Introduction to Python (IntroductionToPythonSolutions.html)
Getting started with Python and the IPython notebook (IPythonNotebookIntroduction.html)
Functions are first class objects (FunctionsSolutions.html)
Function argumnents (FunctionsSolutions.html#function-argumnents)
Higher-order functions (FunctionsSolutions.html#higher-order-functions)
Anonymous functions (FunctionsSolutions.html#anonymous-functions)
Pure functions (FunctionsSolutions.html#pure-functions)
Recursion (FunctionsSolutions.html#recursion)
Iterators (FunctionsSolutions.html#iterators)
Generators (FunctionsSolutions.html#generators)
Decorators (FunctionsSolutions.html#decorators)
The `operator` module (FunctionsSolutions.html#the-operator-module)
The `functools` module (FunctionsSolutions.html#the-functools-module)
The `itertools` module (FunctionsSolutions.html#the-itertools-module)
The `toolz`, `fn` and `funcy` modules (FunctionsSolutions.html#the-toolz-fn-and-funcy-modules)
Exercises (FunctionsSolutions.html#exercises)
Data science is OSEMN (DataProcessingSolutions.html)
Working with text (TextProcessingSolutions.html)
Preprocessing text data (TextProcessingExtras.html)
Working with structured data (WorkingWithStructuredData.html)
Using numpy (UsingNumpySolutions.html)
Using Pandas (UsingPandas.html)
Using R from IPython (UsingPandas.html#using-r-from-ipython)
Computational problems in statistics (ComputationalStatisticsMotivation.html)
Computer numbers and mathematics (ComputerArithmetic.html)
Algorithmic complexity (AlgorithmicComplexity.html)

Linear Algebra and Linear Systems (LinearAlgebraReview.html)

Linear Algebra and Matrix Decompositions (LinearAlgebraMatrixDecompWithSolutions.html)

Change of Basis (PCASolutions.html)

Optimization and Non-linear Methods (OptimizationInOneDimension.html)

Practical Optimizatio Routines (BlackBoxOptimization.html)

Fitting ODEs with the Levenberg–Marquardt algorithm (CalibratingODEs.html)

Algorithms for Optimization and Root Finding for Multivariate Problems (MultivariateOptimizationAlgortihms.html)

Expectation Maximizatio (EM) Algorithm (EMAlgorithm.html)

Monte Carlo Methods (MonteCarlo.html)

Resampling methods (ResamplingAndMonteCarloSimulations.html)

Markov Chain Monte Carlo (MCMC) (MCMC.html)

Using PyMC2 ()

Using PyMC3 (PyMC3.html)

Using PyStan (PyStan.html)

Animations of Metropolis, Gibbs and Slice Sampler dynamics (Animation.html)

C Crash Course (CrashCourseInC.html)

Code Optimization (MakingCodeFast.html)

Using C code in Python (FromCToPython.html)

Using functions from various compiled languages in Python (FromCompiledToPython.html)

Julia and Python (FromJuliaToPython.html)

Converting Python Code to C for speed (FromPythonToC.html)

Optimization bake-off (Optimization_Bakeoff.html)

Writing Parallel Code (WritingParallelCode.html)

Massively parallel programming with GPUs (CUDAPython.html)

Writing CUDA in C (GPUsAndCUDAC.html)

Distributed computing for Big Data (DistributedComputing.html)

Hadoop MapReduce on AWS EMR with **`mrjob`** (MapReduce.html)

Spark on a local mahcine using 4 nodes (Spark.html)

Modules and Packaging (ModulesAndPackaging.html)

Tour of the Jupyter (IPython3) notebook (Jupyter.html)

Polyglot programming (MultiKernel.html)

What you should know and learn more about (ReveiwAndTrends.html)

Search

```
from __future__ import division
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')
```

```
np.random.seed(1234)
import pymc
import scipy.stats as stats
```

Using PyMC2

Install PyMC2 with

```
conda install -c pymc pymc
```

- Dcoumentation for PyMC2 (http://pymc-devs.github.io/pymc/)

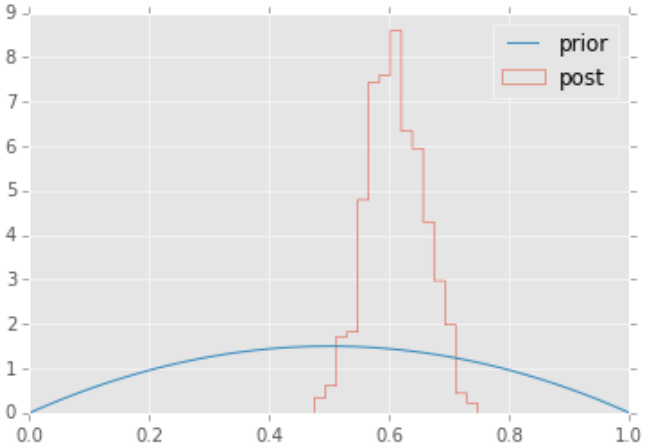
Coin toss

We'll repeat the example of determining the bias of a coin from observed coin tosses. The likelihood is binomial, and we use a beta prior.

```
n = 100
h = 61
alpha = 2
beta = 2

p = pymc.Beta('p', alpha=alpha, beta=beta)
y = pymc.Binomial('y', n=n, p=p, value=h, observed=True)
m = pymc.Model([p, y])
```

```
mc = pymc.MCMC(m, )
mc.sample(iter=11000, burn=10000)
plt.hist(p.trace(), 15, histtype='step', normed=True, label='post');
x = np.linspace(0, 1, 100)
plt.plot(x, stats.beta.pdf(x, alpha, beta), label='prior');
plt.legend(loc='best');
```

```
[-----------------100%-----------------] 11000 of 11000 complete in 1.5 sec
```

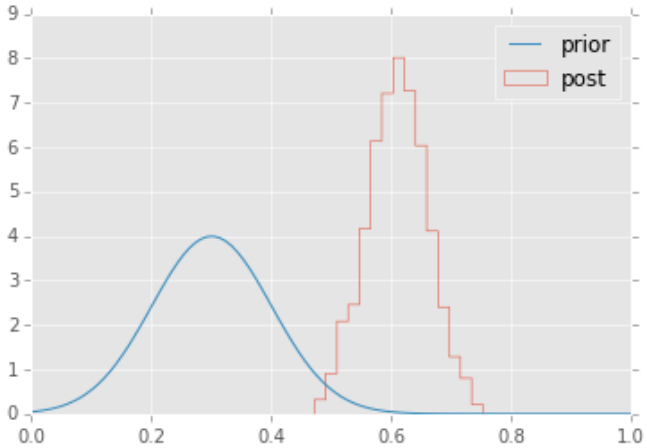


Since the computer is doing all the work, we don't need to use a conjugate prior if we have good reasons not to.

```
p = pymc.TruncatedNormal('p', mu=0.3, tau=10, a=0, b=1)
y = pymc.Binomial('y', n=n, p=p, value=h, observed=True)
m = pymc.Model([p, y])
```

```
mc = pymc.MCMC(m)
mc.sample(iter=11000, burn=10000)
plt.hist(p.trace(), 15, histtype='step', normed=True, label='post');
a, b = plt.xlim()
x = np.linspace(0, 1, 100)
a, b = (0 - 0.3) / 0.1, (1 - 0.3) / 0.1
plt.plot(x, stats.truncnorm.pdf(x, a, b, 0.3, 0.1), label='prior');
plt.legend(loc='best');
```

```
[-----------------100%-----------------] 11000 of 11000 complete in 1.5 sec
```



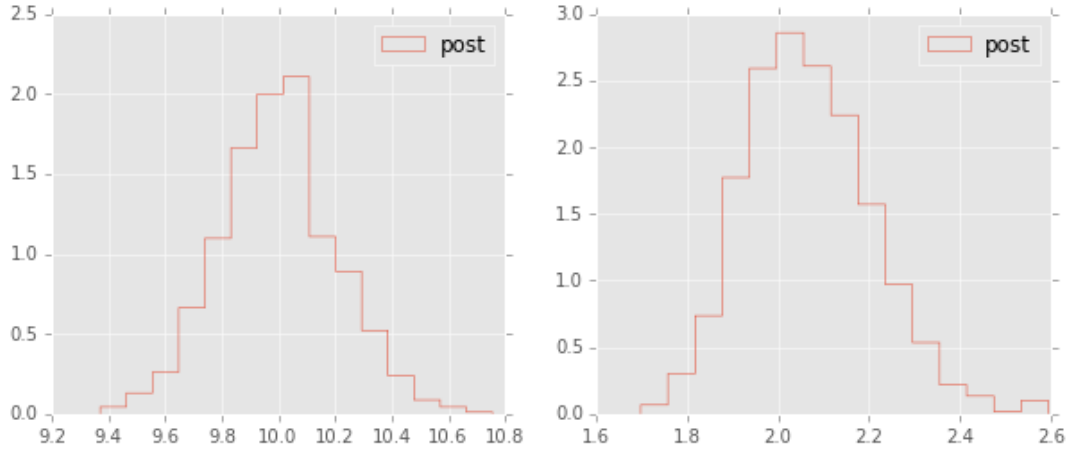
Estimating mean and standard deviation of normal distribution

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

```
# generate observed data
N = 100
y = np.random.normal(10, 2, N)

# define priors
mu = pymc.Uniform('mu', lower=0, upper=100)
tau = pymc.Uniform('tau', lower=0, upper=1)

# define likelihood
y_obs = pymc.Normal('Y_obs', mu=mu, tau=tau, value=y, observed=True)

# inference
m = pymc.Model([mu, tau, y])
mc = pymc.MCMC(m)
mc.sample(iter=11000, burn=10000)
```

```
[-----------------100%-----------------] 11000 of 11000 complete in 3.2 sec
```

```
plt.figure(figsize=(10,4))
plt.subplot(121)
plt.hist(mu.trace(), 15, histtype='step', normed=True, label='post');
plt.legend(loc='best');
plt.subplot(122)
plt.hist(np.sqrt(1.0/tau.trace()), 15, histtype='step', normed=True, label='pos
plt.legend(loc='best');
```



Estimating parameters of a linear regreession model

We will show how to estimate regression parameters using a simple linear modesl

$$y \sim ax + b$$

We can restate the linear model

$$y = ax + b + \epsilon$$

as sampling from a probability distribution

$$y \sim \mathcal{N}(ax + b, \sigma^2)$$

Now we can use pymc to estimate the paramters a, b and σ (pymc2 uses precision τ which is $1/\sigma^2$ so we need to do a simple transformation). We will assume the following priors

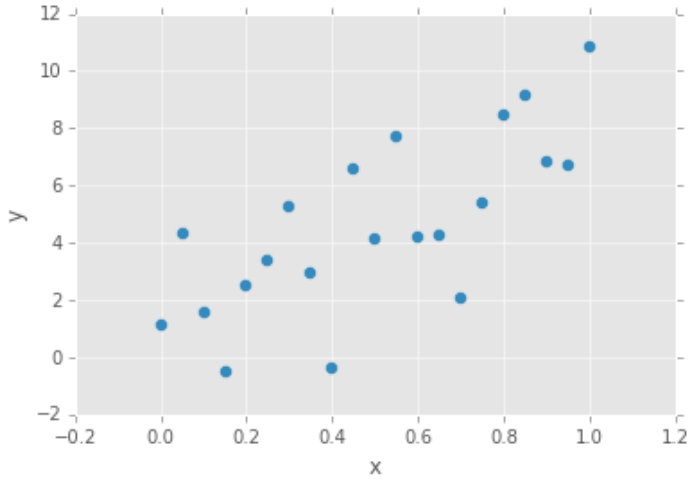
$$\begin{aligned} a &\sim \mathcal{N}(0, 100) \\ b &\sim \mathcal{N}(0, 100) \\ \tau &\sim \text{Gamma}(0.1, 0.1) \end{aligned}$$

Here we need a helper function to let PyMC know that the mean is a deterministic function of the parameters a, b and x. We can do this with a decorator, like so:

```
@pymc.deterministic
def mu(a=a, b=b, x=x):
    return a*x + b
```

```
# observed data
n = 21
a = 6
b = 2
sigma = 2
x = np.linspace(0, 1, n)
y_obs = a*x + b + np.random.normal(0, sigma, n)
data = pd.DataFrame(np.array([x, y_obs]).T, columns=['x', 'y'])
```

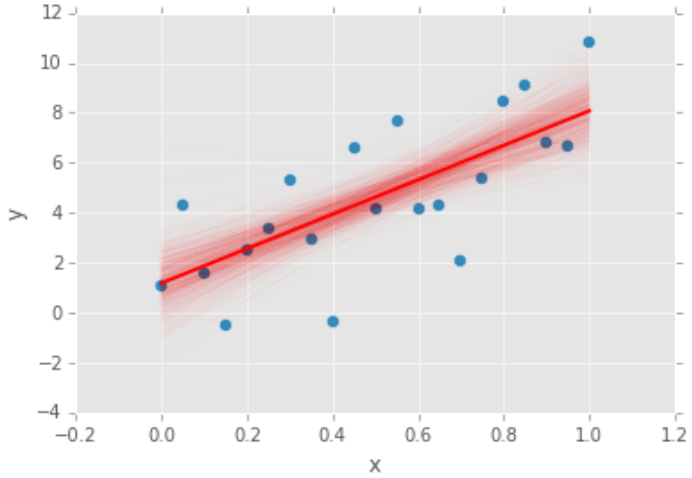
```
data.plot(x='x', y='y', kind='scatter', s=50);
```



```
# define priors
a = pymc.Normal('slope', mu=0, tau=1.0/10**2)
b = pymc.Normal('intercept', mu=0, tau=1.0/10**2)
tau = pymc.Gamma("tau", alpha=0.1, beta=0.1)

# define likelihood
@pymc.deterministic
def mu(a=a, b=b, x=x):
    return a*x + b

y = pymc.Normal('y', mu=mu, tau=tau, value=y_obs, observed=True)

# inference
m = pymc.Model([a, b, tau, x, y])
mc = pymc.MCMC(m)
mc.sample(iter=11000, burn=10000)
```

```
[-----------------100%-----------------] 11000 of 11000 complete in 6.1 sec
```

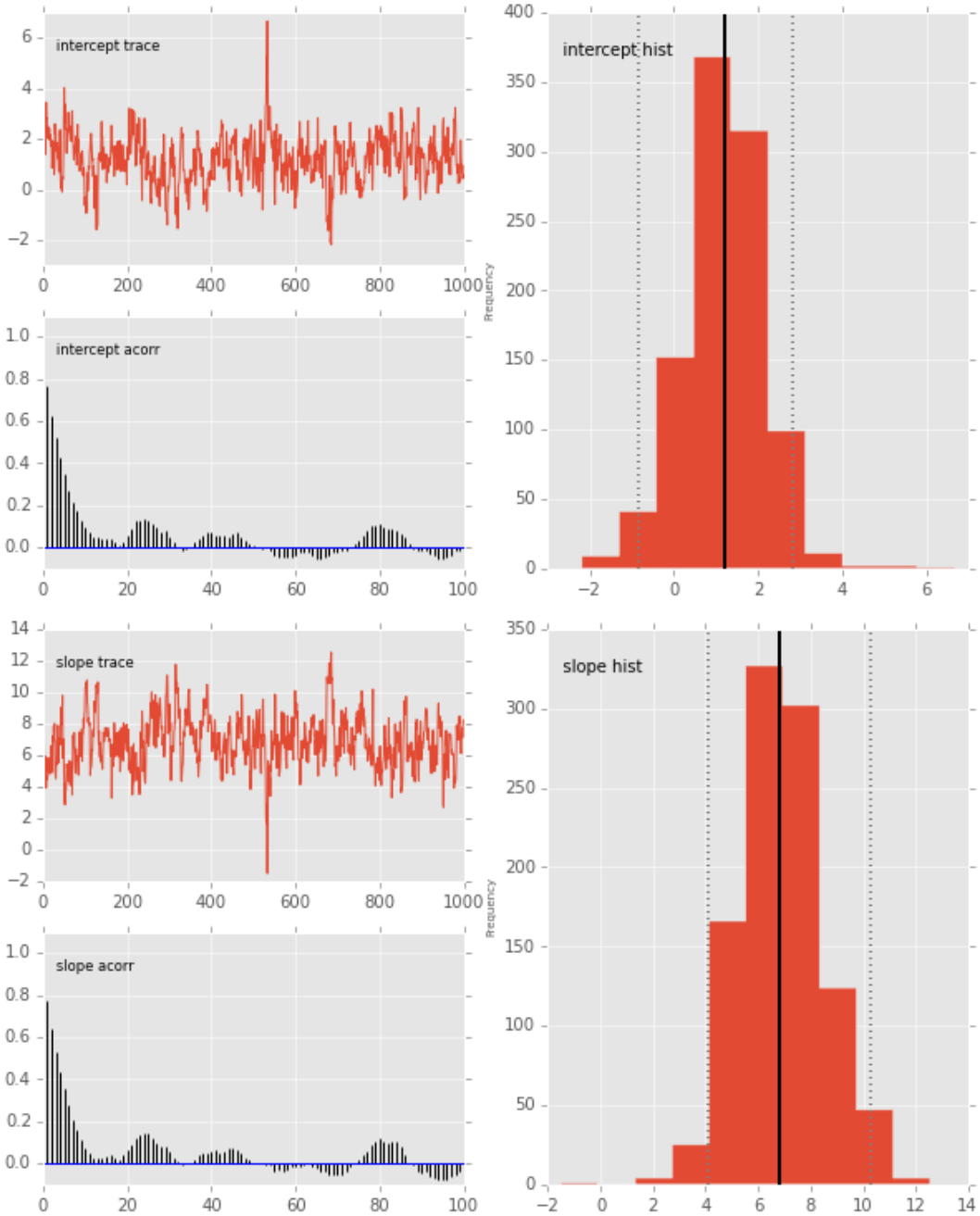
```
abar = a.stats()['mean']
bbar = b.stats()['mean']
data.plot(x='x', y='y', kind='scatter', s=50);
xp = np.array([x.min(), x.max()])
plt.plot(a.trace()*xp[:, None] + b.trace(), c='red', alpha=0.01)
plt.plot(xp, abar*xp + bbar, linewidth=2, c='red');
```

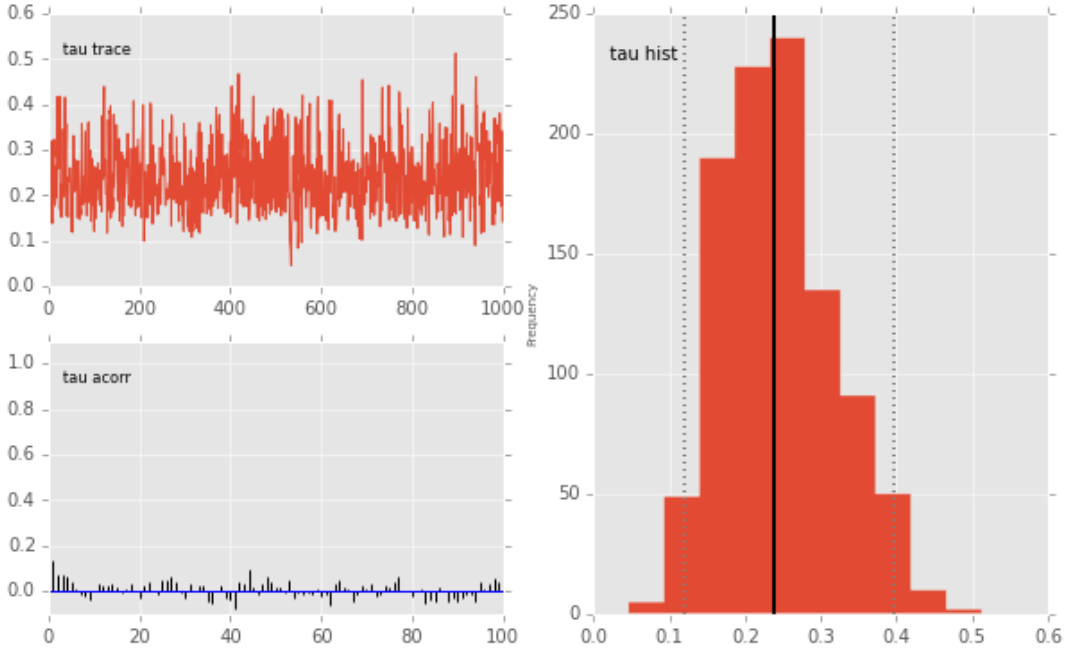



```
pymc.Matplot.plot(mc)
```

```
Plotting intercept
Plotting slope
Plotting tau
```

```
/Users/cliburn/anaconda/lib/python2.7/site-packages/numpy/core/fromnumeric.py:2
  VisibleDeprecationWarning)
```


intercept trace
intercept hist
intercept acorr
slope trace
slope hist
slope acorr
Frequency
Frequency



Estimating parameters of a logistic model

Gelman's book has an example where the dose of a drug may be affected to the number of rat deaths in an experiment.

Dose (log g/ml)	# Rats	# Deaths
-0.896	5	0
-0.296	5	1
-0.053	5	3
0.727	5	5

We will model the number of deaths as a random sample from a binomial distribution, where n is the number of rats and p the probabbility of a rat dying. We are given $n = 5$, but we believve that p may be related to the drug dose x. As x increases the number of rats dying seems to increase, and since p is a probability, we use the

following model:

$$
\begin{aligned}
y &\sim \text{Bin}(n, p) \\
\text{logit}(p) &= \alpha + \beta x \\
\alpha &\sim \mathcal{N}(0, 5) \\
\beta &\sim \mathcal{N}(0, 10)
\end{aligned}
$$

where we set vague priors for α and β, the parameters for the logistic model.

```
# define invlogit function
def invlogit(x):
    return pymc.exp(x) / (1 + pymc.exp(x))
```

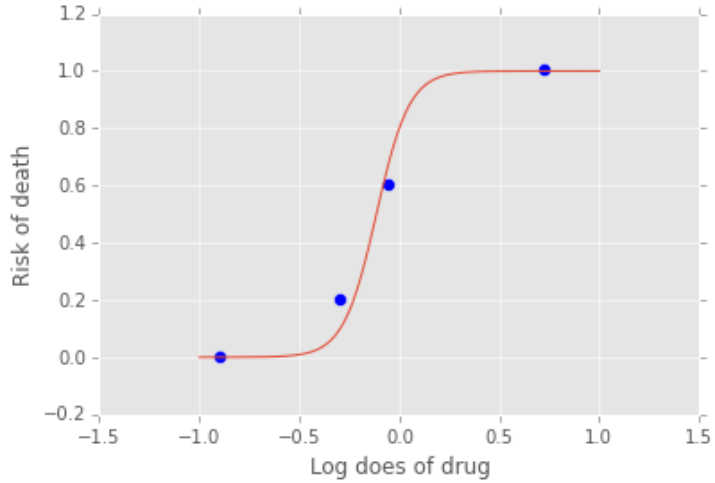
```
# observed data
n = 5 * np.ones(4)
x = np.array([-0.896, -0.296, -0.053, 0.727])
y_obs = np.array([0, 1, 3, 5])

# define priors
alpha = pymc.Normal('alpha', mu=0, tau=1.0/5**2)
beta = pymc.Normal('beta', mu=0, tau=1.0/10**2)

# define likelihood
p = pymc.InvLogit('p', alpha + beta*x)
y = pymc.Binomial('y_obs', n=n, p=p, value=y_obs, observed=True)

# inference
m = pymc.Model([alpha, beta, y])
mc = pymc.MCMC(m)
mc.sample(iter=11000, burn=10000)
```

```
[-----------------100%-----------------] 11000 of 11000 complete in 6.9 sec
```

```
beta.stats()
```

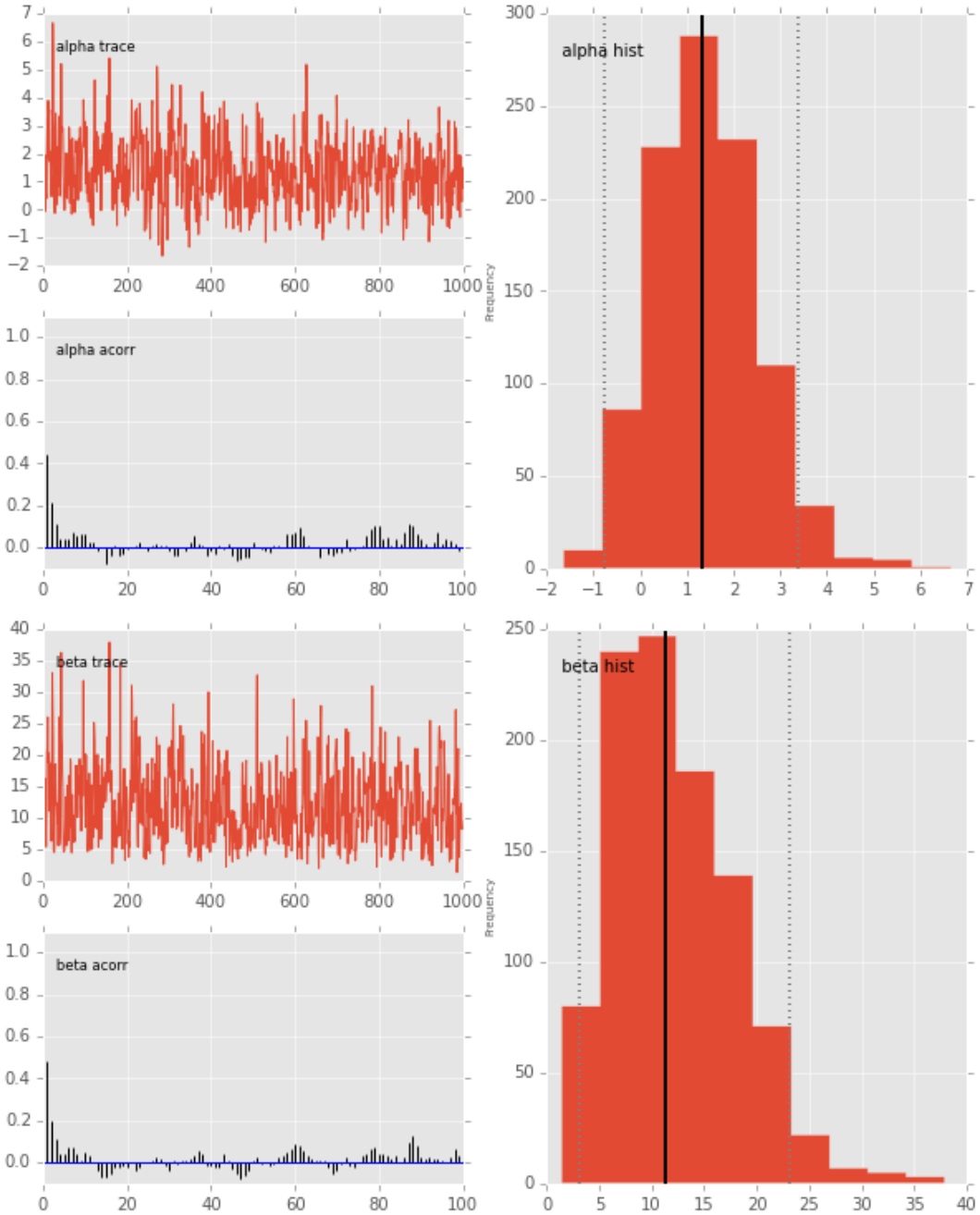
```
{'95% HPD interval': array([  3.1131,  23.0992]),
 'mc error': 0.2998,
 'mean': 12.1401,
 'n': 1000,
 'quantiles': {2.5000: 3.5785,
  25: 7.5365,
  50: 11.3823,
  75: 15.9492,
  97.5000: 25.4258},
 'standard deviation': 5.8260}
```

```
xp = np.linspace(-1, 1, 100)
a = alpha.stats()['mean']
b = beta.stats()['mean']
plt.plot(xp, invlogit(a + b*xp).value)
plt.scatter(x, y_obs/5, s=50);
plt.xlabel('Log does of drug')
plt.ylabel('Risk of death');
```




```
pymc.Matplot.plot(mc)
```

```
Plotting alpha
Plotting beta
```


alpha trace
alpha acorr
alpha hist
beta trace
beta acorr
beta hist
Frequency
Frequency

Using a hierarchcical model

This uses the Gelman radon data set and is based off this IPython notebook (http://twiecki.github.io/blog/2014/03/17/bayesian-glms-3/). Radon levels were measured in houses from all counties in several states. Here we want to know if the preence of a basement affects the level of radon, and if this is affected by which county the house is located in.

The data set provided is just for the state of Minnesota, which has 85 counties with 2 to 116 measurements per county. We only need 3 columns for this example `county`, `log_radon`, `floor`, where `floor=0` indicates that there is a basement.

We will perfrom simple linear regression on log_radon as a function of county and floor.

```
radon = pd.read_csv('radon.csv')[['county', 'floor', 'log_radon']]
radon.head()
```

	county	floor	log_radon
0	AITKIN	1	0.832909
1	AITKIN	0	0.832909
2	AITKIN	0	1.098612
3	AITKIN	0	0.095310
4	ANOKA	0	1.163151

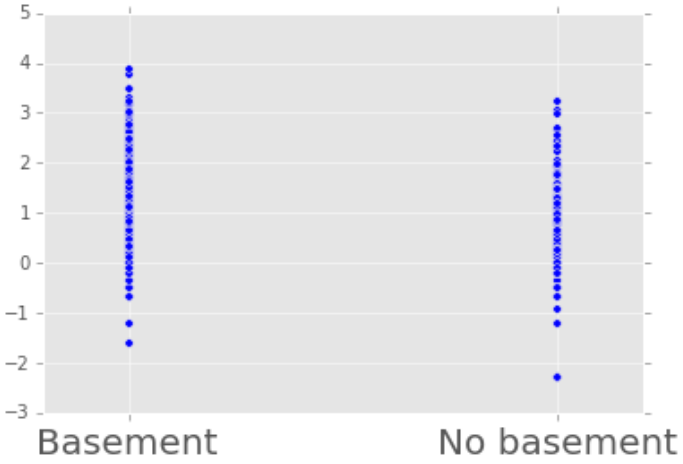
We will be creating lots of similar models, so it is worth wrapping definitions into a function to avoid repetition.

```
def make_model(x, y):
    # define priors
    a = pymc.Normal('slope', mu=0, tau=1.0/10**2)
    b = pymc.Normal('intercept', mu=0, tau=1.0/10**2)
    tau = pymc.Gamma("tau", alpha=0.1, beta=0.1)

    # define likelihood
    @pymc.deterministic
    def mu(a=a, b=b, x=x):
        return a*x + b

    y = pymc.Normal('y', mu=mu, tau=tau, value=y, observed=True)

    return locals()
```

Pooled model

If we pool the data across counties, this is the same as the simple linear regression model.

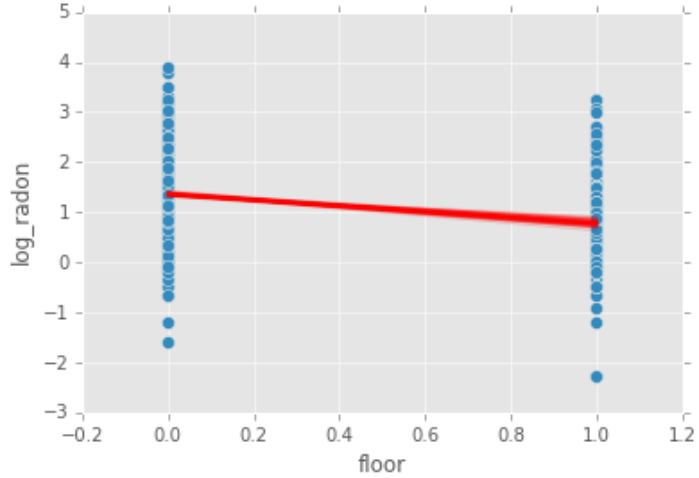
```
plt.scatter(radon.floor, radon.log_radon)
plt.xticks([0, 1], ['Basement', 'No basement'], fontsize=20);
```



```
m = pymc.Model(make_model(radon.floor, radon.log_radon))
mc = pymc.MCMC(m)
mc.sample(iter=1100, burn=1000)
```

```
[-----------------100%-----------------] 1100 of 1100 complete in 5.2 sec
```

```
abar = mc.stats()['slope']['mean']
bbar = mc.stats()['intercept']['mean']
radon.plot(x='floor', y='log_radon', kind='scatter', s=50);
xp = np.array([0, 1])
plt.plot(mc.trace('slope')()*xp[:, None] + mc.trace('intercept')(), c='red', al
plt.plot(xp, abar*xp + bbar, linewidth=2, c='red');
```

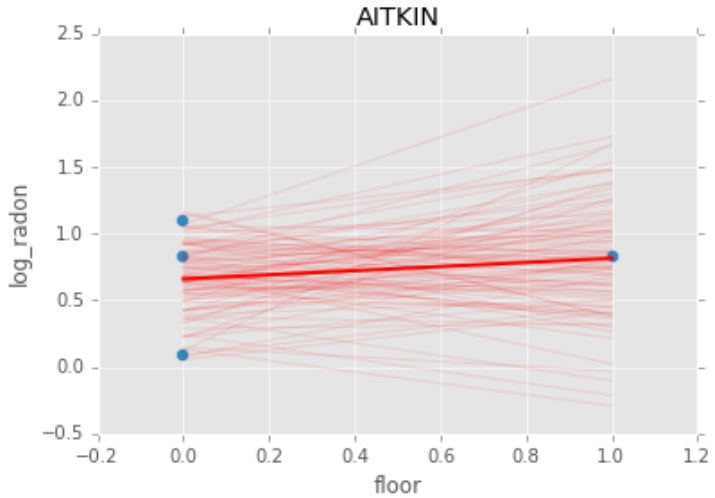


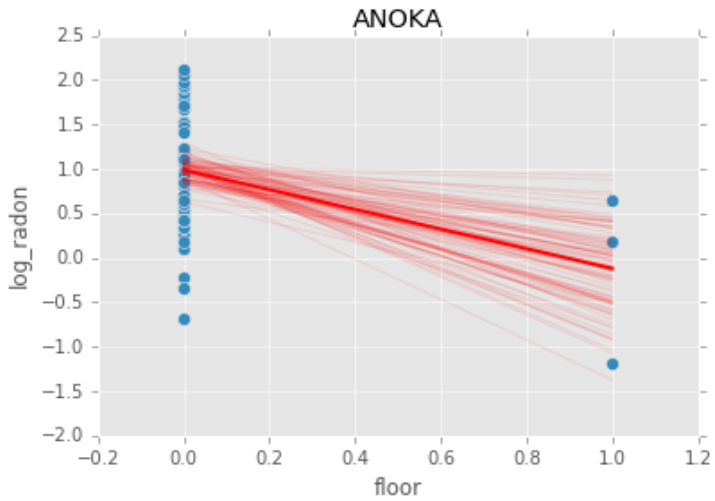
Individual couty model

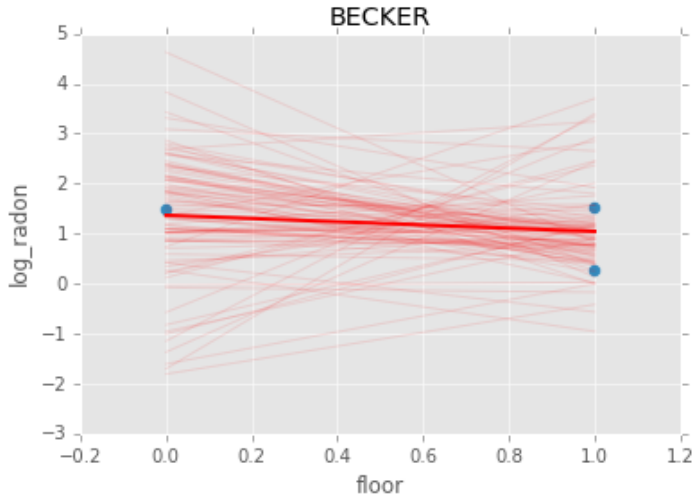
Inidividual couty models are done in the same way, except that we create a model for each county.

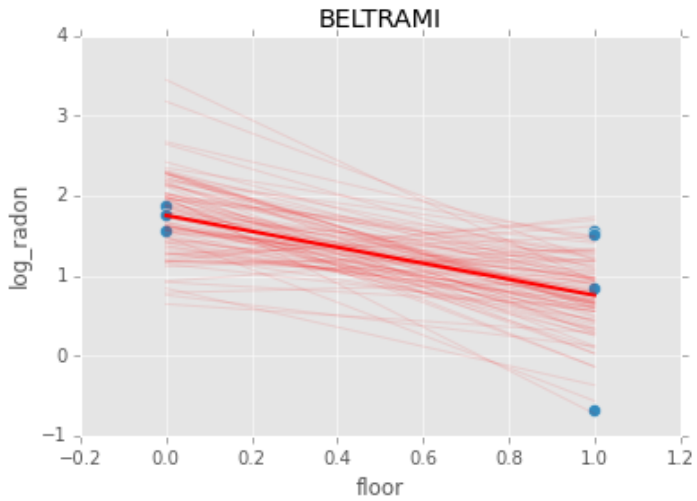
```
n = 0
i_as = []
i_bs = []
for i, group in radon.groupby('county'):

    m = pymc.Model(make_model(group.floor, group.log_radon))
    mc = pymc.MCMC(m)
    mc.sample(iter=1100, burn=1000)

    abar = mc.stats()['slope']['mean']
    bbar = mc.stats()['intercept']['mean']
    group.plot(x='floor', y='log_radon', kind='scatter', s=50);
    xp = np.array([0, 1])
    plt.plot(mc.trace('slope')()*xp[:, None] + mc.trace('intercept')(), c='red'
    plt.plot(xp, abar*xp + bbar, linewidth=2, c='red');
    plt.title(i)

    n += 1
    if n > 3:
        break
```

```
[-----------------100%-----------------] 1100 of 1100 complete in 3.0 sec
```


AITKIN
log_radon
floor
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


ANOKA
log_radon
floor
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2





Hiearchical model

With a hierarchical model, there is an a_c and a b_c for each county c just as in the individual couty model, but they are no longer indepnedent but assumed to come from a common group distribution

$$a_c \sim \mathcal{N}(\mu_a, \sigma_a^2)$$
$$b_c \sim \mathcal{N}(\mu_b, \sigma_b^2)$$

we furhter assume that the hyperparameters come from the following distributions