

distributions

May 2, 2017

1 Some Probability Distributions

This notebook introduces the **scipy.stats** package through a review of common probability distributions.

1.1 1 Introduction

1.1.1 1.1 Preparation

You should already have Python 3 installed, hopefully through the **conda** utility. But you may not have installed the **scipy** module. It's easy enough to check. If you have **scipy** installed, you can list its details in the following way.

```
workshops/distributions/python$ conda list scipy
# packages in environment at /Users/pglezen/apps/miniconda3:
#
scipy                0.17.0                np111py35_4
```

If instead of **scipy** you see a blank line, you can install it with the following **conda** command.

```
conda install scipy
```

If you already have it installed, you may still wish to update it to the latest version.

```
conda update scipy
```

Of course, this only applies to the current conda execution environment. If you don't know what that is, then you probably only have one, so you're probably fine.

Once you have the package installed, you should bookmark the documentation in your web browser for easy reference. The **scipy** project has many nice packages. The API for the **stats** package is available at

<https://docs.scipy.org/doc/scipy/reference/stats.html>

The SciPy homepage is <https://scipy.org/>.

1.1.2 1.2 Conventions

SciPy has consistent naming conventions for methods on probability distribution objects.

- **pmf** - probability mass function for discrete distributions
- **cdf** - cumulative distribution
- **ppf** - percentage point function (inverse of CDF)
- **rvs** - random variate sampling

1.1.3 Import

The top level package name for the SciPy statistical objects is `scipy.stats`. To make working with individual objects more convenient, we'll import them directly into our namespace rather than referencing them through their full package references. So instead of

```
import scipy.stats.binom
```

to reference binomial distribution functions through `scipy.stats.binom`, we'll use

```
In [1]: from scipy.stats import binom
        from matplotlib import pyplot as plt
        import numpy as np

        %matplotlib inline
```

This makes the `binom` object referenceable directly. The

```
from <this package> import <that object>
```

style of import will be used throughout this workshop to reference Python distribution objects.

1.1.4 Frozen Parameters

Most probability distributions require certain parameters to be completely specified. For example, the binomial distribution requires two parameters:

- `n` - the number of bernoulli trials
- `p` - the probability of success for each trial

These two parameters must be specified for each invocation of a `binom` object, along with the function arguments. If the distribution parameters are expected to be constant through out a session, one can freeze the parameter values.

```
b = binom(100, 0.2)
```

You can then invoke all the `binom` methods on `myRV` without specifying the `n` and `p` parameters.

1.2 2 Discrete Distributions

The following sections introduce SciPy objects working with discrete distributions. In addition to methods particular to each distribution, the [rv_discrete](#) page documents methods common to all discrete distributions.

1.2.1 2.1 Binomial

The binomial distribution is handled by the `binom` object. Let's create an instance **frozen** at 100 trials with probability of 0.2 success rate. We'll invoke our distribution methods on this instance.

```
In [2]: b = binom(100, 0.2)
```

With `b` as our binomial instance frozen at 100 trials with 0.2 chance of succeeding each time, let's start with the probability mass function, or `pmf`. This is equation (7) of the math supplement.

$$f_X(i) = \sum_{i=0}^n \binom{n}{i} p^i (1-p)^{n-i}$$

Here we consider the case of $n = 100$ and $p = 0.2$. We suspect that the highest probability will be around 20 successes since the probability of each success is 0.2 and there are 100 trials. The `pmf` function accepts numeric arrays, so we'll check 10, 20, and 30 in one go.

```
In [3]: b.pmf([10, 20, 30])
```

```
Out[3]: array([ 0.00336282,  0.09930021,  0.00518964])
```

Sure enough, the PMF function shows a 9.9% chance of obtaining exactly 20 successes. The probabilities for 10 and 30 are each less than 1%.

Now let's check the associated cumulative distribution function, or CDF. The CDF at a point is the cumulation of probability from the PMF for all points equal or less. Since we expect the average to be around 20, the CDF should be close to 0.5.

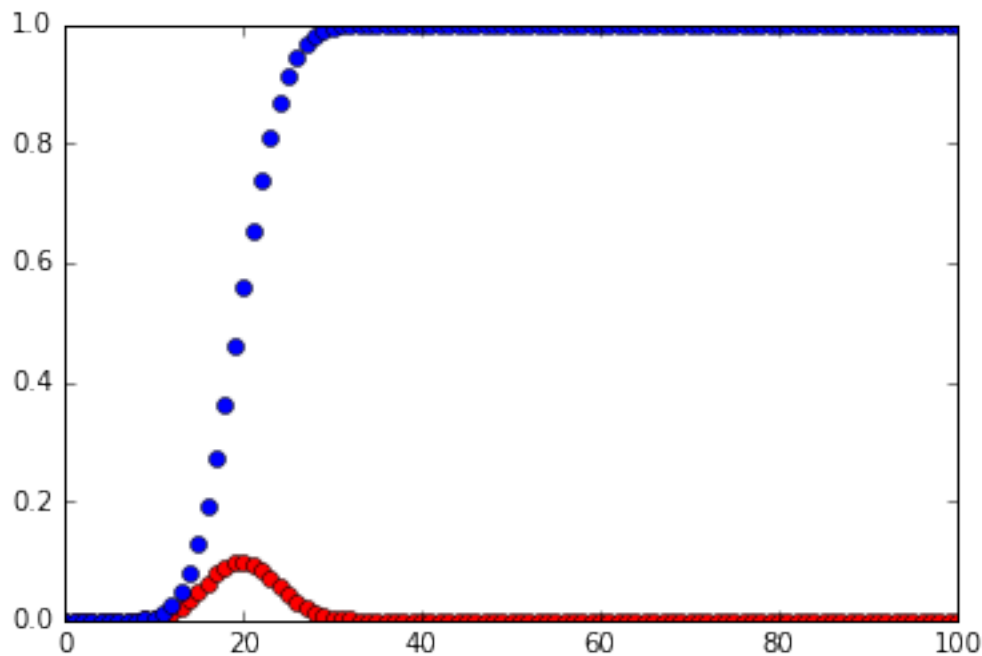
```
In [4]: b.cdf([10, 20, 30])
```

```
Out[4]: array([ 0.00569638,  0.55946158,  0.99394066])
```

This shows that the probability of having less than or equal to 10 successes is 0.57%. The probability of having less than or equal to 20 successes is 56%. The probability of having less than or equal to 30 successes is 99.4%.

To better see the relationship between the PMF and CDF, we can plot them.

```
In [5]: fig, ax = plt.subplots(1,1)
        x = np.arange(101)
        ax.plot(x, b.pmf(x), 'ro')
        ax.plot(x, b.cdf(x), 'bo')
        plt.show()
```



The PMF is in red. The CDF is in blue. The blue value at each point is the sum of all the red points at or before it. Let's verify this for the probability of 22 or fewer successes. The `np.arange` function is used which returns a sequence of integers from 0 to one less than the argument.

```
In [6]: b.pmf(np.arange(23)).sum()
```

```
Out[6]: 0.73893275233089206
```

```
In [7]: b.cdf(22)
```

```
Out[7]: 0.73893275233090572
```

With the `pmf`, we summed all the values of the PMF from 0 to 22. But this is precisely what the CDF does.

Quantiles are the inverse of the CDF function. For any probability, a quantile function tells us the value of the random variable for which the CDF is equal to that probability. The SciPy stats function for this is `ppf`, or percentage point function.

How many successes can we expect to happen 25% of the time?

```
In [8]: b.ppf(0.25)
```

```
Out[8]: 17.0
```

This means that 25% of the time, the number of successes will be 17 or less. We can pass in arrays just like we did for `pmf`.

```
In [9]: b.ppf([0.25, 0.5, 0.75])
```

```
Out[9]: array([ 17.,  20.,  23.])
```

The reason these come out exact integers is the binomial random variables only take integer values. The `ppf` can be interpreted as the smallest integer number of successes with probability greater than or equal to the argument. So it's not a strict inverse.

```
In [10]: b.cdf([16, 17, 19, 20, 22, 23])
```

```
Out[10]: array([ 0.19233758,  0.27118896,  0.46016137,  0.55946158,  0.73893275,
                0.81091279])
```

Sometimes it's useful to generate random samples from a distribution. In other words, the numbers are generated as if they came from random variable values associated with the outcomes of the experiment. The `rvs` method is used for this.

```
In [11]: b_sample = b.rvs(size=1000)
         b_sample[0:100]
```

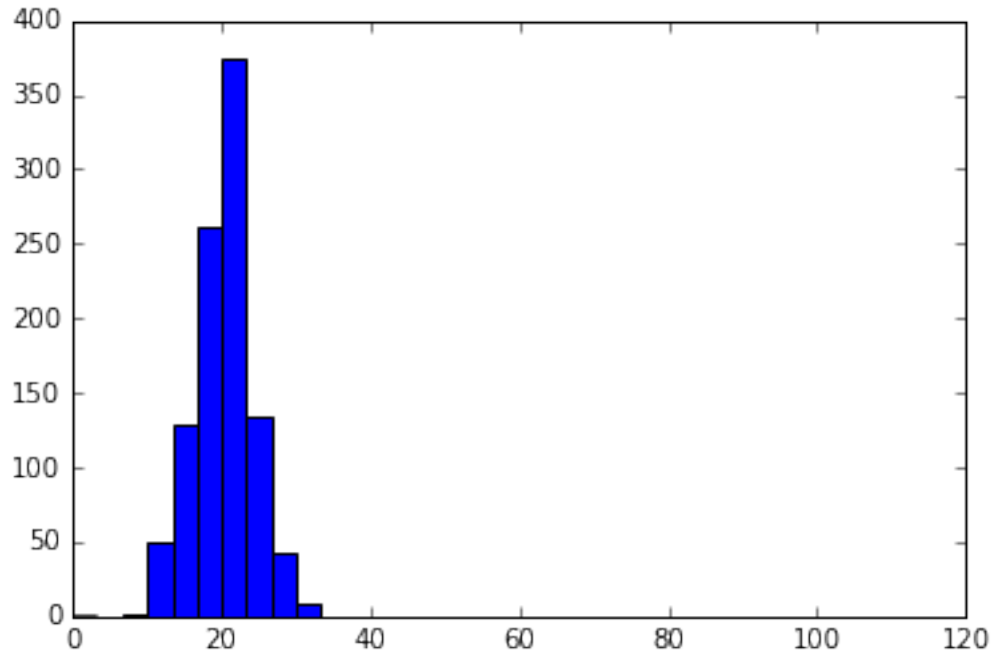
```
Out[11]: array([16, 26, 13, 17, 18, 16, 20, 14, 15, 19, 15, 13, 16, 15, 22, 21, 19,
                27, 23, 20, 13, 19, 20, 21, 17, 17, 23, 17, 17, 17, 22, 16, 23, 24,
                18, 22, 17, 21, 10, 22, 17, 29, 21, 16, 26, 18, 21, 19, 22, 20, 20,
                16, 20, 19, 26, 18, 22, 21, 24, 11, 25, 20, 21, 16, 32, 23, 22, 17,
                15, 24, 18, 25, 20, 17, 12, 24, 23, 19, 20, 19, 26, 25, 28, 18, 17,
                19, 21, 18, 25, 20, 16, 22, 17, 18, 16, 17, 16, 28, 19, 18])
```

This simulates the arduous task of

1. flipping an unfair coin (with 20% chance of heads) 100 times,
2. noting the number of heads,
3. writing it down,
4. repeat steps 1-3 for 1,000 times

It's fun to plot a histogram of our sample. It should have a shape close to our PMF.

```
In [12]: fig, ax = plt.subplots()
         ax.hist(b_sample, bins=30, range=(0,100))
         plt.show()
```



1.3 2.2 Poisson

The Poisson random variable represents the number of occurrences of an event over a fixed interval of time. An example could be the number of new cases opened per day. The parameter of the random variable, usually denoted by λ , turns out to be the expected number of occurrences. The Poisson density function is given by

$$f_X(i; \lambda) = e^{-\lambda} \frac{\lambda^i}{i!}$$

For each i , this represents the probability of observing i counts within the unit time period. The SciPy package provides the usual discrete random variable methods for a Poisson random variable object.

- `pmf` - probability mass function
- `cdf` - cumulative distribution function
- `ppf` - percentage point function (inverse of CDF)
- `rvs` - random variate sample

Let's consider a case of a process where the expected number of occurrences per time period is 5. We'll freeze a SciPy Poisson object with this parameter.

```
In [13]: from scipy.stats import poisson
         p5 = poisson(5)
```

Since 5 is the expected value, we expect the mass function to be strongest around 5.

```
In [14]: p5.pmf([3,5,7]) * 100
```

```
Out[14]: array([ 14.03738958,  17.54673698,  10.4444863 ])
```

We see a 14% chance of getting a count of **3**, a 17.5% chance of getting a **5** count, and a 10.4% chance of getting a **7** count.

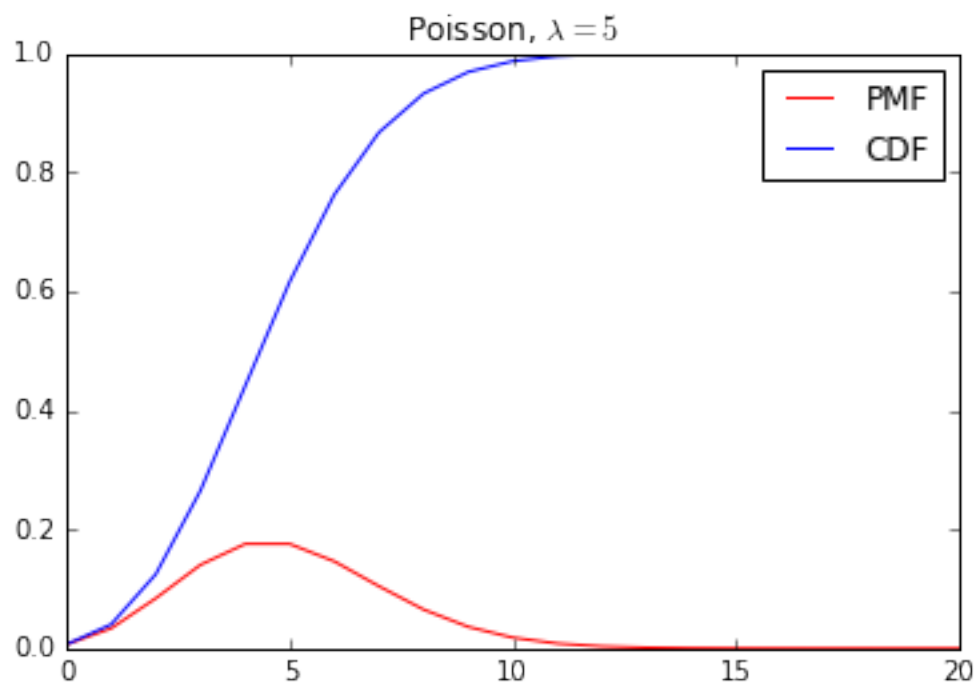
Now let's check the cumulative distribution function (CDF).

```
In [15]: p5.cdf([3,5,7]) * 100
```

```
Out[15]: array([ 26.50259153,  61.59606548,  86.66283259])
```

The chance of getting * 3 or less is 27% * 5 or less is 62% * 7 or less is 87%
Let's view this graphically.

```
In [16]: x = np.arange(21)
fig, ax = plt.subplots(1,1)
ax.plot(x, p5.pmf(x), 'r-', label="PMF")
ax.plot(x, p5.cdf(x), 'b-', label="CDF")
ax.legend()
plt.title("Poisson,  $\lambda = 5$ ")
plt.show()
```



The standard quartiles are given by

```
In [17]: p5.ppf([0.25, 0.5, 0.75])
```

```
Out[17]: array([ 3.,  5.,  6.])
```

- 3 is the smallest count that occurs at least 25% of the time
- 5 is the smallest count that occurs at least 50% of the time
- 6 is the smallest count that occurs at least 75% of the time

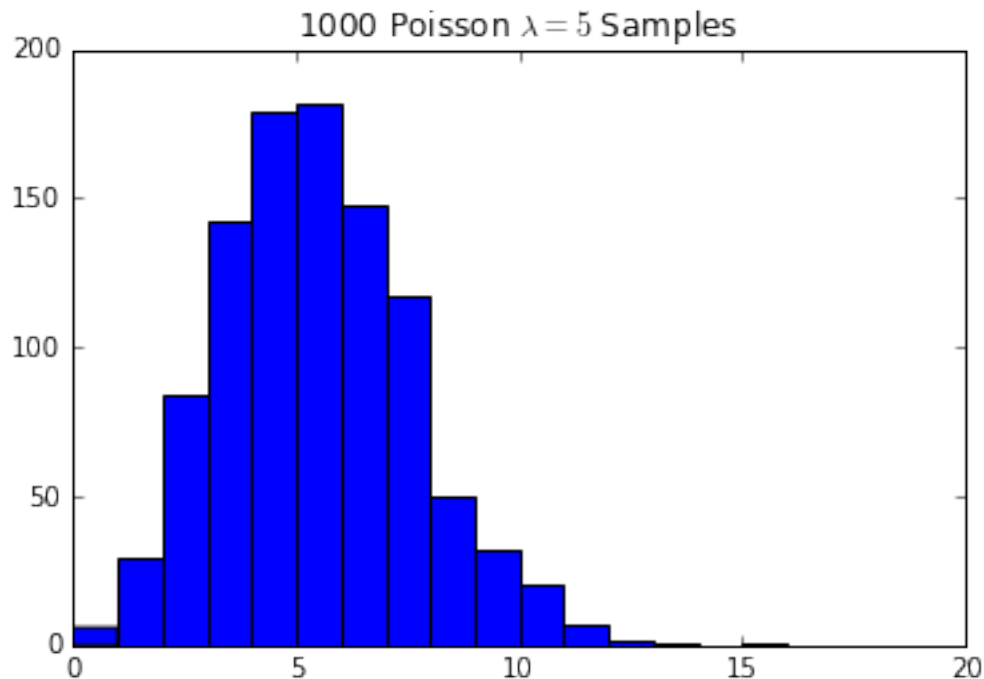
Finally, let's generate some Poisson random numbers. This is like counting occurrences of 1,000 outcomes for a Poisson process with $\lambda = 5$.

```
In [18]: p5_sample = p5.rvs(size=1000)
p5_sample[0:100]
```

```
Out[18]: array([ 4,  9,  4,  7,  4,  2,  6,  5,  2,  7,  3,  6,  3,  3,  3,  4,  7,
                2,  7, 10,  6,  7,  6,  4,  5,  9,  4,  3,  5,  6,  4,  4,  9,  3,
                7,  7,  2,  0,  5,  4,  8,  5,  3,  7,  7,  7,  6,  2,  3,  4,  4,
                9,  9,  6,  6,  2,  4,  4,  4,  8,  4,  4,  6,  5, 11,  4,  3,  5,
                3,  4,  4,  7,  5,  4,  7, 10,  2,  7,  5,  6,  3,  1,  5,  3,  3,
                5, 10,  4,  3,  5,  7,  1,  5,  5,  8,  6,  8,  7,  5, 10])
```

Let's check that the histogram of our sample looks anything like our PMF.

```
In [19]: fig, ax = plt.subplots(1,1)
         ax.hist(p5_sample, bins=20, range=(0, 20))
         plt.title("1000 Poisson  $\lambda=5$  Samples")
         plt.show()
```



1.4 3 Continuous Distributions

The following sections introduce SciPy objects working with continuous distributions. In addition to methods particular to each distribution, the [rv_continuous](#) page documents methods common to all continuous distributions.

The SciPy package provides the following continuous random variable methods.

- **pdf** - probability density function
- **cdf** - cumulative distribution function
- **ppf** - percentage point function (inverse of CDF)
- **rvs** - random variate sample

1.4.1 3.1 Normal

The normal distribution is handled by the [norm](#) object. We won't create a [frozen](#) instance since we'll work with mean 0 and standard deviation 1.

```
In [20]: from scipy.stats import norm
         norm.pdf([-3, -2, -1, 0, 1, 2, 3])
```

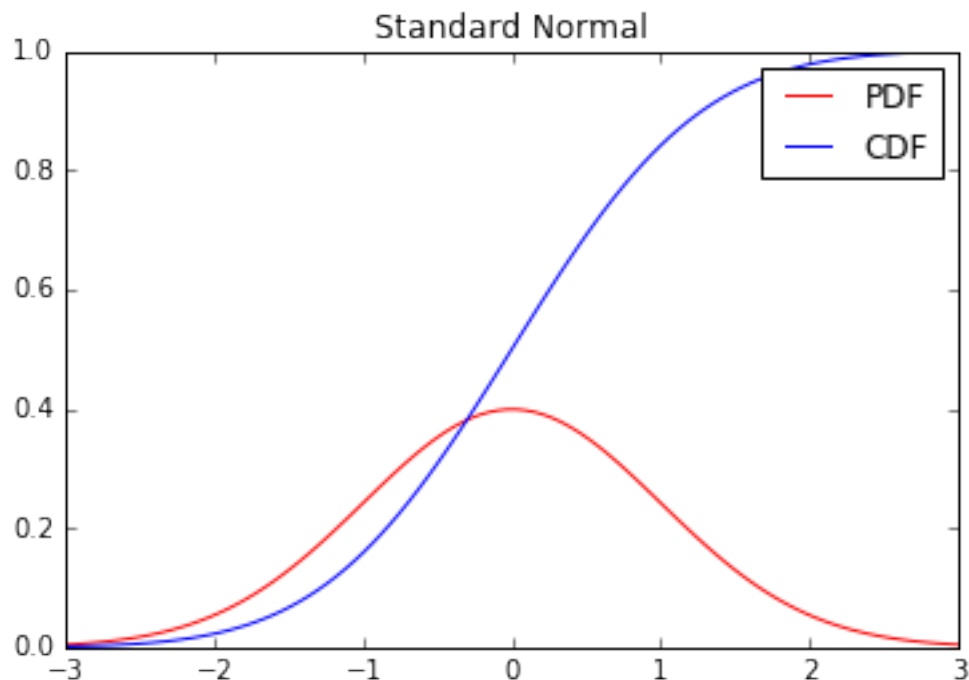
```
Out[20]: array([ 0.00443185,  0.05399097,  0.24197072,  0.39894228,  0.24197072,
                0.05399097,  0.00443185])
```

The symmetric character of the normal distribution is apparent here. We don't interpret these as probabilities (like we did with discrete mass function) since the continuous case represents an infinitesimal range of the random variable. But we still extract probabilities from the CDF.

```
In [21]: norm.cdf([-3, -2, -1, 0, 1, 2, 3]) * 100
```

```
Out[21]: array([ 0.1349898 ,  2.27501319, 15.86552539, 50.          ,
                84.13447461, 97.72498681, 99.8650102 ])
```

```
In [22]: x = np.linspace(-3, 3, 100)
         fig, ax = plt.subplots(1,1)
         ax.plot(x, norm.pdf(x), 'r-', label="PDF")
         ax.plot(x, norm.cdf(x), 'b-', label="CDF")
         ax.legend()
         plt.title("Standard Normal")
         plt.show()
```



The `ppf` function shows us where the quantiles are. In most of the distribution examples, we've been displaying quantiles for quarters, i.e. the quantiles. But due to role of normal distribution in the [Central Limit Theorem](#), we'll examine quantiles for 90%, 95%, and 97.5% since these are often referenced in the study of confidence intervals.

```
In [23]: norm.ppf([0.5, 0.9, 0.95, 0.975])
```

```
Out[23]: array([ 0.          ,  1.28155157,  1.64485363,  1.95996398])
```


Since the normal distribution is symmetric about the origin, we expect half to lie on one side of 0 and half on the other side. The last number says that 97.5% of the curve lies to the left of **1.96**. This is a number we'll commit to memory during the study of confidence intervals.

1.4.2 3.2 Exponential Distribution

The exponential distribution is a continuous distribution that complements the discrete Poisson distribution that we encountered above. Whereas a Poisson random variable represents a count of events that occur over a fixed unit of time, an exponential random variable represents the amount of time between one such event and the next.

The density function for the exponential distribution is

$$f_X(x; \lambda) = \lambda e^{-\lambda x}$$

where the expected wait time is $1/\lambda$. The SciPy object is `expon`.

```
In [24]: from scipy.stats import expon
        exp3 = expon(scale=3)
```

The second line creates a “frozen” exponential object with $1/\lambda = 3$. Let's go through the usual motions. The exponential density function decreases as x gets larger.

```
In [25]: x = exp3.pdf([1,2,3,4])
        np.round(x, 2)
```

```
Out[25]: array([ 0.24,  0.17,  0.12,  0.09])
```

Since the expected wait time is 3.0 units of time, we expect the CDF to cross 50% there.

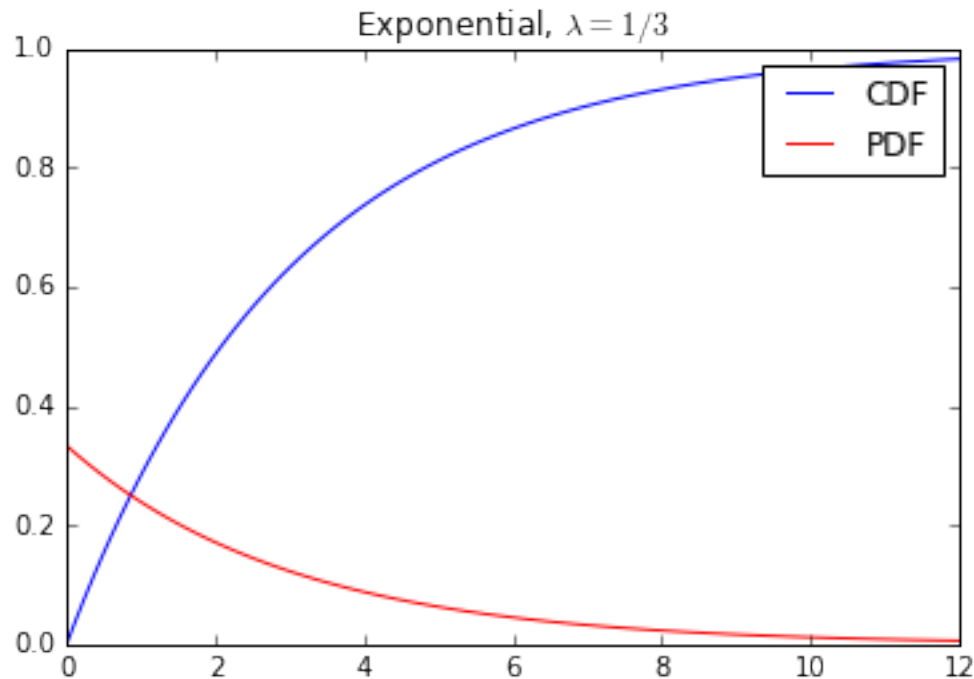
```
In [26]: x = exp3.cdf([1,2,3,4])
        np.round(x, 2)
```

```
Out[26]: array([ 0.28,  0.49,  0.63,  0.74])
```

This is the exponential CDF and PDF graphically.

```
In [27]: x = np.linspace(0, 12, 100)
        fig, ax = plt.subplots(1, 1)
        ax.plot(x, exp3.cdf(x), 'b-', label="CDF")
        ax.plot(x, exp3.pdf(x), 'r-', label="PDF")
        ax.legend()
        ax.set_title("Exponential,  $\lambda = 1/3$ ")
```

```
Out[27]: <matplotlib.text.Text at 0x11443e400>
```



The standard quartile are given by

```
In [28]: x = exp3.ppf([0.25, 0.5, 0.75])
         np.round(x, 2)
```

```
Out[28]: array([ 0.86,  2.08,  4.16])
```

Notice that the **median** (the .5 quantile) is 2.08. This is an example where the median and the average do not agree, even for a continuous random variable. These values tell us that we should expect a wait time of

- 0.86 time units or less 25% of the time
- 2.08 time units or less 50% of the time
- 4.16 time units or less 75% of the time

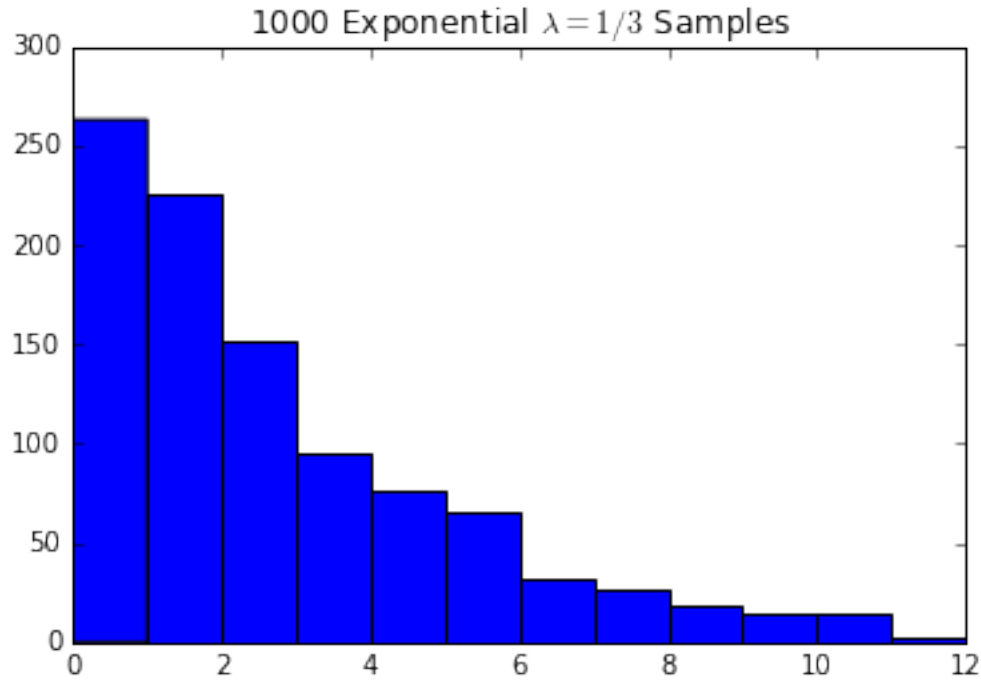
Let's generate some exponentially distributed times for a simulated Poisson process.

```
In [29]: e_sample = exp3.rvs(size=1000)
         np.round(e_sample[0:40], 2)
```

```
Out[29]: array([ 0.26,  0.34,  0.88,  4.09,  3.96,  2.24,  2.25,  0.29,  2.87,
                  1.75,  0.14,  2.54,  1.85,  2.6 ,  4.34,  8.25,  0.18,  8.08,
                  1.09,  3.63,  0.75,  0.54,  2.33,  0.37,  1.57,  1.04,  4.83,
                  0.56,  0.4 ,  5.2 ,  1.07,  1.31,  3.74,  2.57,  0.48,  0.48,
                  0. ,  1.69,  5.89,  3.82])
```

```
In [30]: fig, ax = plt.subplots(1,1)
         ax.hist(e_sample, bins=12, range=(0, 12))
         ax.set_title("1000 Exponential  $\lambda = 1/3$  Samples")
```

```
Out[30]: <matplotlib.text.Text at 0x114776048>
```



1.4.3 3.3 Gamma Distribution

A Gamma random variable is often interpreted as a sum of r independent exponential random variables with the same parameter λ . The density function is

$$f_X(x; \lambda, r) = \frac{\lambda^r}{\Gamma(r)} (\lambda x)^{r-1} e^{-\lambda x}$$

So we have a r Poisson processes with an expected wait time of $1/\lambda$. Let's continue using our Poisson process from before with an expected wait time of 3 time units; but this time let's assume we are waiting for five of them (one after another). So in this particular case, our $\lambda = 1/3$ and $r = 5$.

```
In [31]: from scipy.stats import gamma
         g35 = gamma(a=5, scale=3)
         x = g35.pdf([5, 10, 15, 20, 25])
         np.round(x, 2)
```

```
Out[31]: array([ 0.02,  0.06,  0.06,  0.03,  0.02])
```

The cumulative distribution function gives a better feel for the distribution.

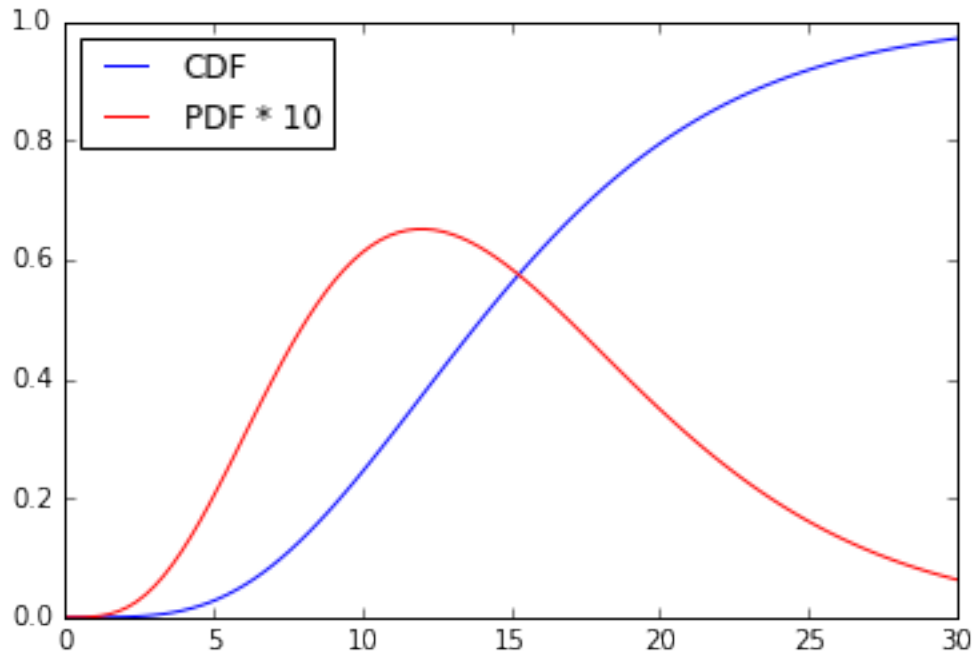
```
In [32]: x = g35.cdf([5, 10, 15, 20, 25])
         np.round(x, 2)
```

```
Out[32]: array([ 0.03,  0.24,  0.56,  0.79,  0.92])
```

If we expect each event to take 3 time units on average, and there are five events, then 15 is a reasonable half-way point.

```
In [34]: x = np.linspace(0, 30, 100)
fig, ax = plt.subplots(1,1)
ax.plot(x, g35.cdf(x), 'b-', label="CDF")
ax.plot(x, g35.pdf(x) * 10, 'r-', label="PDF * 10")
ax.legend(loc="upper left")
```

```
Out[34]: <matplotlib.legend.Legend at 0x113e7bc88>
```



I multiplied the PDF by 10 to make its shape more pronounced against the CDF.
The quartiles are given by

```
In [35]: x = g35.ppf([0.25, 0.5, 0.75])
np.round(x, 2)
```

```
Out[35]: array([ 10.11,  14.01,  18.82])
```

This that we can expect a trial (which consists of 5 independent Poisson processes, each with expected duration of 3 time units) to take less than

- 10.11 time units 25% of the time
- 14.01 time units 50% of the time
- 18.82 time units 75% of the time

Let's simulate 1,000 trials through a random number generator.

```
In [37]: g35_sample = g35.rvs(size=1000)
np.round(g35_sample[0:50], 2)
```

```
Out[37]: array([ 13.59,  11.32,  17.64,  25.96,  22.57,   6.71,  13.45,  39.36,
                21.89,   6.35,  14.85,  19.43,  15.41,   9.01,  14.04,   1.74,
                16.72,  20.34,  19.12,  11.55,  18.87,  10.84,  10.52,   5.79,
                11.17,  53.16,  41.61,  23.08,  21.57,  17.6 ,  11.62,  26.77,
```

```
3.67, 13.68, 8.79, 11.6 , 4.78, 13.86, 15.94, 14.19,  
9.41, 15.88, 13. , 10.82, 17.79, 12.49, 7.86, 13.52,  
9.65, 21.69])
```

```
In [38]: fig, ax = plt.subplots(1,1)  
ax.hist(g35_sample, bins=15, range=(0,30))  
ax.set_title("Gamma Sample,  $r=5$ ,  $\lambda=1/3$ ")
```

```
Out[38]: <matplotlib.text.Text at 0x113b7bbe0>
```

