[1]:
```
%matplotlib inline
# make figures better for projector:
import matplotlib
font = {'size':18}
matplotlib.rc('font', **font)
#matplotlib.rc('figure', figsize=(9.0, 6.0))

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%config InlineBackend.figure_format = 'retina'
```

DS1 Lecture 10

Jim Bagrow

Last time:

1. Data cleaning:
    - Rejecting bad data, combining data, filtering and processing data
2. Histograms as data "microscopes"
    - See the distribution of the data
    - Binning is key!
    - "broad" and log-vs-linear scales

Today's plan:

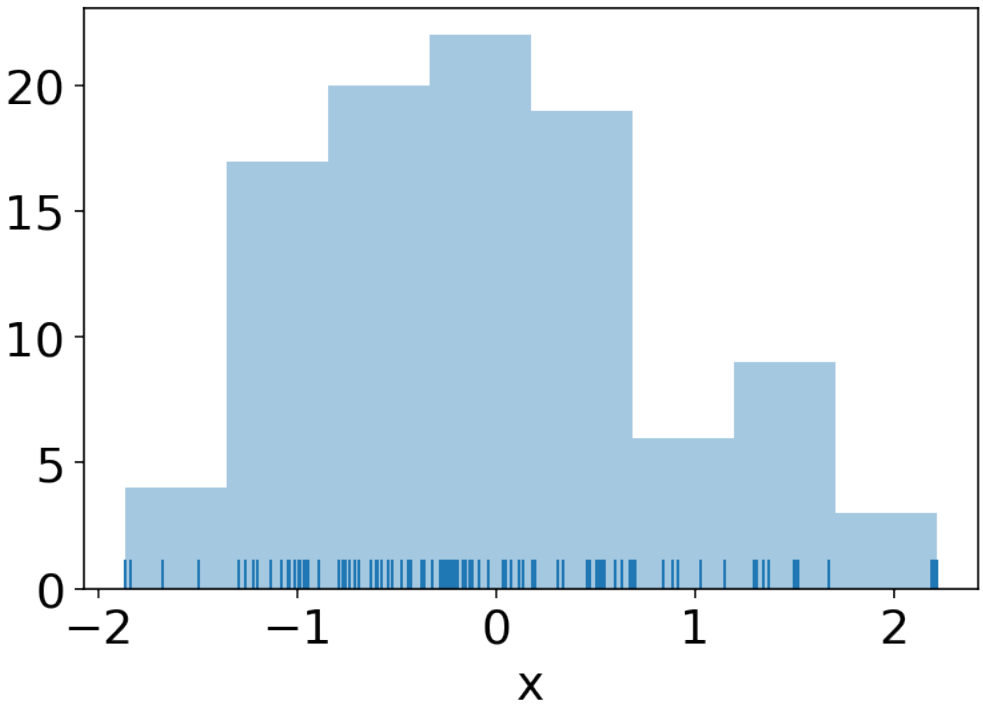
1. More on histograms
    - box plots
    - $\rightarrow$ tools for *distributions*
2. Scatter plots

Recall, last time we used histograms on some **fish mass data** to reveal that (probably) three species of fish were being caught.

- For distributions of numeric data (or what I call "X-data"), histograms are an amazing and powerful tool!

The idea behind a histogram is to take the *domain* of your data, chop it into bins, and count how many data points fall into each bin. This brings out the underlying distribution:

[3]:
```
import numpy as np

import seaborn as sns # another plotting library


x = np.random.randn(100)
ax = sns.distplot(x, bins='auto', rug=True, kde=False)
#                                 ^^^^^^^^  ^^^^^^^^^
plt.xlabel("x")
plt.show()
```



Dealing with many histograms

Histograms are one of the most important EDA (exploratory data analysis) tools.

Often you want to explore larger datasets, meaning you may want to visualize many histograms.

As a data scientist working away, you may find yourself looking at histogram after histogram after histogram ad nauseum.

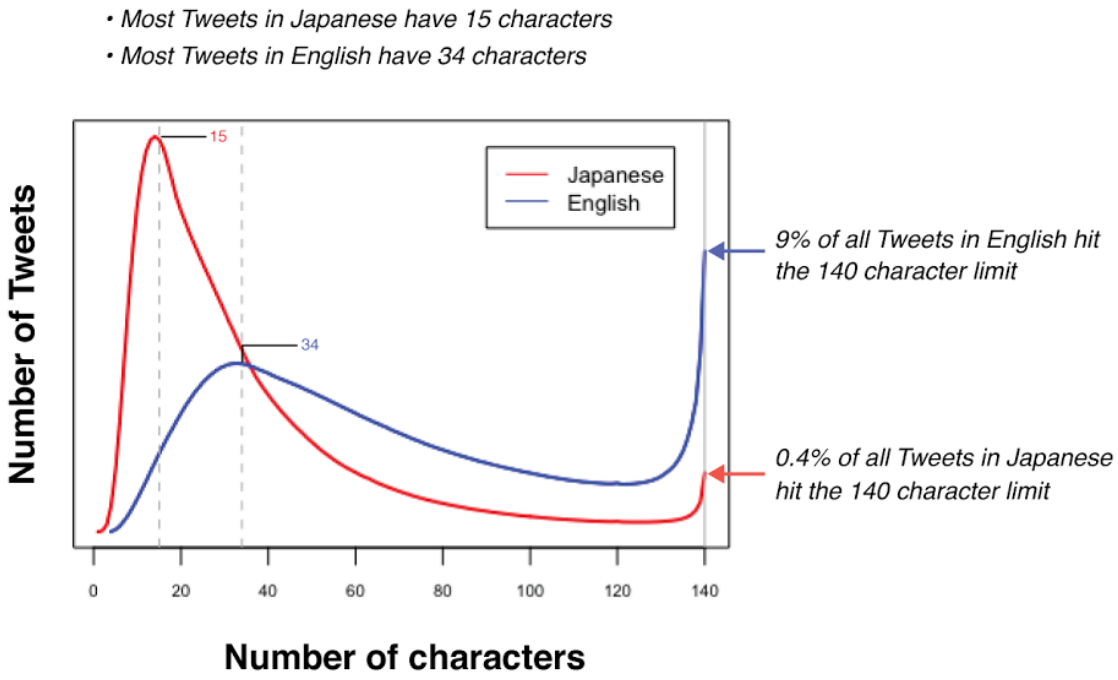
But how to compactly summarize many histograms for an audience? (This is getting more into *presentation* than exploration.)

Sometimes we have **multiple collections** of X-data (e.g., X-data by age: for subjects in their 20s, their 30s, 40s, etc.) We want to compare these different distributions to see if something in the X-data is changing as the (e.g.) age changes.

A great example is the Twitter tweet-length data science question, looking at English and Japanese tweets:

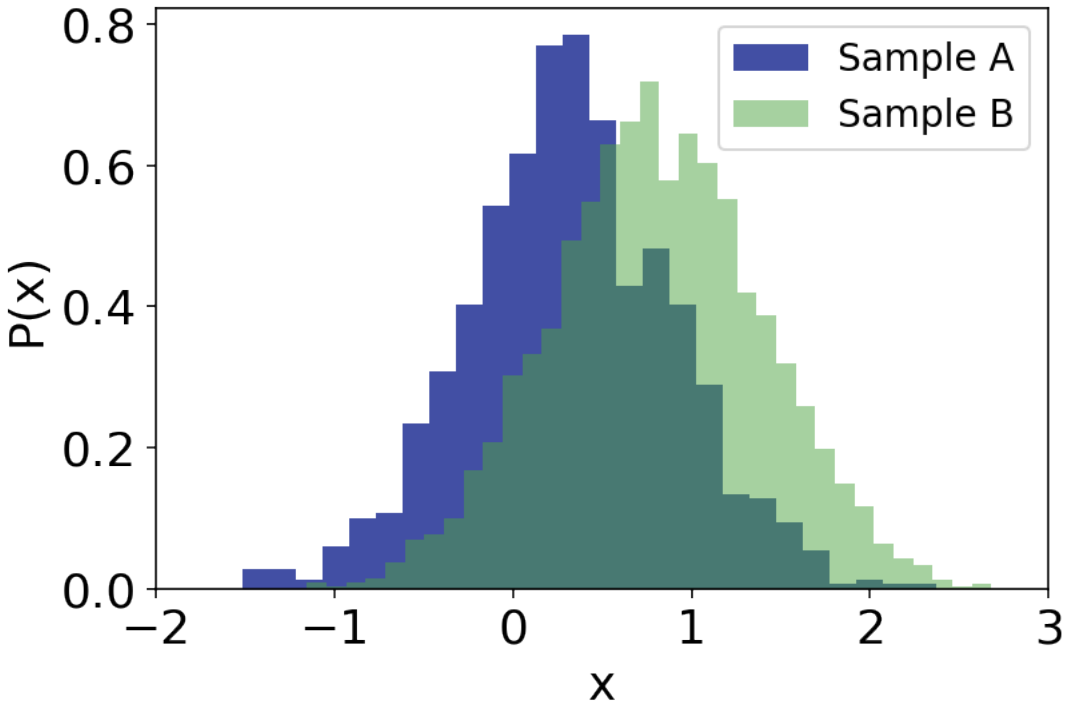
[4]:
```
from IPython.display import Image
Image("figures/more-chars-1.png", width=700)
```

[4]:



Two-ish histograms

[5]:
```
data_A = [ np.random.randn()*0.6 + 0.34 for _ in range(1000) ]
data_B = [ np.random.randn()*0.6 + 0.82 for _ in range(3000) ]

plt.hist(data_A,bins='auto', density=True,
         color="#424FA4")

plt.hist(data_B,bins='auto', density=True,
         color="#4FA442",
         alpha=0.5) # ***
plt.xlim(-2,3)

plt.xlabel("x");
plt.ylabel("P(x)");
plt.legend(["Sample A", "Sample B"], fontsize=14)
plt.show()
```



More than two histograms

Transparency can help visualize the overlap (similarity/difference) of two distributions.

We can draw multiple histograms but too much overlap will make it hard to read.

- This can be visually confusing if you need to show many histograms at once.
- Transparency can introduce **artifacts** (people will confuse the overlapping segment as a third histogram).

Histogram curves can get complicated and you can only cram so much information into one plot!

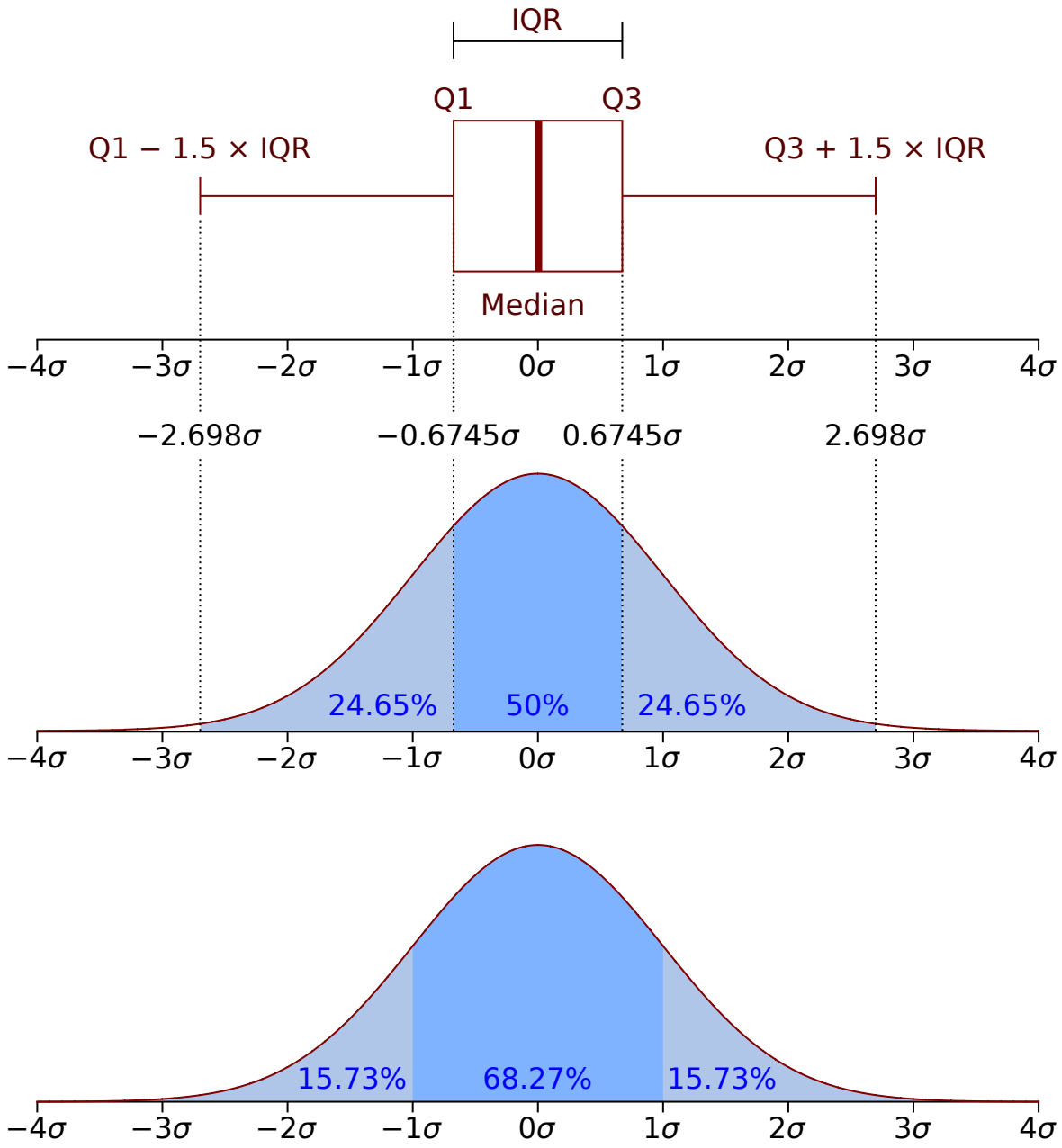
Another idea is to **further reduce the information about the distribution**

Box plots

A boxplot is just a simple way to summarize the center of a distribution of data and its width. A picture is worth a thousand words:

[6]:
```
#Image("figures/Boxplot_vs_PDF.svg", width=700)
from IPython.core.display import SVG
SVG(filename='figures/Boxplot_vs_PDF.svg')
```

[6]:



A Boxplot (top) compared to the underlying gaussian distribution.

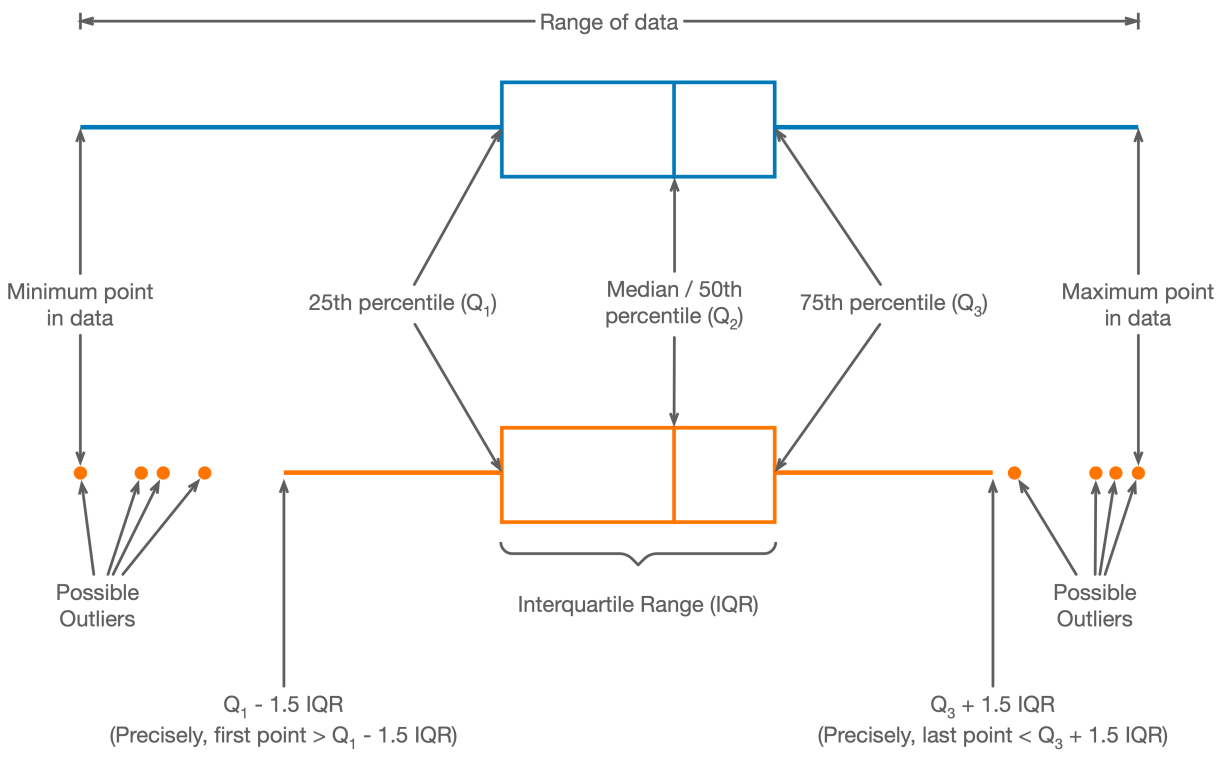
Boxplots illustrate in a reduced schematic the:

1. Median (center stripe)
2. IQR (width of box)
3. 1.5 IQR $\pm$ the outer quartiles, using error bars
    - sometimes the errorbars denote the 5th and 95th percentile or other features
4. **Outliers** are sometimes drawn as individual points beyond the error bars

Box plot "anatomy":

[7]:
```
Image("figures/boxplots-annotated.png", width=750)
```

[7]:

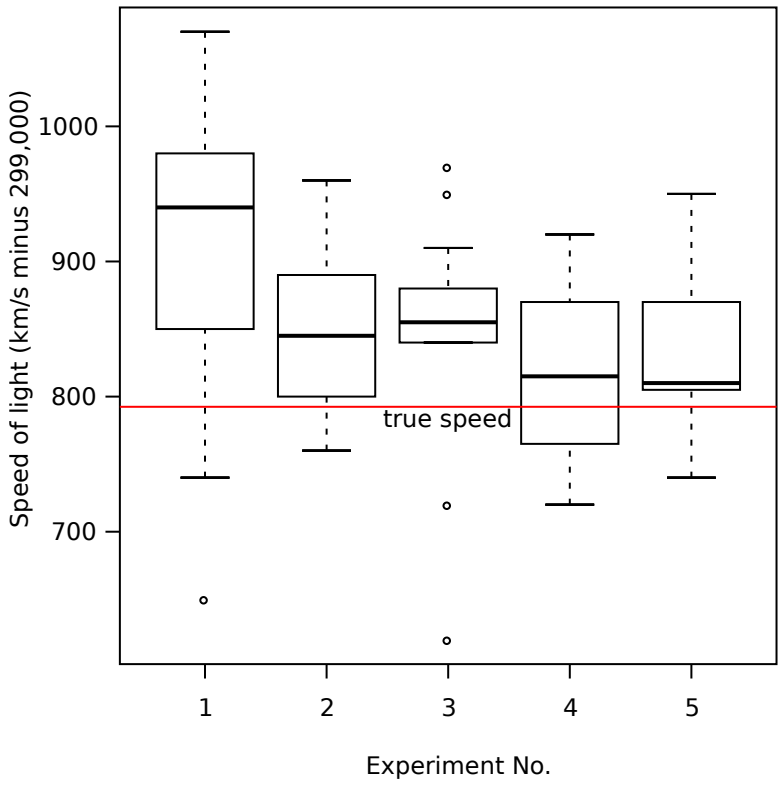


- The definition of the *whiskers* hanging off the central box is not as standardized as the central box itself.

By simplifying the full shape of the distribution, boxplots let us compactly **compare and contrast** distributions:

[8]:
```
SVG(filename="figures/Michelsonmorley-boxplot.svg")
```

[8]:



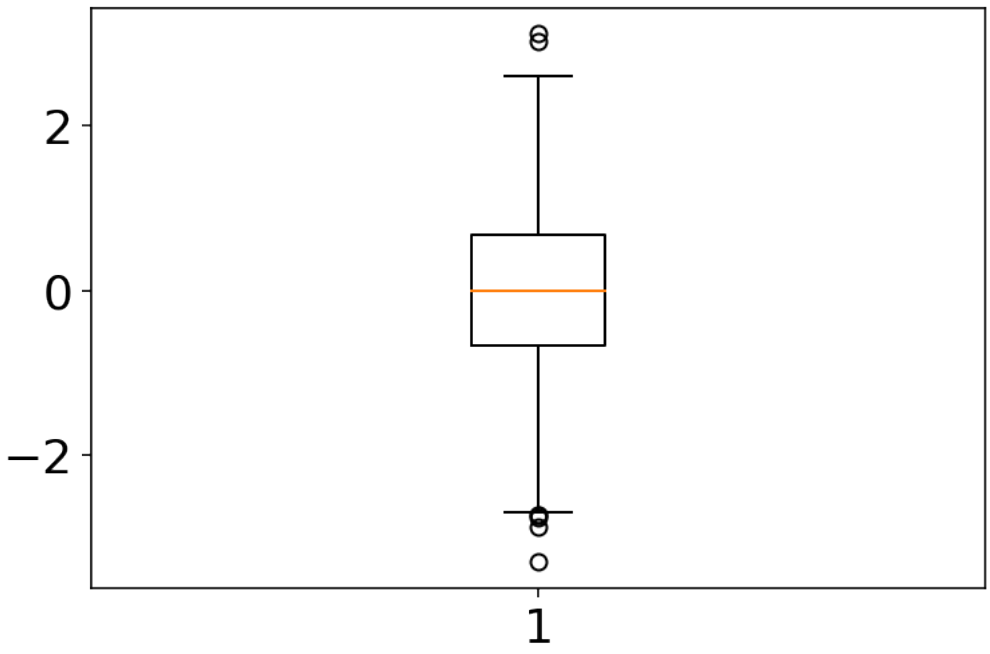
(Try fitting those five distributions into one plot using histograms.)

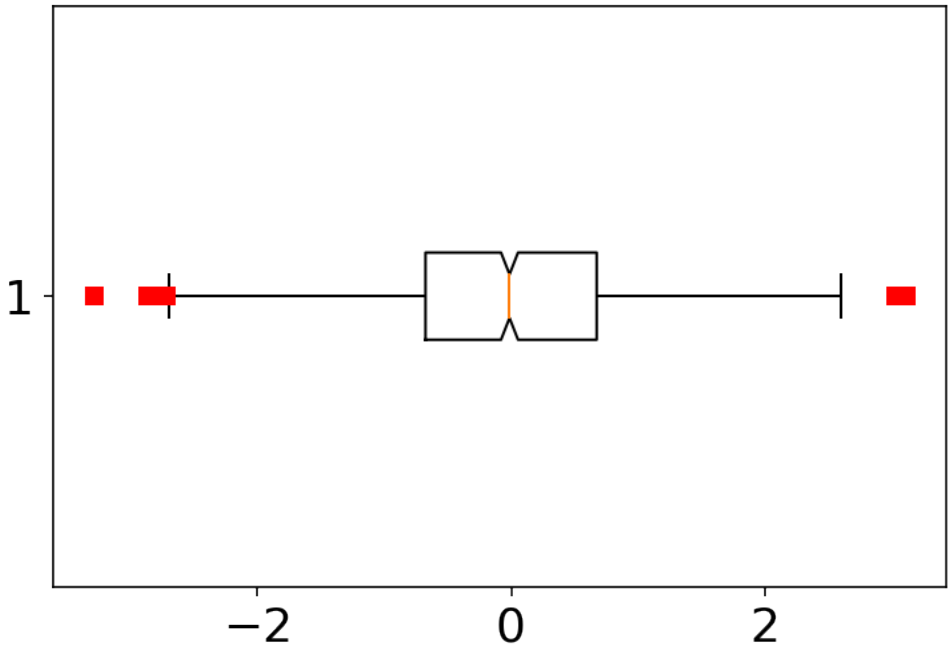
Boxplots are an ***old-fashioned idea*** from a time when computers were not used to visualize data. Instead, graphics were drawn by hand! But they can still be useful now, when multiple overlapping histograms become too crowded to display!

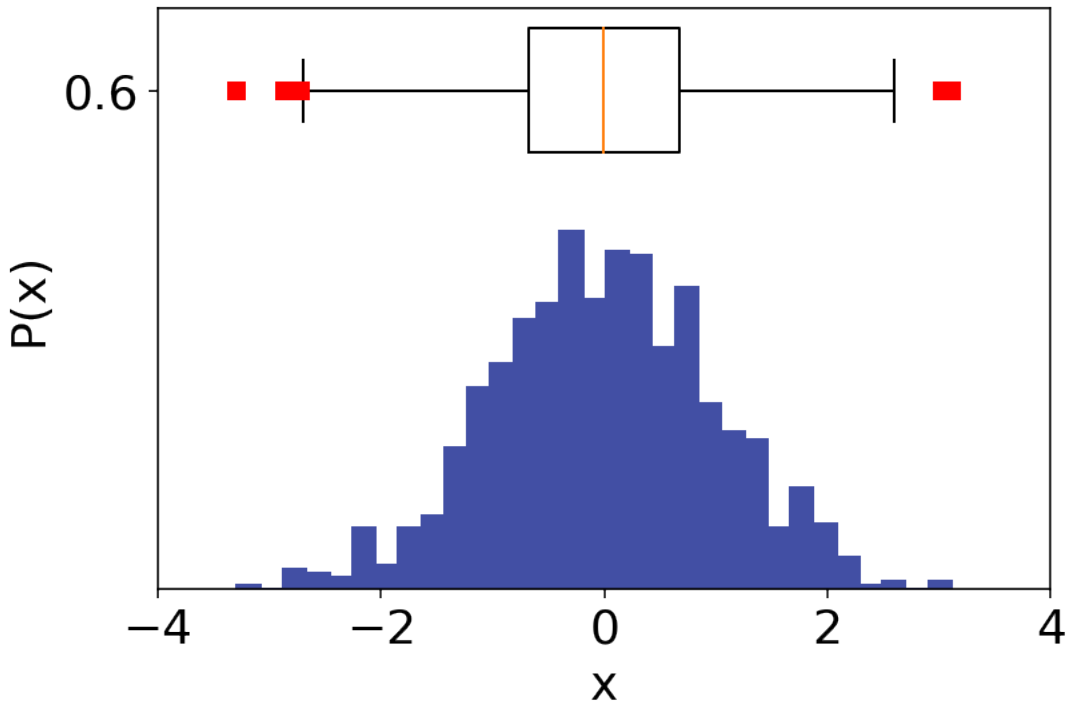
How to make boxplots in Python:

[9]:
```
data = [ np.random.randn() for _ in range(1000) ]
numbins =  int(np.sqrt(len(data))) # int is floor?


# default box plot
plt.boxplot(data)

# horizontal box plot with red squares for outliers...
plt.figure() # newfigure
plt.boxplot(data,notch=True,sym='rs',vert=False)


# compare histogram to boxplot on same plot:
plt.figure() # new figure
plt.hist(data,numbins, density=True, facecolor="#424FA4",edgecolor="#424FA4")
plt.boxplot(data,notch=False,sym='rs',vert=False, positions=[0.6])
plt.xlim(-4,4); plt.ylim(0,0.7)
plt.xlabel("x"); plt.ylabel("P(x)")
```

```
plt.show()
```



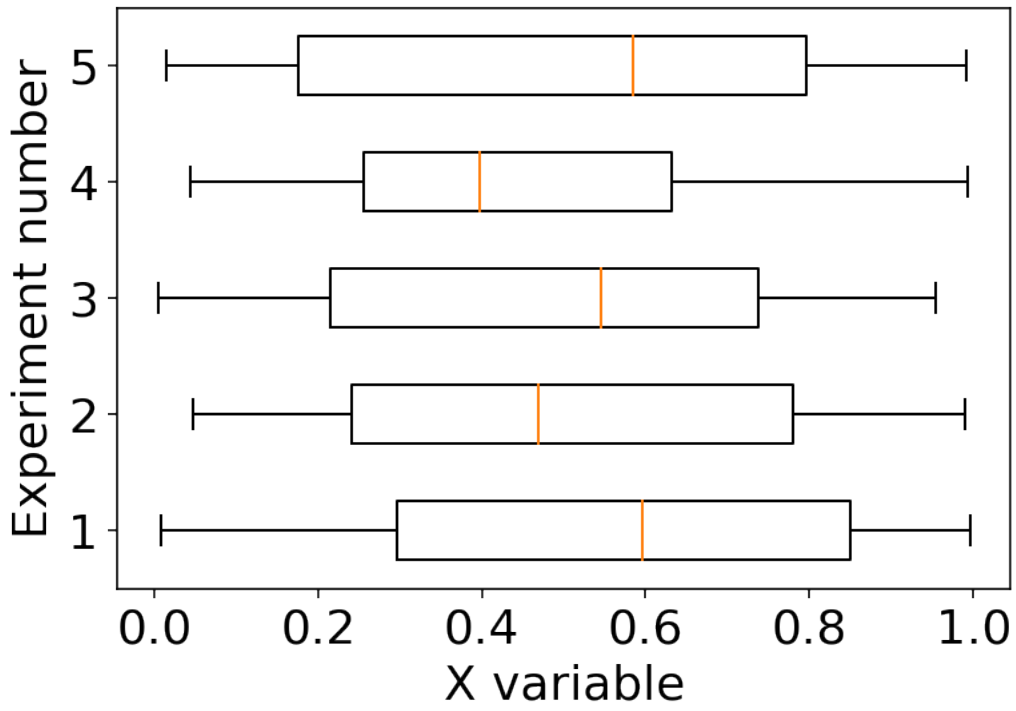




Compactly summarizing many distributions

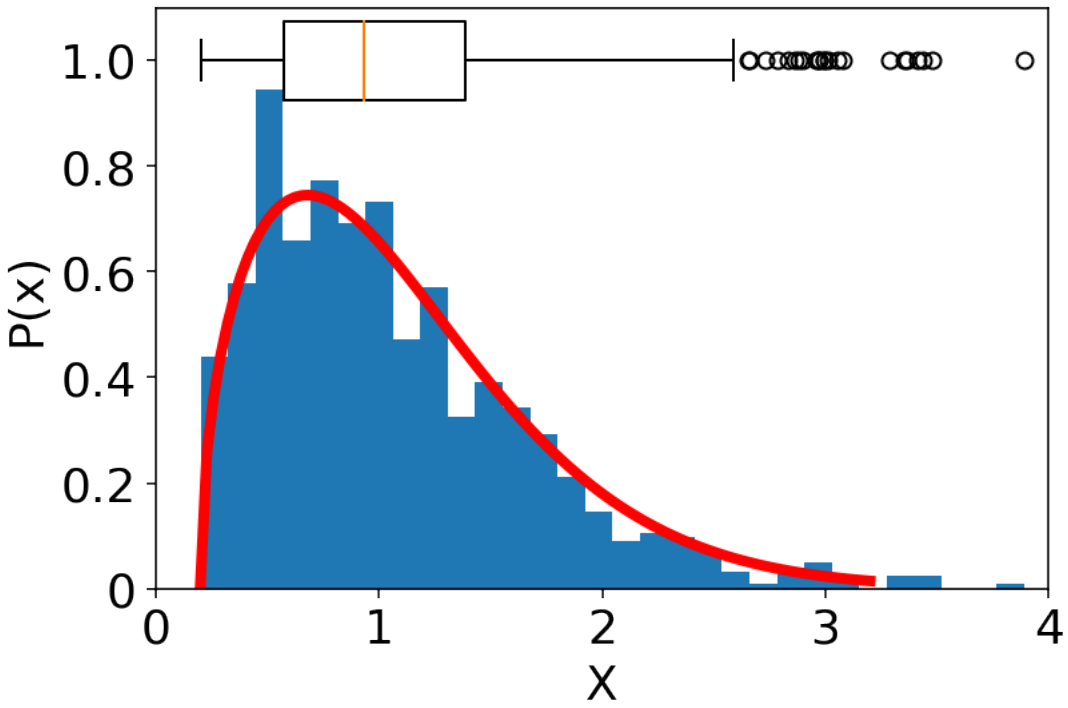
- Trade off quality (details within a single distribution) for quantity

[10]:
```
datasets=[]
for i in np.arange(5):
    data = np.random.random(50)
    datasets.append(data)

plt.boxplot(datasets,vert=False);
plt.xlabel("X variable"); plt.ylabel("Experiment number")
plt.show()
```



Of course, the underlying distribution need not be **Gaussian** (normal):

[11]:
```
def weibull(x, n,a):
    """PDF of weibull distribution"""
    return (a/n)*(x/n)**(a-1) * np.exp(-(x/n)**a)

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

plt.hist(data, 30, density=True)
x = np.linspace(0,3,100)

plt.boxplot(data,vert=False, )#positions=[0.95]);
plt.plot(x+0.2, weibull(x, 1., 1.5), 'r-', lw=4)


plt.xlim(0,4)
plt.ylim(0,1.1)
yts = [0,0.2,0.4,0.6,0.8,1.0]
plt.yticks(yts,yts)
plt.xlabel("X")
plt.ylabel("P(x)")
plt.show()
```

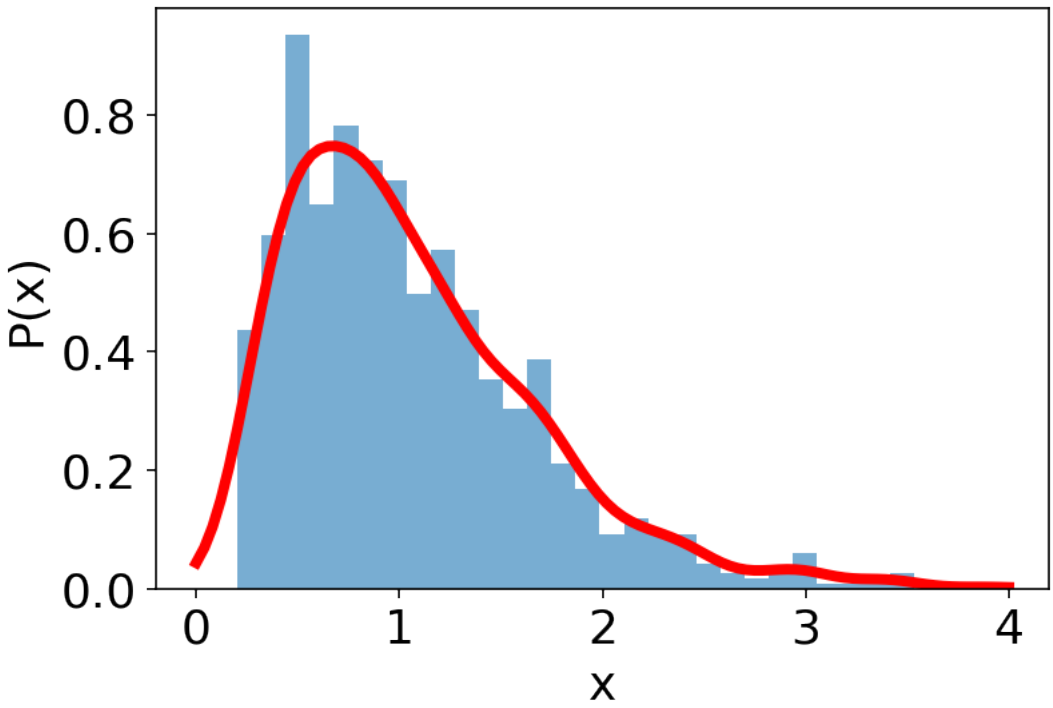


Kernel density estimation (KDE)

Recall that KDE is a technique for "smoothing" out a histogram.

- Sometimes you want to make things **pretty**, and these pictures are a little old-fashion.
- Plus if the data are noisy, the histograms look **jagged**.

[12]:
```
# scipy!!!
from scipy.stats.kde import gaussian_kde


# 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,numbins,density=True,alpha=.6) # histogram
plt.xlabel("x"); plt.ylabel("P(x)")
plt.show()
```

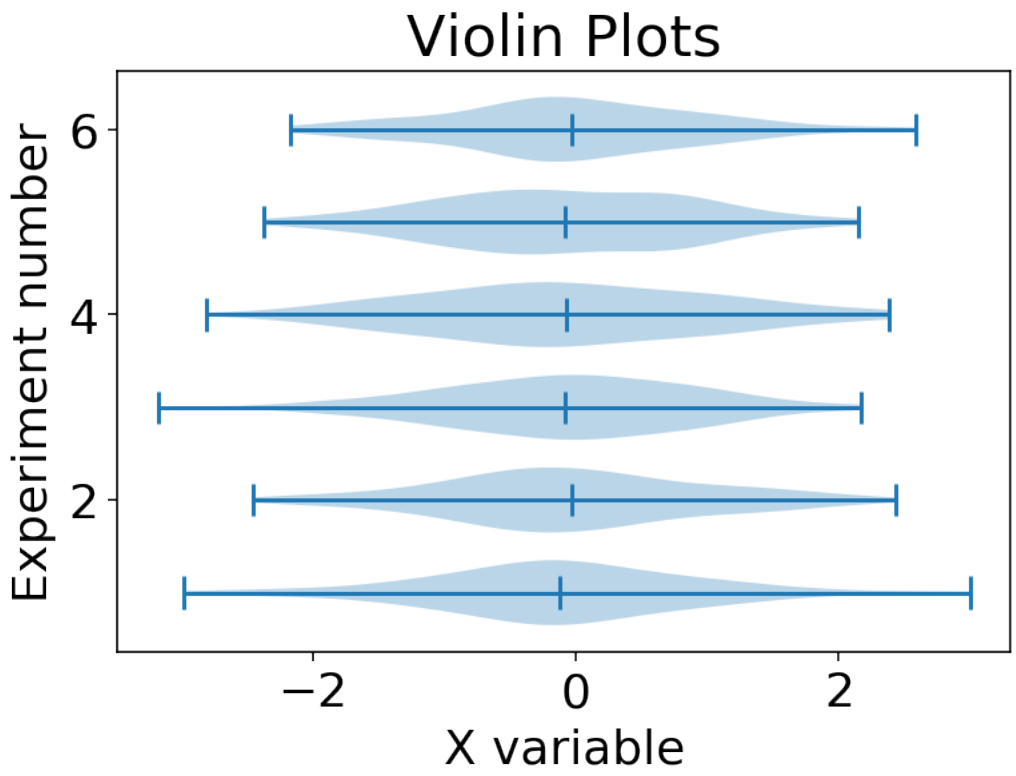


You can also **combine** boxplots and KDE in a nice way:

Violin plots

Slightly fancier box plots. Crossing box plots with KDEs.

[13]:
```
# fake data
fs = 10 # fontsize
pos = [1,2,3,4,5,6] # y-variable...?
data = [np.random.normal(size=100) for i in pos]

plt.violinplot(data, pos,
               points=80, vert=False, widths=0.7,
               showmeans=True, showextrema=True)

plt.title('Violin Plots')
plt.xlabel("X variable")
plt.ylabel("Experiment number")
plt.show()
```



Here's an example in action, from a recent paper of mine (with an alumni of this very course!):

[14]:
```
Image("figures/violin_plots_in_action.png", width=700)
```

[14]:

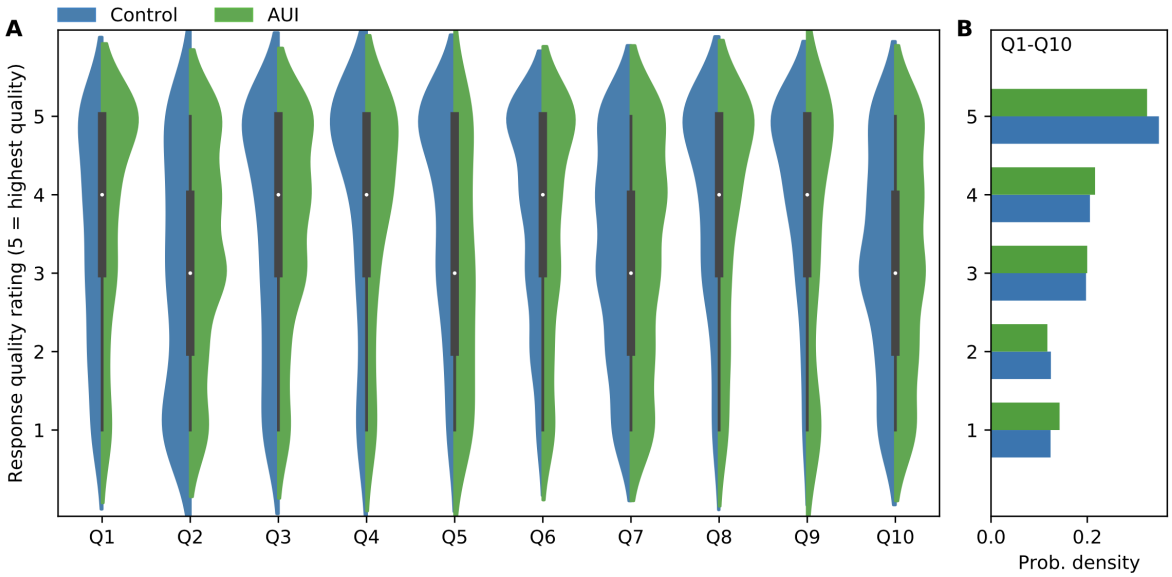


Figure 5: Quality of responses. All question-response pairs were rated independently by workers on a 1-5 scale of perceived quality (1–lowest quality, 5–highest quality).

Here's an incredibly useful and powerful idea (slides).

Cumulative probability distribution

Read the notation as "The probability that a randomly chosen piece of data is less than a particular value x". (More precisely, "the probability that a *random variable* X takes on a value less than x".)

- Also known as the "Cumulative Distribution Function" (CDF) or even "Distribution Function".
- I sometimes like to denote this as $P_<(x)$, dispensing with the X.
- For a continuous distribution:

$$P(X < x) = \int_{-\infty}^{x} P(x)dx$$

- For a discrete random variable:

$$P(X \leq x) = \sum_{x_i \leq x} P(X = x_i) = \sum_{x_i \leq x} P(x_i)$$

- Complementary cumulative distribution $P(X \geq x) = 1 - P(X < x)$
- Standard practice to use uppercase variable for CDF to correspond to the lowercase variable for the PDF, $F(x)$ (cdf) vs. $f(x)$ (pdf).

So by sorting the data we have computed an ***empirical estimate*** of the CDF!

$$P(X \leq x) = \frac{\text{number of datapoints} \leq x}{\text{number of datapoints}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}_{x_i \leq x},$$

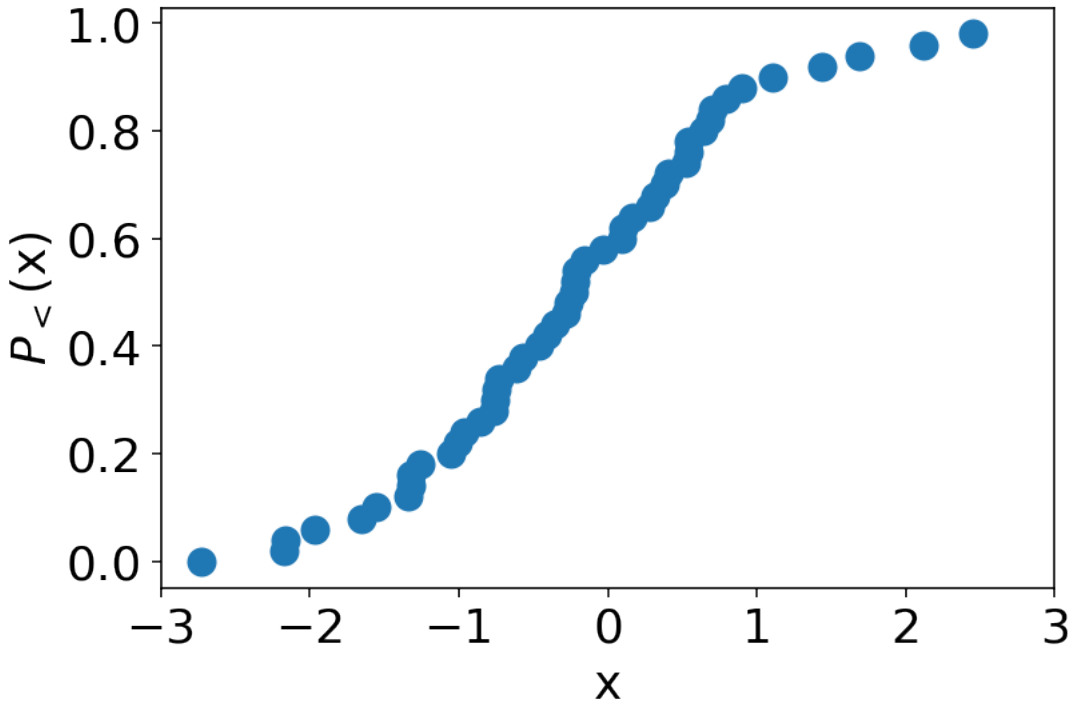
- Sorting generated all these counts at once!
- Sometimes called the **ECDF** for "Empirical CDF"
  - a subscript N is used to distinguish the ECDF for a sample of N datapoints from the "real" or "true" or "underlying" CDF.
  - For example, $F_N(x)$ vs. $F(x)$.
- For $P(X \leq x)$ just use $i\ /N$ instead of $(i-1)/N$
- (Common to write {number of datapoints} as #{datapoints}. Other variants also used.)

Let's see it in action:

How to compute the table from the slides?

[15]:
```
N = 50
data = np.random.normal(size=N)

X = sorted(data)
Y = np.arange(N)/N # that's the whole calculation
                   # that's it!!!
# np.arange(N) gives [0,1,...,N-1]

plt.plot(X, Y, '.', markersize=20)
```

```
plt.xlabel("x"); plt.ylabel("$P_<$(x)");
plt.xlim(-3,3)
plt.show()
```



(Technically, we should plot this using `plt.step` because the ECDF is defined for all values of x.)

Let's compare to the **true CDF**:

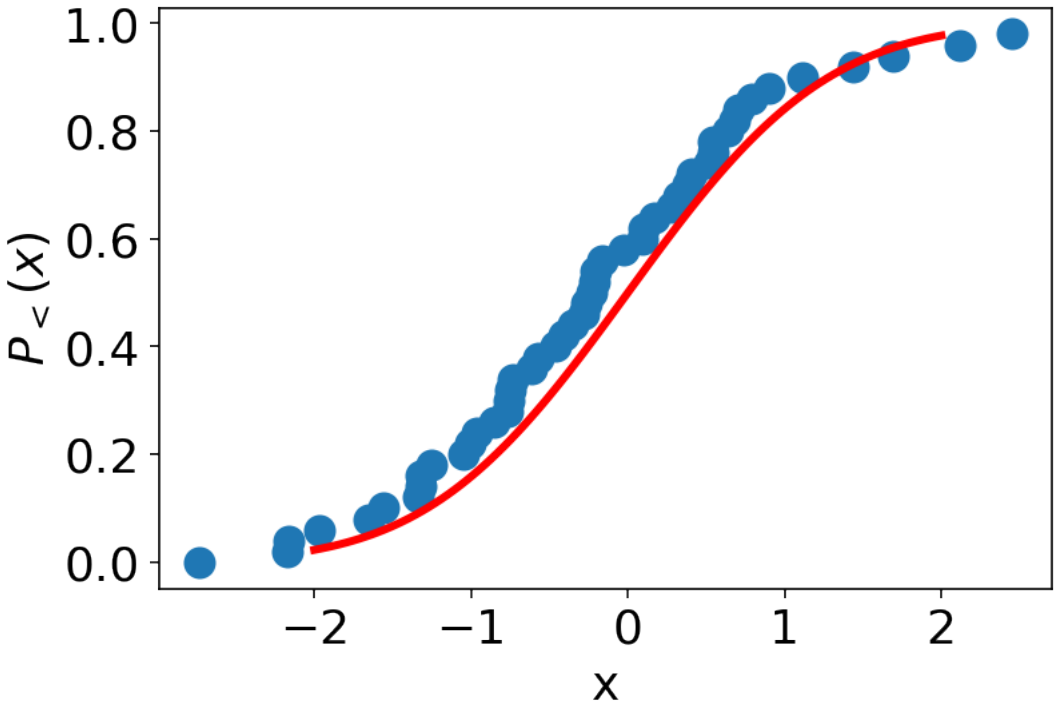
Remember that the PDF for a **normal distribution** with mean μ and variance σ^2 is

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

By integrating, the CDF is

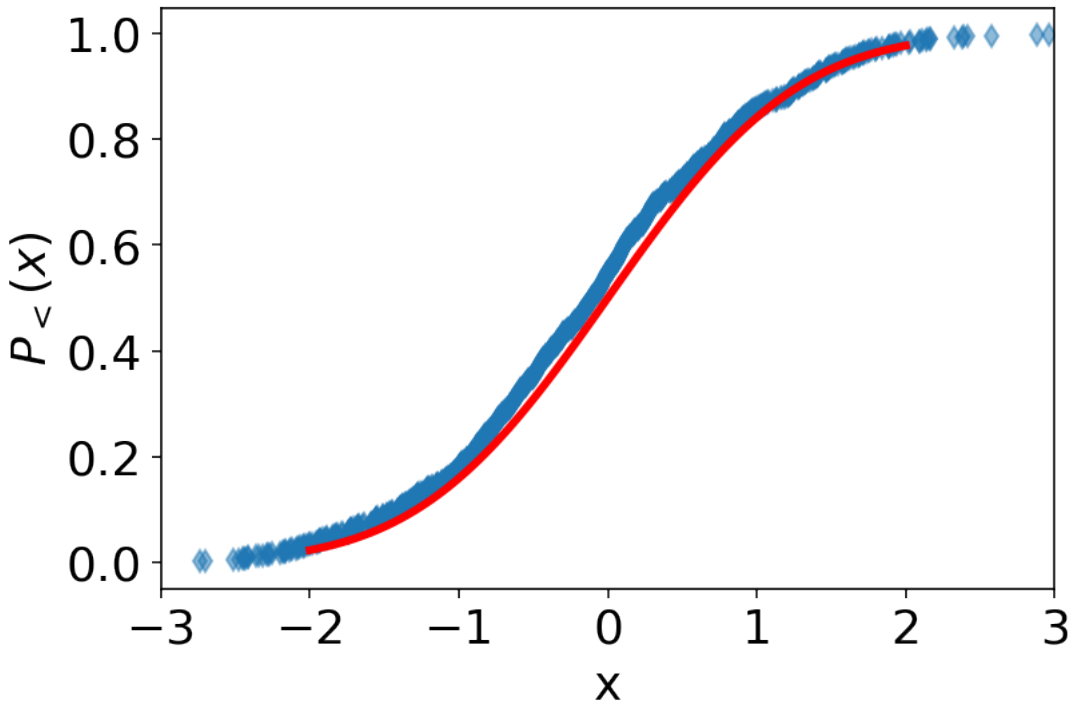
$$P(X < x) = \int_{-\infty}^{x} P(x)dx = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

[16]:
```
x_true = np.linspace(-2,2,100)
y_true = [0.5*(1+math.erf(x/math.sqrt(2))) for x in x_true] # integral of gaussian

#plt.plot(X,Y, 'bx', x_true,y_true, 'r-');
plt.plot(X,Y, '.', markersize=22)
plt.plot(x_true,y_true, 'r-', lw=3);

plt.xlabel("x")
plt.ylabel("$P_<(x)$")
plt.show()
```



Pretty good agreement with $N = 50$ points. What if we have more?

[17]:
```
N = 1000
data = np.random.normal(size=N)

X = sorted(data)
Y = 1.0*np.arange(N)/N

plt.plot(X,     Y,      'd', alpha=0.5)
plt.plot(x_true,y_true, 'r-', lw=3);
plt.xlabel("x"); plt.ylabel("$P_<(x)$");
plt.xlim(-3,3)
plt.show()
```



Pros CDF vs. PDF

- CDF uses ***all*** the data, nothing is lost.
- CDF has no choices for binning or kernel bandwidth — method is automatic!
- Works equally well for continuous and discrete data

Cons

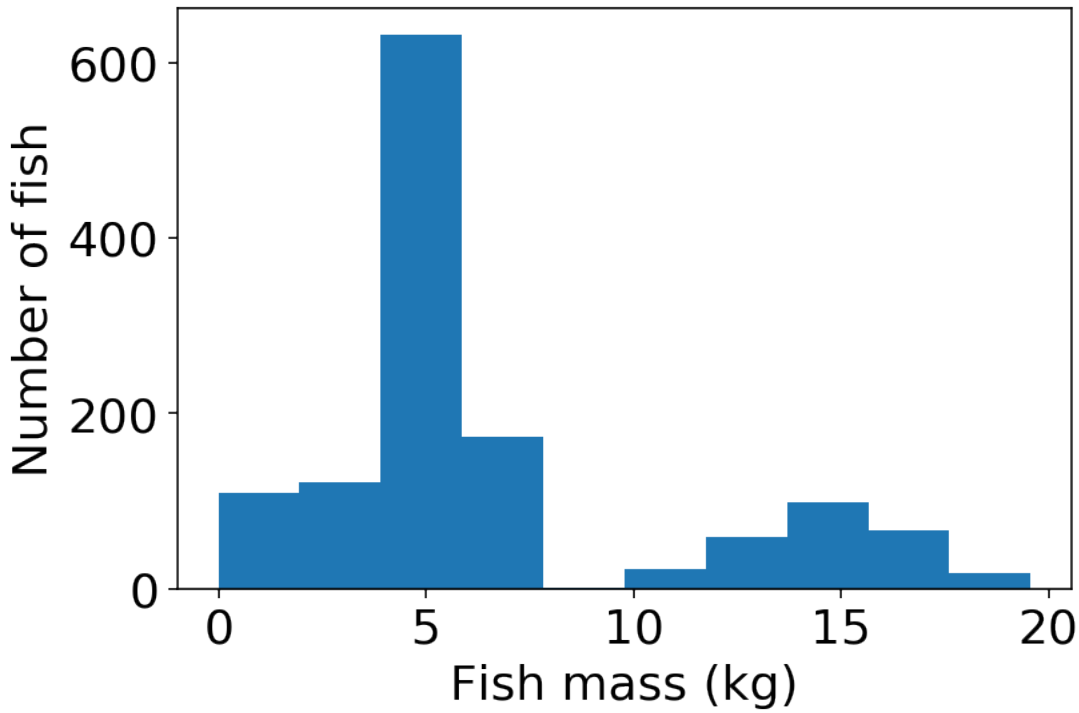
- It is harder to think about for people, especially those who have weak calculus backgrounds
- Because it's cumulative, a bias at one region will affect all regions to the right (we will address this shortly)

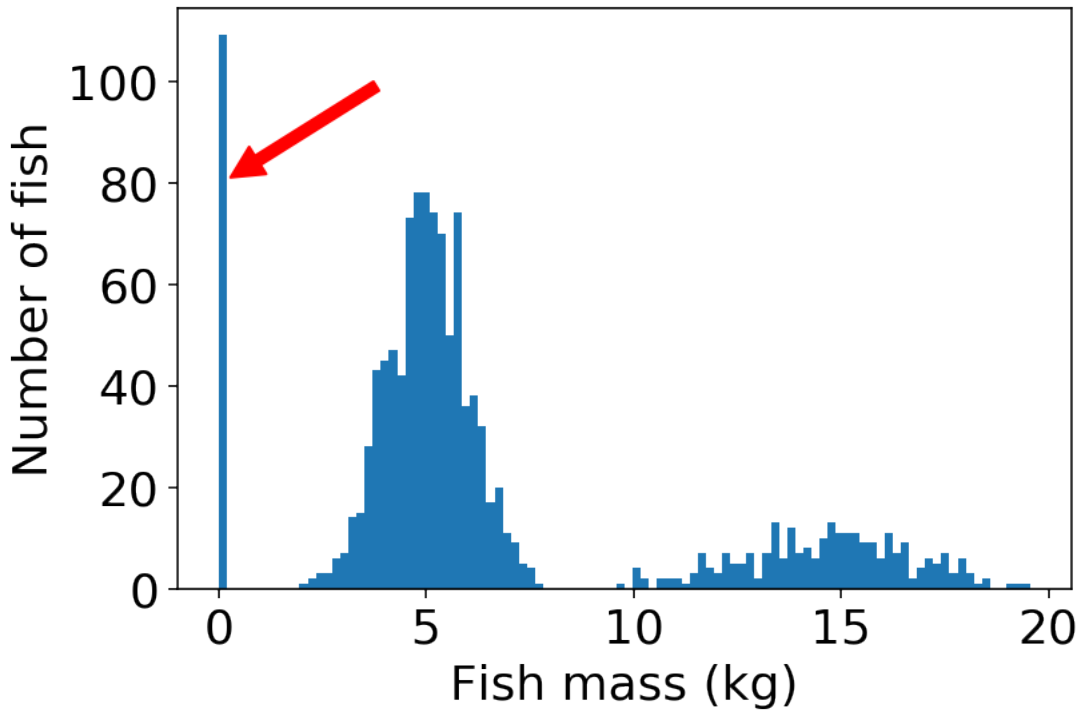
Recall our **fish mass** data?

A bad binning hides a third mode of data:

[18]:
```
data = []
for line in open("fishMass_kilograms.txt"):
    data.append( float(line.strip()) )


plt.hist(data)

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

# more bins with an arrow annotation:
plt.figure()
plt.hist(data, bins=100);
```

```
plt.gca().annotate('', xy=(0.1, 80), xytext=(4, 100),
  arrowprops=dict(facecolor='red', edgecolor='red',shrink=0.05),
)

plt.xlabel("Fish mass (kg)");
plt.ylabel("Number of fish");
plt.show()
```

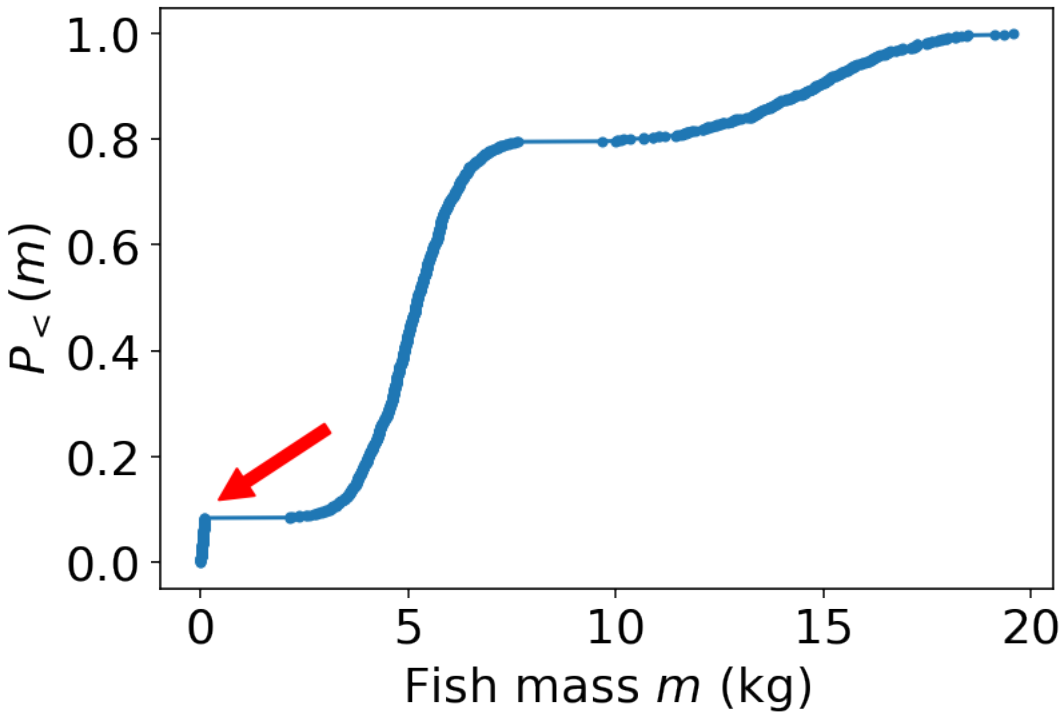




Question
How many fish are very small? It looks like a lot, but it's hard to read out the **proportion of data** in that position from just the histogram.

Now let's try the CDF:

[19]:
```
X = sorted(data)
N = len(data)
Y = 1.0*np.arange(N)/N

plt.plot(X,Y,'.-')
#plt.step(X,Y, where='pre', lw=4)

plt.gca().annotate('', xy=(0.3, 0.11), xytext=(3.2, 0.26),
  arrowprops=dict(facecolor='red', edgecolor='red',shrink=0.05),
)

plt.xlabel("Fish mass $m$ (kg)")
plt.ylabel("$P_<(m)$") # shorthand for P(M < m)
plt.show()
```



Ahh, we see immediately that it's around 10% of the data!

CDFs are **easier** to read than PDFs in one sense: * To find the proportion of data between two values, just look at how much the CDF travels vertically. For example, in the plot above the CDF is at about 10% for 2.5kg and about 80% for 7.5 kg. Therefore, we can immediately tell that about 70% of the data lie in the second mode.

- To find this value for a histogram requires estimating the *area* of the second mode relative to the total area. This is a *much harder visual task*.

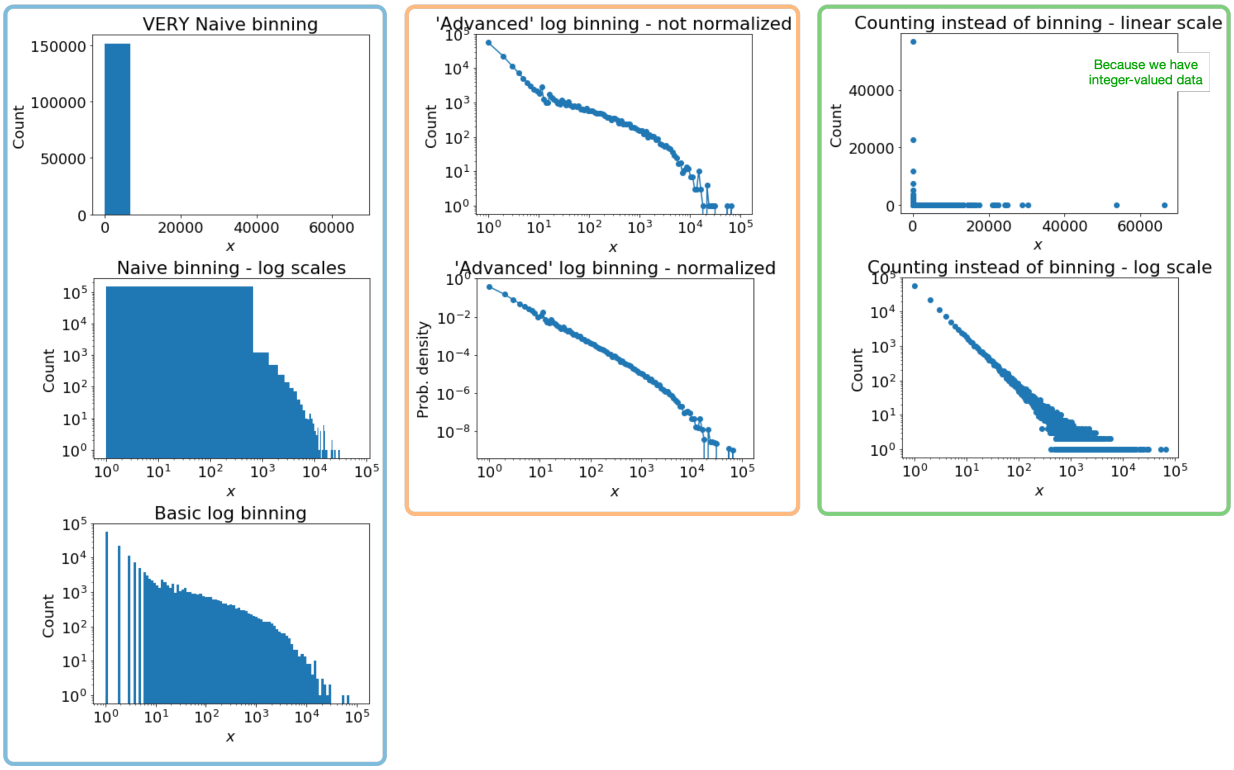
Using CDFs to examine broadly distributed data

Recall our **skewed/broad** data?

Summarizing from the previous lecture: 1. Simple binning 2. "Advanced" binning 3. Counting

[20]:
```
print()
Image("figures/broad-data-binning-summary.png", width=800)
```

[20]:



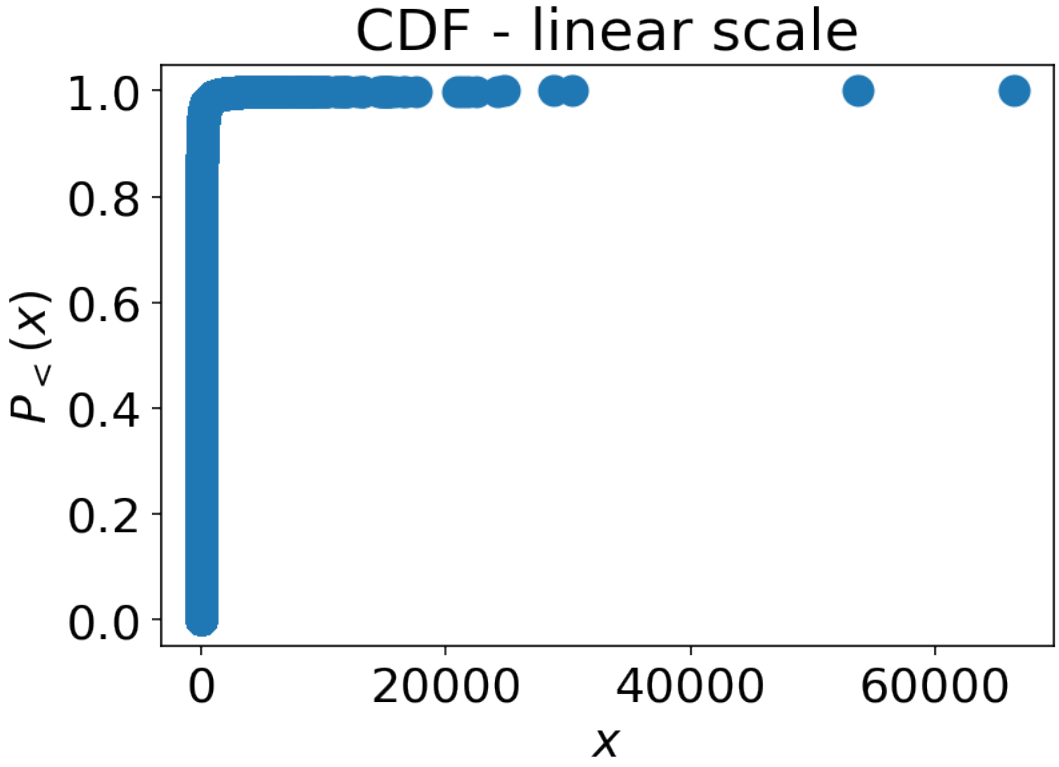
Takeaway?

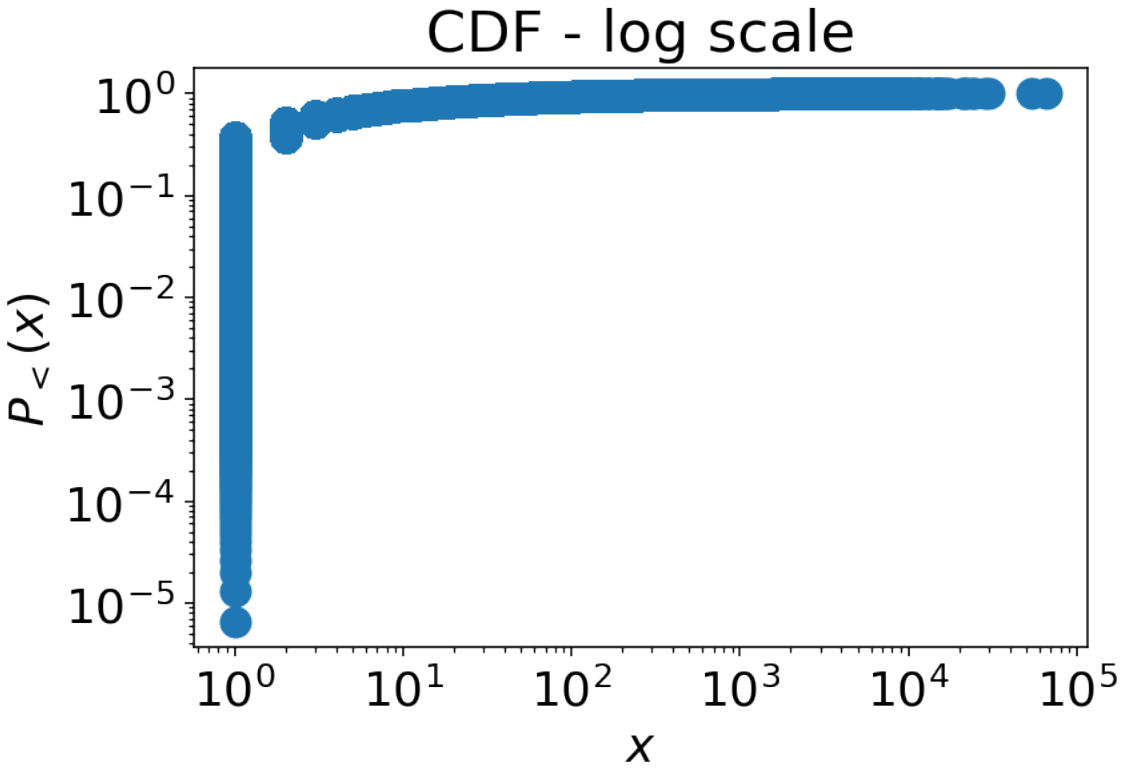
So many different option, between the linear/log scales and the (vast) set of binning options.

What can we do?

Use the Cumulative distribution!

[21]:
```
data = [float(line) for line in open("7impact_counts.txt")]

X = sorted(data)
N = len(data)
Y = 1.0*np.arange(N)/N

plt.plot(X,Y, '.', markersize=22)
plt.xlabel("$x$");plt.ylabel("$P_<(x)$")
plt.title("CDF - linear scale")
plt.show()

plt.loglog(X,Y, '.', markersize=22)
plt.xlabel("$x$");plt.ylabel("$P_<(x)$")
plt.title("CDF - log scale")
plt.show()
```

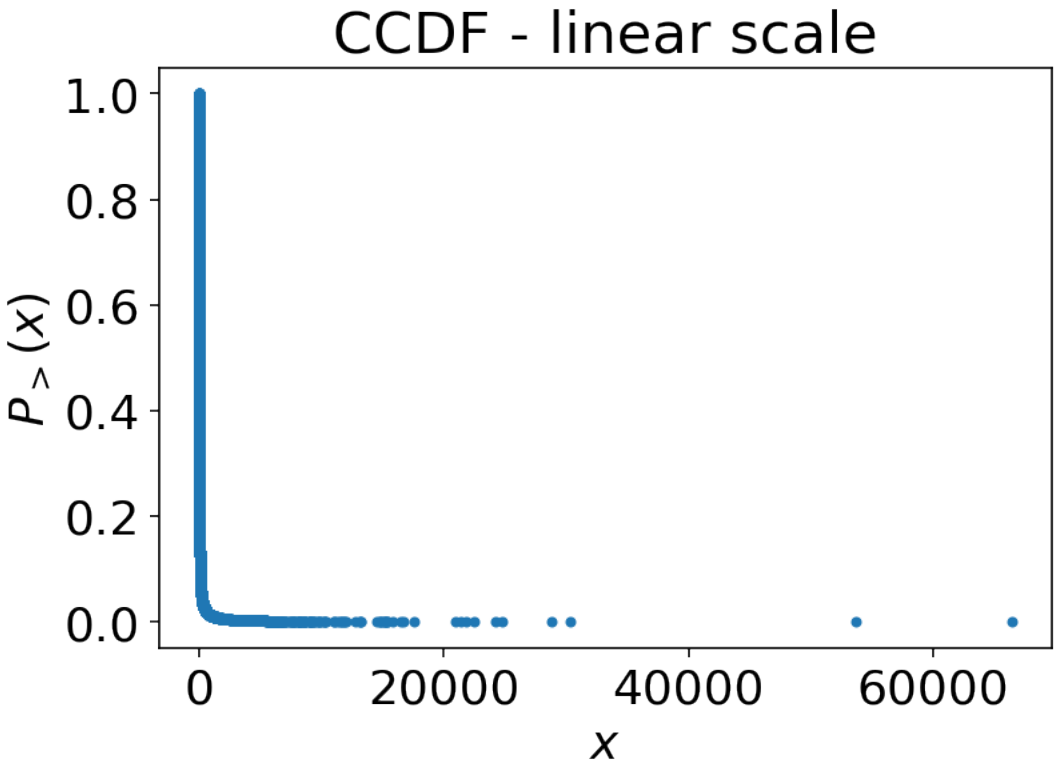


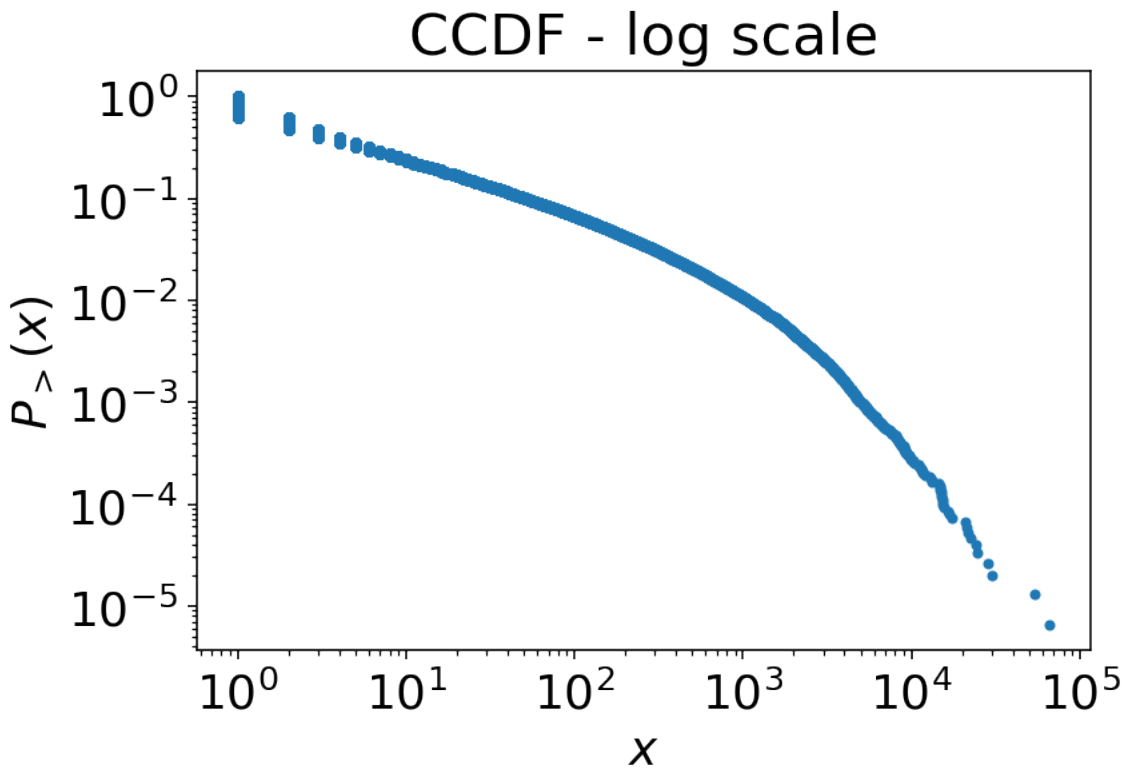


Turns out to not be so helpful. Why? Because we are looking at $P(X < x)$. Let's look at $P(X > x)$:

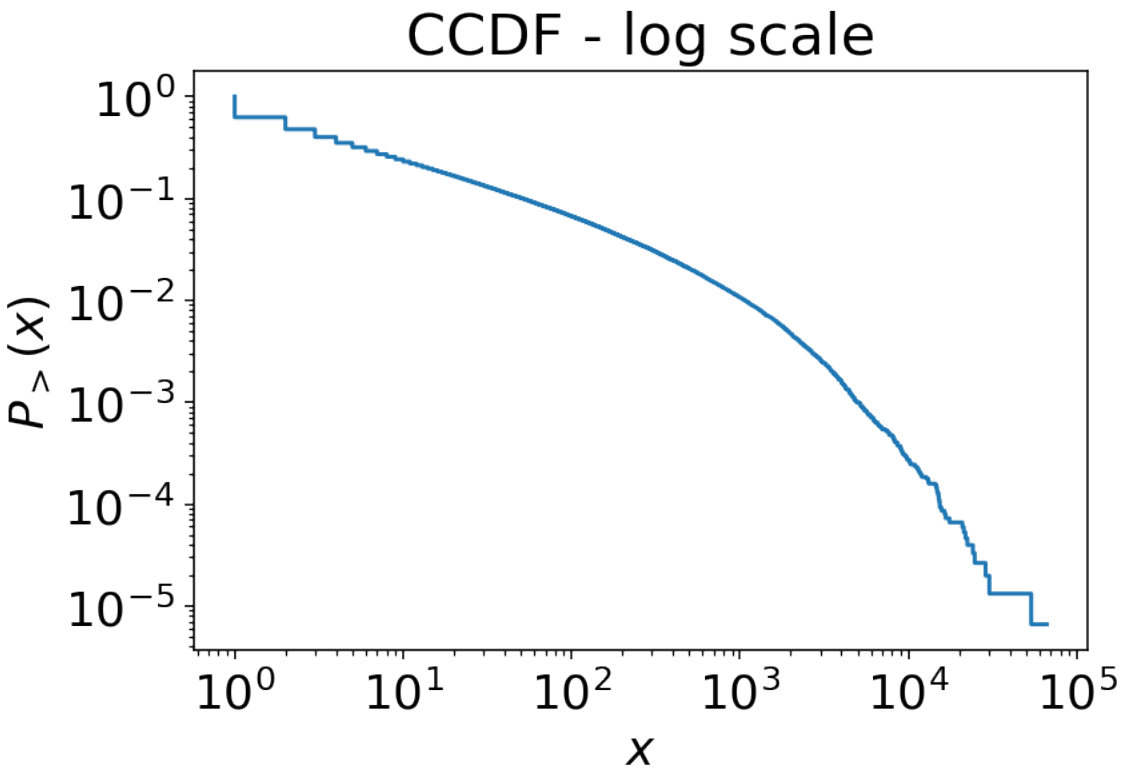
- This is sometimes called the CCDF (complementary CDF), but some fields define CDF and CCDF in reverse compared with other fields

[22]:
```
Y = 1 - 1.0*np.arange(N)/N # CCDF

plt.plot(X,Y, '.')
plt.xlabel("$x$");plt.ylabel("$P_>(x)$")
plt.title("CCDF - linear scale")
plt.show()

plt.loglog(X,Y, '.')
plt.xlabel("$x$");plt.ylabel("$P_>(x)$")
plt.title("CCDF - log scale")
plt.show()
```





[23]:
```
plt.step(X,Y)
plt.xscale('log')
plt.yscale('log')
plt.xlabel("$x$");plt.ylabel("$P_>(x)$")
plt.title("CCDF - log scale")
plt.show()
```



I cannot impress upon you more strongly just how *useful* and *powerful* the CDF is, even for just basic data exploration. I use it all the time. Even when I go back and report a histogram because it's simpler for presenting (interdisciplinary) research, I still **start** with the CDF.

- Python has a `cumulative=True` option for histograms, but it's still binning! ***They got it wrong!!!*** Easier to just sort!
- I've been a little lax in some details. For example, the limits $x \to -\infty$ and $x \to \infty$. At the largest datapoint, the ECDF curve jumps vertically from a y = (N-1)/N to y = 1. Likewise, the distinction between $P(X < x)$ and $P(X \leq x)$; this can be easily worked out by comparing i/N instead of $(i-1)/N$.

Errors and sampling in CDFs

(This addresses the cumulative bias that a sampling effect or error in one region of the ECDF will affect all regions to the right.)

We just saw how much better the data represents the distribution when $N = 1000$ than when $N = 50$.

- How many data points do we need?

There are some beautiful results relating the **empirical** distribution function $F_n(x)$ to the true **distribution** $F(x)$.

1. **Glivenko–Cantelli** (1933):

$$\sup_{x \in \mathbb{R}} \left| F_N(x) - F(x) \right| \to 0 \text{ as } N \to \infty$$

assuming the data $x_1, \ldots, x_N$ are iid real numbers.

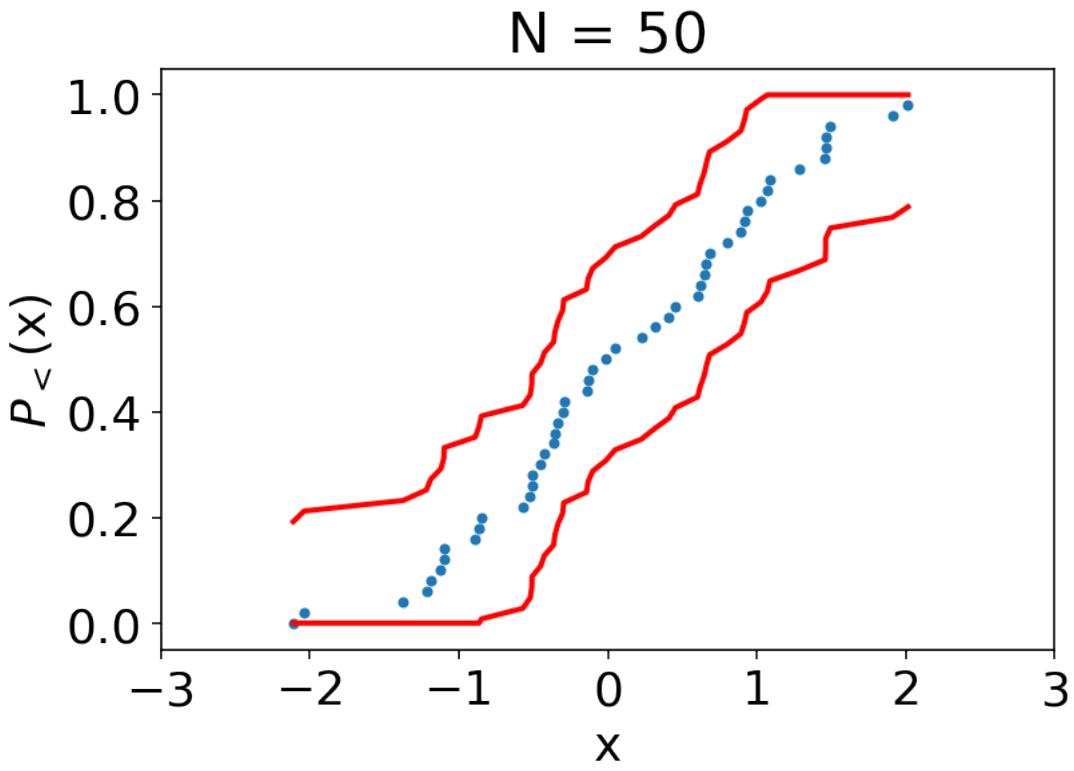
2. This result was strengthened, **Dvoretzky–Kiefer–Wolfowitz** inequality (1956) and Massart (1990).

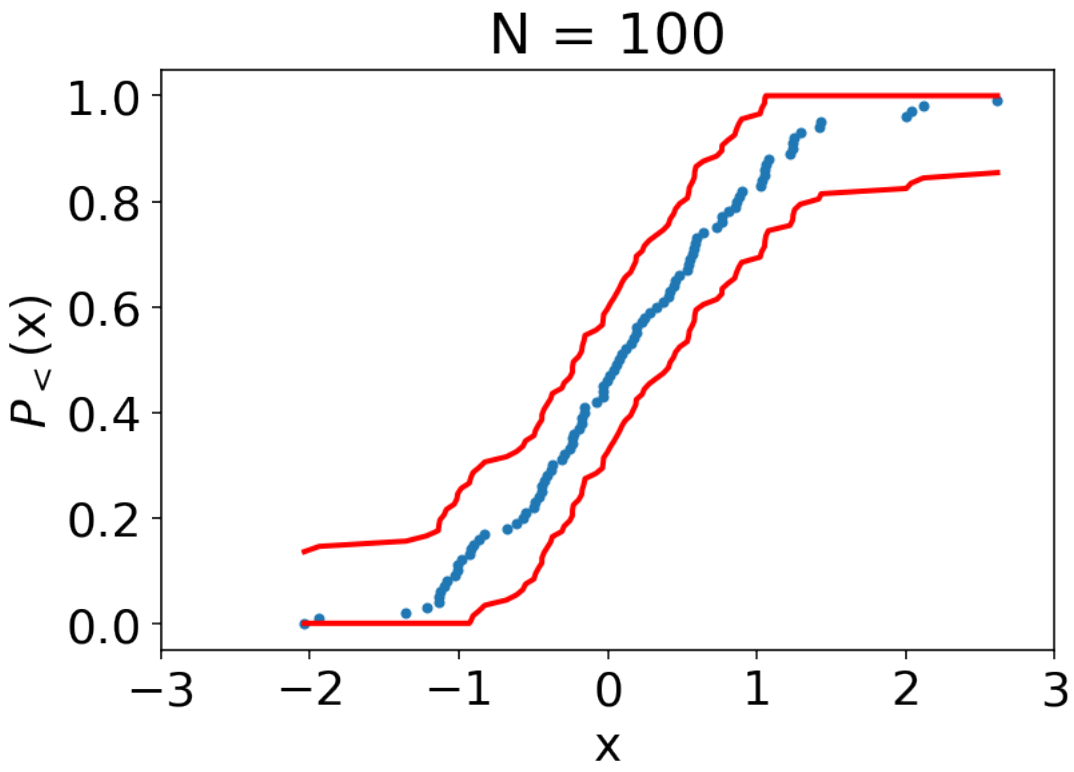
$$\Pr\left(\sup_{x\in\mathbb{R}} |F_n(x) - F(x)| > \varepsilon\right) \leq 2e^{-2n\varepsilon^2} \qquad \text{for every } \varepsilon > 0.$$

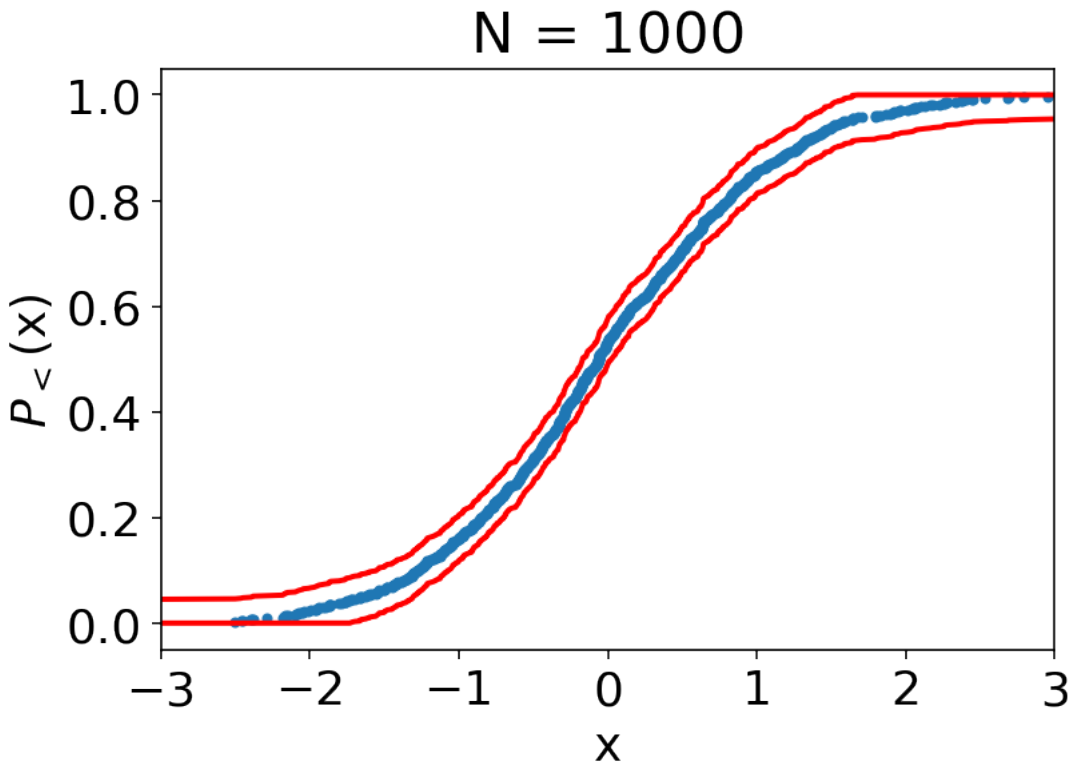
This shows that not only does F_N converge to F but how quickly!

Let's implement it by reversing DKW to find the largest value of ϵ that gives $\Pr\left(\sup_{x\in\mathbb{R}} |F_n(x) - F(x)| > \varepsilon\right) = \alpha$, for a given α:

[24]:
```
def DKW_CI(ecdf, alpha=0.05):
    """Dvoretzky-Kiefer-Wolfowitz confidence intervals at significance level alpha"""
    N = len(ecdf)
    eps = np.sqrt(np.log(2./alpha) / (2 * N))
    lower = np.clip(ecdf - eps, 0, 1)
    upper = np.clip(ecdf + eps, 0, 1)
    return lower, upper

for N in [50,100,1000]:
    plt.figure()
    data = np.random.normal(size=N)

    # get the ecdf:
    X = sorted(data)
    Y = 1.0*np.arange(N)/N

    plt.plot(X, Y, '.')

    lower,upper = DKW_CI(Y)

    plt.plot(X, lower, 'r-', X, upper, 'r-', lw=2)

    plt.xlabel("x"); plt.ylabel("$P_<$(x)");
    plt.xlim(-3,3);
    plt.title("N = %i" % N)
    plt.show()
```


N = 50
1.0
0.8
0.6
0.4
0.2
0.0
$P_<(x)$
−3
−2
−1
0
1
2
3
x


N = 100
1.0
0.8
0.6
0.4
0.2
0.0
$P_<(x)$
−3
−2
−1
0
1
2
3
x



Pros and cons of CDF vs. histogram

Pros

- No need to bin
- All data is used, nothing is lost
- Very easy to code up!
- Easy to tell how much of the data falls between two values x_L and x_R: just subtract: $P(X < x_R) - P(X < x_L)$. For the PDF you need to find the total *area* between x_L and x_R.

Cons

- **None!**

OK, OK!

Cons people mention that are not big deals:

- Every data point is plotted, which can lead to large figure files
  - This can be avoided by removing ties, but you need to be a little bit careful
- Errors compound as the data are "accumulated" left-to-right — An unusual value at the start of the function will change all subsequent values
  - This is addressed by the DKW confidence intervals derived above.

The one true con of CDFs over PDFs/histograms:

- Harder to communicate to some audiences; less intuitive
  - Need calculus skills. The CDF is the integral of the PDF, so you need to be comfortable with looking at a curve and seeing its derivative—the **steepness** of the curve tells you how much data is there.

Advice

I **always** go to the CDF to start, but I will also check the histogram: If the PDF via the histogram works well, I will tend to use that.

- Don't overcomplicate things

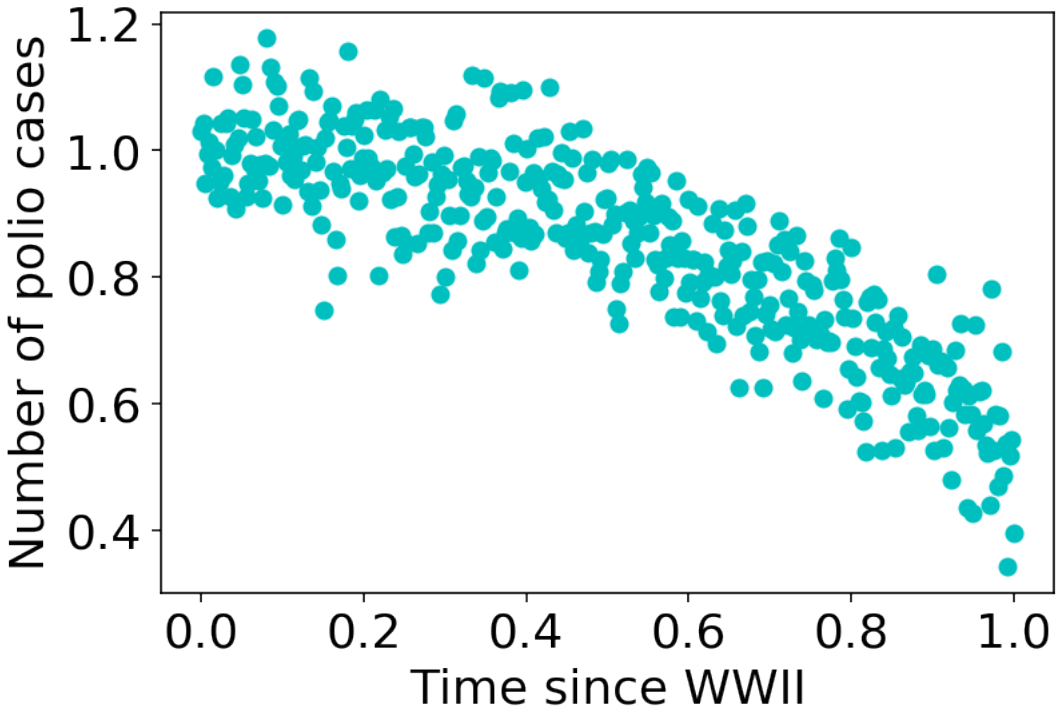
XY-data - Scatter plots

Often times you have pairs of (x, y) values. We've all seen plots of functions like $y = \cos(x)$. That's exactly what this is.

- When exploring data and looking at *pairs* of variables

When you have a set of "paired" data (or what I call XY-data), and you want to see if they exhibit a **trend** or are otherwise **related**, bust out the trusty old

[25]:
```
# Let's gussy up some fake data:

# 400 points evenly spaced in $[0,1]$:
X = np.linspace(0,1,400)

# y = cos(x) + some small noise:
Y = np.cos(X) + 0.075*(np.random.randn(*X.shape))

plt.plot(X,Y, 'o', color='c')

plt.xlabel("Time since WWII")
plt.ylabel("Number of polio cases") # not really
plt.show()
```



Just a quick scatter plot and we immediately see a nice **decreasing** trend.

- As simple as this is, this is one of the **fastest** and **most powerful** ways to explore your data.
- The two biggest tools in your data science toolbelt: histograms and scatter plots.

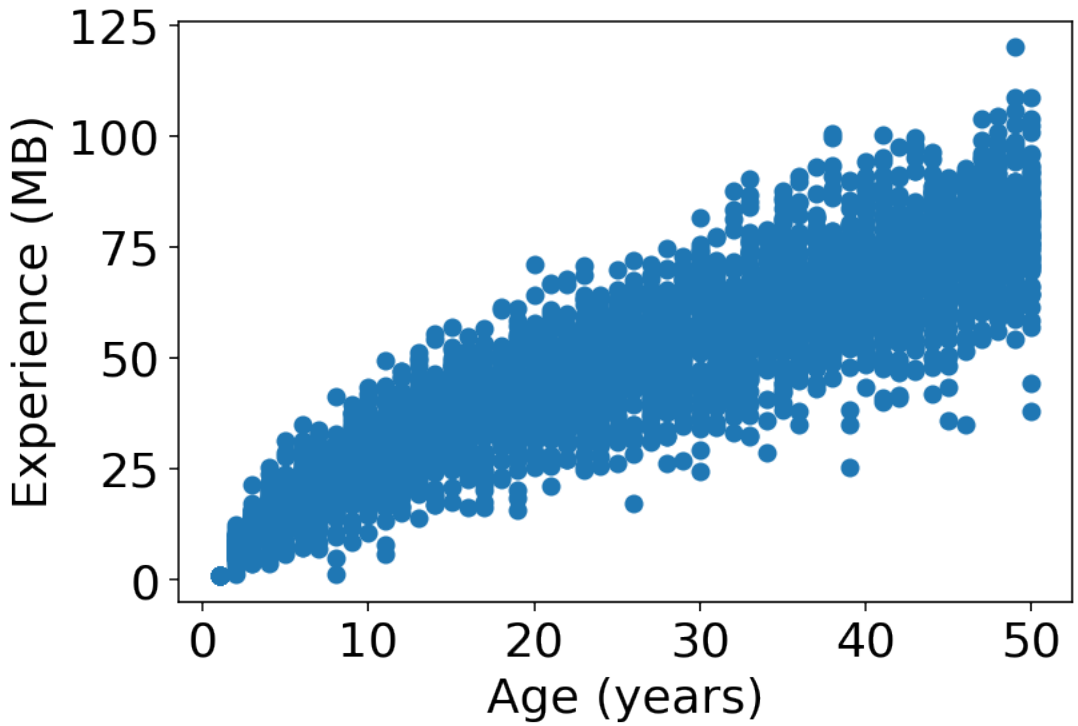
Some of the ideas behind **histograms** for X-data also translate well to working with scatter plots.

- We've got some noisy scatter. Let's compute a smoother **trendline** by averaging.

There's many ways to do this, a simple way is a **generalization** of the histogram.

If your x-values are discrete:

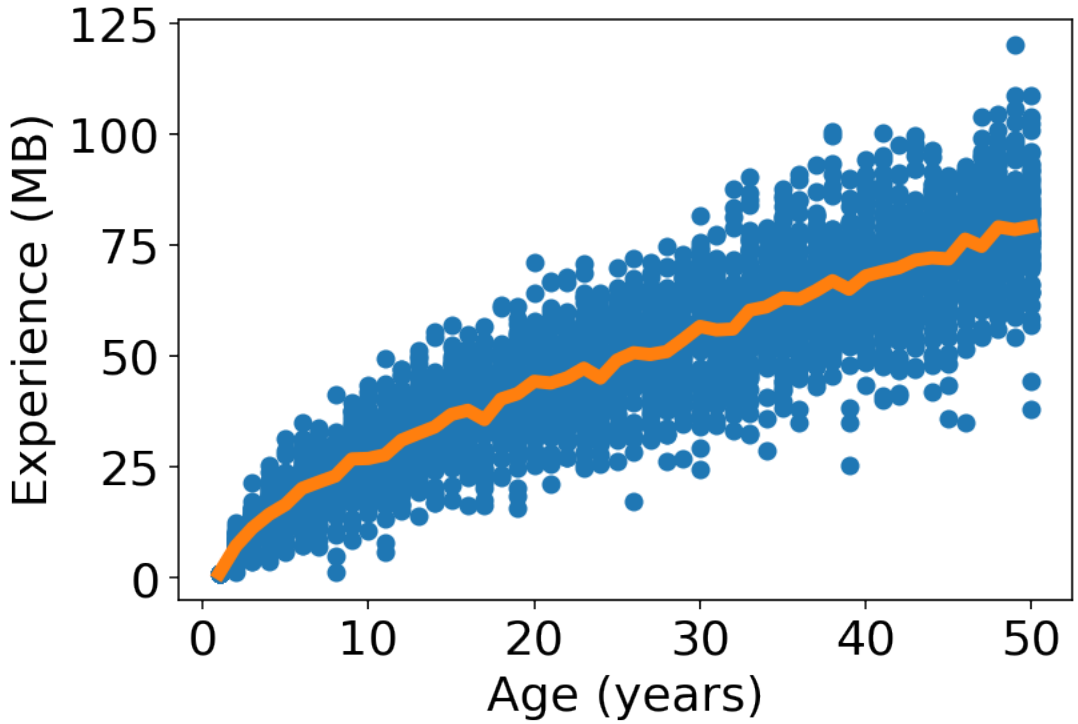
[26]:
```
# MOAR fake data:
Xs = []
Ys = []

for x in range(1,50+1):
    for _ in range(75):
        y = x + 3*np.log(x)*(np.random.randn()+2.5)
        Xs.append(x)
        Ys.append(y)

plt.plot(Xs,Ys,'o')
plt.xlabel("Age (years)")
plt.ylabel("Experience (MB)")
plt.show()
```



Then you can "bin" the y-data based on each unique x-value:

- Let's take each unique value of x and make a list of all corresponding y-values, then compute the **mean** of each list:

[27]:
```
# build list of all y's for each x:
x2listY = {}
for x,y in zip(Xs,Ys):
    try:
        x2listY[x].append(y)
    except KeyError:
        x2listY[x] = [y]

# compute mean of each list, break into separate lists
# for plotting:
Xs_line = sorted(x2listY.keys())
Ys_line = []
for x in Xs_line:
    corresponding_ys = x2listY[x]
    meanY = np.mean(corresponding_ys)
    Ys_line.append(meanY)

# faster/shorter/less easy to understand
#x2meanY = { x : np.mean(x2listY[x]) for x in x2listY }
#x_meanY = sorted(x2meanY.items())
#Xs_line,Ys_line = zip(*x_meanY)

# plot averaged trend on top of "raw" scatter:
plt.plot(Xs,Ys, 'o')
plt.plot(Xs_line,Ys_line, '-', linewidth=5)
plt.xlabel("Age (years)")
plt.ylabel("Experience (MB)")
plt.show()
```



(This is also a great time for a `groupby` if you are so inclined.)

(And of course, in practice we wouldn't repeat these code blocks, we would update the previous code block to include the trend line. These code blocks are *unpacked* for the lecture.)

If your x-values are continuous
(or the data are noisy or you only have few points) you can **bin them** just like a histogram.

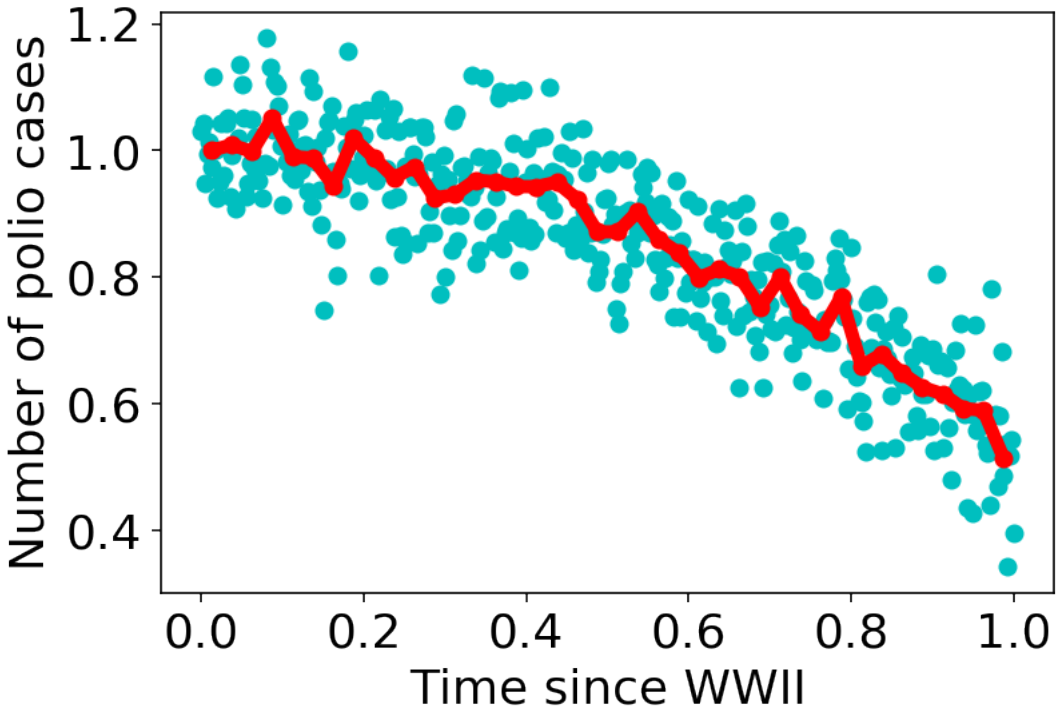
Let's take all the pairs of data and **bin them** by their x-value. Then instead of counting how many pairs fall into a bin (which would be a histogram) let's compute the means of their y-values.

In *pseudocode*, for one bin:

```
data = [(x1,y1),(x2,y2), ...]

this_bin_l = 0.0
this_bin_r = 0.1

y_values_for_this_bin = []
for x,y in data:
    if this_bin_l <= x < this_bin_r:
        y_values_for_this_bin.append(y)

x_this_bin = (this_bin_l + this_bin_r)/2 # bin center
y_this_bin = mean(y_values_for_this_bin)

# repeat for all bins
```

And to do this **for real**:

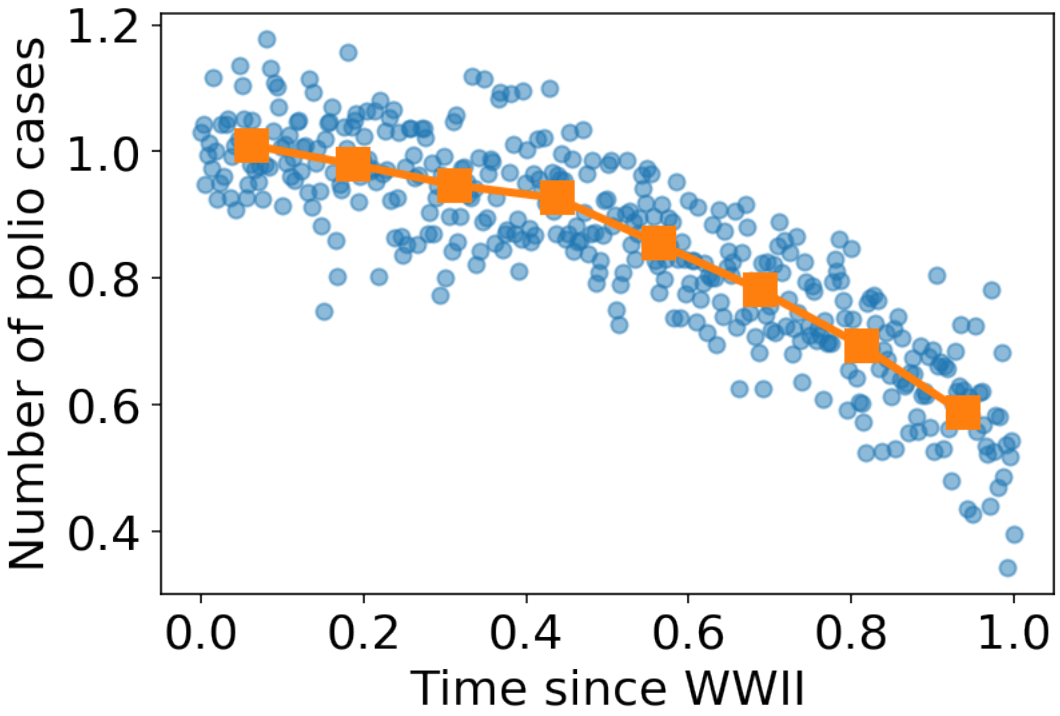
[28]:
```
import scipy.stats

binned_stat = scipy.stats.binned_statistic

# nice builtin function!
y_bins,bin_edges, misc = binned_stat(X,Y, statistic="mean", bins=40)

# but this function doesn't return the bin CENTERS:
x_bins = (bin_edges[:-1]+bin_edges[1:])/2


plt.plot(X,Y, 'co')
plt.plot(x_bins, y_bins, "ro-", linewidth=5)


plt.xlabel("Time since WWII") # fake!
plt.ylabel("Number of polio cases")
plt.show()
```



OK, that's a little noisy still, let's **dial back** the number of bins...

[29]:
```
y_bins,bin_edges, misc = binned_stat(X,Y, statistic="mean", bins=8)
x_bins = (bin_edges[:-1]+bin_edges[1:])/2



plt.plot(X,Y, 'o', alpha=0.5)
plt.plot(x_bins, y_bins, "s-", linewidth=3, markersize=12)
```

```
plt.xlabel("Time since WWII")
plt.ylabel("Number of polio cases")
plt.show()
```



There exist many other techniques, especially when the x-values are NOT noisy, but this works well in many situations.

- **Moving** (or rolling) **averages** are also common, although they can sometimes introduce artifacts at the edges of a dataset's domain, especially if data are sparse there.
- It's especially important when there's so many points you just see a cloud of data. When the points are **so dense they overlap**, you can't see the **underlying density** with a scatterplot:

Binning on the x lets us use our one-dimensional data analysis **toolbox** repeatedly on small segments of the y-data:

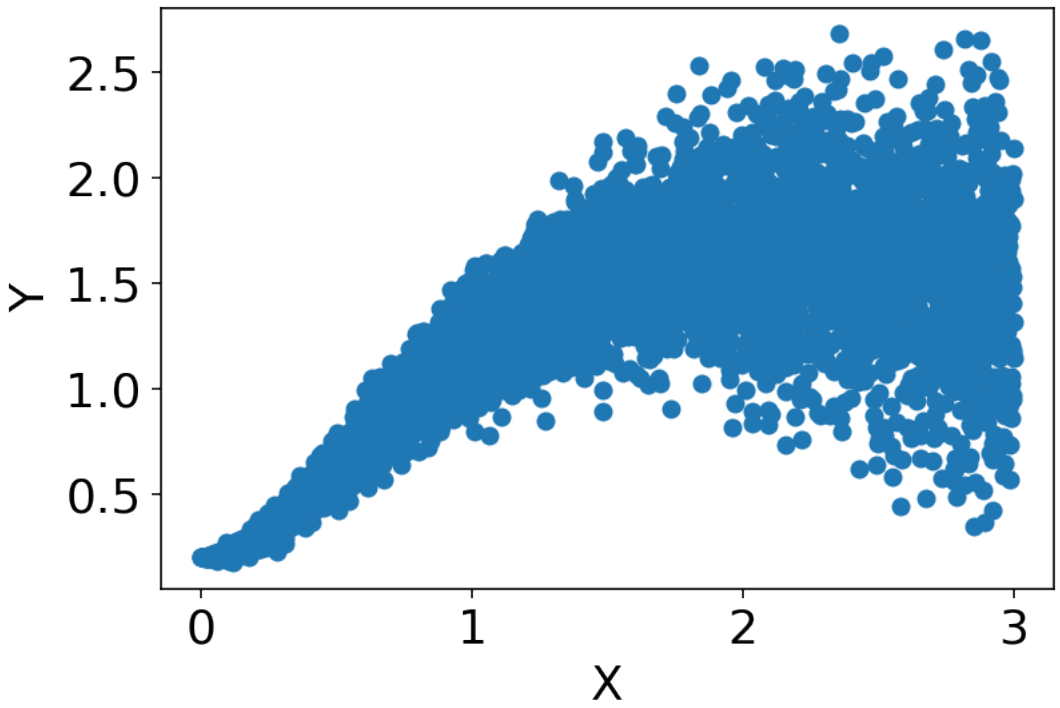
- When the x-values are discrete we can look at all the **unique** x-values.
- We can compute the mean of all y-values in an x-bin, but we can also use the *median*, *percentiles*, or really any measure of *central tendency*.

Error bars: measures of the *spread* of the y-values in a bin can tell us how broad the data are.

First, let's make a scatter plot:

[30]:
```
n = 5000
x = np.linspace(0,3,n)

x_data = x
y_data = np.exp(-x+1)*x**2 + 0.15*(np.random.randn(n))*x + 0.2
```

```
plt.plot(x_data,y_data, 'o')

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

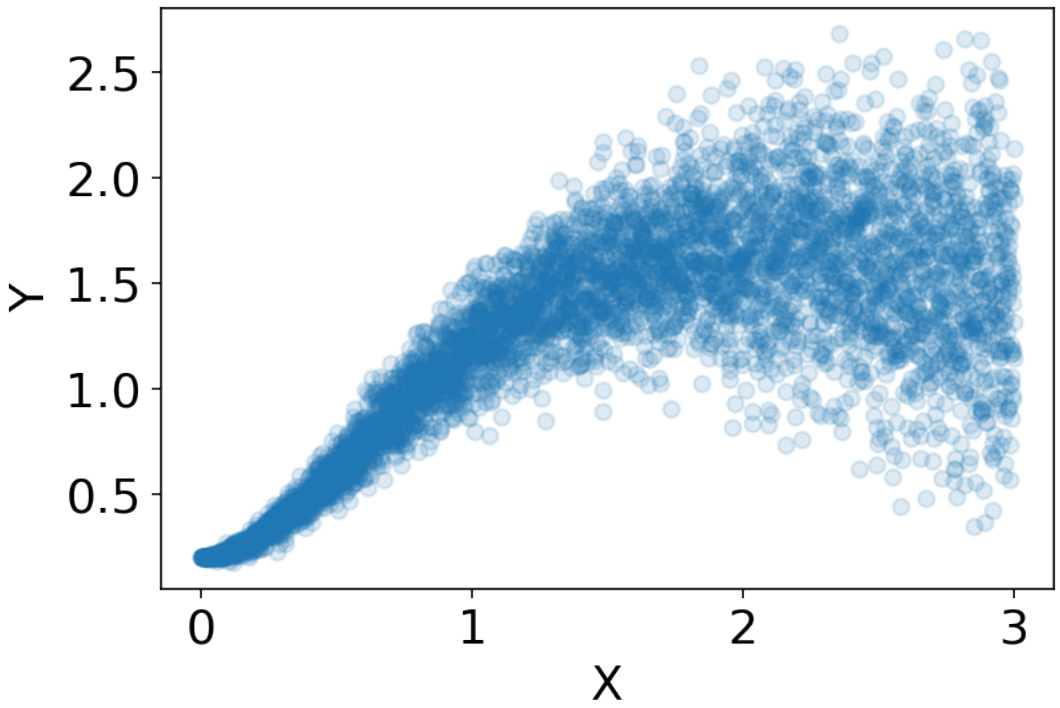


Notice how the data spread out more as x increases?

Also notice how it's hard to see what's going on inside the plot—it's just a wall of points.

- A bit of ***transparency*** can sometimes help with so much scatter overlap:

[31]:
```
plt.plot(x_data,y_data, 'o', alpha=0.15)
#                            ^^^^^^^^^
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

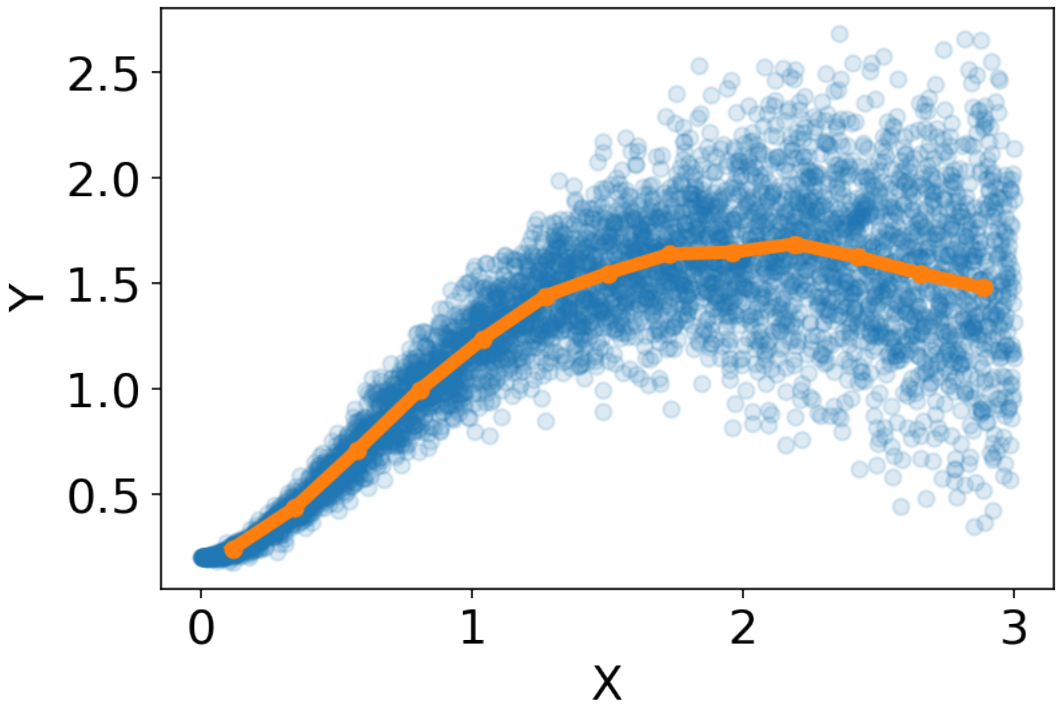


We can emphasize this higher density in the middle by adding the **binned trend**:

- Binned trend is also helpful when we want to avoid transparency, for whatever reason.

(Again, we wouldn't duplicate this code in practice—this is just for the lecture.)

[32]:
```
y_bins,bin_edges, misc = binned_stat(x_data,y_data, statistic="mean", bins=13)
x_bins = (bin_edges[:-1]+bin_edges[1:])/2


plt.plot(x_data,y_data, 'o', alpha=0.15)
plt.xlabel("X"); plt.ylabel("Y")

plt.plot(x_bins, y_bins, "o-", linewidth=5);
```



Now let's do the same thing but instead of computing the **mean** of each bin, let's compute the **standard deviation**.

One can represent the mean, or average, or **expected value** of a random variable X as:

$$\mathrm{E}(X) \text{ or } \langle X \rangle .$$

For a discrete collection of values x this becomes the familiar sum

$$\langle x \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

where N is the number of data points.

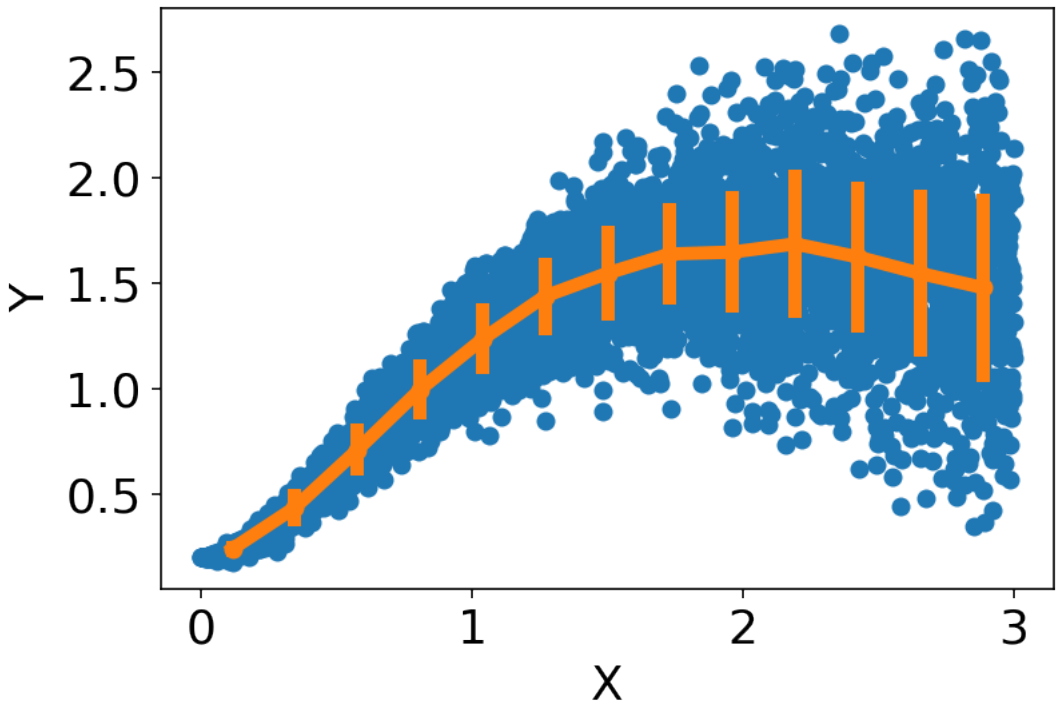
The **standard deviation** σ of that data is related to the mean of the square of the data:

$$\sigma = \sqrt{\left\langle \left(x - \langle x \rangle\right)^2 \right\rangle} = \sqrt{\langle x^2 \rangle - \langle x \rangle^2}$$

To the codez!

[33]:
```
s_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                     statistic=np.std, bins=13)
#                                      ^^^^^^
x_bins = (bin_edges[:-1]+bin_edges[1:])/2

# plot the raw data
plt.plot(x_data,y_data, 'o')
```

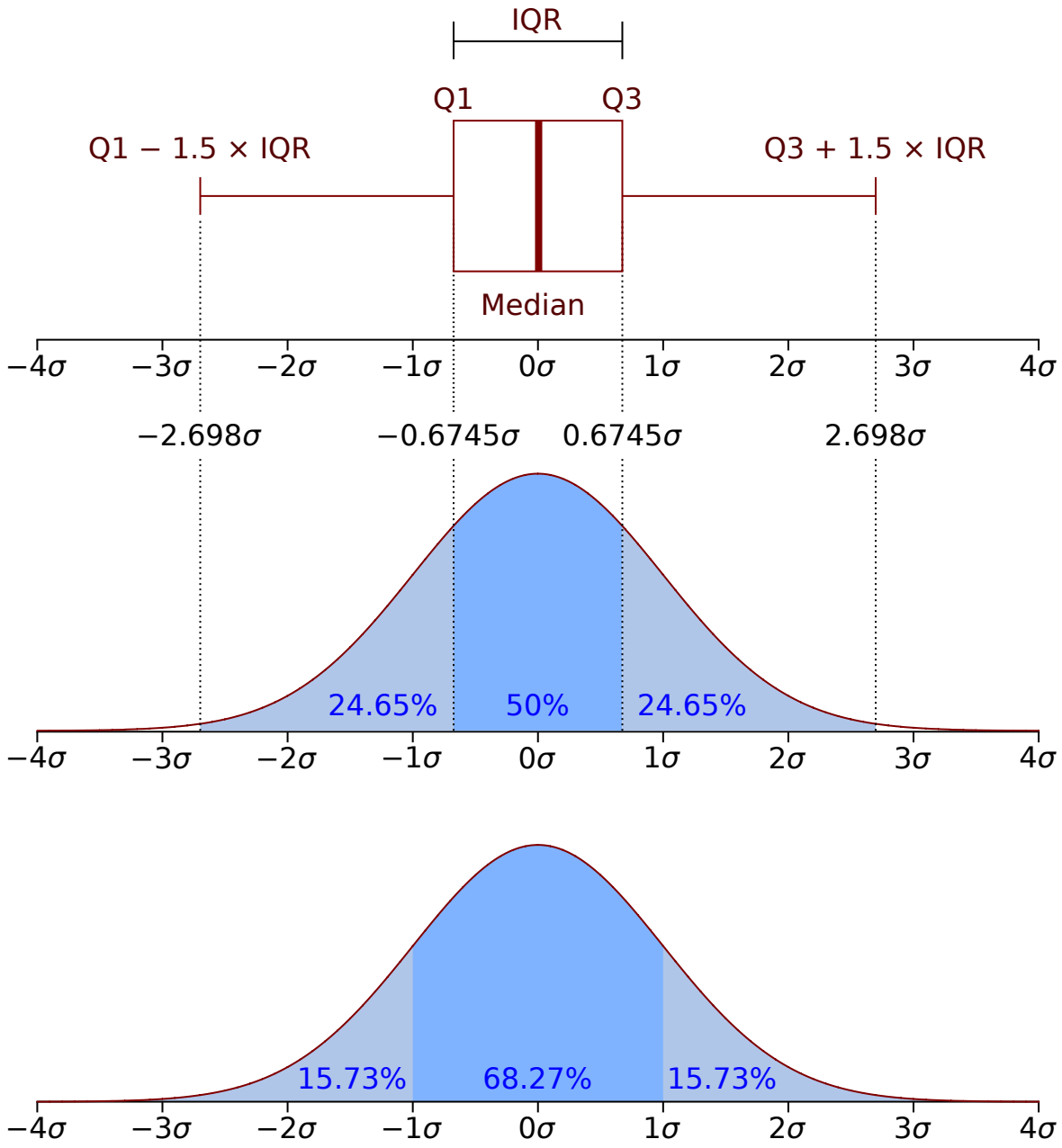
```
# plot the means of each bin:
plt.plot(x_bins, y_bins, "o-", linewidth=5)

# Now, plot the std as errorbars! (check out the docstring)
plt.errorbar( x_bins, y_bins, yerr=s_bins,
              fmt="none",     # only show errorbars
              elinewidth=5,
              ecolor='C1',
              zorder=5      # plot bars ABOVE scatter
             )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



This idea of using **standard deviation** for errorbars (whiskers) brings box plots and the normal distribution back into play:

[34]:
```
SVG("figures/Boxplot_vs_PDF.svg")
```

[34]:



We've combined the **mean** of the data with the **standard deviation** in an **errorbar plot**. This is a great way for seeing both trends and spreads at once, assuming you can interpret it.

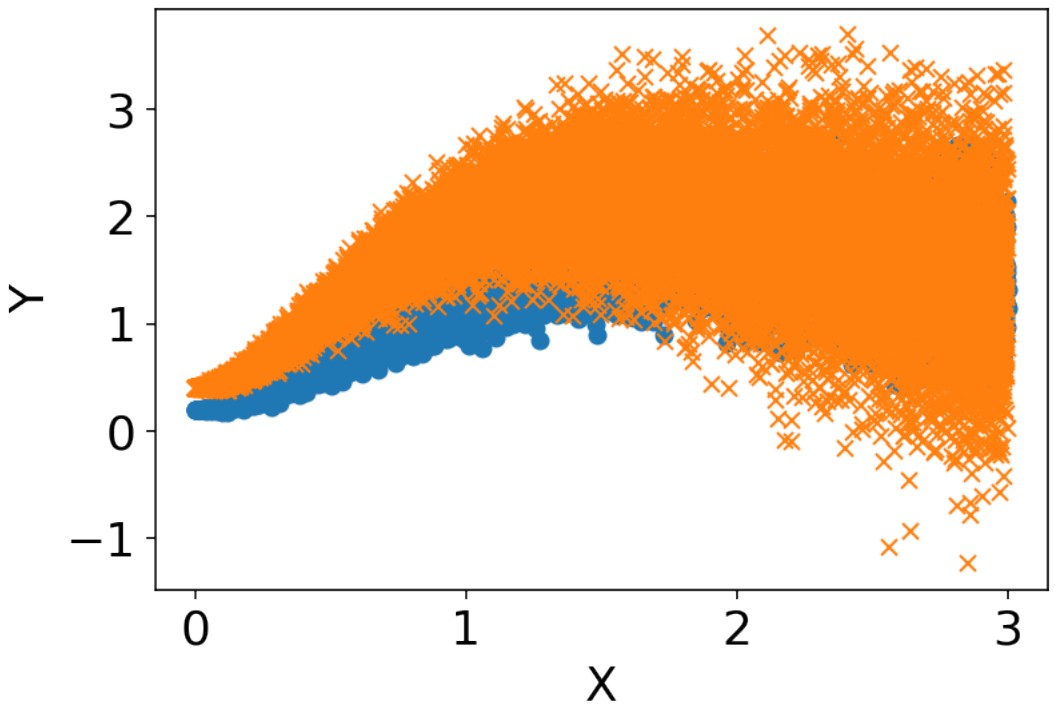
- We can intepret an errorbar of ± 1 std as representing the middle ≈ 68 of the data ***if*** the data within each bin are normally distributed.
- Also, a trend line composed of **box plots** can make a great choice for visualization purposes (see below).

Summarizing multiple scatter plots

Now, in the scatter plot above, all the trend line really shows us is a different view of the same information. The spread of the data as x increases is already obvious from the scatter. But the reduction from a huge **blob** of points let's us avoid mess.

- For example, compare this plot:

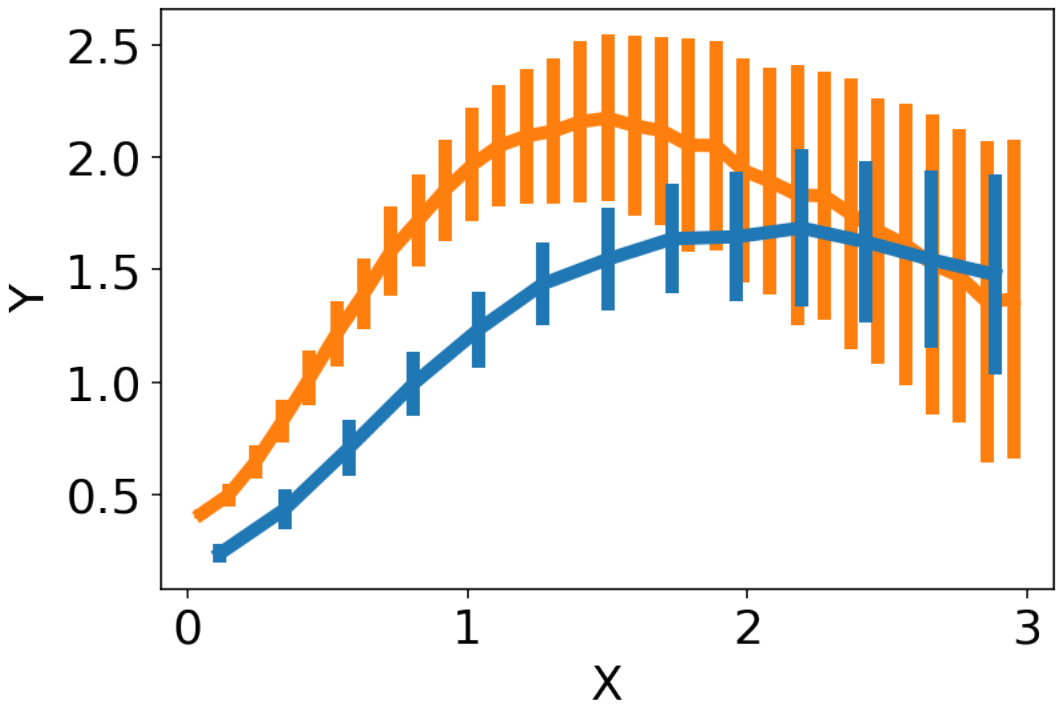
[35]:
```
# second set of points:
x_data2 = np.linspace(0,3,20000)
x2 = x_data2
y_data2 = np.exp(-x2*1.4+1)*(x2*1.5)**2.05 + \
                  0.25*(np.random.randn(20000))*x2 + 0.4

# plot two scatter plots
plt.plot(x_data, y_data , 'o', x_data2,y_data2, 'x')

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



with this plot

[36]:
```
def helper_binned_trendline(x_data, y_data, bins=10, ystat='mean', sstat='std'):
    if sstat == "std":
        sstat = np.std

    y_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                         statistic=ystat, bins=bins)
    s_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                         statistic=sstat, bins=bins)
    x_bins = (bin_edges[:-1]+bin_edges[1:])/2

    return x_bins, y_bins, s_bins
```

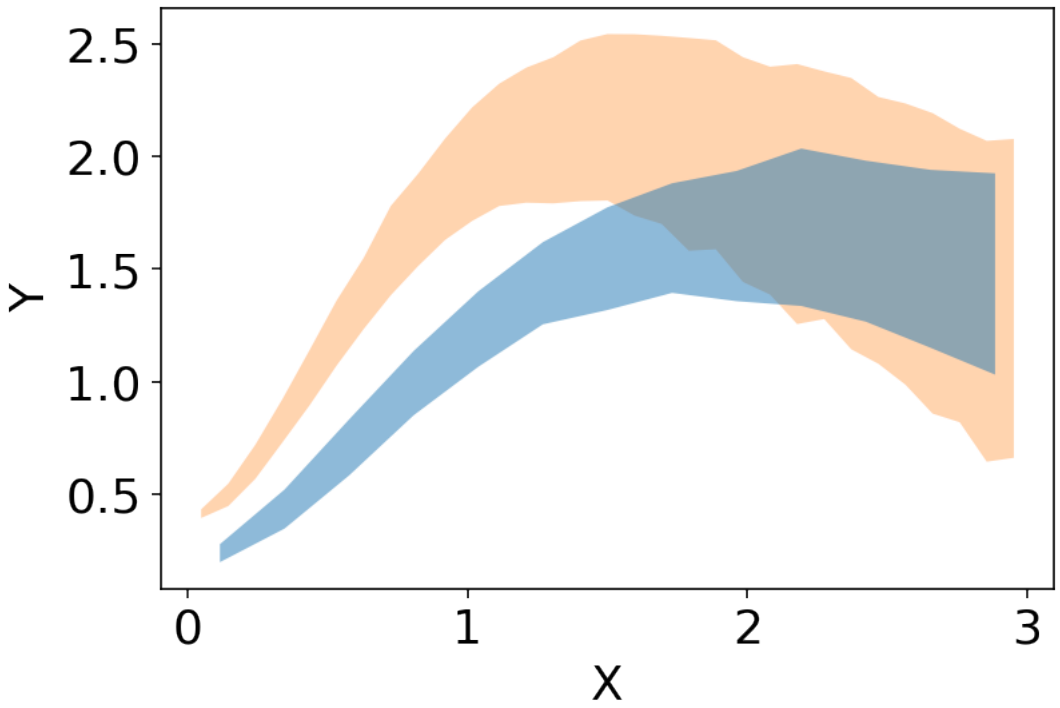
```
# redo the stats for the original blue data:
x_bins, y_bins, s_bins = helper_binned_trendline(x_data, y_data,
                                                 bins=13)

# now the stats for the new orange data:
x_bins2, y_bins2, s_bins2 = helper_binned_trendline(x_data2, y_data2,
                                                    bins=31)


# plot original (blue) curve
plt.errorbar( x_bins, y_bins, yerr=s_bins,
              linewidth=5,
              color='C0', zorder=15 )

# now add the new (orange) curve
plt.errorbar( x_bins2, y_bins2, yerr=s_bins2,
              linewidth=5,
              color='C1', zorder=10 )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



More readable now that we can see the curve in the back

And instead of using vertical bars, we can used **filled curves**:

[37]:
```
# blue data
plt.fill_between(x_bins, y_bins - s_bins, y_bins + s_bins,
                 facecolor='C0', alpha=0.5, linewidth=0
                ) # 50% transparent blue, no edge

# orange data
plt.fill_between(x_bins2, y_bins2 - s_bins2, y_bins2 + s_bins2,
                 facecolor='C1', alpha=0.33, linewidth=0,
                 zorder=-3
                )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



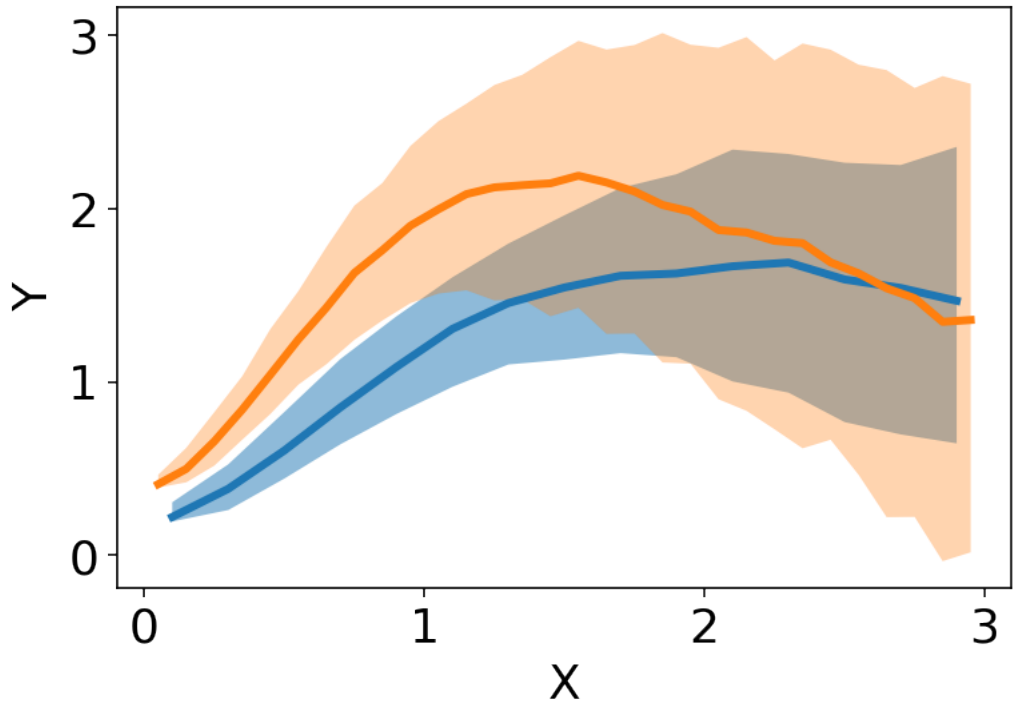
(This is just a visual flourish; a different way to present the same information.)

This plot lets us see the mean ± one standard deviation. But *we are not limited to these statistics*.

- Let's look at the median and 95% confidence intervals (CI):

[38]:
```
def prctile_025(data):
    return np.percentile(data,2.5)

def prctile_975(data):
    return np.percentile(data,97.5)

def xyBin_md_95ci(xdata,ydata, num_bins):
    """Return xbincenters,ymedian,y025,y975."""
```

```
    y_md, be, m = binned_stat(xdata,ydata, statistic="median",    bins=num_bins)
    y_025,be, m = binned_stat(xdata,ydata, statistic=prctile_025, bins=num_bins)
    y_975,be, m = binned_stat(xdata,ydata, statistic=prctile_975, bins=num_bins)
    x_bins = (be[:-1]+be[1:])/2
    return x_bins, y_md, y_025, y_975 # another helper function


# now do the calc:
x1,y1_md,y1_025,y1_975 = xyBin_md_95ci(x_data, y_data,  15)
x2,y2_md,y2_025,y2_975 = xyBin_md_95ci(x_data2,y_data2, 30)


# blue data
plt.fill_between(x1, y1_025, y1_975, facecolor='C0', alpha=0.5, linewidth=0 )
plt.plot(x1,y1_md, '-', linewidth=3)

# red data
plt.fill_between(x2, y2_025, y2_975, facecolor='C1', alpha=0.33, linewidth=0 )
plt.plot(x2,y2_md, '-', linewidth=3)

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

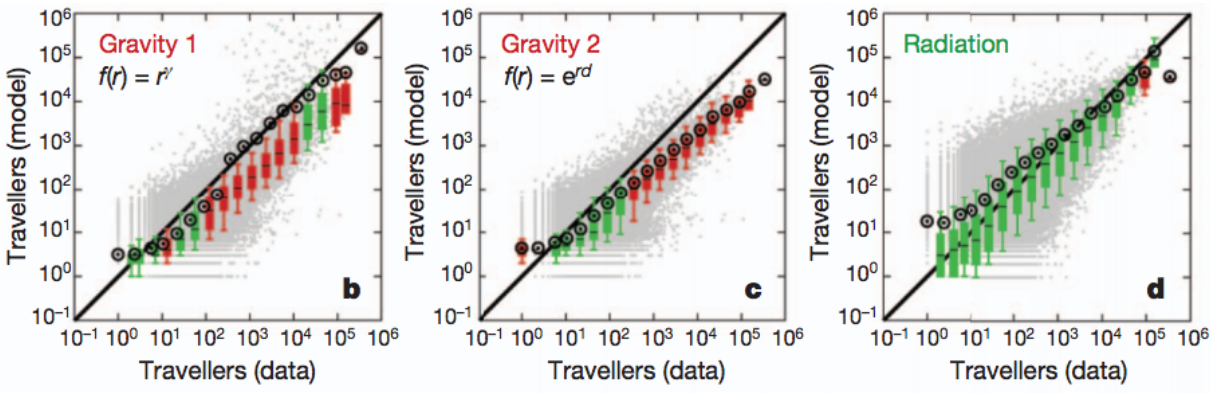


Now the shaded region represents 95% of the (observed) data, regardless of the underlying distribution. Moreover using separate (2.5,97.5%) CI (I'm abusing the term "confidence interval" a bit here) lets us represent **nonsymmetric** or **skewed** data.

Example in practice

Combining scatter plots, box plots, xy-binning and summary statistics with **color** leads to a concise but information-rich graphic:

[39]:
```
Image("figures/simini_nature_scatterplot_examples.png", width=800)
```

[39]:


(Simini *et al*, Nature 2012)

Here the predictions for different modeling approaches are compared using a scatter plot. The more accurate the predictions, the closer to the line of equality ($y = x$), drawn in black.

The raw scatter (light gray) shows a lot of spread, so a trend line of box plots was used to show where the "bulk" of the points lie. Black circles represent the mean trend.

Boxes that go through the line of equality are colored **green**, otherwise colored **red**, to show how close the central tendency of the data are to the line of equality.

- Red = bad, green = good makes this choice of colors nice. But red-green colorblindness is the most common form of color blindness, so you do not want to rely upon these two colors that much

---

Takeaways

- **Boxplots** - May be a leftover artifact of limited computing resources but still useful for comparing many distributions
- **CDF** - Cumulative Distribution Function - sorting is integrating (!) - MANY advantages(!)
  - I never explore data with histograms. I only use (E)CDFs!
- **Scatter plots** - a primary tool for comparing pairs of variables!

X-data
If you have X-data you want to understand, think about summarizing it using it's statistics (mean, median, IQR, etc.) and drawing the underlying distribution (histogram, KDE, etc.)

XY-data
If you have XY-data, think about visualizing a trend of relationship between X and Y using a plain old boring scatter plot. ← much more to come along these lines!

To understand the relationship between two variables, you've accomplished a **great deal** by just understanding how the **average** and **spread** of one variable depends on the other.