Download `terrorism.txt` from http://tuvalu.santafe.edu/~aaronc/powerlaws/data.htm (http://tuvalu.santafe.edu/~aaronc/powerlaws/data.htm) (#7).

In [1]:

```
fname = 'terrorism.txt'
```

You can open in a text editor or simply:

In [2]:

```
!head 'terrorism.txt'
```

```
1
1
1
1
1
1
1
1
1
1
```

In [3]:

```
!tail 'terrorism.txt'
```

```
'tail' is not recognized as an internal or external command,
operable program or batch file.
```

This file contains a sorted list of "degree" or size values. In the case of terrorism, each line represents a terrorist attack and the number represents the number of deaths in the attack. Note that it doesn't have to come from a network. In network case, each line corresponds to a node and the values would represent the degree of the node.

We can load the numbers like the following:

In [4]:

```
nums = [int(x) for x in open(fname)]
print(nums[:5], '...', nums[-5:])
```

```
[1, 1, 1, 1, 1] ... [317, 329, 331, 400, 2749]
```

Let's try to plot the distribution. First, we want to import `matplotlib` for plotting. `%matplotlib inline` allows us to see the plot directly in this Jupyter notebook.

In [5]:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

A useful container is `Counter`. It is a dictionary that is specialized for counting.

In [6]:

```
from collections import Counter

a = Counter([1,1,1,1,2,2])
a
```

Out[6]:

```
Counter({1: 4, 2: 2})
```

Thus we can just pass the `nums` to get the degree distribution. First, let's count the number of data points that we have and construct the counter.

In [7]:

```
N = len(nums)
N
```

Out[7]:

```
9101
```

In [8]:

```
nk = Counter(nums)
```

now, we'll create two lists: x stores the unique degree or the values and y stores the number of data points with the value. For instance, according to

In [9]:

```
print(nk[1], nk[2])
```

```
4802 1600
```

There are 4802 data points with one death and 1600 data points with two deaths. If we just have them in our dataset, x and y will be:

```
x = [1, 2]
y = [4802, 1600]
```

Let's construct them.

In [10]:

```
x = []
y = []
for k in sorted(nk):
    x.append(k)
    y.append(nk[k])
```

Directly plotting the degree distribution

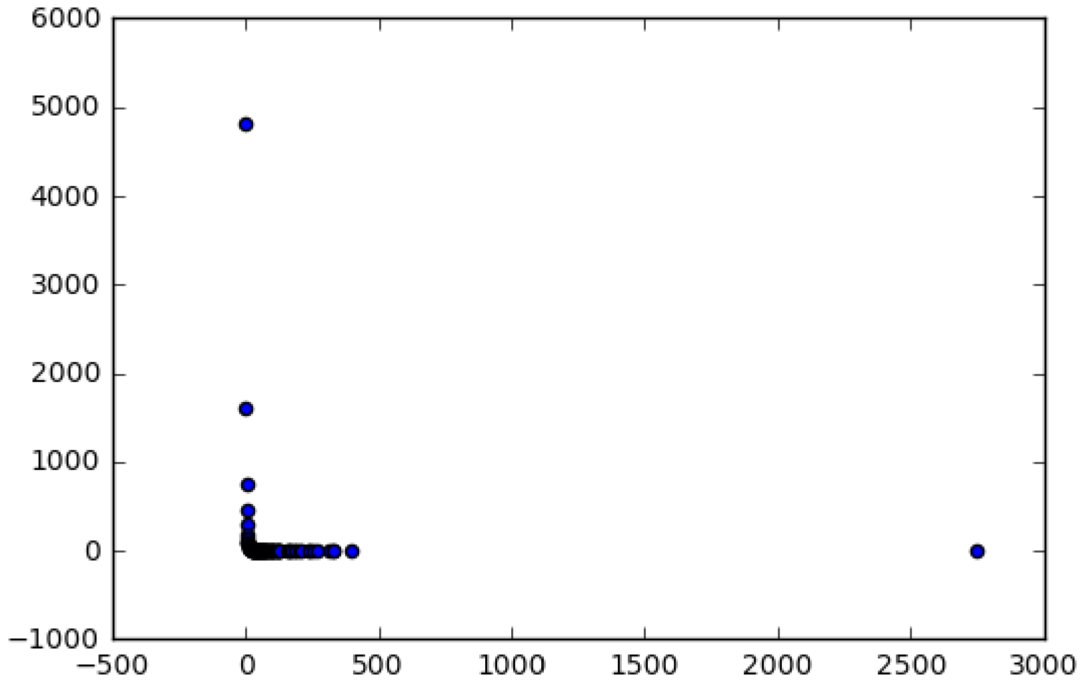
If we just plot them, it's the raw (degree) distribution. Let's plot in the normal scale first.

In [11]:

```
plt.scatter(x,y)
```

Out[11]:

```
<matplotlib.collections.PathCollection at 0x1ae4afefa20>
```

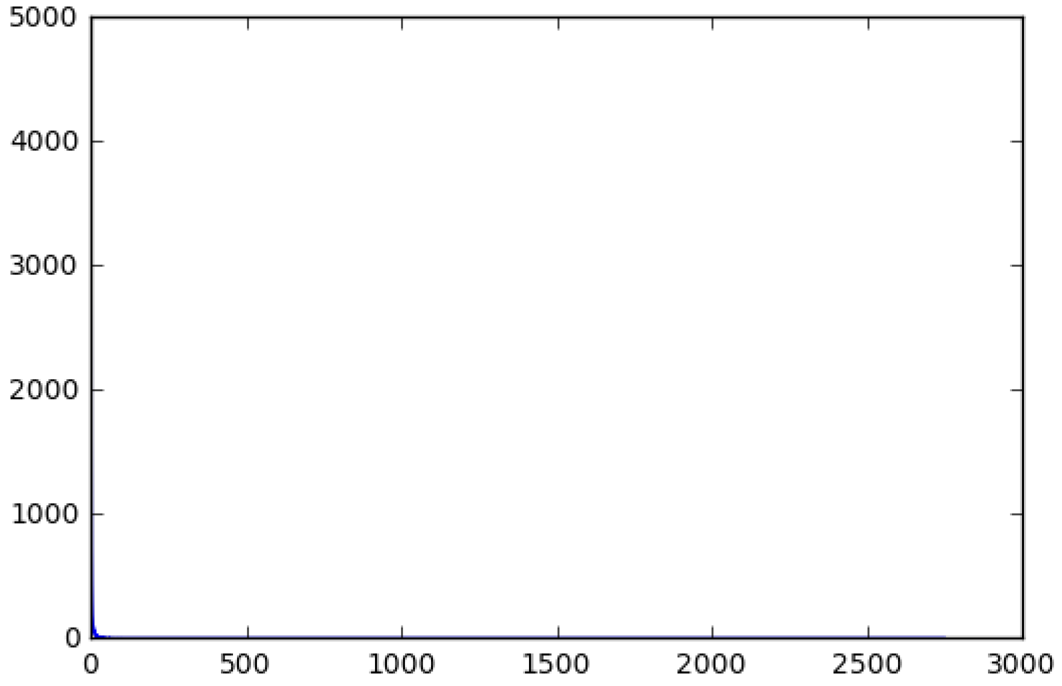


In [12]:

```
plt.plot(x,y)
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x1ae4b75e198>]
```



Log-Log scale

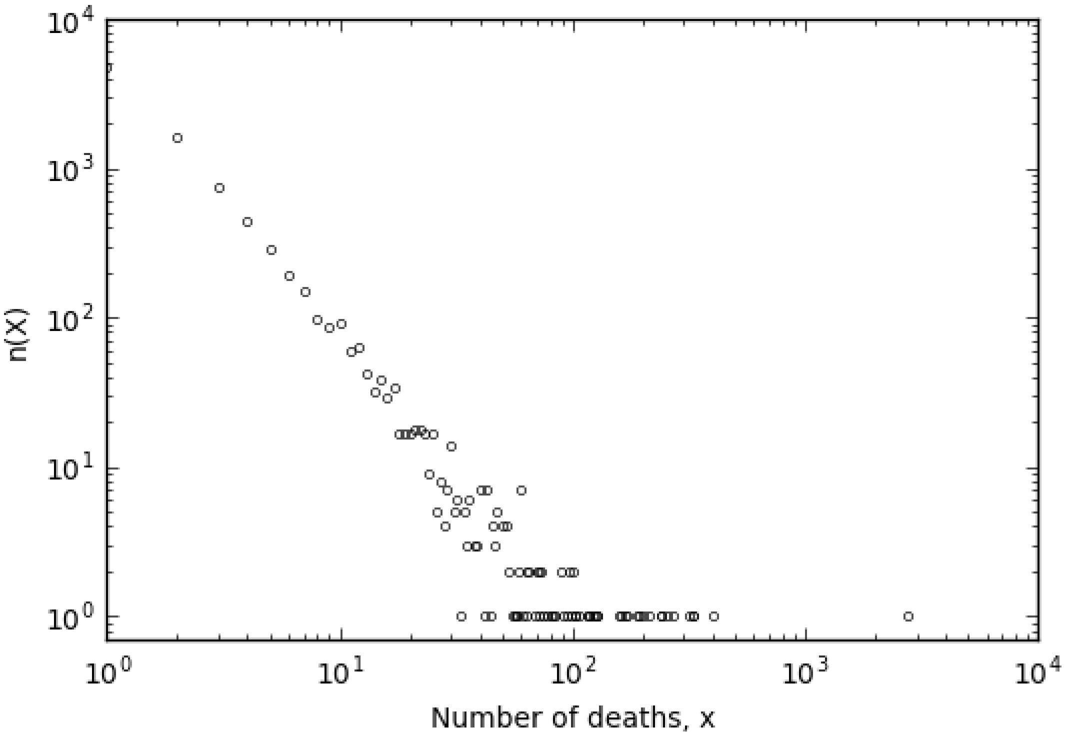
This is how a heavy-tailed distribution look like in normal scale. You can't see much. Let's try log-log scale.

In [13]:

```
plt.ylim((0.7, 10000)) # more clearly show the points at the bottom.
plt.xlabel('Number of deaths, x')
plt.ylabel("n(X)")
plt.loglog(x,y, 'o', markersize=3, markerfacecolor='none')
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x1ae4b71ae10>]
```



Log-binning

Log-binning indicates a way to create bins for histogram in a way that they are equally distributed in log-scale. So, if the first bin is [1.0, 2.0], the next bin will be [2.0, 4.0], and [4.0, 8.0], and so on. `numpy` has a very conveninent function for this. As we know that the data ranges from 1.0 to several thousands, we can set the range to be [0.0, 4.0] (`0.0 = log 1.0; 4.0 = log 10000`).

```
In [14]:
```

```
import numpy as np

bins = np.logspace(0.0, 4.0, num=40)
bins
```

```
Out[14]:
```

```
array([  1.00000000e+00,   1.26638017e+00,   1.60371874e+00,
         2.03091762e+00,   2.57191381e+00,   3.25702066e+00,
         4.12462638e+00,   5.22334507e+00,   6.61474064e+00,
         8.37677640e+00,   1.06081836e+01,   1.34339933e+01,
         1.70125428e+01,   2.15443469e+01,   2.72833338e+01,
         3.45510729e+01,   4.37547938e+01,   5.54102033e+01,
         7.01703829e+01,   8.88623816e+01,   1.12533558e+02,
         1.42510267e+02,   1.80472177e+02,   2.28546386e+02,
         2.89426612e+02,   3.66524124e+02,   4.64158883e+02,
         5.87801607e+02,   7.44380301e+02,   9.42668455e+02,
         1.19377664e+03,   1.51177507e+03,   1.91448198e+03,
         2.42446202e+03,   3.07029063e+03,   3.88815518e+03,
         4.92388263e+03,   6.23550734e+03,   7.89652287e+03,
         1.00000000e+04])
```

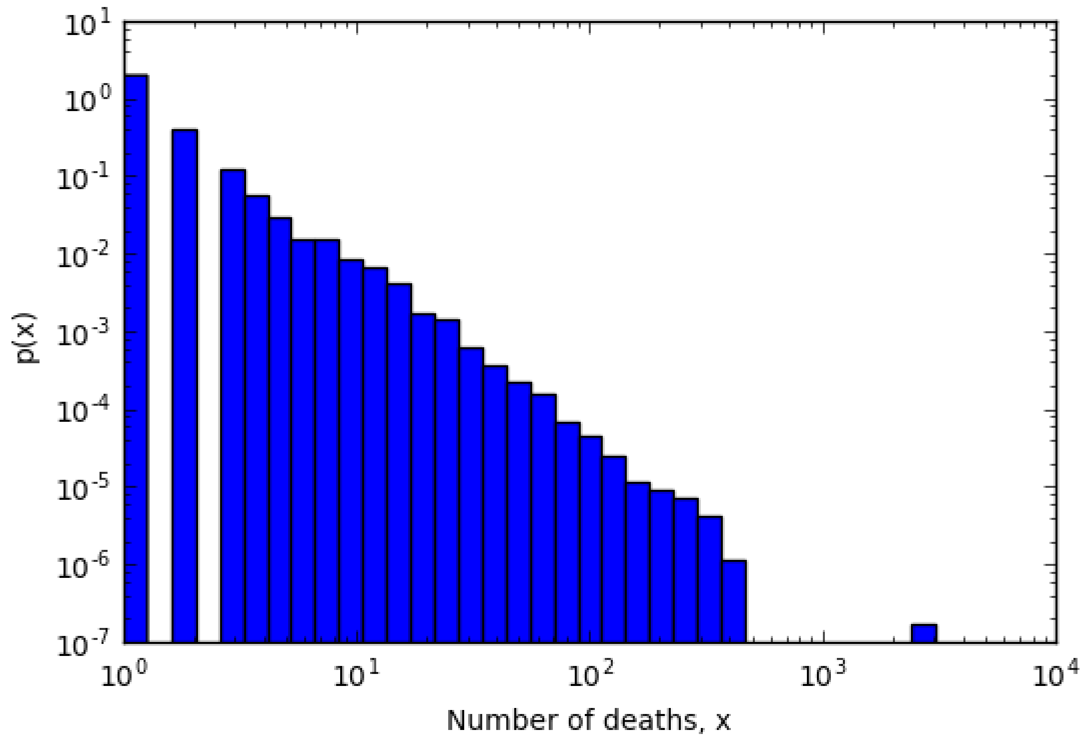
In log-binnning, you *multiply* a constant to create the subsequent bin, rather than *adding* a constant.

Then we can literally draw a "histogram" based on these bins.

In [15]:

```
ax = plt.subplot(1,1,1)

ax.set_xlabel('Number of deaths, x')
ax.set_ylabel('p(x)')
ax.hist(nums, bins=bins, normed=True)
ax.set_xscale('log')
ax.set_yscale('log')
```



Alternatively, we can obtain the histogram using numpy's `histogram` function (http://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html) and then plot the points in the same style as we did with the raw degree distribution.

In [16]:

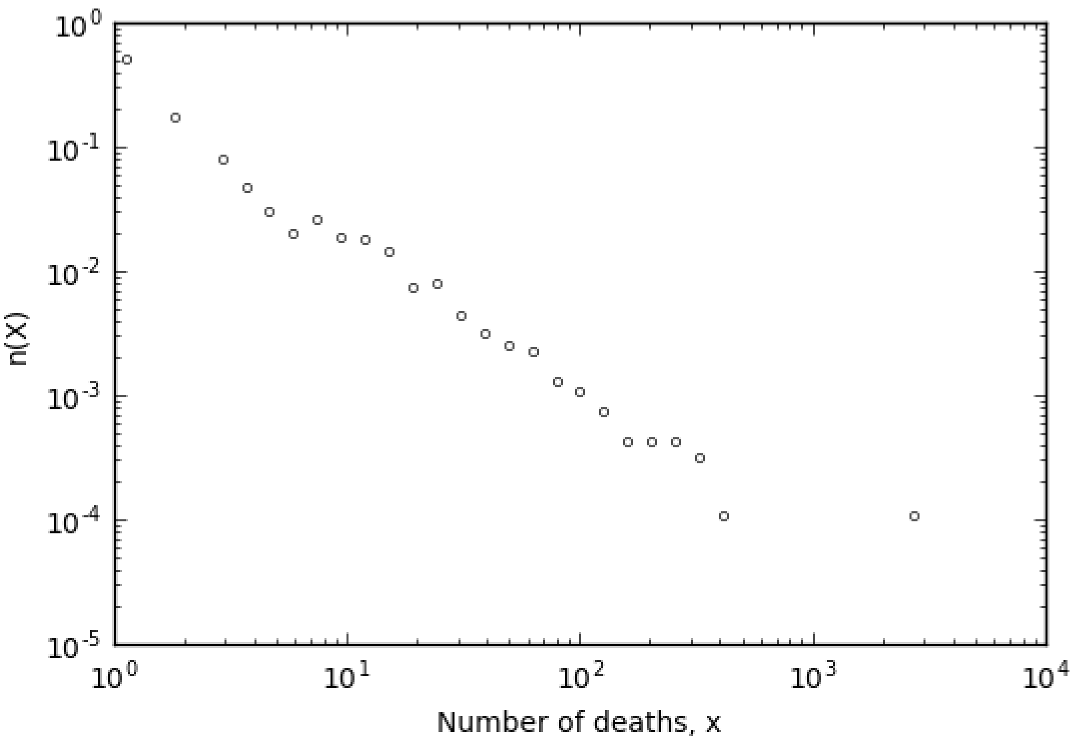
```
Y, X = np.histogram(nums, bins=bins, normed=True)

X = [x*np.sqrt(bins[1]) for x in X][:-1]  # find the center point for each bin. can you exp

plt.ylim((0.00001, 1))
plt.xlabel('Number of deaths, x')
plt.ylabel("n(X)")
plt.loglog(X,Y, 'o', markersize=3, markerfacecolor='none')
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x1ae4cc9acf8>]
```



Q: Can you plot the complementary cumulative distribution function?

First of all, you may be confused between CDF (cumulative distribution function) and CCDF (complementary cumulative distribution function). They are all "cumulative", but the difference is that the former starts from left and the latter starts from the right.

So, as Power laws, Pareto distributions and Zipf's law (https://arxiv.org/pdf/cond-mat/0412004v3.pdf) (Fig. 3 (d)) paper explains, plotting the CCDF (complementary cumulative distribution function) is probably the best method to show a heavy-tailed distribution. Below, calculate the CCDF and plot it in a similar style (log-log, with symbols).