UsingNumpy

February 21, 2015

```
In [62]: import os
         import sys
         import glob
         import matplotlib.pyplot as plt
         import numpy as np
         import pandas as pd
         %matplotlib inline
         %precision 4
         plt.style.use('ggplot')
```

```
In [63]: %install_ext http://raw.github.com/jrjohansson/version_information/master/version_information.p
```

```
Installed version_information.py. To use it, type:
  %load_ext version_information
```

0.1 References

- Basics of numpy
- Advanced numpy
- Some numpy exercises
- Numpy reference
- Numpy for Matlab users
- Numpy for R users
- Tutorials on Pandas
- Blaze documentation

0.2 Why is `numpy` important?

1. Speed - `numpy` calculations are based on C
2. Convenient for working with arrays - extended indexing and sliciing, broadcasting, ufuncs
3. Numpy provides very useful libraries - e.g. `random`, `linalg`
4. Foundation for essentialy all Python numerical libraries - e.g. pandas, scipy, pymc

```
In [64]: x = np.arange(1000000)
         print x.dtype

         def my_sum(x):
             s = 0
             for _ in x:
                 s += _
             return s

         %timeit -n3 my_sum(x) # explicit for loops are slow
```

```
%timeit -n3 sum(x) # built-in sum() cannot assume all values in x have same data type
%timeit -n3 np.sum(x) # numpy sum() knows that the dataytpe of x is np.int64
```

```
int64
3 loops, best of 3: 215 ms per loop
3 loops, best of 3: 149 ms per loop
3 loops, best of 3: 2.17 ms per loop
```

0.3 Example using numpy

From http://scipy-lectures.github.io/intro/numpy/exercises.html#data-statistics

The data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years:

Computes and print, based on the data in populations.txt…

- The mean and std of the populations of each species for the years in the period.
- Which year each species had the largest population.
- Which species has the largest population for each year. (Hint: argsort & fancy indexing of np.array(['H', 'L', 'C']))
- Which years any of the populations is above 50000. (Hint: comparisons and np.any)
- The top 2 years for each species when they had the lowest populations. (Hint: argsort, fancy indexing)
- Compare (plot) the change in hare population (see help(np.gradient)) and the number of lynxes - Check correlation (see help(np.corrcoef)).

… all without for-loops.

In [65]:
```
# download the data locally
if not os.path.exists('populations.txt'):
    ! wget http://scipy-lectures.github.io/_downloads/populations.txt
```

In [66]:
```
# peek at the file to see its structure
! head -n 6 populations.txt
```

```
# year        hare        lynx        carrot
1900        30e3        4e3        48300
1901        47.2e3        6.1e3        48200
1902        70.2e3        9.8e3        41500
1903        77.4e3        35.2e3        38200
1904        36.3e3        59.4e3        40600
```

In [67]:
```
# load data into a numpy array (text I/O convenience functions)
data = np.loadtxt('populations.txt').astype('int')
data[:5, :]
```

Out[67]:
```
array([[ 1900, 30000,  4000, 48300],
       [ 1901, 47200,  6100, 48200],
       [ 1902, 70200,  9800, 41500],
       [ 1903, 77400, 35200, 38200],
       [ 1904, 36300, 59400, 40600]])
```

In [68]:
```
# provide convenient named variables (indexing and slicing)
populations = data[:, 1:]
year, hare, lynx, carrot = data.T
```

In [69]:
```
# The mean and std of the populations of each species for the years in the period (use of axis
print "Mean (hare, lynx, carrot):", populations.mean(axis=0)
print "Std (hare, lynx, carrot):", populations.std(axis=0)
```

```
Mean (hare, lynx, carrot): [ 34080.9524  20166.6667  42400.    ]
Std (hare, lynx, carrot): [ 20897.9065  16254.5915   3322.5062]
```

```
In [70]: # Which year each species had the largest population. (argmax and axis)
         print "Year with largest population (hare, lynx, carrot)",
         print year[np.argmax(populations, axis=0)]
```

```
Year with largest population (hare, lynx, carrot) [1903 1904 1900]
```

```
In [71]: # Which species has the largest population for each year. (subsetting)
         species = ['hare', 'lynx', 'carrot']
         zip(year, np.take(species, np.argmax(populations, axis=1)))
```

```
Out[71]: [(1900, 'carrot'),
          (1901, 'carrot'),
          (1902, 'hare'),
          (1903, 'hare'),
          (1904, 'lynx'),
          (1905, 'lynx'),
          (1906, 'carrot'),
          (1907, 'carrot'),
          (1908, 'carrot'),
          (1909, 'carrot'),
          (1910, 'carrot'),
          (1911, 'carrot'),
          (1912, 'hare'),
          (1913, 'hare'),
          (1914, 'hare'),
          (1915, 'lynx'),
          (1916, 'carrot'),
          (1917, 'carrot'),
          (1918, 'carrot'),
          (1919, 'carrot'),
          (1920, 'carrot')]
```

```
In [72]: # Which years any of the populations is above 50000 (logical indexing)
         print year[np.any(populations > 50000, axis=1)]
```

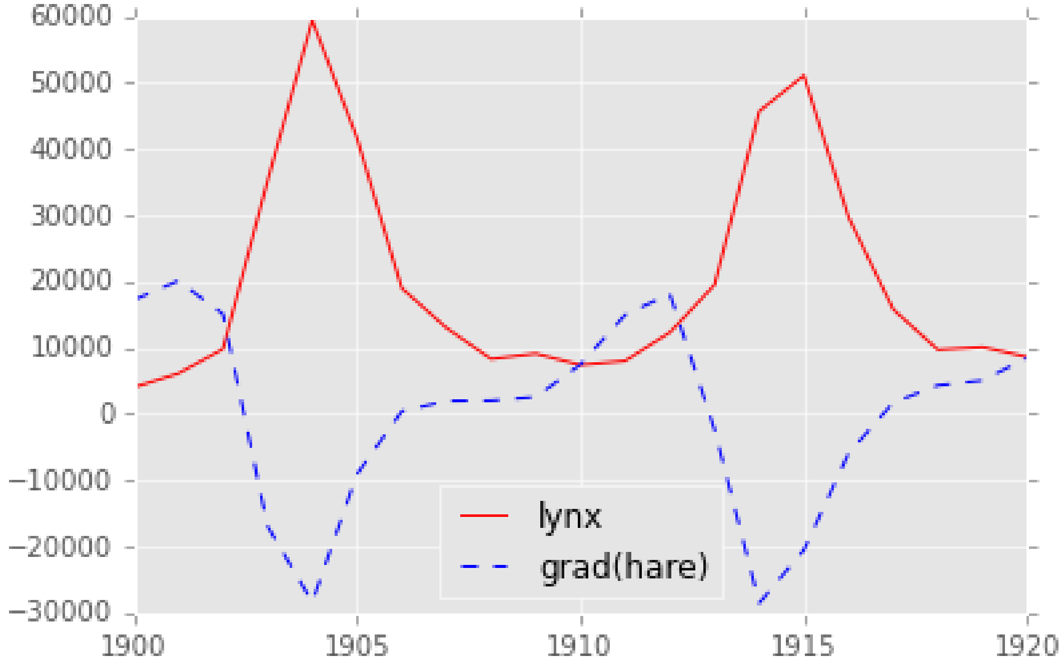
```
[1902 1903 1904 1912 1913 1914 1915]
```

```
In [73]: # The top 2 years for each species when they had the lowest populations. (sorting)
         print year[np.argsort(populations, axis=0)[:2]]
```

```
[[1917 1900 1916]
 [1916 1901 1903]]
```

```
In [74]: # works with plotting routines; numeritcal differencing to find gradient
         plt.plot(year, lynx, 'r-', year, np.gradient(hare), 'b--')
         plt.legend(['lynx', 'grad(hare)'], loc='best')
         print np.corrcoef(lynx, np.gradient(hare))
```

```
[[ 1.     -0.9179]
 [-0.9179  1.    ]]
```



1 Numerical computing in Python

1.0.1 Numpy data types

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa

Data type	Description
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Numpy does not have a *bit* data type - the bool_ data type takes up 1 byte or 8 bits. If you need to work with very large bit arrays, you can use the bitarray package.

```
In [75]: np.sctypes
```

```
Out[75]: {'complex': [numpy.complex64, numpy.complex128, numpy.complex256],
          'float': [numpy.float16, numpy.float32, numpy.float64, numpy.float128],
          'int': [numpy.int8, numpy.int16, numpy.int32, numpy.int64],
          'others': [bool, object, str, unicode, numpy.void],
          'uint': [numpy.uint8, numpy.uint16, numpy.uint32, numpy.uint64]}
```

1.0.2 Using character codes

There are also some possibly cryptic typecodes you can uase as a shorthand, but these are discouraged because, well, they are cryptic.

```
In [76]: np.typecodes
```

```
Out[76]: {'All': '?bhilqpBHILQPefdgFDGSUVOMm',
          'AllFloat': 'efdgFDG',
          'AllInteger': 'bBhHiIlLqQpP',
          'Character': 'c',
          'Complex': 'FDG',
          'Datetime': 'Mm',
          'Float': 'efdg',
          'Integer': 'bhilqp',
          'UnsignedInteger': 'BHILQP'}
```

```
In [77]: # Example use of typecode to specify dtype
         x = np.zeros(1, dtype='f8')
         print x.dtype
         print np.sctypeDict['f8']
```

```
float64
<type 'numpy.float64'>
```

1.0.3 NDArray

The base structure in `numpy` is `ndarray`, used to represent vectors, matrices and higher-dimensional arrays. Each `ndarray` has the following attributes:

- dtype = correspond to data types in C
- shape = dimensionns of array
- strides = number of bytes to step in each direction when traversing the array

Notes

1. That a 3-vector is most ofen specified with a shape (3,) rather than as an explicit column vector with shape (3,1) or row vector with shape (1,3). Most of the time, this will "just work", but if necessary, you can coerce to a desired shape with the **resahpe** method or function or by directly modifying the **shape** attribute.
2. Numpy arrays are created in row-order - the first row is filled up first, then the second and so on. For R users, this is like setting `byrow=TRUE` in a call to `matrix()`.

In [78]:
```
np.array([1,2,3,4,5,6], order='F').reshape(2,3)
```

Out[78]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [79]:
```
x = np.array([1,2,3,4,5,6]) # create array from list
print x
print 'dytpe', x.dtype
print 'shape', x.shape
print 'strides', x.strides
```

```
[1 2 3 4 5 6]
dytpe int64
shape (6,)
strides (8,)
```

In [80]:
```
x.shape = (2,3)
print x
print 'dytpe', x.dtype
print 'shape', x.shape
print 'strides', x.strides
```

```
[[1 2 3]
 [4 5 6]]
dytpe int64
shape (2, 3)
strides (24, 8)
```

In [81]:
```
x = x.astype('complex')
print x
print 'dytpe', x.dtype
print 'shape', x.shape
print 'strides', x.strides
```

```
[[ 1.+0.j  2.+0.j  3.+0.j]
 [ 4.+0.j  5.+0.j  6.+0.j]]
dytpe complex128
shape (2, 3)
strides (48, 16)
```

1.0.4 Creating arrays

In [82]:
```
# from lists
x_list = [(i,j) for i in range(2) for j in range(3)]
print x_list, '\n'
x_array = np.array(x_list)
print x_array
```

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

[[0 0]
 [0 1]
 [0 2]
 [1 0]
 [1 1]
 [1 2]]
```

In [83]:
```
# Using convenience functions

print np.ones((3,2)), '\n'
print np.zeros((3,2)), '\n'
print np.eye(3), '\n'
print np.diag([1,2,3]), '\n'
print np.fromfunction(lambda i, j: (i-2)**2+(j-2)**2, (5,5))
```

```
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]

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

[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]

[[1 0 0]
 [0 2 0]
 [0 0 3]]

[[ 8.  5.  4.  5.  8.]
 [ 5.  2.  1.  2.  5.]
 [ 4.  1.  0.  1.  4.]
 [ 5.  2.  1.  2.  5.]
 [ 8.  5.  4.  5.  8.]]
```

1.0.5 Array indexing

In [84]:
```
# Create a 10 by 6 array from normal deviates and convert to ints
n, nrows, ncols = 100, 10, 6
xs = np.random.normal(n, 15, size=(nrows, ncols)).astype('int')
xs
```

Out[84]:
```
array([[ 95,  82,  94, 116, 111,  84],
       [107,  82,  98, 105, 114,  66],
       [ 99,  94,  81, 100,  78,  78],
       [ 89, 115,  82, 113,  76, 113],
       [ 90,  91,  98, 109,  96,  60],
       [ 79,  96, 102,  93,  76,  98],
       [105,  91,  96, 106, 117, 124],
       [108, 113,  83, 117, 102, 114],
       [101, 104,  97, 100, 127,  85],
       [ 78,  65, 123,  82, 103,  70]])
```

```
In [85]: # Use slice notation
         print(xs[0,0])
         print(xs[-1,-1])
         print(xs[3,:])
         print(xs[:,0])
         print(xs[::2,::2])
         print(xs[2:5,2:5])
```

```
95
70
[ 89 115  82 113  76 113]
[ 95 107  99  89  90  79 105 108 101  78]
[[ 95  94 111]
 [ 99  81  78]
 [ 90  98  96]
 [105  96 117]
 [101  97 127]]
[[ 81 100  78]
 [ 82 113  76]
 [ 98 109  96]]
```

```
In [86]: #  Indexing with list of integers
         print(xs[0, [1,2,4,5]])
```

```
[ 82  94 111  84]
```

```
In [87]: # Boolean indexing
         print(xs[xs % 2 == 0])
         xs[xs % 2 == 0] = 0 # set even entries to zero
         print(xs)
```

```
[ 82  94 116  84  82  98 114  66  94 100  78  78  82  76  90  98  96  60
  96 102  76  98  96 106 124 108 102 114 104 100  78  82  70]
[[ 95   0   0   0 111   0]
 [107   0   0 105   0   0]
 [ 99   0  81   0   0   0]
 [ 89 115   0 113   0 113]
 [  0  91   0 109   0   0]
 [ 79   0   0  93   0   0]
 [105  91   0   0 117   0]
 [  0 113  83 117   0   0]
 [101   0  97   0 127  85]
 [  0  65 123   0 103   0]]
```

```
In [88]: # Extracting lower triangular, diagonal and upper triangular matrices

         a = np.arange(16).reshape(4,4)
         print a, '\n'
         print np.tril(a, -1), '\n'
         print np.diag(np.diag(a)), '\n'
         print np.triu(a, 1)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
[[ 0  0  0  0]
 [ 4  0  0  0]
 [ 8  9  0  0]
 [12 13 14  0]]

[[ 0  0  0  0]
 [ 0  5  0  0]
 [ 0  0 10  0]
 [ 0  0  0 15]]

[[ 0  1  2  3]
 [ 0  0  6  7]
 [ 0  0  0 11]
 [ 0  0  0  0]]
```

1.0.6 Broadcasting, row, column and matrix operations

```
In [89]: # operations across rows, cols or entire matrix - xs has shape (10,6)
         print(xs.max())
         print(xs.max(axis=0)) # max of each col
         print(xs.max(axis=1)) # max of each row
```

```
127
[107 115 123 117 127 113]
[111 107  99 115 109  93 117 117 127 123]
```

```
In [90]: # A functional rather than object-oriented approacha also wokrs
         print(np.max(xs, axis=0))
         print(np.max(xs, axis=1))
```

```
[107 115 123 117 127 113]
[111 107  99 115 109  93 117 117 127 123]
```

Broadcasting

```
In [91]: from IPython.display import Image
```

```
In [92]: Image(url="https://scipy-lectures.github.io/_images/numpy_broadcasting.png")
```

```
Out[92]: <IPython.core.display.Image at 0x114039290>
```

```
In [93]: x1 = np.repeat([0,10,20,30], 3).reshape((4,3))
         print x1
         y1 = np.tile([0,1,2], 4).reshape((4,3))
         print y1
         y2 = y1[[0], :]
         print y2
         x2 = x1[:, [0]]
         print x2
```

```
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
[[0 1 2]
```

```
 [0 1 2]
 [0 1 2]
 [0 1 2]]
[[0 1 2]]
[[ 0]
 [10]
 [20]
 [30]]
```

```
In [94]: print x1 + y1
         print
         print x1 + y2
         print
         print x2 + y2
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]

[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]

[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

1.0.7 Adding a new axis to meet broadcasting rules

```
In [150]: xs = np.arange(12).reshape(2,6)
          print(xs, '\n')
          print(xs * 10, '\n')

          # broadcasting just works when doing column-wise operations
          # xs is (2, 6)
          # col_meeans is (6,) -> this works because the 6s line up
          col_means = xs.mean(axis=0)
          print(col_means, '\n')
          print(xs + col_means, '\n')

          # but needs a little more work for row-wise operations
          # xs is (2, 6)
          # row means is (2,) -> we want (2,1) for broadcasting
          row_means = xs.mean(axis=1)[:, np.newaxis]
          print(row_means)
          print(xs + row_means)
```

```
(array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]]), '\n')
(array([[  0,  10,  20,  30,  40,  50],
       [ 60,  70,  80,  90, 100, 110]]), '\n')
(array([ 3.,  4.,  5.,  6.,  7.,  8.]), '\n')
```

```
(array([[  3.,   5.,   7.,   9.,  11.,  13.],
       [  9.,  11.,  13.,  15.,  17.,  19.]]), '\n')
[[ 2.5]
 [ 8.5]]
[[  2.5   3.5   4.5   5.5   6.5   7.5]
 [ 14.5  15.5  16.5  17.5  18.5  19.5]]
```

```
In [151]: # convert matrix to have zero mean and unit standard deviation using col summary statistics
          mu = xs.mean(axis=0)
          sd = xs.std(axis=0)
          print (xs - mu)/sd
```

```
[[-1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.]]
```

```
In [152]: # convert matrix to have zero mean and unit standard deviation using row summary statistics
          mu = xs.mean(axis=1)[:, np.newaxis]
          sd = xs.mean(axis=1)[:, np.newaxis]
          print (xs - mu)/sd
```

```
[[-1.     -0.6    -0.2     0.2     0.6     1.    ]
 [-0.2941 -0.1765 -0.0588  0.0588  0.1765  0.2941]]
```

```
In [98]: # broadcasting for outer product
         # e.g. create the 12x12 multiplication toable
         u = np.arange(1, 13)
         u[:,None] * u[None,:]
```

```
Out[98]: array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12],
                [  2,   4,   6,   8,  10,  12,  14,  16,  18,  20,  22,  24],
                [  3,   6,   9,  12,  15,  18,  21,  24,  27,  30,  33,  36],
                [  4,   8,  12,  16,  20,  24,  28,  32,  36,  40,  44,  48],
                [  5,  10,  15,  20,  25,  30,  35,  40,  45,  50,  55,  60],
                [  6,  12,  18,  24,  30,  36,  42,  48,  54,  60,  66,  72],
                [  7,  14,  21,  28,  35,  42,  49,  56,  63,  70,  77,  84],
                [  8,  16,  24,  32,  40,  48,  56,  64,  72,  80,  88,  96],
                [  9,  18,  27,  36,  45,  54,  63,  72,  81,  90,  99, 108],
                [ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110, 120],
                [ 11,  22,  33,  44,  55,  66,  77,  88,  99, 110, 121, 132],
                [ 12,  24,  36,  48,  60,  72,  84,  96, 108, 120, 132, 144]])
```

Example: Calculating pairwise distance matrix using broadcasting and vectorization Calculate the pairwise distance matrix between the following points

- (0,0)
- (4,0)
- (4,3)
- (0,3)

```
In [99]: def distance_matrix_py(pts):
             """Returns matrix of pairwise Euclidean distances. Pure Python version."""
             n = len(pts)
             p = len(pts[0])
             m = np.zeros((n, n))
             for i in range(n):
```

```
        for j in range(n):
            s = 0
            for k in range(p):
                s += (pts[i,k] - pts[j,k])**2
            m[i, j] = s**0.5
    return m
```

```
In [100]: def distance_matrix_np(pts):
              """Returns matrix of pairwise Euclidean distances. Vectorized numpy version."""
              return np.sum((pts[None,:] - pts[:, None])**2, -1)**0.5
```

```
In [101]: pts = np.array([(0,0), (4,0), (4,3), (0,3)])
```

```
In [102]: distance_matrix_py(pts)
```

```
Out[102]: array([[ 0.,  4.,  5.,  3.],
                 [ 4.,  0.,  3.,  5.],
                 [ 5.,  3.,  0.,  4.],
                 [ 3.,  5.,  4.,  0.]])
```

```
In [103]: distance_matrix_np(pts)
```

```
Out[103]: array([[ 0.,  4.,  5.,  3.],
                 [ 4.,  0.,  3.,  5.],
                 [ 5.,  3.,  0.,  4.],
                 [ 3.,  5.,  4.,  0.]])
```

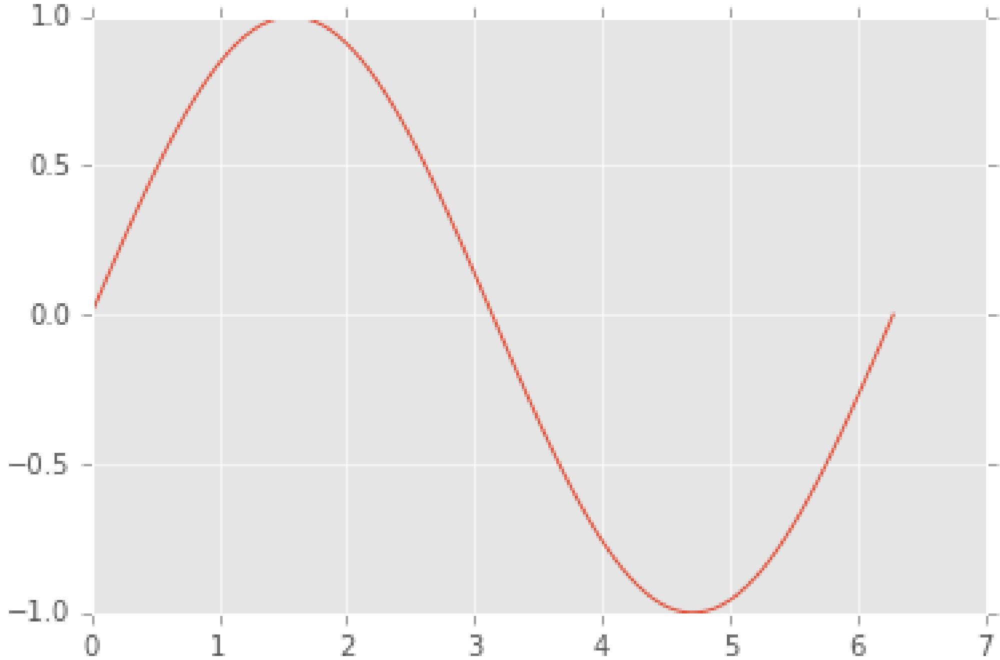
```
In [104]: # Broaccasting and vectorization is faster than looping
          %timeit distance_matrix_py(pts)
          %timeit distance_matrix_np(pts)
```

```
1000 loops, best of 3: 197 µs per loop
10000 loops, best of 3: 30.7 µs per loop
```

1.0.8 Universal functions (Ufuncs)

Functions that work on both scalars and arrays are known as ufuncs. For arrays, ufuncs apply the function in an element-wise fashion. Use of ufuncs is an esssential aspect of vectorization and typically much more computtionally efficient than using an explicit loop over each element.

```
In [105]: xs = np.linspace(0, 2*np.pi, 100)
          ys = np.sin(xs) # np.sin is a universal function
          plt.plot(xs, ys);
```



```
In [106]: # operators also perform elementwise operations by default

          xs = np.arange(10)
          print xs
          print -xs
          print xs+xs
          print xs*xs
          print xs**3
          print xs < 5
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0 -1 -2 -3 -4 -5 -6 -7 -8 -9]
[ 0  2  4  6  8 10 12 14 16 18]
[ 0  1  4  9 16 25 36 49 64 81]
[  0   1   8  27  64 125 216 343 512 729]
[ True  True  True  True  True False False False False False]
```

1.0.9 Generalized ufucns

A universal function performs vectorized looping over scalars. A generalized ufucn performs looping over vectors or arrays. Currently, numpy only ships with a single generalized ufunc. However, they play an important role for JIT compilation with `numba`, a topic we will cover in future lectures.

```
In [107]: from numpy.core.umath_tests import matrix_multiply

          print matrix_multiply.signature
```

```
(m,n),(n,p)->(m,p)
```

```
In [108]: us = np.random.random((5, 2, 3)) # 5 2x3 matrics
          vs = np.random.random((5, 3, 4)) # 5 3x4 matrices
          # perform matrix multiplication for each of the 5 sets of matrices
          ws = matrix_multiply(us, vs)
          print ws.shape
          print ws
```

```
(5, 2, 4)
[[[ 0.6287  0.3012  0.9293  0.5439]
  [ 0.7728  0.6826  1.6712  0.8018]]

 [[ 0.4425  0.7135  0.9703  0.9812]
  [ 0.2253  0.7995  0.8728  0.8934]]

 [[ 1.2528  0.917   0.2512  0.7338]
  [ 2.0581  1.4807  0.3652  1.0819]]

 [[ 0.2102  0.4713  0.4547  0.2371]
  [ 0.0925  0.1447  0.1473  0.0707]]

 [[ 1.1039  0.5479  1.3152  1.0523]
  [ 1.3219  0.6746  1.596   1.2698]]]
```

1.0.10 Random numbers

There are two modules for (pseudo) random numbers that are commonly used. When all you need is to generate random numbers from some distribtuion, the `numpy.random` moodule is the simplest to use. When you need more information realted to a disttribution such as quantiles or the PDF, you can use the `scipy.stats` module.

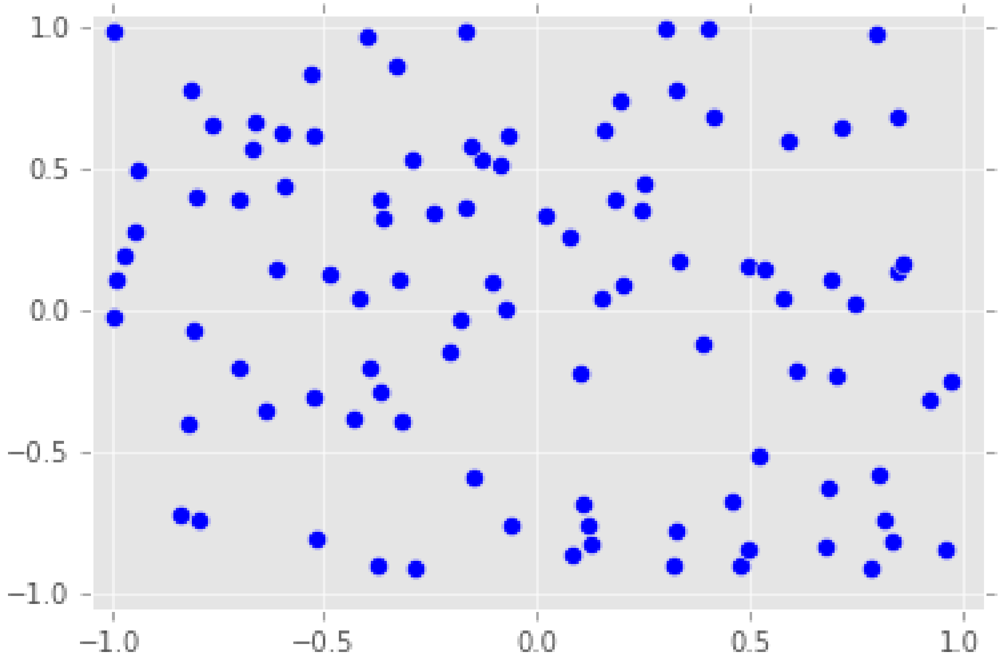
Using `numpy.random` Module Reference

```
In [109]: import numpy.random as npr
          npr.seed(123) # fix seed for reproducible results
```

```
In [110]: # 10 trials of rolling a fair 6-sided 100 times
          roll = 1.0/6
          x = npr.multinomial(100, [roll]*6, 10)
          x
```

```
Out[110]: array([[18, 14, 14, 18, 20, 16],
                 [16, 25, 16, 14, 14, 15],
                 [15, 19, 16, 12, 18, 20],
                 [19, 13, 14, 18, 18, 18],
                 [18, 20, 17, 16, 16, 13],
                 [15, 16, 15, 16, 20, 18],
                 [12, 17, 17, 18, 17, 19],
                 [15, 16, 22, 21, 13, 13],
                 [18, 12, 16, 17, 22, 15],
                 [14, 17, 25, 15, 15, 14]])
```

```
In [111]: # uniformly distributed numbers in 2D
          x = npr.uniform(-1, 1, (100, 2))
          plt.scatter(x[:,0], x[:,1], s=50)
          plt.axis([-1.05, 1.05, -1.05, 1.05]);
```




```
In [112]: # ranodmly shuffling a vector
          x = np.arange(10)
          npr.shuffle(x)
          x

Out[112]: array([5, 8, 6, 4, 3, 9, 1, 7, 2, 0])

In [113]: # radnom permutations
          npr.permutation(10)

Out[113]: array([1, 4, 9, 8, 6, 5, 3, 2, 0, 7])

In [114]: # radnom selection without replacement
          x = np.arange(10,20)
          npr.choice(x, 10, replace=False)

Out[114]: array([14, 16, 15, 12, 19, 11, 13, 10, 18, 17])

In [115]: # radnom selection with replacement
          npr.choice(x, (5, 10), replace=True) # this is default

Out[115]: array([[15, 13, 10, 14, 18, 14, 19, 13, 15, 11],
                 [18, 10, 19, 11, 15, 18, 18, 14, 16, 18],
                 [17, 19, 12, 10, 10, 19, 19, 15, 13, 15],
                 [15, 12, 12, 17, 13, 11, 13, 19, 13, 16],
                 [12, 13, 11, 19, 18, 10, 12, 13, 17, 19]])

In [116]: # toy example - estimating pi inefficiently
          n = 1e6
          x = npr.uniform(-1,1,(n,2))
          4.0*np.sum(x[:,0]**2 + x[:,1]**2 < 1)/n

Out[116]: 3.1416
```

Using scipy.stats Module refernce

```
In [117]: import scipy.stats as stats
```

```
In [118]: # Create a "frozen" distribution - i.e. a partially applied function
          dist = stats.norm(10, 2)
```

```
In [119]: #  same a rnorm
          dist.rvs(10)
```

```
Out[119]: array([ 11.629 ,   9.5777,   8.5607,   8.5777,   8.6464,  11.5398,
                  10.8751,  11.8244,  10.1772,   9.3056])
```

```
In [120]: # same as pnorm
          dist.pdf(np.linspace(5, 15, 10))
```

```
Out[120]: array([ 0.0088,  0.0301,  0.076 ,  0.141 ,  0.1919,  0.1919,  0.141 ,
                  0.076 ,  0.0301,  0.0088])
```

```
In [121]: # same as dnorm
          dist.cdf(np.linspace(5, 15, 11))
```

```
Out[121]: array([ 0.0062,  0.0228,  0.0668,  0.1587,  0.3085,  0.5   ,  0.6915,
                  0.8413,  0.9332,  0.9772,  0.9938])
```

```
In [122]: # same as qnorm
          dist.ppf(dist.cdf(np.linspace(5, 15, 11)))
```

```
Out[122]: array([  5.,   6.,   7.,   8.,   9.,  10.,  11.,  12.,  13.,  14.,  15.])
```

1.0.11 Linear algebra

In general, the linear algebra functions can be found in scipy.linalg. You can also get access to BLAS and LAPACK function via scipy.linagl.blas and scipy.linalg.lapack.

```
In [123]: import scipy.linalg as la
```

```
In [124]: A = np.array([[1,2],[3,4]])
          b = np.array([1,4])
          print(A)
          print(b)
```

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

```
In [125]: # Matrix operations
          import numpy as np
          import scipy.linalg as la
          from functools import reduce

          A = np.array([[1,2],[3,4]])
          print(np.dot(A, A))
          print(A)
          print(la.inv(A))
          print(A.T)
```

```
[[ 7 10]
 [15 22]]
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1 3]
 [2 4]]
```

```
In [126]: x = la.solve(A, b) # do not use x = dot(inv(A), b) as it is inefficient and numerically unsta
          print(x)
          print(np.dot(A, x) - b)
```

```
[ 2.  -0.5]
[ 0.  0.]
```

1.0.12 Matrix decompositions

```
In [127]: A = np.floor(npr.normal(100, 15, (6, 10)))
          print(A)
```

```
[[  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
 [ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]]
```

```
In [128]: P, L, U = la.lu(A)
          print(np.dot(P.T, A))
          print
          print(np.dot(L, U))
```

```
[[ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]]

[[ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]]
```

```
In [129]: Q, R = la.qr(A)
          print(A)
          print
          print(np.dot(Q, R))
```

```
[[  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
```

```
 [ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]]

[[  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
 [ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]]
```

```
In [130]: U, s, V = la.svd(A)
          m, n = A.shape
          S =  np.zeros((m, n))
          for i, _s in enumerate(s):
              S[i,i] = _s
          print(reduce(np.dot, [U, S, V]))
```

```
[[  94.   82.  125.  108.  105.   88.   99.   82.   97.  112.]
 [  83.  124.   67.  103.   73.  111.  125.   81.  122.   62.]
 [  93.   84.  107.  107.   80.   85.   96.   89.   85.  102.]
 [ 116.  116.   64.   98.   82.   98.  121.   70.  122.   98.]
 [ 118.  108.  103.  102.   68.   98.   88.   78.  103.   95.]
 [ 112.  115.   74.   80.  106.  104.  114.  105.   80.   99.]]
```

```
In [131]: B = np.cov(A)
          print(B)
```

```
[[ 187.7333 -182.4667   94.9333 -105.4444    1.2     -137.2   ]
 [-182.4667  609.6556  -83.3111  371.0556   90.8778   70.5667]
 [  94.9333  -83.3111   97.2889  -48.8889   45.0222  -79.8   ]
 [-105.4444  371.0556  -48.8889  438.5     145.5     109.0556]
 [   1.2      90.8778   45.0222  145.5     215.4333  -39.7667]
 [-137.2      70.5667  -79.8     109.0556  -39.7667  234.1   ]]
```

```
In [132]: u, V = la.eig(B)
          print(np.dot(B, V))
          print
          print(np.real(np.dot(V, np.diag(u))))
```

```
[[-280.8911  157.1032   12.1003  -60.7161    8.8142   -1.5134]
 [ 739.1179   34.4268    3.8974    4.3778   14.9092 -122.8749]
 [-134.1449  128.3162  -11.0569   -6.6382   37.3675   13.4467]
 [ 598.7992   77.4348   -5.3372  -52.7843  -14.996    94.553 ]
 [ 170.8339  193.7335    5.8732   67.6135    1.1042   90.1451]
 [ 199.7105 -218.1547    6.1467   -5.6295   26.3372  101.0444]]

[[-280.8911  157.1032   12.1003  -60.7161    8.8142   -1.5134]
 [ 739.1179   34.4268    3.8974    4.3778   14.9092 -122.8749]
 [-134.1449  128.3162  -11.0569   -6.6382   37.3675   13.4467]
 [ 598.7992   77.4348   -5.3372  -52.7843  -14.996    94.553 ]
 [ 170.8339  193.7335    5.8732   67.6135    1.1042   90.1451]
 [ 199.7105 -218.1547    6.1467   -5.6295   26.3372  101.0444]]
```

```
In [133]: C = la.cholesky(B)
          print(np.dot(C.T, C))
          print
          print(B)
```

```
[[ 187.7333 -182.4667   94.9333 -105.4444    1.2     -137.2   ]
 [-182.4667  609.6556  -83.3111  371.0556   90.8778   70.5667]
 [  94.9333  -83.3111   97.2889  -48.8889   45.0222  -79.8   ]
 [-105.4444  371.0556  -48.8889  438.5     145.5     109.0556]
 [   1.2      90.8778   45.0222  145.5     215.4333  -39.7667]
 [-137.2      70.5667  -79.8     109.0556  -39.7667  234.1   ]]

[[ 187.7333 -182.4667   94.9333 -105.4444    1.2     -137.2   ]
 [-182.4667  609.6556  -83.3111  371.0556   90.8778   70.5667]
 [  94.9333  -83.3111   97.2889  -48.8889   45.0222  -79.8   ]
 [-105.4444  371.0556  -48.8889  438.5     145.5     109.0556]
 [   1.2      90.8778   45.0222  145.5     215.4333  -39.7667]
 [-137.2      70.5667  -79.8     109.0556  -39.7667  234.1   ]]
```

1.0.13 Least squares solution

Suppose we want to solve a system of noisy linear equations

$$y_1 = b_0 x_1 + b_1 y_2 = b_0 x_2 + b_1 y_3 = b_0 x_2 + b_1 y_4 = b_0 x_4 + b_1$$

Since the system is noisy (implies full rank) and overdetermined, we cannot find an exact solution. Instead, we will look for the least squares solution. First we can rewrrite in matrix notation $Y = AB$, treating b_1 as the coefficient of $x^0 = 1$:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ x_4 & 1 \end{pmatrix} \begin{pmatrix} b_0 & b_1 \end{pmatrix}$$

The solution of this (i.e. the B matrix) is solved by multipling the psudoinverse of A (the Vandermonde matrix) with Y

$$(A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}Y$$

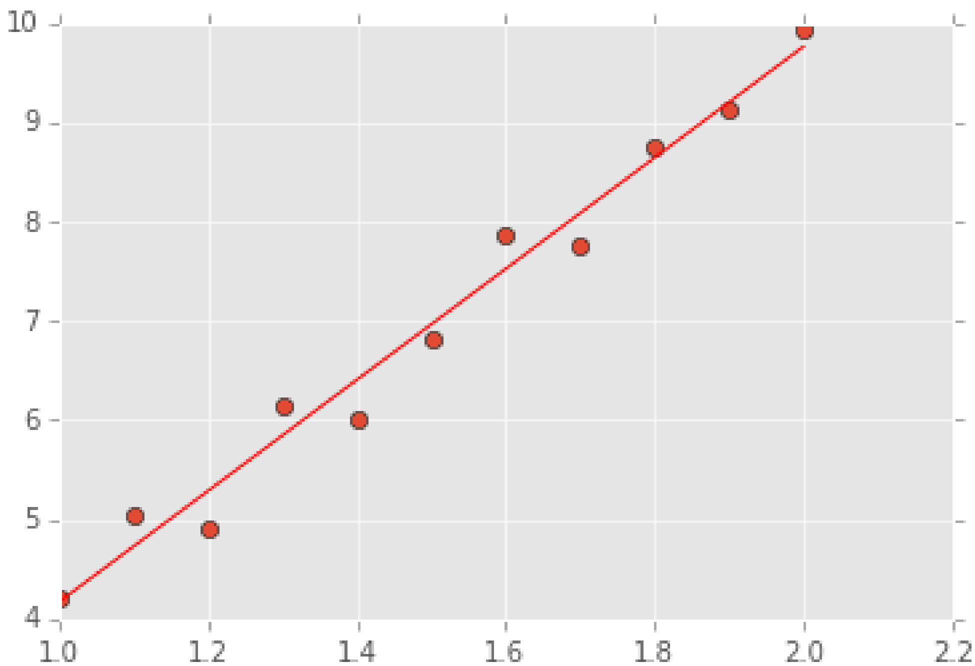
Note that higher order polynomials have the same structure and can be solved in the same way

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \\ x_4^2 & x_4 & 1 \end{pmatrix} \begin{pmatrix} b_0 & b_1 & b_2 \end{pmatrix}$$

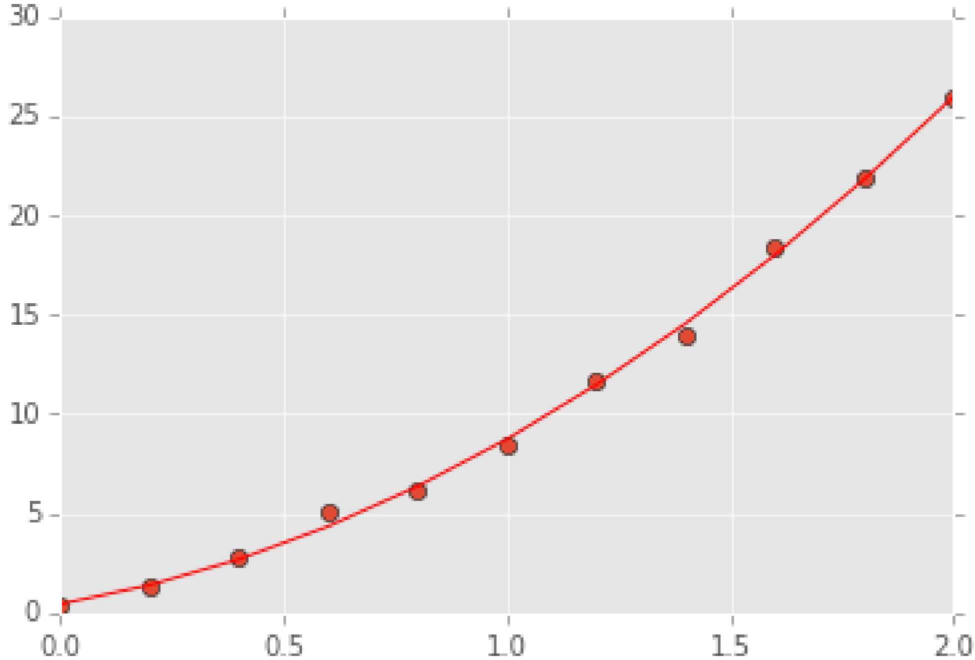
In [135]:
```
# Set up a system of 11 linear equations
x = np.linspace(1,2,11)
y = 6*x - 2 + npr.normal(0, 0.3, len(x))

# Form the VanderMonde matrix
A = np.vstack([x, np.ones(len(x))]).T

# The linear algebra librayr has a lstsq() function
# that will do the above calculaitons for us

b, resids, rank, sv = la.lstsq(A, y)

# Check against pseudoinverse and the normal equation
print("lstsq solution".ljust(30), b)
print("pseudoinverse solution".ljust(30), np.dot(la.pinv(A), y))
print("normal euqation solution".ljust(30), np.dot(np.dot(la.inv(np.dot(A.T, A)), A.T), y))
```

```
# Now plot the solution
xi = np.linspace(1,2,11)
yi = b[0]*xi + b[1]

plt.plot(x, y, 'o')
plt.plot(xi, yi, 'r-');
```

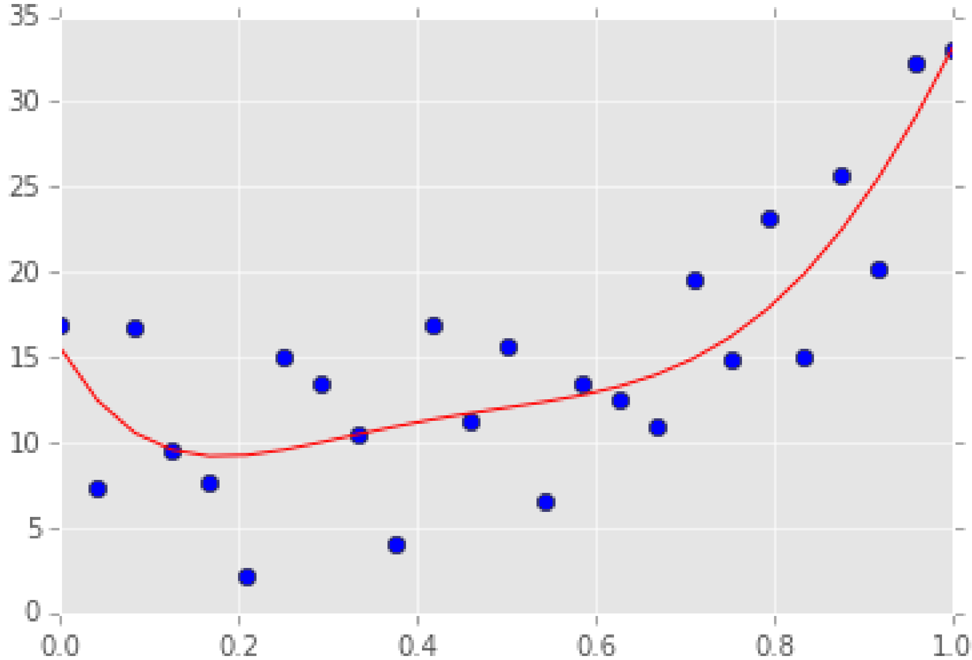
```
('lstsq solution                ', array([ 5.5899, -1.4177]))
('pseudoinverse solution        ', array([ 5.5899, -1.4177]))
('normal euqation solution      ', array([ 5.5899, -1.4177]))
```



In [136]:
```
# As advertised, this works for finding coeefficeints of a polynomial too

x = np.linspace(0,2,11)
y = 6*x*x + .5*x + 2 + npr.normal(0, 0.6, len(x))
plt.plot(x, y, 'o')
A = np.vstack([x*x, x, np.ones(len(x))]).T
b = la.lstsq(A, y)[0]

xi = np.linspace(0,2,11)
yi = b[0]*xi*xi + b[1]*xi + b[2]
plt.plot(xi, yi, 'r-');
```




```
In [137]: # It is important to understand what is going on,
          # but we don't have to work so hard to fit a polynomial

          b = np.random.randint(0, 10, 6)
          x = np.linspace(0, 1, 25)
          y = np.poly1d(b)(x)
          y += np.random.normal(0, 5, y.shape)

          p = np.poly1d(np.polyfit(x, y, len(b)-1))
          plt.plot(x, y, 'bo')
          plt.plot(x, p(x), 'r-')
          list(zip(b, p.coeffs))

Out[137]: [(6, -250.9964),
           (7, 819.7606),
           (1, -909.5724),
           (5, 449.7862),
           (7, -91.2660),
           (9, 15.5274)]
```



1.1 Exercises

1. Find the row, column and overall means for the following matrix:

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

```
In [138]: # YOUR CODE HERE
```

2. Find the outer product of the following two vecotrs

```
u = np.array([1,3,5,7])
v = np.array([2,4,6,8])
```

Do this in the following ways:

- Using the function `outer` in numpy
- Using a nested for loop or list comprehension
- Using numpy broadcasting operatoins

```
In [139]: # YOUR CODE HERE
```

3. Create a 10 by 6 matrix of random uniform numbers. Set all rows with *any* entry less than 0.1 to be zero. For example, here is a 4 by 10 version:

```
array([[ 0.49722235,  0.88833973,  0.07289358,  0.12375223,  0.39659254,
         0.70267114],
       [ 0.3954172 ,  0.889077  ,  0.71286225,  0.06353112,  0.68107965,
         0.17186995],
       [ 0.74821206,  0.92692111,  0.24871227,  0.26904958,  0.80410194,
         0.22304055],
       [ 0.22582605,  0.37671244,  0.96510957,  0.88819053,  0.14654176,
         0.33987323]])
```

becomes

```
array([[ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.74821206,  0.92692111,  0.24871227,  0.26904958,  0.80410194,
         0.22304055],
       [ 0.22582605,  0.37671244,  0.96510957,  0.88819053,  0.14654176,
         0.33987323]])
```

Hint: Use the following numpy functions - `np.random.random`, `np.any` as well as Boolean indexing and the axis argument.

```
In [140]: # YOUR CODE HERE
```

4. Use `np.linspace` to create an array of 100 numbers between 0 and 2π (includsive).

- Extract every 10th element using slice notation
- Reverse the array using slice notation
- Extract elements where the absolute difference between the sine and cosine functions evaluated at that element is less than 0.1
- Make a plot showing the sin and cos functions and indicate where they are close

```
In [141]: # YOUR CODE HERE
```

5. Create a matrix that shows the 10 by 10 multiplication table.

- Find the trace of the matrix
- Extract the anto-diagonal (this should be `array([10, 18, 24, 28, 30, 30, 28, 24, 18, 10])`)
- Extract the diagnoal offset by 1 upwards (this should be `array([ 2, 6, 12, 20, 30, 42, 56, 72, 90])`)

```
In [142]: # YOUR CODE HERE
```

6. Diagonalize the follwoing matrix

```
A = np.array([
    [1,  2, 1],
    [6, -1, 0],
    [-1,-2,-1]
])
```

In other words, find the invertible matrix P and the diagonal matrix D such that $A = PDP^{-1}$. Confirm by calculating the value of PDP^{-1}.

- Do this mnaully
- Then use numpy.linalg functions to do the same

```
In [143]: # YOUR CODE HERE
```

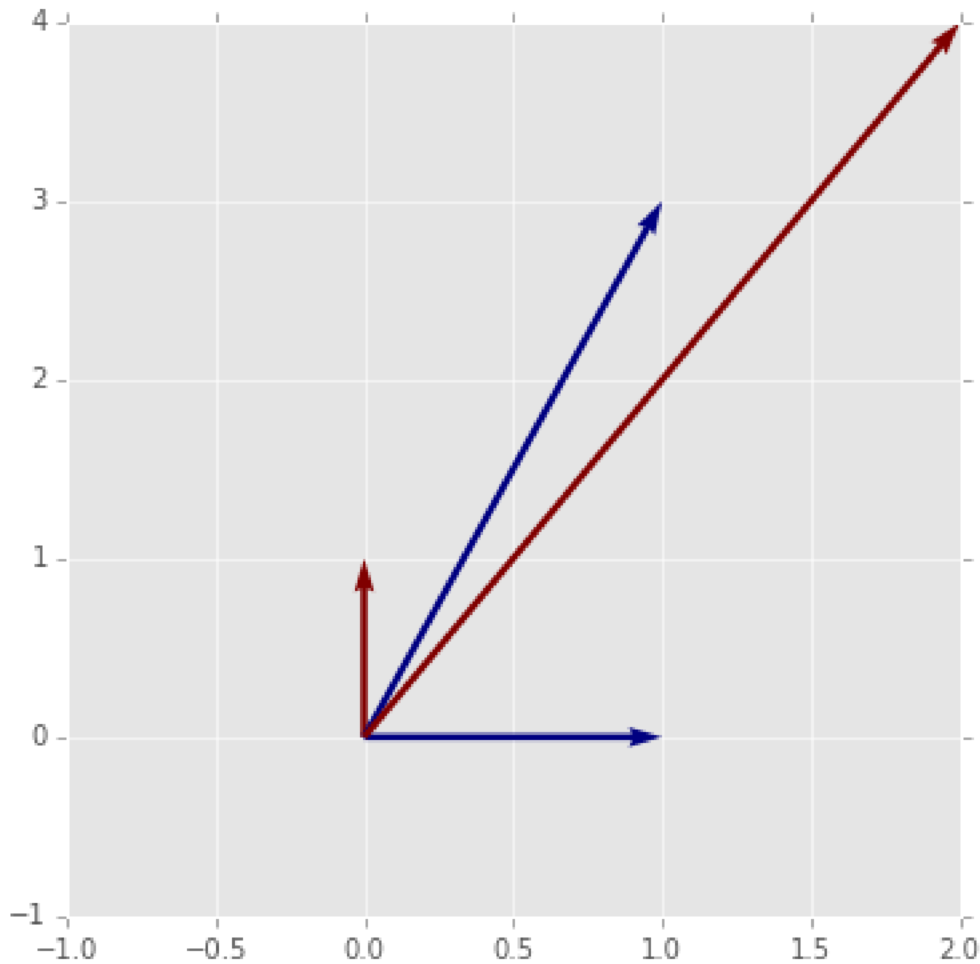
7. Use the function provided below to visualize matrix multiplication as a geometric transformation by experiment with differnt values of the matrix m.

- What does a diagonal matrix do to the origianl vectors?
- What does a non-invertible matrix do to the original vectors?

- What property results in matrices that preserves the area of the parallelogram spanned by the two vectors?
- What property results in matrices that also preserve the length and angle of the original vectors?
- What additional property is necessary to preserve the orientation of the original vecotrs?
- What does the transpose of the matrix that preserves the length and angle of the original vectors do?
- Write a function that when given any two non-colinear 2D vectors u, v, finds a transformation that converts u into e1 (1,0) and v into e2 (0,1).

In [144]:
```
#  Provided function
def plot_matrix_transform(m):
    """Show the geometric effect of m on the standard unit vectors e1 and e2."""

    e1 = np.array([1,0])
    e2 = np.array([0,1])
    v1 = np.dot(m, e1)
    v2 = np.dot(m, e2)

    X = np.zeros((2,2))
    Y = np.zeros((2,2))
    pts = np.array([e1,e2,v1,v2])
    U = pts[:, 0]
    V = pts[:, 1]
    C = [0,1,0,1]

    xmin = min(-1, U.min())
    xmax = max(1, U.max())
    ymin = min(-1, V.min())
    ymax = max(-1, V.max())

    plt.figure(figsize=(6,6))
    plt.quiver(X, Y, U, V, C, angles='xy', scale_units='xy', scale=1)
    plt.axis([xmin, xmax, ymin, ymax]);
```

In [145]:
```
### Example usage
m = np.array([[1,2],[3,4]])
plot_matrix_transform(m)
```



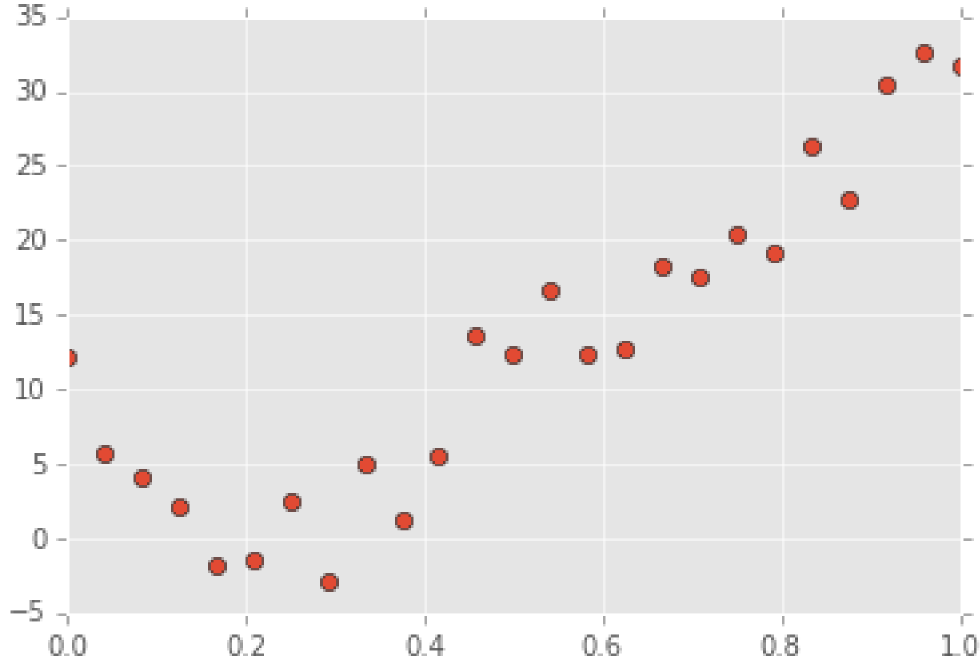

```
In [146]: # YOUR CODE HERE
```

8. Find and plot the least squares fit to the given values of x and y for the following:

- a constant
- a quadratic equation
- a 5th order polynomial
- a polynomial of order 50

```
In [147]: x = np.load('x.npy')
          y = np.load('y.npy')
          plt.plot(x, y, 'o')

          ### YOUR CODE HERE
```

```
Out[147]: [<matplotlib.lines.Line2D at 0x118cc3310>]
```



In [148]:
```
%load_ext version_information

%version_information numpy, scipy
```

Out[148]:

Software	Version
Python	2.7.9 64bit [GCC 4.2.1 (Apple Inc. build 5577)]
IPython	2.3.1
OS	Darwin 13.4.0 x86_64 i386 64bit
numpy	1.9.1
scipy	0.14.0
Wed Jan 21 11:51:33 2015 EST	

In [148]: