PWC Python: NumPy & SciPy

Ramses van Zon

SciNet HPC Consortium

11 December 2014

Arrays: Numpy

Lists aren't the ideal data type

Lists can do funny things that you don't expect, if you're not careful.

- Lists are just a collection of items, of any type.
- If you do mathematical operations on a list, you won't get what you expect.
- These are not the ideal data type for scientific computing.
- Arrays are a much better choice, but are not a native Python data type.

```
In [1]: a = [1, 2, 3, 4]
In [2]: a
Out[2]: [1, 2, 3, 4]

In [3]: b = [3, 5, 5, 6]
In [4]: b
Out[4]: [3, 5, 5, 6]

In [5]: 2 * a
Out[5]: [1, 2, 3, 4, 1, 2, 3, 4]

In [6]: a + b
Out[6]: [1, 2, 3, 4, 3, 5, 5, 6]
```

SciNet

Useful arrays

Arrays are what we want to use

Almost everything that you want to do starts with NumPy.

- Contains arrays of various types and forms: zeros, ones, linspace, etc.
- linspace takes 2 or 3 arguments, the default number of entries is 50.

```
In [7]: from numpy import zeros,
        ones, linspace
In [8]: zeros(5)
Out[8]: array([ 0., 0., 0., 0., 0.

In [9]: ones(5, dtype = int)
Out[9]: array([ 1, 1, 1, 1, 1])
```

```
In [10]: zeros([2,2])
Out[10]:
array([[ 0., 0.],
       [ 0., 0.]])

In [11]: arange(5)
Out[11]: array([ 0, 1, 2, 3, 4])

In [12]: linspace(1,5)
Out[12]: array([ 1., 1.08163265,
  1.16326531, 1.24489796,
  .
  .
  4.67346939, 4.75510204,
  4.83673469, 4.91836735, 5.  ])

In [13]: linspace(1, 5, 6)
Out[13]: array([1.,1.8,2.6,3.4,
                4.2,5.])
```

Accessing array elements

Elements of arrays are accessed using square brackets.

- Python is *row major* (like C++, Mathematica), NOT *column major* (like Fortran, MATLAB, R).
- This means the first index is the row, not the column.
- Indexing starts at zero.

```
In [14]: zeros([2, 3])
Out[14]:
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
In [15]: a = zeros([2,3])
In [16]: a[1,2] = 1
In [17]: a[0,1] = 2
In [18]: a
Out[18]:
array([[  0.,   2.,   0.],
       [  0.,   0.,   1.]])
```

```
In [19]: a[2,1] = 1
---------------------------------
IndexError
<ipython-input-21-83f146d6c508> in
----> 1 a[2,1] = 1

IndexError: index (2) out of range

In [20]:
```

SciNet

Copying arrays

Copying array variables

Use caution when copying array variables. There's a 'feature' here that is unexpected.

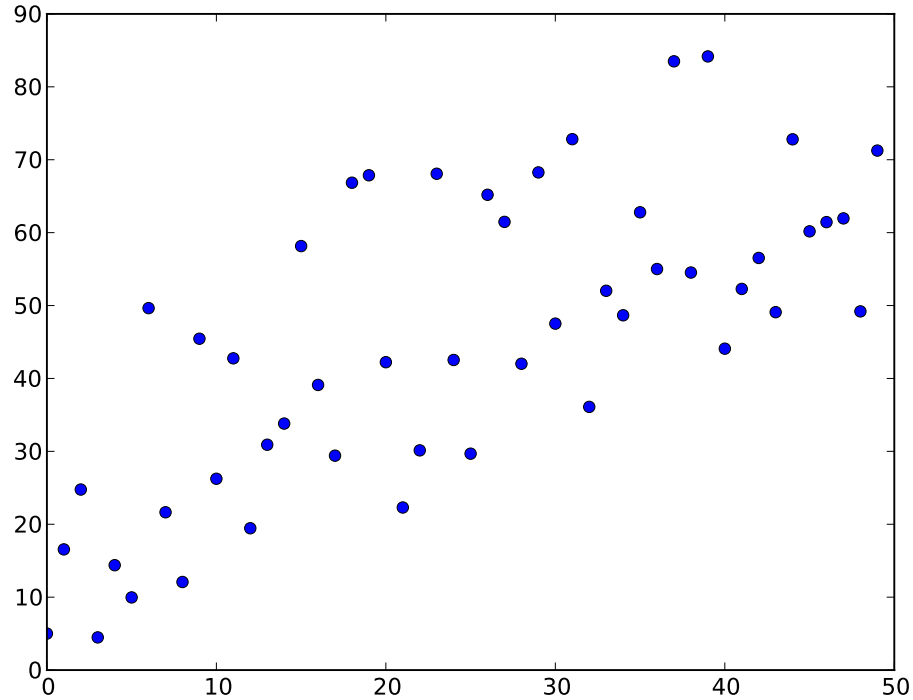
```
In [20]: a = 10; b = a; a = 20
In [21]: a, b
Out[21]: (20, 10)
In [22]: a = array([[1,2,3],
                    [2,3,4]])
```

```
In [23]: b = a
In [24]: a[1,0] = -10
In [25]: a
Out[25]:
array([[1, 2, 3],
       [-10, 3, 4]])
In [26]: b
Out[26]:
array([[1, 2, 3],
       [-10, 3, 4]])
```

```
In [27]: b = a.copy()
In [28]: a[1,0] = 16
In [29]: a
Out[29]:
array([[1, 2, 3],
       [16, 3, 4]])
In [30]: b
Out[30]:
array([[1, 2, 3],
       [-10, 3, 4]])
```

Polynomial fitting

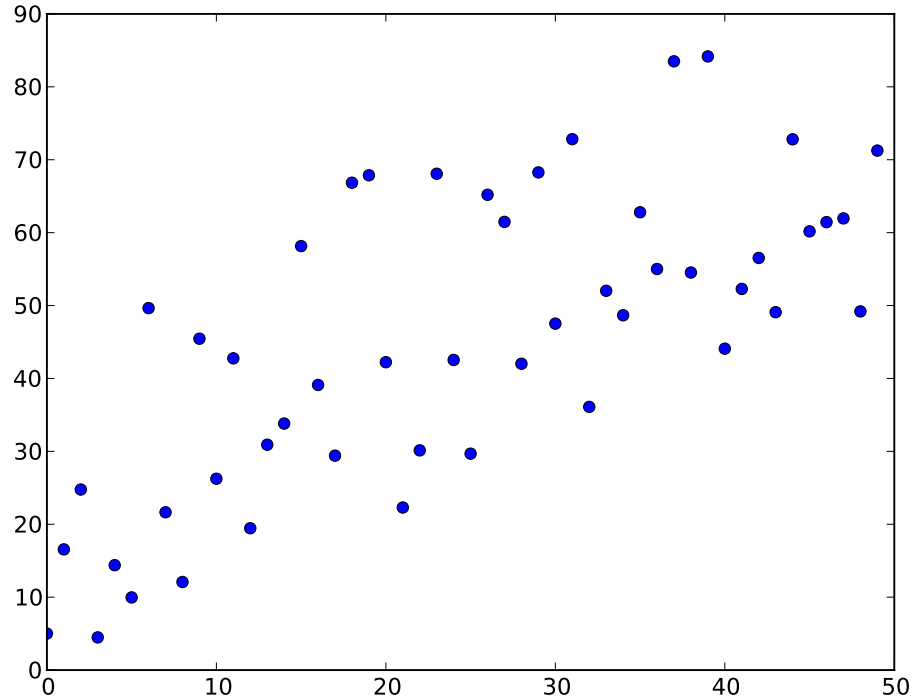
```
In [96]: from numpy import arange,
          zeros, polyfit, polyval
In [97]: import random
In [98]: x = arange(50.)
In [99]: y = zeros(50.)
In [100]: for i in arange(50.):
              y[i] = x[i] + 50.0 *
In [101]: plot(x, y, 'o')
In [102]:
.
.
.
.
```




```
.
.
.
.
.
.
.
```

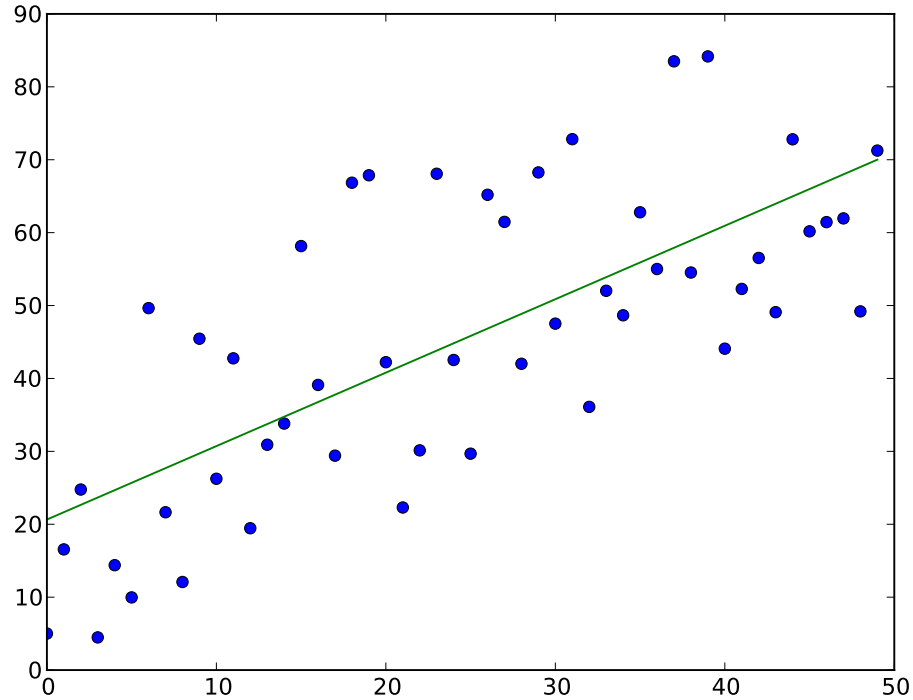
Polynomial fitting

```
In [96]: from numpy import arange,
           zeros, polyfit, polyval
In [97]: import random
In [98]: x = arange(50.)
In [99]: y = zeros(50.)
In [100]: for i in arange(50.):
              y[i] = x[i] + 50.0 *
In [101]: plot(x, y, 'o')
In [102]: fit = polyfit(x, y, 1)
In [103]: fit
Out[103]: array([  1.0073584,
                  20.64695036])

In [104]:
```



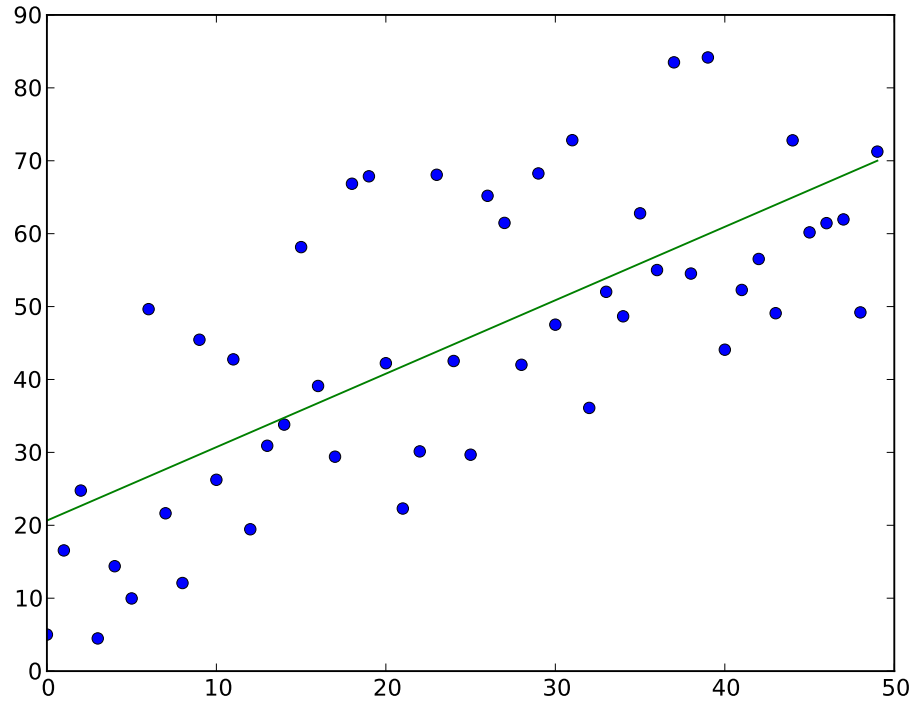
Polynomial fitting

```
In [96]: from numpy import arange,
          zeros, polyfit, polyval
In [97]: import random
In [98]: x = arange(50.)
In [99]: y = zeros(50.)
In [100]: for i in arange(50.):
             y[i] = x[i] + 50.0 *
In [101]: plot(x, y, 'o')
In [102]: fit = polyfit(x, y, 1)
In [103]: fit
Out[103]: array([  1.0073584,
                  20.64695036])

In [104]: plot(x,polyval(fit, x))
```



Polynomial fitting

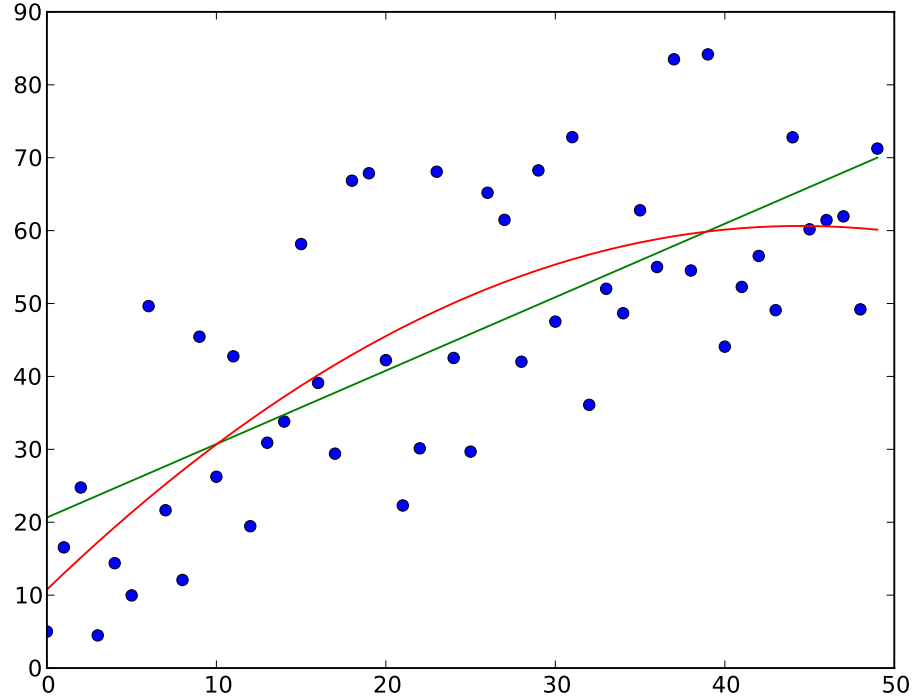
```
In [96]: from numpy import arange,
           zeros, polyfit, polyval
In [97]: import random
In [98]: x = arange(50.)
In [99]: y = zeros(50.)
In [100]: for i in arange(50.):
              y[i] = x[i] + 50.0 *
In [101]: plot(x, y, 'o')
In [102]: fit = polyfit(x, y, 1)
In [103]: fit
Out[103]: array([  1.0073584,
                  20.64695036])

In [104]: plot(x,polyval(fit, x))
```




```
In [105]: fit = polyfit(x, y, 2)
In [106]: fit
Out[106]: array([ -0.02520835,
                   2.24256777,
                  10.76527546])
In [107]:
```

Polynomial fitting

```
In [96]: from numpy import arange,
           zeros, polyfit, polyval
In [97]: import random
In [98]: x = arange(50.)
In [99]: y = zeros(50.)
In [100]: for i in arange(50.):
              y[i] = x[i] + 50.0 *
In [101]: plot(x, y, 'o')
In [102]: fit = polyfit(x, y, 1)
In [103]: fit
Out[103]: array([  1.0073584,
                  20.64695036])

In [104]: plot(x,polyval(fit, x))
```




```
In [105]: fit = polyfit(x, y, 2)
In [106]: fit
Out[106]: array([ -0.02520835,
                   2.24256777,
                  10.76527546])
In [107]: plot(x, polyval(fit, x))
```

Matrix arithmetic

vector-vector & vector-scalar multiplication

1-D arrays are often called ‘vectors’.

- When vectors are multiplied you get element-by-element multiplication.
- When vectors are multiplied by a scalar (a 0-D array), you also get element-by-element multiplication.

```
In [31]: a = arange(4)
In [32]: a
Out[32]: array([0,1,2,3])

In [33]: b = arange(4.) + 3
In [34]: b
Out[34]: array([3.,4.,5.,6.])

In [35]: c = 2
In [36]: c
Out[36]: 2

In [37]: a * b
Out[37]: array([0.,4.,10.,18.])

In [38]: a * c
Out[38]: array([0,2,4,6])

In [39]: b * c
Out[39]: array([6.,8.,10.,12.])
```

Matrix-vector multiplication

A 2-D array is sometimes called a 'matrix'.

- Matrix-scalar multiplication gives element-by-element multiplication.
- Matrix-vector multiplication DOES NOT give the standard result!

```
In [40]: a = array([[1,2,3],
                    [2,3,4]])
In [41]: a
Out[41]:
array([[1, 2, 3],
       [2, 3, 4]])

In [42]: b = arange(3) + 1
In [43]: b
Out[43]: array([1, 2, 3])

In [44]: a * b
Out[44]:
array([[ 1,  4,  9],
       [ 2,  6, 12]])
```

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11} * b_1 + a_{12} * b_2 + a_{13} * b_3 \\ a_{21} * b_1 + a_{22} * b_2 + a_{23} * b_3 \\ a_{31} * b_1 + a_{32} * b_2 + a_{33} * b_3 \end{bmatrix}$$

SciNet

Matrix-matrix multiplication

Not surprisingly, matrix-matrix multiplication doesn't work as expected either, instead doing the same thing as vector-vector multiplication.

```
In [45]: a = array([[1,2,3],
                    [2,3,4]])
In [46]: b = array([[1,2,3],
                    [2,3,4]])
In [47]: a
Out[47]:
array([[1, 2, 3],
       [2, 3, 4]])

In [48]: a * b
Out[48]:
array([[ 1,  4,  9],
       [ 4,  9, 16]])
```

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

SciNet

Scipy: matrix algebra, statistics, and more

How to do matrix algebra?

There are two solutions to these matrix multiplication problems.

- The specially built-in array fixes (using 'array' types).
- The matrix module (using 'matrix' types).

The latter option is a bit clunkier, so we recommend the 'fixes'.

```
In [49]: from scipy import dot
In [50]: a = array([[1,2,3],
                    [2,3,4]])
In [51]: b = array([[1,2,3],
                    [2,3,4]])
In [52]: a
Out[52]:
array([[1, 2, 3],
       [2, 3, 4]])
```

```
In [53]: a.transpose()
Out[53]:
  array([[1, 2],
         [2, 3],
         [3, 4]])

In [54]: dot(a.transpose(), b)
Out[54]:
  array([[ 5,  8, 11],
         [ 8, 13, 18],
         [11, 18, 25]])

In [55]: dot(b, a.transpose())
Out[55]:
array([[14, 20],
       [20, 29]])

In [56]: c = arange(3) + 1
In [57]: dot(a,c)
Out[57]: array([14, 20])
```

linalg module

The linalg module

The linalg module contains useful functions for matrix algebra.

- Typical matrix functions: inv, det, norm...
- More advanced functions: eig, SVD, cholesky...
- Both NumPy and SciPy have a linalg module. Use SciPy, because it is always compiled with BLAS/LAPACK support.

```
In [58]: from scipy import dot, linalg
In [59]: a = array([[1,2,3], [3,4,5], [1,1,2]])
In [60]: linalg.det(a)
Out[60]: -2.0

In [61]: dot(a, linalg.inv(a))
Out[61]:
  array([[ 1.00000000e+00,   0.00000000e+00,   0.00000000e+00],
         [ 2.77555756e-16,   1.00000000e+00,   0.00000000e+00],
         [ 0.00000000e+00,   5.55111512e-17,   1.00000000e+00]])
```

Solving systems of equations

The linalg module comes with an important function: solve. linalg.solve is used to solve the system of equations

$$Ax = b$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} -0.5 \\ -0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

```
In [62]: a = array([[1,2,3],
                    [3,4,5],
                    [1,1,2]])

In [63]: a
Out[63]:
  array([[ 1,  2, 3],
         [ 3, 4, 5],
         [ 1, 1, 2]])

In [64]: b = array([3, 4, 2])
In [65]: b
Out[65]: array([3, 4, 2])
In [66]: x = linalg.solve(a, b)
In [67]: x
Out[67]: array([-0.5, -0.5, 1.5])
```

SciNet

Statistics

Statistics

SciPy contains all of the statistical functions that you'll probably ever need.

- The scipy.stats module is based around the idea of a 'random variable' type.
- A whole variety of standard distributions are available:
  - Continuous distributions: Normal, Maxwell, Cauchy, Chi-squared, Gumbel Left-scewed, Gilbrat, Nakagami, ...
  - Discrete distributions: Poisson, Binomial, Geometric, Bernoulli, ...
- The 'random variables' have all of the statistical properties of the distributions built into them already: cdf, pdf, mean, variance, moments, ...

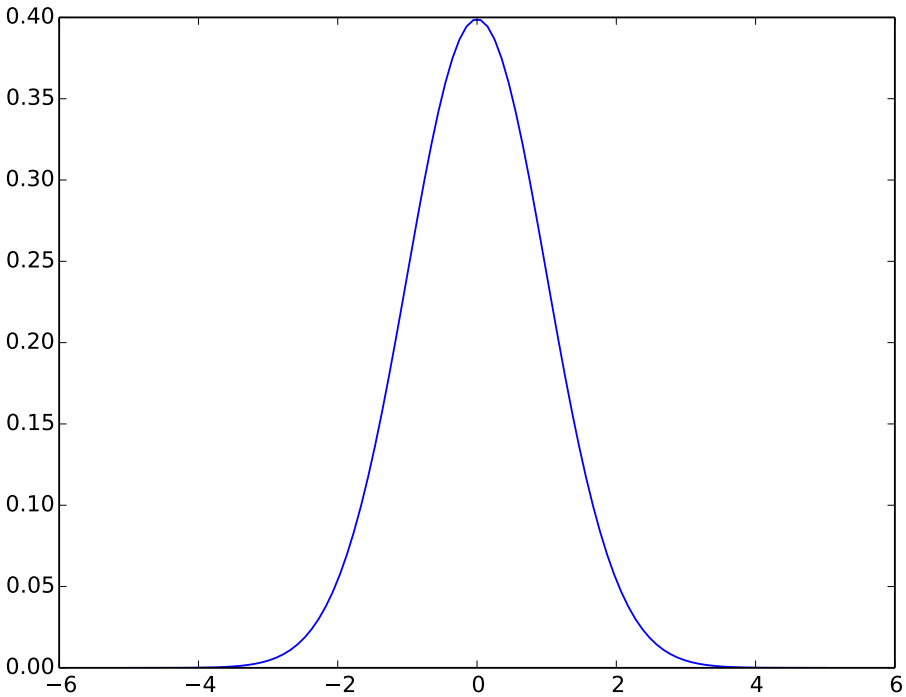
SciNet

Normal distribution

All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
In [68]: from scipy.stats import norm
In [69]: x = linspace(-5, 5, 100)
In [70]:


.
```

Normal distribution

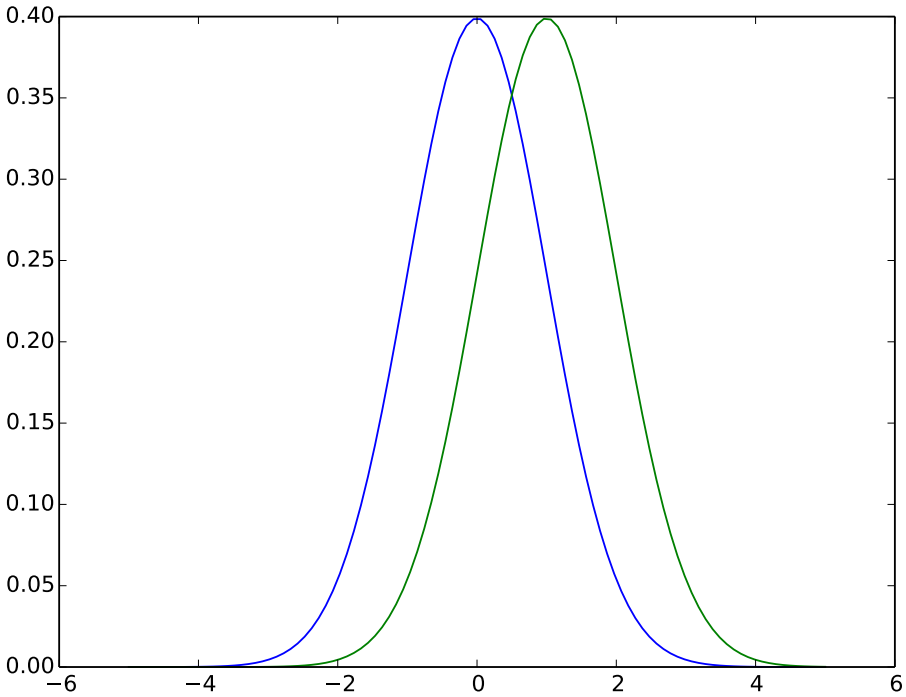


All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
In [68]: from scipy.stats import norm
In [69]: x = linspace(-5, 5, 100)
In [70]: plot(x, norm.pdf(x))
In [71]:
.
```

SciNet

Normal distribution

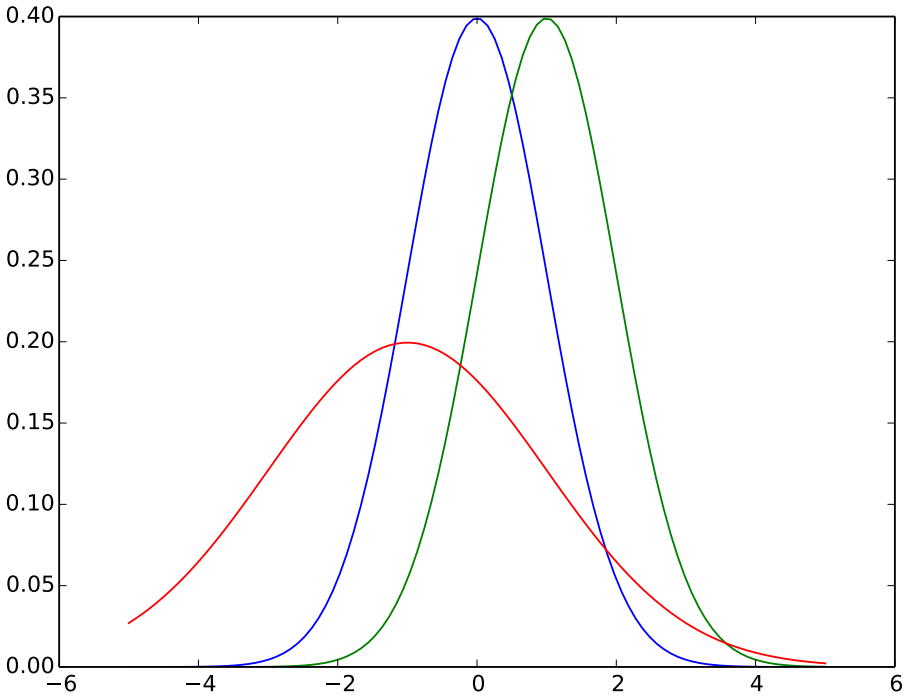


All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
In [68]: from scipy.stats import norm
In [69]: x = linspace(-5, 5, 100)
In [70]: plot(x, norm.pdf(x))
In [71]: plot(x, norm.pdf(x, loc = 1))
In [72]:
```

SciNet

Normal distribution



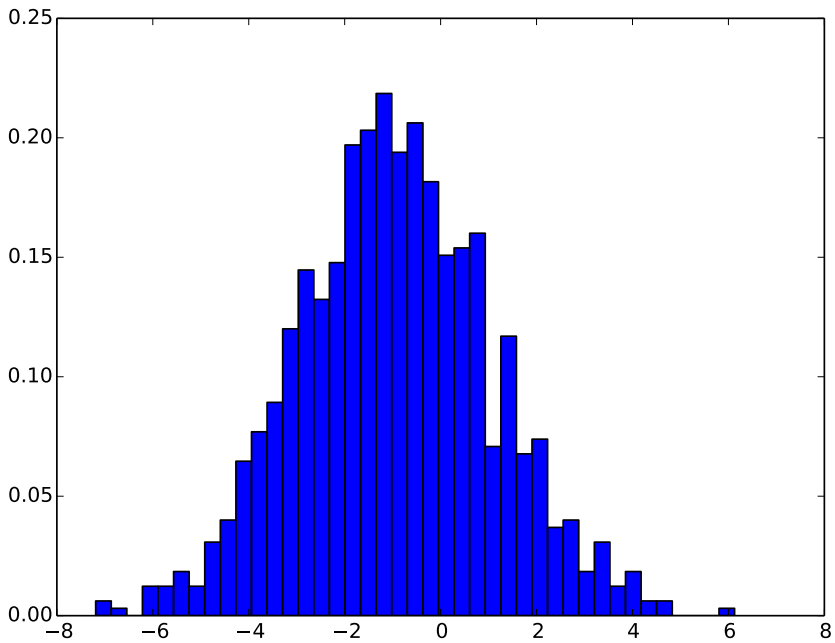
All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
In [68]: from scipy.stats import norm
In [69]: x = linspace(-5, 5, 100)
In [70]: plot(x, norm.pdf(x))
In [71]: plot(x, norm.pdf(x, loc = 1))
In [72]: plot(x, norm.pdf(x, loc = -1, scale = 2))
```

SciNet

Normal Distribution, continued

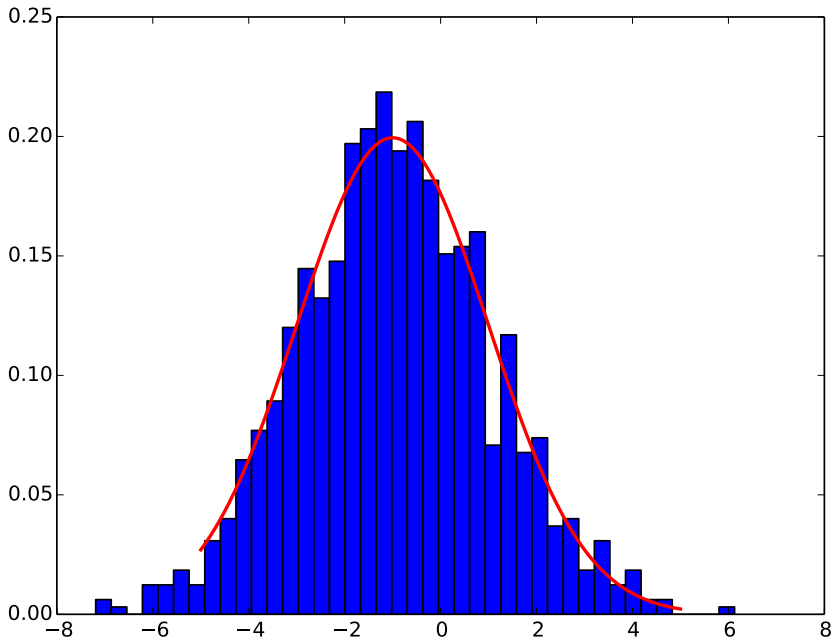
```
In [73]: from pylab import hist
In [74]: norm.mean(loc = -1, scale = 2)
Out[74]: -1.0
In [75]: norm.std(loc = -1, scale = 2)
Out[75]: 2.0
In [76]: norm.moment(3, loc = -1, scale = 2)
Out[76]: -13.0
In [77]: samples = norm.rvs(size = 1000, loc = -1, scale = 2)
In [78]: h = hist(samples, bins = 41, normed = True)
In [79]:
```



SciNet

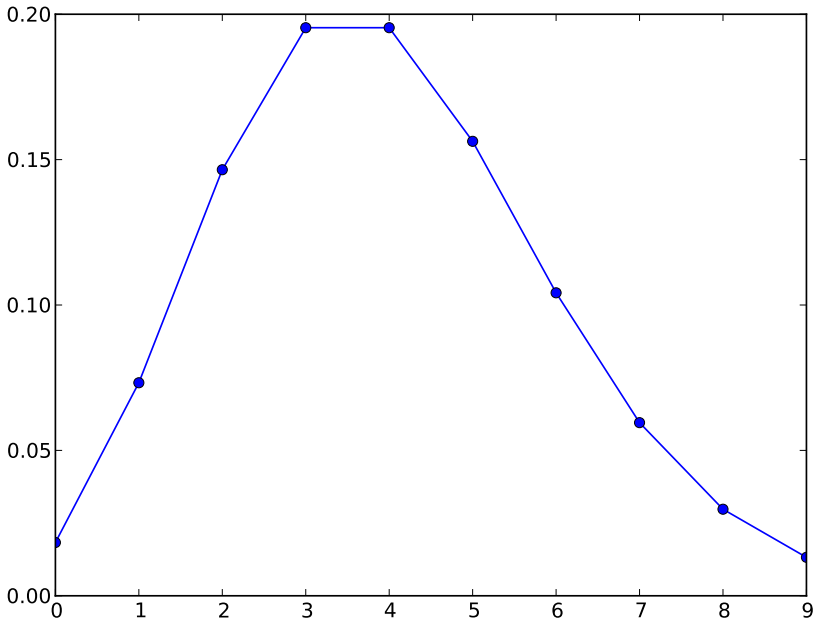
Normal Distribution, continued

```
In [73]: from pylab import hist
In [74]: norm.mean(loc = -1, scale = 2)
Out[74]: -1.0
In [75]: norm.std(loc = -1, scale = 2)
Out[75]: 2.0
In [76]: norm.moment(3, loc = -1, scale = 2)
Out[76]: -13.0
In [77]: samples = norm.rvs(size = 1000, loc = -1, scale = 2)
In [78]: h = hist(samples, bins = 41, normed = True)
In [79]: plot(x,norm.pdf(x,loc = -1,scale = 2),'r',linewidth=2)
```



Poisson distribution

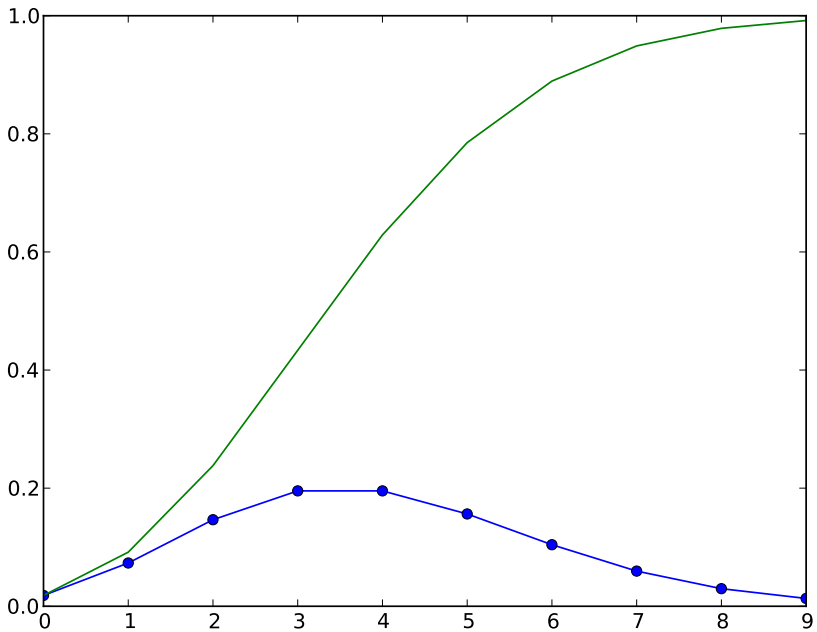
```
In [80]: from scipy.stats import poisson
In [81]: x = arange(10)
In [82]: poisson.mean(4)
Out[82]: 4.0
In [83]: poisson.var(4)
Out[83]: 4.0
In [84]: plot(x, poisson.pmf(x, 4), 'o-')
In [85]:
```



SciNet

Poisson distribution

```
In [80]: from scipy.stats import poisson
In [81]: x = arange(10)
In [82]: poisson.mean(4)
Out[82]: 4.0
In [83]: poisson.var(4)
Out[83]: 4.0
In [84]: plot(x, poisson.pmf(x, 4), 'o-')
In [85]: plot(x, poisson.cdf(x, 4)
```



Note that discrete distributions have Probability Mass Functions (PMF) instead of Probability Distribution Functions (PDF).

SciNet

What else is in SciPy?

SciPy is large

- Linear Algebra
- Statistics
- Special functions
- Numeric integration and ODEs
- Optimization
- Interpolation
- Fourier Transforrs
- File IO
- and more..

SciNet

Summary

Summary

- When manipulating numbers in blocks, use the array type, not lists.
- Use the copy() function to copy array variables.
- Array types do element-by-element multiplication, addition, etc.
- If you want to do standard array arithmetic, use the built-in scipy.linalg functions: dot, inv, det, eig, *etc.*
- SciPy comes with fairly complete set of statistical distributions, tests, and functions. If you need it it's probably there.

NumPy: `http://wiki.scipy.org/Tentative_NumPy_Tutorial`

SciPy: `http://docs.scipy.org/doc/scipy/reference/tutorial`

SciNet