Probabilistic Programming

Marius Popescu

popescunmarius@gmail.com

2019 - 2020

PyMC Framework

PyMC is a python module that implements Bayesian statistical models and fitting algorithms

Two Types of Variables

- Bayesian *random variables* have not necessarily arisen from a physical random process. The Bayesian interpretation of probability is epistemic, meaning random variable Z's probability distribution $P(Z)$ represents our knowledge and uncertainty about Z's value

- Random variables are represented in PyMC by the classes `Stochastic` and `Deterministic`

Parent and Child Relationships

- *Parent variables* are variables that influence another variable
- *Child variables* are variables that are affected by other variables, i.e. are the subject of parent variables
- A variable can be both a parent and child
- a child can have more than one parent, and a parent can have many children

```
import pymc as pm


parameter = pm.Exponential("poisson_param", 1)
data_generator = pm.Poisson("data_generator", parameter)
data_plus_one = data_generator + 1
```

- `parameter` is a parent of `data_generator`
- `data_generator` is a child of `parameter`
- `data_generator` is a parent of `data_plus_one`
- `data_plus_one` is a child of `data_generator`

Parent and Child Relationships

A variable's children and parent variables can be accessed using the `children` and `parents` attributes attached to variables:

```
print("Children of `parameter`: "); print(parameter.children)
print("\nParents of `data_generator`: "); print(data_generator.parents)
print("\nChildren of `data_generator`: "); print(data_generator.children)
```

```
[Output]:
Children of `parameter`:
{<pymc.distributions.new_dist_class.<locals>.new_class 'data_generator' at 0x7f812404a8d0>}

Parents of `data_generator`:
{'mu': <pymc.distributions.new_dist_class.<locals>.new_class 'poisson_param' at 0x7f812404a898>}

Children of `data_generator`:
{<pymc.PyMCObjects.Deterministic '(data_generator_add_1)' at 0x7f812404a908>}
```

Parent and Child Relationships

PyMC variables have the following additional attributes:

- `extended_parents`:

  A set containing all the stochastic variables on which the variable depends either directly or via a sequence of deterministic variables. If the value of any of these variables changes, the variable will need to recompute its log-probability.

- `extended_children`:

  A set containing all the stochastic variables and potentials that depend on the variable either directly or via a sequence of deterministic variables. If the variable’s value changes, all of these variables will need to recompute their log-probabilities.

PyMC Variables

All PyMC variables have an attribute called `value` that stores the current (possibly random) internal value of that variable:

```
print("parameter.value =", parameter.value)
print("data_generator.value =", data_generator.value)
print("data_plus_one.value =", data_plus_one.value)
```

```
[Output]:
parameter.value = 0.032177775515776275
data_generator.value = 0
data_plus_one.value = 1
```

PyMC Stochastic Variables

PyMC Stochastic Variables

A stochastic variable can optionally be endowed with a method called `random`, which draws a value for the variable given the values of its parents:

```
Z = pm.DiscreteUniform("Z", lower = 1, upper = 6)
print(Z.value)
Z.random(); print(Z.value)
Z.random(); print(Z.value)
```

```
[Output]:
4
6
2
```

PyMC Stochastic Variables

The logarithm of a stochastic object's probability mass or density can be accessed via the `logp` attribute:

```
Z = pm.DiscreteUniform("Z", lower = 1, upper = 6); print(Z.logp, np.log(1./6.))
X = pm.Exponential("X", np.e, value=[1], observed=True); print(X.logp, 1-np.e)
X = pm.Exponential("X", np.e, value=[0], observed=True); print(X.logp)
```

```
[Output]:
-1.791759469228055 -1.791759469228055
-1.718281828459045 -1.718281828459045
1.0
```

PyMC Stochastic Variables

Stochastic objects have the following additional attributes:

- `observed:`

  A flag (boolean) indicating whether the variable's value has been observed (is fixed).

- `dtype`:

  A NumPy dtype object (such as numpy.int) that specifies the type of the variable's value to fitting methods. If this is None (default) then no type is enforced.

Creation of Stochastic Variables

There are three main ways to create stochastic variables, called the *automatic*, *decorator*, and *direct* interfaces

Creation of Stochastic Variables Automatic

Stochastic variables with standard distributions provided by PyMC

PyMC provides a large suite of built-in probability distributions

For each distribution, it provides:

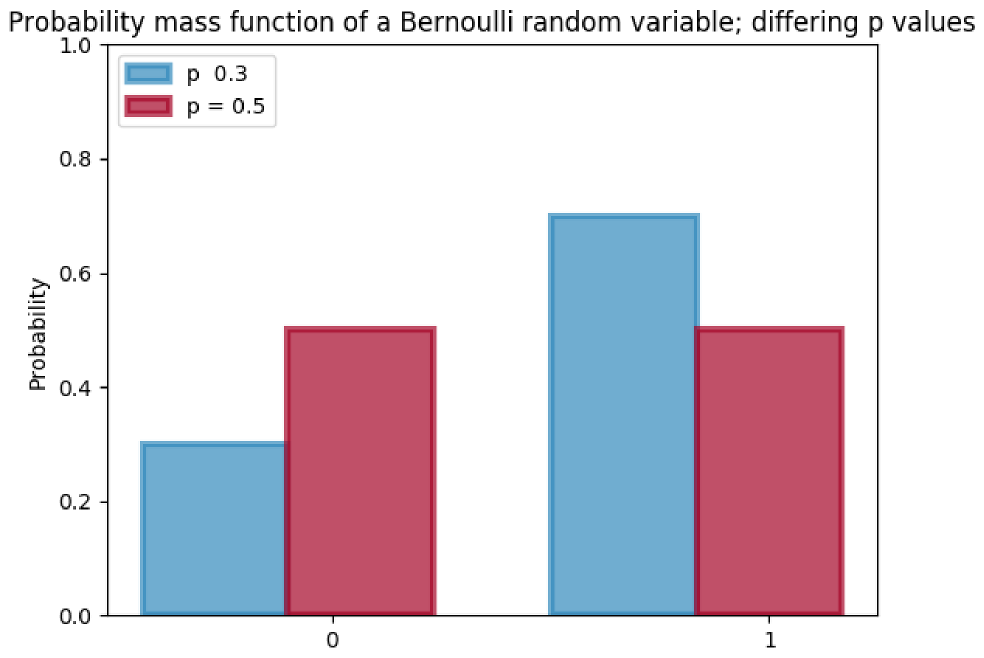
- A function that evaluates its log-probability or log-density: `normal_like()`
- A function that draws random variables: `rnormal()`
- A function that computes the expectation associated with the distribution: `normal_expval()`
- A Stochastic subclass generated from the distribution: `Normal`

Bernoulli Distribution

$$Z \sim \mathrm{Ber}(p)$$

$$P(Z = k) = p^k (1 - p)^{1-k}, \qquad k = 0,1$$

$$E(Z|p) = \sum_{k=0}^{1} kP(Z = k) = p$$



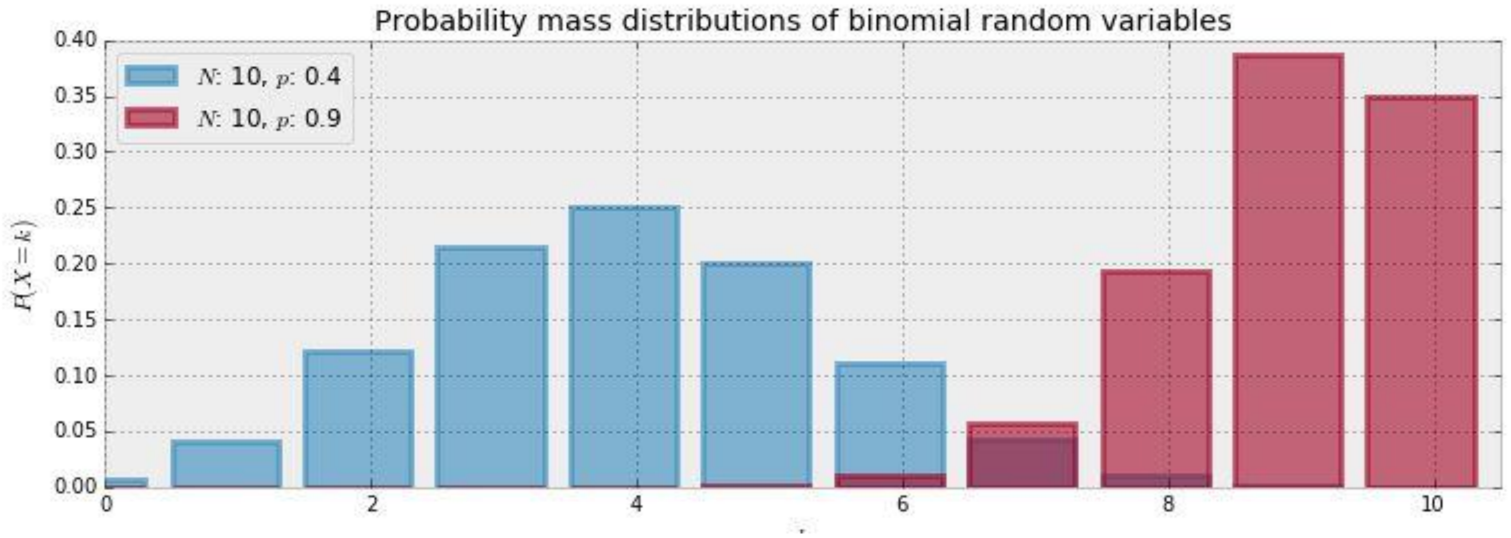
```
Z = pymc.Bernoulli(“Z”, p)
```

Binomial Distribution

$$Z \sim \text{Bin}(N, p)$$

$$P(Z = k) = \binom{N}{k} p^k (1-p)^{N-k}, \qquad k = 0,$$

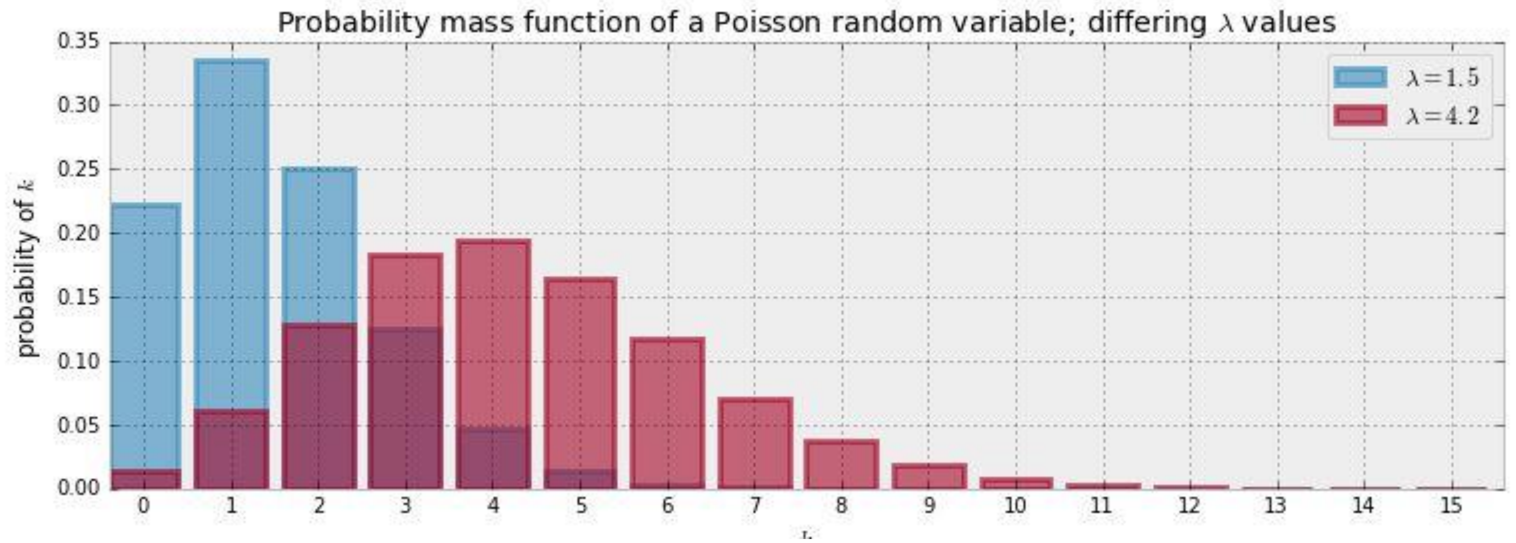
$$E(Z|N,p) = \sum_{k=0}^{1} kP(Z = k) = Np$$



Z is the number of events that occurred in the N trials, and p is the probability of a single event

```
Z = pymc.Binomial("Z", N, p)
```

Poisson Distribution



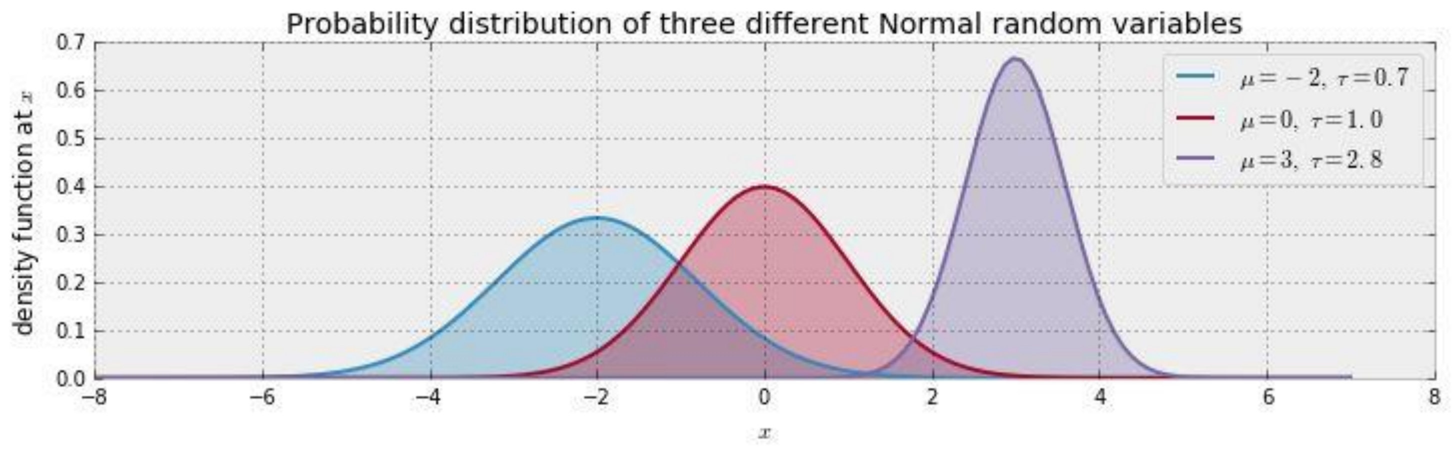
$$Z \sim \text{Poi}(\lambda)$$

$$P(Z=k) = \frac{\lambda^k e^{-\lambda}}{k!}, \qquad k = 0,1,2,\dots, \qquad \lambda \in \mathbb{R}_{>0}$$

$$E(Z|\lambda) = \sum_{k=0}^{\infty} kP(Z=k) = \lambda$$

```
Z=pymc.Poisson("Z", lambda_)
```

Normal Distribution



$$Z \sim N\left(\mu, \frac{1}{\tau}\right)$$

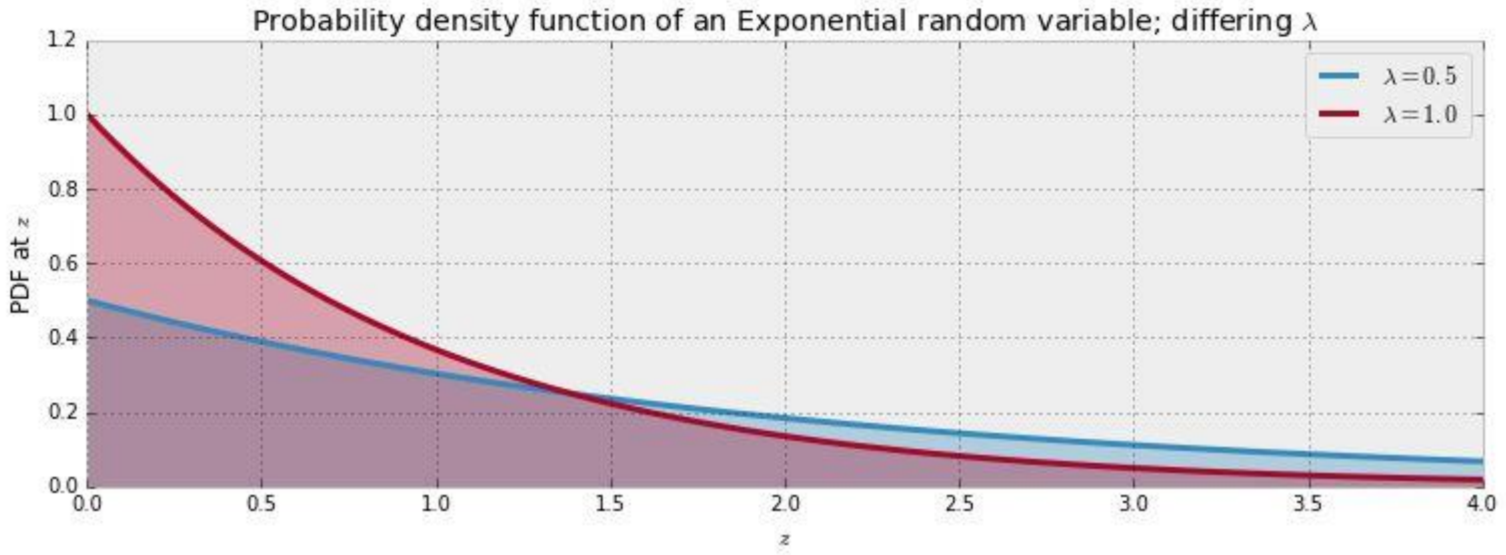
τ (precision of the distribution) corresponds to $\frac{1}{\sigma^2}$

$$f_Z(z|\mu,\tau) = \sqrt{\frac{\tau}{2\pi}}\, e^{-\frac{\tau}{2}(z-\mu)^2}, \qquad \tau > 0$$

$$E(Z|\mu,\tau) = \int_{-\infty}^{\infty} z f_Z(z|\mu,\tau)\,dz = \mu$$

```
Z=pymc.Normal(“Z”, mu, tau)
```

Exponential Distribution



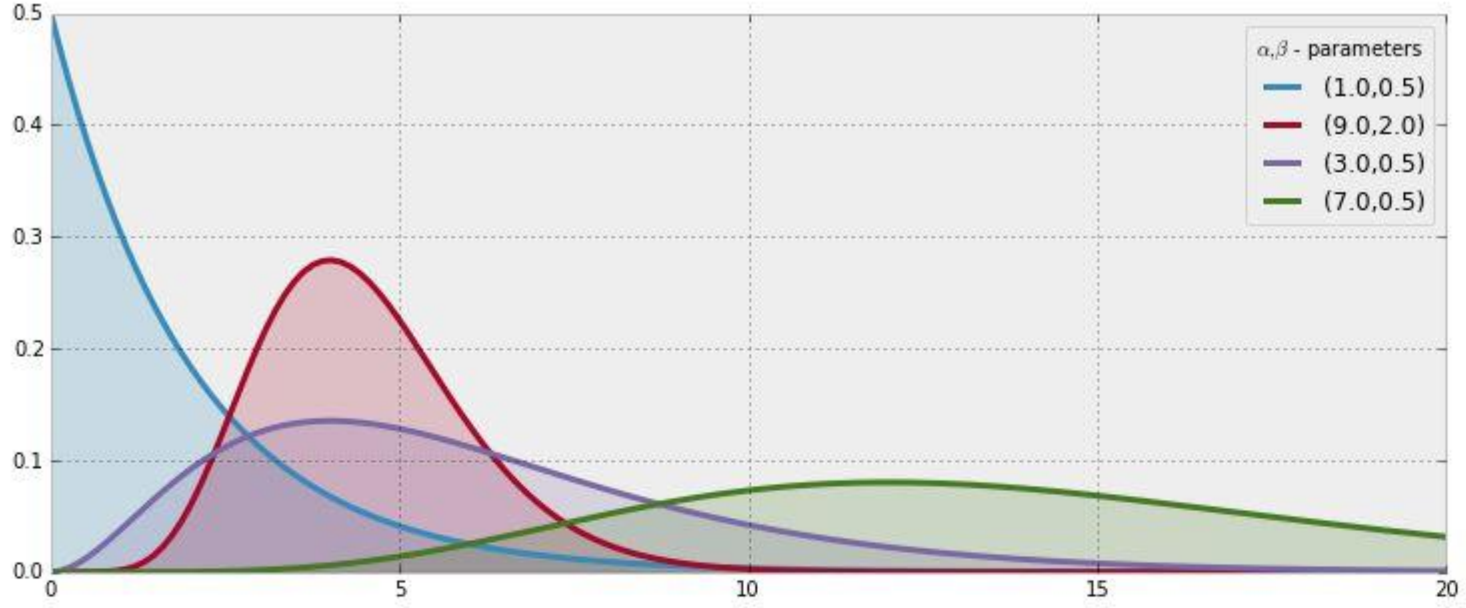
$$Z \sim \text{Exp}(\lambda)$$

$$f_Z(z|\lambda) = \lambda e^{-\lambda z}, \qquad z \geq 0$$

$$E(Z|\lambda) = \int_{-\infty}^{\infty} z f_Z(z|\lambda) dz = \frac{1}{\lambda}$$

```
Z=pymc.Exponential(“Z”, lambda_)
```

Gamma Distribution



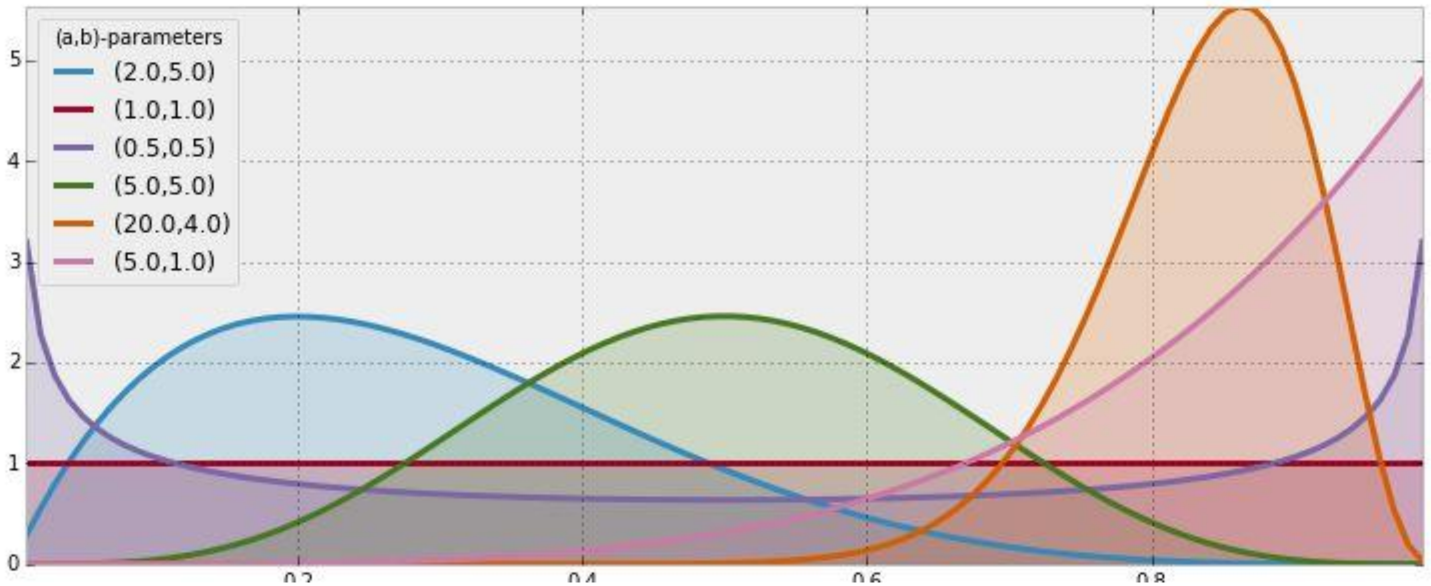
$$Z \sim \text{Gamma}(\alpha, \beta)$$

$$f_Z(z|\alpha,\beta) = \frac{\beta^\alpha z^{\alpha-1} e^{-\beta z}}{\Gamma(\alpha)}, \qquad \alpha > 0, \beta > 0, z \geq 0$$

$$E(Z|\alpha,\beta) = \int_{-\infty}^{\infty} z f_Z(z|\alpha,\beta) dz = \frac{\alpha}{\beta}$$

```
Z=pymc.Gamma(“Z”, alpha, beta)
```

Beta Distribution



$$Z \sim \text{Beta}(\alpha, \beta)$$

$$f_Z(z|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} z^{\alpha-1}(1-z)^{\beta-1},\ \alpha > 0, \beta > 0, 0 < z < 1$$

$$E(Z|\alpha,\beta) = \int_{-\infty}^{\infty} z f_Z(z|\alpha,\beta)dz = \frac{\alpha}{\alpha+\beta}$$

```
Z=pymc.Beta(“Z”, alpha, beta)
```

Multivariate Normal Distribution

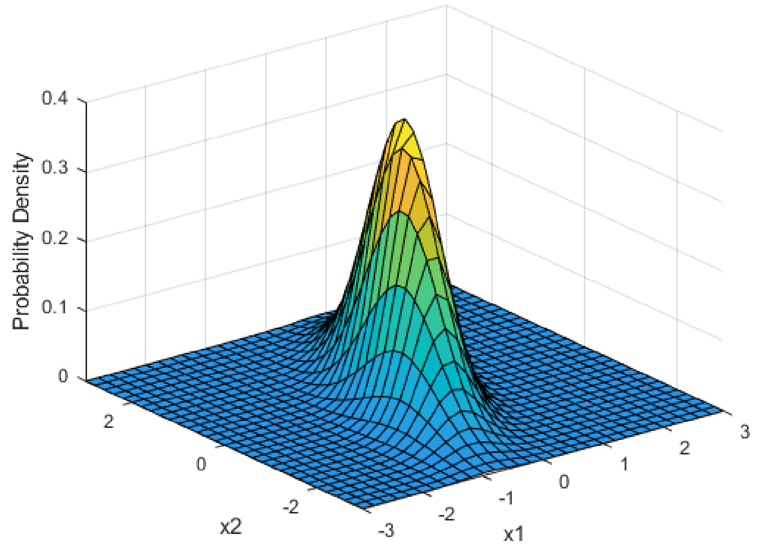
$$Z \sim N(\mu, \mathrm{T})$$

T (precision of the distribution) is the inverse of the covariance matrix Σ

$$f_Z(z|\mu,\tau) = \sqrt{\frac{|\mathrm{T}|}{2\pi}}\, e^{-\frac{1}{2}(z-\mu)'\mathrm{T}(z-\mu)}$$

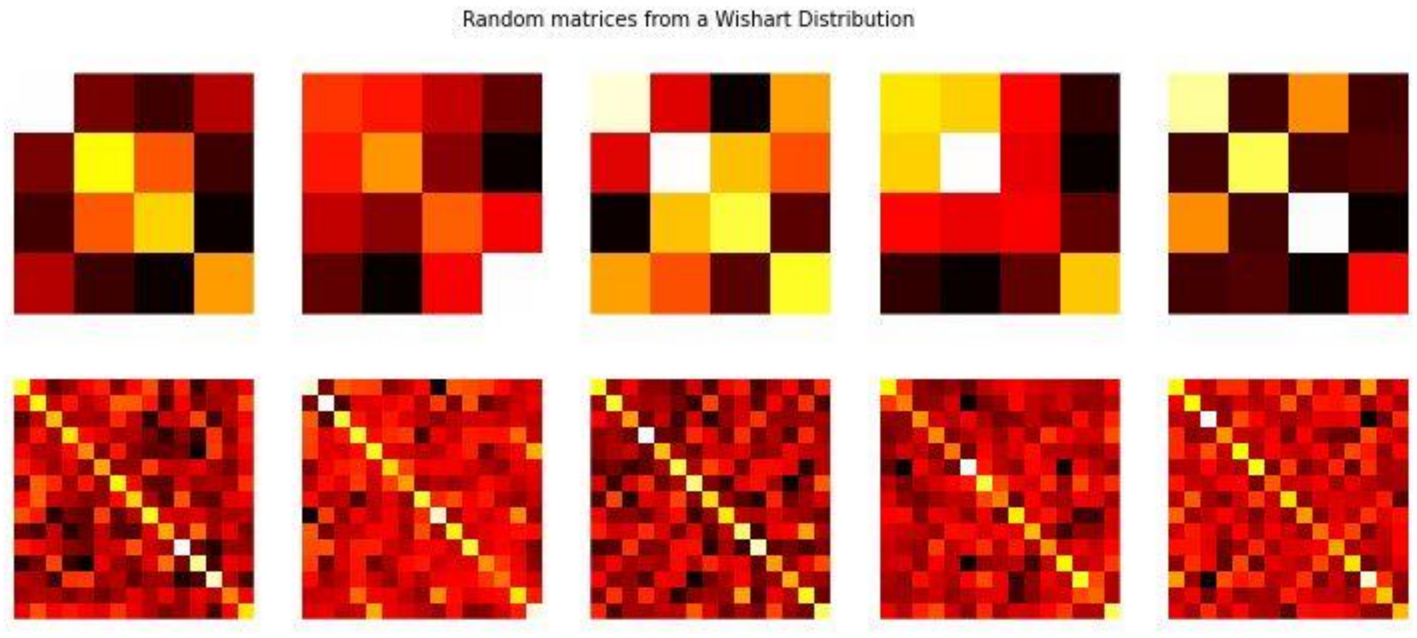
```
Z = pm.MvNormal("Z", np.array([2.,4., 6.]), np.identity(3))
Z.random(); print(Z.value)
```

```
[Output]:

[ 2.66051582  4.46602181  6.6511556 ]
```



Wishart Distribution

- Wishart distribution is a distribution over all positive semi-definite matrices
- Covariance matrices are positive-definite, hence the Wishart is an appropriate prior for covariance matrices



There Are More

https://pymc-devs.github.io/pymc/distributions.html

Creation of Stochastic Variables
Stochastic Decorator

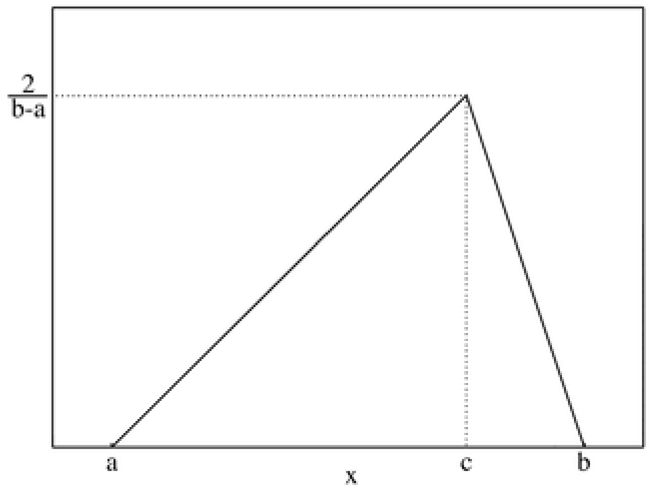
@pymc.stochastic

- A python function preceded by `@pymc.stochastic` decorator will create a `pymc.Stochastic` object, which takes its name from the function it decorates, and has all the familiar methods and attributes
- The `Stochastic` object produced by the `@stochastic` decorator will evaluate its log-probability using the function. The value argument, which is required, provides an initial value for the variable. The remaining arguments will be assigned as parents
- The `value` and parents of stochastic variables may be any objects, provided the log-probability function returns a real number (`float`)
- The decorator `stochastic` can take any of the arguments `Stochastic.__init__` takes except `parents`, `logp`, `random`, `doc` and `value`. These arguments include `trace`, `plot`, `verbose`, `dtype`, `rseed` and `name`
- The decorator interface allows to specify a `random` method for sampling the stochastic variable's value conditional on its parents

Example: Triangular Distribution

- The *triangular distribution* is a continuous probability distribution with lower limit a, upper limit b and mode c, where a < b and a ≤ c ≤ b
- The probability density function:

$$f(z) = \begin{cases} 0 & \text{for } z < a \\ \dfrac{2(z-a)}{(b-a)(c-a)} & \text{for } a \le z < c \\ \dfrac{2}{b-a} & \text{for } z = c \\ \dfrac{2(b-z)}{(b-a)(b-c)} & \text{for } c < z \le b \\ 0 & \text{for } b < z \end{cases}$$



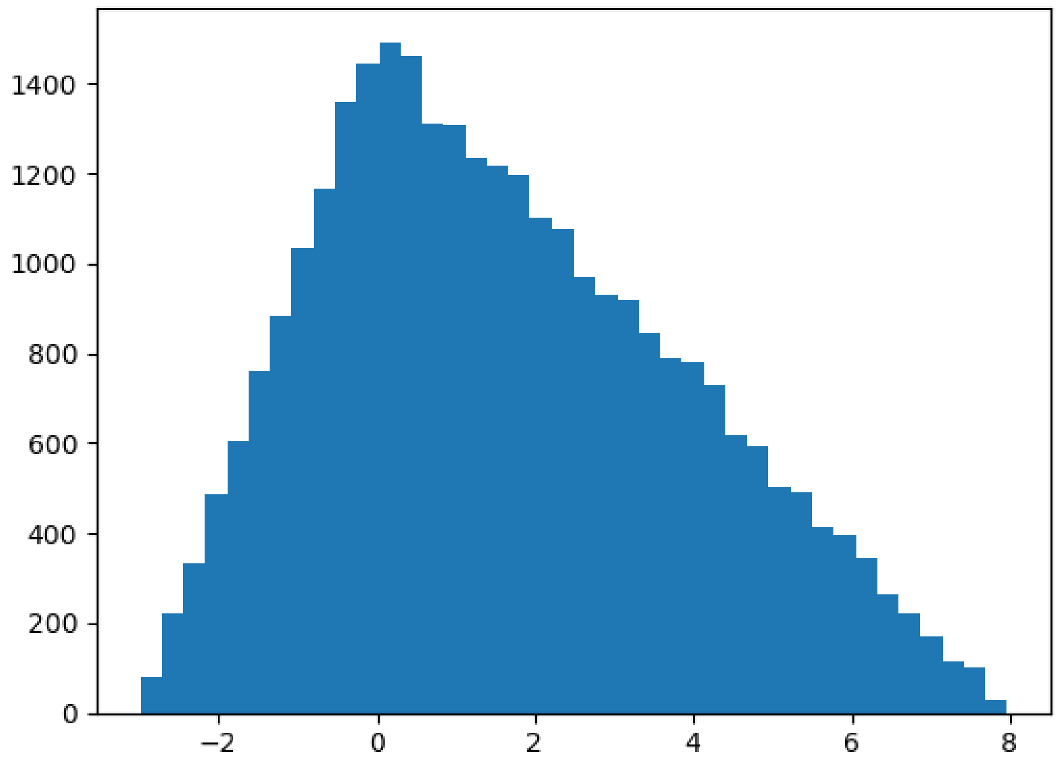
- The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists

Example: Triangular Distribution

```
@pm.stochastic(dtype=float)
def Z(value = c, a = a, b = b, c = c):

    def logp(value, a, b, c):
        if a <= value < c:
            return np.log((2. * (value - a)) / ((b - a) * (c - a)))
        if value == c:
            return np.log(2. / (b - a))
        if c < value <= b:
            return np.log((2. * (b - value)) / ((b - a) * (b - c)))
        return -np.inf

    def random(a, b, c):
        return np.random.triangular(a, c, b)
```

Example: Triangular Distribution

```
a = -3
b = 8
c = 0
.......................
model = pm.Model([Z])

mcmc = pm.MCMC(model)
mcmc.sample(40000, 10000, 1)
Z_samples = mcmc.trace('Z')[:]

plt.hist(Z_samples, bins = 40)
plt.show()
```



Creation of Stochastic Variables Direct

It’s possible to instantiate `pymc.Stochastic` directly

```
def triangular_logp(value, a, b, c):
    if a <= value < c:
        return np.log((2. * (value - a)) / ((b - a) *
(c - a)))
    if value == c:
        return np.log(2. / (b - a))
    if c < value <= b:
        return np.log((2. * (b - value)) / ((b - a) *
(b - c)))
    return -np.inf

def triangular_random(a, b, c):
    return np.random.triangular(a, c, b)
```

```
Z = pm.Stochastic( logp = triangular_logp,
                   doc = 'Triangular Distribution',
                   name = 'Z',
                   parents = {'a': -3, 'b': 8, 'c': 0},
                   random = triangular_random,
                   trace = True,
                   value = 0,
                   dtype = float,
                   rseed = 1.,
                   observed = False,
                   cache_depth = 2,
                   plot=True,
                   verbose = 0)
```

A Warning: Don't update stochastic variables' values in-place

- `Stochastic` objects' values should not be updated in-place. This confuses PyMC's caching scheme and corrupts the process used for accepting or rejecting proposed values in the MCMC algorithm
- The only way a stochastic variable's value should be updated is using statements of the following form:

```
A.value = new_value
```

- The following are in-place updates and should **never** be used:

```
A.value += 3

A.value[2] = 5
```

PyMC Deterministic Variables

The Deterministic Class

- The `Deterministic` class represents variables whose values are completely determined by the values of their parents
- A deterministic variable’s most important attribute is `value`, this attribute is computed on-demand and cached for efficiency
- Deterministic variables have no methods

Creation of Deterministic Variables

There are three main ways to create deterministic variables, called the *automatic*, *decorator*, and *direct* interfaces

Creation of Deterministic Variables Automatic

A handful of common functions have been wrapped in Deterministic objects

LinearCombination

Has two parents x and y, both of which must be iterable (i.e. vector-valued). This function returns:

$$\sum_i x_i' y_i$$

```
X = pm.DiscreteUniform("X", lower = 1, upper = 6, size = 5); print(X.value)
Y = pm.DiscreteUniform("Y", lower = 1, upper = 6, size = 5); print(Y.value)
Z = pm.LinearCombination("Z", X.value, Y.value); print(Z.value)
```

Index

Has two parents x and index. x must be iterables, index must be valued as an integer

```
X = pm.DiscreteUniform("X", lower = 1, upper = 6, size = 5); print(X.value)
Y = pm.DiscreteUniform("Y", lower = 0, upper = 5); print(Y.value)
Z = pm.Index("Z", X.value, Y.value); print(Z.value)
```

`Index` is useful for implementing dynamic models, in which the parent-child connections change

Lambda

Converts an anonymous function (in Python, called *lambda functions*) to a `Deterministic` instance on a single line

```
beta = pm.Normal("coefficients", 0, size=(N, 1))

x = np.random.randn((N, 1))

linear_combination = pm.Lambda(lambda x=x, beta=beta: np.dot(x.T, beta))
```

`CompletedDirichlet`:

PyMC represents Dirichlet variables of length *k* by the first *k*−1 elements; since they must sum to 1, the k^{th} element is determined by the others. `CompletedDirichlet` appends the k^{th} element to the value of its parent *D*

Logit, InvLogit, StukelLogit, StukelInvLogit

Common link functions for generalized linear models, and their inverses

Elementary Operations on Variables

elementary operations, like addition, exponentials etc. implicitly create deterministic variables

```
>>> x = pymc.MvNormalCov('x',np.ones(3),np.eye(3))
>>> y = pymc.MvNormalCov('y',np.ones(3),np.eye(3))
>>> print x+y
<pymc.PyMCObjects.Deterministic '(x_add_y)' at 0x105c3bd10>
>>> print x[0]
<pymc.CommonDeterministics.Index 'x[0]' at 0x105c52390>
>>> print x[1]+y[2]
<pymc.PyMCObjects.Deterministic '(x[1]_add_y[2])' at 0x105c52410>
```

All the objects thus created have `trace=False` and `plot=False` by default

Creation of Deterministic Variables
Deterministic Decorator

@pymc.deterministic

```
@pm.deterministic
def some_deterministic_var(v1=v1,):
     #jelly goes here.
```

- For all purposes, we can treat the object `some_deterministic_var` as a variable and not a Python function
- Notice that rather than returning the log-probability, as is the case for `Stochastic` objects, the function returns the value of the deterministic object, given its parents

Creation of Deterministic Variables Direct

Deterministic objects can also be instantiated directly

```
def lambda_eval(tau=tau, lambda_1=lambda_1, lambda_2=lambda_2):
    out = np.zeros(n_count_data)
    out[:tau] = lambda_1  # lambda before tau is lambda1
    out[tau:] = lambda_2  # lambda after (and including) tau is lambda2
    return out

Lambda_ = pymc.Deterministic(eval = lamnda_eval,
                  name = ‘lambda_',
                  parents = {‘tau’: tau, ‘lambda_1’: lambda_1, ‘lambda_2': lambda_2},
                  doc = 'The lambda before and after tau',
                  trace = True,
                  verbose = 0,
                  dtype=float,
                  plot=False,
                  cache_depth = 2)
```

Containers

In some situations it would be inconvenient to assign a unique label to each parent of some variable. Consider *y* in the following model:

$$\begin{aligned} x_0 &\sim N(0, \tau_x) \\ x_{i+1}|x_i &\sim \mathrm{N}(x_i, \tau_x) \\ i &= 0, \ldots, N-2 \\ y|x &\sim N\left(\sum_{i=0}^{N-1} x_i^2, \tau_y\right) \end{aligned}$$

Here, y depends on every element of the Markov chain *x*, but we wouldn't want to manually enter *N* parent labels `'x_0'`, `'x_1'`, etc.

Containers

This situation can be handled naturally in PyMC:

```
N = 10
x_0 = pymc.Normal('x_0', mu=0, tau=1)

x = np.empty(N, dtype=object)
x[0] = x_0

for i in range(1, N):
    x[i] = pymc.Normal('x_%i' % i, mu=x[i-1], tau=1)

@pymc.observed
def y(value=1, mu=x, tau=100):
    return pymc.normal_like(value, np.sum(mu**2), tau)
```

Containers

- Containers, like variables, have an attribute called `value`. This attribute returns a copy of the iterable that was passed into the container function, but with each variable inside replaced with its corresponding value
- Containers can be constructed from lists, tuples, dictionaries, Numpy arrays, modules, sets or any object with a `__dict__` attribute
- Containers have the following useful attributes in addition to `value`:
  - `variables`
  - `stochastics`
  - `potentials`
  - `deterministics`
  - `data_stochastics`
  - `step_methods`

  Each of these attributes is a set containing all the objects of each type in a container, and within any containers in the container.

Not covered yet

- The `Potential` class
- The `Model` class
- objects that fit models: `MCM`, `MAP`, `NormApprox`
- The `Sampler` class: Metropolis, `AdaptativeMetropolis`, `DiscreteMetropolis`, `BinaryMetropolis`, `Slicer`, `Gibbs`

Example: Cheating among students

Privacy Algorithm

In the interview process for each student, the student flips a coin, hidden from the interviewer. The student agrees to answer honestly if the coin comes up heads. Otherwise, if the coin comes up tails, the student (secretly) flips the coin again, and answers "Yes, I did cheat" if the coin flip lands heads, and "No, I did not cheat", if the coin flip lands tails. This way, the interviewer does not know if a "Yes" was the result of a guilty plea, or a Heads on a second coin toss. Thus privacy is preserved and the researchers receive honest answers.

Warner, S. L. (March 1965). "Randomised response: a survey technique for eliminating evasive answer bias". Journal of the American Statistical Association.

Using PyMC to dig through this noisy model

- Suppose 100 students are being surveyed for cheating, and we wish to find p, the proportion of cheaters
- Since we are quite ignorant about p, we will assign it a Uniform(0,1) prior

```
N = 100
p = pm.Uniform("freq_cheating", 0, 1)
```

Using PyMC to dig through this noisy model

- We assign Bernoulli random variables to the 100 students: 1 implies they cheated and 0 implies they did not

```
true_answers = pm.Bernoulli("truths", p, size=N)
```

- The first coin-flip can be modeled by sampling 100 Bernoulli random variables with $p = \frac{1}{2}$: denote a 1 as a Heads and 0 a Tails

```
first_coin_flips = pm.Bernoulli("first_flips", 0.5, size=N)
```

- Although not everyone flips a second time, we can still model the possible realization of second coin-flips:

```
second_coin_flips = pm.Bernoulli("second_flips", 0.5, size=N)
```

Using PyMC to dig through this noisy model

- Using previous variables, we can return a possible realization of the observed proportion of "Yes" responses

```
@pm.deterministic
def observed_proportion(t_a=true_answers, fc=first_coin_flips, sc=second_coin_flips):

    observed = fc * t_a + (1 - fc) * sc
    return observed.sum() / float(N)
```

Using PyMC to dig through this noisy model

- Suppose that after performing our coin-flipped interviews the researchers received 35 "Yes" responses. The researchers observe a Binomial random variable, with `N = 100` and `p = observed_proportion` with `value = 35`:

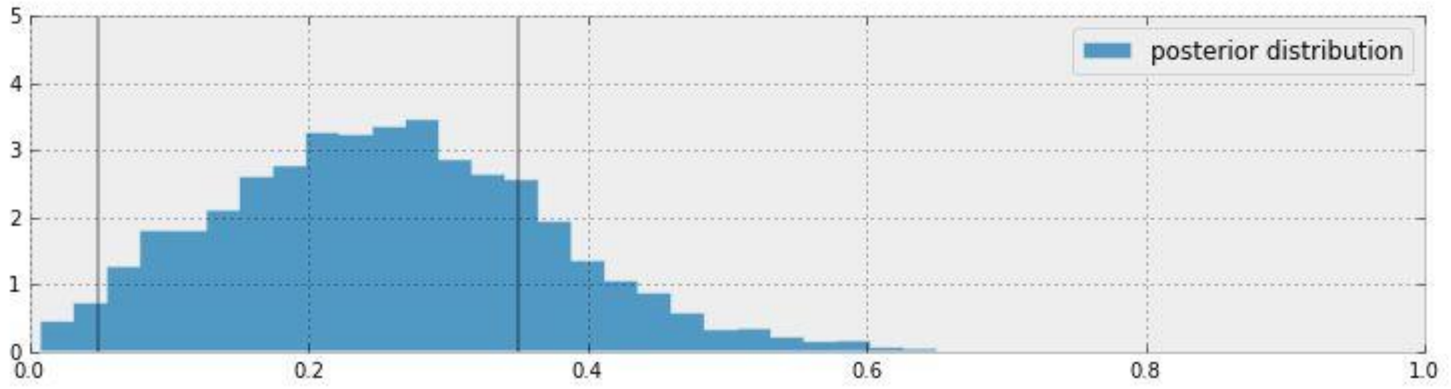
```
X = 35

observations = pm.Binomial("obs", N, observed_proportion, observed=True, value=X)
```

Using PyMC to dig through this noisy model

- The model:

```
model = pm.Model([p, true_answers, first_coin_flips, second_coin_flips, observed_proportion, observations])

mcmc = pm.MCMC(model)
mcmc.sample(40000, 15000)

p_trace = mcmc.trace("freq_cheating")[:]
```

Posterior Distribution



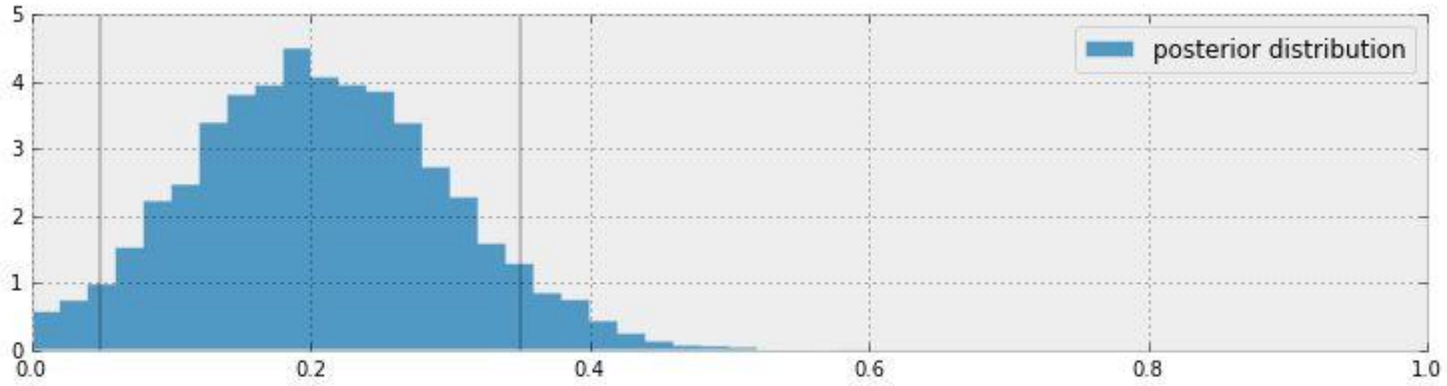
Alternative Simplified Model

Given a value for p we can find the probability the student will answer yes:

$$P(\text{Yes}) = P(\text{Heads on first coin})P(\text{cheater}) + P(\text{Tails on first coin})\ P(\text{Heads on second coin}) = \frac{1}{2}p + \frac{1}{2}\frac{1}{2} = \frac{p}{2} + \frac{1}{4}$$

```
p = pm.Uniform("freq_cheating", 0, 1)
p_skewed = 0.5 * p + 0.25
yes_responses = pm.Binomial("number_cheaters", 100, p_skewed, value=35, observed=True)

model = pm.Model([yes_responses, p_skewed, p])

mcmc = pm.MCMC(model)
mcmc.sample(25000, 2500)
p_trace = mcmc.trace("freq_cheating")[:]
```

Posterior Distribution


posterior distribution