[1]:
```
%matplotlib inline
# make figures better:
import matplotlib
font = {'size':18}
matplotlib.rc('font', **font)
```

DS1 Lecture 09

Jim Bagrow

Last time:

1. Spreadsheets are bad for data science
2. Cleaning data → exploring data
   - rejecting bad data or flagging bad data
   - combining *duplicate* data
   - filtering or normalizing data?

Today's plan:

1. Major tools for exploring data

Determine bad data by identifying good data

- What are your requirements or **validity conditions** for the individual datapoints (or fields)?

For example, you have some **ecological data** consisting of the **number of grass species** at **different locations** and **times** in VT:

- Each datapoint should be `(lat,lon, timestamp, N)`

If you load a `lat` or `lon` and it's not a number (or something like `51°28'38" N`) then you know something is wrong.

But you also have information about the **ranges** of each variable:

1. `lat`,`lon` must be a point in Vermont,
2. `timestamp` must be in your data **window** (maybe between `2013-01-01` and `2014-01-01`, for this example),
3. `N` ≥ 0.

Example data checker:

```
t_start = datetime.strptime("2013-01-01", "%Y-%m-%d")
t_stop  = datetime.strptime("2014-01-01", "%Y-%m-%d")

bad_data = [] # T or F for each datapoint
for lat,lon,t,N in dataset:
    if not in_vermont(lat,lon)      or \
       not (t_start <= t <= t_stop) or \
       N < 0:
        bad_data.append(True)
    else:
        bad_data.append(False) # data seems OK
```

Depending on your problem, you may want to **repair** the data instead of just **flagging** it as good or bad:

```
if N < 0:
    N = 0
# etc
```

Determining “good” data is sometimes called **data modeling**, although “data modeling” often specifically refers to capture relationships between different groups of data (“this table was derived from that table”, “joining these two tables produces that table”, etc.).

How to detect bad data

Often it’s very easy to determine a bad value using external knowledge to build validity conditions, like in the above ecology example.

But what if you lack this knowledge?

This is where exploratory tools are helpful.

- Let’s take a look at an example for numeric data:

Example: fish mass data Suppose your job is to hang out on an **Alaskan fishing boat** and monitor their daily catch so the crew doesn’t exceed their quota, etc. (This is a real job!) Suppose over time you measure the mass of many fish. Being a diligent scientist you log these readings in a `fishMass_kilograms.txt` file.

Let’s read in the data and print the first few values:

(Always *look* at the data!)

[2]:

```
data = []
for line in open("fishMass_kilograms.txt"):
    data.append( float(line.strip()) )

print("There are %i data points." % len(data) )
print()

for d in data[:20]:
    print(d)
print(".\n"*3)
```

```
There are 1300 data points.

5.77967973162
3.26834145824
0.06418251738
4.38979192127
4.68302244707
4.82366715649
4.68587041117
0.04360063509
5.90498807235
4.3618070355
0.0017977901
4.9891841837
4.56259294774
```

```
5.44050157565
5.19592386044
15.6959515181
3.22732340991
5.57228018649
3.7148892443
5.00286245308
.
.
.
```

OK, so it looks like most fish are around 3-5 kg. But look **more closely**: There's a few fish that are **very tiny** and one fish is *huge*!

Here we just have a big list of numbers. This is what I call "X-data" because there's nothing else to serve as Y to plot against.

- We can transform this into "XY-data" by looking at the **distribution**, how much of the data is big, how much is small, etc.

Histogram:

A technique to visualize the underlying distribution of X-data.

(Histograms are simple, yet they are one of the **most important tools** in our "exploratory data analysis" (EDA) toolbox!

I will actually spend a long time talking about histograms, because they are so important and because there are some subtle points that are hidden by how simple histograms are!)

To build a histogram:

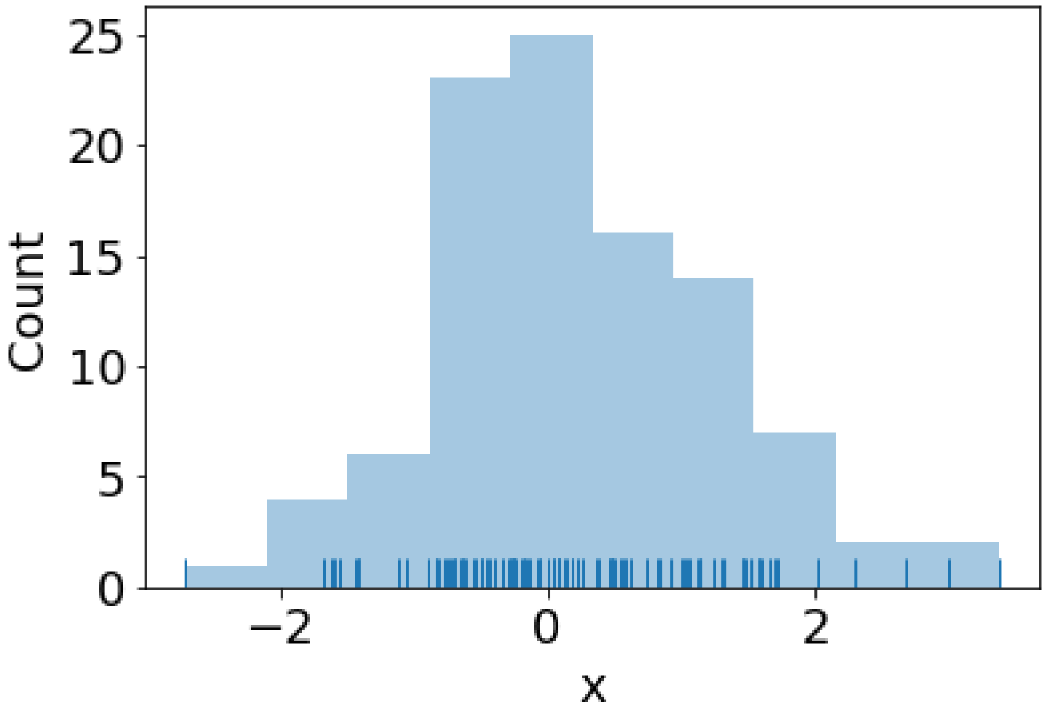
1. Take the **range** of data (`range = [min(data),max(data)]`) and chop it up into consecutive intervals, called **bins**.
2. Then calculate, for each data point, which bin it falls into. Keep a count of the number of points inside each bin.
3. Plot the number of data points in each bin vs. the *center* coordinate of the bin. This draws the histogram. (It is also very common to represent each bin as a box.)

This is very easy to implement, especially if all the bins are the **same width**. But we've even got a **built in function** to do this.

[3]:

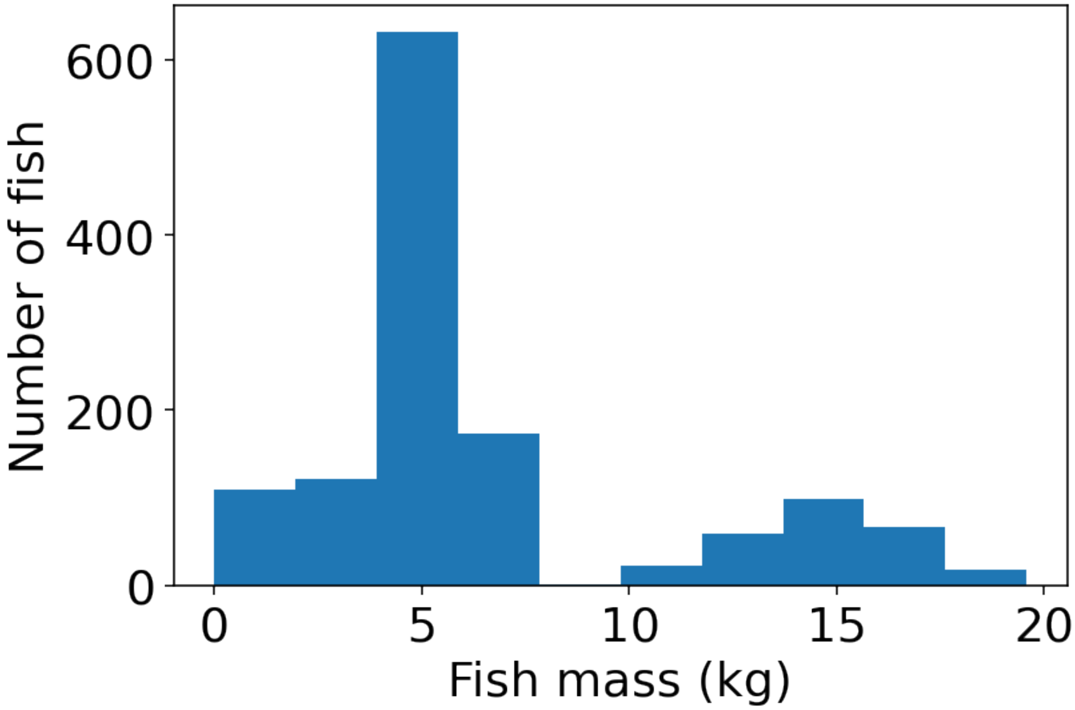
```python
from IPython.display import Image
Image("figures/histogram-illustration.png")
```

[3]:



Let’s bin our data!

[4]:
```
import matplotlib
import matplotlib.pyplot as plt

# look at the docstring for `hist` to see how to use it!
plt.hist(data)

# pretty up the figure a little
plt.xlabel("Fish mass (kg)")
plt.ylabel("Number of fish")
plt.show()
```



Now we're getting a picture of the data!

It looks like the data are **bimodal**: there's a bunch of fish around 5kg and another group between 10-20kg. Maybe they are different species?

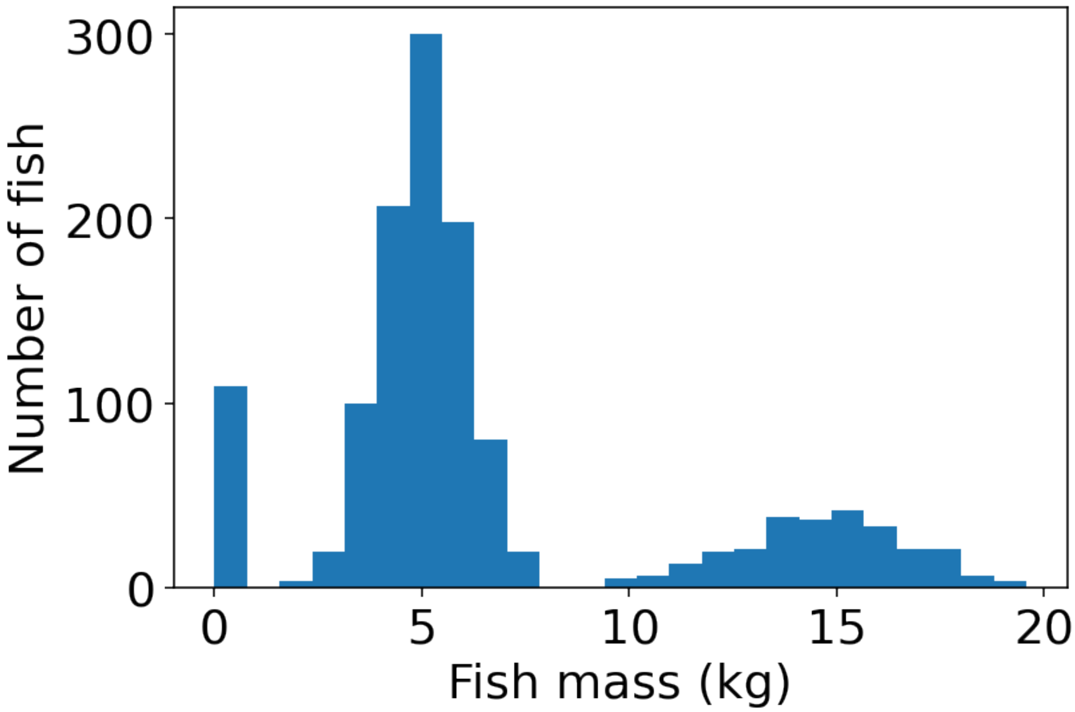
Bins

- The default for `plt.hist()` is 10 bins. But we've got lots of data, so let's try **more bins**:

(Always vary the bins!)

[5]:

```
plt.hist(data, bins=25);
#              ^^^^^^^
plt.xlabel("Fish mass (kg)")
plt.ylabel("Number of fish")
plt.show()
```

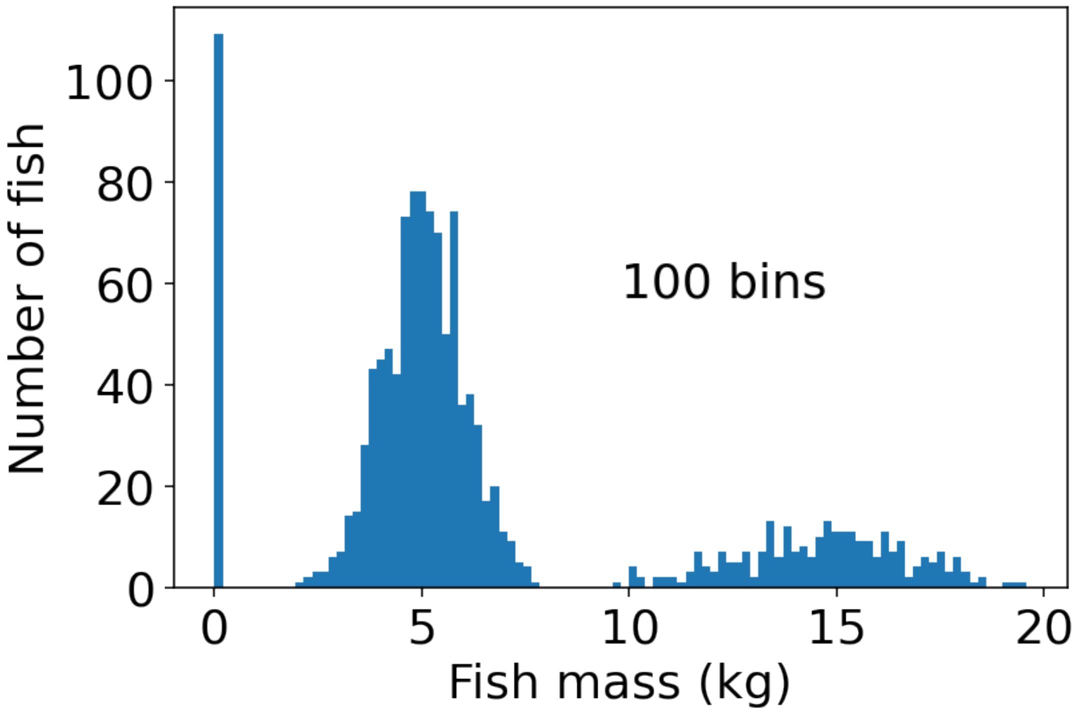


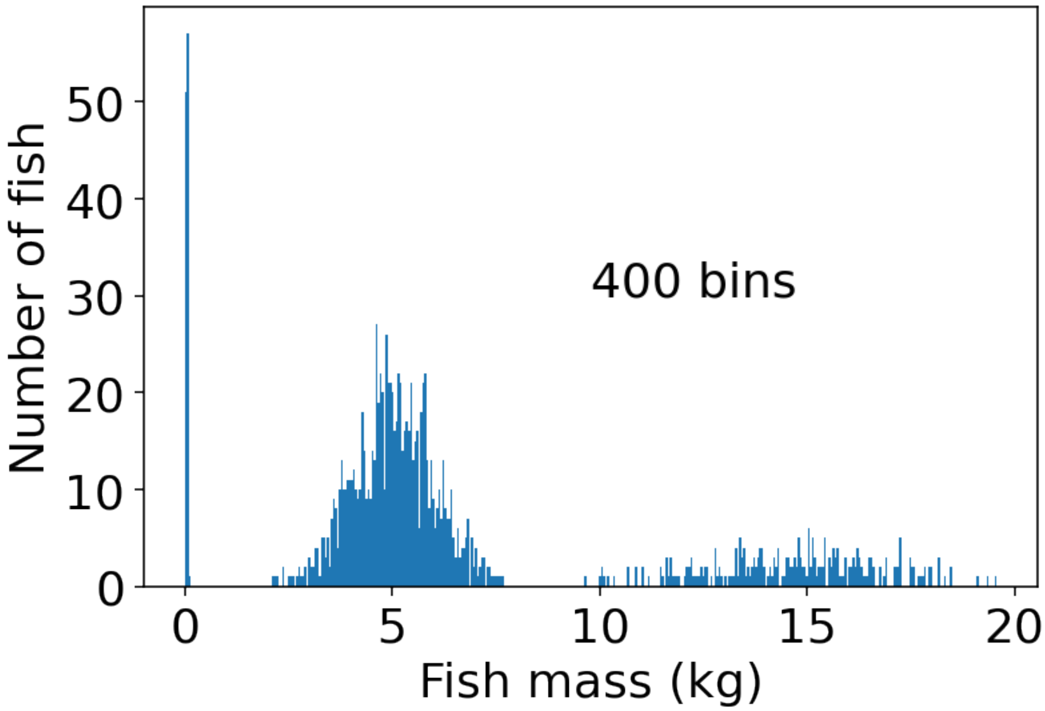
Nice smooth curves… **WAIT** What's that isolated bin going on near 0 kg? Is there a *third* species?

- *MOAR BINS*

[6]:

```
for b in [100,400]:
    plt.hist(data, bins=b);

    plt.xlabel("Fish mass (kg)")
    plt.ylabel("Number of fish");
    plt.text(0.5, 0.5,f'{b} bins',
             horizontalalignment='left',
             transform=plt.gca().transAxes)
    plt.show()
```





Very small fish!

- Now it could be that mixed in with the huge sturgeon are very tiny guppies and minnows, but that seems unlikely.
- More likely is that some readings got corrupted somehow, or were misrecorded…

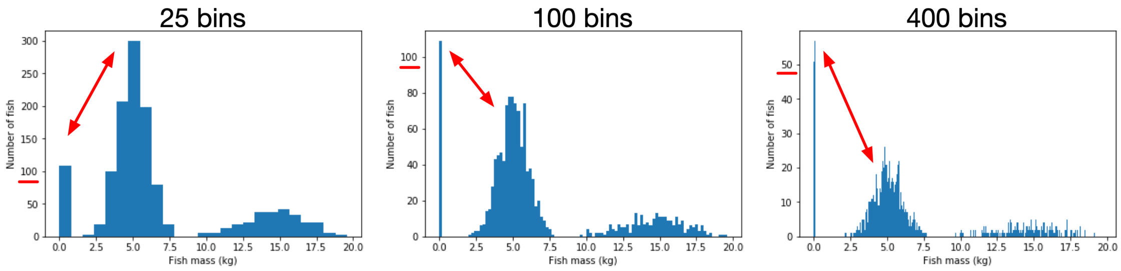
Choosing the number of bins

Histograms live or die by the appropriate binning. **You must always tune the bins accordingly.**

[7]:

```
Image("figures/compare-bins-fish.png")
```

[7]:



The number of bins has some subtle but important effects.

Notice how it appears that the relative proportion of ultra-small fish grows with the number of bins. This is due to counting the number of points that fall within each bin: as the bins get narrower, the counts per bin decrease, making the two right-most modes appear less tall. But no data has been lost, there are just more bins that the data are spread over. This is one reason why it is critical to normalize histograms by their ***area***.

- **Overbinning**. The "fuzziness" that starts to appear in the 100-bin plot and is very obvious in the 400-bin plot is a prime symptom of overbinning. When you see this, that is the clue to dial back the number of bins.

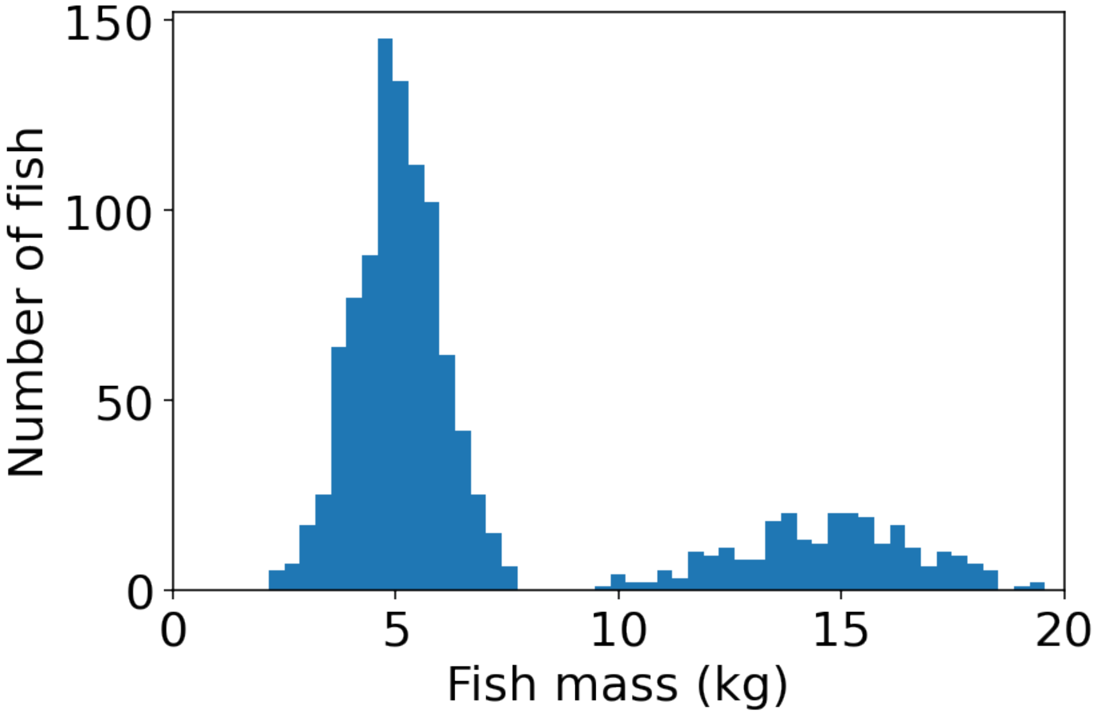
Cleaning the "bad" points Having used our exploratory tool (the histogram) to **identify** the third group of observations, we can now decide if we want to investigate further, flag them as suspicious and move on, or remove them altogether.

Let's remove them:

[8]:

```
print("original range = [%f, %f]" % (min(data),max(data)))

new_data = [d for d in data if d >= 1.0] # only 1 kg+ fish can be caught?

print("     new range = [%f, %f]" % (min(new_data),max(new_data)))

plt.hist(new_data, bins=50)
plt.xlabel("Fish mass (kg)")
plt.ylabel("Number of fish")

plt.xlim(0,20); # override plt default x-axis range
plt.show()
```

```
original range = [0.000120, 19.573999]
     new range = [2.152409, 19.573999]
```



This code enforced a new validity condition: that all valid readings are at least 1 kg. The big question is whether or not this was the right decision to make…

What we have just done is **outlier removal**. We've found anomalous points in our data using a basic analysis and then we've removed them from our sample.

Outliers
There is no mathematically precise definition of what an outlier is. It is a *subjective* notion, and has vexed science for hundreds of years.

(Even) More on histograms — Binning vs. counting

Binning

Histograms divide the "domain" of the data into bins and count how many observations fall within each bin.

Specifically, for a bin w defined to start at $x = x_\mathrm{L}$ and end at $x = x_\mathrm{R}$, the set of data X_w that fall within w are

$$X_w \equiv \{x_i \mid x_\mathrm{L} \leq x_i < x_\mathrm{R}\},$$

where x_i is the ith observation. For an unnormalized histogram, bin w can be visualized as a box with left side at $x = x_\mathrm{L}$, right side at $x = x_\mathrm{R}$, bottom at $y = 0$, and top at $y = |X_w|$. Alternatively, it is also common to visualize the bin as a point at $x = (X_\mathrm{L} + X_\mathrm{R})/2$ and $y = |X_w|$.

- A "binning" is a **coarse-graining** of the data. All the values inside a bin are replaced by the count of those values which is shown either as a box or as a point placed at the horizontal center of the bin.

Continuous data

“Real” numbers (floats). Often you have to bin these data to see the underlying distribution. (But stay tuned to next class!)

Integer/discrete data

Some data naturally divide into groups, for example integers. Integers can be **counted** just as well as binned and if we retain a count for each unique value → no data loss.

- Counting is often superior to binning when appropriate, as no data loss is incurred.
- You’ve been making little (unplotted) histograms every time you’ve used `collections.Counter`!

Rational numbers

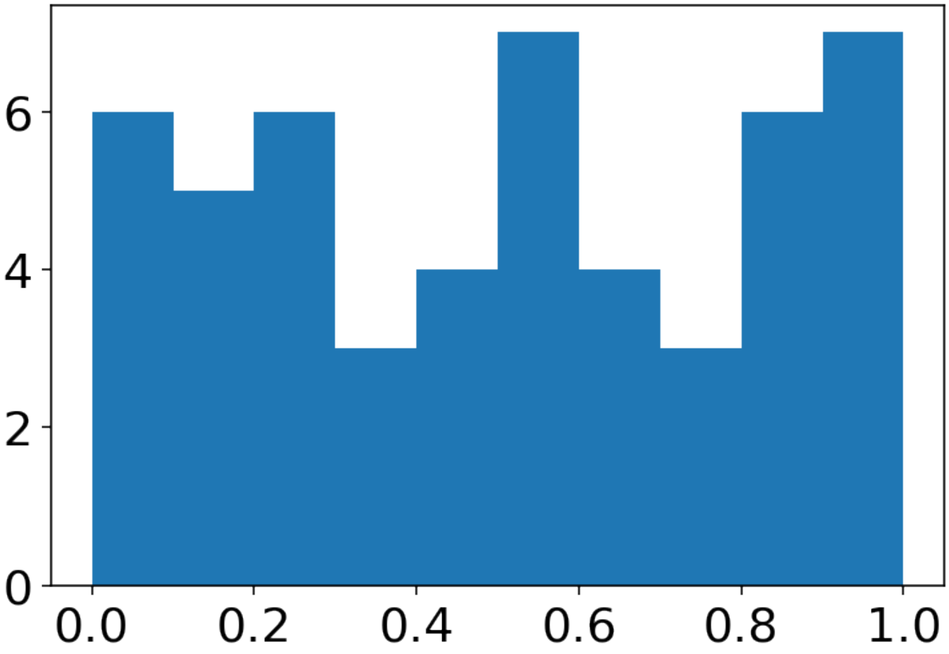
Sometimes data are defined as rational numbers. For example, suppose we are studying a **disease outbreak** in a school and we determine for each student how many friends they have and how many friends are sick. We can then define the following fraction for each student:

$$f_{sick} = \frac{\#\{\text{sick friends}\}}{\#\{\text{friends}\}}$$

If most students have 5-10 friends, then all the values of f_{sick} we will see will be in $0/5, 1/5, 2/5, \ldots, 5/5$, or $0/6, 1/6, 2/6, \ldots, 6/6, \ldots, 0/10, 1/10, \ldots, 10/10$. These all fall into a small set of points inside $[0, 1]$:

[9]:

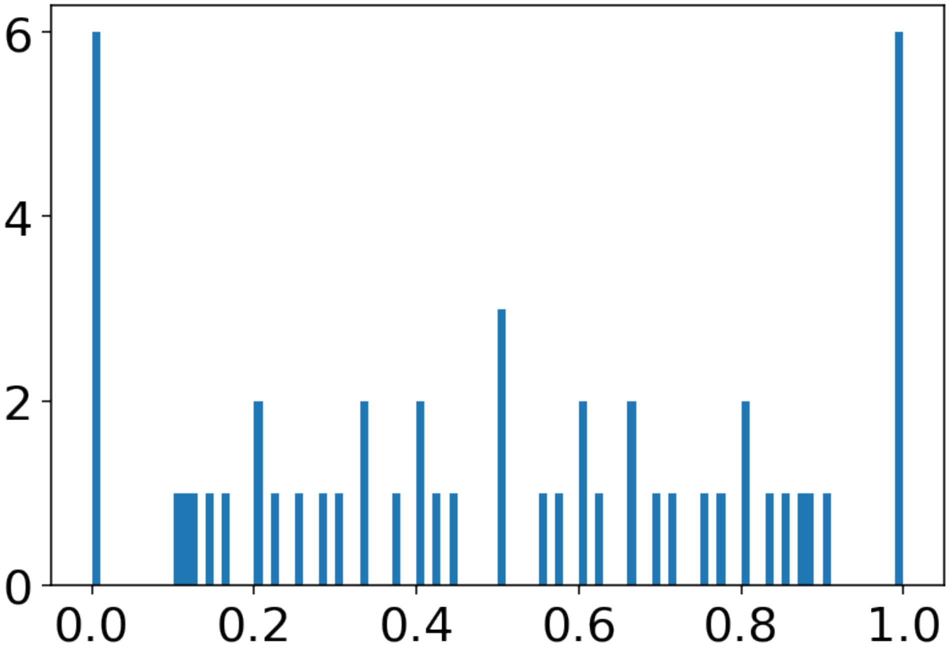
```
all_values = []
for n in range(5,10+1):
    for i in range(0,n+1):
        all_values.append(i/n)

plt.hist(all_values, bins=10);
```



And if we narrow our bins by using more of them:

[10]:

```
plt.hist(all_values, bins=100);
```



So there's a pattern there, but it is just due to (i) the definintion of the variable as a rational number and (ii) the counts in the numerator and denominator of the variable do not span a wide range. This leads to a **strong discretization effect**.

- It may be worth only looking at the numerator, or examining the numerator (# of sick friends) only, but **conditioned** on the denominator (total # of friends): $P(N_{sick} = n \mid n_{friends})$.

Choosing the number of bins

When using a histogram as a "data microscope," the number of bins (or alternatively, the width of the bins) act as a resolution setting for how sharply the histogram resolves the underlying distribution. The more data points are available, the more bins can be used, and many narrow bins gives a higher resolution for the curve. However, if there are too few data points, narrow bins will be very noisy. When there are too few bins, we miss the concentration of data at very small values.

So what's the right number of bins?

The "ideal" number of bins relates to the number of data points in the sample and the size of the domain of the data. Two **rules of thumb** are used for this:

- **Freedman-Diaconis (FD) estimator** The width of a bin h is

  $$h = 2\,\mathrm{IQR}\ /\ n^{1/3}$$

  where IQR is the interquartile range of the data and n is the number of data points (Freedman, 1981).

- **Sturges rule** The number of bins n_h is given by

$$n_h = \lceil \log_2 n \rceil + 1$$

(Sturges, 1926).

Sturges is the default for R, but its derivation makes normality assumptions.

It is also common to **combine** these rules together, which is done by the `hist(data,bins='auto', ...)` option used by matplotlib and numpy. From the `numpy.histogram` documentation:

```
`Auto' (maximum of the `Sturges' and `FD' estimators)

        A compromise to get a good value. For small datasets the Sturges
        value will usually be chosen, while larger datasets will usually
        default to FD.  Avoids the overly conservative behavior of FD
        and Sturges for small and large datasets, respectively.
        Switchover point is usually $n \approx 1000$.
```

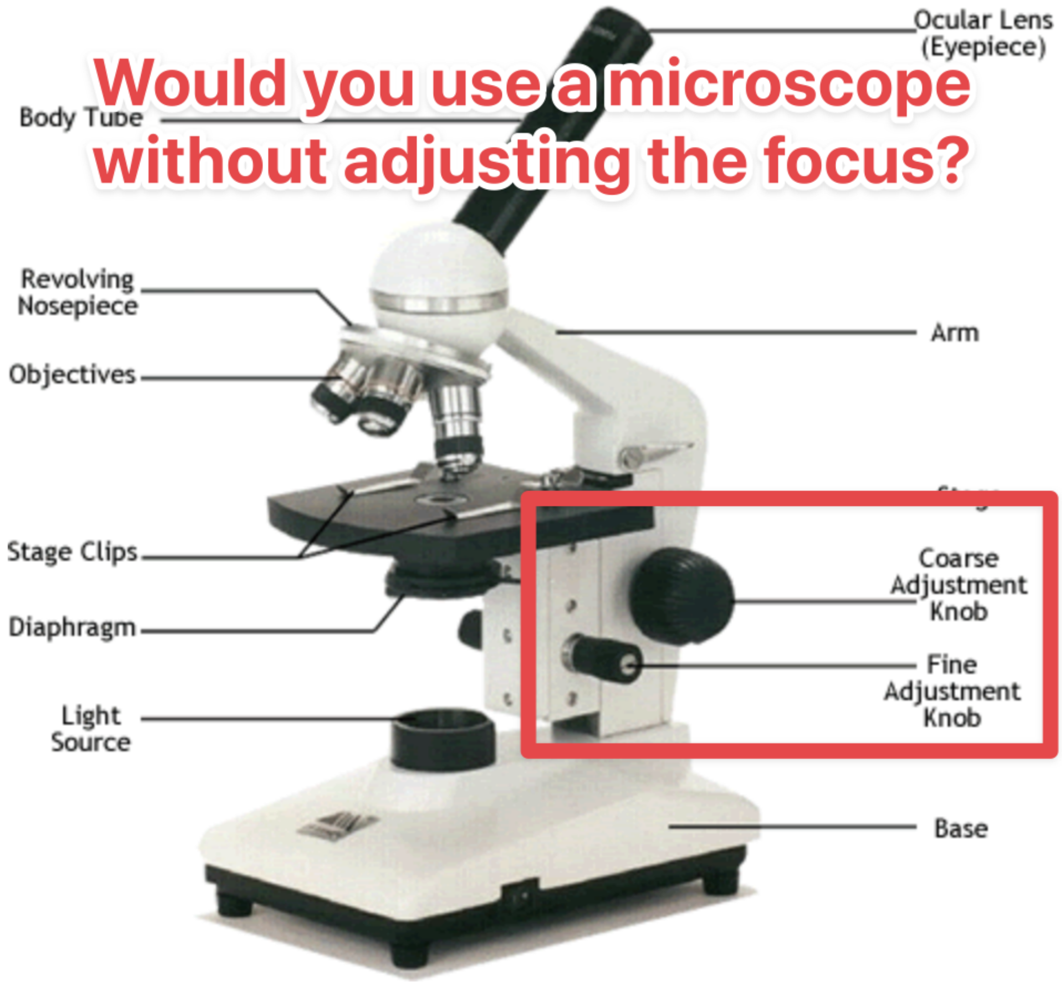
All else being equal, `bins='auto'` is a good starting point when making a histogram.

Course objective: never use histograms without *DIALING IN* an appropriate number of bins. If all else fails, try `bins='auto'`!

[39]:

```
Image("focus-microscope-histogram-bins.png", width=300)
```

[39]:



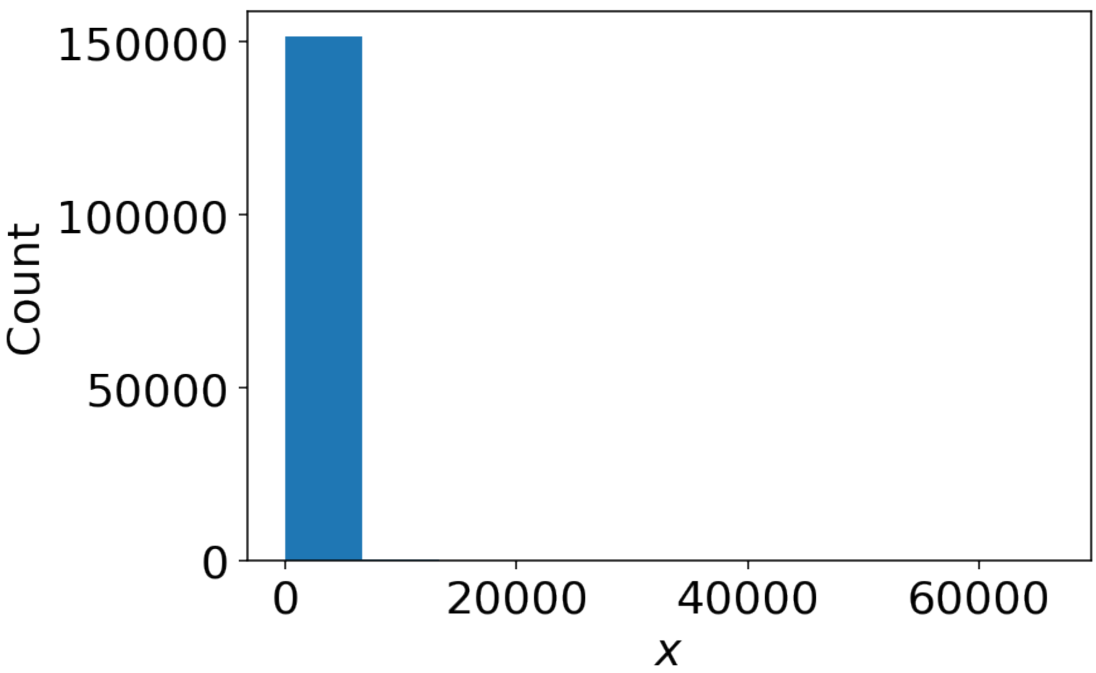
“Broad” data and variable-width bins

One common problem we encounter with data is very **broadly distributed data**, meaning the values are not concentrated around a central value but are spread over **many orders of magnitude**.

If we try to bin these data without accounting for that, we are going to have a bad time.

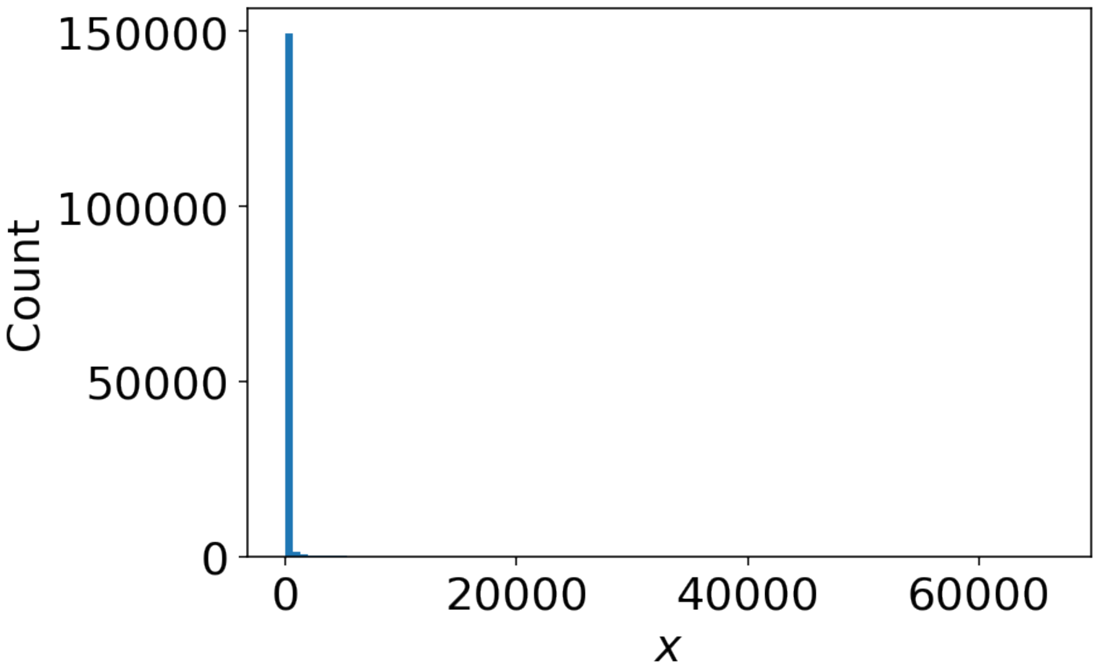
[11]:

```
data = []
for line in open("7impact_counts.txt"):
    data.append(float(line))

plt.hist(data)
plt.xlabel("$x$")
plt.ylabel("Count");
```



Weird... Let's do more bins!

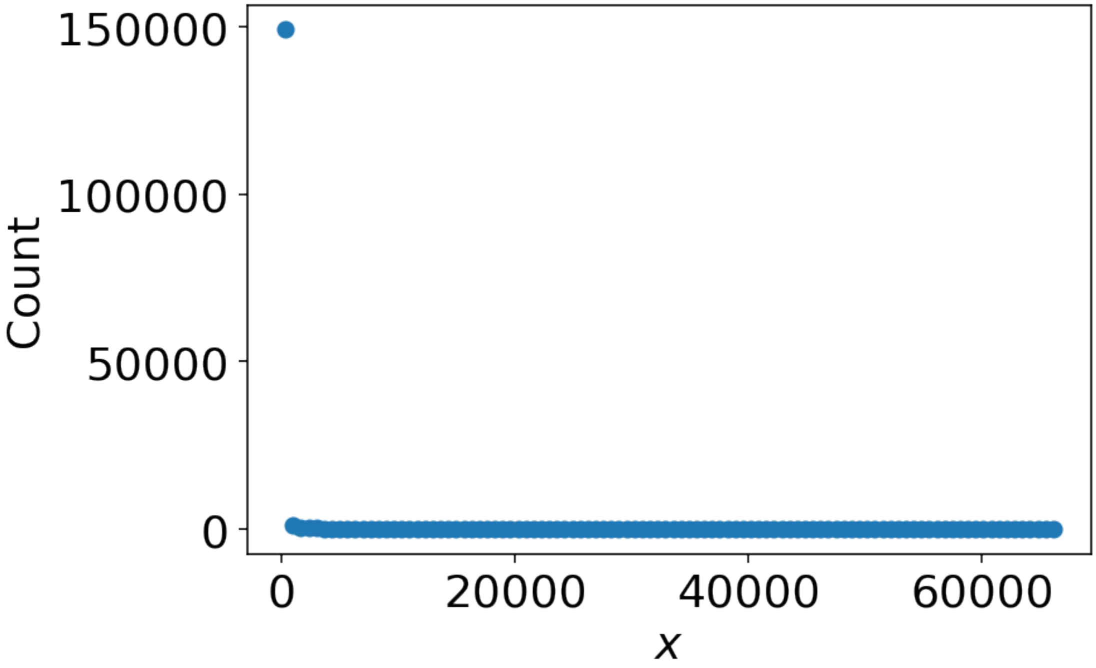
```
[12]: plt.hist(data, bins=100)
      plt.xlabel("$x$"); plt.ylabel("Count");
```



Really weird

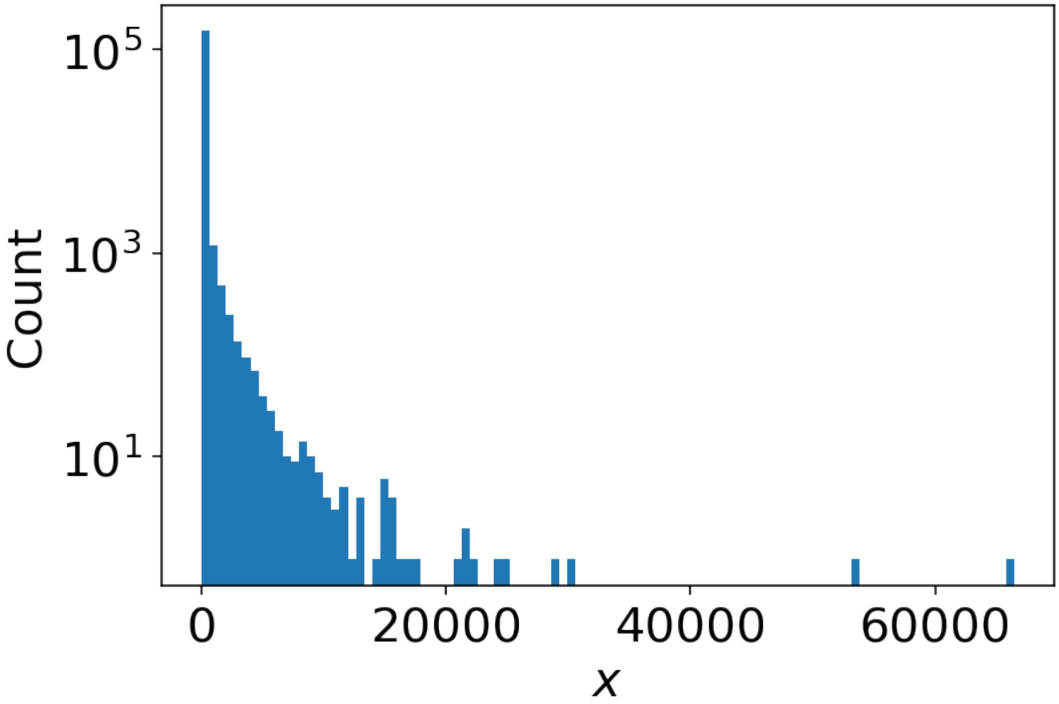
- As a check, let's plot the raw bin information (bin centers and heights) as points, instead of relying on the plotting software drawing boxes for us (maybe there's a bug?)

[13]:
```
bin_counts, bin_edges = np.histogram(data, bins=100)
bin_centers = (bin_edges[1:] + bin_edges[:-1])/2

plt.plot(bin_centers,bin_counts, 'o')
plt.xlabel("$x$"); plt.ylabel("Count");
```



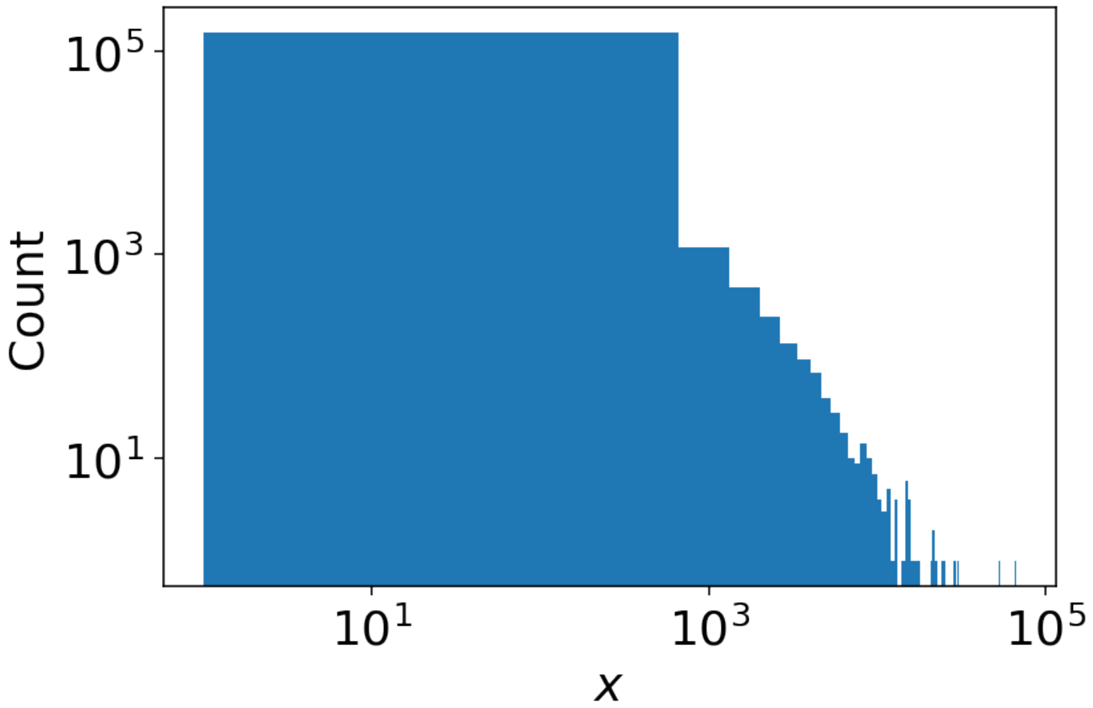
OK, so almost all the data are falling in the first bin or so. Let's emphasize this with a **logarithmic scale**:

[14]:
```
plt.hist(data, bins=100)
plt.yscale('log')
plt.xlabel("$x$"); plt.ylabel("Count");
```



Or a **double** log-scale:

```
[15]: plt.hist(data, bins=100)
      plt.xscale('log')
      plt.yscale('log')
      plt.xlabel("$x$"); plt.ylabel("Count");
```



Whoa! What happened?

Here's a binning strategy that may work well for "broad" data:

[16]:
```
#np.logspace?
print(np.logspace.__doc__[:681]) # print docstring, usually use np.logspace? in ipython
```

```
    Return numbers spaced evenly on a log scale.

    In linear space, the sequence starts at ``base ** start``
    (`base` to the power of `start`) and ends with ``base ** stop``
    (see `endpoint` below).

    .. versionchanged:: 1.16.0
        Non-scalar `start` and `stop` are now supported.

    Parameters
    ----------
    start : array_like
        ``base ** start`` is the starting value of the sequence.
    stop : array_like
        ``base ** stop`` is the final value of the sequence, unless `endpoint`
        is False.  In that case, ``num + 1`` values are spaced over the
        interval in log-space, of which all but the last (a sequence of
        length `num
```

Let's use this for a binning:

(this vector defines the edges, the endpoints, of each bin.)

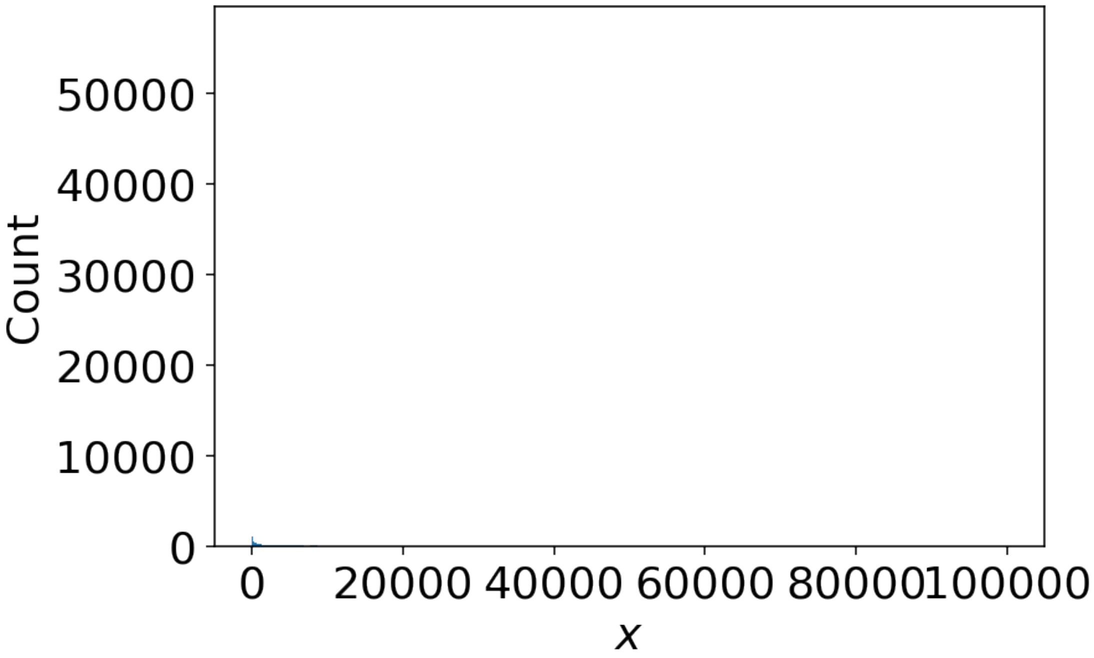
[17]:

```
log_bin_edges = np.logspace(0,5,100) # <-- bins go from 10^0 to 10^5
print(log_bin_edges)
```

```
[1.00000000e+00 1.12332403e+00 1.26185688e+00 1.41747416e+00
 1.59228279e+00 1.78864953e+00 2.00923300e+00 2.25701972e+00
 2.53536449e+00 2.84803587e+00 3.19926714e+00 3.59381366e+00
 4.03701726e+00 4.53487851e+00 5.09413801e+00 5.72236766e+00
 6.42807312e+00 7.22080902e+00 8.11130831e+00 9.11162756e+00
 1.02353102e+01 1.14975700e+01 1.29154967e+01 1.45082878e+01
 1.62975083e+01 1.83073828e+01 2.05651231e+01 2.31012970e+01
 2.59502421e+01 2.91505306e+01 3.27454916e+01 3.67837977e+01
 4.13201240e+01 4.64158883e+01 5.21400829e+01 5.85702082e+01
 6.57933225e+01 7.39072203e+01 8.30217568e+01 9.32603347e+01
 1.04761575e+02 1.17681195e+02 1.32194115e+02 1.48496826e+02
 1.66810054e+02 1.87381742e+02 2.10490414e+02 2.36448941e+02
 2.65608778e+02 2.98364724e+02 3.35160265e+02 3.76493581e+02
 4.22924287e+02 4.75081016e+02 5.33669923e+02 5.99484250e+02
 6.73415066e+02 7.56463328e+02 8.49753436e+02 9.54548457e+02
 1.07226722e+03 1.20450354e+03 1.35304777e+03 1.51991108e+03
 1.70735265e+03 1.91791026e+03 2.15443469e+03 2.42012826e+03
 2.71858824e+03 3.05385551e+03 3.43046929e+03 3.85352859e+03
 4.32876128e+03 4.86260158e+03 5.46227722e+03 6.13590727e+03
 6.89261210e+03 7.74263683e+03 8.69749003e+03 9.77009957e+03
 1.09749877e+04 1.23284674e+04 1.38488637e+04 1.55567614e+04
 1.74752840e+04 1.96304065e+04 2.20513074e+04 2.47707636e+04
 2.78255940e+04 3.12571585e+04 3.51119173e+04 3.94420606e+04
 4.43062146e+04 4.97702356e+04 5.59081018e+04 6.28029144e+04
 7.05480231e+04 7.92482898e+04 8.90215085e+04 1.00000000e+05]
```

[18]:

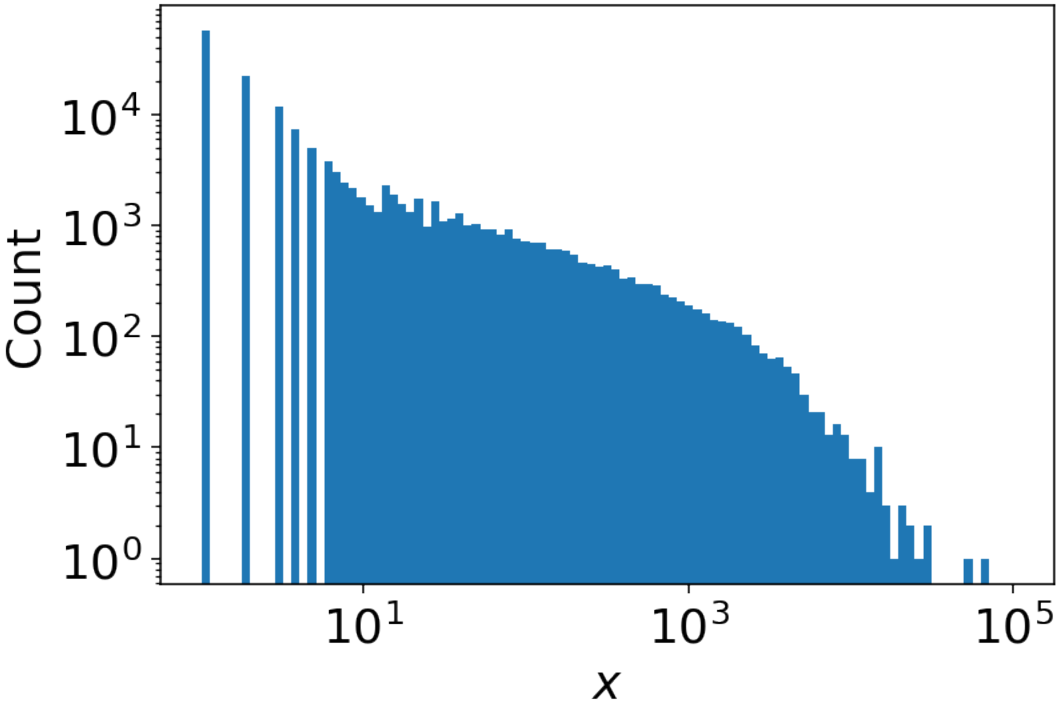
```
plt.hist(data, bins=log_bin_edges)
plt.xlabel("$x$")
plt.ylabel("Count")
plt.show()
```



What???? Uh oh!

Let's look at this on a log-log scale:

```
[19]: plt.hist(data, bins=log_bin_edges)

plt.xscale('log')
plt.yscale('log')

plt.xlabel("$x$")
plt.ylabel("Count")
plt.show()
```



What's up with the stripes on the left? And why couldn't we see them in the previous plot, they're pretty tall?

Let's take a look at data (a crucial step we should always do):

[20]:
```
print(data[:20])
```

```
[2.0, 1.0, 2.0, 3.0, 3.0, 13.0, 1.0, 2.0, 4.0, 1.0, 2854.0, 1.0, 2.0, 3.0, 11.0,
2.0, 2.0, 1.0, 2.0, 1.0]
```

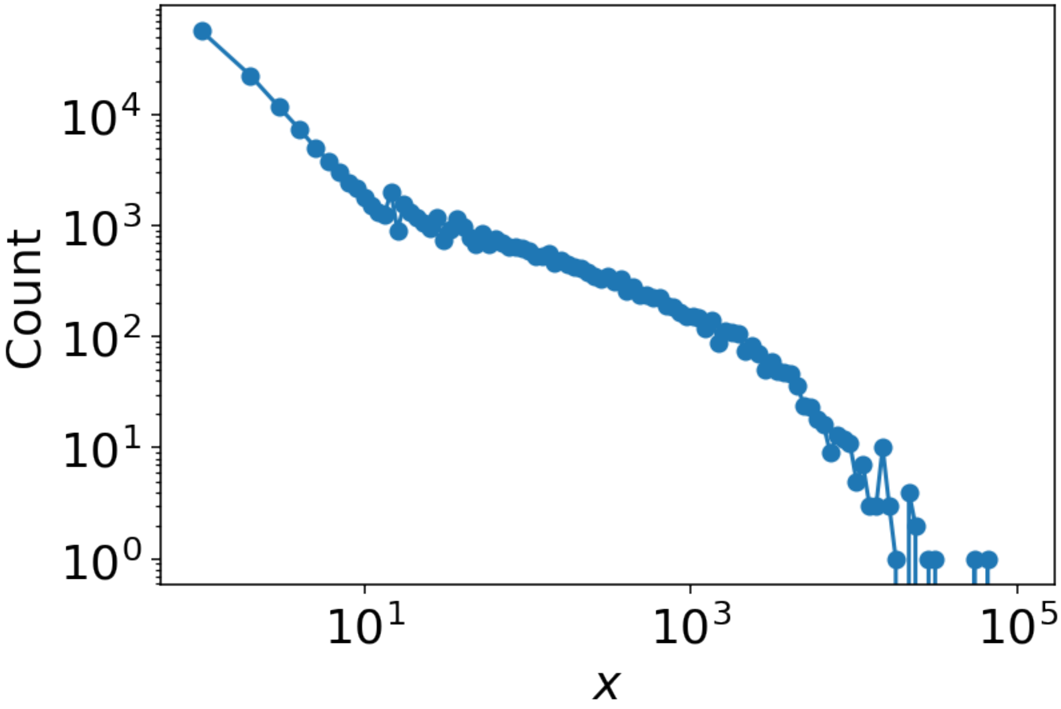
Ah!, we have integer-valued data (we should have known this already—and it may mean we should be counting not binning. For the left-most bins, from $x = 10^0 = 1$ to $x = 10^1 = 10$, any bins between integers will be empty.

Log binning integers

[21]:
```
def count_plus_log_bins(right_exp, num_bins):
    """Binning vector that bin on integers below 10, then use num_bins
    logarithmically spaced bins from 10 to 10**right_exp. Useful for averaging
    heavy-tailed X,Y data, where X are integers.
    """
    count_bins = np.arange(1,11)-0.5
    log_bins = np.logspace(np.log10(10.5), right_exp, num_bins)
    return np.hstack([count_bins, log_bins])

def get_centers(bin_edges):
    return 0.5*(bin_edges[1:]+ bin_edges[:-1])
```

```
new_bins = count_plus_log_bins(5, 100)
print(new_bins[:20], "...")
bin_counts, bin_edges = np.histogram(data, bins=new_bins)
bin_centers = get_centers(bin_edges)

plt.loglog(bin_centers, bin_counts, 'o-')
plt.xlabel("$x$")
plt.ylabel("Count")
plt.show()
```

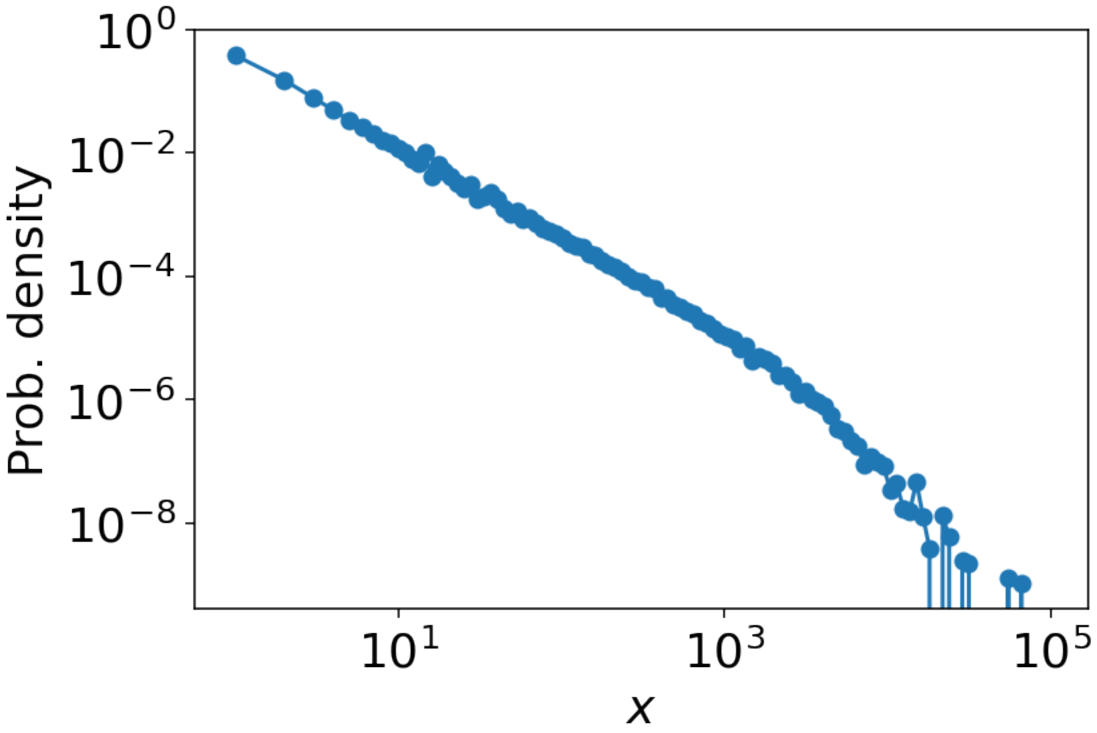
```
[ 0.5         1.5         2.5         3.5         4.5         5.5
  6.5         7.5         8.5         9.5        10.5        11.51805919
 12.63482739 13.85987522 15.20370126 16.677822   18.29487058 20.06870498
 22.01452685 24.14901175] ...
```



We’ve changed how the bin widths change as $x$ increases. They are constant until $x = 10$, then the bins start to get wider and wider in such a way that they appear linearly spaced on a log x-axis.

What is that cusp-looking thing at $x = 10$? That’s when we change how bins are defined... Is that a coincidence? A feature of the data that occurs around values of 10? Or is it an artifact of our binning?

[22]:
```
binCnts, binEdgs = np.histogram(data, bins=new_bins, density=True)
#                                                    ^^^^^^^^^^^^

bin_centers = get_centers(binEdgs)

plt.loglog(bin_centers, binCnts, 'o-')
plt.xlabel("$x$")
plt.ylabel("Prob. density")
plt.show()
```



Log bins are tricky. Remember to normalize them so the AREA sums to 1.

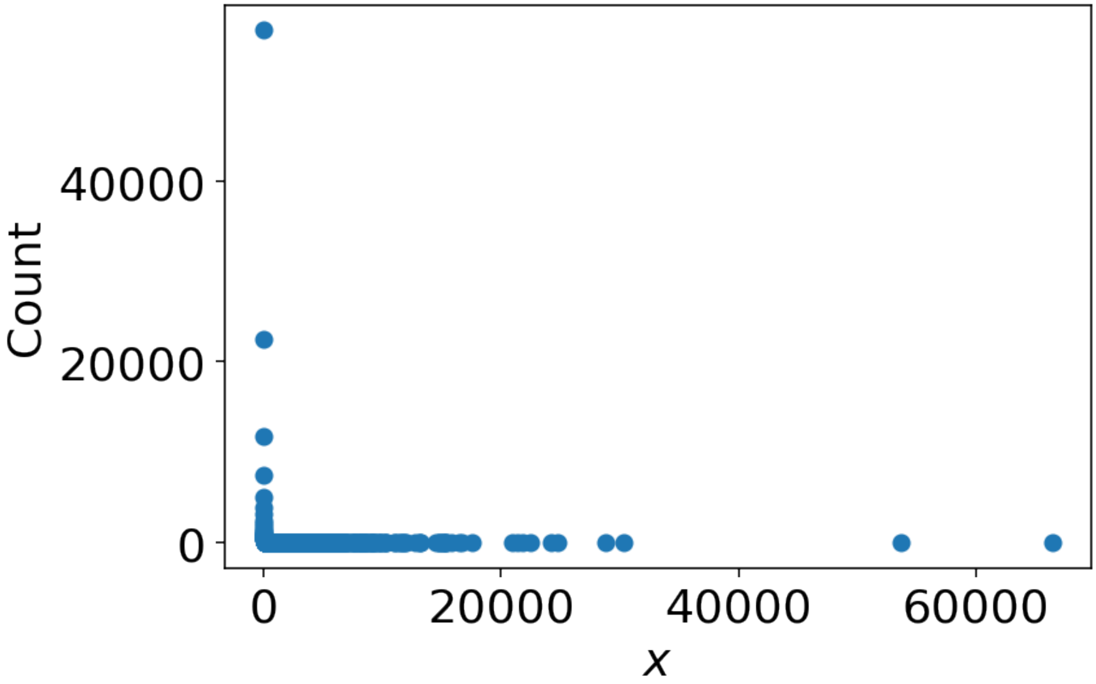
- If bins are not constant-width, all sorts of problems can creep in when unnormalized

If the data are integers can we just count? Yes! Recall this function from several weeks ago:

[23]:
```
def count_integers(L):
    """Takes a list L (of integers) and counts how many times each integer occurred.
    Returns two lists, the sorted values within L and the corresponding number
    of times each value occurred. This is useful for plotting.
    """

    value2count = {} # or use collections.Counter()
    for v in L:
        try:
            value2count[v] += 1
        except KeyError: # never seen this v before, so initialize it
            value2count[v] = 1

    list_values = sorted(value2count.keys())
    list_counts = []
    for v in list_values:
        list_counts.append(value2count[v])
    return list_values, list_counts
```

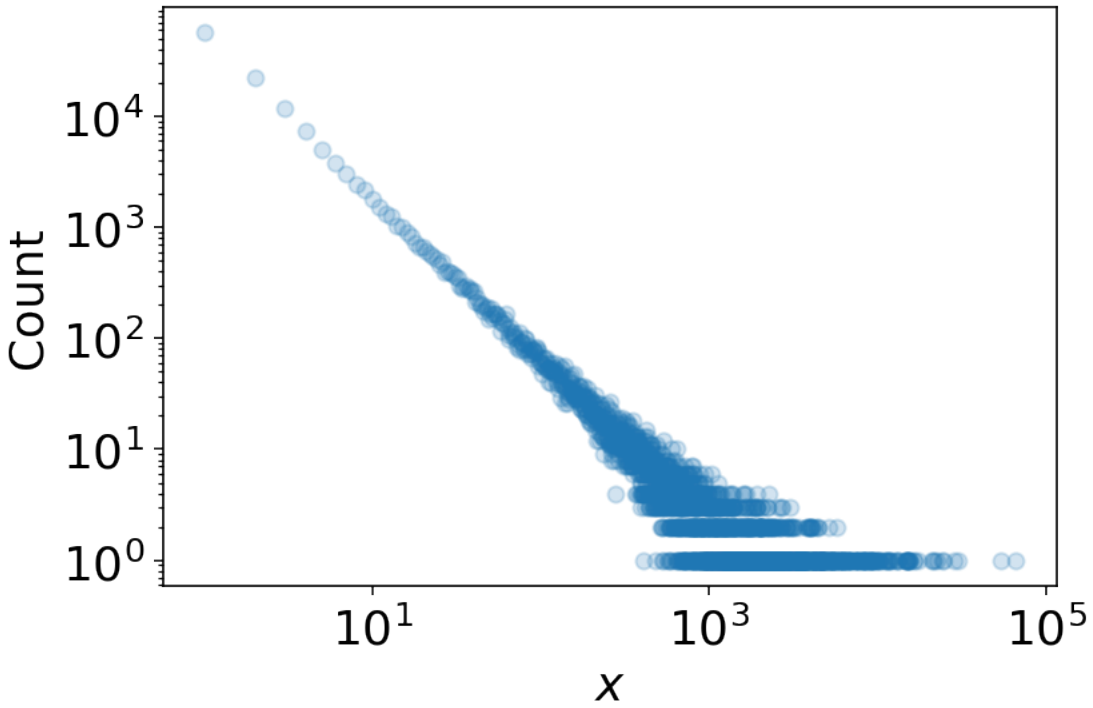
Let's apply this to data:

```
[24]: values, counts = count_integers(data)

      plt.plot(values, counts, 'o')
      plt.xlabel("$x$")
      plt.ylabel("Count")
      plt.show()
```



Looks about the same. Let's go for the log scale:

```
[25]: values, counts = count_integers(data)

      plt.loglog(values, counts, 'o', alpha=0.2)
      #                               ^^^^^^^^^
      #                               more readable?
      plt.xlabel("$x$")
      plt.ylabel("Count")
      plt.show()
```



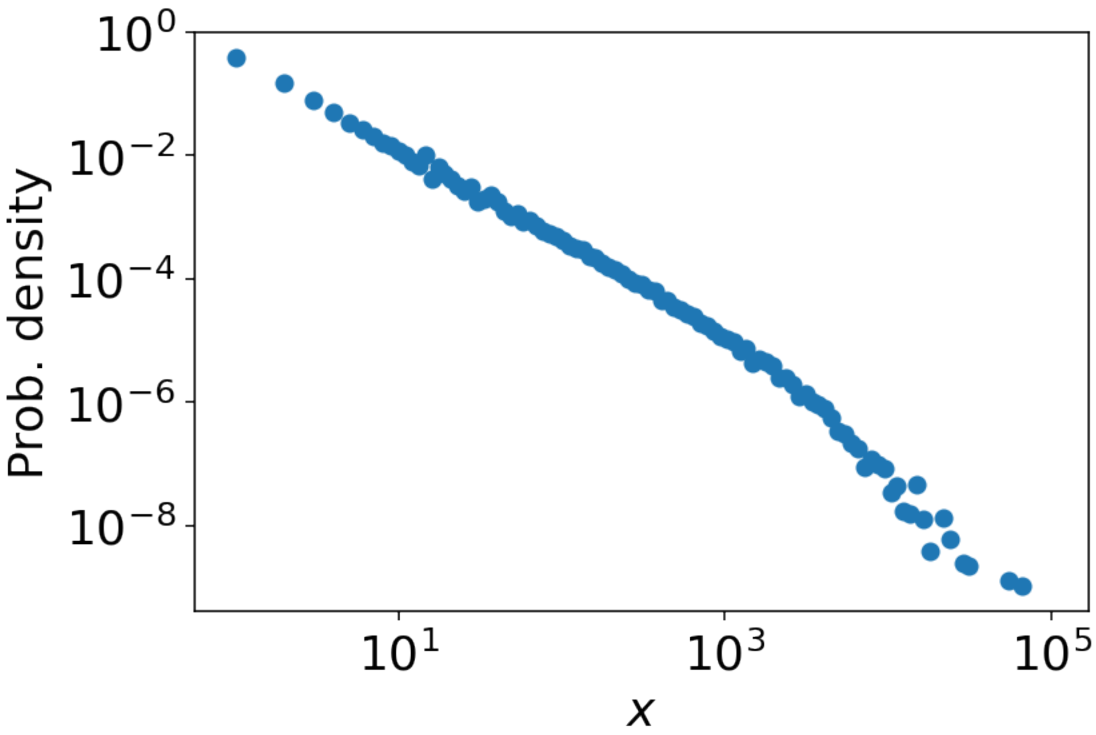
(Should we normalize this too?)

Look at how **drastically** different the same data are, just by changing the scale of the axes!

Here's the log-binned data again for comparison:

[26]:

```
plt.loglog(bin_centers, binCnts, 'o')

plt.xlabel("$x$")
plt.ylabel("Prob. density")
plt.show()
```



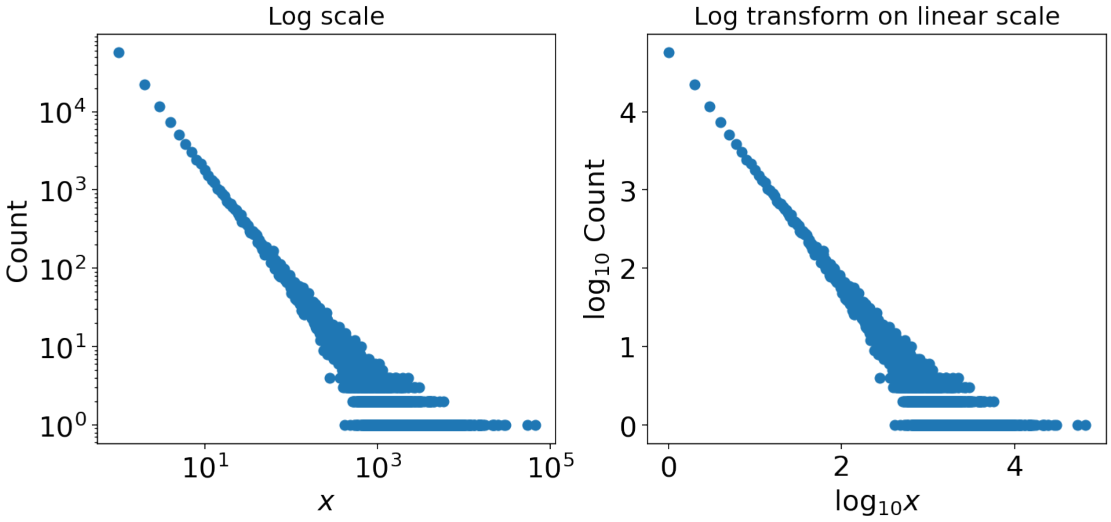
The x-axes are identical in both cases, and the y-axes are very different. But these are both views of the same data. Which one is correct?

Brief aside: log scale vs. log transform:

Plotting the original data with a logarithmic scale is the same as taking the log of the data (log transforming it) and plotting that on a linear scale:

[27]:

```
fig, axes = plt.subplots(1,2, figsize=(6.4*1.8,4.8)) # side-by-side figures

# plot x,y with double-log scale:

plt.sca(axes[0]) # sca = set current axes
plt.loglog(values, counts, 'o')

plt.xlabel("$x$")
plt.ylabel("Count")
plt.title("Log scale", fontsize=16)


# log transform x,y, plot as usual
plt.sca(axes[1]) # sca = set current axes
plt.plot(np.log10(values), np.log10(counts), 'o')

plt.xlabel("$\log_{10} x$")
plt.ylabel("$\log_{10}$ Count")
plt.title("Log transform on linear scale", fontsize=16)
plt.show()
```



Same thing! (Be sure to label the axes properly)

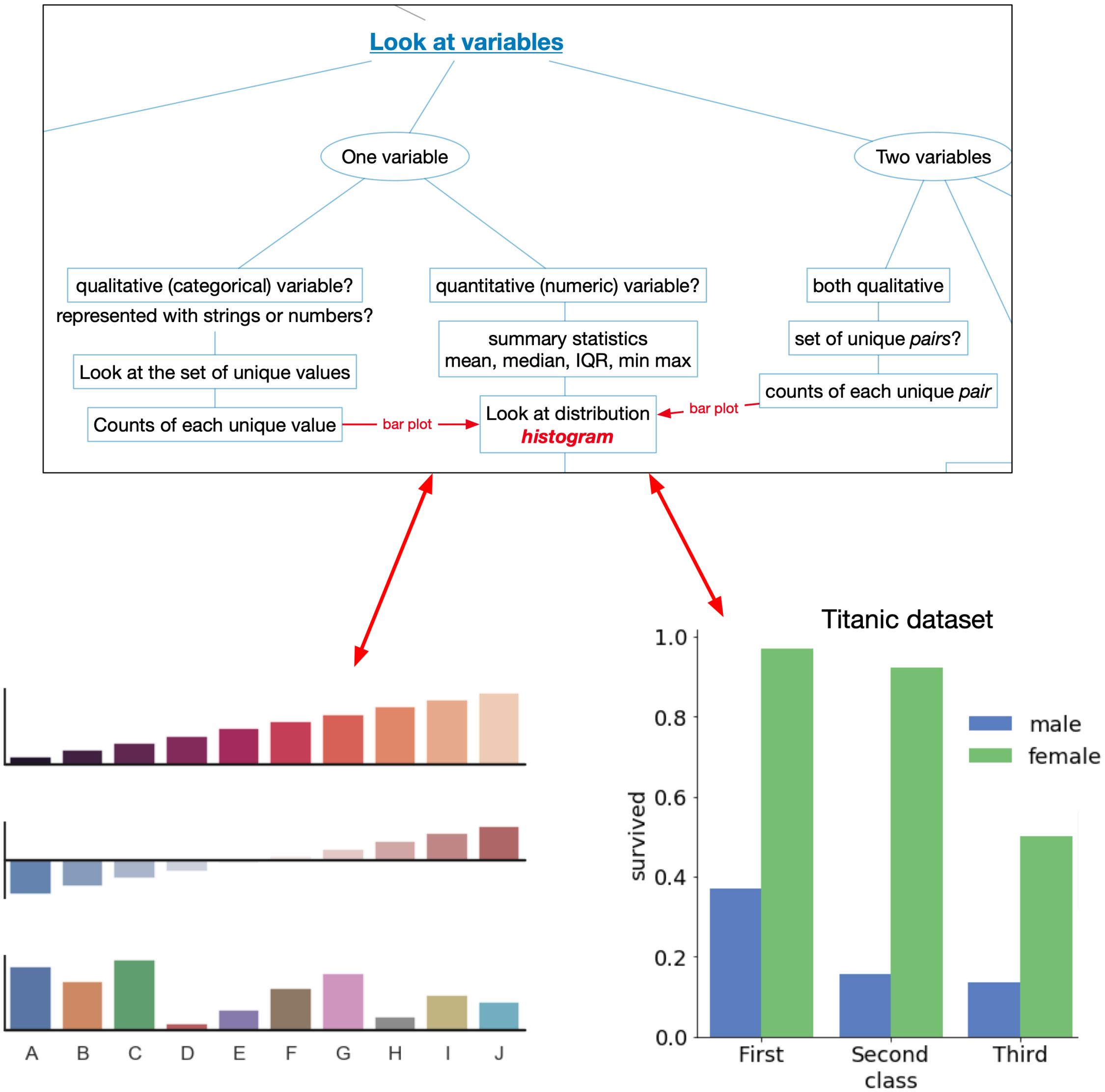
Generalizing histograms

Histograms can be **generalized** in a number of ways, some of which we will cover soon. * Common to *apply a function* to the data in each bin that is not just a count!

I also think of **bar charts/bar plots** as histogram-like visualizations (depending on what is being computed):

[28]:
```
Image("figures/barplots-as-histograms.png", width=700)
```

[28]:



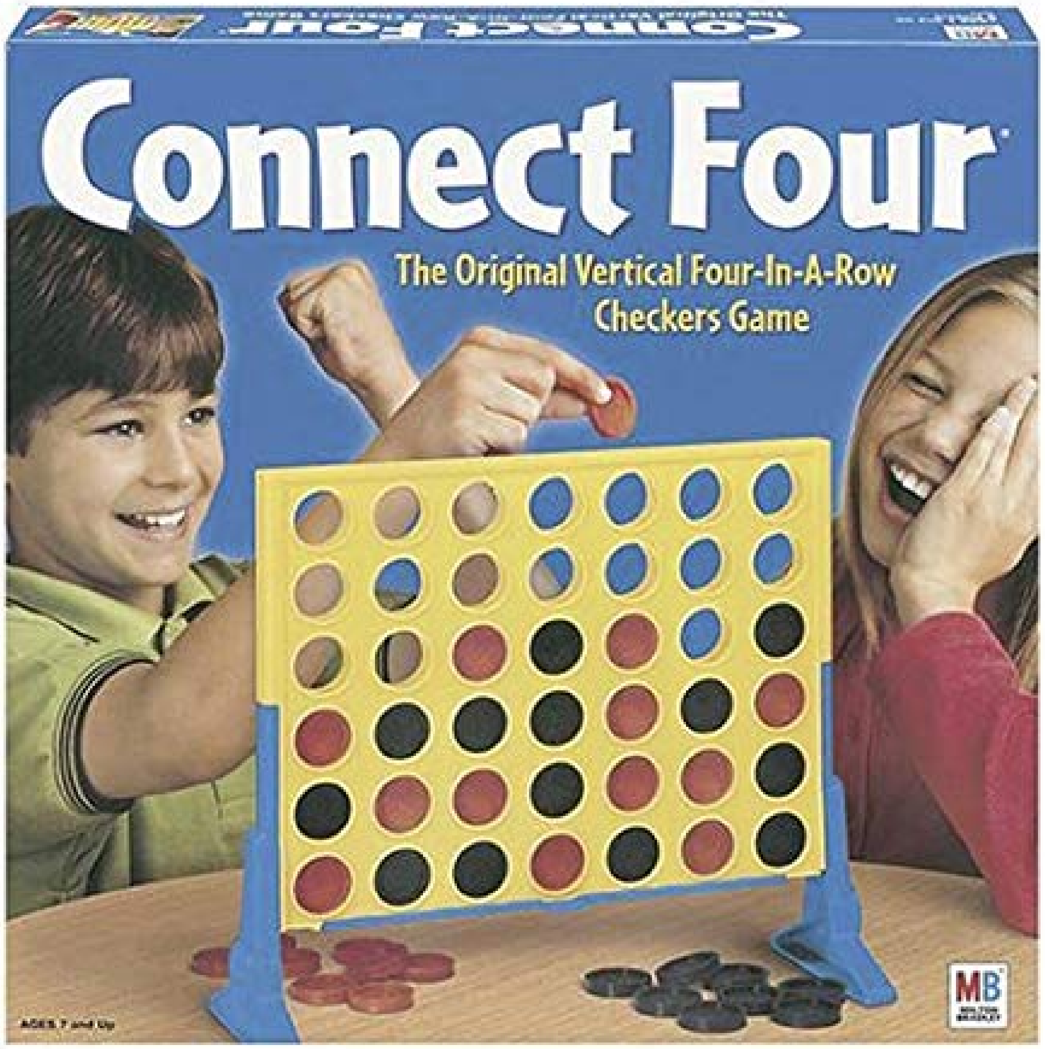
Kernel density estimation (KDE)

Here's a powerful tool to make smoother curves that are often (though not always) even more accurate representation of the underlying distribution).

Anyone remember this:

```
[29]: Image("figures/conn-four.jpg", width=400)
```

[29]:



A histogram is sort of like dropping a single little box down a slot for each data point. This box has width equal to the width of the bin and height equal to 1 / (number-of-datapoints × bin-width).

With that in mind:

- Why drop boxes?
- Why force the boxes to align in columns (the bins)?

Assume a particular data i has value x_i.

1. It's possible there's a little **noise** or **error** in this number.
2. Likewise, if our experiment was bigger, we would expect **more datapoints** near this value x_i

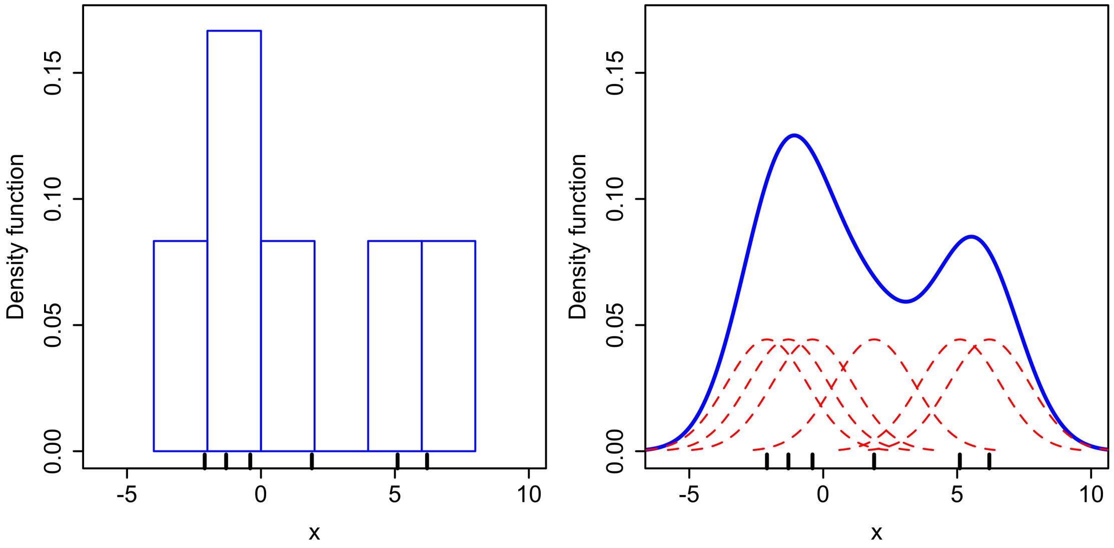
We can represent this noise and these extra data by assuming (really, *pretending*) that the true x_i was actually drawn from a small distribution of values near the observed x_i.

This is called a probability *kernel*. We can choose any small distribution to put near x_i, including a box, but the natural one is a **normal distribution** with mean x_i.

Let's look at a picture:

[30]:
```
Image("figures/Comparison_of_1D_histogram_and_KDE.png", width=600)
```

[30]:



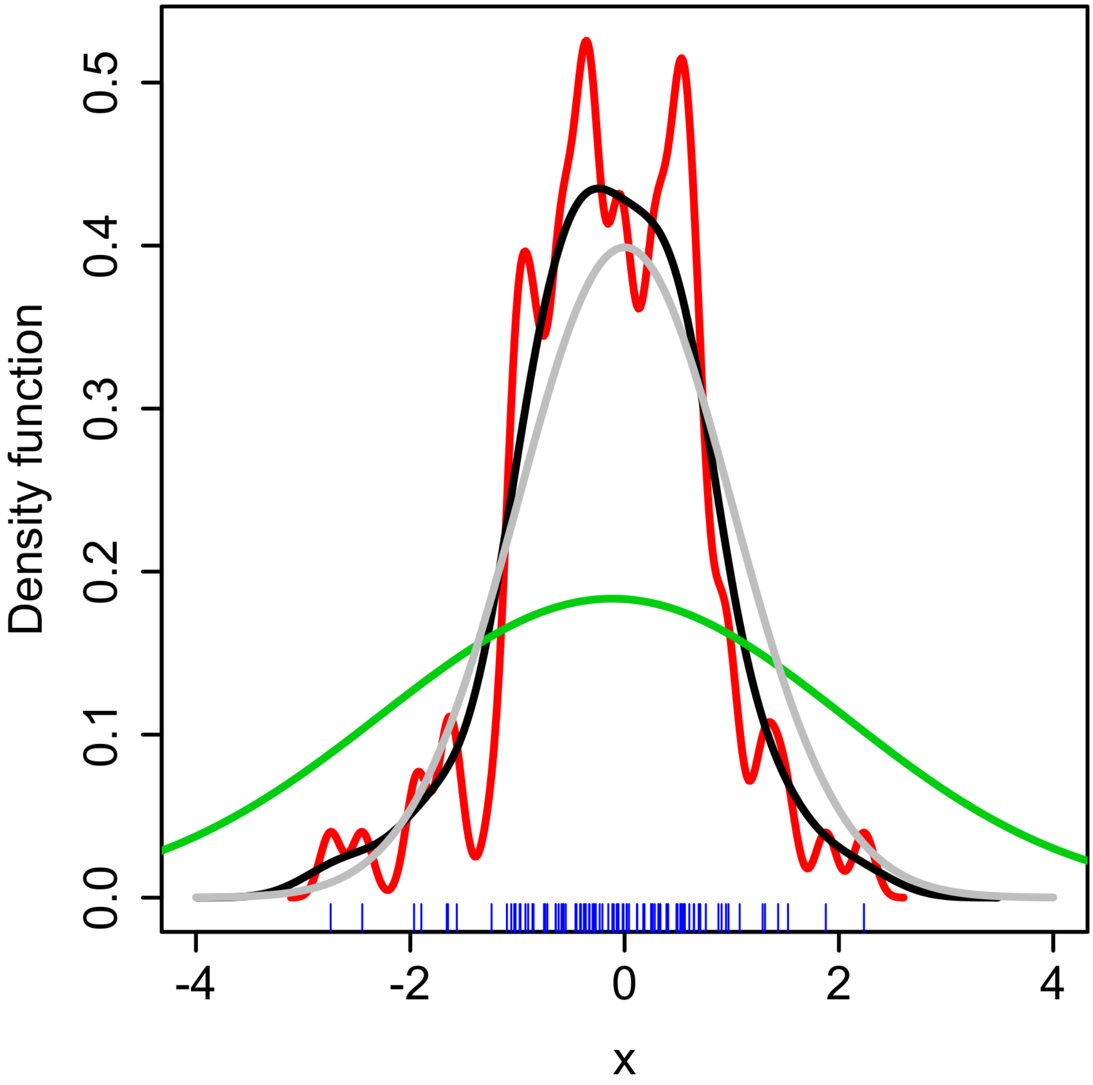
We see the individual kernels in red and their sum in blue. The blue curve looks **much nicer** than the histogram.

The *mean* of each kernel is easy to pick, it's just the value of the corresponding data point. But a gaussian kernel also has a **width**, captured by the standard deviation. How do we pick this

- In KDE this is known as **bandwidth selection** and it can strongly alter the final curve:

[31]:
```
Image("figures/Comparison_of_1D_bandwidth_selectors.png", width=400)
```

[31]:



(Via. Grey: true density (standard normal). Red: KDE with bandwidth h=0.05. Green: KDE with h=2. Black: KDE with h=0.337.)

The black curve is pretty close to the true distribution (grey).

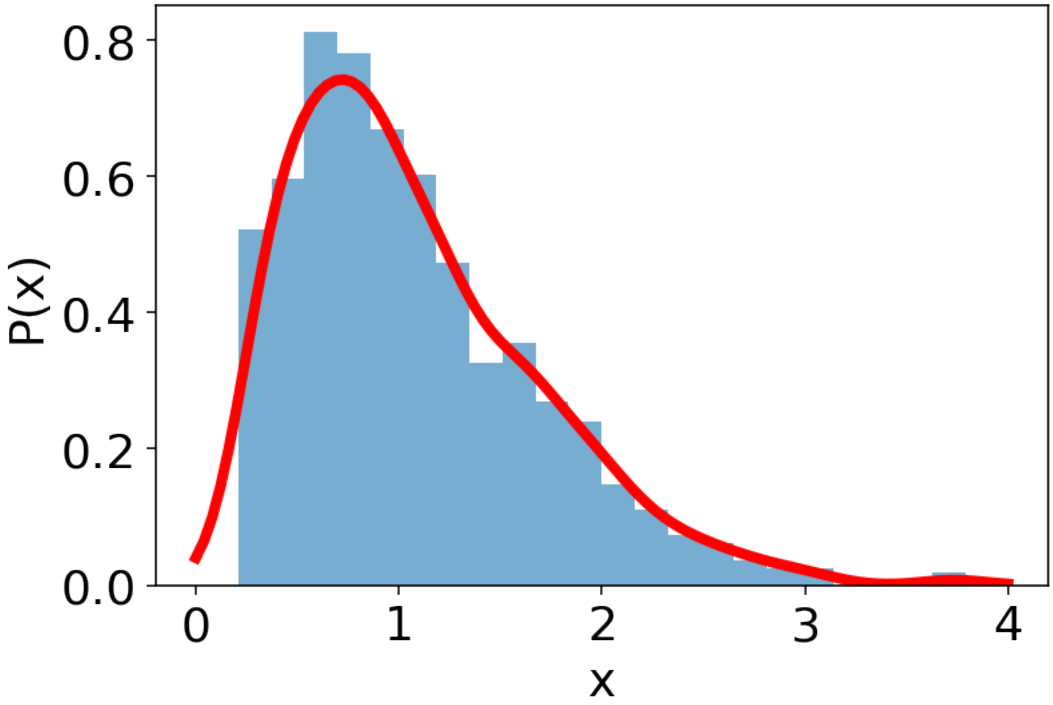
- So you can make a nice smooth picture but if you choose the bandwidth poorly it can still look jagged. Bandwidth is very similar to a histogram's binwidth: there's no free lunch.
- (If you assume the underlying distribution is normal you can compute an optimal bandwidth. This is what most scientific packages do by default)

To the Pythons!

[32]:

```
# scipy!!!
from scipy.stats.kde import gaussian_kde

data = np.random.weibull(1.5,1000)+0.2
```

```
# obtaining the KD-estimate of the Probability distribution function
# or PDF (kde_pdf is a function!)
kde_pdf = gaussian_kde( data ) # weibull data, from above...

# plotting the result
x = np.linspace(0,4,100)
plt.plot(x,kde_pdf(x),'r', linewidth=4) # distribution function
plt.hist(data,bins='auto', density=True,alpha=.6) # histogram
plt.xlabel("x")
plt.ylabel("P(x)")
plt.show()
```

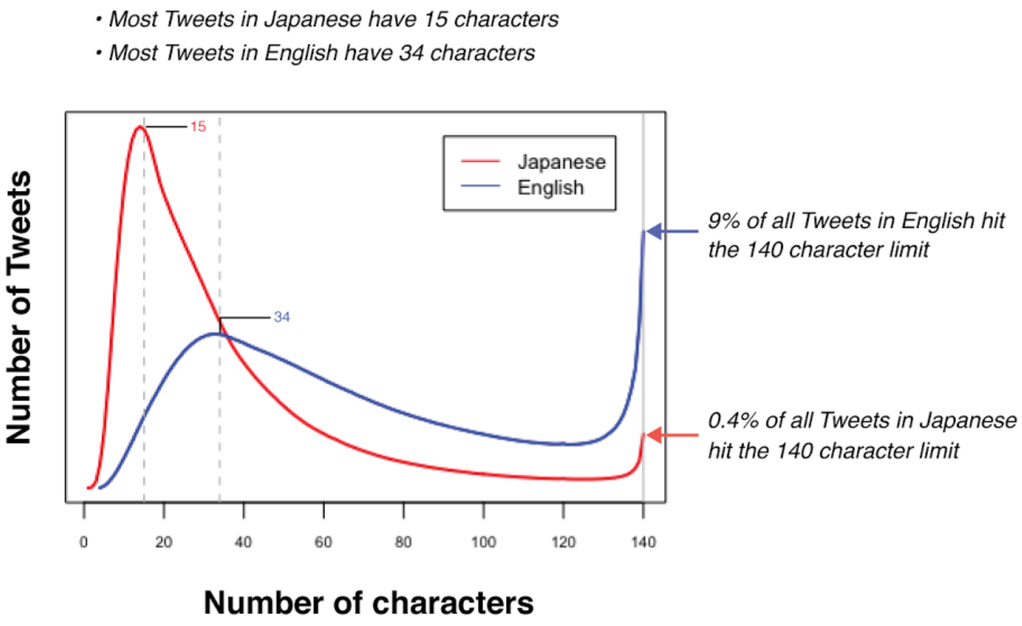


- Notice how the red curve assigns nonzero values to areas with no data. This may be unwanted, especially if the data are bound to a given region (for example, x must be between 0 and 1).
- KDE becomes useful in higher dimensions (although not too high—curse of dimensionality kicks in)

Recall the Data Science question from Lecture 1:

[33]:
```
Image("figures/more-chars-1.png", width=700)
```

[33]:



"Number of characters" is integer-valued, but these curves are nice and smooth. How? KDE! (Probably)

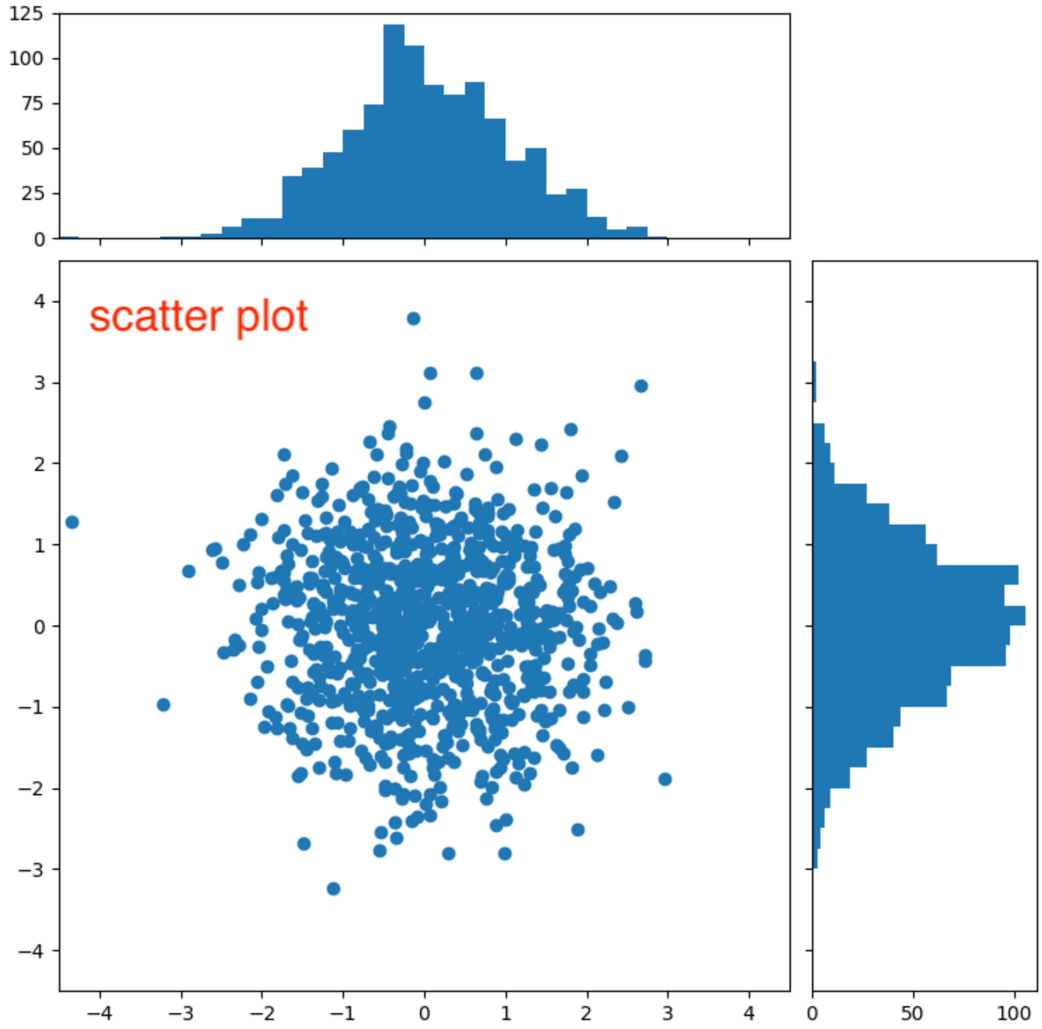
Histograms in higher dimensions

Histograms are a tool to visualize the distribution (pdf or pmf) $Pr(X)$ for sample(s) of a **single variable** X.

With **two variables** X_1, X_2 they are still useful:

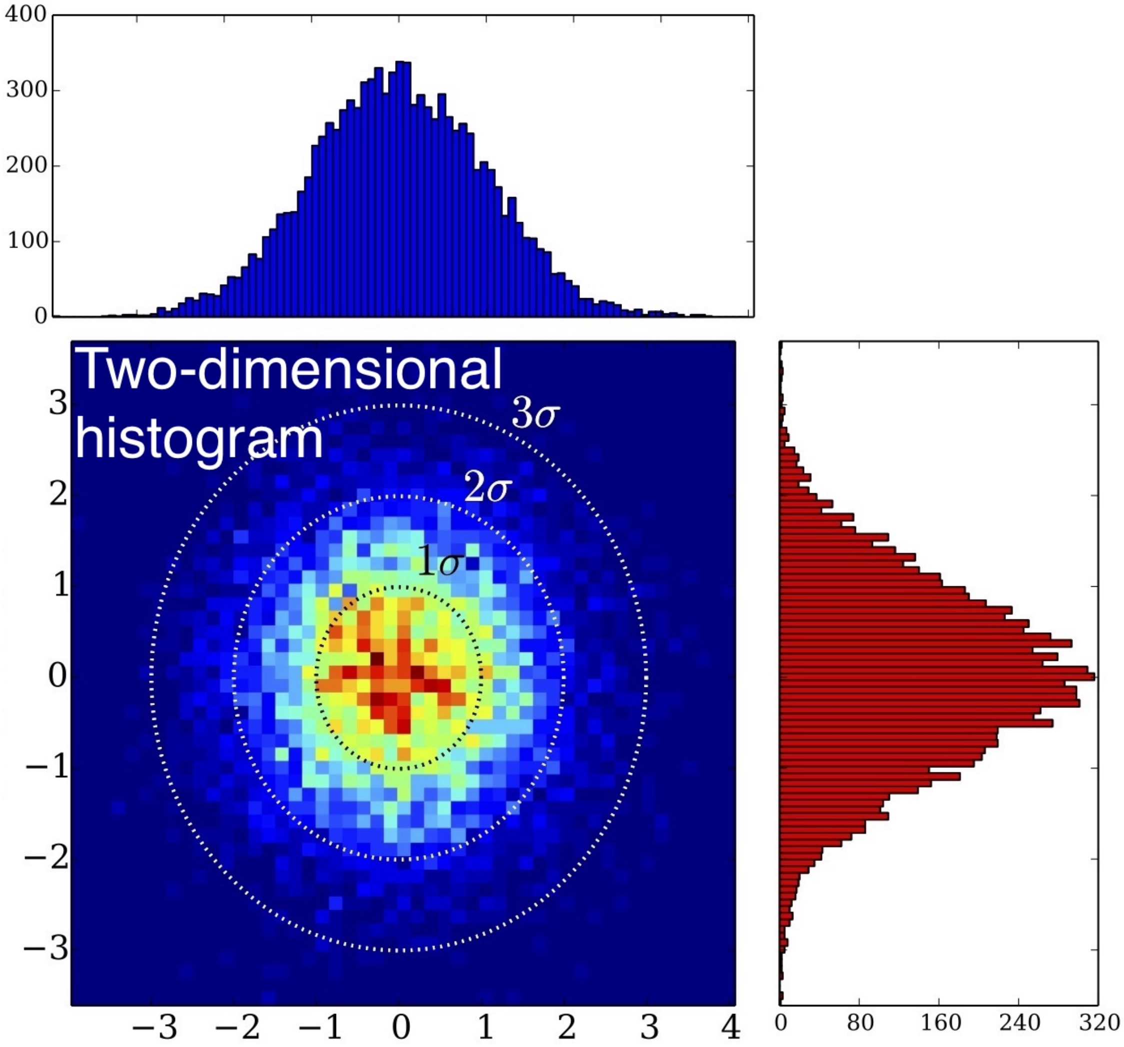
```
[34]: Image("figures/scatter_hist1.png", width=500)
```

```
[34]:
```



```
[35]: Image("figures/scatter_hist2.jpg", width=500)
```

[35]:

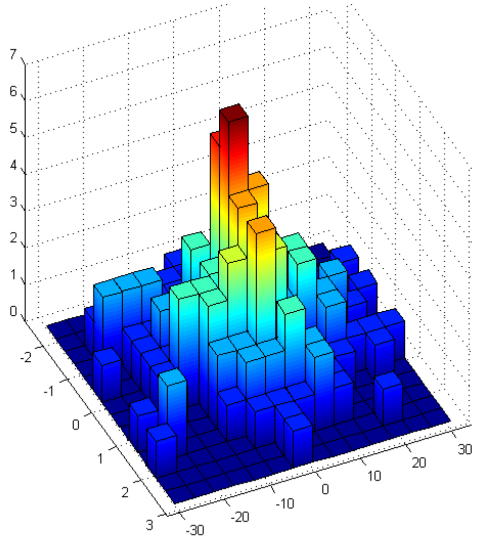


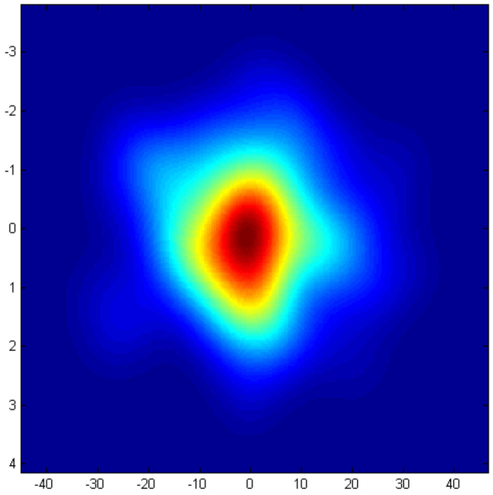
Here we see the *joint distribution* and the two *marginal distributions* of two normal variables

KDE is also useful in two dimensions (Kernel is now a 2d normal distribution.)

```
[36]: Image("figures/hist_kde.png", width=500)
```

[36]:

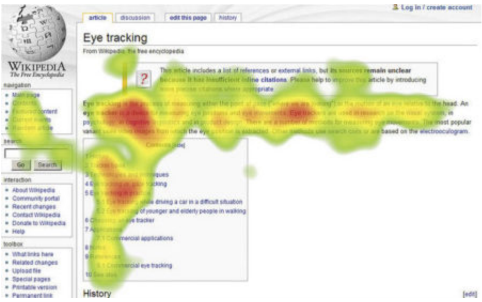




Lots of applications, for example as **heatmaps**:

[37]:
```
Image("figures/heatmap_eyetracking-e1524233535406.jpg", width=500)
```

[37]:



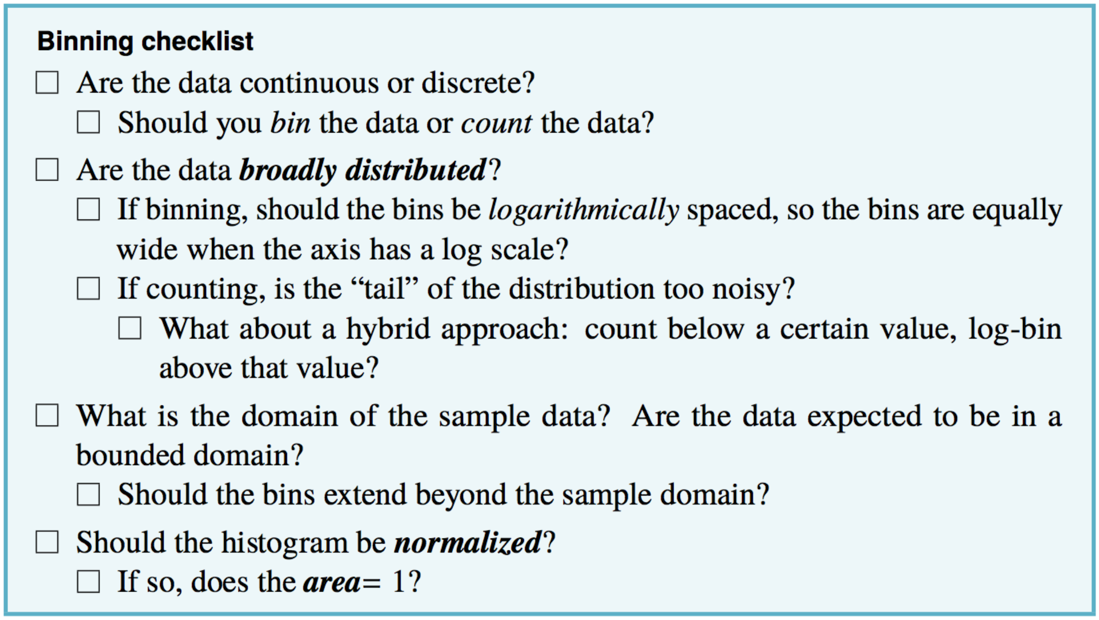
Unfortunately, for visualization purposes these techniques become **less helpful at higher dimensions**, when you have N variables of interest, finding an estimate for $Pr(X_1, X_2, \ldots, X_N)$ is challenging!

Histogram checklist

Something I've put together that I find helpful. Please use it when applying histograms during the rest of this course (and later, as you wish)!

[38]:
```
Image("figures/binning_checklist_shot.png", width=550)
```

[38]:



Binning checklist

- □ Are the data continuous or discrete?
  - □ Should you *bin* the data or *count* the data?
- □ Are the data ***broadly distributed***?
  - □ If binning, should the bins be *logarithmically* spaced, so the bins are equally wide when the axis has a log scale?
  - □ If counting, is the "tail" of the distribution too noisy?
    - □ What about a hybrid approach: count below a certain value, log-bin above that value?
- □ What is the domain of the sample data? Are the data expected to be in a bounded domain?
  - □ Should the bins extend beyond the sample domain?
- □ Should the histogram be ***normalized***?
  - □ If so, does the ***area***= 1?

Takeaways

- Starting to move from acquiring data to cleaning/filter/validating data
- Cleaning data requires exploring data.
  - Filtering data
  - Always **look** at your data - SO IMPORTANT!
- One of the most important tools for exploring data are histograms
  - histograms!
  - histograms!
  - histograms!
    - It's all about the bins!
  - Binning vs. counting
  - Histograms for "broad" data
  - Generalizing histograms to higher dimensions, functions besides counts
  - Kernel Density estimation
- Histogram checklist

Open questions:

- Are histograms useful when you can't order/rank data?
  - What abou non-numeric data?
- What about higher-dimensional data? → density estimation
- Can we be more rigorous in our use of histograms? Move beyond visualization and use the data to inform statistical models?