Numpy Quickstart tutorial

Pre-requisites

- Python
- Installation - [https://scipy.org/install.html (https://scipy.org/install.html)](https://scipy.org/install.html)

The Basics

- Homogeneous multidimensional array.
- Table of elements (usually numbers).
- All of the same type.
- Indexed by a tuple of non-negative integers.
- Dimensions are called axes.

Example: Coordinates of a point in 3D space [1, 2, 1]:

- Has one axis.
- Axis has 3 elements
- 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.

In [1]:
```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```
Out[1]:
```
[[1.0, 0.0, 0.0], [0.0, 1.0, 2.0]]
```

Important attributes of an ndarray object are:

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.

An example

In [2]:
```python
import numpy as np
a = np.arange(15).reshape(3, 5)
a
```

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

In [3]:
```
a.shape
```
Out[3]:
```
(3, 5)
```

In [4]:
```
a.ndim
```
Out[4]:
```
2
```

In [5]:
```
a.dtype.name
```
Out[5]:
```
'int64'
```

In [6]:
```
a.itemsize
```
Out[6]:
```
8
```

In [7]:
```
a.size
```
Out[7]:
```
15
```

In [8]:
```
type(a)
```
Out[8]:
```
numpy.ndarray
```

In [9]:
```
b = np.array([6,7,8])
```

In [10]:
```
type(b)
```
Out[10]:
```
numpy.ndarray
```

Array Creation

- Create an array from a regular Python list or tuple using the array function.
- The type of the resulting array is deduced from the type of the elements in the sequences.

In [11]:

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

Out[11]:

```
array([2, 3, 4])
```

In [12]:
```
a.dtype
```
Out[12]:
```
dtype('int64')
```

In [13]:
```
b = np.array([1.2, 3.5, 5.1])
b.dtype
```

Out[13]:
```
dtype('float64')
```

A frequent error consists in calling array with multiple numeric arguments, rather than providing a single list of numbers as an argument.

In [14]:

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

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

In [15]:

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

Out[15]:

```
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

The type of the array can also be explicitly specified at creation time:

In [16]:
```
c = np.array( [ [1,2], [3,4] ], dtype = complex )
c
```

Out[16]:
```
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.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64.

In [17]:
```
np.zeros( (3,4) )
```

Out[17]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [18]:
```python
np.ones( (2,3,4), dtype=np.int16 ) # dtype can also be specified
```

Out[18]:
```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
```

In [19]:
```
np.empty( (2,3) ) # uninitialized, output may vary
```

Out[19]:
```
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.

In [20]:
```
np.arange( 10, 30, 5 )
```
Out[20]:
```
array([10, 15, 20, 25])
```

In [21]:
```
np.arange( 0, 2, 0.3 ) # it accepts float arguments
```

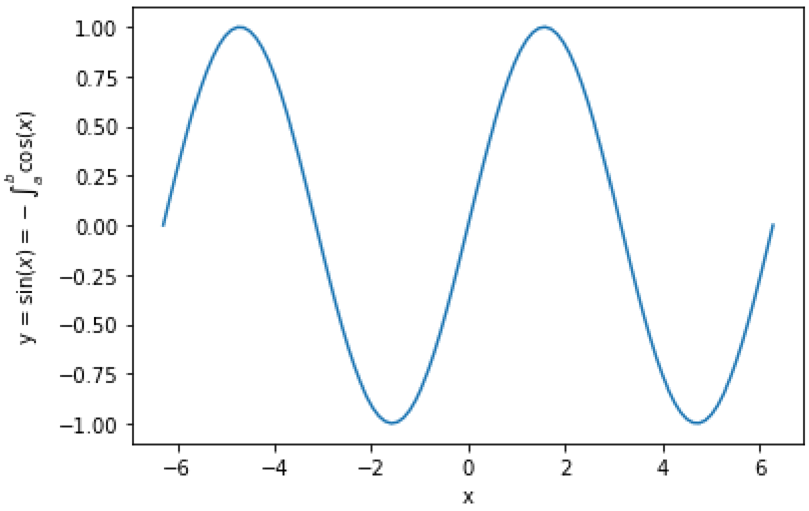
Out[21]:
```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

In [22]:
```
from numpy import pi
np.linspace( 0, 2, 9 ) # 9 numbers from 0 to 2
```
Out[22]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
```

In [23]:
```
x = np.linspace( -2*pi, 2*pi, 100 ) # useful to evaluate function at lots of poi
nts
f = np.sin(x)

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x, f)
plt.xlabel('x')
plt.ylabel('y = $\sin(x) = - \int_a^b \cos(x)$')
plt.show()
```



Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

In [24]:

```
a = np.arange(6)                                    # 1d array
print(a)
```

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

In [25]:

```
b = np.arange(12).reshape(4,3)          # 2d array
print(b)
```

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

In [26]:
```
c = np.arange(24).reshape(2,3,4)          # 3d array
print(c)
```

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

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

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

In [27]:

```
print(np.arange(10000))
```

```
[   0    1    2 ... 9997 9998 9999]
```

In [28]:
```
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 [29]:

```
import sys
np.set_printoptions(threshold=sys.maxsize)       # sys module should be imported
```

In [30]:
```python
print(np.arange(400).reshape(20,20))
```

```
[[  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]
 [240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257
  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
```

```
 [340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357
  358 359]
 [360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
  378 379]
 [380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397
  398 399]]
```

Basic Operations

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

In [31]:
```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
b
```

Out[31]:
```
array([0, 1, 2, 3])
```

In [32]:
```
c = a-b
c
```

Out[32]:
```
array([20, 29, 38, 47])
```

In [33]:
```
b**2
```
Out[33]:
```
array([0, 1, 4, 9])
```

In [34]:
```
10*np.sin(a)
```

Out[34]:
```
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

In [35]:
```
a < 35
```
Out[35]:
```
array([ True,  True, False, False])
```

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

In [36]:
```
A = np.array( [[1,1],
            [0,1]] )
B = np.array( [[2,0],
            [3,4]] )
A * B                       # elementwise product
```

Out[36]:
```
array([[2, 0],
       [0, 4]])
```

In [37]:

```
A @ B                            # matrix product
```

Out[37]:

```
array([[5, 4],
       [3, 4]])
```

In [38]:
```
A.dot(B)                    # another matrix product
```
Out[38]:
```
array([[5, 4],
       [3, 4]])
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

In [39]:
```
a = np.ones((2,3), dtype=int)
b = np.random.random((2,3))
a *= 3
a
```

Out[39]:
```
array([[3, 3, 3],
       [3, 3, 3]])
```

In [40]:
```
b += a
b
```

Out[40]:
```
array([[3.854513  , 3.86847312, 3.87509364],
       [3.8116518 , 3.93729461, 3.04439952]])
```

In [41]:

```
# a += b                          # b is not automatically converted to integer type
```

Upcasting

- When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one.

In [42]:

```
a = np.ones(3, dtype=np.int32)
b = np.linspace(0,pi,3)
b.dtype.name
```

Out[42]:

```
'float64'
```

In [43]:

```
c = a+b
c
```

Out[43]:

```
array([1.        , 2.57079633, 4.14159265])
```

In [44]:
```
c.dtype.name
```
Out[44]:
```
'float64'
```

In [45]:

```
d = np.exp(c*1j)
d
```

Out[45]:

```
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
```

In [46]:
```
d.dtype.name
```
Out[46]:
```
'complex128'
```

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

In [47]:
```
a = np.random.random((2,3))
a
```

Out[47]:
```
array([[0.46758266, 0.25969734, 0.14802777],
       [0.4213831 , 0.31284864, 0.83502871]])
```

In [48]:
```
a.sum()
```
Out[48]:
```
2.4445682068744197
```

In [49]:
```
a.min()
```
Out[49]:
```
0.14802777142121226
```

In [50]:

```
a.max()
```

Out[50]: 0.8350287063339855

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

In [51]:

```
b = np.arange(12).reshape(3,4)
b
```

Out[51]:

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

In [52]:

```
b.sum(axis=0)                            # sum of each column
```

Out[52]:

```
array([12, 15, 18, 21])
```

In [53]:

```
b.min(axis=1)                              # min of each row
```

Out[53]:

```
array([0, 4, 8])
```

In [54]:
```
b.cumsum(axis=1)                          # cumulative sum along each row
```

Out[54]:
```
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). Within NumPy, these functions operate elementwise on an array, producing an array as output.

In [55]:
```
B = np.arange(3)
B
```
Out[55]:
```
array([0, 1, 2])
```

In [56]:
```
np.exp(B)
```
Out[56]:
```
array([1.        , 2.71828183, 7.3890561 ])
```

In [57]:
```
np.sqrt(B)
```
Out[57]:
```
array([0.        , 1.        , 1.41421356])
```

In [58]:

```
C = np.array([2., -1., 4.])
np.add(B, C)
```

Out[58]:

```
array([2., 0., 6.])
```

Indexing, Slicing and Iterating

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

In [59]:

```
a = np.arange(10)**3
a
```

Out[59]:

```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

In [60]:

```
a[2]
```

Out[60]: 8

In [61]:
```
a[2:5]
```

Out[61]:
```
array([ 8, 27, 64])
```

In [62]:
```
a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, e
xclusive, set every 2nd element to -1000
a
```

Out[62]:
```
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,
         729])
```

In [63]:
```
a[ : :-1]                                      # reversed a
```

Out[63]:
```
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1,
       -1000])
```

In [64]:

```
# for i in a:                                   # NumPy, RuntimeWarning: invalid value
encountered in power
for i in a.astype('complex'):
    print(i**(1/3.0))
```

```
(5+8.660254037844384j)
(1+0j)
(5+8.660254037844384j)
(2.9999999999999996+0j)
(5+8.660254037844384j)
(5.000000000000001+0j)
(6+0j)
(6.999999999999998+0j)
(7.999999999999998+0j)
(8.999999999999998+0j)
```

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

In [65]:
```python
def f(x,y):
    return 10*x+y

b = np.fromfunction(f,(5,4),dtype=int)
b
```

Out[65]:
```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

In [66]:

```
b[2,3]
```

Out[66]:

```
23
```

In [67]:
```
b[0:5, 1]                              # each row in the second column of b
```

Out[67]:
```
array([ 1, 11, 21, 31, 41])
```

In [68]:

```
b[ : ,1]                              # equivalent to the previous example
```

Out[68]:

```
array([ 1, 11, 21, 31, 41])
```

In [69]:
```
b[1:3, : ]                      # each column in the second and third row of b
```

Out[69]:
```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

In [70]:

```
b[-1]                                  # the last row. Equivalent to b[-1,:]
```

Out[70]:

```
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 an array with 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 [71]:

```
c = np.array( [[[  0,  1,  2],               # a 3D array (two stacked 2D array
s)
                [ 10, 12, 13]],
               [[100,101,102],
                [110,112,113]]])
print(c)
```

```
[[[  0   1   2]
  [ 10  12  13]]

 [[100 101 102]
  [110 112 113]]]
```

In [72]:
```
c.shape
```
Out[72]:
```
(2, 2, 3)
```

In [73]:

```
c[1,...]                                         # same as c[1,:,:] or c[1]
```

Out[73]:

```
array([[100, 101, 102],
       [110, 112, 113]])
```

In [74]:
```
c[...,2]                                             # same as c[:,:,2]
```

Out[74]:
```
array([[  2,  13],
       [102, 113]])
```

Iterating over multidimensional arrays is done with respect to the first axis:

In [75]:
```python
print(b)
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]]
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

In [76]:

```
for element in b.flat:
    print(element)
```

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

Shape Manipulation

Changing the shape of an array

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

In [77]:
```
a = np.floor(10*np.random.random((3,4)))
a
```

Out[77]:
```
array([[0., 5., 5., 5.],
       [5., 9., 2., 2.],
       [3., 2., 0., 5.]])
```

In [78]:

```
a.shape
```

Out[78]:

```
(3, 4)
```

The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

In [79]:

```
a.ravel()  # returns the array, flattened
```

Out[79]:

```
array([0., 5., 5., 5., 5., 9., 2., 2., 3., 2., 0., 5.])
```

In [80]:
```
a.reshape(6,2)  # returns the array with a modified shape
```

Out[80]:
```
array([[0., 5.],
       [5., 5.],
       [5., 9.],
       [2., 2.],
       [3., 2.],
       [0., 5.]])
```

In [81]:
```
a.T  # returns the array, transposed
```
Out[81]:
```
array([[0., 5., 3.],
       [5., 9., 2.],
       [5., 2., 0.],
       [5., 2., 5.]])
```

In [82]:
```
a.T.shape
```
Out[82]:
```
(4, 3)
```

In [83]:
```
a.shape
```
Out[83]:
```
(3, 4)
```

- The order of the elements in the array resulting from ravel() is normally “C-style”, that is, the rightmost index “changes the fastest”, so the element after a[0,0] is a[0,1].
- If the array is reshaped to some other shape, again the array is treated as “C-style”.
- NumPy normally creates arrays stored in this order, so ravel() will usually not need to copy its argument
- **But** if the array was made by taking slices of another array or created with unusual options, it may need to be copied.
- The functions ravel() and reshape() can also be instructed, using an optional argument, to use **FORTRAN-style** arrays, in which the leftmost index changes the fastest.

- The **reshape** function returns its argument with a modified shape
- The ndarray.resize method modifies the array itself:

In [84]:
```
a
```

Out[84]:
```
array([[0., 5., 5., 5.],
       [5., 9., 2., 2.],
       [3., 2., 0., 5.]])
```

In [85]:
```
a.resize((2,6))
a
```

Out[85]:
```
array([[0., 5., 5., 5., 5., 9.],
       [2., 2., 3., 2., 0., 5.]])
```

If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated:

In [86]:

```
a.reshape(2,-1)
```

Out[86]:

```
array([[0., 5., 5., 5., 5., 9.],
       [2., 2., 3., 2., 0., 5.]])
```

Stacking together different arrays

Several arrays can be stacked together along different axes:

In [87]:

```python
a = np.floor(10*np.random.random((2,2)))
print(a)
b = np.floor(10*np.random.random((2,2)))
print(b)
```

```
[[0. 6.]
 [2. 3.]]
[[3. 6.]
 [8. 8.]]
```

In [88]:
```
np.vstack((a,b))
```

Out[88]:
```
array([[0., 6.],
       [2., 3.],
       [3., 6.],
       [8., 8.]])
```

In [89]:
```
np.hstack((a,b))
```

Out[89]:
```
array([[0., 6., 3., 6.],
       [2., 3., 8., 8.]])
```

The function **column_stack** stacks 1D arrays as columns into a 2D array. It is equivalent to hstack only for 2D arrays:

In [90]:

```
from numpy import newaxis
c = np.column_stack((a,b))     # with 2D arrays
print('\na:')
print(a)
print('\nb:')
print(b)
print('\nc = np.column_stac((a,b)):')
print(c)
```

```
a:
[[0. 6.]
 [2. 3.]]

b:
[[3. 6.]
 [8. 8.]]

c = np.column_stac((a,b)):
[[0. 6. 3. 6.]
 [2. 3. 8. 8.]]
```

In [91]:

```python
a = np.array([4.,2.])
b = np.array([3.,8.])
c = np.column_stack((a,b))     # returns a 2D array

print('\na:')
print(a)
print('\na.shape:')
print(a.shape)
print('\nb:')
print(b)
print('\nb.shape:')
print(b.shape)
print('\nc = np.column_stac((a,b)):')
print(c)
print('\nc.shape:')
print(c.shape)
```

```
a:
[4. 2.]

a.shape:
(2,)

b:
[3. 8.]

b.shape:
(2,)

c = np.column_stac((a,b)):
[[4. 3.]
 [2. 8.]]

c.shape:
(2, 2)
```

In [92]:

```
c = np.hstack((a,b))          # the result is different
print('\na:')
print(a)
print('\na.shape:')
print(a.shape)
print('\nb:')
print(b)
print('\nb.shape:')
print(b.shape)
print('\nc = np.column_stac((a,b)):')
print(c)
print('\nc.shape:')
print(c.shape)
```

```
a:
[4. 2.]

a.shape:
(2,)

b:
[3. 8.]

b.shape:
(2,)

c = np.column_stac((a,b)):
[4. 2. 3. 8.]

c.shape:
(4,)
```

In [93]:

```
c = a[:,newaxis]                # this allows to have a 2D columns vector
print('\nc = a[:,newaxis]:')
print(c)
print('\nc.shape:')
print(c.shape)
```

```
c = a[:,newaxis]:
[[4.]
 [2.]]

c.shape:
(2, 1)
```

In [94]:

```
c = np.column_stack((a[:,newaxis],b[:,newaxis]))
c
```

Out[94]:

```
array([[4., 3.],
       [2., 8.]])
```

In [95]:
```
c = np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same as column
 stack
c
```

Out[95]:
```
array([[4., 3.],
       [2., 8.]])
```

1. On the other hand, the function **ma.row_stack** is equivalent to **vstack** for any input arrays.

2. In general:

- For arrays with more than two dimensions, **hstack** stacks along their **second** axes (columns).
- **vstack** stacks along their **first** axes (rows).
- Concatenate allows for an optional arguments giving the number of the axis along which the concatenation should happen.

Note

In complex cases, r *and c* are useful for creating arrays by stacking numbers along one axis. They allow the use of range literals (“:”)

In [96]:
```
np.r_[1:4, 11:3:-2, -4]
```

Out[96]:
```
array([ 1,  2,  3, 11,  9,  7,  5, -4])
```

When used with arrays as arguments, r *and c* are similar to vstack and hstack in their default behavior, but allow for an optional argument giving the number of the axis along which to concatenate.

See also hstack, vstack, column*stack*, *concatenate*, *c*, r_

Splitting one array into several smaller ones

- **hsplit** -- split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur:

In [97]:
```
a = np.floor(10*np.random.random((2,12)))
a
```

Out[97]:
```
array([[5., 0., 3., 6., 4., 2., 0., 8., 3., 9., 8., 8.],
       [8., 4., 6., 3., 9., 8., 5., 4., 0., 5., 5., 8.]])
```

In [98]:
```python
b = np.hsplit(a,3)   # Split a into 3
print('\na:')
print(a)
print("\nb = np.hsplit(a,3):")
for section in b:
    print(section)
```

```
a:
[[5. 0. 3. 6. 4. 2. 0. 8. 3. 9. 8. 8.]
 [8. 4. 6. 3. 9. 8. 5. 4. 0. 5. 5. 8.]]

b = np.hsplit(a,3):
[[5. 0. 3. 6.]
 [8. 4. 6. 3.]]
[[4. 2. 0. 8.]
 [9. 8. 5. 4.]]
[[3. 9. 8. 8.]
 [0. 5. 5. 8.]]
```

In [99]:

```python
b = np.hsplit(a,2)   # Split a into 3
print('\na:')
print(a)
print("\nb = np.hsplit(a,3):")
for section in b:
    print(section)
```

```
a:
[[5. 0. 3. 6. 4. 2. 0. 8. 3. 9. 8. 8.]
 [8. 4. 6. 3. 9. 8. 5. 4. 0. 5. 5. 8.]]

b = np.hsplit(a,3):
[[5. 0. 3. 6. 4. 2.]
 [8. 4. 6. 3. 9. 8.]]
[[0. 8. 3. 9. 8. 8.]
 [5. 4. 0. 5. 5. 8.]]
```

In [100]:

```python
b = np.hsplit(a,(3,4))   # Split a after the third and the fourth column
print('\na:')
print(a)
print("\nb = np.hsplit(a,(3,4)):")
for section in b:
    print(section)
```

```
a:
[[5. 0. 3. 6. 4. 2. 0. 8. 3. 9. 8. 8.]
 [8. 4. 6. 3. 9. 8. 5. 4. 0. 5. 5. 8.]]

b = np.hsplit(a,(3,4)):
[[5. 0. 3.]
 [8. 4. 6.]]
[[6.]
 [3.]]
[[4. 2. 0. 8. 3. 9. 8. 8.]
 [9. 8. 5. 4. 0. 5. 5. 8.]]
```

- **vsplit** splits along the vertical axis
- **array_split** allows one to specify along which axis to split.

Copies and Views

- When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not.
- This is often a source of confusion for beginners. There are three cases:

No Copy at All

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

In [101]:

```
a = np.arange(12)
b = a                  # no new object is created
print('\na:')
print(a)
print('\nb:')
print(b)
```

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

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

In [102]:

```
b is a          # a and b are two names for the same ndarray object
```

Out[102]:

```
True
```

In [103]:

```
b.shape = 3,4    # changes the shape of a
a.shape
print('\na:')
print(a)
print('\nb:')
print(b)
```

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

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

Python passes mutable objects as references, so function calls make no copy.

In [104]:

```python
def f(x):
    print(id(x))

id(a)                                # id is a unique identifier of an object
```

Out[104]:

```
140172266996240
```

In [105]:

```
f(a)
```

```
140172266996240
```

View or Shallow Copy

Different array objects can share the same data. The view method creates a new array object that looks at the same data.

In [106]:

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

Out[106]:

```
False
```

In [107]:

```
id(a)
```

Out[107]:

```
140172266996240
```

In [108]:
```
id(c)
```
Out[108]:
```
140172267030448
```

In [109]:

```python
c.base is a                        # c is a view of the data owned by a
```

Out[109]:

```
True
```

In [110]:

```
c.flags.owndata
```

Out[110]:

```
False
```

In [111]:

```
c.shape = 2,6                                    # a's shape doesn't change
print(a.shape)
print(c.shape)
```

```
(3, 4)
(2, 6)
```

In [112]:

```python
c[0,4] = 1234                   # but a's data changes
a
```

Out[112]:

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

Slicing an array returns a view of it:

In [113]:

```
s = a[ : , 1:3]
s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and
 s[:]=10
a
```

Out[113]:

```
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Deep Copy

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

In [114]:
```
d = a.copy()                    # a new array object with new data is crea
ted
d is a
```
Out[114]:
```
False
```

In [115]:
```
id(a)
```
Out[115]:
```
140172266996240
```

In [116]:
```
id(c)
```
Out[116]:
```
140172267030448
```

In [117]:

```
d.base is a                        # d doesn't share anything with a
```

Out[117]:

```
False
```

In [118]:

```
d[0,0] = 9999                                          # d doesn't share anything with a
a
```

Out[118]:

```
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Sometimes copy should be called after slicing if the original array is not required anymore.

For example:

- If a is a huge intermediate result and the final result b only contains a small fraction of a, a deep copy should be made when constructing b with slicing:

In [119]:

```
a = np.arange(int(1e8))
b = a[:100].copy()
del a  # the memory of ``a`` can be released.
print('\nb:')
print(b)
print('\nb.shape:')
print(b.shape)
```

```
b:
[ 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]

b.shape:
(100,)
```

If b = a[:100] is used instead, a is referenced by b and will persist in memory even if del a is executed.

Functions and Methods Overview¶

Array Creation

arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r, zeros, zeros_like

Conversions ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

Manipulations

array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

Questions

all, any, nonzero, where

Ordering

argmax, argmin, argsort, max, min, ptp, searchsorted, sort

Operations

choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

Basic Statistics

cov, mean, std, var

Basic Linear Algebra

cross, dot, outer, linalg.svd, vdot

Less Basic

Broadcasting rules

- Broadcasting allows universal functions to deal in a meaningful way with inputs that do not have exactly the same shape.

1. The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a “1” will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.
2. The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the “broadcast” array.

After application of the broadcasting rules, the sizes of all arrays must match. More details can be found in Broadcasting.

Fancy indexing and index tricks

- NumPy offers more indexing facilities than regular Python sequences.
- In addition to indexing by integers and slices, as we saw before, arrays can be indexed by:

1. arrays of integers, and;
2. arrays of booleans.

Indexing with Arrays of Indices

```
In [120]:
a = np.arange(12)**2                          # the first 12 square numbers
i = np.array( [ 1,1,3,8,5 ] )                 # an array of indices
a[i]                                          # the elements of a at the positions
 i

print('\na:')
print(a)
print('\ni:')
print(i)
print('\na[i]:')
print(a[i])
```

```
a:
[  0   1   4   9  16  25  36  49  64  81 100 121]

i:
[1 1 3 8 5]

a[i]:
[ 1  1  9 64 25]
```

When the indexed array a is multidimensional, a single array of indices refers to the first dimension of a. The following example shows this behavior by converting an image of labels into a color image using a palette.

In [121]:
```
palette = np.array( [ [0,0,0],                  # black
                      [255,0,0],                # red
                      [0,255,0],                # green
                      [0,0,255],                # blue
                      [255,255,255] ] )         # white

image = np.array( [ [ 0, 1, 2, 0 ],             # each value corresponds to a colo
r in the palette
                    [ 0, 3, 4, 0 ]  ] )
palette[image]                              # the (2,4,3) color image
```

Out[121]:
```
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]]])
```

We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

In [122]:
```python
a = np.arange(12).reshape(3,4)
print('\na:')
print(a)
```

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

In [123]:
```python
i = np.array( [ [0,1],                          # indices for the first dim of a
                [1,2] ] )
j = np.array( [ [2,1],                          # indices for the second dim
                [3,3] ] )
print('\ni:')
print(i)
print('\nj:')
print(j)
```

```
i:
[[0 1]
 [1 2]]

j:
[[2 1]
 [3 3]]
```

In [124]:
```python
print('\na:')
print(a)
print('\ni:')
print(i)
print('\nj:')
print(j)
b = a[i,j]                                # i and j must have equal shape
print('\nb = a[i,j]:')
print(b)
```

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

i:
[[0 1]
 [1 2]]

j:
[[2 1]
 [3 3]]

b = a[i,j]:
[[ 2  5]
 [ 7 11]]
```

In [125]:

```
print('\na:')
print(a)
print('\ni:')
print(i)
print('\nj:')
print(j)
b = a[i,2]
print('\nb = a[i,2]:')
print(b)
```

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

i:
[[0 1]
 [1 2]]

j:
[[2 1]
 [3 3]]

b = a[i,2]:
[[ 2  6]
 [ 6 10]]
```

In [126]:
```python
print('\na:')
print(a)
print('\ni:')
print(i)
print('\nj:')
print(j)
b = a[:,j]                                          # i and j must have equal shape
                                                    # ':' = [0,1,2]

print('\nb = a[:,j]:')
print(b)
```

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

i:
[[0 1]
 [1 2]]

j:
[[2 1]
 [3 3]]

b = a[:,j]:
[[[ 2  1]
  [ 3  3]]

 [[ 6  5]
  [ 7  7]]

 [[10  9]
  [11 11]]]
```

Naturally, we can put i and j in a sequence (say a list) and then do the indexing with the list.

In [127]:

```python
print('\na:')
print(a)
print('\ni:')
print(i)
print('\nj:')
print(j)
l = (i,j)
print('\nl:')
print(l)
b = a[l]                                        # equivalent to a[i,j]
print('\nb = a[l]:')
print(b)
```

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

i:
[[0 1]
 [1 2]]

j:
[[2 1]
 [3 3]]

l:
(array([[0, 1],
       [1, 2]]), array([[2, 1],
       [3, 3]]))

b = a[l]:
[[ 2  5]
 [ 7 11]]
```

However, we **can not** do this by putting i and j into an array, because this array will be interpreted as indexing the first dimension of a.

In [128]:

```
# s = np.array( [i,j] )
# a[s]
```

In [129]:

```
# a[tuple(s)]                                        # same as a[i,j]
```

Another common use of indexing with arrays is the search of the maximum value of time-dependent series:

In [130]:
```
time = np.linspace(20, 145, 5)                 # time scale
data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
time
```

Out[130]:
```
array([ 20.  ,  51.25,  82.5 , 113.75, 145.  ])
```

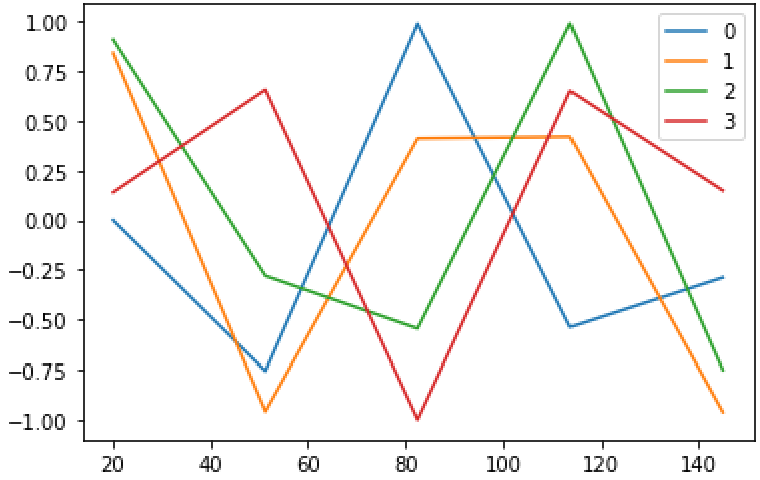
In [131]:

```
data
```

Out[131]:

```
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 [132]:
```
import matplotlib.pyplot as plt
plt.plot(time,data)
plt.legend(['0','1','2','3'])
plt.show()
```



In [133]:

```
ind = data.argmax(axis=0)                         # index of the maxima for each series
ind
```

Out[133]:

```
array([2, 0, 3, 1])
```

In [134]:

```
time_max = time[ind]                    # times corresponding to the maxima
time_max
```

Out[134]:

```
array([ 82.5 ,  20.  , 113.75,  51.25])
```

In [135]:

```
data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],
1]...
```

In [136]:

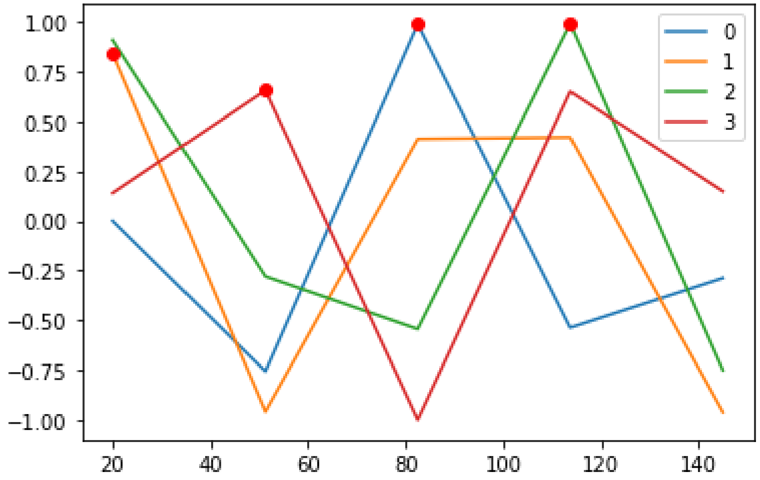
```
np.all(data_max == data.max(axis=0))
```

Out[136]:

```
True
```

In [137]:

```
import matplotlib.pyplot as plt
plt.plot(time,data)
plt.plot(time_max, data[(ind, np.arange(data.shape[1]))], 'ro')
plt.legend(['0','1','2','3'])
plt.show()
```



You can also use indexing with arrays as a target to assign to:

In [138]:

```
a = np.arange(5)
a[[1,3,4]] = 0
a
```

Out[138]:

```
array([0, 0, 2, 0, 0])
```

However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:

In [139]:

```
a = np.arange(5)
a[[0,0,2]]=[1,2,3]
a
```

Out[139]:

```
array([2, 1, 3, 3, 4])
```

This is reasonable enough, but watch out if you want to use Python’s += construct, as it may not do what you expect:

In [140]:

```
a = np.arange(5)
print('\na:')
print(a)
a[[0,0,2]]+=1
a
print('\na:')
print(a)
```

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

a:
[1 1 3 3 4]
```

Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires “a+=1” to be equivalent to “a = a + 1”.

Indexing with Boolean Arrays

- When we index arrays with arrays of (integer) indices we are providing the list of indices to pick.
- With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don’t.
- The most natural way one can think of for boolean indexing is to use boolean arrays that have the same shape as the original array:

In [141]:

```
a = np.arange(12).reshape(3,4)
print('\na:')
print(a)
b = a > 4
b                                           # b is a boolean with a's shape
print('\nb = a > 4:')
print(b)
```

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

b = a > 4:
[[False False False False]
 [False  True  True  True]
 [ True  True  True  True]]
```

In [142]:
```
a[b]                                        # 1d array with the selected elements
```

Out[142]:
```
array([ 5,  6,  7,  8,  9, 10, 11])
```

This property can be very useful in assignments:

In [143]:
```
print('\na:')
print(a)
a[b] = 0                                          # All elements of 'a' higher than 4 b
ecome 0
a
print('\na[a > 4] = 0:')
print(a)
```

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

a[a > 4] = 0:
[[0 1 2 3]
 [4 0 0 0]
 [0 0 0 0]]
```

You can look at the following example to see how to use boolean indexing to generate an image of the Mandelbrot set:

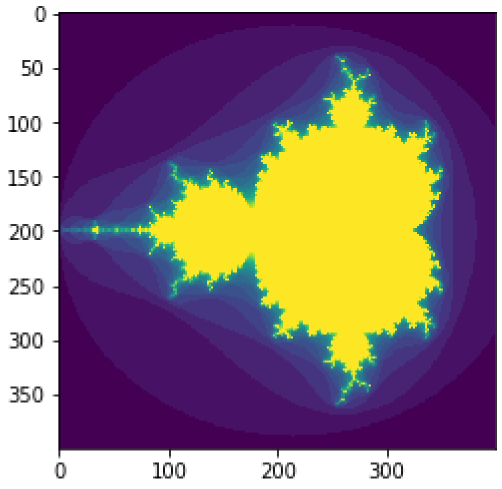
In [144]:

```
import numpy as np
import matplotlib.pyplot as plt
def mandelbrot( h,w, maxit=20 ):
    """Returns an image of the Mandelbrot fractal of size (h,w)."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
    c = x+y*1j
    z = c
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + c
        diverge = z*np.conj(z) > 2**2            # who is diverging
        div_now = diverge & (divtime==maxit)  # who is diverging now
        divtime[div_now] = i                  # note when
        z[diverge] = 2                        # avoid diverging too much

    return divtime

plt.imshow(mandelbrot(400,400))
plt.show()
```



The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want:

In [145]:
```
a = np.arange(12).reshape(3,4)
b1 = np.array([False,True,True])             # first dim selection
b2 = np.array([True,False,True,False])       # second dim selection
a[b1,:]                                   # selecting rows
```

Out[145]:
```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [146]:

```
a[b1]                                              # same thing
```

Out[146]:

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [147]:

```
a[:,b2]                                           # selecting columns
```

Out[147]:

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

In [148]:
```
a[b1,b2]                                # a weird thing to do
```
Out[148]:
```
array([ 4, 10])
```

Note that the length of the 1D boolean array must coincide with the length of the dimension (or axis) you want to slice. In the previous example, b1 has length 3 (the number of rows in a), and b2 (of length 4) is suitable to index the 2nd axis (columns) of a.

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:

In [149]:

```python
a = np.array([2,3,4,5])
a
```

Out[149]:

```
array([2, 3, 4, 5])
```

In [150]:

```
b = np.array([8,5,4])
b
```

Out[150]:

```
array([8, 5, 4])
```

In [151]:
```
c = np.array([5,4,6,8,3])
c
```
Out[151]:
```
array([5, 4, 6, 8, 3])
```

In [152]:
```
ax,bx,cx = np.ix_(a,b,c)
ax
```

Out[152]:
```
array([[[2]],

       [[3]],

       [[4]],

       [[5]]])
```

In [153]:

```
bx
```

Out[153]:

```
array([[[8],
        [5],
        [4]]])
```

In [154]:
```
cx
```

Out[154]:
```
array([[[5, 4, 6, 8, 3]]])
```

In [155]:
```
ax.shape, bx.shape, cx.shape
```

Out[155]:
```
((4, 1, 1), (1, 3, 1), (1, 1, 5))
```

In [156]:
```
result = ax+bx*cx
result
```

Out[156]:
```
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],

       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],

       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],

       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
```

In [157]:
```
result[3,2,4]
```
Out[157]:
```
17
```

In [158]:

```
a[3]+b[2]*c[4]
```

Out[158]:

```
17
```

You could also implement the reduce as follows:

In [159]:

```python
def ufunc_reduce(ufct, *vectors):
    vs = np.ix_(*vectors)
    r = ufct.identity
    for v in vs:
        r = ufct(r,v)
    return r

ufunc_reduce(np.add,a,b,c)
```

Out[159]:

```
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],

       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],

       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],

       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

The advantage of this version of reduce compared to the normal ufunc.reduce is that it makes use of the Broadcasting Rules in order to avoid creating an argument array the size of the output times the number of vectors.

Indexing with strings

See Structured arrays.

Linear Algebra

In [160]:

```
import numpy as np
a = np.array([[1.0, 2.0], [3.0, 4.0]])
print(a)
b = a.transpose()
print(b)
```

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

In [161]:

```
np.linalg.inv(a)
```

Out[161]:

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

In [162]:

```
u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
u
```

Out[162]:

```
array([[1., 0.],
       [0., 1.]])
```

In [163]:

```
j = np.array([[0.0, -1.0], [1.0, 0.0]])
j @ j        # matrix product
```

Out[163]:

```
array([[-1.,  0.],
       [ 0., -1.]])
```

In [164]:

```
np.trace(u)  # trace
```

Out[164]:

```
2.0
```

In [165]:

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

Out[165]:

```
array([[-3.],
       [ 4.]])
```

In [166]:

```
np.linalg.eig(j)
```

Out[166]:

```
(array([0.+1.j, 0.-1.j]),
 array([[0.70710678+0.j        , 0.70710678-0.j        ],
        [0.        -0.70710678j, 0.        +0.70710678j]]))
```

Tricks and Tips

"Automatic" Reshaping

To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

In [167]:
```
a = np.arange(30)
a.shape = 2,-1,3  # -1 means "whatever is needed"
a.shape
```

Out[167]:
```
(2, 5, 3)
```

In [168]:
```
a
```

Out[168]:
```
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]]])
```

Vector Stacking

How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if x and y are two vectors of the same length you only need do m=[x;y]. In NumPy this works via the functions column_stack, dstack, hstack and vstack, depending on the dimension in which the stacking is to be done. For example:

In [169]:
```
x = np.arange(0,10,2)                          # x=([0,2,4,6,8])
y = np.arange(5)                               # y=([0,1,2,3,4])
m = np.vstack([x,y])                           # m=([[0,2,4,6,8],
                                               #     [0,1,2,3,4]])

m
```

Out[169]:
```
array([[0, 2, 4, 6, 8],
       [0, 1, 2, 3, 4]])
```

In [170]:

```
xy = np.hstack([x,y])                    # xy =([0,2,4,6,8,0,1,2,3,4])
xy
```

Out[170]:

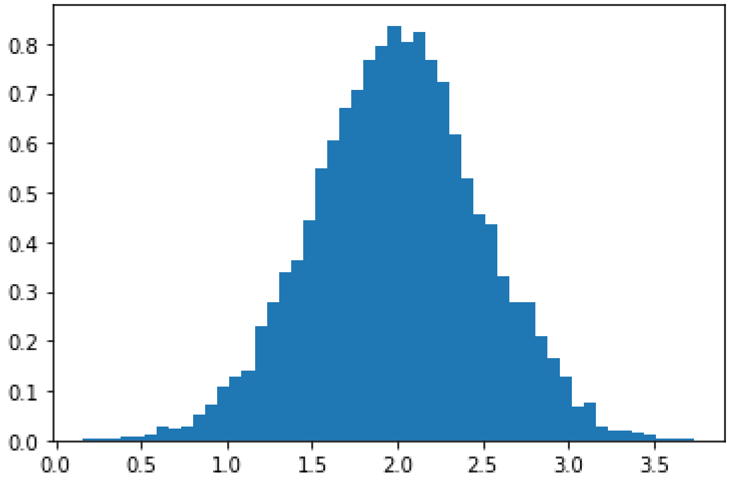
```
array([0, 2, 4, 6, 8, 0, 1, 2, 3, 4])
```

Histograms

The NumPy histogram function applied to an array returns a pair of vectors: the histogram of the array and the vector of bins. Beware: matplotlib also has a function to build histograms (called hist, as in Matlab) that differs from the one in NumPy. The main difference is that pylab.hist plots the histogram automatically, while numpy.histogram only generates the data.

In [171]:

```
import numpy as np
import matplotlib.pyplot as plt
# Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
mu, sigma = 2, 0.5
v = np.random.normal(mu,sigma,10000)
# Plot a normalized histogram with 50 bins
plt.hist(v, bins=50, density=1)       # matplotlib version (plot)
plt.show()
```



In [172]:

```
# Compute the histogram with numpy and then plot it
(n, bins) = np.histogram(v, bins=50, density=True)  # NumPy version (no plot)
plt.plot(.5*(bins[1:]+bins[:-1]), n)
plt.show()
```