])

In [355]: a[b,b] # The intersection of Rows 1-2, Columns 1-2 (See a above)

Out[355]: array([ 5, 10])

In [373]: # The ix_() Function
          # The ix_ function can be used to combine different vectors
          # so as to obtain the result for each n-uplet. For example,
          # if you want to compute all the a+b+c for all the triplets
          # taken from each of the vectors a, b and c:

          a = array([0,1,2,3])
          b = array([10,20,30])
          c = array([100,200,300,400,500])
          ax,bx,cx = ix_(a,b,c)

In [374]: ax

Out[374]: array([[[0]],

                 [[1]],

                 [[2]],

                 [[3]]])

In [375]: bx

Out[375]: array([[[10],
                  [20],
                  [30]]])
```

```
In [376]: cx
```

```
Out[376]: array([[[100, 200, 300, 400, 500]]])
```

```
In [377]: ax.shape, bx.shape, cx.shape
```

```
Out[377]: ((4, 1, 1), (1, 3, 1), (1, 1, 5))
```

```
In [378]: result = ax+bx+cx
          result
```

```
Out[378]: array([[[110, 210, 310, 410, 510],
                  [120, 220, 320, 420, 520],
                  [130, 230, 330, 430, 530]],

                 [[111, 211, 311, 411, 511],
                  [121, 221, 321, 421, 521],
                  [131, 231, 331, 431, 531]],

                 [[112, 212, 312, 412, 512],
                  [122, 222, 322, 422, 522],
                  [132, 232, 332, 432, 532]],

                 [[113, 213, 313, 413, 513],
                  [123, 223, 323, 423, 523],
                  [133, 233, 333, 433, 533]]])
```

```
In [379]: result[3,2,4]
```

```
Out[379]: 533
```

```
In [381]: a[3] + b[2] + c[4] # !!
```

```
Out[381]: 533
```

```
In [382]: # Simple array operations include
          # a.transpose()
          # a.inv()
          # eye(2) # Unit 2 X 2 Matrix
          # dot (a,b) # Dot product
          # solve (see below)
          # eig(a)
          # trace(a) # Sum across diagonals
```

```
In [396]: x = array([[1,2],[3,7]])
          y = array([10,20])
```

```
In [397]: linalg.solve(x,y)
```

```
Out[397]: array([ 30., -10.])
```

```
In [399]: trace(x)

Out[399]: 8
```

The Matrix Class

Note that there are some important differences between NumPy arrays and matrices. NumPy provides two fundamental objects: an N-dimensional array object and a universal function object. Other objects are built on top of these. In particular, matrices are 2-dimensional array objects that inherit from the NumPy array object. For both arrays and matrices, indices must consist of a proper combination of one or more of the following: integer scalars, ellipses, a list of integers or boolean values, a tuple of integers or boolean values, and a 1-dimensional array of integer or boolean values. A matrix can be used as an index for matrices, but commonly an array, list, or other form is needed to accomplish a given task. As usual in Python, indexing is zero-based. Traditionally we represent a 2D array or matrix as a rectangular array of rows and columns, where movement along axis 0 is movement across rows, while movement along axis 1 is movement across columns.

```
In [414]: A = arange(12)
          A

Out[414]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [416]: A.shape = (3,4)
          M = mat(A.copy())
          print type(A),"\n", type(M)

          <type 'numpy.ndarray'>
          <class 'numpy.matrixlib.defmatrix.matrix'>
```

```
In [417]: A

Out[417]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])
```

```
In [421]: M

Out[421]: matrix([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11]])
```

```
In [422]: M.T

Out[422]: matrix([[ 0,  4,  8],
                  [ 1,  5,  9],
                  [ 2,  6, 10],
                  [ 3,  7, 11]])
```

```
In [423]: M.I
```

```
Out[423]: matrix([[-0.3375    , -0.1       ,  0.1375    ],
                  [-0.13333333, -0.03333333,  0.06666667],
                  [ 0.07083333,  0.03333333, -0.00416667],

                  [ 0.275     ,  0.1       , -0.075     ]])
```

```
In [426]: # Automatic Reshaping
          # Use -1 to indicate "whatever is needed"
          a = arange(30)
          a.shape = (2,-1,3)
          a
```

```
Out[426]: array([[[ 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]]])
```

```
In [427]: a.shape
```

```
Out[427]: (2, 5, 3)
```

```
In [ ]:
```