Download notebook

Python for R Programmers

Christopher Fonnesbeck

Vanderbilt University

14 June, 2012

What is Python?

Python is a **dynamic**, **interpreted** programming language that is **flexible** enough to be used for a variety of tasks, from scripts to mission-critical applications. It currently has a very strong presence in **scientific computing**. Despite being an easy-to-use, high-level language, it is also capable of achieving very high performance.

Runnable pseudocode: Python's syntax is readable and clear.

In [1]:
```
from numpy import zeros, random, sqrt
gamma = random.gamma
normal = random.normal

def pygibbs(N=20000, thin=200):
    mat = zeros((N,2))
    x,y = mat[0]
    for i in range(N):
        for j in range(thin):
            x = gamma(3, y**2 + 4)
            y = normal(1./(x+1), 1./sqrt(2*(x+1)))
        mat[i] = x,y

    return mat
```

Python objects are **dynamically referenced**

In [2]:
```
y = 5
y = 'foo'
x = y = [1, 2, 3]
y[0] = -9
x
```

Out[2]:
```
[-9, 2, 3]
```

... but **strongly typed**!

In [3]:
```
'5' + 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-cfde8aeadc63> in <module>()
----> 1 '5' + 6

TypeError: cannot concatenate 'str' and 'int' objects
```

NumPy and SciPy

NumPy and SciPy are fundamental scientific modules for Python.

In particular, **NumPy** provides:

- fast, efficient mulitidimensional array data structure: `ndarray`
- functions for operating on arrays
- tools for integrating code from compiled languages
- RNGs, linear algebra functions, Fourier transform

SciPy provides a large suite of standard scientific computing:

- statistical distribtuion classes
- sparse matrices
- signal processing
- optimizers
- numerical integration and DE solvers
- additional linear algebra tools

$$\text{NumPy} + \text{SciPy} \approx \text{MATLAB}$$

In [3]:
```
# Calculate second difference matrix
import numpy as np

I2 = -2*np.eye(8)
E = np.diag(np.ones(7), k=-1)
I2 + E + E.T
```

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

In [4]:
```
from scipy import linalg

def lse(A, b, B, d, cond=None):
    """
    Equality-contrained least squares.

    The following algorithm minimizes ||Ax - b|| subject to the
    constrain Bx = d.
    """

    A, b, B, d = map(np.asanyarray, (A, b, B, d))
    p = B.shape[0]

    # QR decomposition of constraint matrix B
    Q, R = linalg.qr(B.T)

    # Solve Ax = b, assuming A is triangular
    y = linalg.solve_triangular(R[:p, :p], d, trans='T', lower=False)
    A = np.dot(A, Q)

    # Least squares solution to Ax = b
    z = linalg.lstsq(A[:, p:], b - np.dot(A[:, :p], y),
                     cond=cond)[0].ravel()
```

```
    return np.dot(Q[:, :p], y) + np.dot(Q[:, p:], z)
```

In [5]:
```
A, b = [[0, 2, 3], [1, 3, 4.5]], [1, 1]
B, d = [[1, 1, 0]], [1]
lse(A, b, B, d)
```

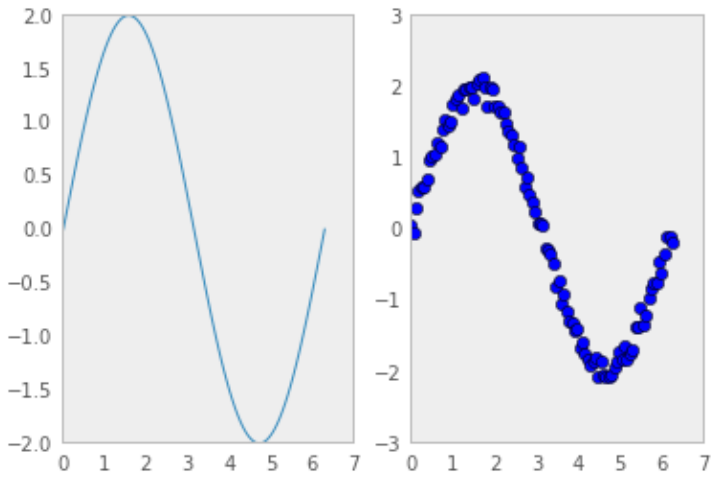
Out[5]:
```
array([-0.5   ,  1.5   , -0.6667])
```

Matplotlib

This is the most widely-used 2-D plotting module for Python, allowing users to generate publication-quality plots. **Matplotlib** offers a high level of control and the ability to generate interactive output.

In [6]:
```
import matplotlib.pyplot as plt

fig = plt.figure()
x = np.linspace(0,2*np.pi,100)
y = 2*np.sin(x)
ax = fig.add_subplot(1,2,1)
ax.plot(x,y)
y2 = y + 0.1*np.random.normal( size=x.shape )
ax = fig.add_subplot(1,2,2)
ax.plot(x,y2,'bo')
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x10c269fd0>]
```

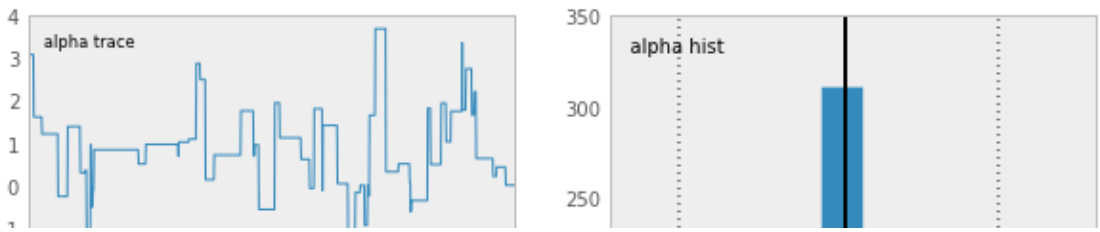


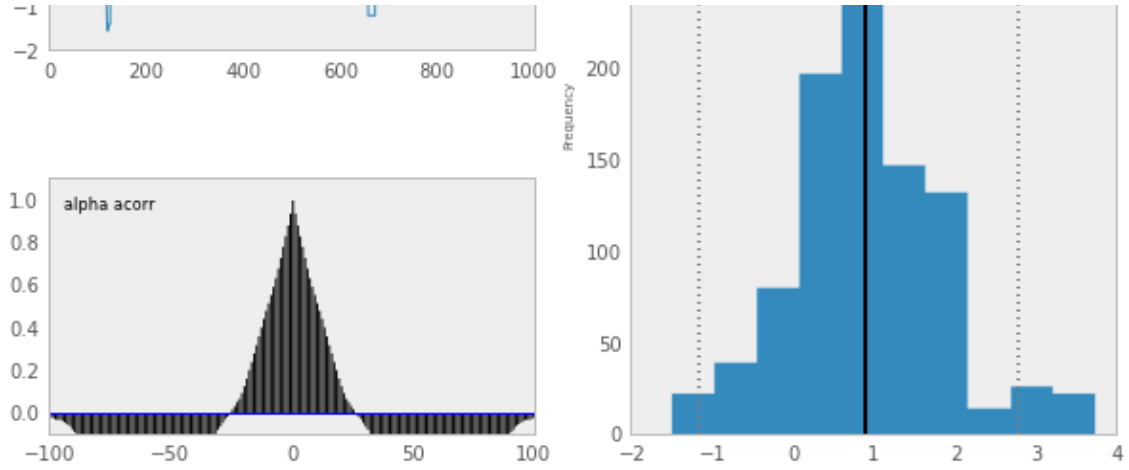
In [7]:
```
from pymc.examples import gelman_bioassay
from pymc import MCMC, Matplot
M = MCMC(gelman_bioassay)
M.sample(1000,verbose=0)
```

```
[****************100%******************]  1000 of 1000 complete
```

In [8]:
```
Matplot.plot(M.alpha)
```

```
Plotting alpha
```

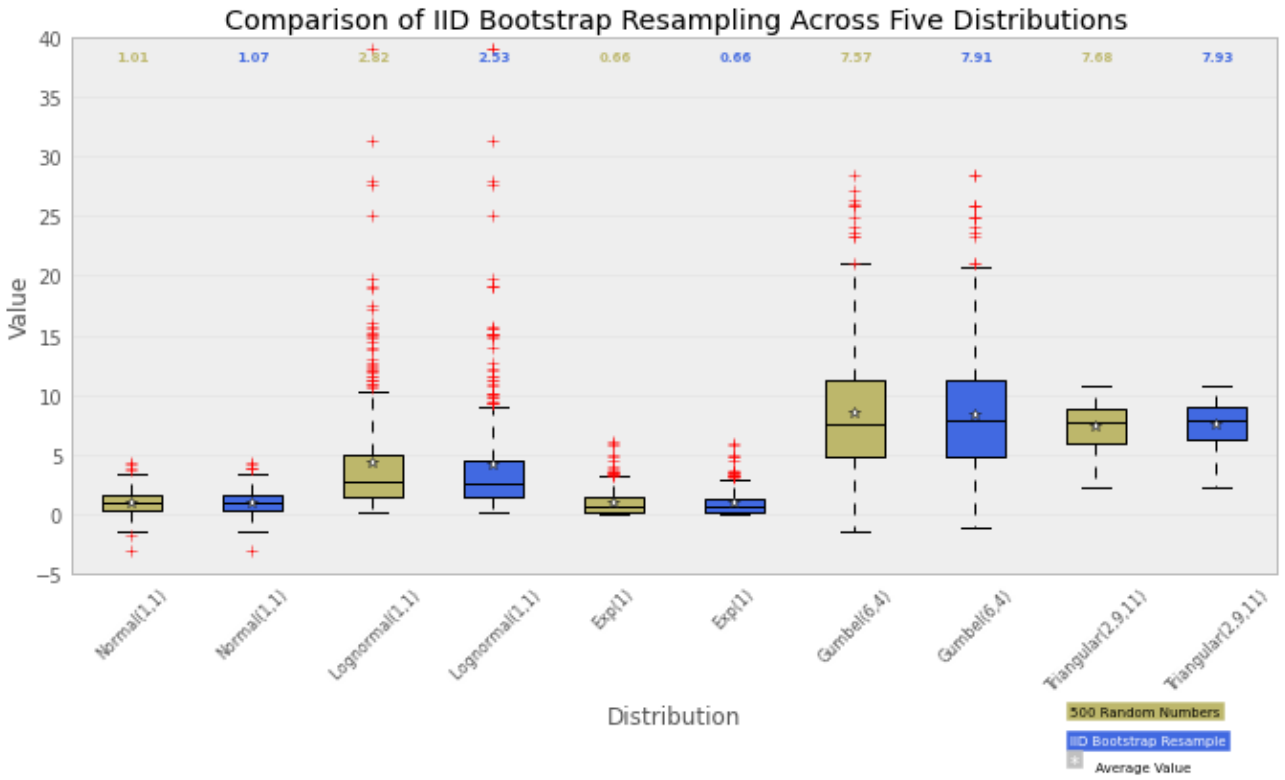




In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon


# Generate some data from five different probability distributions,
# each with different characteristics. We want to play with how an IID
# bootstrap resample of the data preserves the distributional
# properties of the original sample, and a boxplot is one visual tool
# to make this assessment
numDists = 5
randomDists = ['Normal(1,1)',' Lognormal(1,1)', 'Exp(1)', 'Gumbel(6,4)',
               'Triangular(2,9,11)']
N = 500
norm = np.random.normal(1,1, N)
logn = np.random.lognormal(1,1, N)
expo = np.random.exponential(1, N)
gumb = np.random.gumbel(6, 4, N)
tria = np.random.triangular(2, 9, 11, N)

# Generate some random indices that we'll use to resample the original data
# arrays. For code brevity, just use the same random indices for each array
bootstrapIndices = np.random.random_integers(0, N-1, N)
normBoot = norm[bootstrapIndices]
expoBoot = expo[bootstrapIndices]
gumbBoot = gumb[bootstrapIndices]
lognBoot = logn[bootstrapIndices]
triaBoot = tria[bootstrapIndices]

data = [norm, normBoot,  logn, lognBoot, expo, expoBoot, gumb, gumbBoot,
        tria, triaBoot]

fig = plt.figure(figsize=(10,6))
fig.canvas.set_window_title('A Boxplot Example')
ax1 = fig.add_subplot(111)
plt.subplots_adjust(left=0.075, right=0.95, top=0.9, bottom=0.25)

bp = plt.boxplot(data, notch=0, sym='+', vert=1, whis=1.5)
plt.setp(bp['boxes'], color='black')
plt.setp(bp['whiskers'], color='black')
plt.setp(bp['fliers'], color='red', marker='+')

# Add a horizontal grid to the plot, but make it very light in color
# so we can use it for reading data values but not be distracting
ax1.yaxis.grid(True, linestyle='-', which='major', color='lightgrey',
               alpha=0.5)
```

```
# Hide these grid behind plot objects
ax1.set_axisbelow(True)
ax1.set_title('Comparison of IID Bootstrap Resampling Across Five Distributions')
ax1.set_xlabel('Distribution')
ax1.set_ylabel('Value')

# Now fill the boxes with desired colors
boxColors = ['darkkhaki','royalblue']
numBoxes = numDists*2
medians = range(numBoxes)
for i in range(numBoxes):
  box = bp['boxes'][i]
  boxX = []
  boxY = []
  for j in range(5):
      boxX.append(box.get_xdata()[j])
      boxY.append(box.get_ydata()[j])
  boxCoords = zip(boxX,boxY)
  # Alternate between Dark Khaki and Royal Blue
  k = i % 2
  boxPolygon = Polygon(boxCoords, facecolor=boxColors[k])
  ax1.add_patch(boxPolygon)
  # Now draw the median lines back over what we just filled in
  med = bp['medians'][i]
  medianX = []
  medianY = []
  for j in range(2):
      medianX.append(med.get_xdata()[j])
      medianY.append(med.get_ydata()[j])
      plt.plot(medianX, medianY, 'k')
      medians[i] = medianY[0]
  # Finally, overplot the sample averages, with horixzontal alignment
  # in the center of each box
  plt.plot([np.average(med.get_xdata())], [np.average(data[i])],
           color='w', marker='*', markeredgecolor='k')

# Set the axes ranges and axes labels
ax1.set_xlim(0.5, numBoxes+0.5)
top = 40
bottom = -5
ax1.set_ylim(bottom, top)
xtickNames = plt.setp(ax1, xticklabels=np.repeat(randomDists, 2))
plt.setp(xtickNames, rotation=45, fontsize=8)

# Due to the Y-axis scale being different across samples, it can be
# hard to compare differences in medians across the samples. Add upper
# X-axis tick labels with the sample medians to aid in comparison
# (just use two decimal places of precision)
pos = np.arange(numBoxes)+1
upperLabels = [str(np.round(s, 2)) for s in medians]
weights = ['bold', 'semibold']
for tick,label in zip(range(numBoxes),ax1.get_xticklabels()):
   k = tick % 2
   ax1.text(pos[tick], top-(top*0.05), upperLabels[tick],
        horizontalalignment='center', size='x-small', weight=weights[k],
        color=boxColors[k])

# Finally, add a basic legend
plt.figtext(0.80, 0.08,  str(N) + ' Random Numbers' ,
           backgroundcolor=boxColors[0], color='black', weight='roman',
           size='x-small')
plt.figtext(0.80, 0.045, 'IID Bootstrap Resample',
backgroundcolor=boxColors[1],
           color='white', weight='roman', size='x-small')
```

```
plt.figtext(0.80, 0.015, '*', color='white', backgroundcolor='silver',
            weight='roman', size='medium')
plt.figtext(0.815, 0.013, ' Average Value', color='black', weight='roman',
            size='x-small')

plt.show()
```



IPython

IPython is an enhanced Python shell which provides a more robust and productive development environment for users. It includes the **HTML notebook** featured here, as well as support for **interactive data visualization** and easy high-performance **parallel computing**.

Magic functions

IPython has a set of predefined 'magic functions' that you can call with a command line style syntax. These include:

- `%run`
- `%edit`
- `%debug`
- `%timeit`
- `%paste`
- `%load_ext`

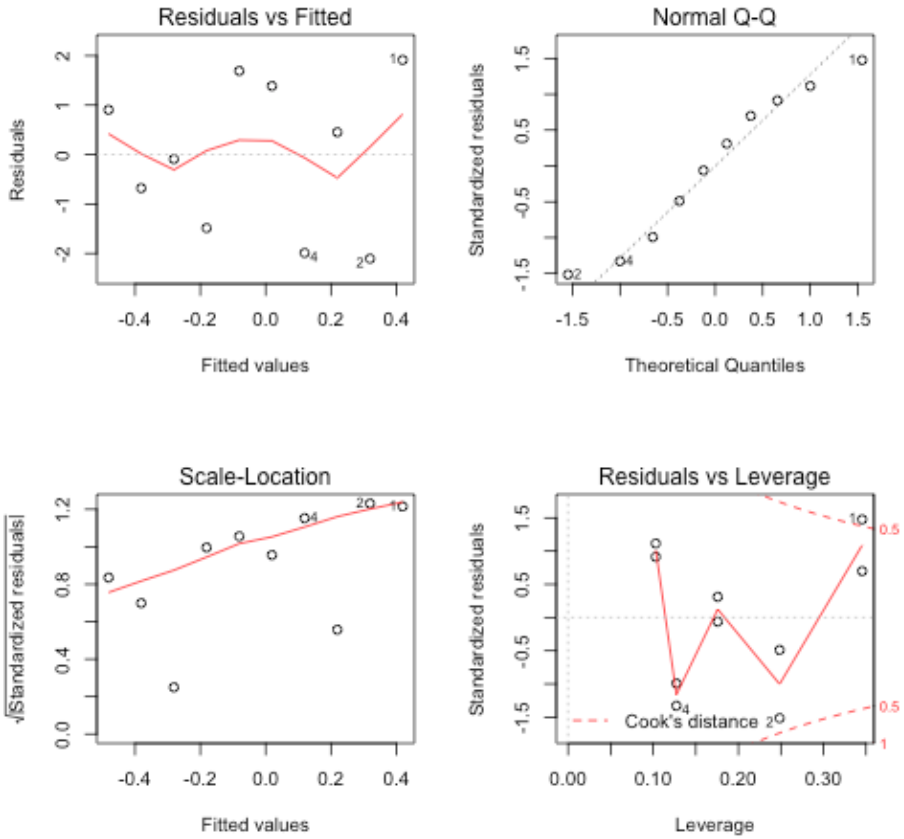
For example, we can use `%load_ext` to load an extension to run R code within IPython:

In [12]:
```
%load_ext rmagic
```

In [13]:
```
x,y = arange(10), random.normal(size=10)
```

In [14]:
```
%%R -i x,y -o XYcoef
lm.fit <- lm(y~x)
par(mfrow=c(2,2))
```

```
plot(lm.fit)
XYcoef <- coef(lm.fit)
```



In [15]:
```
XYcoef
```

Out[15]:
```
array([ 0.41949922, -0.10004703])
```

Dirk's Rcpp benchmarks

In [16]:
```
%%R
library(compiler)
library(rbenchmark)
library(inline)
library(RcppGSL)

Rgibbs <- function(N,thin) {
    mat <- matrix(0,ncol=2,nrow=N)
    x <- 0
    y <- 0
    for (i in 1:N) {
        for (j in 1:thin) {
            x <- rgamma(1,3,y*y+4)
```

```
            y <- rnorm(1,1/(x+1),1/sqrt(2*(x+1)))
        }
        mat[i,] <- c(x,y)
    }
    mat
}


RCgibbs <- cmpfun(Rgibbs)

gibbscode <- '

  // n and thin are SEXPs which the Rcpp::as function maps to C++ vars
  int N   = as<int>(n);
  int thn = as<int>(thin);

  int i,j;
  NumericMatrix mat(N, 2);

  RNGScope scope;          // Initialize Random number generator

  // The rest of the code follows the R version
  double x=0, y=0;

  for (i=0; i<N; i++) {
    for (j=0; j<thn; j++) {
      x = ::Rf_rgamma(3.0,1.0/(y*y+4));
      y = ::Rf_rnorm(1.0/(x+1),1.0/sqrt(2*x+2));
    }
    mat(i,0) = x;
    mat(i,1) = y;
  }

  return mat;              // Return to R
'

RcppGibbs <- cxxfunction(signature(n="int", thin = "int"), gibbscode, plugin="Rcpp")

N <- 1000
thn <- 10
res <- benchmark(Rgibbs(N, thn), RCgibbs(N, thn), RcppGibbs(N, thn),
order="relative", replications=10)
print(res)
```

```
Loading required package: Rcpp
               test replications elapsed relative user.self sys.self user.child
3 RcppGibbs(N, thn)           10   0.027  1.00000     0.027    0.001          0
2   RCgibbs(N, thn)           10   0.888 32.88889     0.883    0.005          0
1    Rgibbs(N, thn)           10   1.126 41.70370     1.115    0.010          0
  sys.child
3         0
2         0
1         0
```

Pandas

Pandas provides high-performance data structures for data manipulation and analysis. In particular, it allows for intelligent **data alignment** and integrated handling of **missing data**. Datasets can be easily reshaped, sliced, subsetted, and indexed hierarchically.

data.frame $\subsetneq$ pandas.DataFrame

Here's a trivial example of a pandas `DataFrame`, populated with columns of various types:

In [17]:
```
from pandas import *
df = DataFrame({'A' : ['one', 'one', 'two', 'three'] * 6,
                'B' : ['A', 'B', 'C'] * 8,
                'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
                'D' : np.random.randn(24),
                'E' : np.random.randn(24),
                'F' : DateRange(start='4/1/2012', periods=24)})

df
```

Out[17]:

	A	B	C	D	E	F
0	one	A	foo	0.799663	0.030021	2012-04-02 00:00:00
1	one	B	foo	-0.522646	0.220671	2012-04-03 00:00:00
2	two	C	foo	-0.468866	0.016852	2012-04-04 00:00:00
3	three	A	bar	-0.393212	0.593464	2012-04-05 00:00:00
4	one	B	bar	-0.575404	-1.621361	2012-04-06 00:00:00
5	one	C	bar	0.228977	-1.256740	2012-04-09 00:00:00
6	two	A	foo	-1.226878	-0.467621	2012-04-10 00:00:00
7	three	B	foo	-1.449318	1.317463	2012-04-11 00:00:00
8	one	C	foo	0.158045	-0.388078	2012-04-12 00:00:00
9	one	A	bar	-0.149894	0.498263	2012-04-13 00:00:00
10	two	B	bar	0.061315	-0.999789	2012-04-16 00:00:00
11	three	C	bar	-0.087721	0.697154	2012-04-17 00:00:00
12	one	A	foo	0.849112	0.044553	2012-04-18 00:00:00
13	one	B	foo	-1.305085	-1.332287	2012-04-19 00:00:00
14	two	C	foo	-1.435599	-1.261241	2012-04-20 00:00:00
15	three	A	bar	-0.560977	-1.133941	2012-04-23 00:00:00
16	one	B	bar	-0.672480	-0.093903	2012-04-24 00:00:00
17	one	C	bar	0.299152	-0.886278	2012-04-25 00:00:00
18	two	A	foo	1.912571	-1.428091	2012-04-26 00:00:00
19	three	B	foo	0.474440	-0.104162	2012-04-27 00:00:00
20	one	C	foo	-0.451927	-0.044103	2012-04-30 00:00:00
21	one	A	bar	0.135511	1.126078	2012-05-01 00:00:00
22	two	B	bar	-1.067146	0.057069	2012-05-02 00:00:00
23	three	C	bar	0.485477	-0.241935	2012-05-03 00:00:00

Pandas ships with a few sample datasets, such as 100 records of baseball batting data, which is easily imported into a `DataFrame` using the `read_csv` function:

In [18]:
```
baseball = read_csv("/Users/fonnescj/Code/pandas/doc/data/baseball.csv")
baseball
```

Out[18]:
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 88641 to 89534
Data columns:
id       100  non-null values
```

```
year      100  non-null values
stint     100  non-null values
team      100  non-null values
lg        100  non-null values
g         100  non-null values
ab        100  non-null values
r         100  non-null values
h         100  non-null values
X2b       100  non-null values
X3b       100  non-null values
hr        100  non-null values
rbi       100  non-null values
sb        100  non-null values
cs        100  non-null values
bb        100  non-null values
so        100  non-null values
ibb       100  non-null values
hbp       100  non-null values
sh        100  non-null values
sf        100  non-null values
gidp      100  non-null values
dtypes: float64(9), int64(10), object(3)
```

The `DataFrame` possesses a suite of methods for indexing, slicing and manipulating its contents. Here's an arbitrary record:

In [19]:
```
baseball.ix[88650]
```

Out[19]:
```
id        johnsra05
year           2006
stint             1
team            NYA
lg               AL
g                33
ab                6
r                 0
h                 1
X2b               0
X3b               0
hr                0
rbi               0
sb                0
cs                0
bb                0
so                4
ibb               0
hbp               0
sh                0
sf                0
gidp              0
Name: 88650
```

We can carry out spreadsheet-like actions on a `DataFrame`, such as pivoting and crosstabs:

In [20]:
```
pivot_table(baseball, values=['hr','so'], rows=['team', 'year'], aggfunc=sum)
```

Out[20]:

		hr	so
team	**year**		
ARI	**2006**	15	58
ARI	**2007**	0	13
ATL	**2007**	0	29

BAL	**2007**	1	23
BOS	**2006**	0	1
BOS	**2007**	20	101
CHA	**2007**	35	134
CHN	**2006**	1	4
CHN	**2007**	9	48
CIN	**2007**	36	127
CLE	**2007**	0	29
COL	**2007**	2	14
DET	**2007**	37	176
FLO	**2007**	0	0
HOU	**2007**	14	212
KCA	**2007**	2	15
LAA	**2007**	0	0
LAN	**2006**	0	7
LAN	**2007**	36	141
MIL	**2006**	0	2
MIL	**2007**	9	60
MIN	**2007**	6	32
NYA	**2006**	0	4
NYA	**2007**	0	0
NYN	**2007**	61	310
OAK	**2007**	8	65
PHI	**2007**	0	26
SDN	**2007**	0	31
SFN	**2006**	6	55
SFN	**2007**	40	188
SLN	**2007**	13	98
TBA	**2007**	0	0
TEX	**2007**	28	140
TOR	**2007**	58	265

In [21]:
```
crosstab(baseball.year, baseball.lg, margins=True, colnames=['league'])
```

Out[21]:

league	**AL**	**NL**	**All**
year			
2006	3	5	8
2007	35	57	92
All	38	62	100

Not to mention powerful split/apply/combine methods:

In [22]:
```
def get_stats(group):
    return {'max': group.max(), 'count': group.count(), 'mean': group.mean()}
```

In [23]:
```
baseball.groupby('lg')["sb"].apply(get_stats)
```

Out[23]:
```
lg
AL    {'count': 38, 'max': 22.0, 'mean': 1.4210526315789473}
NL    {'count': 62, 'max': 14.0, 'mean': 1.3548387096774193}
```

Statsmodels

statsmodels is a rapidly-evolving module for statistical modeling in Python, providing a set of models, statistical tests, plotting functions including:

- Linear regression models
- Generalized linear models
- Discrete choice models
- Robust linear models
- Time series analysis
- Nonparametric estimators

In [24]:
```
import statsmodels.api as sm
data = sm.datasets.scotland.load()
data.exog = sm.add_constant(data.exog)
```

```
/Library/Python/2.7/site-packages/statsmodels-0.5.0-py2.7-macosx-10.7-
intel.egg/statsmodels/tools/tools.py:301: FutureWarning: The default of `prepend` wil
changed to True in 0.5.0, use explicit prepend
  FutureWarning)
```

In [25]:
```
gamma_model = sm.GLM(data.endog, data.exog, family=sm.families.Gamma())
gamma_results = gamma_model.fit()
gamma_results.summary()
```

Out[25]:

Generalized Linear Model Regression Results

Dep. Variable:	y	**No. Observations:**	32
Model:	GLM	**Df Residuals:**	24
Model Family:	Gamma	**Df Model:**	7
Link Function:	inverse_power	**Scale:**	0.00358428317349
Method:	IRLS	**Log-Likelihood:**	-83.017
Date:	Thu, 14 Jun 2012	**Deviance:**	0.087389
Time:	13:07:46	**Pearson chi2:**	0.0860
No. Iterations:	5		

	coef	**std err**	**t**	**P>|t|**	**[95.0% Conf. Int.]**
x1	4.962e-05	1.62e-05	3.060	0.005	1.78e-05 8.14e-05
x2	0.0020	0.001	3.824	0.001	0.001 0.003
x3	-7.181e-05	2.71e-05	-2.648	0.014	-0.000 -1.87e-05
x4	0.0001	4.06e-05	2.757	0.011	3.23e-05 0.000
x5	-1.468e-07	1.24e-07	-1.187	0.247	-3.89e-07 9.56e-08

x6	-0.0005	0.000	-2.159	0.041	-0.001 -4.78e-05
x7	-2.427e-06	7.46e-07	-3.253	0.003	-3.89e-06 -9.65e-07
const	-0.0178	0.011	-1.548	0.135	-0.040 0.005

Cython and f2py

f2py is a Fortran interface generator for Python that ships with NumPy. It is compatible with NumPy's `ndarray` objects, and allows for the use of Fortran 77/90/95 code within Python by simply adding structured comments to functions and subroutines.

```
      SUBROUTINE bernoulli(x,p,nx,np,like)

Cf2py logical dimension(nx),intent(in) :: x
Cf2py double precision dimension(np),intent(in) :: p
Cf2py integer intent(hide),depend(x) :: nx=len(x)
Cf2py integer intent(hide),depend(p),check(len(p)==1 || len(p)==len(x)):: np=len(p)
Cf2py double precision intent(out) :: like
Cf2py threadsafe

      IMPLICIT NONE

      INTEGER np,nx,i
      DOUBLE PRECISION p(np), ptmp, like
      LOGICAL x(nx)
      LOGICAL not_scalar_p
      DOUBLE PRECISION infinity
      PARAMETER (infinity = 1.7976931348623157d308)

C     Check parameter size
      not_scalar_p = (np .NE. 1)

      like = 0.0
      ptmp = p(1)
      do i=1,nx
        if (not_scalar_p) ptmp = p(i)
        if (ptmp .LT. 0.0) then
          like = -infinity
          RETURN
        endif

        if (x(i)) then
          like = like + dlog(ptmp)
        else
          like = like + dlog(1.0D0 - ptmp)
        endif

      enddo
      return
      END
```

In [26]:
```
from pymc.flib import bernoulli
bernoulli([1,0,1], 0.3)
```

Out[26]: -2.7646205525906047

Cython is a language that allows Python programmers to write fast code without having to write C/C++/Fortran directly. It looks much like Python code, but with type declarations. Cython takes this code, translates it to C, then compiles the generated C code to create a Python extension.

Benchmark example: Gibbs sampling

Gibbs sampler for function:

$$f(x,y) = xx^2 \exp(-xy^2 - y^2 + 2y - 4x)$$

using conditional distributions:

$$x|y \sim Gamma(3, y^2 + 4)$$
$$y|x \sim Normal(\frac{1}{1+x}, \frac{1}{2(1+x)})$$

Here again is the pure Python implementation:

In [27]:
```
from numpy import zeros, random, sqrt
gamma = random.gamma
normal = random.normal

def pygibbs(N=20000, thin=200):
    mat = zeros((N,2))
    x,y = mat[0]
    for i in range(N):
        for j in range(thin):
            x = gamma(3, y**2 + 4)
            y = normal(1./(x+1), 1./sqrt(2*(x+1)))
        mat[i] = x,y

    return mat
```

In [28]:
```
timeit pygibbs(1000, 10)
```

```
10 loops, best of 3: 86.3 ms per loop
```

We can usually get a marginal speedup simply by compiling the Python code, unchanged, using cython:

In [29]:
```
%load_ext cythonmagic
```

In [30]:
```
%%cython
from numpy import zeros, random, sqrt
gamma = random.gamma
normal = random.normal

def pygibbs2(N=20000, thin=200):
    mat = zeros((N,2))
    x,y = mat[0]
    for i in range(N):
        for j in range(thin):
            x = gamma(3, y**2 + 4)
            y = normal(1./(x+1), 1./sqrt(2*(x+1)))
        mat[i] = x,y

    return mat
```

In [31]:
```
timeit pygibbs2(1000, 10)
```

```
10 loops, best of 3: 76.5 ms per loop
```

For additional gains, we can simply declare variable types, like you would in C:

In [32]:
```
%%cython
from numpy import zeros, random, sqrt
```

```
from numpy cimport *
gamma = random.gamma
normal = random.normal

def pygibbs3(int N=20000, int thin=200):
    cdef ndarray[float64_t, ndim=2] mat = zeros((N,2))
    cdef float64_t x,y = 0
    cdef int i,j
    for i in range(N):
        for j in range(thin):
            x = gamma(3, y**2 + 4)
            y = normal(1./(x+1), 1./sqrt(2*(x+1)))
        mat[i] = x,y

    return mat
```

In [33]:
```
timeit pygibbs3(1000, 10)
```

```
10 loops, best of 3: 68.7 ms per loop
```

A full-flown "cythonization" involves using GSL's random number generators, and giving cython a few more instructions:

In [34]:
```
%%cython -lm -lgsl -lgslcblas

cimport cython
import numpy as np
from numpy cimport *

cdef extern from "math.h":
    double sqrt(double)

cdef extern from "gsl/gsl_rng.h":
    ctypedef struct gsl_rng_type
    ctypedef struct gsl_rng

    gsl_rng_type *gsl_rng_mt19937
    gsl_rng *gsl_rng_alloc(gsl_rng_type * T) nogil

cdef extern from "gsl/gsl_randist.h":
    double gamma "gsl_ran_gamma"(gsl_rng * r,double,double)
    double gaussian "gsl_ran_gaussian"(gsl_rng * r,double)

cdef gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937)

@cython.wraparound(False)
@cython.boundscheck(False)
def gibbs(int N=20000,int thin=500):
    cdef:
        double x=0
        double y=0
        int i, j
        ndarray[float64_t, ndim=2] samples

    samples = np.empty((N,thin))
    for i from 0 <= i < N:
        for j from 0 <= j < thin:
            x = gamma(r,3,1.0/(y*y+4))
            y = gaussian(r,1.0/sqrt(x+1))
        samples[i,0] = x
        samples[i,1] = y
    return samples
```

In [35]:
```
timeit gibbs(1000, 10)
```

```
1000 loops, best of 3: 1.38 ms per loop
```

Cython parallel

Before running the next cell, make sure you have first started your cluster, you can use the clusters tab in the dashboard to do so. Because this example transfers lots of large arrays, we recommend that you first configure your cluster to use the 'NoDB' hub messaging support, which removes a few features but has the lowest memory footprint. You can do so by putting in your IPython profile directory a file called `ipcontroller_config.py` that contains simply:

```
# Configuration file for ipcontroller.
c = get_config()
# The class to use for the DB backend
c.HubFactory.db_class = 'IPython.parallel.controller.dictdb.NoDB'
```

See the IPython docs for further details.

In [37]:
```
from IPython.parallel import Client
rc = Client()
dv = rc[:]
dv.block = True
dv.activate()
```

Now, we load the cython magic on all engines and execute the cython magic as well on all engines:

In [38]:
```
%px %load_ext cythonmagic
```

```
Parallel execution on engine(s): [0, 1]
```

In [42]:
```
%%px
%%cython -lm -lgsl -lgslcblas

cimport cython
import numpy as np
from numpy cimport *

cdef extern from "math.h":
    double sqrt(double)

cdef extern from "gsl/gsl_rng.h":
    ctypedef struct gsl_rng_type
    ctypedef struct gsl_rng

    gsl_rng_type *gsl_rng_mt19937
    gsl_rng *gsl_rng_alloc(gsl_rng_type * T) nogil

cdef extern from "gsl/gsl_randist.h":
    double gamma "gsl_ran_gamma"(gsl_rng * r,double,double)
    double gaussian "gsl_ran_gaussian"(gsl_rng * r,double)

cdef gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937)

@cython.wraparound(False)
@cython.boundscheck(False)
def gibbs(int N=20000,int thin=500):
    cdef:
        double x=0
        double y=0
        int i, j
        ndarray[float64_t, ndim=2] samples

    samples = np.empty((N,thin))
```

```
    for i from 0 <= i < N:
        for j from 0 <= j < thin:
            x = gamma(r,3,1.0/(y*y+4))
            y = gaussian(r,1.0/sqrt(x+1))
        samples[i,0] = x
        samples[i,1] = y
    return samples
```

```
Parallel execution on engine(s): [0, 1]
```

Divide the array by the number of nodes. In this case they divide evenly, a more general partitioning of sizes is easy to do as well.

In [43]:
```
N = 1000
thin = 10
n = N/len(rc.ids)
dv.push(dict(n=n, thin=thin))
# Let's just confirm visually we got what we expect
dv['n']
```

Out[43]:
```
[500, 500]
```

We can time purely the execution of the gibbs sampler on the remote nodes

In [44]:
```
%%timeit
dv.execute('gibbs(n, thin)')
```

```
100 loops, best of 3: 8.23 ms per loop
```

But a more realistic (and costly) benchmark must also include the cost of bringing the results back from the cluster engines to our local namespace. For that, we assign the call to the variable `a` on each node and then use the view's `gather` method to pull them back in:

In [45]:
```
%%timeit
dv.execute('a = gibbs(n, thin)')
a = dv.gather('a')
```

```
100 loops, best of 3: 13.9 ms per loop
```

Here we see how on a 4-core machine we get close to a 4-fold speedup on pure execution, but once we have to pay the price of communicating the results back to the local engine, the benefit is less. In a real parallel application, much of the art of optimization comes in trying to keep the movement of large amounts of data to a minimum, as communication is always *much* more expensive than computation in today's architectures.

Gibbs Sampler Shootout

Timed on a 11" MacBook Air (1.8 GHz Intel Core i7)

- Python 2.7.1, Cython 0.16
- Julia 0.0.0
- R 2.14.1, Rcpp 0.9.10

Elapsed time for 10 replications of the Gibbs sampler code, each run for 4 million iterations.

model	elapsed	relative
Julia distributed	3.38	1.0
GSLGibbs	5.46	1.6
Cython	5.58	1.7
Julia native RNG	7.03	2.1
RcppGibbs	9.95	2.9

BoostGibbs	10.65	3.2
Julia libRMath	17.27	5.1
Pure Python	278.51	82.4
RCgibbs	328.56	97.2
Rgibbs	419.87	124.2