PWC Python Course: 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.

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4]

>>> b = [3, 5, 5, 6]
>>> b
[3, 5, 5, 6]

>>> 2 * a
[1, 2, 3, 4, 1, 2, 3, 4]

>>> a + b
[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.

```
>>> from numpy import zeros,
... ones, linspace
>>> zeros(5)
array([ 0., 0., 0., 0., 0.])

>>> ones(5, dtype = int)
array([ 1, 1, 1, 1, 1])
```

```
>>> zeros([2,2])
array([[ 0., 0.],
       [ 0., 0.]])

>>> arange(5)
array([ 0, 1, 2, 3, 4])

>>> linspace(1,5)
array([ 1., 1.08163265,
  1.16326531, 1.24489796,
  .
  .
  4.67346939, 4.75510204,
  4.83673469, 4.91836735, 5.  ])

>>> linspace(1, 5, 6)
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.

```
>>> zeros([2, 3])
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> a = zeros([2,3])
>>> a[1,2] = 1
>>> a[0,1] = 2
>>> a
array([[  0.,   2.,   0.],
       [  0.,   0.,   1.]])
```

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

IndexError: index (2) out of range

>>>
```

Copying arrays

Copying array variables

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

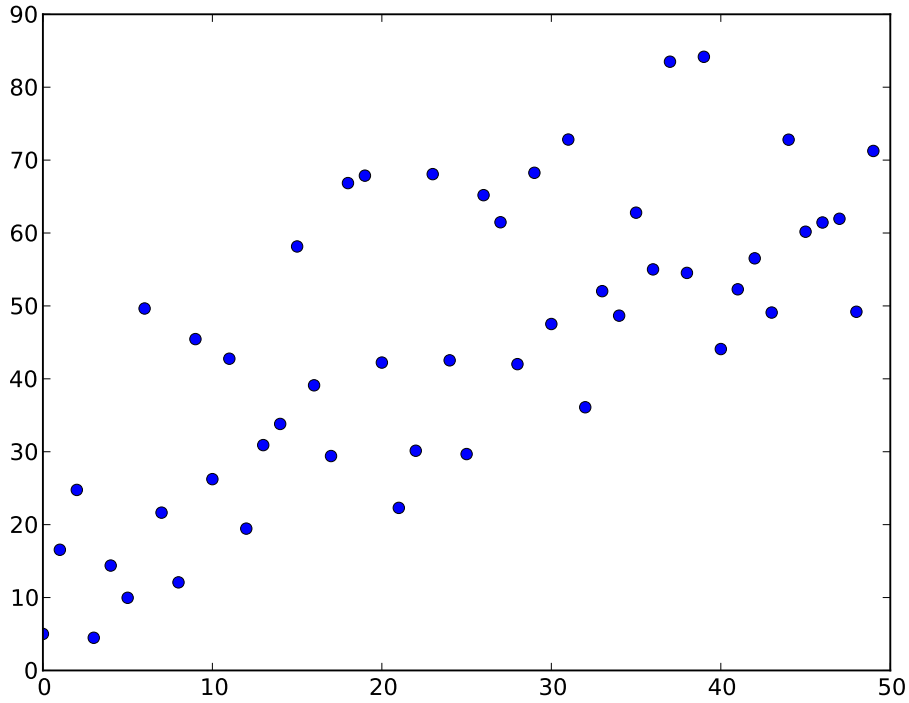
```
>>> a = 10; b = a; a = 20
>>> a, b
(20, 10)
>>> a = array([[1,2,3],
               [2,3,4]])
```

```
>>> b = a
>>> a[1,0] = -10
>>> a
array([[1, 2, 3],
       [-10, 3, 4]])
>>> b
array([[1, 2, 3],
       [-10, 3, 4]])
```

```
>>> b = a.copy()
>>> a[1,0] = 16
>>> a
array([[1, 2, 3],
       [16, 3, 4]])
>>> b
array([[1, 2, 3],
       [-10, 3, 4]])
```

Polynomial fitting

```
>>> from numpy import arange,
... zeros, polyfit, polyval
...
>>> import random
>>> x = arange(50.)
>>> y = zeros(50.)
>>> for i in arange(50.):
...  y[i]=x[i]+50*random.random()
...
>>> plot(x, y, 'o')
```



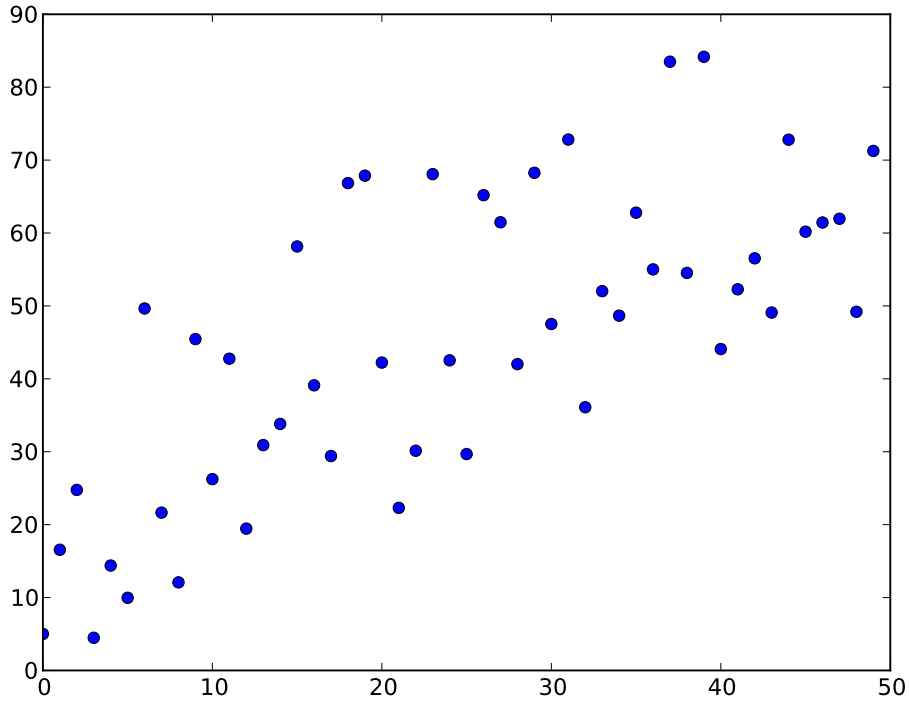
-
-
-
-
-
-

SciNet

Polynomial fitting

```
>>> from numpy import arange,
... zeros, polyfit, polyval
...
>>> import random
>>> x = arange(50.)
>>> y = zeros(50.)
>>> for i in arange(50.):
...  y[i]=x[i]+50*random.random()
...
>>> plot(x, y, 'o')
>>> fit = polyfit(x, y, 1)
>>> fit
array([  1.0073584,
        20.64695036])

>>>
```



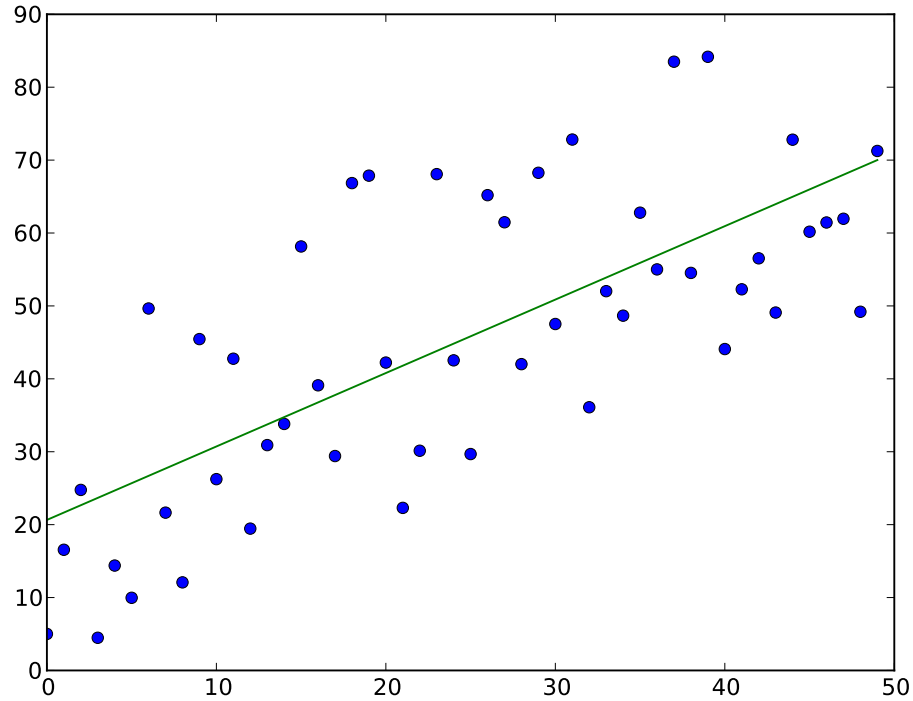

```
.
.
.
.
.
.
```

SciNet

Polynomial fitting

```
>>> from numpy import arange,
... zeros, polyfit, polyval
...
>>> import random
>>> x = arange(50.)
>>> y = zeros(50.)
>>> for i in arange(50.):
...  y[i]=x[i]+50*random.random()
...
>>> plot(x, y, 'o')
>>> fit = polyfit(x, y, 1)
>>> fit
array([  1.0073584,
        20.64695036])

>>> plot(x,polyval(fit, x))
```



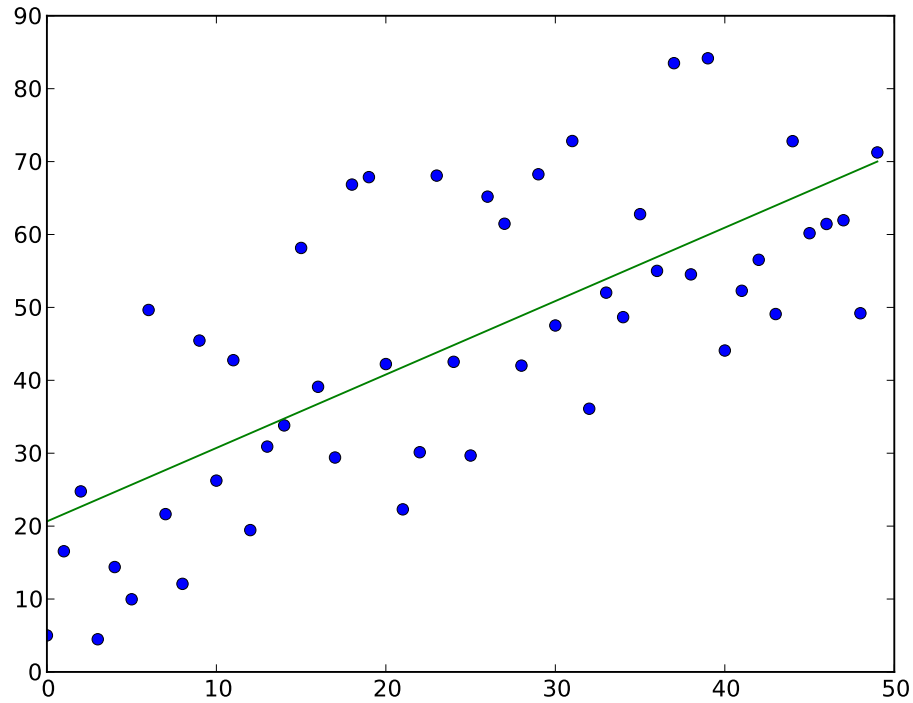
.
.
.
.
.
.

SciNet

Polynomial fitting

```
>>> from numpy import arange,
... zeros, polyfit, polyval
...
>>> import random
>>> x = arange(50.)
>>> y = zeros(50.)
>>> for i in arange(50.):
...  y[i]=x[i]+50*random.random()
...
>>> plot(x, y, 'o')
>>> fit = polyfit(x, y, 1)
>>> fit
array([  1.0073584,
        20.64695036])

>>> plot(x,polyval(fit, x))
```

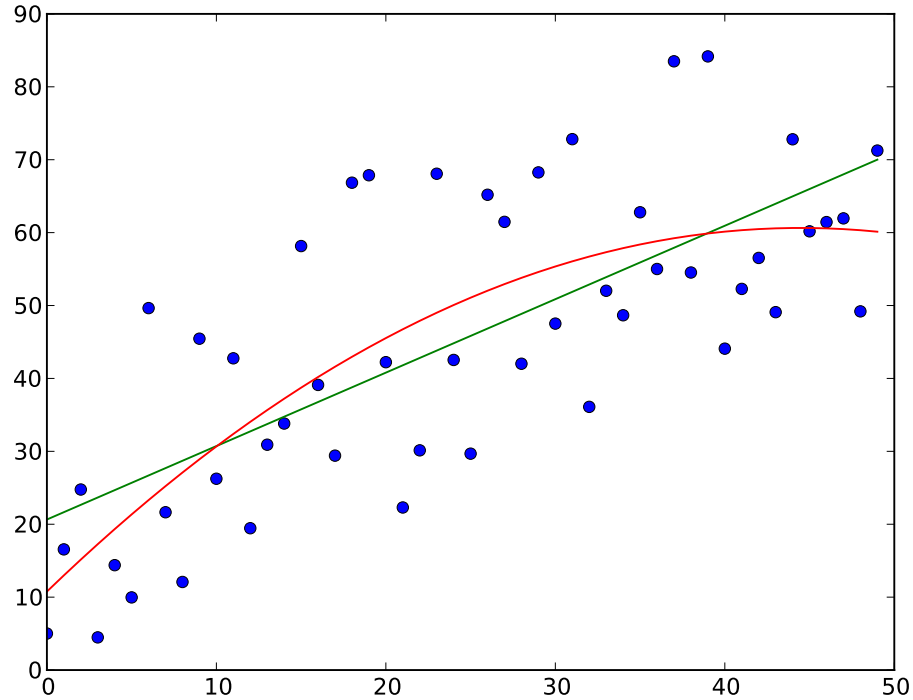



```
>>> fit = polyfit(x, y, 2)
>>> fit
array([ -0.02520835,
         2.24256777,
        10.76527546])
>>>
```

Polynomial fitting

```
>>> from numpy import arange,
... zeros, polyfit, polyval
...
>>> import random
>>> x = arange(50.)
>>> y = zeros(50.)
>>> for i in arange(50.):
...  y[i]=x[i]+50*random.random()
...
>>> plot(x, y, 'o')
>>> fit = polyfit(x, y, 1)
>>> fit
array([  1.0073584,
        20.64695036])

>>> plot(x,polyval(fit, x))
```




```
>>> fit = polyfit(x, y, 2)
>>> fit
array([ -0.02520835,
         2.24256777,
        10.76527546])
>>> 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.

```
>>> a = arange(4)
>>> a
array([0, 1, 2, 3])

>>> b = arange(4.) + 3
>>> b
array([3., 4., 5., 6.])

>>> c = 2
>>> c
2

>>> a * b
array([0., 4., 10., 18.])

>>> a * c
array([0, 2, 4, 6])

>>> b * c
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!

```
>>> a = array([[1, 2, 3],
               [2, 3, 4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])

>>> b = arange(3) + 1
>>> b
array([1, 2, 3])

>>> a * b
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.

```
>>> a = array([[1, 2, 3],
...            [2, 3, 4]])
>>> b = array([[1, 2, 3],
...            [2, 3, 4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])

>>> a * b
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'.

```
>>> from scipy import dot
>>> a = array([[1, 2, 3],
...            [2, 3, 4]])
>>> b = array([[1, 2, 3],
...            [2, 3, 4]])
>>> a
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a.transpose()
array([[1, 2],
       [2, 3],
       [3, 4]])

>>> dot(a.transpose(), b)
array([[ 5,  8, 11],
       [ 8, 13, 18],
       [11, 18, 25]])

>>> dot(b, a.transpose())
array([[14, 20],
       [20, 29]])

>>> c = arange(3) + 1
>>> dot(a,c)
array([14, 20])
```

SciNet

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.

```
>>> from scipy import dot, linalg
>>> a = array([[1,2,3], [3,4,5], [1,1,2]])
>>> linalg.det(a)
-2.0

>>>dot(a, linalg.inv(a))
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]])
```

SciNet

Solving systems of equations

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

$$Ax = b$$

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

```
>>> a = array([[1,2,3],
               [3,4,5],
               [1,1,2]])

>>> a
array([[ 1,  2, 3],
       [ 3, 4, 5],
       [ 1, 1, 2]])

>>> b = array([3, 4, 2])
>>> b
array([3, 4, 2])

>>> x = linalg.solve(a, b)
>>> x
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

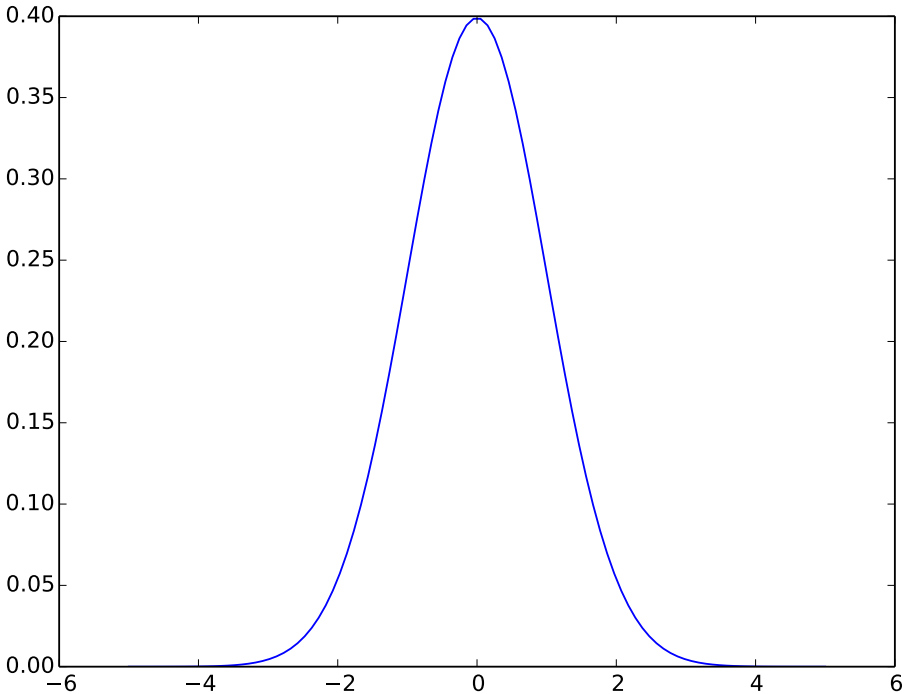
Normal distribution

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`.

```
>>> from scipy.stats import norm
>>> x = linspace(-5, 5, 100)
>>>


.
```

Normal distribution

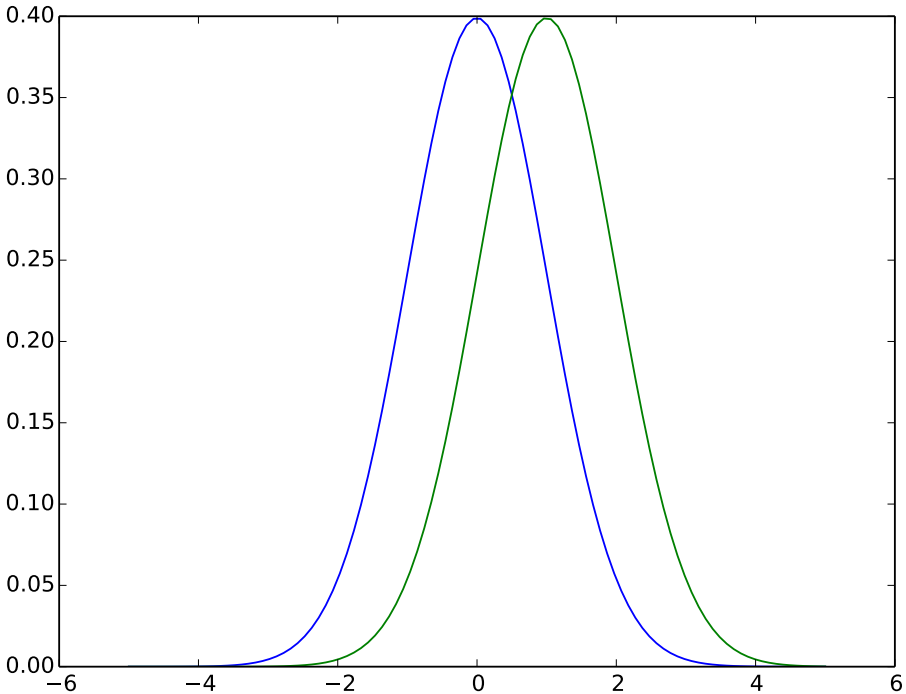


All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
>>> from scipy.stats import norm
>>> x = linspace(-5, 5, 100)
>>> plot(x, norm.pdf(x))
>>>
.
```

SciNet

Normal distribution

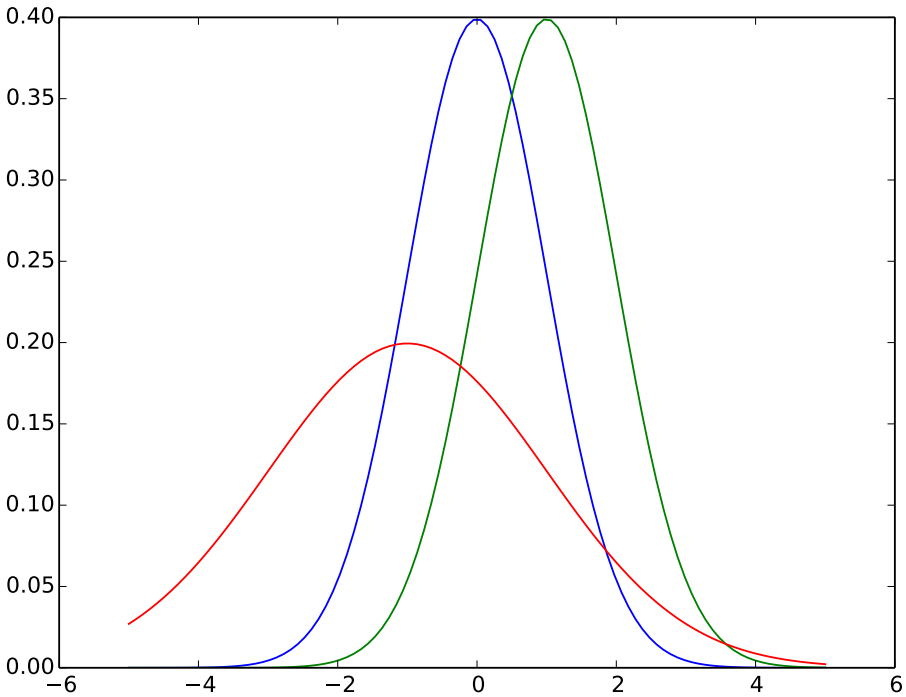


All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
>>> from scipy.stats import norm
>>> x = linspace(-5, 5, 100)
>>> plot(x, norm.pdf(x))
>>> plot(x, norm.pdf(x, loc = 1))
>>>
```

SciNet

Normal distribution



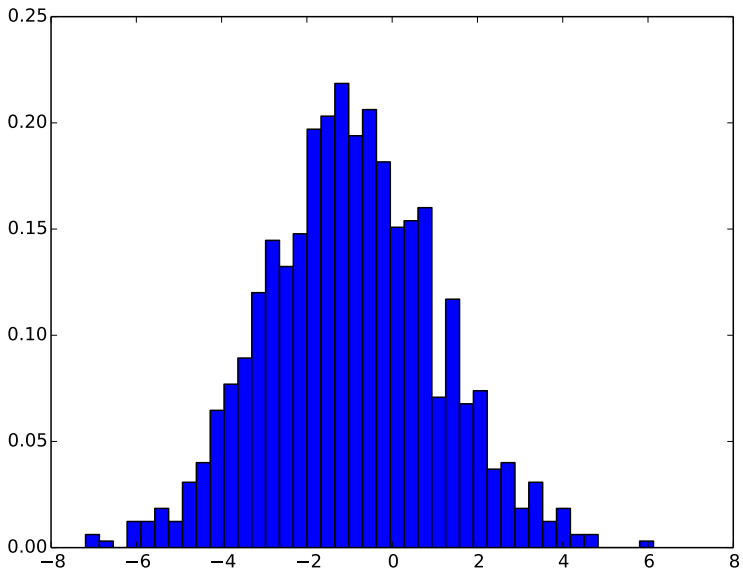
All continuous distributions take `loc` and `scale` as keyword parameters to adjust the location and scale of the distribution. In general the distribution of a random variable `X` is obtained from `(X - loc) / scale`. The default values are `loc = 0` and `scale = 1`.

```
>>> from scipy.stats import norm
>>> x = linspace(-5, 5, 100)
>>> plot(x, norm.pdf(x))
>>> plot(x, norm.pdf(x, loc = 1))
>>> plot(x, norm.pdf(x, loc = -1, scale = 2))
```

SciNet

Normal Distribution, continued

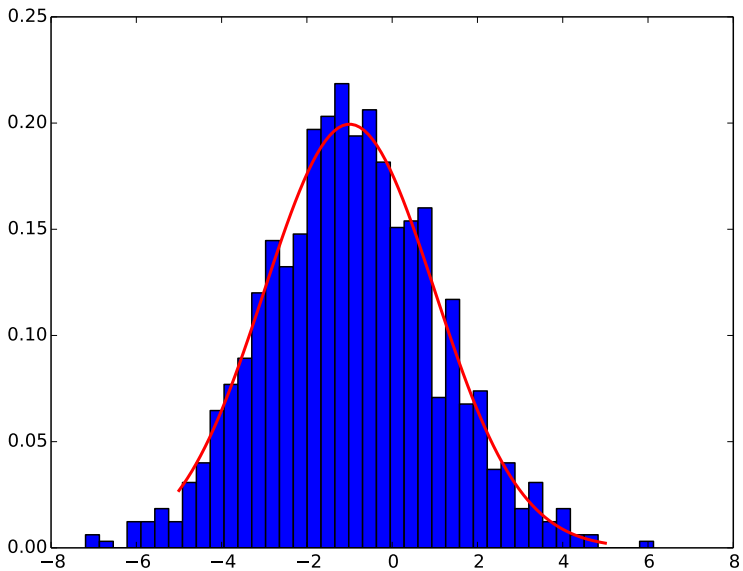
```
>>> from pylab import hist
>>> norm.mean(loc = -1, scale = 2)
-1.0
>>> norm.std(loc = -1, scale = 2)
2.0
>>> norm.moment(3, loc = -1, scale = 2)
-13.0
>>> samples = norm.rvs(size = 1000, loc = -1, scale = 2)
>>> h = hist(samples, bins = 41, normed = True)
>>>
```



SciNet

Normal Distribution, continued

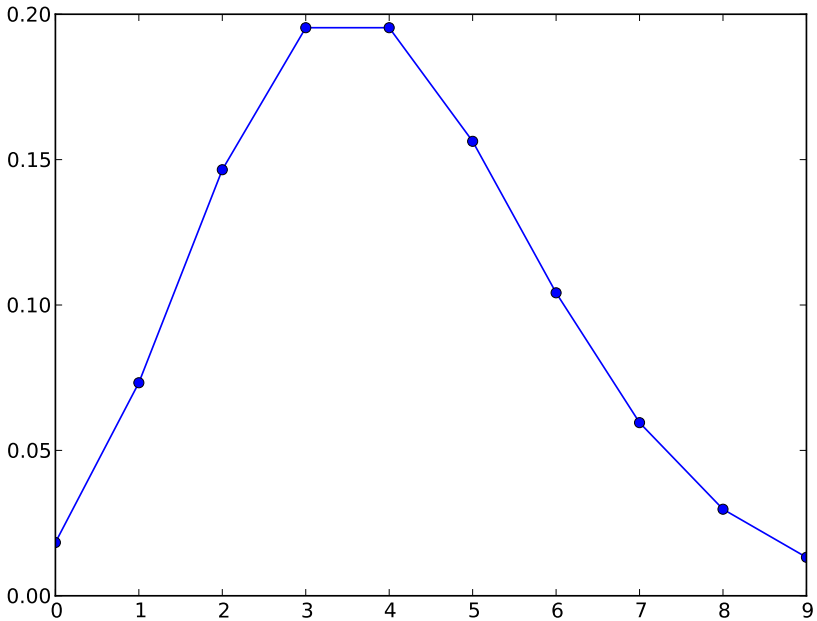
```
>>> from pylab import hist
>>> norm.mean(loc = -1, scale = 2)
-1.0
>>> norm.std(loc = -1, scale = 2)
2.0
>>> norm.moment(3, loc = -1, scale = 2)
-13.0
>>> samples = norm.rvs(size = 1000, loc = -1, scale = 2)
>>> h = hist(samples, bins = 41, normed = True)
>>> plot(x,norm.pdf(x,loc = -1,scale = 2),'r',linewidth=2)
```



SciNet

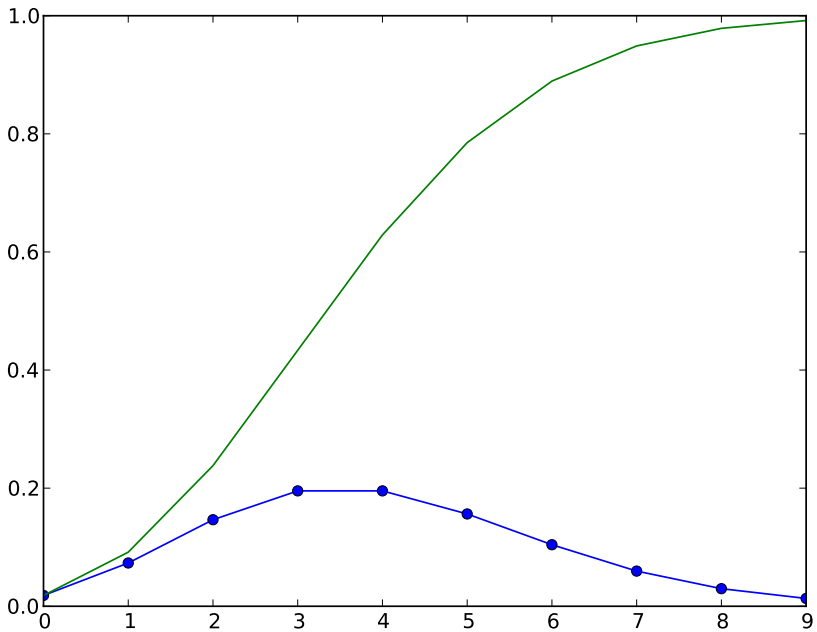
Poisson distribution

```
>>> from scipy.stats import poisson
>>> x = arange(10)
>>> poisson.mean(4)
4.0
>>> poisson.var(4)
4.0
>>> plot(x, poisson.pmf(x, 4), 'o-')
>>>
```



Poisson distribution

```
>>> from scipy.stats import poisson
>>> x = arange(10)
>>> poisson.mean(4)
4.0
>>> poisson.var(4)
4.0
>>> plot(x, poisson.pmf(x, 4), 'o-')
>>> 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