numpy-scipy

February 11, 2015

Contents

0.1 Introduction to Scientific Programming with Python

0.1.1 Session 4: NumPy, SciPy & matplotlib

Heiko Strathmann – heiko.strathmann@gmail.com – http://herrstrathmann.de/

0.1.2 Outline:

- Overview/Review
- Basic plotting with Matplotlib
- NumPy arrays
- Array object
- Indexing & slicing
- Useful function
- Broadcasting
- More advanced plotting
- SciPy intro

0.1.3 Review

So far we have talked about very generic programming constructs and about how they work in Python.

- Writing programs in files & running them with the Python interpreter
- Data types: Strings, lists, tuples dictionaries
- Flow control: loops & conditionals
- Structuring code: functions & classes

0.1.4 Scientific Programming

- What is scientific programming?
- Analyze data from scientific experiments
- “Number crunching”
- Turning mathematical formulae into code
- Aspects of scientific programming
- Handling data: vectors, matrices & arrays
- Visualizing data
- Computing with arrays
- Useful functions and algorithms

0.1.5 NumPy, SciPy, matplotlib

- NumPy
- a powerful n-dimensional array object
- useful linear algebra, Fourier transform, and random number capabilities
- SciPy
- built ontop of NumPy
- provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization
- matplotlib
- flexible 2D plotting library
- produces publication quality figures
- See http://matplotlib.sourceforge.net/users/screenshots.html

0.1.6 Importing numpy, scipy, and matplotlib

Start the Ipython process with

`ipython qtconsole` or `ipython notebook`, depending on your prefered interface.

Start your notebook with the following command - it enables the interactive plotting in your notebook or qtconsole:

```
In [1]: %matplotlib inline
```

or use
`%matplotlib qt`
if you are using the qtconsole interface

Get plotting tools

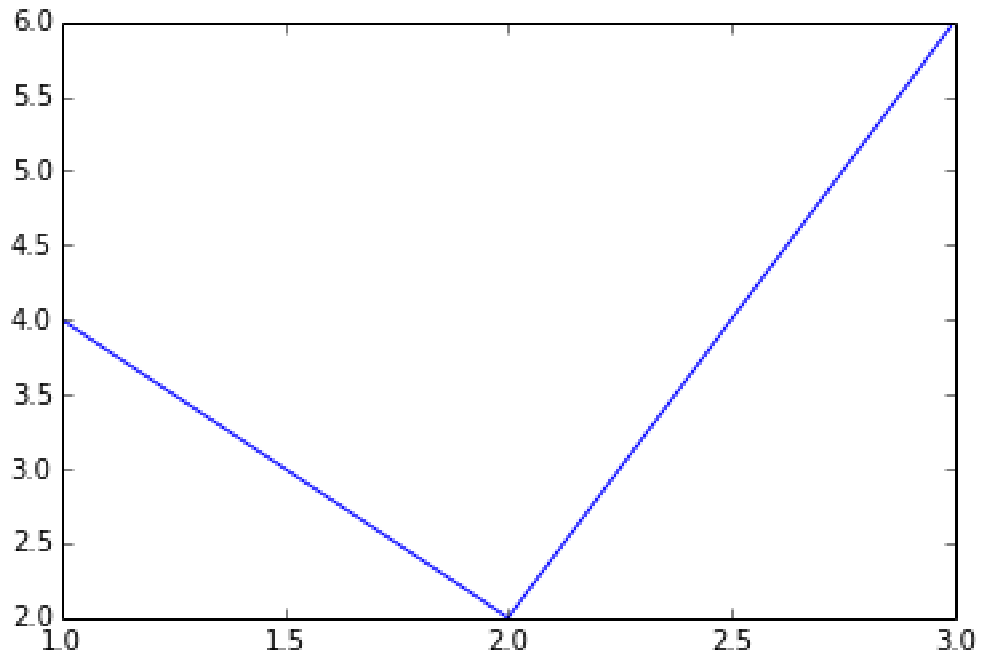
we need both plotting tools and numpy for this tutorial

```
In [2]: import matplotlib.pyplot as plt
        import numpy as np
```

0.1.7 Basic Plotting

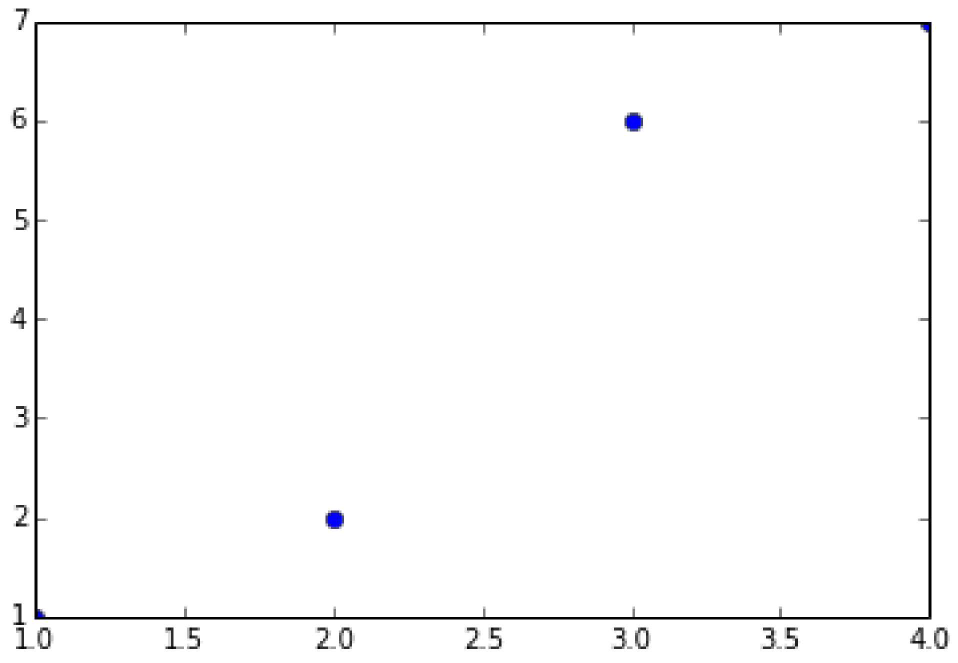
- Basic line and scatter plots can be created using the *plot* function from the *pylab* package
- Below, the first list determines the x-coordinates and the second list the y -coordinates.

```
In [3]: plt.plot([1,2,3],[4,2,6])

Out[3]: [<matplotlib.lines.Line2D at 0x9bb9ef0>]
```



- Use the *clf()* function to clear a figure, or the *figure()* function to start a new one.
- Commonly used markers are “.”, “o”, “x” and line types “-”, “–”, “-”. but have a look at the help of the plot command if you need more.

```
In [4]: plt.plot([1,2,3,4],[1,2,6,7], 'o');
```



Note the ; at the end of the plot statement above: it stops the statement from returning and so the [<matplotlib.lines.Line2D at 0xb64b0b8>] is not displayed.

```
In [5]: help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, **kwargs)
    Plot lines and/or markers to the
    :class:‘~matplotlib.axes.Axes‘.  *args* is a variable length
    argument, allowing for multiple *x*, *y* pairs with an
    optional format string.  For example, each of the following is
    legal::

        plot(x, y)        # plot x and y using default line style and color
        plot(x, y, ’bo’)  # plot x and y using blue circle markers
        plot(y)           # plot y using x as index array 0..N-1
        plot(y, ’r+’)     # ditto, but with red plusses

    If *x* and/or *y* is 2-dimensional, then the corresponding columns
    will be plotted.

    An arbitrary number of *x*, *y*, *fmt* groups can be
    specified, as in::

        a.plot(x1, y1, ’g^’, x2, y2, ’g-’)

    Return value is a list of lines that were added.

    By default, each line is assigned a different color specified by a
    ’color cycle’.  To change this behavior, you can edit the
    axes.color_cycle rcParam. Alternatively, you can use
    :meth:‘~matplotlib.axes.Axes.set_default_color_cycle‘.

    The following format string characters are accepted to control
    the line style or marker:

    ================    ===============================
    character           description
    ================    ===============================
    ‘‘’-’‘‘             solid line style
    ‘‘’--’‘‘            dashed line style
    ‘‘’-.’‘‘            dash-dot line style
    ‘‘’:’‘‘             dotted line style
    ‘‘’.’‘‘             point marker
    ‘‘’,’‘‘             pixel marker
    ‘‘’o’‘‘             circle marker
    ‘‘’v’‘‘             triangle_down marker
    ‘‘’^’‘‘             triangle_up marker
    ‘‘’<’‘‘             triangle_left marker
    ‘‘’>’‘‘             triangle_right marker
    ‘‘’1’‘‘             tri_down marker
    ‘‘’2’‘‘             tri_up marker
    ‘‘’3’‘‘             tri_left marker
    ‘‘’4’‘‘             tri_right marker
```

‘‘’s’‘‘	square marker
‘‘’p’‘‘	pentagon marker
‘‘’*’‘‘	star marker
‘‘’h’‘‘	hexagon1 marker
‘‘’H’‘‘	hexagon2 marker
‘‘’+’‘‘	plus marker
‘‘’x’‘‘	x marker
‘‘’D’‘‘	diamond marker
‘‘’d’‘‘	thin_diamond marker
‘‘’|’‘‘	vline marker
‘‘’_’‘‘	hline marker

The following color abbreviations are supported:

character	color
’b’	blue
’g’	green
’r’	red
’c’	cyan
’m’	magenta
’y’	yellow
’k’	black
’w’	white

In addition, you can specify colors in many weird and wonderful ways, including full names (‘‘’green’‘‘), hex strings (‘‘’#008000’‘‘), RGB or RGBA tuples (‘‘(0,1,0,1)‘‘) or grayscale intensities as a string (‘‘’0.8’‘‘). Of these, the string specifications can be used in place of a ‘‘fmt‘‘ group, but the tuple forms can be used only as ‘‘kwargs‘‘.

Line styles and colors are combined in a single format string, as in ‘‘’bo’‘‘ for blue circles.

The *kwargs* can be used to set line properties (any property that has a ‘‘set_*‘‘ method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an example::

```
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs',  label='line 2')
axis([0, 4, 0, 10])
legend()
```

If you make multiple lines with one plot command, the kwargs apply to all those lines, e.g.::

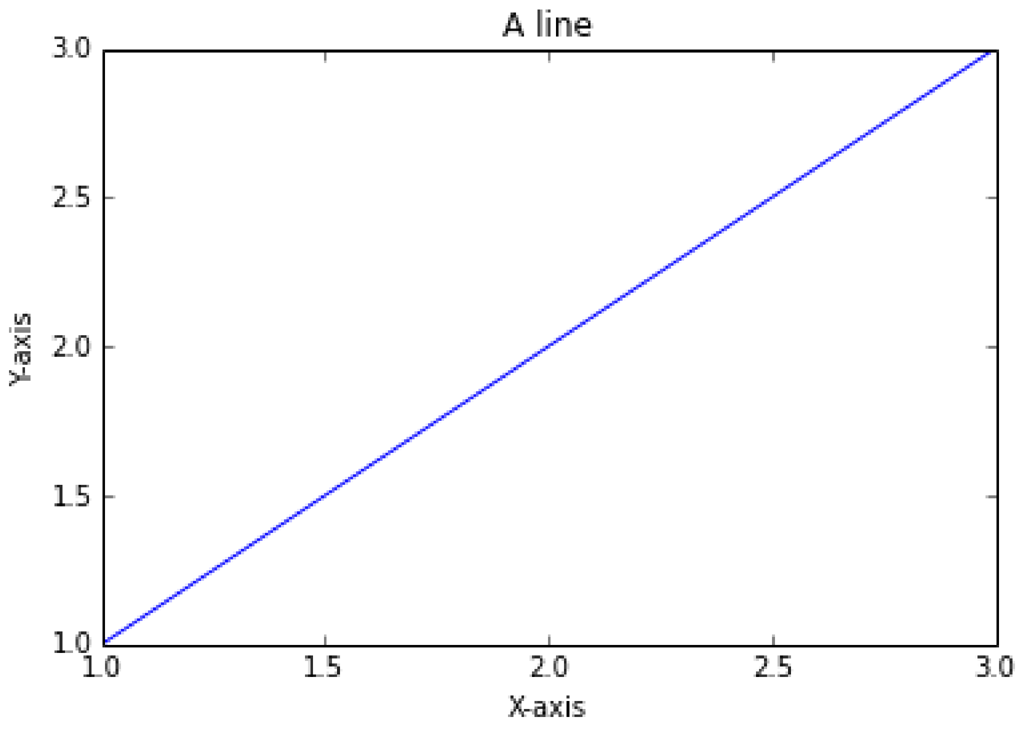
```
plot(x1, y1, x2, y2, antialised=False)
```

```
Neither line will be antialiased.

You do not need to use format strings, which are just
abbreviations.  All of the line properties can be controlled
by keyword arguments.  For example, you can set the color,
marker, linestyle, and markercolor with::

    plot(x, y, color='green', linestyle='dashed', marker='o',
         markerfacecolor='blue', markersize=12).

See :class:`~matplotlib.lines.Line2D` for details.

The kwargs are :class:`~matplotlib.lines.Line2D` properties:

  agg_filter: unknown
  alpha: float (0.0 transparent through 1.0 opaque)
  animated: [True | False]
  antialiased or aa: [True | False]
  axes: an :class:`~matplotlib.axes.Axes` instance
  clip_box: a :class:`matplotlib.transforms.Bbox` instance
  clip_on: [True | False]
  clip_path: [ (:class:`~matplotlib.path.Path`,         :class:`~matplotlib.transforms.Transform`) |
  color or c: any matplotlib color
  contains: a callable function
  dash_capstyle: ['butt' | 'round' | 'projecting']
  dash_joinstyle: ['miter' | 'round' | 'bevel']
  dashes: sequence of on/off ink in points
  drawstyle: ['default' | 'steps' | 'steps-pre' | 'steps-mid' |                  'steps-post']
  figure: a :class:`matplotlib.figure.Figure` instance
  fillstyle: ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
  gid: an id string
  label: string or anything printable with '%s' conversion.
  linestyle or ls: [``'-'`` | ``'--'`` | ``'-.'`` | ``':'`` | ``'None'`` |                  `` '`
  linewidth or lw: float value in points
  lod: [True | False]
  marker: unknown
  markeredgecolor or mec: any matplotlib color
  markeredgewidth or mew: float value in points
  markerfacecolor or mfc: any matplotlib color
  markerfacecoloralt or mfcalt: any matplotlib color
  markersize or ms: float
  markevery: None | integer | (startind, stride)
  path_effects: unknown
  picker: float distance in points or callable pick function         ``fn(artist, event)``
  pickradius: float distance in points
  rasterized: [True | False | None]
  sketch_params: unknown
  snap: unknown
  solid_capstyle: ['butt' | 'round' |  'projecting']
  solid_joinstyle: ['miter' | 'round' | 'bevel']
  transform: a :class:`matplotlib.transforms.Transform` instance
  url: a url string
  visible: [True | False]
  xdata: 1D array
```

```
      ydata: 1D array
      zorder: any number

    kwargs *scalex* and *scaley*, if defined, are passed on to
    :meth:`~matplotlib.axes.Axes.autoscale_view` to determine
    whether the *x* and *y* axes are autoscaled; the default is
    *True*.

    Additional kwargs: hold = [True|False] overrides default hold state
```

- Other useful functions (see below)
- Try it out!

```
In [6]: plt.plot([1,2,3],[1,2,3])
        plt.title("A line")
        plt.xlabel("X-axis")
        plt.ylabel("Y-axis");
```

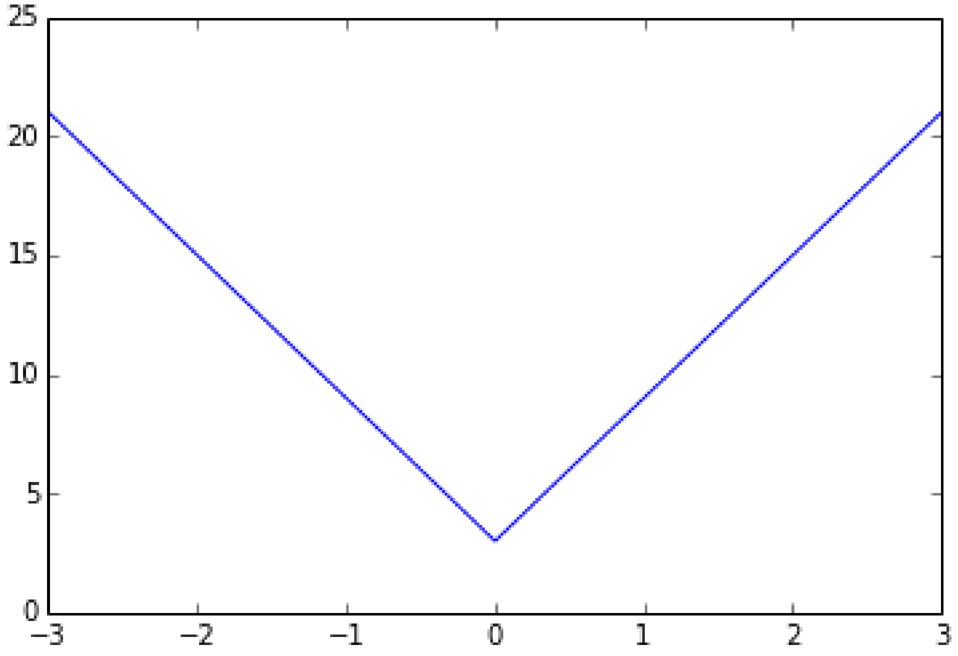


0.1.8 Motivating Example: Plotting a function

- Let's say you want to plot the function (in the math sense) $f(x) = 2x^2 + 3$ for x in the range $[-3, 3]$. How would you do it?

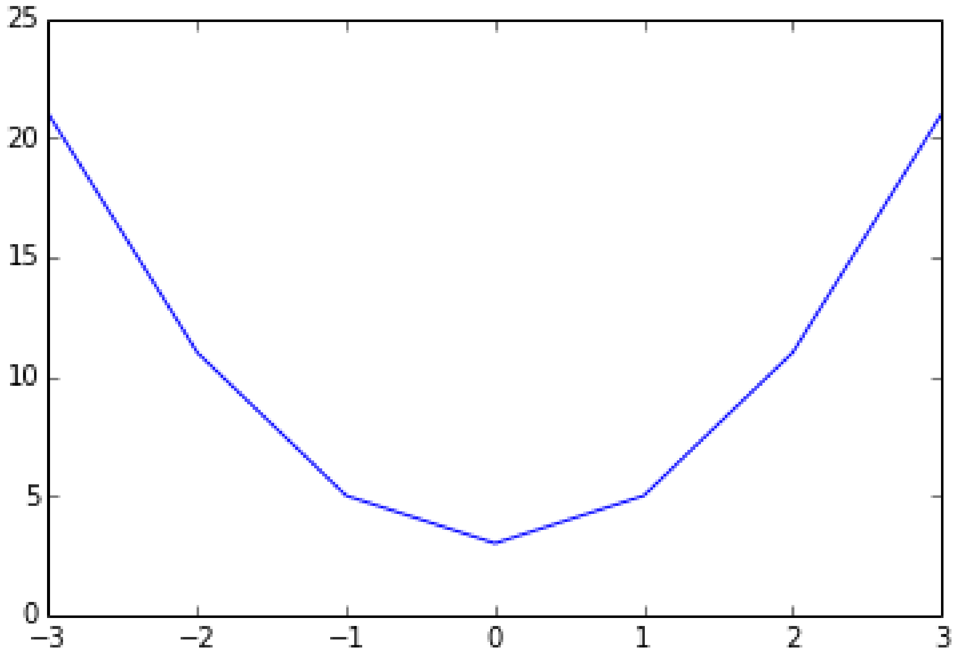
```
In [7]: # Define a Python function that computes the function value
        def f(x):
            return 2*x**2 + 3

        print f(0)
        print f(-3)
        print f(3)

        plt.plot([-3, 0, 3], [f(-3), f(0), f(3)]);
```

```
3
21
21
```



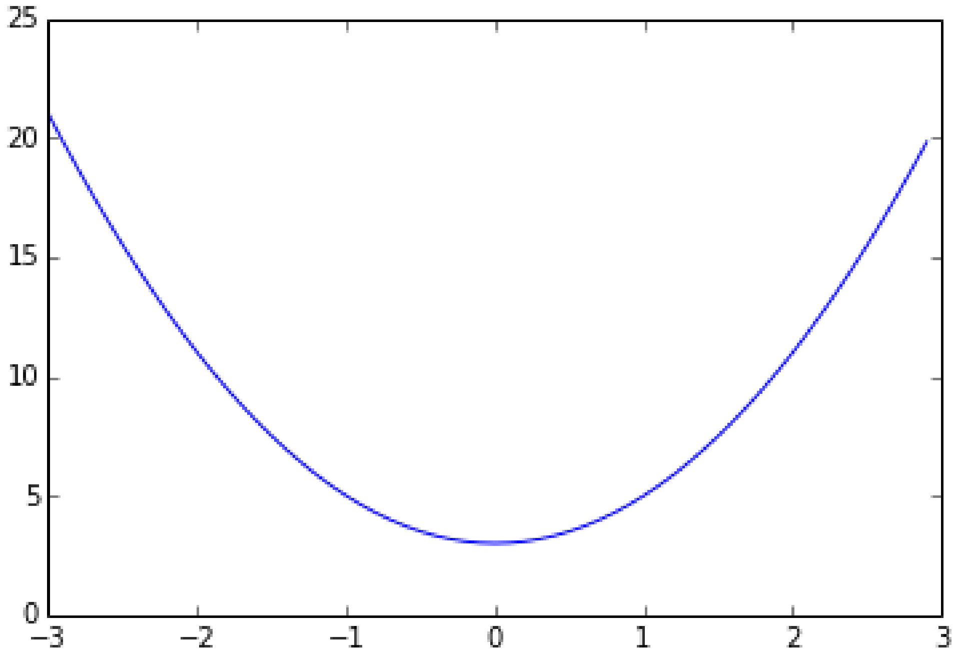
- We can use *range()* and list comprehensions

```
In [8]: xs = range(-3, 4)
        plt.plot(xs, [f(x) for x in xs]);
```

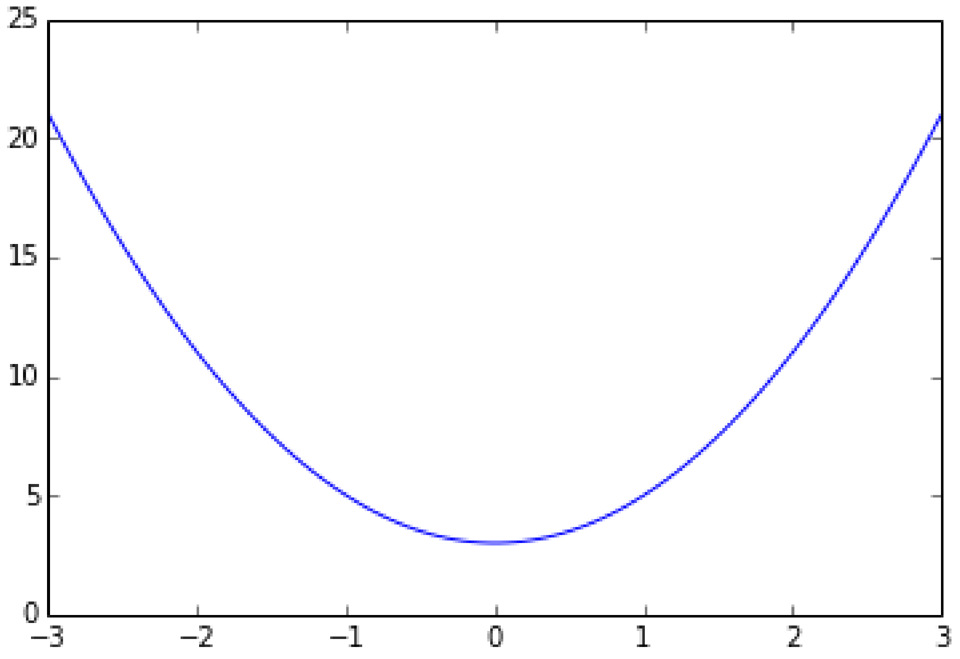


- Need smaller intervals ⇒ use *linspace* or *arange*

```
In [9]: xs1=np.arange(-3,3,0.1)
        plt.plot(xs1, [f(x) for x in xs1]);
```

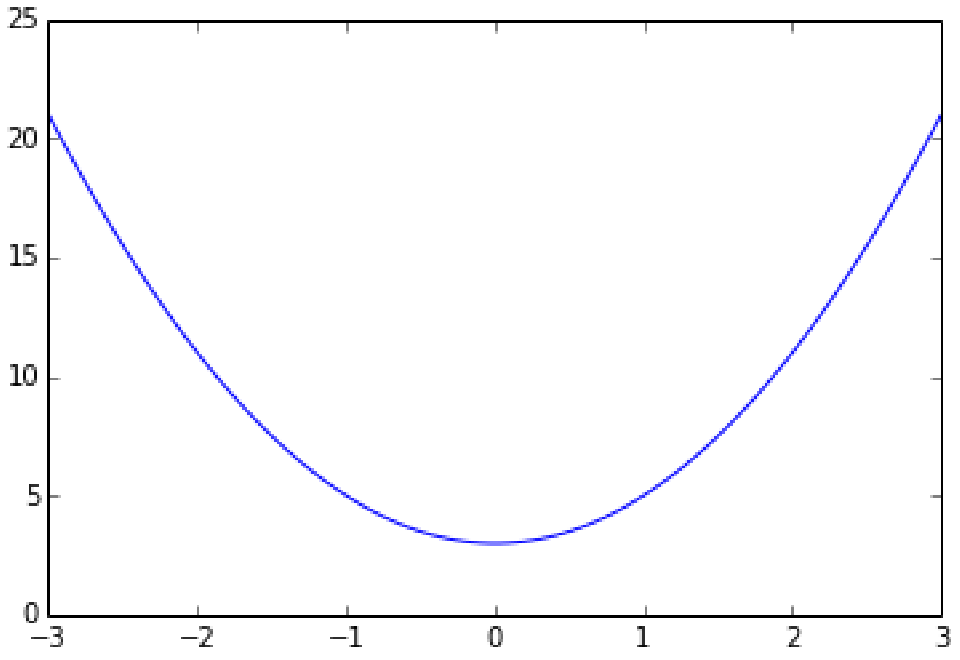


```
In [10]: xs2=np.linspace(-3,3,100)
         plt.plot(xs2, [f(x) for x in xs2]);
```



- Wouldn't it be nice if we could just write

```
In [11]: plt.plot(xs2, 2*xs2**2 + 3);
```



- Oh, wait, we can!
- What is going on? The magic here is that `arange` and `linspace` does not return a list like `range`, but a `numpy.ndarray`.
- If we try it with a list, it doesn't work

```
In [13]: xs = range(-3,4)
         plt.plot(xs, 2*xs**2 + 3)
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-13-c9b93f34ec9e> in <module>()
      1 xs = range(-3,4)
----> 2 plt.plot(xs, 2*xs**2 + 3)


    TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

0.1.9 Running Example: Linear Regression

- Given pairs $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})\}$, where $x^{(i)} \in \mathbb{R}^D, y^{(i)} \in \mathbb{R}$
- Goal find w such that $y^{(i)} \approx f(x^{(i)}) := w^T x^{(i)}$
- Measure of (mean squared) error

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - f(x^{(i)}))^2$$

- Rewrite as matrices/vectors

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{pmatrix}, \quad X = \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(N)} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & \cdots & x_D^{(1)} \\ x_1^{(2)} & \cdots & x_D^{(2)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_D^{(N)} \end{pmatrix}$$

- *Solution:* $w = (X^T X)^{-1} X^T y$

Simpler scalar case:

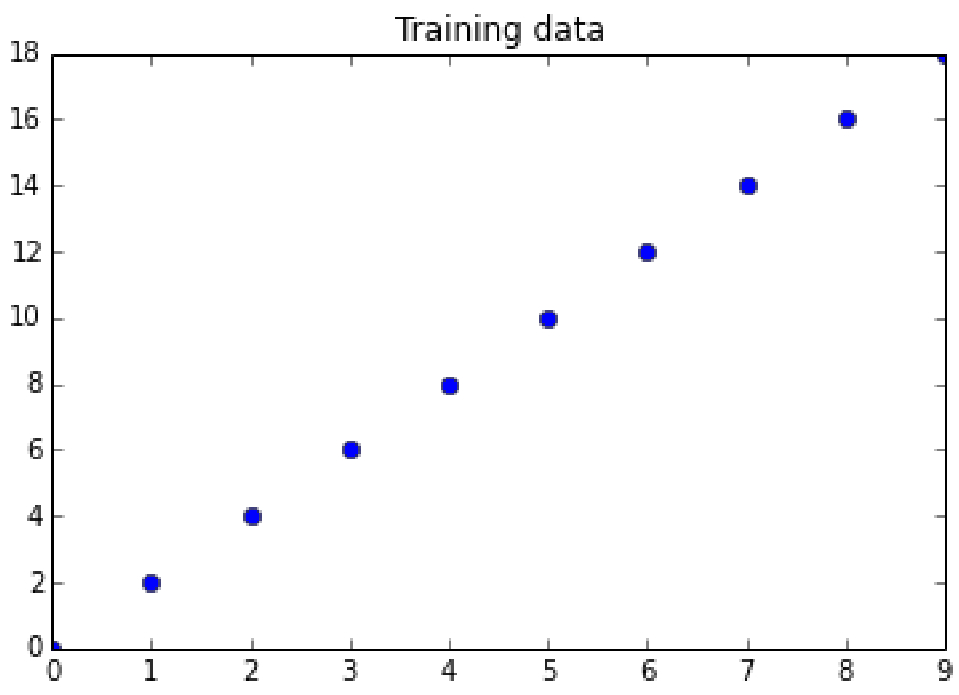
If each of the $x^{(i)}$ is just a scalar, i.e. it is in $\mathbb{R}$, the solutions simplifies to

$$\Rightarrow w = \frac{\sum_i x^{(i)} y^{(i)}}{\sum_i x^2}$$

Turn the math into python

- Using what we have learned so far

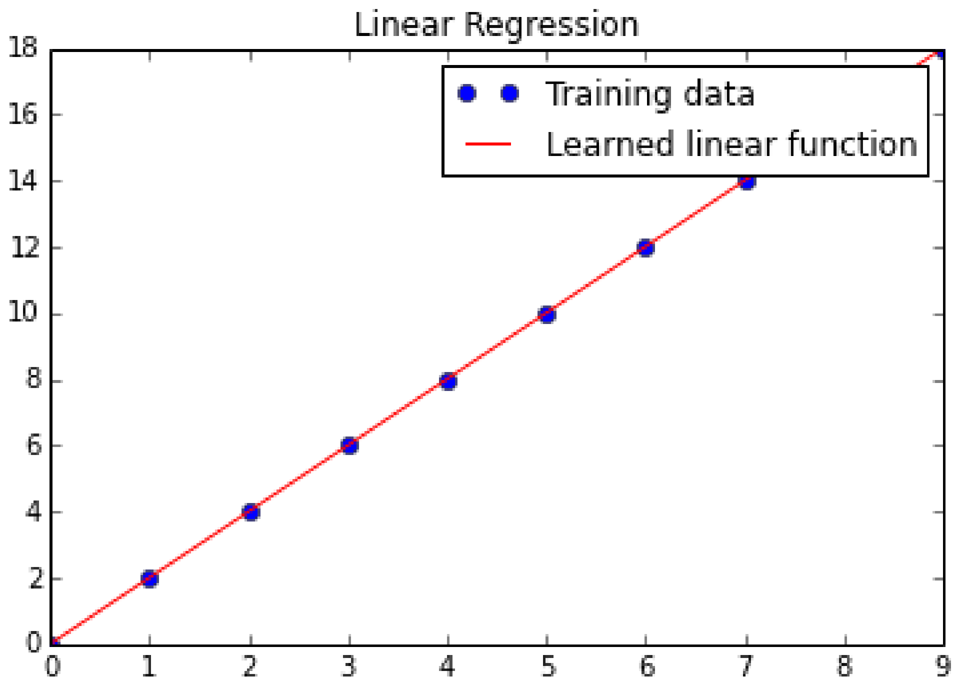
```
In [14]: def f(x):
             return 2 * x

         # Training data (x,y) pairs)
         X = np.arange(10)
         y = [f(xi) for xi in X]

         # plot training data
         plt.plot(X,y, 'o')
         plt.title("Training data")
         plt.show() # causes the plot to be shown NOW
         # linear regression in 1D
         w = ( sum([X[i]*y[i] for i in range(len(X))])/ float(sum([x**2 for x in X])) )
         print "Learned w:", w
```



```
Learned w: 2.0
```

- How does the learned function look like?

```
In [15]: plt.plot(X,y, 'o')
         plt.plot(X, w*X, 'r-')
         plt.title("Linear Regression")
         plt.legend(["Training data", "Learned linear function"]);
```



0.1.10 The *ndarray* type

- The *numpy.ndarray* type represents a multidimensional, homogeneous array of fixed-size items
- You can think of this as a table, where each cell is of the same type and is indexed by a tuple of integer indices
- Vectors (1-D) and matrices (2-D) are the most common examples, but higher-dimensional arrays are also often useful
- For example, you can represent a video as a 4-D array (x, y, frame, channel)

- A 1-D ndarray is similar to a list of numbers
- However it makes it very easy to apply the same (mathematical) operation to all elements in the array
- They also make it very easy to compute aggregates, like sums, means, products, sum-products, etc.

```
In [16]: r = np.arange(1,4,1)
         print "r:", r
         print 3*r + 2
         print np.sqrt(r)

         print np.sum(r)
         print np.mean(r)
         print np.prod(r)
```

```
r: [1 2 3]
[ 5  8 11]
[ 1.          1.41421356  1.73205081]
6
2.0
6
```

- *ndarray* objects have two important attributes:
- The *data type* (*dtype*) of the objects in the cells
- The *shape* of the array, describing the dimensionality and the size of the individual dimensions

```
In [17]: a = np.zeros(shape = (3, 5), dtype = np.int64)
         print "a", a
         print type(a)
         print a.dtype
         print a.shape
```

```
a [[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
<type 'numpy.ndarray'>
int64
(3L, 5L)
```

Accessing Elements

- Access to the individual elements is done using the familiar [] syntax

```
In [18]: a = np.zeros(shape = (3, 5), dtype = np.int64)
         print a[0,0]
         print a
```

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

```
In [19]: a[0,0] = 1
         a[2,3] = 5
         print a
```

```
[[1 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 5 0]]
```

Creating arrays

- Arrays can be created in various ways:
- from (nested) list objects (using *array*) by creating an empty array and then assigning values (using *empty*/*zeros*)
- using built-in functions for special arrays (*ones*, *eye*, *rand*, *randn*, . . .)

```
In [20]: a=np.array([3,2,1,2])
         print a
```

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

```
In [21]: b=np.array([2.0,4.1,3.14])
         print b
```

```
[ 2.    4.1   3.14]
```

```
In [22]: c=np.array([[3,1,2], [2,3,4]])
         print c
```

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

- Note that numpy tries to guess the data type:

```
In [23]: print a.dtype
         print b.dtype
```

```
int32
float64
```

- We can force a particular data type by providing a dtype

```
In [24]: a = np.array([3,2,1,2], dtype=np.float64)
         print a.dtype
         print a
```

```
float64
[ 3.  2.  1.  2.]
```

```
In [25]: b = np.array([3,2,1,2], dtype=np.uint8)
         print b.dtype
         print b
```

```
uint8
[3 2 1 2]
```

- Careful about over/underflows!

```
In [26]: b[0] = 256
         print b
```

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

0.1.11 Data types

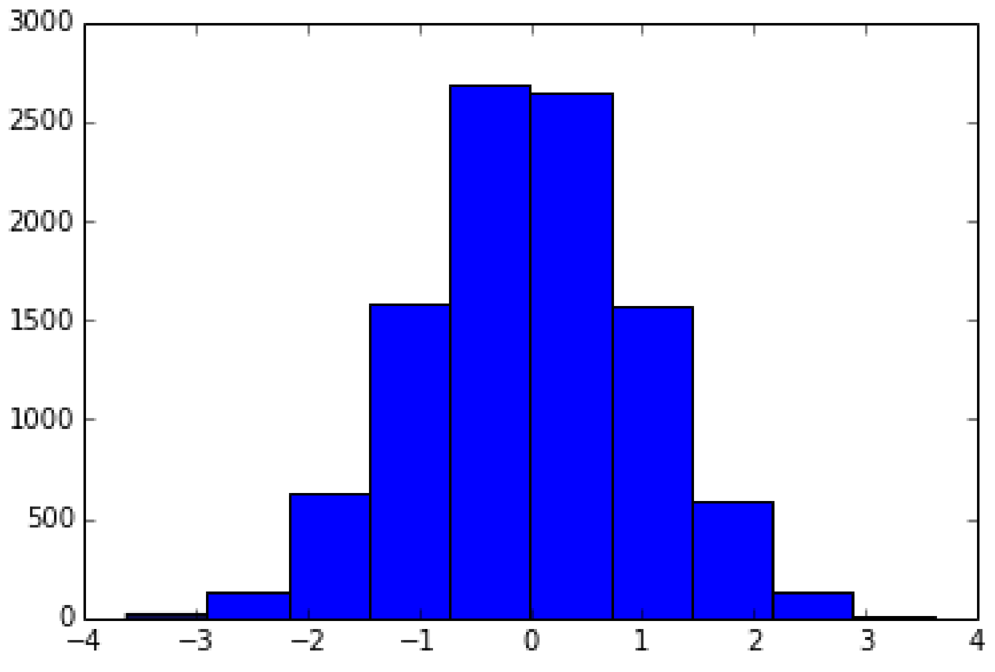
- Numpy provides a range of data types
- floating point data types: `float32`, `float64`
- integer data types: `int64`, `int32`, . . . , `uint8`
- object data type: object – any Python object
- Unless you are sure you need something else, use `float64`. This is the default data type in numpy.
- Exceptions to this rule:
- use `int32`, `int64` when you need to store (large) integers (e.g. counts)
- use objects when you need to store other Python objects (dicts, lists, etc.)

Creating special arrays

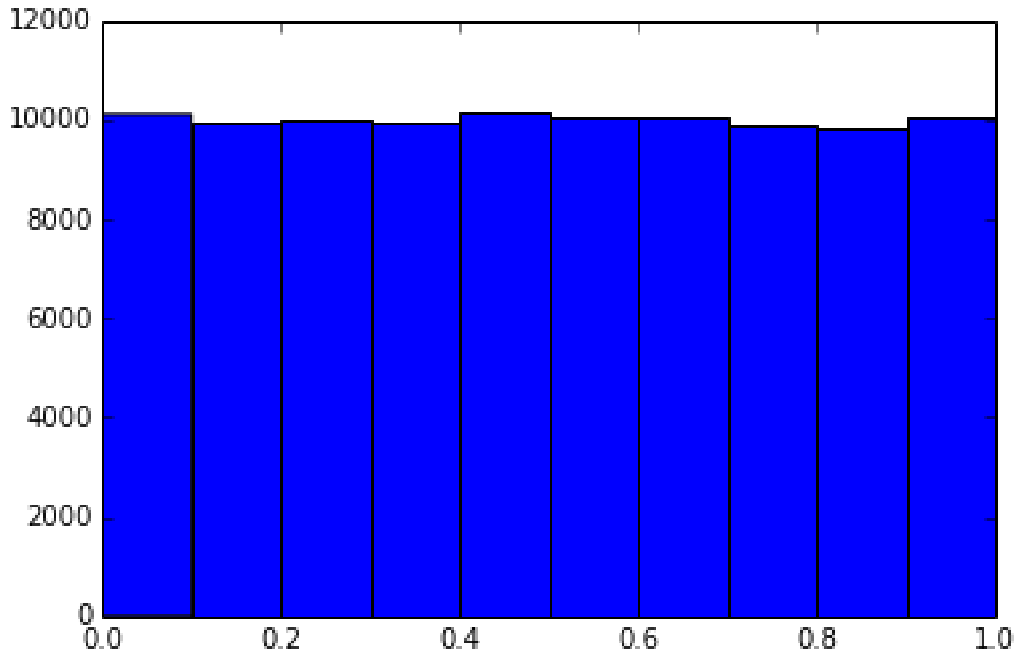
- Numpy provides various functions for creating useful special arrays. The most common ones are:
- `zeros` – create an array filled with zeros
- `ones` – create an array filled with ones
- `empty` – create an array, but don't initialize it's elements.
- `arange` – similar to range
- `eye` – identity matrix of a given size
- `random.rand`, `random.randn` – random arrays (uniform, normal)

```
In [27]: np.zeros((2,3))
```

```
Out[27]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])

In [28]: np.ones((2,3))

Out[28]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])

In [29]: np.empty((2,3))

Out[29]: array([[  2.56808920e-316,   2.56809196e-316,   2.56809473e-316],
                [  2.56809750e-316,   2.56810026e-316,   2.56810303e-316]])

In [30]: np.arange(5,10)

Out[30]: array([5, 6, 7, 8, 9])

In [31]: np.eye(2)

Out[31]: array([[ 1.,  0.],
                [ 0.,  1.]])

In [32]: np.random.rand(2,3)

Out[32]: array([[ 0.31436232,  0.90153585,  0.37472228],
                [ 0.07875608,  0.34302848,  0.62666036]])

In [33]: np.random.randn(2,3)

Out[33]: array([[ 0.23023132, -0.60236058,  0.70351169],
                [ 1.76881097,  1.54750863, -0.67678484]])

In [34]: plt.hist(np.random.randn(10000));
```



```
In [35]: plt.hist(np.random.rand(100000));
```



Dimensionality

- Numpy arrays support the slice notation familiar from strings and lists
- Can be used to index contiguous sub-arrays

```
In [36]: a = np.eye(3)
         print a

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

In [37]: a[0,:]

Out[37]: array([ 1.,  0.,  0.])

In [38]: a[0:2,0:2]

Out[38]: array([[ 1.,  0.],
                [ 0.,  1.]])

In [39]: a[1:2,0:2]

Out[39]: array([[ 0.,  1.]])

In [40]: a[0:1,:]

Out[40]: array([[ 1.,  0.,  0.]])
```

- We can also assign new values to a subarray

```
In [41]: print a

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

In [42]: a[0:2,0:2] = 5
         print a

[[ 5.  5.  0.]
 [ 5.  5.  0.]
 [ 0.  0.  1.]]

In [43]: a[1:3,1:3] = np.ones((2, 2))
         print a

[[ 5.  5.  0.]
 [ 5.  1.  1.]
 [ 0.  1.  1.]]
```

Accessing Subarrays – Boolean indexing

- Comparison operatorions on arrays return boolean arrays of the same size

```
In [44]: a=np.eye(2)

In [45]: a == 1

Out[45]: array([[ True, False],
                [False,  True]], dtype=bool)

In [46]: a > 0

Out[46]: array([[ True, False],
                [False,  True]], dtype=bool)

In [47]: a<=0

Out[47]: array([[False,  True],
                [ True, False]], dtype=bool)
```

- These boolean arrays can then be used for indexing the original array

```
In [48]: a=np.eye(2)
         print a

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

In [49]: a[a == 1]

Out[49]: array([ 1.,  1.])

In [50]: a[a == 1] = 2
         print a

[[ 2.  0.]
 [ 0.  2.]]
```

Elementwise operations

- Numpy contains many standard mathematical functions that operate elementwise on arrays
- These are called universal functions (ufuncs)
- Included are things like *add*, *multiply*, *log*, *exp*, *sin*, . . .
- See http://docs.scipy.org/doc/numpy/reference/ufuncs.html
- These are very fast (in constrast to doing the same by loops)

```
In [51]: a = np.ones((2,2))*2
         print a

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

In [52]: np.add(a,a)

Out[52]: array([[ 4.,  4.],
                [ 4.,  4.]])

In [53]: a * a

Out[53]: array([[ 4.,  4.],
                [ 4.,  4.]])

In [54]: np.log(a)

Out[54]: array([[ 0.69314718,  0.69314718],
                [ 0.69314718,  0.69314718]])
```

Don't do this, it is *very* slow (for large arrays):

```
In [55]: result=np.empty(a.shape)
         for i in range(a.shape[0]):
             for j in range(a.shape[1]):
                 result[i,j]=a[i,j]*2

         print result

[[ 4.  4.]
 [ 4.  4.]]
```

Broadcasting

- Broadcasting is how numpy handles operations between arrays of different, but compatible shapes
- For example, you might want to add a column vector to all columns of a matrix

```
In [56]: a = np.ones((2,2))
         b = 2*np.ones((2,1))
         print "a:"
         print a
         print "b"
         print b
         print "a+b"
         print a + b
```

```
a:
[[ 1.  1.]
 [ 1.  1.]]
b
[[ 2.]
 [ 2.]]
a+b
[[ 3.  3.]
 [ 3.  3.]]
```

```
In [57]: a = np.ones((3,2))
         b = np.array([4,5])
         print "a:", a.shape
         print a
         print "b:", b.shape
         print b
```

```
a: (3L, 2L)
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
b: (2L,)
[4 5]
```

```
In [58]: print a+b
```

```
[[ 5.  6.]
 [ 5.  6.]
 [ 5.  6.]]
```

```
In [59]: print a* b
```

```
[[ 4.  5.]
 [ 4.  5.]
 [ 4.  5.]]
```

Reductions

- One of the most common operation one needs to do to an array is to sum its values along one of it's dimensions.
- Again, this is much faster than looping over the array by hand.

```
In [60]: a = np.array([[1,2],[3,4]])
         print a
```

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

```
In [61]: np.sum(a)

Out[61]: 10

In [62]: np.sum(a,0)

Out[62]: array([4, 6])

In [63]: np.sum(a,1)
```

```
Out[63]: array([3, 7])
```

- There are other reductions the behave the same way as *sum*, for example

```
In [64]: print a

[[1 2]
 [3 4]]

In [65]: np.prod(a)

Out[65]: 24

In [66]: np.mean(a)

Out[66]: 2.5

In [67]: np.mean(a,0)

Out[67]: array([ 2.,  3.])

In [68]: np.cumsum(a)

Out[68]: array([ 1,  3,  6, 10])
```

- See http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#calculation

Vector/Matrix operations

- Many algorithms in scientific computing can be cast in a form that makes use of only a few linear algebra primitves.
- Probably the most common such primitives are the dot-product between vectors, matrix-vector products, as well as matrix-matrix products.
- Numpy uses a single functions for all these operations: *dot*, both for arrays and matrices:

```
In [69]: a = np.ones(2)*3
         print "a"
         print a

a
[ 3.  3.]

In [70]: b = np.array([[1,2]]).T
         print b

[[1]
 [2]]

In [71]: A = np.random.rand(2,2)
         print A

[[ 0.20497529  0.06516387]
 [ 0.20523552  0.21273193]]

In [72]: np.dot(a,A)

Out[72]: array([ 1.23063242,  0.83368743])

In [73]: np.dot(A,b)

Out[73]: array([[ 0.33530304],
                [ 0.63069939]])

In [74]: np.dot(A,A)

Out[74]: array([[ 0.05538881,  0.02721942],
                [ 0.08572836,  0.05862882]])
```

0.1.12 Regression revisited

- Numpy makes your life much easier

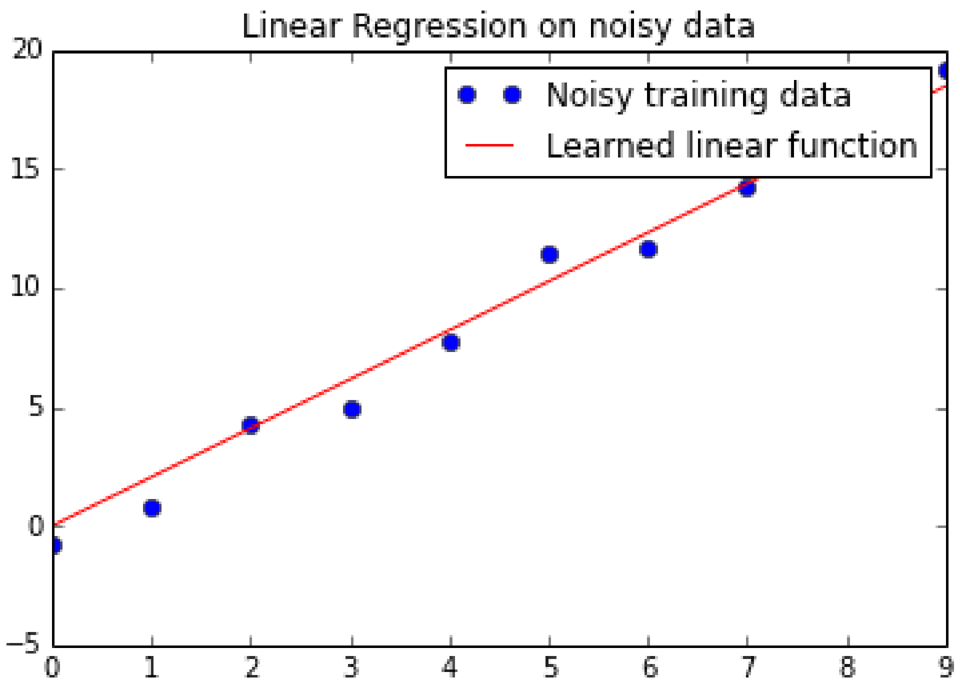
```
In [75]: # Training data (x,y) pairs)
         X = np.array(range(10))
         y = 2 * X

         # linear regression in 1D
         w = X.T.dot(y) / X.T.dot(X)
         print "Learned w:", w
```

```
Learned w: 2
```

Linear Regression on noisy data

- Use `randn` to add (Gaussian) noise $\mathcal{N}(0,1)$ to data
- Real Life data almost always contains noise!
- The normal distribution often is a good approximation to that, cf https://en.wikipedia.org/wiki/Gaussian_noise

```
In [76]: X = np.array(range(10))
         y = 2 * X + np.random.randn(10)
         w = X.T.dot(y) / X.T.dot(X)

         plt.plot(X,y, 'o')
         plt.plot(X, w*X, 'r-')
         plt.title("Linear Regression on noisy data")
         plt.legend(["Noisy training data", "Learned linear function"]);
```



- Compare with the old code
- The numpy one is not only easier to read but also much (!) faster
- Worth diving into numpy documentation and built-in functions

```
In [77]: w1 = X.T.dot(y) / X.T.dot(X)
         w2 = ( sum([X[i]*y[i] for i in range(len(X))])/ float(sum([x**2 for x in X])) )
         print w1,w2
```

```
2.05142200181 2.05142200181
```

0.1.13 Scipy Overview

- SciPy is large collection of (sub-)packages that contains a variety of functions that are useful for scientific computing
- Impossible to cover all, check out the documentation and examples at http://wiki.scipy.org/Cookbook
- In particular, there are functions for
- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)

```
In [77]:
```

```
In [ ]:
```