

# Table of Contents

Introduction	0
Data Science	1
Introduction	1.1
Computational Tools	1.1.1
Statistical Techniques	1.1.2
Why Data Science?	1.2
Example: Plotting the Classics	1.2.1
Causality and Experiments	1.3
Observation and Visualization: John Snow and the Broad Street Pump	1.3.1
Snow's "Grand Experiment"	1.3.2
Establishing Causality	1.3.3
Randomization	1.3.4
Endnote	1.3.5
Expressions	1.4
Names	1.4.1
Example: Growth Rates	1.4.2
Call Expressions	1.4.3
Data Types	1.4.4
Comparisons	1.4.5
Sequences	1.5
Arrays	1.5.1
Ranges	1.5.2
Tables	1.6
Functions	1.6.1
Functions and Tables	1.6.2
Visualization	2
Bar Charts	2.1
Histograms	2.2
Sampling	3
Sampling	3.1

Iteration	3.2
Estimation	3.3
Center	3.4
Spread	3.5
The Normal Distribution	3.6
Exploration: Privacy	3.7
Prediction	4
Correlation	4.1
Regression	4.2
Prediction	4.3
Higher-Order Functions	4.4
Errors	4.5
Multiple Regression	4.6
Classification	4.7
Features	4.8
Inference	5
Confidence Intervals	5.1
Distance Between Distributions	5.2
Hypothesis Testing	5.3
Permutation	5.4
A/B Testing	5.5
Regression Inference	5.6
Regression Diagnostics	5.7
Probability	6
Conditional Probability	6.1

# Computational and Inferential Thinking

## The Foundations of Data Science

By [Ani Adhikari](#) and [John DeNero](#)

Contributions by [David Wagner](#)

This is the textbook for the [Foundations of Data Science](#) class at UC Berkeley.

[View this textbook online on Gitbooks.](#)

[launch](#) [binder](#)

# What is Data Science

Data Science is about drawing useful conclusions from large and diverse data sets through exploration, prediction, and inference. Exploration involves identifying patterns in information. Prediction involves using information we know to make informed guesses about values we wish we knew. Inference involves quantifying our degree of certainty: will those patterns we found also appear in new observations? How accurate are our predictions? Our primary tools for exploration are visualizations, for prediction are machine learning and optimization, and for inference are statistical tests and models.

Statistics is a central component of data science because statistics studies how to make robust conclusions with incomplete information. Computing is a central component because programming allows us to apply analysis techniques to the large and diverse data sets that arise in real-world applications: not just numbers, but text, images, videos, and sensor readings. Data science is all of these things, but it more than the sum of its parts because of the applications. Through understanding a particular domain, data scientists learn to ask appropriate questions about their data and correctly interpret the answers provided by our inferential and computational tools.

# Chapter 1: Introduction

Data are descriptions of the world around us, collected through observation and stored on computers. Computers enable us to infer properties of the world from these descriptions. Data science is the discipline of drawing conclusions from data using computation. There are three core aspects of effective data analysis: exploration, prediction, and inference. This text develops a consistent approach to all three, introducing statistical ideas and fundamental ideas in computer science concurrently. We focus on a minimal set of core techniques that they apply to a vast range of real-world applications. A foundation in data science requires not only understanding statistical and computational techniques, but also recognizing how they apply to real scenarios.

For whatever aspect of the world we wish to study---whether it's the Earth's weather, the world's markets, political polls, or the human mind---data we collect typically offer an incomplete description of the subject at hand. The central challenge of data science is to make reliable conclusions using this partial information.

In this endeavor, we will combine two essential tools: computation and randomization. For example, we may want to understand climate change trends using temperature observations. Computers will allow us to use all available information to draw conclusions. Rather than focusing only on the average temperature of a region, we will consider the whole range of temperatures together to construct a more nuanced analysis. Randomness will allow us to consider the many different ways in which incomplete information might be completed. Rather than assuming that temperatures vary in a particular way, we will learn to use randomness as a way to imagine many possible scenarios that are all consistent with the data we observe.

Applying this approach requires learning to program a computer, and so this text interleaves a complete introduction to programming that assumes no prior knowledge. Readers with programming experience will find that we cover several topics in computation that do not appear in a typical introductory computer science curriculum. Data science also requires careful reasoning about quantities, but this text does not assume any background in mathematics or statistics beyond basic algebra. You will find very few equations in this text. Instead, techniques are described to readers in the same language in which they are described to the computers that execute them---a programming language.

# Computational Tools

This text uses the Python 3 programming language, along with a standard set of numerical and data visualization tools that are used widely in commercial applications, scientific experiments, and open-source projects. Python has recruited enthusiasts from many professions that use data to draw conclusions. By learning the Python language, you will join a million-person-strong community of software developers and data scientists.

**Getting Started.** The easiest and recommended way to start writing programs in Python is to log into the companion site for this text, [data8.berkeley.edu](http://data8.berkeley.edu). If you are enrolled in a course offering that uses this text, you should have full access to the programming environment hosted on that site. The first time you log in, you will find a brief tutorial describing how to proceed.

You are not at all restricted to using this web-based programming environment. A Python program can be executed by any computer, regardless of its manufacturer or operating system, provided that support for the language is installed. If you wish to install the version of Python and its accompanying libraries that will match this text, we recommend the [Anaconda](#) distribution that packages together the Python 3 language interpreter, IPython libraries, and the Jupyter notebook environment.

This text includes a complete introduction to all of these computational tools. You will learn to write programs, generate images from data, and work with real-world data sets that are published online.

# Statistical Techniques

The discipline of statistics has long addressed the same fundamental challenge as data science: how to draw conclusions about the world using incomplete information. One of the most important contributions of statistics is a consistent and precise vocabulary for describing the relationship between observations and conclusions. This text continues in the same tradition, focusing on a set of core inferential problems from statistics: testing hypotheses, estimating confidence, and predicting unknown quantities.

Data science extends the field of statistics by taking full advantage of computing, data visualization, and access to information. The combination of fast computers and the Internet gives anyone the ability to access and analyze vast datasets: millions of news articles, full encyclopedias, databases for any domain, and massive repositories of music, photos, and video.

Applications to real data sets motivate the statistical techniques that we describe throughout the text. Real data often do not follow regular patterns or match standard equations. The interesting variation in real data can be lost by focusing too much attention on simplistic summaries such as average values. Computers enable a family of methods based on resampling that apply to a wide range of different inference problems, take into account all available information, and require few assumptions or conditions. Although these techniques have often been reserved for graduate courses in statistics, their flexibility and simplicity are a natural fit for data science applications.

# Why Data Science?

Most important decisions are made with only partial information and uncertain outcomes. However, the degree of uncertainty for many decisions can be reduced sharply by public access to large data sets and the computational tools required to analyze them effectively. Data-driven decision making has already transformed a tremendous breadth of industries, including finance, advertising, manufacturing, and real estate. At the same time, a wide range of academic disciplines are evolving rapidly to incorporate large-scale data analysis into their theory and practice.

Studying data science enables individuals to bring these techniques to bear on their work, their scientific endeavors, and their personal decisions. Critical thinking has long been a hallmark of a rigorous education, but critiques are often most effective when supported by data. A critical analysis of any aspect of the world, may it be business or social science, involves inductive reasoning --- conclusions can rarely be proven outright, only supported by the available evidence. Data science provides the means to make precise, reliable, and quantitative arguments about any set of observations. With unprecedented access to information and computing, critical thinking about any aspect of the world that can be measured would be incomplete without the inferential techniques that are core to data science.

The world has too many unanswered questions and difficult challenges to leave this critical reasoning to only a few specialists. However, all educated adults can build the capacity to reason about data. The tools, techniques, and data sets are all readily available; this text aims to make them accessible to everyone.

# Example: Plotting the Classics

## Interact

In this example, we will explore statistics for two classic novels: *The Adventures of Huckleberry Finn* by Mark Twain, and *Little Women* by Louisa May Alcott. The text of any book can be read by a computer at great speed. Books published before 1923 are currently in the *public domain*, meaning that everyone has the right to copy or use the text in any way. [Project Gutenberg](#) is a website that publishes public domain books online. Using Python, we can load the text of these books directly from the web.

The features of Python used in this example will be explained in detail later in the course. This example is meant to illustrate some of the broad themes of this text. Don't worry if the details of the program don't yet make sense. Instead, focus on interpreting the images generated below. The "Expressions" section later in this chapter will describe most of the features of the Python programming language used below.

First, we read the text of both books into lists of chapters, called `huck_finn_chapters` and `little_women_chapters`. In Python, a name cannot contain any spaces, and so we will often use an underscore `_` to stand in for a space. The `=` in the lines below give a name on the left to the result of some computation described on the right. A *uniform resource locator* or *URL* is an address on the Internet for some content; in this case, the text of a book.

```
# Read two books, fast!

huck_finn_url = 'http://www.gutenberg.org/cache/epub/76/pg76.txt'
huck_finn_text = read_url(huck_finn_url)
huck_finn_chapters = huck_finn_text.split('CHAPTER ')[44:]

little_women_url = 'http://www.gutenberg.org/cache/epub/514/pg514.t
little_women_text = read_url(little_women_url)
little_women_chapters = little_women_text.split('CHAPTER ')[1:]
```

While a computer cannot understand the text of a book, we can still use it to provide us with some insight into the structure of the text. The name `huck_finn_chapters` is currently bound to a list of all the chapters in the book. We can place those chapters into a table to see how each begins.

```
Table([huck_finn_chapters], ['Chapters'])
```

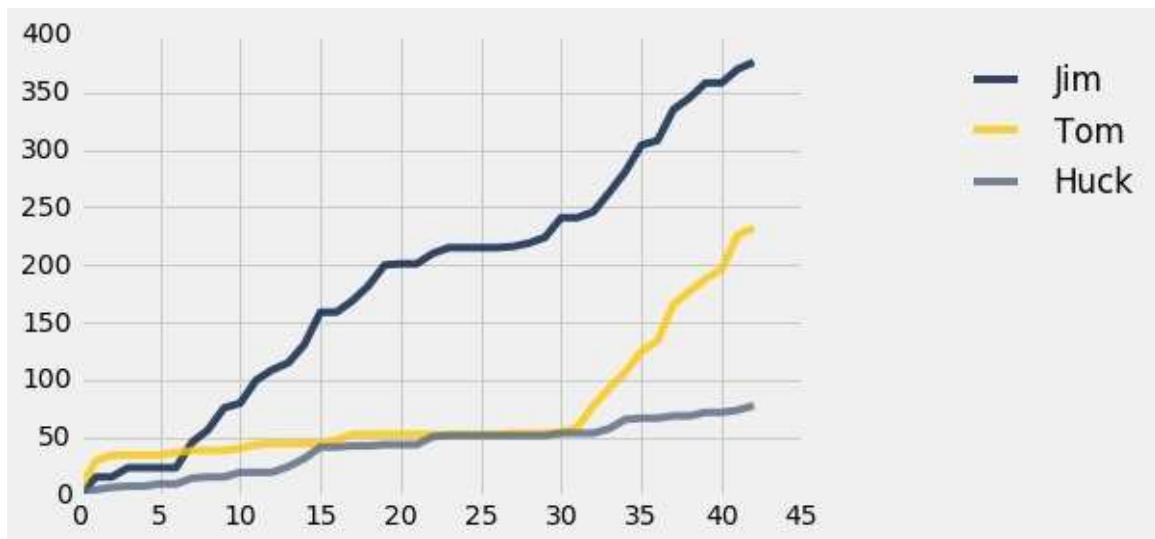
Chapters
I. YOU don't know about me without you have read a book ...
II. WE went tiptoeing along a path amongst the trees bac ...
III. WELL, I got a good going-over in the morning from o ...
IV. WELL, three or four months run along, and it was wel ...
V. I had shut the door to. Then I turned around and ther ...
VI. WELL, pretty soon the old man was up and around agai ...
VII. "GIT up! What you 'bout?" I opened my eyes and look ...
VIII. THE sun was up so high when I waked that I judged ...
IX. I wanted to go and look at a place right about the m ...
X. AFTER breakfast I wanted to talk about the dead man a ...

... (33 rows omitted)

Each chapter begins with a chapter number in Roman numerals, followed by the first sentence of the chapter. Project Gutenberg has printed the first word of each chapter in upper case.

The book describes a journey that Huck and Jim take along the Mississippi River. Tom Sawyer joins them towards the end as the action heats up. We can quickly visualize how many times these characters have each been mentioned at any point in the book.

```
counts = Table([np.char.count(huck_finn_chapters, "Jim"),
               np.char.count(huck_finn_chapters, "Tom"),
               np.char.count(huck_finn_chapters, "Huck")],
               ["Jim", "Tom", "Huck"])
counts.cumsum().plot(overlay=True)
```



In the plot above, the horizontal axis shows chapter numbers and the vertical axis shows how many times each character has been mentioned so far. You can see that Jim is a central character by the large number of times his name appears. Notice how Tom is hardly mentioned for much of the book until after Chapter 30. His curve and Jim's rise sharply at that point, as the action involving both of them intensifies. As for Huck, his name hardly appears at all, because he is the narrator.

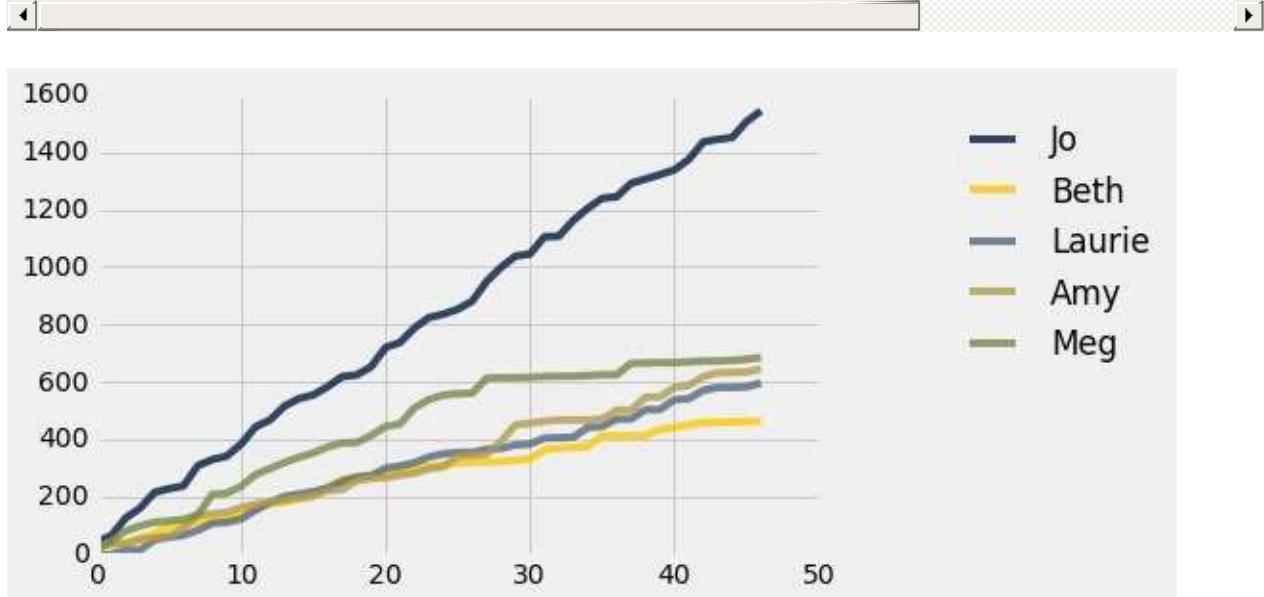
*Little Women* is a story of four sisters growing up together during the civil war. In this book, chapter numbers are spelled out and chapter titles are written in all capital letters.

```
Table({'Chapters': little_women_chapters})
```

Chapters
ONE PLAYING PILGRIMS "Christmas won't be Christmas witho ...
TWO A MERRY CHRISTMAS Jo was the first to wake in the gr ...
THREE THE LAURENCE BOY "Jo! Jo! Where are you?" cried Me ...
FOUR BURDENS "Oh, dear, how hard it does seem to take up ...
FIVE BEING NEIGHBORLY "What in the world are you going t ...
SIX BETH FINDS THE PALACE BEAUTIFUL The big house did pr ...
SEVEN AMY'S VALLEY OF HUMILIATION "That boy is a perfect ...
EIGHT JO MEETS APOLLYON "Girls, where are you going?" as ...
NINE MEG GOES TO VANITY FAIR "I do think it was the most ...
TEN THE P.C. AND P.O. As spring came on, a new set of am ...
... (37 rows omitted)

We can track the mentions of main characters to learn about the plot of this book as well. The protagonist Jo interacts with her sisters Meg, Beth, and Amy regularly, up until Chapter 27 when she moves to New York alone.

```
people = ["Meg", "Jo", "Beth", "Amy", "Laurie"]
people_counts = Table({pp: np.char.count(little_women_chapters, pp)}
people_counts.cumsum().plot(overlay=True)
```



Laurie is a young man who marries one of the girls in the end. See if you can use the plots to guess which one.

In addition to visualizing data to find patterns, data science is often concerned with whether two similar patterns are really the same or different. In the context of novels, the word "character" has a second meaning: a printed symbol such as a letter or number. Below, we compare the counts of this type of character in the first chapter of Little Women.

```
from collections import Counter

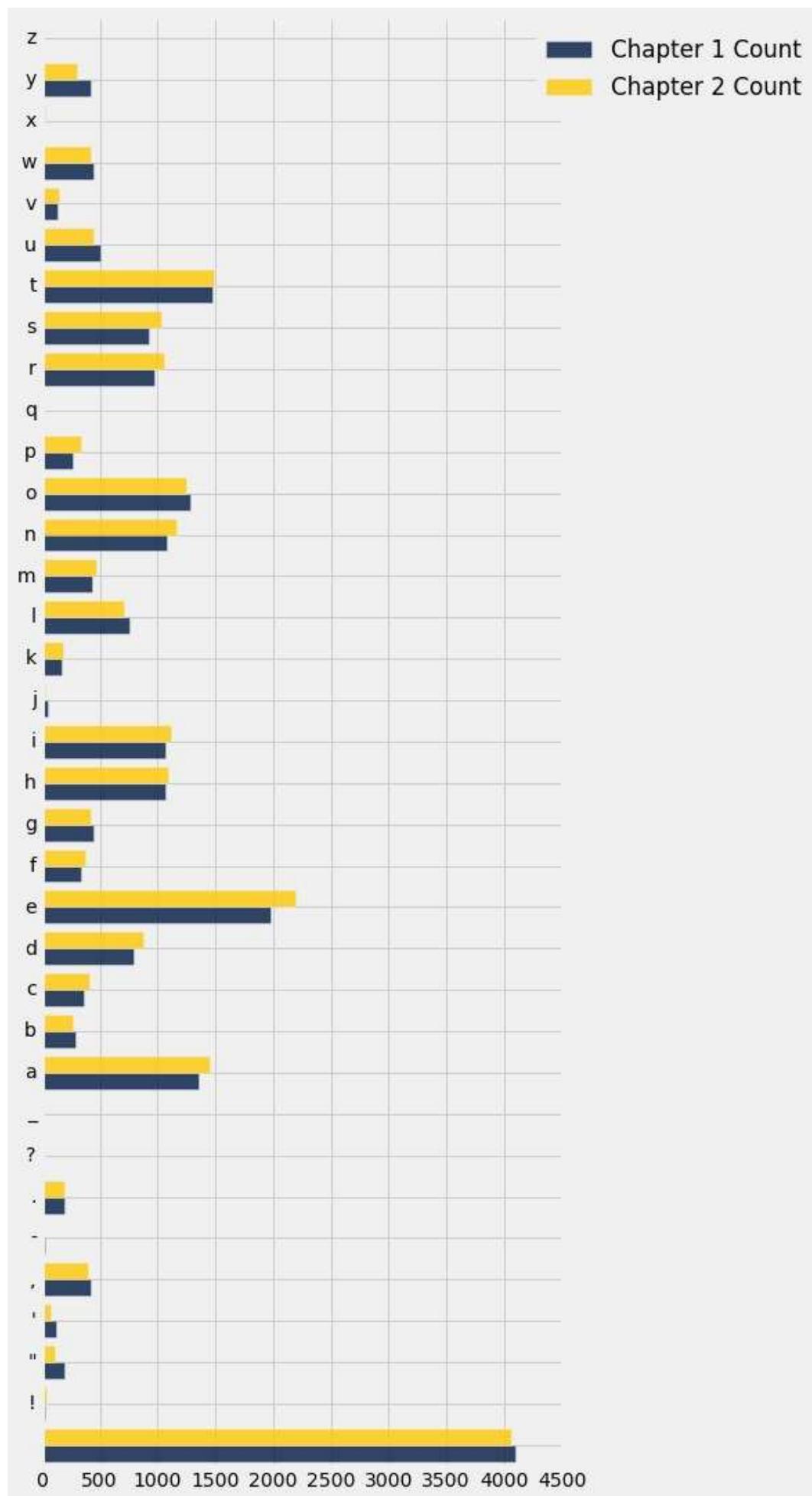
chapter_one = little_women_chapters[0]
counts = Counter(chapter_one.lower())
letters = Table([counts.keys(), counts.values()], ['Letters', 'Counts'])
letters.sort('Letters').show(20)
```

<b>Letters</b>	<b>Chapter 1 Count</b>
	4102
!	28
"	190
'	125
,	423
-	21
.	189
;	4
?	16
—	6
a	1352
b	290
c	366
d	788
e	1981
f	339
g	446
h	1069
i	1071
j	51

... (16 rows omitted)

How do these counts compare with the character counts in Chapter 2? By plotting both sets of counts together, we can see slight variations for every character. Is that difference meaningful? How sure can we be? Such questions will be addressed precisely in this text.

```
counts = Counter(little_women_chapters[1].lower())
two_letters = Table([counts.keys(), counts.values()], ['Letters', 'Counts'])
compare = letters.join('Letters', two_letters)
compare.bahr('Letters', overlay=True)
```



In some cases, the relationships between quantities allow us to make predictions. This text will explore how to make accurate predictions based on incomplete information and develop methods for combining multiple sources of uncertain information to make decisions.

As an example, let us first use the computer to get us some information that would be really tedious to acquire by hand: the number of characters and the number of periods in each chapter of both books.

```
chlen_per_hf = Table([[len(s) for s in huck_finn_chapters],
                     np.char.count(huck_finn_chapters, '.')],
                     ["HF Chapter Length", "Number of periods"])
chlen_per_lw = Table([[len(s) for s in little_women_chapters],
                     np.char.count(little_women_chapters, '.')],
                     ["LW Chapter Length", "Number of periods"])
```

Here are the data for *Huckleberry Finn*. Each row of the table below corresponds to one chapter of the novel and displays the number of characters as well as the number of periods in the chapter. Not surprisingly, chapters with fewer characters also tend to have fewer periods, in general; the shorter the chapter, the fewer sentences there tend to be, and vice versa. The relation is not entirely predictable, however, as sentences are of varying lengths and can involve other punctuation such as question marks.

chlen\_per\_hf

HF Chapter Length	Number of periods
7026	66
11982	117
8529	72
6799	84
8166	91
14550	125
13218	127
22208	249
8081	71
7036	70

... (33 rows omitted)

Here are the corresponding data for *Little Women*.

```
chlen_per_lw
```

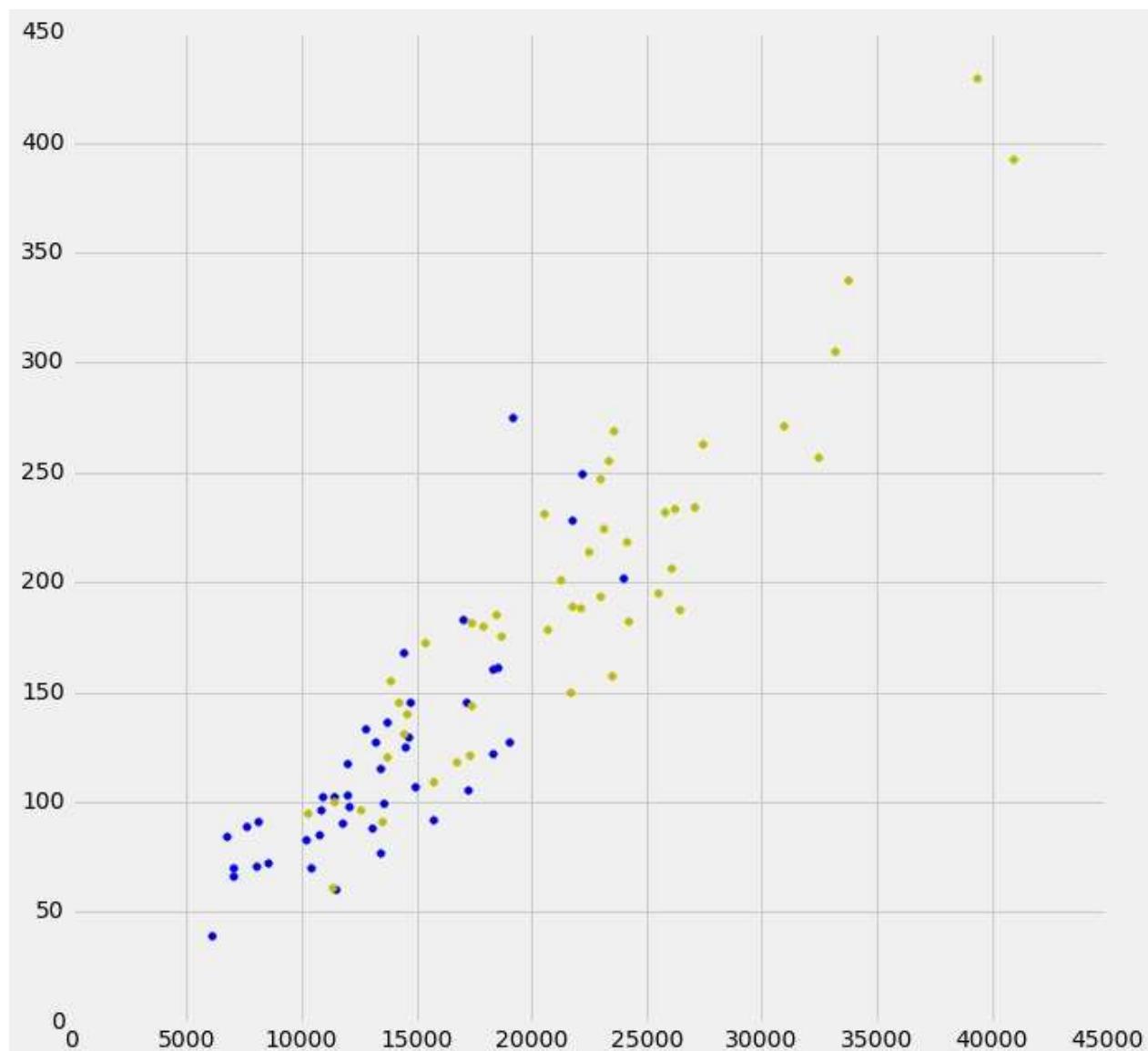
LW Chapter Length	Number of periods
21759	189
22148	188
20558	231
25526	195
23395	255
14622	140
14431	131
22476	214
33767	337
18508	185

... (37 rows omitted)

You can see that the chapters of *Little Women* are in general longer than those of *Huckleberry Finn*. Let us see if these two simple variables – the length and number of periods in each chapter – can tell us anything more about the two authors. One way for us to do this is to plot both sets of data on the same axes.

In the plot below, there is a dot for each chapter in each book. Blue dots correspond to *Huckleberry Finn* and yellow dots to *Little Women*. The horizontal axis represents the number of characters and the vertical axis represents the number of periods.

```
plots.figure(figsize=(10,10))
plots.scatter([len(s) for s in huck_finn_chapters], np.char.count(h
plots.scatter([len(s) for s in little_women_chapters], np.char.count(l
[<1>] [>1>]
<matplotlib.collections.PathCollection at 0x109ebee10>
```



The plot shows us that many but not all of the chapters of *Little Women* are longer than those of *Huckleberry Finn*, as we had observed by just looking at the numbers. But it also shows us something more. Notice how the blue points are roughly clustered around a straight line, as are the yellow points. Moreover, it looks as though both colors of points might be clustered around the *same* straight line.

Indeed, it appears from looking at the plot that on average both books tend to have somewhere between 100 and 150 characters between periods. Perhaps these two great 19th century novels were signaling something so very familiar us now: the 140-character limit of Twitter.

# Causality and Experiments

*"These problems are, and will probably ever remain, among the inscrutable secrets of nature. They belong to a class of questions radically inaccessible to the human intelligence."*  
—The Times of London, September 1849, on how cholera is contracted and spread

Does the death penalty have a deterrent effect? Is chocolate good for you? What causes breast cancer?

All of these questions attempt to assign a cause to an effect. A careful examination of data can help shed light on questions like these. In this section you will learn some of the fundamental concepts involved in establishing causality.

Observation is a key to good science. An *observational study* is one in which scientists make conclusions based on data that they have observed but had no hand in generating. In data science, many such studies involve observations on a group of individuals, a factor of interest called a *treatment*, and an *outcome* measured on each individual.

It is easiest to think of the individuals as people. In a study of whether chocolate is good for the health, the individuals would indeed be people, the treatment would be eating chocolate, and the outcome might be a measure of blood pressure. But individuals in observational studies need not be people. In a study of whether the death penalty has a deterrent effect, the individuals could be the 50 states of the union. A state law allowing the death penalty would be the treatment, and an outcome could be the state's murder rate.

The fundamental question is whether the treatment has an effect on the outcome. Any relation between the treatment and the outcome is called an *association*. If the treatment causes the outcome to occur, then the association is *causal*. *Causality* is at the heart of all three questions posed at the start of this section. For example, one of the questions was whether chocolate directly causes improvements in health, not just whether there is a relation between chocolate and health.

The establishment of causality often takes place in two stages. First, an association is observed. Next, a more careful analysis leads to a decision about causality.

# Observation and Visualization: John Snow and the Broad Street Pump

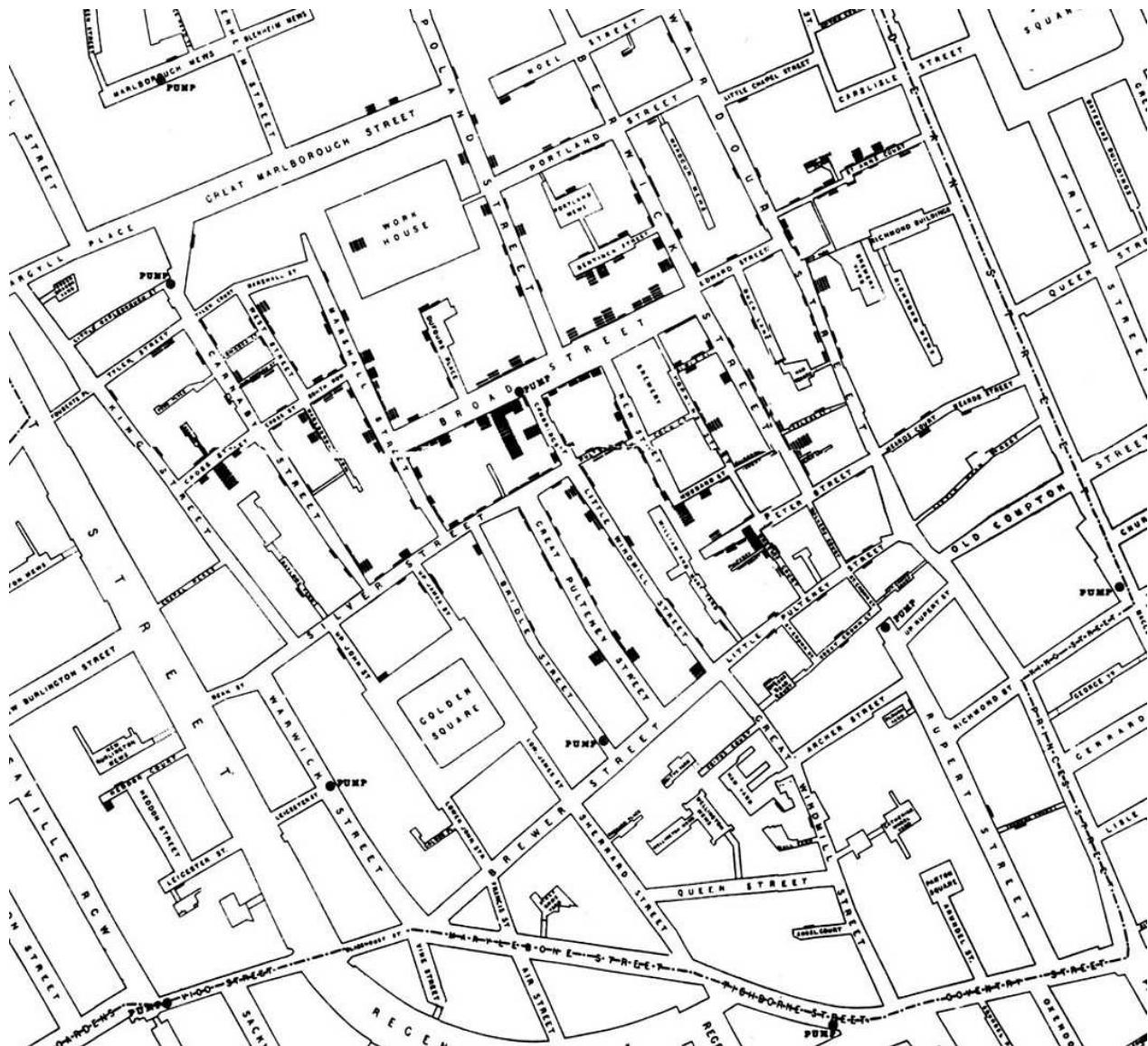
One of the earliest examples of astute observation eventually leading to the establishment of causality dates back more than 150 years. To get your mind into the right timeframe, try to imagine London in the 1850's. It was the world's wealthiest city but many of its people were desperately poor. Charles Dickens, then at the height of his fame, was writing about their plight. Disease was rife in the poorer parts of the city, and cholera was among the most feared. It was not yet known that germs cause disease; the leading theory was that "miasmas" were the main culprit. Miasmas manifested themselves as bad smells, and were thought to be invisible poisonous particles arising out of decaying matter. Parts of London did smell very bad, especially in hot weather. To protect themselves against infection, those who could afford to held sweet-smelling things to their noses.

For several years, a doctor by the name of John Snow had been following the devastating waves of cholera that hit England from time to time. The disease arrived suddenly and was almost immediately deadly: people died within a day or two of contracting it, hundreds could die in a week, and the total death toll in a single wave could reach tens of thousands. Snow was skeptical of the miasma theory. He had noticed that while entire households were wiped out by cholera, the people in neighboring houses sometimes remained completely unaffected. As they were breathing the same air—and miasmas—as their neighbors, there was no compelling association between bad smells and the incidence of cholera.

Snow had also noticed that the onset of the disease almost always involved vomiting and diarrhea. He therefore believed that that infection was carried by something people ate or drank, not by the air that they breathed. His prime suspect was water contaminated by sewage.

At the end of August 1854, cholera struck in the overcrowded Soho district of London. As the deaths mounted, Snow recorded them diligently, using a method that went on to become standard in the study of how diseases spread: *he drew a map*. On a street map of the district, he recorded the location of each death.

Here is Snow's original map. Each black bar represents one death. The black discs mark the locations of water pumps. The map displays a striking revelation—the deaths are roughly clustered around the Broad Street pump.



Snow studied his map carefully and investigated the apparent anomalies. All of them implicated the Broad Street pump. For example:

- There were deaths in houses that were nearer the Rupert Street pump than the Broad Street pump. Though the Rupert Street pump was closer as the crow flies, it was less convenient to get to because of dead ends and the layout of the streets. The residents in those houses used the Broad Street pump instead.
- There were no deaths in two blocks just east of the pump. That was the location of the Lion Brewery, where the workers drank what they brewed. If they wanted water, the brewery had its own well.
- There were scattered deaths in houses several blocks away from the Broad Street pump. Those were children who drank from the Broad Street pump on their way to school. The pump's water was known to be cool and refreshing.

The final piece of evidence in support of Snow's theory was provided by two isolated deaths in the leafy and genteel Hampstead area, quite far from Soho. Snow was puzzled by these until he learned that the deceased were Mrs. Susannah Eley, who had once lived in Broad

Street, and her niece. Mrs. Eley had water from the Broad Street pump delivered to her in Hampstead every day. She liked its taste.

Later it was discovered that a cesspit that was just a few feet away from the well of the Broad Street pump had been leaking into the well. Thus the pump's water was contaminated by sewage from the houses of cholera victims.

Snow used his map to convince local authorities to remove the handle of the Broad Street pump. Though the cholera epidemic was already on the wane when he did so, it is possible that the disabling of the pump prevented many deaths from future waves of the disease.

The removal of the Broad Street pump handle has become the stuff of legend. At the Centers for Disease Control (CDC) in Atlanta, when scientists look for simple answers to questions about epidemics, they sometimes ask each other, "Where is the handle to this pump?"

Snow's map is one of the earliest and most powerful uses of data visualization. Disease maps of various kinds are now a standard tool for tracking epidemics.

### Towards Causality

Though the map gave Snow a strong indication that the cleanliness of the water supply was the key to controlling cholera, he was still a long way from a convincing scientific argument that contaminated water was causing the spread of the disease. To make a more compelling case, he had to use the method of *comparison*.

Scientists use comparison to identify an association between a treatment and an outcome. They compare the outcomes of a group of individuals who got the treatment (the *treatment group*) to the outcomes of a group who did not (the *control group*). For example, researchers today might compare the average murder rate in states that have the death penalty with the average murder rate in states that don't.

If the results are different, that is evidence for an association. To determine causation, however, even more care is needed.

## Snow's “Grand Experiment”

Encouraged by what he had learned in Soho, Snow completed a more thorough analysis of cholera deaths. For some time, he had been gathering data on cholera deaths in an area of London that was served by two water companies. The Lambeth water company drew its water upriver from where sewage was discharged into the River Thames. Its water was relatively clean. But the Southwark and Vauxhall (S&V) company drew its water below the sewage discharge, and thus its supply was contaminated.

Snow noticed that there was no systematic difference between the people who were supplied by S&V and those supplied by Lambeth. “Each company supplies both rich and poor, both large houses and small; there is no difference either in the condition or occupation of the persons receiving the water of the different Companies … there is no difference whatever in the houses or the people receiving the supply of the two Water Companies, or in any of the physical conditions with which they are surrounded …”

The only difference was in the water supply, “one group being supplied with water containing the sewage of London, and amongst it, whatever might have come from the cholera patients, the other group having water quite free from impurity.”

Confident that he would be able to arrive at a clear conclusion, Snow summarized his data in the table below.

Supply Area	Number of houses	cholera deaths	deaths per 10,000 houses
S&V	40,046	1,263	315
Lambeth	26,107	98	37
Rest of London	256,423	1,422	59

The numbers pointed accusingly at S&V. The death rate from cholera in the S&V houses was almost ten times the rate in the houses supplied by Lambeth.

# Establishing Causality

In the language developed earlier in the section, you can think of the people in the S&V houses as the treatment group, and those in the Lambeth houses at the control group. A crucial element in Snow's analysis was that the people in the two groups were comparable to each other, apart from the treatment.

In order to establish whether it was the water supply that was causing cholera, Snow had to compare two groups that were similar to each other in all but one aspect—their water supply. Only then would he be able to ascribe the differences in their outcomes to the water supply. If the two groups had been different in some other way as well, it would have been difficult to point the finger at the water supply as the source of the disease. For example, if the treatment group consisted of factory workers and the control group did not, then differences between the outcomes in the two groups could have been due to the water supply, or to factory work, or both, or to any other characteristic that made the groups different from each other. The final picture would have been much more fuzzy.

Snow's brilliance lay in identifying two groups that would make his comparison clear. He had set out to establish a causal relation between contaminated water and cholera infection, and to a great extent he succeeded, even though the miasmatists ignored and even ridiculed him. Of course, Snow did not understand the detailed mechanism by which humans contract cholera. That discovery was made in 1883, when the German scientist Robert Koch isolated the *Vibrio cholerae*, the bacterium that enters the human small intestine and causes cholera.

In fact the *Vibrio cholerae* had been identified in 1854 by Filippo Pacini in Italy, just about when Snow was analyzing his data in London. Because of the dominance of the miasmatists in Italy, Pacini's discovery languished unknown. But by the end of the 1800's, the miasma brigade was in retreat. Subsequent history has vindicated Pacini and John Snow. Snow's methods led to the development of the field of *epidemiology*, which is the study of the spread of diseases.

## Confounding

Let us now return to more modern times, armed with an important lesson that we have learned along the way:

In an observational study, if the treatment and control groups differ in ways other than the treatment, it is difficult to make conclusions about causality.

An underlying difference between the two groups (other than the treatment) is called a *confounding factor*, because it might confound you (that is, mess you up) when you try to reach a conclusion.

**Example: Coffee and lung cancer.** Studies in the 1960's showed that coffee drinkers had higher rates of lung cancer than those who did not drink coffee. Because of this, some people identified coffee as a cause of lung cancer. But coffee does not cause lung cancer. The analysis contained a confounding factor – smoking. In those days, coffee drinkers were also likely to have been smokers, and smoking does cause lung cancer. Coffee drinking was associated with lung cancer, but it did not cause the disease.

Confounding factors are common in observational studies. Good studies take great care to reduce confounding.

# Randomization

An excellent way to avoid confounding is to assign individuals to the treatment and control groups *at random*, and then administer the treatment to those who were assigned to the treatment group. Randomization keeps the two groups similar apart from the treatment.

If you are able to randomize individuals into the treatment and control groups, you are running a *randomized controlled experiment*, also known as a *randomized controlled trial* (RCT). Sometimes, people's responses in an experiment are influenced by their knowing which group they are in. So you might want to run a *blind* experiment in which individuals do not know whether they are in the treatment group or the control group. To make this work, you will have to give the control group a *placebo*, which is something that looks exactly like the treatment but in fact has no effect.

Randomized controlled experiments have long been a gold standard in the medical field, for example in establishing whether a new drug works. They are also becoming more commonly used in other fields such as economics.

**Example: Welfare subsidies in Mexico.** In Mexican villages in the 1990's, children in poor families were often not enrolled in school. One of the reasons was that the older children could go to work and thus help support the family. Santiago Levy , a minister in Mexican Ministry of Finance, set out to investigate whether welfare programs could be used to increase school enrollment and improve health conditions. He conducted an RCT on a set of villages, selecting some of them at random to receive a new welfare program called PROGRESA. The program gave money to poor families if their children went to school regularly and the family used preventive health care. More money was given if the children were in secondary school than in primary school, to compensate for the children's lost wages, and more money was given for girls attending school than for boys. The remaining villages did not get this treatment, and formed the control group. Because of the randomization, there were no confounding factors and it was possible to establish that PROGRESA increased school enrollment. For boys, the enrollment increased from 73% in the control group to 77% in the PROGRESA group. For girls, the increase was even greater, from 67% in the control group to almost 75% in the PROGRESA group. Due to the success of this experiment, the Mexican government supported the program under the new name OPORTUNIDADES, as an investment in a healthy and well educated population.

In some situations it might not be possible to carry out a randomized controlled experiment, even when the aim is to investigate causality. For example, suppose you want to study the effects of alcohol consumption during pregnancy, and you randomly assign some pregnant

women to your “alcohol” group. You should not expect cooperation from them if you present them with a drink. In such situations you will almost invariably be conducting an observational study, not an experiment. Be alert for confounding factors.

# Endnote

In the terminology of that we have developed, John Snow conducted an observational study, not a randomized experiment. But he called his study a “grand experiment” because, as he wrote, “No fewer than three hundred thousand people … were divided into two groups without their choice, and in most cases, without their knowledge …”

Studies such as Snow’s are sometimes called “natural experiments.” However, true randomization does not simply mean that the treatment and control groups are selected “without their choice.”

The method of randomization can be as simple as tossing a coin. It may also be quite a bit more complex. But every method of randomization consists of a sequence of carefully defined steps that allow chances to be specified mathematically. This has two important consequences.

1. It allows us to account—mathematically—for the possibility that randomization produces treatment and control groups that are quite different from each other.
2. It allows us to make precise mathematical statements about differences between the treatment and control groups. This in turn helps us make justifiable conclusions about whether the treatment has any effect.

In this course, you will learn how to conduct and analyze your own randomized experiments. That will involve more detail than has been presented in this section. For now, just focus on the main idea: to try to establish causality, run a randomized controlled experiment if possible. If you are conducting an observational study, you might be able to establish association but not causation. Be extremely careful about confounding factors before making conclusions about causality based on an observational study.

## Terminology

- observational study
- treatment
- outcome
- association
- causal association
- causality
- comparison
- treatment group
- control group
- epidemiology

- confounding
- randomization
- randomized controlled experiment
- randomized controlled trial (RCT)
- blind
- placebo

## Fun facts

1. John Snow is sometimes called the father of epidemiology, but he was an anesthesiologist by profession. One of his patients was Queen Victoria, who was an early recipient of anesthetics during childbirth.
2. Florence Nightingale, the originator of modern nursing practices and famous for her work in the Crimean War, was a die-hard miasmatist. She had no time for theories about contagion and germs, and was not one for mincing her words. “There is no end to the absurdities connected with this doctrine,” she said. “Suffice it to say that in the ordinary sense of the word, there is no proof such as would be admitted in any scientific enquiry that there is any such thing as contagion.”
3. A later RCT established that the conditions on which PROGRESA insisted – children going to school, preventive health care – were not necessary to achieve increased enrollment. Just the financial boost of the welfare payments was sufficient.

## Good reads

*The Strange Case of the Broad Street Pump: John Snow and the Mystery of Cholera* by Sandra Hempel, published by our own University of California Press, reads like a whodunit. It was one of the main sources for this section's account of John Snow and his work. A word of warning: some of the contents of the book are stomach-churning.

*Poor Economics*, the best seller by Abhijit V. Banerjee and Esther Duflo of MIT, is an accessible and lively account of ways to fight global poverty. It includes numerous examples of RCTs, including the PROGRESA example in this section.

# Expressions

## Interact

Programming can dramatically improve our ability to collect and analyze information about the world, which in turn can lead to discoveries through the kind of careful reasoning demonstrated in the previous section. In data science, the purpose of writing a program is to instruct a computer to carry out the steps of an analysis. Computers cannot study the world on their own. People must describe precisely what steps the computer should take in order to collect and analyze data, and those steps are expressed through programs.

To learn to program, we will begin by learning some of the grammar rules of a programming language. Programs are made up of *expressions*, which describe to the computer how to combine pieces of data. For example, a multiplication expression consists of a `*` symbol between two numerical expressions. Expressions, such as `3 * 4`, are *evaluated* by the computer. The value (the result of *evaluation*) of the last expression in each cell, `12` in this case, is displayed below the cell.

```
3 * 4
```

```
12
```

The rules of a programming language are rigid. In Python, the `*` symbol cannot appear twice in a row. The computer will not try to interpret an expression that differs from its prescribed expression structures. Instead, it will show a `SyntaxError` error. The *Syntax* of a language is its set of grammar rules, and a `SyntaxError` indicates that some rule has been violated.

```
3 * * 4
```

```
File "<ipython-input-4-d90564f70db7>", line 1
 3 * * 4
          ^
SyntaxError: invalid syntax
```

Small changes to an expression can change its meaning entirely. Below, the space between the `*`'s has been removed. Because `**` appears between two numerical expressions, the expression is a well-formed *exponentiation* expression (the first number raised to the power of the second: 3 times 3 times 3 times 3). The symbols `*` and `**` are called *operators*, and the values they combine are called *operands*.

```
3 ** 4
```

```
81
```

**Common Operators.** Data science often involves combining numerical values, and the set of operators in a programming language are designed to so that expressions can be used to express any sort of arithmetic. In Python, the following operators are essential.

Expression Type	Operator	Example	Value
Addition	<code>+</code>	<code>2 + 3</code>	5
Subtraction	<code>-</code>	<code>2 - 3</code>	-1
Multiplication	<code>*</code>	<code>2 * 3</code>	6
Division	<code>/</code>	<code>7 / 3</code>	2.666667
Remainder	<code>%</code>	<code>7 % 3</code>	1
Exponentiation	<code>**</code>	<code>2 ** 0.5</code>	1.41421

Python expressions obey the same familiar rules of *precedence* as in algebra: multiplication and division occur before addition and subtraction. Parentheses can be used to group together smaller expressions within a larger expression.

```
1 + 2 * 3 * 4 * 5 / 6 ** 3 + 7 + 8
```

```
16.555555555555557
```

```
1 + 2 * ((3 * 4 * 5 / 6) ** 3) + 7 + 8
```

```
2016.0
```

This chapter introduces many types of expressions. Learning to program involves trying out everything you learn in combination, investigating the behavior of the computer. What happens if you divide by zero? What happens if you divide twice in a row? You don't always need to ask an expert (or the Internet); many of these details can be discovered by trying them out yourself.

# Names

## Interact

Names are given to values in Python using an *assignment* expression. In an assignment, a name is followed by `=`, which is followed by another expression. The value of the expression to the right of `=` is *assigned* to the name. Once a name has a value assigned to it, the value will be substituted for that name in future expressions.

```
a = 10  
b = 20  
a + b
```

30

A previously assigned name can be used in the expression to the right of `=`.

```
quarter = 2  
whole = 4 * quarter  
whole
```

8

However, only the current value of an expression is assigned to a name. If that value changes later, names that were defined in terms of that value will not change automatically.

```
quarter = 4  
whole
```

8

Names must start with a letter, but can contain both letters and numbers. A name cannot contain a space; instead, it is common to use an underscore character `_` to replace each space. Names are only as useful as you make them; it's up to the programmer to choose

names that are easy to interpret. Typically, more meaningful names can be invented than `a` and `b`. For example, to describe the sales tax on a \$5 purchase in Berkeley, CA, the following names clarify the meaning of the various quantities involved.

```
purchase_price = 5
state_tax_rate = 0.075
county_tax_rate = 0.02
city_tax_rate = 0
sales_tax_rate = state_tax_rate + county_tax_rate + city_tax_rate
sales_tax = purchase_price * sales_tax_rate
sales_tax
```

```
0.475
```

# Example: Growth Rates

## Interact

The relationship between two measurements of the same quantity taken at different times is often expressed as a *growth rate*. For example, the United States federal government employed 2,766,000 people in 2002 and 2,814,000 people in 2012

\footnote{\url{http://www.bls.gov/opub/mlr/2013/article/industry-employment-and-output-projections-to-2022-1.htm}}. To compute a growth rate, we must first decide which value to treat as the `initial` amount. For values over time, the earlier value is a natural choice. Then, we divide the difference between the `changed` and `initial` amount by the `initial` amount.

```
initial = 2766000
changed = 2814000
(changed - initial) / initial
```

```
0.01735357917570499
```

It is more typical to subtract one from the ratio of the two measurements, which yields the same value.

```
(changed/initial) - 1
```

```
0.017353579175704903
```

This value is the growth rate over 10 years. A useful property of growth rates is that they don't change even if the values are expressed in different units. So, for example, we can express the same relationship between thousands of people in 2002 and 2012.

```
initial = 2766
changed = 2814
(changed/initial) - 1
```

```
0.017353579175704903
```

In 10 years, the number of employees of the US Federal Government has increased by only 1.74%. In that time, the total expenditures of the US Federal Government increased from \$2.37 trillion to \$3.38 trillion in 2012.

```
initial = 2.37  
changed = 3.38  
(changed/initial) - 1
```

```
0.4261603375527425
```

A 42.6% increase in the federal budget is much larger than the 1.74% increase in federal employees. In fact, the number of federal employees has grown much more slowly than the population of the United States, which increased 9.21% in the same time period from 287.6 million people in 2002 to 314.1 million in 2012.

```
initial = 287.6  
changed = 314.1  
(changed/initial) - 1
```

```
0.09214186369958277
```

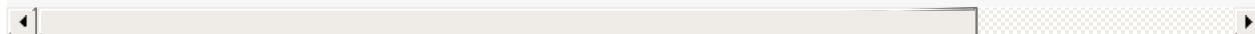
A growth rate can be negative, representing a decrease in some value. For example, the number of manufacturing jobs in the US decreased from 15.3 million in 2002 to 11.9 million in 2012, a -22.2% growth rate.

```
initial = 15.3  
changed = 11.9  
(changed/initial) - 1
```

```
-0.2222222222222222
```

An annual growth rate is a growth rate of some quantity over a single year. An annual growth rate of 0.035, accumulated each year for 10 years, gives a much larger ten-year growth rate of 0.41 (or 41%).

```
1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035
```



```
0.410598760621121
```

This same computation can be expressed using names and exponents for clarity.

```
annual_growth_rate = 0.035
ten_year_growth_rate = (1 + annual_growth_rate) ** 10 - 1
ten_year_growth_rate
```

```
0.410598760621121
```

Likewise, a ten-year growth rate can be used to compute an equivalent annual growth rate. Below, `t` is the number of years that have passed between measurements. The following computes the annual growth rate of federal expenditures over the last 10 years.

```
initial = 2.37
changed = 3.38
t = 10
(changed/initial) ** (1/t) - 1
```

```
0.03613617208346853
```

The total growth over 10 years is equivalent to a 3.6% increase each year.

# Call Expressions

## Interact

*Call expressions* invoke functions, which are named operations. The name of the function appears first, followed by expressions in parentheses.

```
abs(-12)
```

```
12
```

```
round(5 - 1.3)
```

```
4
```

```
max(2, 2 + 3, 4)
```

```
5
```

In this last example, the `max` function is *called* on three *arguments*: 2, 5, and 4. The value of each expression within parentheses is passed to the function, and the function *returns* the final value of the full call expression. The `max` function can take any number of arguments and returns the maximum.

A few functions are available by default, such as `abs` and `round`, but most functions that are built into the Python language are stored in a collection of functions called a *module*. An *import statement* is used to provide access to a module, such as `math` or `operator`.

```
import math  
import operator  
math.sqrt(operator.add(4, 5))
```

```
3.0
```

An equivalent expression could be expressed using the `+` and `**` operators instead.

```
(4 + 5) ** 0.5
```

```
3.0
```

Operators and call expressions can be used together in an expression. The *percent difference* between two values is used to compare values for which neither one is obviously `initial` or `changed`. For example, in 2014 Florida farms produced 2.72 billion eggs while Iowa farms produced 16.25 billion eggs footnote{<http://quickstats.nass.usda.gov/>}. The percent difference is 100 times the absolute value of the difference between the values, divided by their average. In this case, the difference is larger than the average, and so the percent difference is greater than 100.

```
florida = 2.72
iowa = 16.25
100*abs(florida-iowa)/((florida+iowa)/2)
```

```
142.6462836056932
```

Learning how different functions behave is an important part of learning a programming language. A Jupyter notebook can assist in remembering the names and effects of different functions. When editing a code cell, press the `tab` key after typing the beginning of a name to bring up a list of ways to complete that name. For example, press `tab` after `math.` to see all of the functions available in the `math` module. Typing will narrow down the list of options. To learn more about a function, place a `?` after its name. For example, typing `math.log?` will bring up a description of the `log` function in the `math` module.

```
math.log?
```

```
log(x[, base])
```

Return the logarithm of `x` to the given `base`.  
If the `base` not specified, returns the natural logarithm (base e) of `x`.

The square brackets in the example call indicate that an argument is optional. That is, `log` can be called with either one or two arguments.

```
math.log(16, 2)
```

```
4.0
```

```
math.log(16)/math.log(2)
```

```
4.0
```

# Data Types

## Interact

Python includes several types of values that will allow us to compute about more than just numbers.

Type Name	Python Class	Example 1	Example 2
Integer	int	1	-2
Floating-Point Number	float	1.5	-2.0
Boolean	bool	True	False
String	str	'hello world'	"False"

The first two types represent numbers. A "floating-point" number describes its [representation by a computer](#), which is a topic for another text. Practically, the difference between an integer and a floating-point number is that the latter can represent fractions in addition to whole values.

Boolean values, named for the logician [George Boole](#), represent truth and take only two possible values: `True` and `False`.

# Strings

Strings are the most flexible type of all. They can contain arbitrary text. A string can describe a number or a truth value, as well as any word or sentence. Much of the world's data is text, and text is represented in computers as strings.

The meaning of an expression depends both upon its structure and the types of values that are being combined. So, for instance, adding two strings together produces another string. This expression is still an addition expression, but it is combining a different type of value.

```
"data" + "science"
```

```
'datascience'
```

Addition is completely literal; it combines these two strings together without regard for their contents. It doesn't add a space because these are different words; that's up to the programmer (you) to specify.

```
"data" + " sc" + "ienc" + "e"
```

```
'data science'
```

Single and double quotes can both be used to create strings: `'hi'` and `"hi"` are identical expressions. Double quotes are often preferred because they allow you to include apostrophes inside of strings.

```
"This won't work with a single-quoted string!"
```

```
"This won't work with a single-quoted string!"
```

Why not? Try it out.

The `str` function returns a string representation of any value. Using this function, strings can be constructed that have embedded values.

```
"That's " + str(1 + 1) + ' ' + str(True)
```

```
"That's 2 True"
```

## String Methods

From an existing string, related strings can be constructed using string methods, which are functions that operate on strings. These methods are called by placing a dot after the string, then calling the function.

For example, the following method generates an uppercased version of a string.

```
"loud".upper()
```

```
'LOUD'
```

Perhaps the most important method is `replace`, which replaces all instances of a substring within the string. The `replace` method takes two arguments, the text to be replaced and its replacement.

```
'hitchhiker'.replace('hi', 'ma')
```

```
'matchmaker'
```

String methods can also be invoked using variable names, as long as those names are bound to strings. So, for instance, the following two-step process generates the word "degrade" starting from "train" by first creating "ingrain" and then applying a second replacement.

```
s = "train"  
t = s.replace('t', 'ing')  
u = t.replace('in', 'de')  
u
```

```
'degrade'
```

# Comparisons

## Interact

Boolean values most often arise from comparison operators. Python includes a variety of operators that compare values. For example, `3` is larger than `1 + 1`.

```
3 > 1 + 1
```

True

The value `True` indicates that the comparison is valid; Python has confirmed this simple fact about the relationship between `3` and `1+1`. The full set of common comparison operators are listed below.

Comparison	Operator	True example	False Example
Less than	<	<code>2 &lt; 3</code>	<code>2 &lt; 2</code>
Greater than	>	<code>3 &gt; 2</code>	<code>3 &gt; 3</code>
Less than or equal	<code>&lt;=</code>	<code>2 &lt;= 2</code>	<code>3 &lt;= 2</code>
Greater or equal	<code>&gt;=</code>	<code>3 &gt;= 3</code>	<code>2 &gt;= 3</code>
Equal	<code>==</code>	<code>3 == 3</code>	<code>3 == 2</code>
Not equal	<code>!=</code>	<code>3 != 2</code>	<code>2 != 2</code>

An expression can contain multiple comparisons, and they all must hold in order for the whole expression to be `True`. For example, we can express that `1+1` is between `1` and `3` using the following expression.

```
1 < 1 + 1 < 3
```

True

The average of two numbers is always between the smaller number and the larger number. We express this relationship for the numbers `x` and `y` below. You can try different values of `x` and `y` to confirm this relationship.

```
x = 12  
y = 5  
min(x, y) <= (x+y)/2 <= max(x, y)
```

True

Strings can also be compared, and their order is alphabetical. A shorter string is less than a longer string that begins with the shorter string.

```
"Dog" > "Catastrophe" > "Cat"
```

True

# Sequences

## Interact

Values can be grouped together into collections, which allows programmers to organize those values and refer to all of them with a single name. Separating expressions by commas within square brackets places the values of those expressions into a *list*, which is a built-in sequential collection. After a comma, it's okay to start a new line. Below, we collect four different temperatures into a list called `temps`. These are the estimated average daily high temperatures over all land on Earth (in degrees Celsius) for the decades surrounding 1850, 1900, 1950, and 2000, respectively, expressed as deviations from the average absolute high temperature between 1951 and 1980, which was 14.48 degrees.\footnote{\{http://berkeleyearth.lbl.gov/regions/global-land\}}

```
baseline_high = 14.48
highs = [baseline_high - 0.880, baseline_high - 0.093,
          baseline_high + 0.105, baseline_high + 0.684]
highs
```

```
[13.6, 14.387, 14.585, 15.164]
```

Lists allow us to pass multiple values into a function using a single name. For instance, the `sum` function computes the sum of all values in a list, and the `len` function computes the length of the list. Using them together, we can compute the average of the list.

```
sum(highs)/len(highs)
```

```
14.434000000000001
```

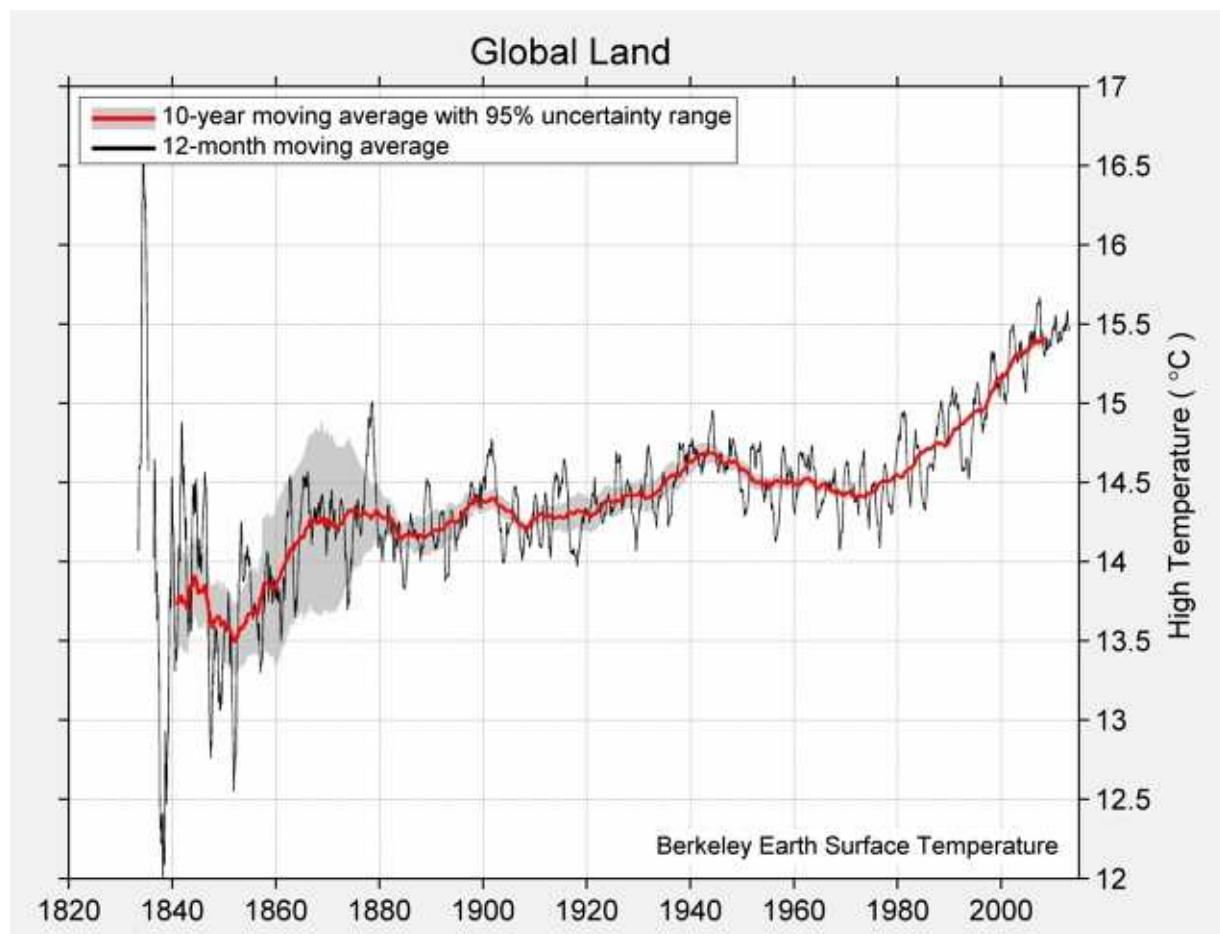
**Tuples.** Values can also be grouped together into *tuples* in addition to *lists*. A tuple is expressed by separating elements using commas within parentheses: `(1, 2, 3)`. In many cases, lists and tuples are interchangeable. We will use a tuple to hold the average daily low temperatures for the same four time ranges. The only change in the code is the switch to parentheses instead of brackets.

```
baseline_low = 3.00
lows = (baseline_low - 0.872, baseline_low - 0.629,
         baseline_low - 0.126, baseline_low + 0.728)
lows
```

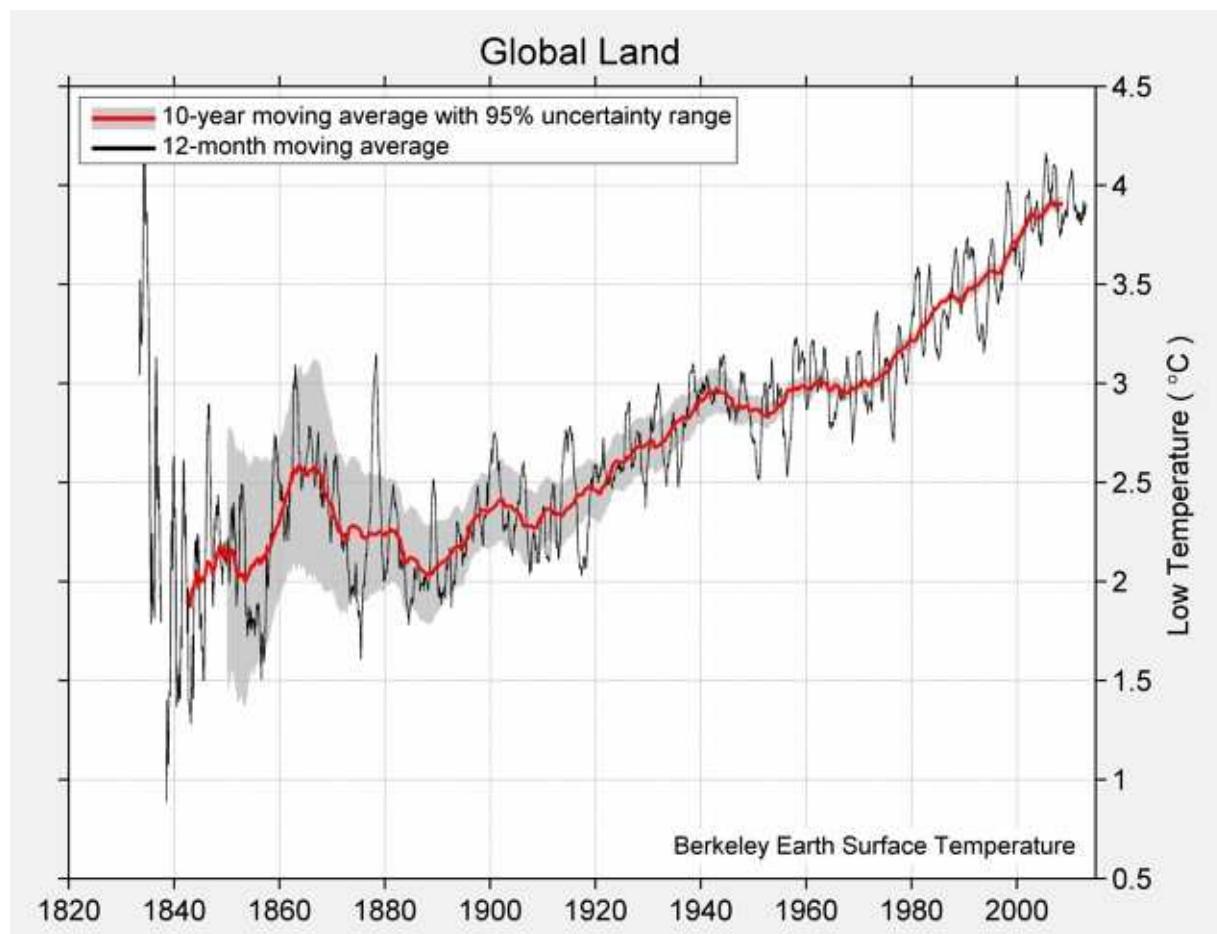
```
(2.128, 2.371, 2.874, 3.727999999999998)
```

The complete chart of daily high and low temperatures appears below.

## Mean of Daily High Temperature



## Mean of Daily Low Temperature



# Arrays

## Interact

Many experiments and data sets involve multiple values of the same type. An *array* is a collection of values that all have the same type. The `numpy` package, abbreviated `np` in programs, provides Python programmers with convenient and powerful functions for creating and manipulating arrays.

```
import numpy as np
```

An array is created using the `np.array` function, which takes a list or tuple as an argument. Here, we create arrays of average daily **high** and **low** temperatures for the decades surrounding 1850, 1900, 1950, and 2000.

```
baseline_high = 14.48
highs = np.array([baseline_high - 0.880,
                  baseline_high - 0.093,
                  baseline_high + 0.105,
                  baseline_high + 0.684])
highs
```

```
array([ 13.6   ,  14.387,  14.585,  15.164])
```

```
baseline_low = 3.00
lows = np.array((baseline_low - 0.872, baseline_low - 0.629,
                 baseline_low - 0.126, baseline_low + 0.728))
lows
```

```
array([ 2.128,  2.371,  2.874,  3.728])
```

Arrays differ from lists and tuples because they can be used in arithmetic expressions to compute over their contents. When an array is combined with a single number, that number is combined with each element of the array. Therefore, we can convert all of these temperatures to Farenheit by writing the familiar conversion formula.

```
(9/5) * highs + 32
```

```
array([ 56.48 , 57.8966, 58.253 , 59.2952])
```

When two arrays are combined together using an arithmetic operator, their individual values are combined. For example, we can compute the daily temperature ranges by subtracting.

```
highs - lows
```

```
array([ 11.472, 12.016, 11.711, 11.436])
```

Arrays also have *methods*, which are functions that operate on the array values. The `mean` of a collection of numbers is its average value: the sum divided by the length. Each pair of parentheses in the example below is part of a call expression; it's calling a function with no arguments to perform a computation on the array called `temps`.

```
[highs.size, highs.sum(), highs.mean()]
```

```
[4, 57.73600000000004, 14.434000000000001]
```

It's common that small rounding errors appear in arithmetic on a computer. For instance, the mean (average) above is in fact `14.434`, but an extra `0.0000000000000001` has been added during the computation of the average. Arithmetic rounding errors are an important topic in scientific computing, but won't be a focus of this text. Everyone who works with data must be aware that some operations will be approximated so that values can be stored and manipulated efficiently.

## Functions on Arrays

In addition to basic arithmetic and methods such as `min` and `max`, manipulating arrays often involves calling functions that are part of the `np` package. For example, the `diff` function computes the difference between each adjacent pair of elements in an array. The first element of the `diff` is the second element minus the first.

```
np.diff(highs)
```

```
array([ 0.787,  0.198,  0.579])
```

The [full Numpy reference](#) lists these functions exhaustively, but only a small subset are used commonly for data processing applications. These are grouped into different packages within `np`. Learning this vocabulary is an important part of learning the Python language, so refer back to this list often as you work through examples and problems.

Each of these functions takes an array as an argument and returns a single value.

Function	Description
<code>np.prod</code>	Multiply all elements together
<code>np.sum</code>	Add all elements together
<code>np.all</code>	Test whether all elements are true values (non-zero numbers are true)
<code>np.any</code>	Test whether any elements are true values (non-zero numbers are true)
<code>np.count_nonzero</code>	Count the number of non-zero elements

Each of these functions takes an array as an argument and returns an array of values.

Function	Description
<code>np.diff</code>	Difference between adjacent elements
<code>np.round</code>	Round each number to the nearest integer (whole number)
<code>np.cumprod</code>	A cumulative product: for each element, multiply all elements so far
<code>np.cumsum</code>	A cumulative sum: for each element, add all elements so far
<code>np.exp</code>	Exponentiate each element
<code>np.log</code>	Take the natural logarithm of each element
<code>np.sqrt</code>	Take the square root of each element
<code>np.sort</code>	Sort the elements

Each of these functions takes an array of strings and returns an array.

Function	Description
<code>np.char.lower</code>	Lowercase each element
<code>np.char.upper</code>	Uppercase each element
<code>np.char.strip</code>	Remove spaces at the beginning or end of each element
<code>np.char.isalpha</code>	Whether each element is only letters (no numbers or symbols)
<code>np.char.isnumeric</code>	Whether each element is only numeric (no letters)

Each of these functions takes both an array of strings and a *search string*; each returns an array.

Function	Description
<code>np.char.count</code>	Count the number of times a search string appears among the elements of an array
<code>np.char.find</code>	The position within each element that a search string is found first
<code>np.char.rfind</code>	The position within each element that a search string is found last
<code>np.char.startswith</code>	Whether each element starts with the search string

# Ranges

## Interact

A *range* is an array of numbers in increasing or decreasing order, each separated by a regular interval. Ranges are defined using the `np.arange` function, which takes either one, two, or three arguments.

```
np.arange(end): An array starting with 0 of increasing integers up to end
np.arange(start, end): An array of increasing integers from start up to end
np.arange(start, end, step): A range with step between each pair of consecutive values
```

A range always includes its `start` value, but does not include its `end` value. The `step` can be either positive or negative and may be a whole number or a fraction.

```
by_four = np.arange(1, 100, 4)
by_four
```

```
array([ 1,  5,  9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57,
       69, 73, 77, 81, 85, 89, 93, 97])
```

Ranges have many uses. For instance, a range can be used to compute part of the Leibniz formula for  $\pi$ , which is typically written as

$$\pi = 4 \cdot \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \dots \right)$$

```
4 * sum(1 / by_four - 1 / (by_four + 2))
```

```
3.1215946525910097
```

# The Birthday Problem

There are `k` students in a class. What is the chance that at least two of the students have the same birthday?

## Assumptions

1. No leap years; every year has 365 days
2. Births are distributed evenly throughout the year
3. No student's birthday is affected by any other (e.g. twins)

Let's start with an easy case: `k` is 4. We'll first find the chance that all four people have different birthdays.

```
all_different = (364/365)*(363/365)*(362/365)
```

The chance that there is at least one pair with the same birthday is equivalent to the chance that the birthdays are not all different. Given the chance of some event occurring, the chance that the event does not occur is one minus the chance that it does. With only 4 people, the chance that any two have the same birthday is less than 2%.

```
1 - all_different
```

```
0.016355912466550215
```

Using a range, we can express this same computation compactly. We begin with the numerators of each factor.

```
k = 4
numerators = np.arange(364, 365-k, -1)
numerators
```

```
array([364, 363, 362])
```

Then, we divide each numerator by 365 to form the factors, multiply them all together, and subtract from 1.

```
1 - np.prod(numerators/365)
```

```
0.016355912466550215
```

If `k` is 40, the chance of two birthdays being the same is much higher: almost 90%!

```
k = 40
numerators = np.arange(364, 365-k, -1)
1 - np.prod(numerators/365)
```

0.89123180981794903

Using ranges, we can investigate how this chance changes as `k` increases. The `np.cumprod` function computes the cumulative product of an array. That is, it computes a new array that has the same length as its input, but the `i`th element is the product of the first `i` terms. Below, the fourth term of the result is `1 * 2 * 3 * 4`.

```
ten = np.arange(1, 9)
np.cumprod(ten)
```

array([ 1, 2, 6, 24, 120, 720, 5040, 40320])

The following cell computes the chance of matching birthdays for every class size from 2 up to 365. Scrolling through the result, you will see that as `k` increases, the chance of matching birthdays reaches 1 long before the end of the array. In fact, for any `k` smaller than 365, there is a chance that all `k` students in a class can have different birthdays. The chance is so small, however, that the difference from 1 is rounded away by the computer.

```
numerators = np.arange(364, 0, -1)
chances = 1 - np.cumprod(numerators/365)
chances
```

array([ 0.00273973,	0.00820417,	0.01635591,	0.02713557,	0.04046
0.0562357 ,	0.07433529,	0.09462383,	0.11694818,	0.14114
0.16702479,	0.19441028,	0.22310251,	0.25290132,	0.28366
0.31500767,	0.34691142,	0.37911853,	0.41143838,	0.44368
0.47569531,	0.50729723,	0.53834426,	0.5686997 ,	0.59824
0.62685928,	0.65446147,	0.68096854,	0.70631624,	0.73045
0.75334753,	0.77497185,	0.79531686,	0.81438324,	0.83218
0.84873401,	0.86406782,	0.87821966,	0.89123181,	0.90315
0.91403047,	0.92392286,	0.93288537,	0.9409759 ,	0.94825
0.9547744 ,	0.96059797,	0.96577961,	0.97037358,	0.97443



The `item` method of an array allows us to select a particular element from the array by its position. The starting position of an array is `item(0)`, so finding the chance of matching birthdays in a 40-person class involves extracting `item(40-2)` (because the starting item is for a 2-person class).

chances.item(40-2)

0.891231809817949

# Tables

## Interact

Tables are a fundamental object type for representing data sets. A table can be viewed in two ways. Tables are a sequence of named columns that each describe a single aspect of all entries in a data set. Tables are also a sequence of rows that each contain all information about a single entry in a data set.

In order to use tables, import all of the module called `datascience`, a module created for this text.

```
from datascience import *
```

Empty tables can be created using the `Table` function, which optionally takes a list of column labels. The `with_row` and `with_rows` methods of a table return new tables with one or more additional rows. For example, we could create a table of `Odd` and `Even` numbers.

```
Table(['Odd', 'Even']).with_row([3, 4])
```

Odd	Even
3	4

```
Table(['Odd', 'Even']).with_rows([[3, 4], [5, 6], [7, 8]])
```

Odd	Even
3	4
5	6
7	8

Tables can also be extended with additional columns. The `with_column` and `with_columns` methods return new tables with additional labeled columns. Below, we begin each example with an empty table that has no columns.

```
Table().with_column('Odd', [3, 5, 7])
```

Odd
3
5
7

```
Table().with_columns([
    'Odd', [3, 5, 7],
    'Even', [4, 6, 8]
])
```

Odd	Even
3	4
5	6
7	8

Tables are more typically created from files that contain comma-separated values, called CSV files. The file below contains "Annual Estimates of the Resident Population by Single Year of Age and Sex for the United States."

```
census_url = 'http://www.census.gov/popest/data/national/asrh/2014/
full_census_table = Table.read_table(census_url)
full_census_table
```

SEX	AGE	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010
0	0	3944153	3944160	3951330
0	1	3978070	3978090	3957888
0	2	4096929	4096939	4090862
0	3	4119040	4119051	4111920
0	4	4063170	4063186	4077552
0	5	4056858	4056872	4064653
0	6	4066381	4066412	4073013
0	7	4030579	4030594	4043047
0	8	4046486	4046497	4025604
0	9	4148353	4148369	4125415

... (296 rows omitted)

A [description of the table](#) appears online. The `SEX` column contains numeric codes: `0` stands for the total, `1` for male, and `2` for female. The `AGE` column contains ages, but the special value `999` is a sum of the total population. The rest of the columns contain estimates of the US population.

Typically, a public table will contain more information than necessary for a particular investigation or analysis. In this case, let us suppose that we are only interested in the population changes from 2010 to 2014. We can select only a subset of the columns using the `select` method.

```
partial_census_table = full_census_table.select(['SEX', 'AGE', 'POPESTIMATE2010', 'POPESTIMATE2014'])
partial_census_table
```

SEX	AGE	POPESTIMATE2010	POPESTIMATE2014
0	0	3951330	3948350
0	1	3957888	3962123
0	2	4090862	3957772
0	3	4111920	4005190
0	4	4077552	4003448
0	5	4064653	4004858
0	6	4073013	4134352
0	7	4043047	4154000
0	8	4025604	4119524
0	9	4125415	4106832

... (296 rows omitted)

The `relabeled` method creates an alternative version of the table that replaces a column label. Using this method, we can simplify the labels of the selected columns.

```
simple = partial_census_table.relabeled('POPESTIMATE2010', '2010')
simple
```

SEX	AGE	2010	2014
0	0	3951330	3948350
0	1	3957888	3962123
0	2	4090862	3957772
0	3	4111920	4005190
0	4	4077552	4003448
0	5	4064653	4004858
0	6	4073013	4134352
0	7	4043047	4154000
0	8	4025604	4119524
0	9	4125415	4106832

... (296 rows omitted)

The `column` method returns an array of the values in a particular column. Each column of a table is an array of the same length, and so columns can be combined using arithmetic.

```
simple.column('2014') - simple.column('2010')
```

```
array([-2980, 4235, -133090, ..., 6717, 13410, 4662996])
```

A table with an additional column can be created from an existing table using the `with_column` method, which takes a label and a sequence of values as arguments. There must be either as many values as there are existing rows in the table or a single value (which fills the column with that value).

```
change = simple.column('2014') - simple.column('2010')
annual_growth_rate = (simple.column('2014') / simple.column('2010'))
census = simple.with_columns(['Change', change, 'Growth', annual_gr
census
```

SEX	AGE	2010	2014	Change	Growth
0	0	3951330	3948350	-2980	-0.000188597
0	1	3957888	3962123	4235	0.000267397
0	2	4090862	3957772	-133090	-0.00823453
0	3	4111920	4005190	-106730	-0.0065532
0	4	4077552	4003448	-74104	-0.00457471
0	5	4064653	4004858	-59795	-0.00369821
0	6	4073013	4134352	61339	0.00374389
0	7	4043047	4154000	110953	0.00679123
0	8	4025604	4119524	93920	0.00578232
0	9	4125415	4106832	-18583	-0.00112804

... (296 rows omitted)

Although the columns of this table are simply arrays of numbers, the format of those numbers can be changed to improve the interpretability of the table. The `set_format` method takes `Formatter` objects, which exist for dates (`DateFormatter`), currencies (`CurrencyFormatter`), numbers, and percentages.

```
census.set_format('Growth', PercentFormatter)
census.set_format(['2010', '2014', 'Change'], NumberFormatter)
```

SEX	AGE	2010	2014	Change	Growth
0	0	3,951,330	3,948,350	-2,980	-0.02%
0	1	3,957,888	3,962,123	4,235	0.03%
0	2	4,090,862	3,957,772	-133,090	-0.82%
0	3	4,111,920	4,005,190	-106,730	-0.66%
0	4	4,077,552	4,003,448	-74,104	-0.46%
0	5	4,064,653	4,004,858	-59,795	-0.37%
0	6	4,073,013	4,134,352	61,339	0.37%
0	7	4,043,047	4,154,000	110,953	0.68%
0	8	4,025,604	4,119,524	93,920	0.58%
0	9	4,125,415	4,106,832	-18,583	-0.11%

... (296 rows omitted)

The information in a table can be accessed in many ways. The attributes demonstrated below can be used for any table.

```
census.labels
```

```
('SEX', 'AGE', '2010', '2014', 'Change', 'Growth')
```

```
census.num_columns
```

```
6
```

```
census.num_rows
```

```
306
```

```
census.row(5)
```

```
Row(SEX=0, AGE=5, 2010=4064653, 2014=4004858, Change=-59795, Growth=-1.478)
```

```
census.column(2)
```

```
array([ 3951330, 3957888, 4090862, ..., 26074, 45058, 157257573])
```

An individual item in a table can be accessed via a row or a column.

```
census.row(0).item(2)
```

```
3951330
```

```
census.column(2).item(0)
```

```
3951330
```

Let's take a look at the growth rates of the total number of males and females by selecting only the *rows* that sum over all ages. This sum is expressed with the special value 999 according to this data set description.

```
census.where('AGE', 999)
```

SEX	AGE	2010	2014	Change	Growth
0	999	309,347,057	318,857,056	9,509,999	0.76%
1	999	152,089,484	156,936,487	4,847,003	0.79%
2	999	157,257,573	161,920,569	4,662,996	0.73%

What ages of males are driving this rapid growth? We can first filter the `census` table to keep only the male entries, then sort by growth rate in decreasing order.

```
males = census.where('SEX', 1)
males.sort('Growth', descending=True)
```

SEX	AGE	2010	2014	Change	Growth
1	99	6,104	9,037	2,933	10.31%
1	100	9,351	13,729	4,378	10.08%
1	98	9,504	13,649	4,145	9.47%
1	93	60,182	85,980	25,798	9.33%
1	96	22,022	31,235	9,213	9.13%
1	94	43,828	62,130	18,302	9.12%
1	97	14,775	20,479	5,704	8.50%
1	95	31,736	42,824	11,088	7.78%
1	91	104,291	138,080	33,789	7.27%
1	92	83,462	109,873	26,411	7.12%

... (92 rows omitted)

The fact that there are more men with `AGE` of 100 than 99 looks suspicious; shouldn't there be fewer? A careful look at the description of the data set reveals that the 100 category actually includes all men who are 100 or older. The growth rates in men at these very old ages could have several explanations, such as a large influx from another country, but the most natural explanation is that people are simply living longer in 2014 than 2010.

The `where` method can also take an array of boolean values, constructed by applying some comparison operator to a column of the table. For example, we can find all of the age groups among males for which the absolute `change` is substantial. The `show` method displays all rows without abbreviating.

```
males.where(males.column('Change') > 2000000).show()
```

SEX	AGE	2010	2014	Change	Growth
1	23	2,151,095	2,399,883	248,788	2.77%
1	24	2,161,380	2,391,398	230,018	2.56%
1	34	1,908,761	2,192,455	283,694	3.52%
1	57	1,910,028	2,110,149	200,121	2.52%
1	59	1,779,504	2,006,900	227,396	3.05%
1	64	1,291,843	1,661,474	369,631	6.49%
1	65	1,272,693	1,607,688	334,995	6.02%
1	66	1,239,805	1,589,127	349,322	6.40%
1	67	1,270,148	1,653,257	383,109	6.81%
1	71	903,263	1,169,356	266,093	6.67%
1	999	152,089,484	156,936,487	4,847,003	0.79%

By far the largest changes are clumped together in the 64-67 range. In 2014, these people would be born between 1947 and 1951, the height of the post-WWII baby boom in the United States.

It is possible to specify multiple conditions using the functions `np.logical_and` and `np.logical_or`. When two conditions are combined with `logical_and`, both must be true for a row to be retained. When conditions are combined with `logical_or`, then either one of them must be true for a row to be retained. Here are two different ways to select 18 and 19 year olds.

```
both = census.where(census.column('SEX') != 0)
both.where(np.logical_or(both.column('AGE') == 18,
                        both.column('AGE') == 19))
```

SEX	AGE	2010	2014	Change	Growth
1	18	2,305,733	2,165,062	-140,671	-1.56%
1	19	2,334,906	2,220,790	-114,116	-1.24%
2	18	2,185,272	2,060,528	-124,744	-1.46%
2	19	2,236,479	2,105,604	-130,875	-1.50%

```
both.where(np.logical_and(both.column('AGE') >= 18,
                           both.column('AGE') <= 19))
```

SEX	AGE	2010	2014	Change	Growth
1	18	2,305,733	2,165,062	-140,671	-1.56%
1	19	2,334,906	2,220,790	-114,116	-1.24%
2	18	2,185,272	2,060,528	-124,744	-1.46%
2	19	2,236,479	2,105,604	-130,875	-1.50%

Here, we observe a rather dramatic decrease in the number of 18 and 19 year olds in the United States; the children of the baby boom generation are now in their 20's, leaving fewer teenagers in America.

**Aggregation and Grouping.** This particular dataset includes entries for sums across all ages and sexes, using the special values `999` and `0` respectively. However, if these rows did not exist, we would be able to aggregate the remaining entries in order to compute those same results.

```
both.where('AGE', 999).select(['SEX', '2014'])
```

SEX	2014
1	156,936,487
2	161,920,569

```
no_sums = both.select(['SEX', 'AGE', '2014']).where(both.column('AGE') > 0)
females = no_sums.where('SEX', 2).select(['AGE', '2014'])
females
```

AGE	2014
0	1,930,493
1	1,938,870
2	1,935,270
3	1,956,572
4	1,959,950
5	1,961,391
6	2,024,024
7	2,031,760
8	2,014,402
9	2,009,560

... (91 rows omitted)

```
sum(females['2014'])
```

```
161920569
```

Some columns express categories, such as the sex or age group in the case of the census table. Aggregation can also be performed on every value for a category using the `group` and `groups` methods. The `group` method takes a single column (or column name) as an argument and collects all values associated with each unique value in that column.

The second argument to `group`, the name of a function, specifies how to aggregate the resulting values. For example, the `sum` function can be used to add together the populations for each age. In this result, the `SEX sum` column is meaningless because the values were simply codes for different categories. However, the `2014 sum` column does in fact contain the total number across all sexes for each age category.

```
no_sums.select(['AGE', '2014']).group('AGE', sum)
```

AGE	2014 sum
0	3948350
1	3962123
2	3957772
3	4005190
4	4003448
5	4004858
6	4134352
7	4154000
8	4119524
9	4106832
... (91 rows omitted)	

**Joining Tables.** There are many situations in data analysis in which two different rows need to be considered together. Two tables can be joined into one, an operation that creates one long row out of two matching rows. These matching rows can be from the same table or different tables.

For example, we might want to estimate which age categories are expected to change significantly in the future. Someone who is 14 years old in 2014 will be 20 years old in 2020. Therefore, one estimate of the number of 20 year olds in 2020 is the number of 14 year olds in 2014. Between the ages of 1 and 50, annual mortality rates are very low (less than 0.5% for men and less than 0.3% for women [1]). Immigration also affects population changes, but for now we will ignore its influence. Let's consider just females in this analysis.

```
females['AGE+6'] = females['AGE'] + 6
females
```

AGE	2014	AGE+6
0	1,930,493	6
1	1,938,870	7
2	1,935,270	8
3	1,956,572	9
4	1,959,950	10
5	1,961,391	11
6	2,024,024	12
7	2,031,760	13
8	2,014,402	14
9	2,009,560	15
... (91 rows omitted)		

In order to relate the age in 2014 to the age in 2020, we will join this table with itself, matching values in the `AGE` column with values in the `AGE in 2020` column.

```
joined = females.join('AGE', females, 'AGE+6')
joined
```

AGE	2014	AGE+6	AGE_2	2014_2
6	2024024	12	0	1930493
7	2031760	13	1	1938870
8	2014402	14	2	1935270
9	2009560	15	3	1956572
10	2015380	16	4	1959950
11	2001949	17	5	1961391
12	1993547	18	6	2024024
13	2041159	19	7	2031760
14	2068252	20	8	2014402
15	2035299	21	9	2009560
... (85 rows omitted)				

The resulting table has five columns. The first three are the same as before. The two new columns are the values for `AGE` and `2014` that appeared in a different row, the one in which that `AGE` appeared in the `AGE+6` column. For instance, the first row contains the number of 6 year olds in 2014 and an estimate of the number of 6 year olds in 2020 (who were 0 in 2014). Some relabeling and selecting makes this table more clear.

```
future = joined.select(['AGE', '2014', '2014_2']).relabeled('2014_2', 'future')
```

AGE	2014	2020 (est)
6	2024024	1930493
7	2031760	1938870
8	2014402	1935270
9	2009560	1956572
10	2015380	1959950
11	2001949	1961391
12	1993547	2024024
13	2041159	2031760
14	2068252	2014402
15	2035299	2009560

... (85 rows omitted)

Adding a `Change` column and sorting by that change describes some of the major changes in age categories that we can expect in the United States for people under 50. According to this simplistic analysis, there will be substantially more people in their late 30's by 2020.

```
predictions = future.where(future['AGE'] < 50)
predictions['Change'] = predictions['2020 (est)'] - predictions['2014']
predictions.sort('Change', descending=True)
```

AGE	2014	2020 (est)	Change
40	1940627	2170440	229813
38	1936610	2154232	217622
30	2110672	2301237	190565
37	1995155	2148981	153826
39	1993913	2135416	141503
29	2169563	2298701	129138
35	2046713	2169563	122850
36	2009423	2110672	101249
28	2144666	2244480	99814
41	1977497	2046713	69216
... (34 rows omitted)			

# Functions

## Interact

We are building up a useful inventory of techniques for identifying patterns and themes in a data set. Sorting and grouping rows of a table can focus our attention. The next approach to analysis we will consider involves grouping rows of a table by arbitrary criteria. To do so, we will explore two core features of the Python programming language: function definition and conditional statements.

We have used functions extensively already in this text, but never defined a function of our own. The purpose of defining a function is to give a name to a computational process that may be applied multiple times. There are many situations in computing that require repeated computation. For example, it is often the case that we will want to perform the same manipulation on every value in a column.

A function is defined in Python using a `def` statement, which is a multi-line statement that begins with a *header* line giving the name of the function and names for the arguments of the function. The rest of the `def` statement, called the *body*, must be indented below the header.

A function expresses a relationship between its inputs (called *arguments*) and its outputs (called *return values*). The number of arguments required to call a function is the number of names that appear within parentheses in the `def` statement header. The values that are returned depend on the body. Whenever a function is called, its body is executed. Whenever a `return` statement within the body is executed, the call to the function completes and the value of the `return` expression is returned as the value of the function call.

The definition of the `percent` function below multiplies a number by 100 and rounds the result to two decimal places.

```
def percent(x):
    return round(100*x, 2)
```

The primary difference between defining a `percent` function and simply evaluating its return expression, `round(100*x, 2)`, is that when a function is defined, its return expression is *not* immediately evaluated. It cannot be, because the value for `x` is not yet defined. Instead, the return expression is evaluated whenever this `percent` function is *called* by placing parentheses after the name `percent` and placing an expression to compute its argument in parentheses.

```
percent(1/6)
```

```
16.67
```

```
percent(1/6000)
```

```
0.02
```

```
percent(1/60000)
```

```
0.0
```

The three expressions above are all *call expressions*. In the first, the value of `1/6` is computed and then passed as the argument named `x` to the `percent` function. When the `percent` function is called in this way, its body is executed. The body of `percent` has only a single line: `return round(100*x, 2)`. Executing this `return` statement completes execution of the `percent` function's body and computes the value of the call expression.

The same result is computed by passing a named value as an argument. The `percent` function does not know or care how its argument is computed or stored; its only job is to execute its own body using the arguments passed to it.

```
sixth = 1/6
percent(sixth)
```

```
16.67
```

**Conditional Statements.** The body of a function can have more than one line and more than one `return` statement. A *conditional statement* is a multi-line statement that allows Python to choose among different alternatives based on the truth value of an expression. While conditional statements can appear anywhere, they appear most often within the body of a function in order to express alternative behavior depending on argument values.

A conditional statement always begins with an `if` header, which is a single line followed by an indented body. The body is only executed if the expression directly following `if` (called the *if expression*) evaluates to a true value. If the *if expression* evaluates to a false value, then the body of the `if` is skipped.

For example, we can improve our `percent` function so that it doesn't round very small numbers to zero so readily. The behavior of `percent(1/6)` is unchanged, but `percent(1/60000)` provides a more useful result.

```
def percent(x):
    if x < 0.00005:
        return 100 * x
    return round(100 * x, 2)
```

```
percent(1/6)
```

```
16.67
```

```
percent(1/6000)
```

```
0.02
```

```
percent(1/60000)
```

```
0.0016666666666666668
```

A conditional statement can also have multiple clauses with multiple bodies, and only one of those bodies can ever be executed. The general format of a multi-clause conditional statement appears below.

```

if <if expression>:
    <if body>
elif <elif expression 0>:
    <elif body 0>
elif <elif expression 1>:
    <elif body 1>
...
else:
    <else body>

```

There is always exactly one `if` clause, but there can be any number of `elif` clauses. Python will evaluate the `if` and `elif` expressions in the headers in order until one is found that is a true value, then execute the corresponding body. The `else` clause is optional. When an `else` header is provided, its `else body` is executed only if none of the header expressions of the previous clauses are true. The `else` clause must always come at the end (or not at all).

Let us continue to refine our `percent` function. Perhaps for some analysis, any value below  $10^{-8}$  should be considered close enough to 0 that it can be ignored. The following function definition handles this case as well.

```

def percent(x):
    if x < 1e-8:
        return 0.0
    elif x < 0.00005:
        return 100 * x
    else:
        return round(100 * x, 2)

```

`percent(1/6)`

16.67

`percent(1/6000)`

0.02

```
percent(1/60000)
```

```
0.001666666666666668
```

```
percent(1/6000000000)
```

```
0.0
```

A well-composed function has a name that evokes its behavior, as well as a *docstring* — a description of its behavior and expectations about its arguments. The docstring can also show example calls to the function, where the call is preceded by `>>>`.

A docstring can be any string that immediately follows the header line of a `def` statement. Docstrings are typically defined using triple quotation marks at the start and end, which allows the string to span multiple lines. The first line is conventionally a complete but short description of the function, while following lines provide further guidance to future users of the function.

A more complete definition of `percent` that includes a docstring appears below.

```
def percent(x):
    """Convert x to a percentage by multiplying by 100.

    Percentages are conventionally rounded to two decimal places,
    but precision is retained for any x above 1e-8 that would
    otherwise round to 0.

    >>> percent(1/6)
    16.67
    >>> percent(1/6000)
    0.02
    >>> percent(1/60000)
    0.001666666666666668
    >>> percent(1/6000000000)
    0.0
    """
    if x < 1e-8:
        return 0.0
    elif x < 0.00005:
        return 100 * x
    else:
        return round(100 * x, 2)
```

# Functions and Tables

## Interact

In this section, we will continue to use the US Census dataset. We will focus only on the 2014 population estimate.

```
census_2014 = census.select(['SEX', 'AGE', '2014']).where(census.cc
census_2014
```

SEX	AGE	2014
0	0	3,948,350
0	1	3,962,123
0	2	3,957,772
0	3	4,005,190
0	4	4,003,448
0	5	4,004,858
0	6	4,134,352
0	7	4,154,000
0	8	4,119,524
0	9	4,106,832

... (293 rows omitted)

Functions can be used to compute new columns in a table based on existing column values. For example, we can transform the codes in the `SEX` column to strings that are easier to interpret.

```
def male_female(code):
    if code == 0:
        return 'Total'
    elif code == 1:
        return 'Male'
    elif code == 2:
        return 'Female'
```

This function takes an individual code — 0, 1, or 2 — and returns a string that describes its meaning.

```
male_female(0)
```

```
'Total'
```

```
male_female(1)
```

```
'Male'
```

```
male_female(2)
```

```
'Female'
```

We could also transform ages into age categories.

```
def age_group(age):
    if age < 2:
        return 'Baby'
    elif age < 13:
        return 'Child'
    elif age < 20:
        return 'Teen'
    else:
        return 'Adult'
```

```
age_group(15)
```

```
'Teen'
```

**Apply.** The `apply` method of a table calls a function on each element of a column, forming a new array of return values. To indicate which function to call, just name it (without quotation marks). The name of the column of input values must still appear within quotation

marks.

```
census_2014.apply(male_female, 'SEX')
```

```
array(['Total', 'Total', 'Total', ..., 'Female', 'Female', 'Female',
       dtype='<U6')
```

This array, which has the same length as the original `SEX` column of the `population` table, can be used as the values in a new column called `Male/Female` alongside the existing `AGE` and `Population` columns.

```
population = Table().with_columns([
    'Male/Female', census_2014.apply(male_female, 'SEX'),
    'Age Group', census_2014.apply(age_group, 'AGE'),
    'Population', census_2014.column('2014')
])
population
```

Male/Female	Age Group	Population
Total	Baby	3948350
Total	Baby	3962123
Total	Child	3957772
Total	Child	4005190
Total	Child	4003448
Total	Child	4004858
Total	Child	4134352
Total	Child	4154000
Total	Child	4119524
Total	Child	4106832

... (293 rows omitted)

**Groups.** The `group` method with a single argument counts the number of rows for each category in a column. The result contains one row per unique value in the grouped column.

```
population.group('Age Group')
```

Age Group	count
Adult	243
Baby	6
Child	33
Teen	21

The optional second argument names the function that will be used to aggregate values in other columns for all of those rows. For instance, `sum` will sum up the populations in all rows that match each category. This result also contains one row per unique value in the grouped column, but it has the same number of columns as the original table.

```
totals = population.where(0, 'Total').select(['Age Group', 'Population'])
totals
```

Age Group	Population
Baby	3948350
Baby	3962123
Child	3957772
Child	4005190
Child	4003448
Child	4004858
Child	4134352
Child	4154000
Child	4119524
Child	4106832

... (91 rows omitted)

```
totals.group('Age Group', sum)
```

Age Group	Population sum
Adult	236721454
Baby	7910473
Child	44755656
Teen	29469473

The `groups` method behaves in the same way, but accepts a list of columns as its first argument. The resulting table has one row for every *unique combination* of values that appear together in the grouped columns. Again, a single argument (a list, in this case) gives row counts.

```
population.groups(['Male/Female', 'Age Group'])
```

Male/Female	Age Group	count
Female	Adult	81
Female	Baby	2
Female	Child	11
Female	Teen	7
Male	Adult	81
Male	Baby	2
Male	Child	11
Male	Teen	7
Total	Adult	81
Total	Baby	2

... (2 rows omitted)

A second argument to `groups` aggregates all other columns that do not appear in the list of grouped columns.

```
population.groups(['Male/Female', 'Age Group'], sum)
```

Male/Female	Age Group	Population sum
Female	Adult	121754366
Female	Baby	3869363
Female	Child	21903805
Female	Teen	14393035
Male	Adult	114967088
Male	Baby	4041110
Male	Child	22851851
Male	Teen	15076438
Total	Adult	236721454
Total	Baby	7910473

... (2 rows omitted)

**Pivot.** The `pivot` method is closely related to the `groups` method: it groups together rows that share a combination of values. It differs from `groups` because it organizes the resulting values in a grid. The first argument to `pivot` is a column that contains the values that will be used to form new columns in the result. The second argument is a column used for grouping rows. The result gives the count of all rows that share the combination of column and row values.

```
population.pivot('Male/Female', 'Age Group')
```

Age Group	Female	Male	Total
Adult	81	81	81
Baby	2	2	2
Child	11	11	11
Teen	7	7	7

An optional third argument indicates a column of values that will replace the counts in each cell of the grid. The fourth argument indicates how to aggregate all of the values that match the combination of column and row values.

```
pivoted = population.pivot('Male/Female', 'Age Group', 'Population')
pivoted
```

Age Group	Female	Male	Total
Adult	121754366	114967088	236721454
Baby	3869363	4041110	7910473
Child	21903805	22851851	44755656
Teen	14393035	15076438	29469473

The advantage of pivot is that it places grouped values into adjacent columns, so that they can be combined. For instance, this pivoted table allows us to compute the proportion of each age group that is male. We find the surprising result that younger age groups are predominantly male, but among adults there are substantially more females.

```
pivoted.with_column('Male Percentage', pivoted.column('Male')/pivot
```

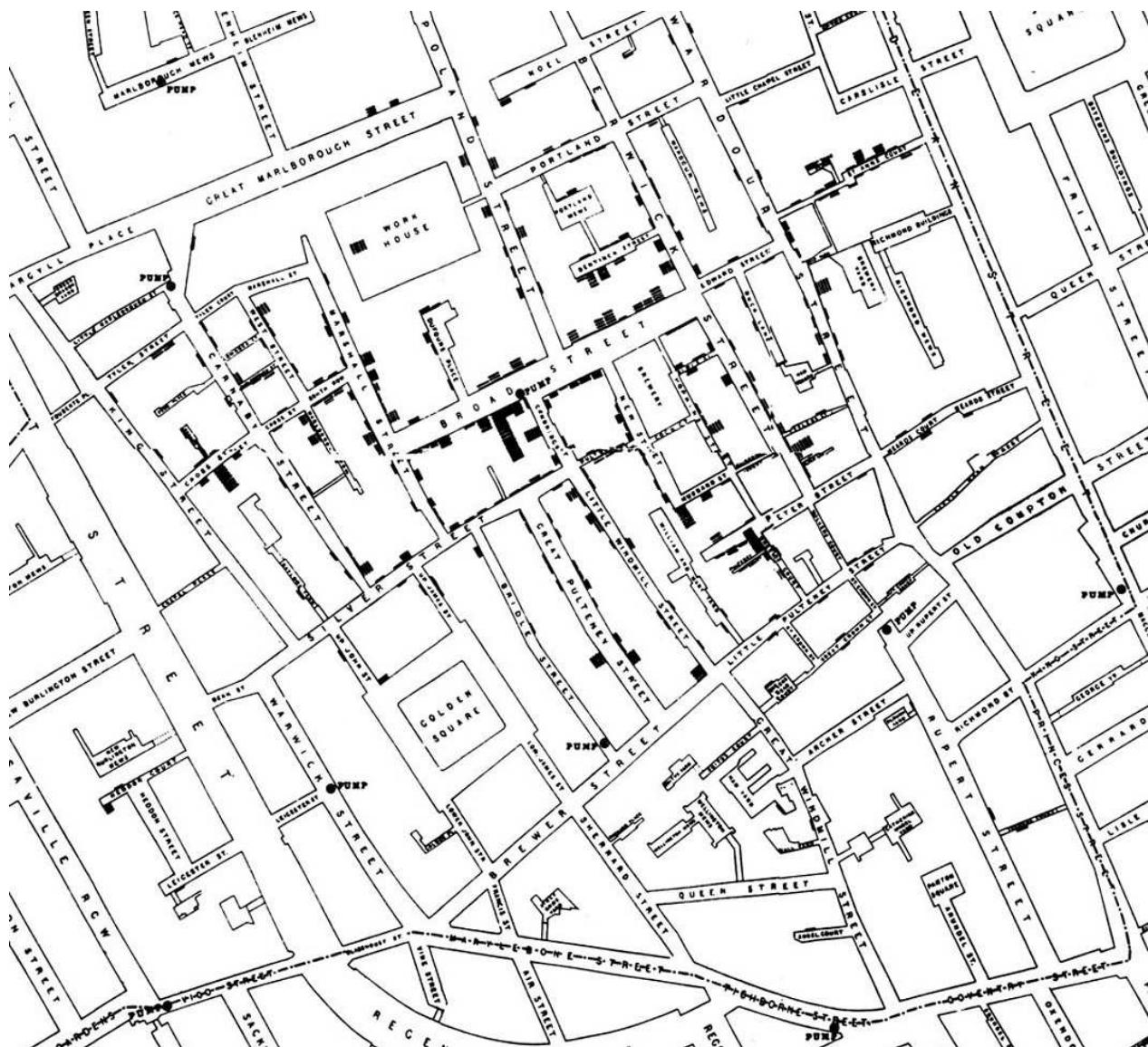
Age Group	Female	Male	Total	Male Percentage
Adult	121754366	114967088	236721454	48.57%
Baby	3869363	4041110	7910473	51.09%
Child	21903805	22851851	44755656	51.06%
Teen	14393035	15076438	29469473	51.16%

# Chapter 2

## Visualizations

While we have been able to make several interesting observations about data by simply running our eyes down the numbers in a table, our task becomes much harder as tables been larger. In data science, a picture can be worth a thousand numbers.

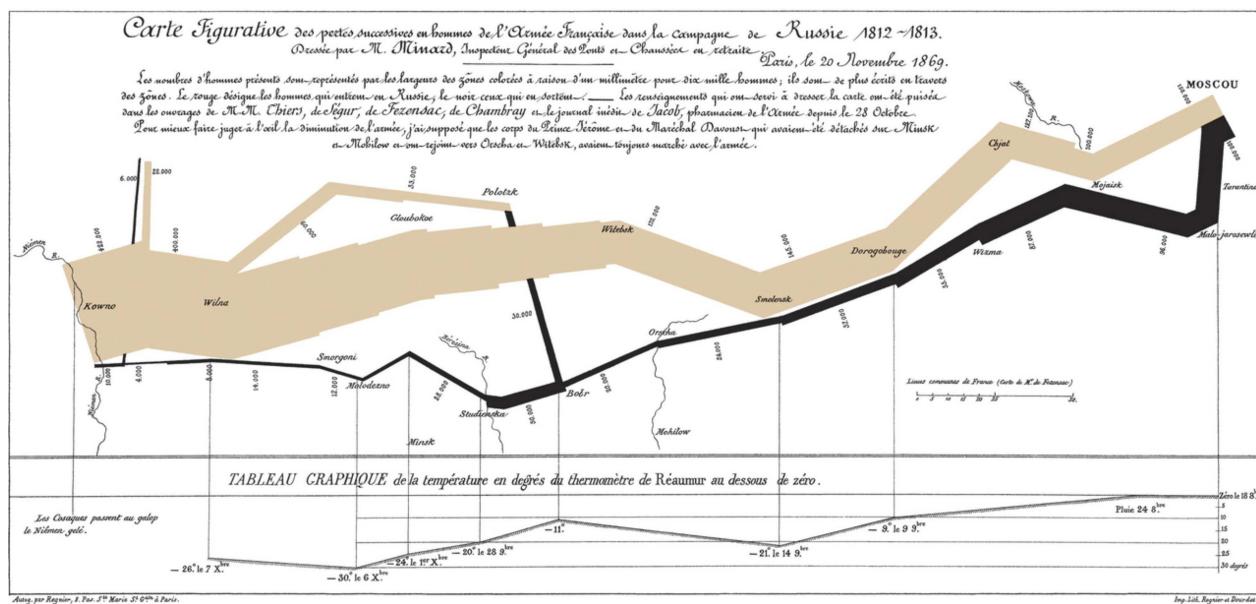
We saw an example of this earlier in the text, when we examined John Snow's map of cholera deaths in London in 1854.



Snow showed each death as a black mark at the location where the death occurred. In doing so, he plotted three variables — the number of deaths, as well as two coordinates for each location — in a single graph without any ornaments or flourishes. The simplicity of his

presentation focuses the viewer's attention on his main point, which was that the deaths were centered around the Broad Street pump.

In 1869, a French civil engineer named Charles Joseph Minard created what is still considered one of the greatest graph of all time. It shows the decimation of Napoleon's army during its retreat from Moscow. In 1812, Napoleon had set out to conquer Russia, with over 400,000 men in his army. They did reach Moscow, but were plagued by losses along the way; the Russian army kept retreating farther and farther into Russia, deliberately burning fields and destroying villages as it retreated. This left the French army without food or shelter as the brutal Russian winter began to set in. The French army turned back without a decisive victory in Moscow. The weather got colder, and more men died. Only 10,000 returned.



The graph is drawn over a map of eastern Europe. It starts at the Polish-Russian border at the left end. The light brown band represents Napoleon's army marching towards Moscow, and the black band represents the army returning. At each point of the graph, the width of the band is proportional to the number of soldiers in the army. At the bottom of the graph, Minard includes the temperatures on the return journey.

Notice how narrow the black band becomes as the army heads back. The crossing of the Berezina river was particularly devastating; can you spot it on the graph?

The graph is remarkable for its simplicity and power. In a single graph, Minard shows six variables:

- the number of soldiers
- the direction of the march
- the two coordinates of location
- the temperature on the return journey
- the location on specific dates in November and December

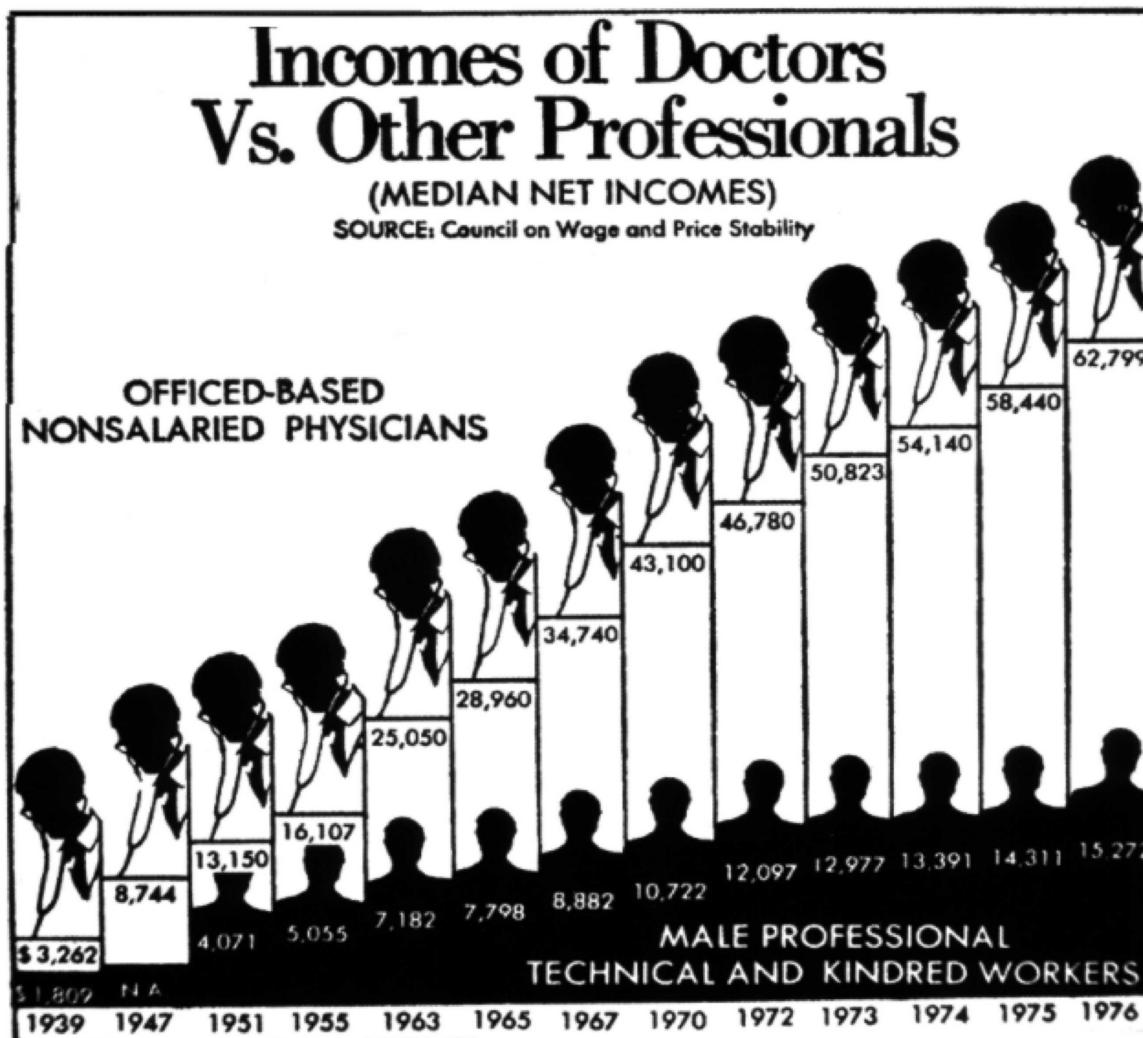
Edward Tufte, Professor at Yale and one of the world's experts on visualizing quantitative information, says that Minard's graph is "probably the best statistical graphic ever drawn."

The technology of our times allows to include animation and color. Used judiciously, without excess, these can be extremely informative, as in [this animation](#) by Gapminder.org of annual carbon dioxide emissions in the world over the past two centuries.

A caption says, "Click Play to see how USA becomes the largest emitter of CO<sub>2</sub> from 1900 onwards." The graph of total emissions shows that China has higher emissions than the United States. However, in the graph of per capita emissions, the US is higher than China, because China's population is much larger than that of the United States.

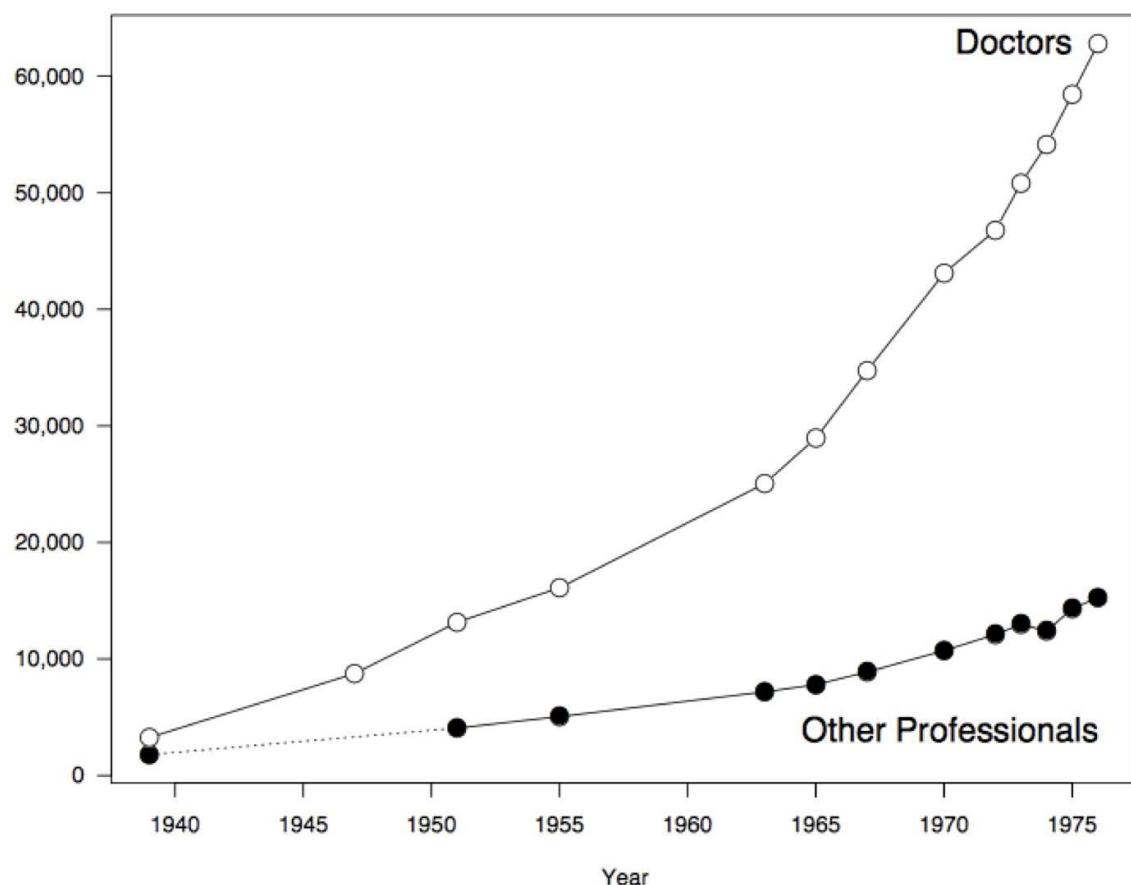
Technology can be a help as well as a hindrance. Sometimes, the ability to create a fancy picture leads to a lack of clarity in what is being displayed. Inaccurate representation of numerical information, in particular, can lead to misleading messages.

Here, from the Washington Post in the early 1980s, is a graph that attempts to compare the earnings of doctors with the earnings of other professionals over a few decades. Do we really need to see two heads (one with a stethoscope) on each bar? Tufts coined the term "chartjunk" for such unnecessary embellishments. He also deplores the "low data-to-ink ratio" which this graph unfortunately possesses.



Most importantly, the horizontal axis of the graph is not drawn to scale. This has a significant effect on the shape of the bar graphs. When drawn to scale and shorn of decoration, the graphs reveal trends that are quite different from the apparently linear growth in the original. The elegant graph below is due to Ross Ihaka, one of the originators of the statistical system R.

## Median Net Incomes



# Bar Charts

[Interact](#)

## Categorical Data and Bar Charts

Now that we have examined several graphics produced by others, it is time to produce some of our own. We will start with *bar charts*, a type of graph with which you might already be familiar. A bar chart shows the distribution of a *categorical variable*, that is, a variable whose values are categories. In human populations, examples of categorical variables include gender, ethnicity, marital status, country of citizenship, and so on.

A bar chart consists of a sequence of rectangular bars, one corresponding to each category. The length of each bar is proportional to the number of entries in the corresponding category.

The Kaiser Family Foundation has complied Census data on the distribution of race and ethnicity in the United States.

<http://kff.org/other/state-indicator/distribution-by-raceethnicity/>

<http://kff.org/other/state-indicator/children-by-raceethnicity/>

Here are some of the data, arranged by state. The table `children` contains data for people who were younger than 18 years old in 2014; `everyone` contains the data for people of all ages in 2014. Some missing values in these tables are represented as 0.

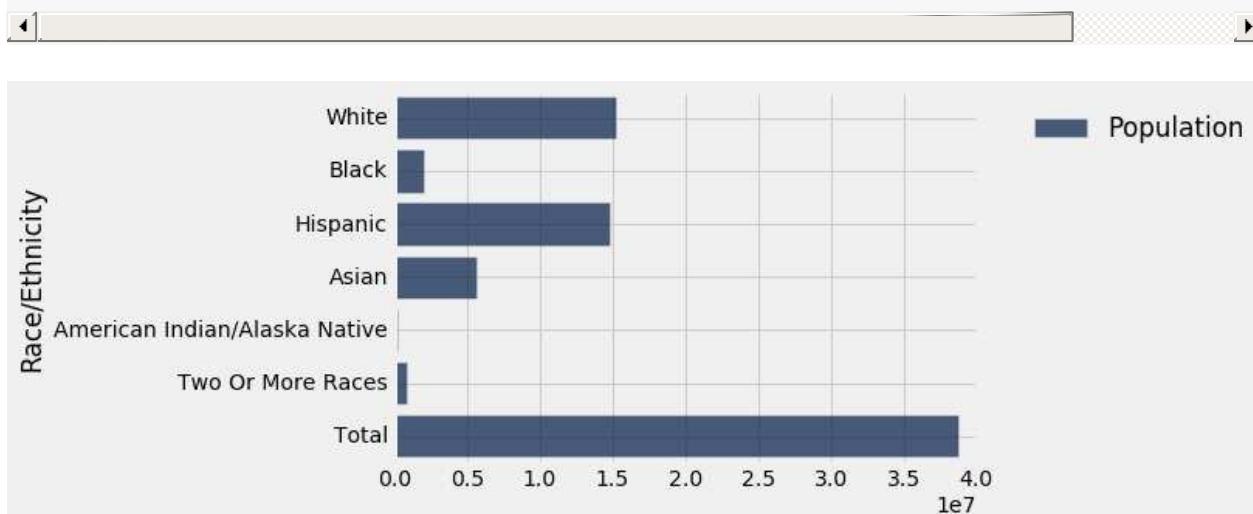
```
everyone = Table.read_table('kaiser_ethnicity_everyone.csv')
children = Table.read_table('kaiser_ethnicity_children.csv')
children.set_format(2, NumberFormatter)
everyone.set_format(2, NumberFormatter)
```

State	Race/Ethnicity	Population
Alabama	White	3,167,600
Alabama	Black	1,269,200
Alabama	Hispanic	191,000
Alabama	Asian	77,300
Alabama	American Indian/Alaska Native	0
Alabama	Two Or More Races	56,100
Alabama	Total	4,768,000
Alaska	White	396,400
Alaska	Black	17,000
Alaska	Hispanic	59,200

... (347 rows omitted)

The method `barh` takes two arguments, a column containing categories and a column containing numeric quantities. It generates a horizontal bar chart. This chart focuses on California.

```
everyone.where('State', 'California').barh('Race/Ethnicity', 'Popul
```



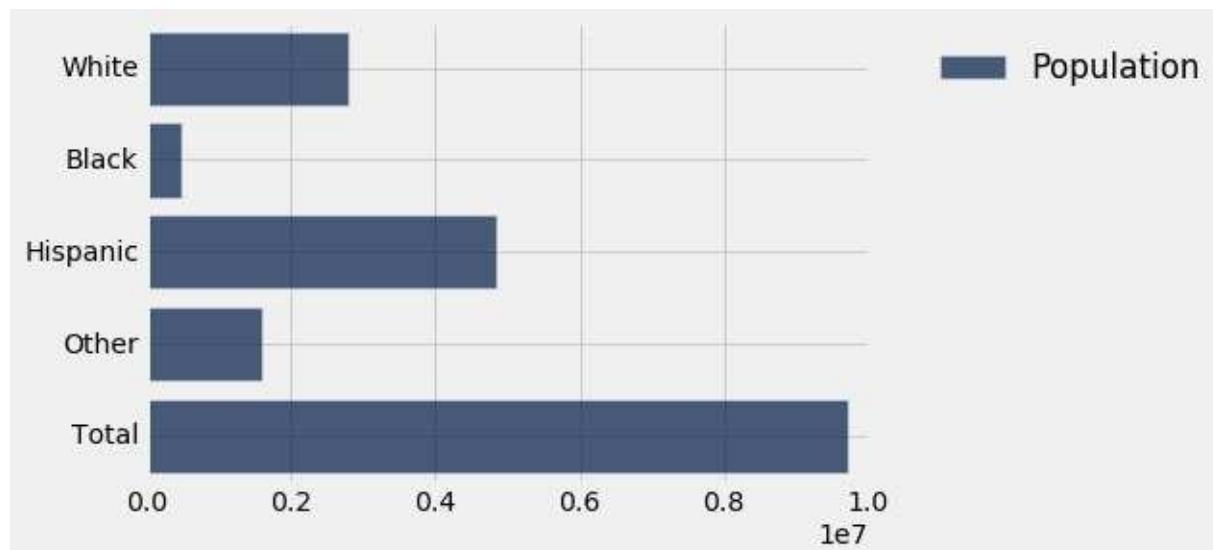
The table of children also contains information about California, although there are fewer categories of race/ethnicity.

```
children.where('State', 'California')
```

State	Race/Ethnicity	Population
California	White	2,786,000
California	Black	484,200
California	Hispanic	4,849,400
California	Other	1,590,100
California	Total	9,709,700

Again, we can display these data in a horizontal bar chart. The `barh` method can take column indices as well as labels. This chart looks similar, but involves fewer categories because the Kaiser Family Foundation only provides these data about children.

```
children.where('State', 'California').barh(1, 2)
```



It's not obvious how to compare these two charts. To help us draw conclusions, we will combine these two tables into one, which contains both everyone and children as separate columns. This transformation will require several steps. First, to ensure that values are comparable, we will divide each population by the total for the state to compute state proportions.

```
totals = everyone.join('State', everyone.where(1, 'Total').relabeled
totals
```

State	Race/Ethnicity	Population	State Total
Alabama	White	3167600	4768000
Alabama	Black	1269200	4768000
Alabama	Hispanic	191000	4768000
Alabama	Asian	77300	4768000
Alabama	American Indian/Alaska Native	0	4768000
Alabama	Two Or More Races	56100	4768000
Alabama	Total	4768000	4768000
Alaska	White	396400	695700
Alaska	Black	17000	695700
Alaska	Hispanic	59200	695700

... (347 rows omitted)

```
proportions = everyone.select([0, 1]).with_column('Proportion', tot
proportions.set_format(2, PercentFormatter)
```

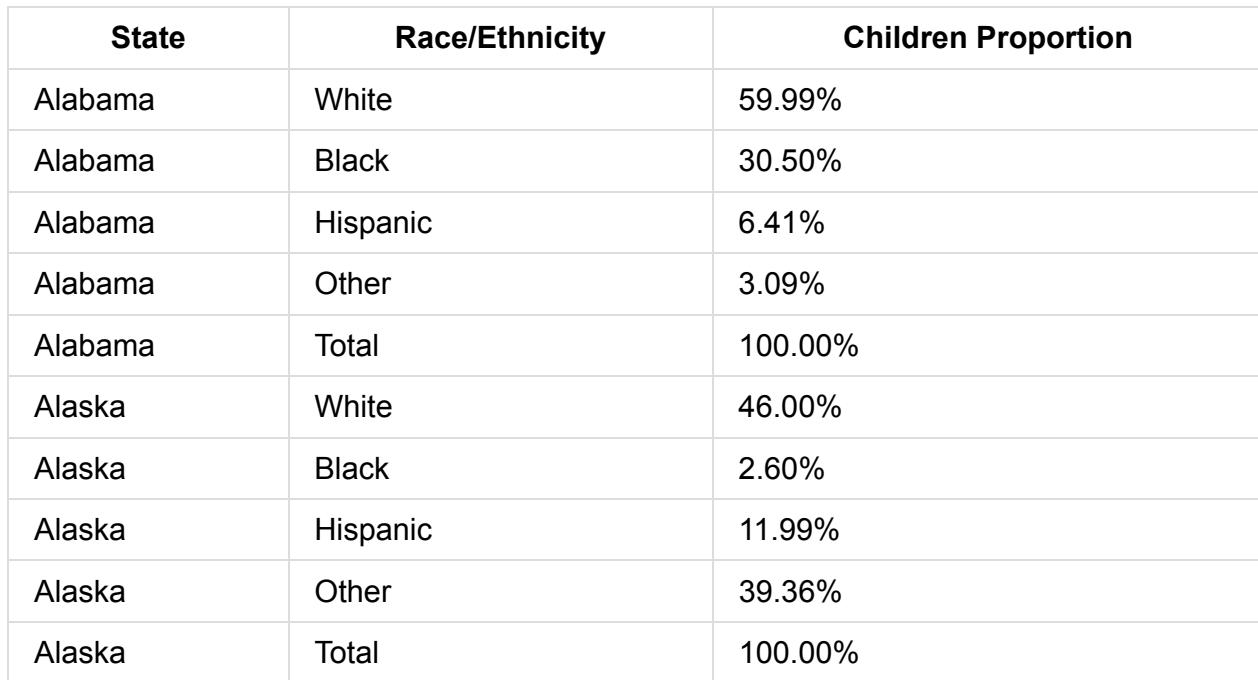
State	Race/Ethnicity	Proportion
Alabama	White	66.43%
Alabama	Black	26.62%
Alabama	Hispanic	4.01%
Alabama	Asian	1.62%
Alabama	American Indian/Alaska Native	0.00%
Alabama	Two Or More Races	1.18%
Alabama	Total	100.00%
Alaska	White	56.98%
Alaska	Black	2.44%
Alaska	Hispanic	8.51%

... (347 rows omitted)

This same transformation can be applied to the `children` table. Below, we chain together several table methods in order to express the transformation in a single long line. Typically, it's best to express transformations in pieces (above), but programming languages are quite

flexible in allowing programmers to combine expressions (below).

```
children_proportions = children.select([0, 1]).with_column(
    'Children Proportion', children.column(2) / children.join('State')
children_proportions.set_format(2, PercentFormatter)
```



State	Race/Ethnicity	Children Proportion
Alabama	White	59.99%
Alabama	Black	30.50%
Alabama	Hispanic	6.41%
Alabama	Other	3.09%
Alabama	Total	100.00%
Alaska	White	46.00%
Alaska	Black	2.60%
Alaska	Hispanic	11.99%
Alaska	Other	39.36%
Alaska	Total	100.00%

... (245 rows omitted)

The following function takes the name of a state. It joins together the proportions for everyone and children into a single table. When the `join` method combines two columns with different values, only the matching elements are retained. In this case, any race/ethnicity not represented in both tables will be excluded.

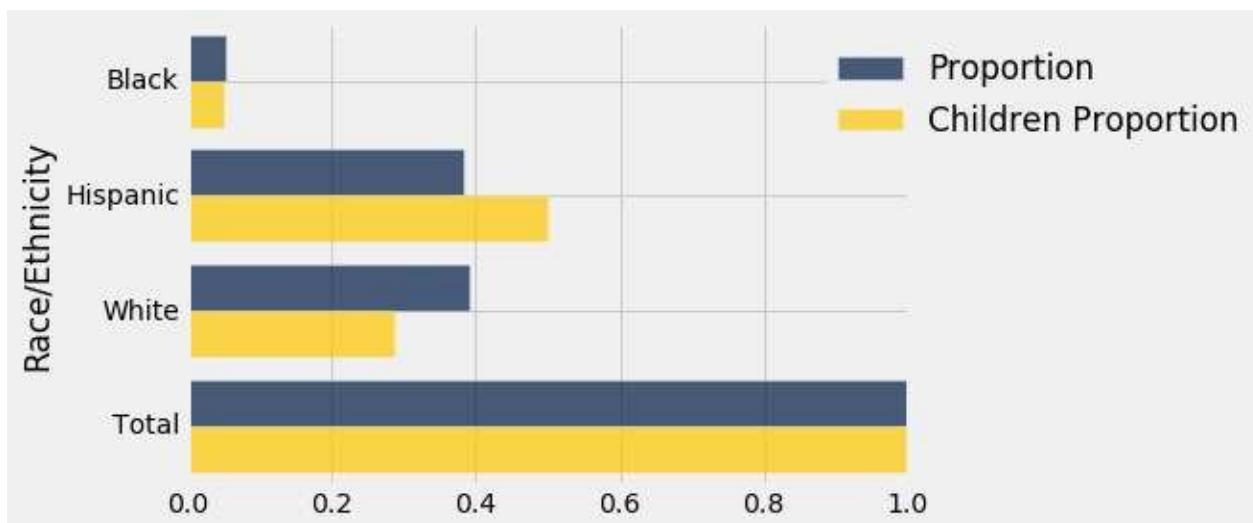
```
def compare(state):
    prop_for_state = proportions.where('State', state).drop(0)
    children_prop_for_state = children_proportions.where('State', s
joined = prop_for_state.join('Race/Ethnicity', children_prop_for_
joined.set_format([1, 2], PercentFormatter)
return joined.sort('Proportion')

compare('California')
```

Race/Ethnicity	Proportion	Children Proportion
Black	5.34%	4.99%
Hispanic	38.21%	49.94%
White	39.20%	28.69%
Total	100.00%	100.00%

The `barh` method can display the data from multiple columns on a single chart. When called with only one argument indicating the column of categories, all other columns are displayed as sets of bars.

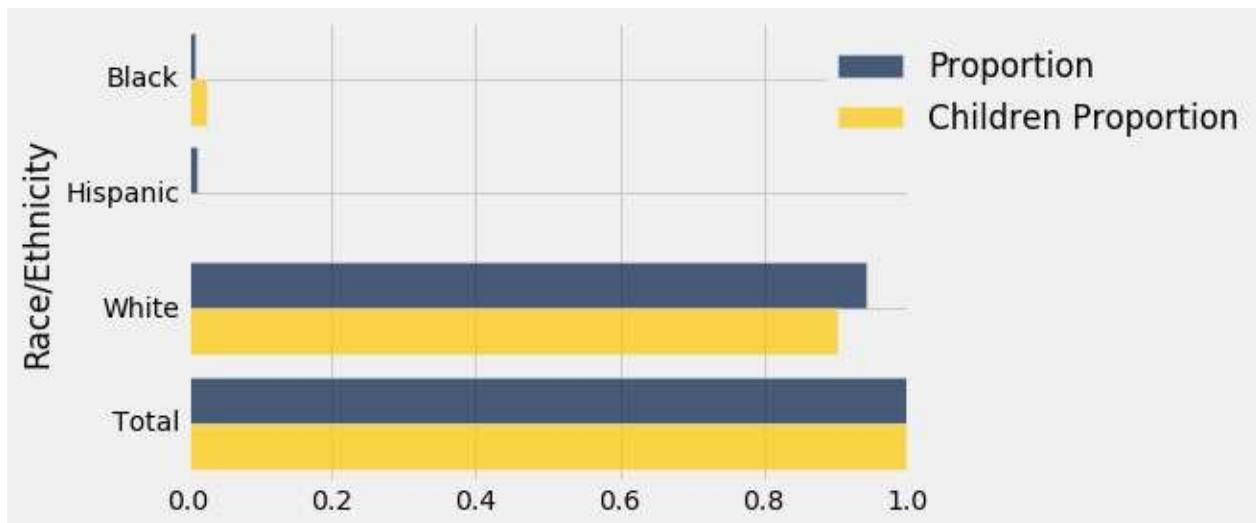
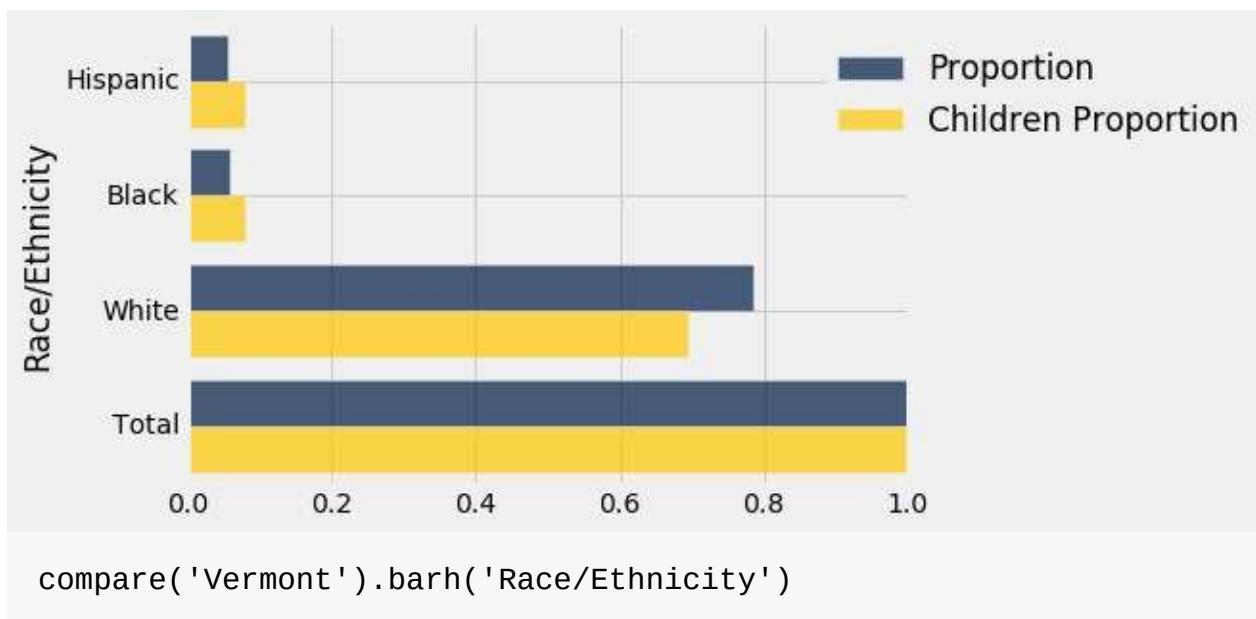
```
compare('California').barh('Race/Ethnicity')
```



This chart shows the changing composition of the state of California. In 2014, fewer than 40% of Californians were Hispanic. But 50% of the children were, which is the crucial ingredient in predictions that Hispanics will eventually be a majority in California.

What about other states? Our function allows quick visual comparisons for any state.

```
compare('Minnesota').barh('Race/Ethnicity')
```



# Histograms

[Interact](#)

## Quantitative Data and Histograms

Many of the variables that data scientists study are *quantitative*. For instance, we can study the amount of revenue earned by movies in recent decades. Our source is the [Internet Movie Database](#), an online database that contains information about movies, television shows, video games, and so on.

The table `top` consists of [U.S.A.'s top grossing movies (<http://www.boxofficemojo.com/alltime/adjusted.htm>) of all time. The first column contains the title of the movie; *Star Wars: The Force Awakens* has the top rank, with a box office gross amount of more than 900 million dollars in the United States. The second column contains the name of the studio; the third contains the U.S. box office gross in dollars; the fourth contains the gross amount that would have been earned from ticket sales at 2016 prices; and the fifth contains the release year of the movie.

There are 200 movies on the list. Here are the top ten.

```
top = Table.read_table('top_movies.csv')
top.set_format([2, 3], NumberFormatter)
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906,723,418	906,723,400	2015
Avatar	Fox	760,507,625	846,120,800	2009
Titanic	Paramount	658,672,302	1,178,627,900	1997
Jurassic World	Universal	652,270,625	687,728,000	2015
Marvel's The Avengers	Buena Vista (Disney)	623,357,910	668,866,600	2012
The Dark Knight	Warner Bros.	534,858,444	647,761,600	2008
Star Wars: Episode I - The Phantom Menace	Fox	474,544,677	785,715,000	1999
Star Wars	Fox	460,998,007	1,549,640,500	1977
Avengers: Age of Ultron	Buena Vista (Disney)	459,005,868	465,684,200	2015
The Dark Knight Rises	Warner Bros.	448,139,099	500,961,700	2012

... (190 rows omitted)

Three-digit numbers are easier to work with than nine-digit numbers. So, we will add an additional *millions* column containing the adjusted gross box office revenue in millions.

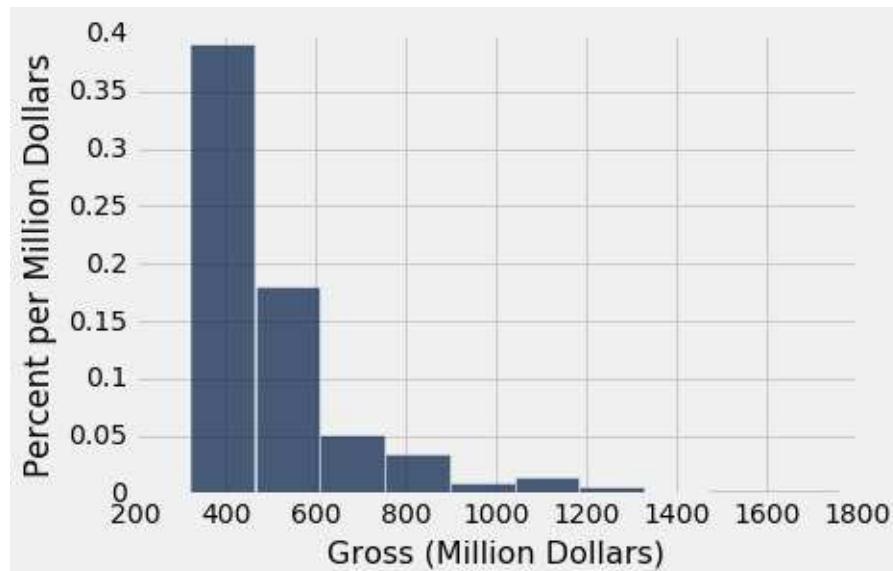
```
millions = top.select(0).with_column('Gross', np.round(top.column('Gross') / 1000000))
```

Title	Gross
Star Wars: The Force Awakens	906.72
Avatar	846.12
Titanic	1178.63
Jurassic World	687.73
Marvel's The Avengers	668.87
The Dark Knight	647.76
Star Wars: Episode I - The Phantom Menace	785.72
Star Wars	1549.64
Avengers: Age of Ultron	465.68
The Dark Knight Rises	500.96

... (190 rows omitted)

The `hist` method generates a *histogram* of the values in a column. The optional `unit` argument is used to label the axes.

```
millions.hist('Gross', unit="Million Dollars")
```



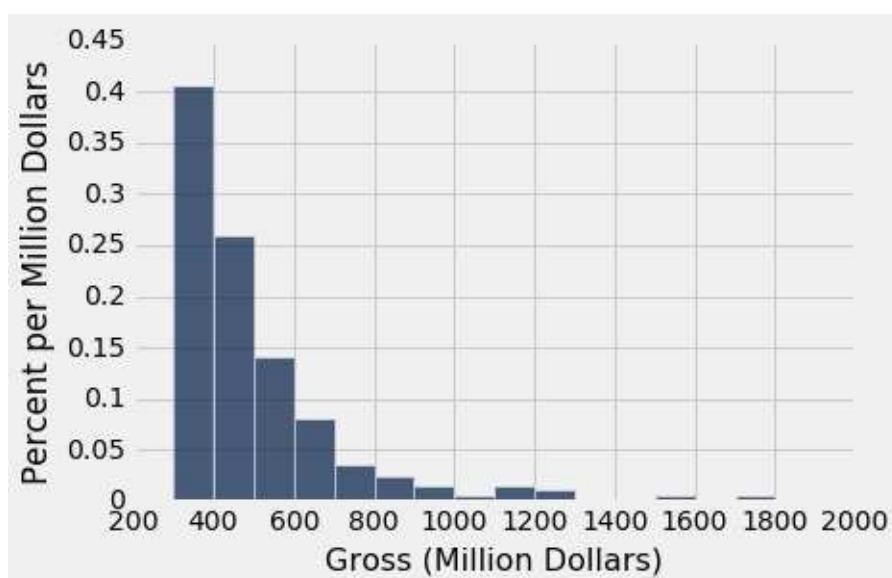
The figure above shows the distribution of the amounts grossed, in millions of dollars. The amounts have been grouped into contiguous intervals called *bins*. Although in this dataset no movie grossed an amount that is exactly on the edge between two bins, it is worth noting that `hist` has an *endpoint convention*: bins include the data at their left endpoint, but not the data at their right endpoint. Sometimes, adjustments have to be made in the first or last bin,

to ensure that the smallest and largest values of the variable are included. You saw an example of such an adjustment in the Census data used in the Tables section, where an age of "100" years actually meant "100 years old or older."

We can see that there are 10 bins (some bars are so low that they are hard to see), and that they all have the same width. We can also see that there the list contains no movie that grossed fewer than 300 million dollars; that is because we are considering only the top grossing movies of all time. It is a little harder to see exactly where the edges of the bins are placed. For example, it is not clear exactly where the value 500 lies on the horizontal axis, and so it is hard to judge exactly where the first bar ends and the second begins.

The optional argument *bins* can be used with *hist* to specify the edges of the bars. It must consist of a sequence of numbers that includes the left end of the first bar and the right end of the last bar. As the highest gross amount is somewhat over 760 on the horizontal scale, we will start by setting *bins* to be the array consisting of the numbers 300, 400, 500, and so on, ending with 2000.

```
millions.hist('Gross', unit="Million Dollars", bins=np.arange(300, 2
```



This figure is easier to read. On the horizontal axis, the labels 200, 400, 600, and so on are centered at the corresponding values. The tallest bar is for movies that grossed between 300 and million and 400 million (adjusted) dollars; the bar for 400 to 500 million is about 5/8th as high. The height of each bar is proportional to the number of movies that fall within the x-axis range of the bar.

A very small number grossed more than 700 million dollars. This results in the figure being "skewed to the right," or, less formally, having "a long right hand tail." Distributions of variables like income or rent often have this kind of shape.

The counts of values in different ranges can be computed from a table using the `bin` method, which takes a column label or index and an optional sequence or number of bins. The result is a tabular form of a histogram; it lists the counts of all values in the 'Millions' column that are greater than or equal to the indicated `bin`, but less than the next `bin`'s starting value.

```
bins = millions.bin('Gross', bins=np.arange(300, 2001, 100))
bins
```

bin	Gross count
300	81
400	52
500	28
600	16
700	7
800	5
900	3
1000	1
1100	3
1200	2
... (8 rows omitted)	

The vertical axis of a histogram is the *density scale*. The height of each bar is the proportion of elements that fall into the corresponding bin, divided by the width of the bin. For example, the height of the bin from 300 to 400, which contains 81 of the 200 movies, is

$$\frac{81}{200} \cdot \frac{1}{400-300} = 0.405\%.$$

```
starts = bins.column(0)
counts = bins.column(1)
counts.item(0) / sum(counts) / (starts.item(1) - starts.item(0))
```

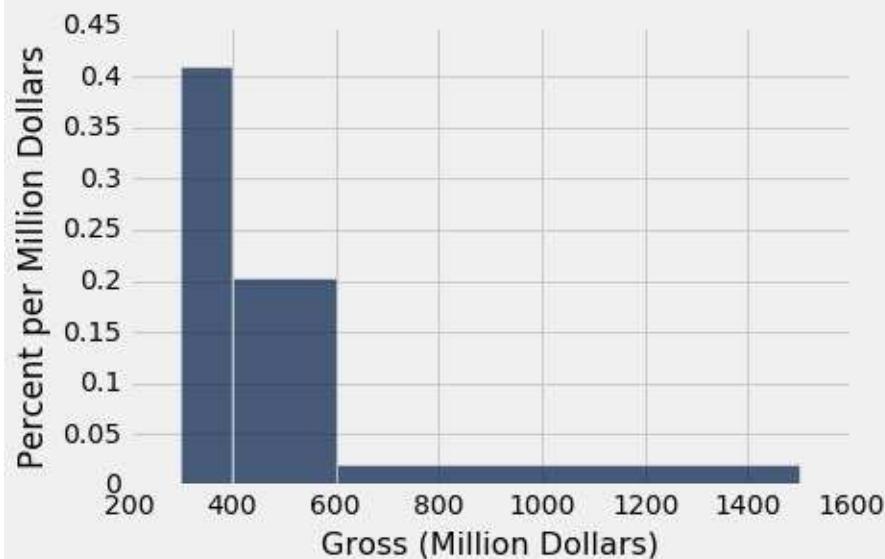
```
0.004050000000000006
```

The total area of this bar is 0.405, 100 times greater because its width is 100. This area is the proportion of all values in the `Millions` column that fall into the 300-400 bin. The area of all bars in a histogram sums to 1.

A histogram may contain bins of unequal size, but the area of all bars will still sum to 1.

Below, the values in the `Millions` column are binned into four categories.

```
millions.hist('Gross', unit="Million Dollars", bins=[300, 400, 600,
```



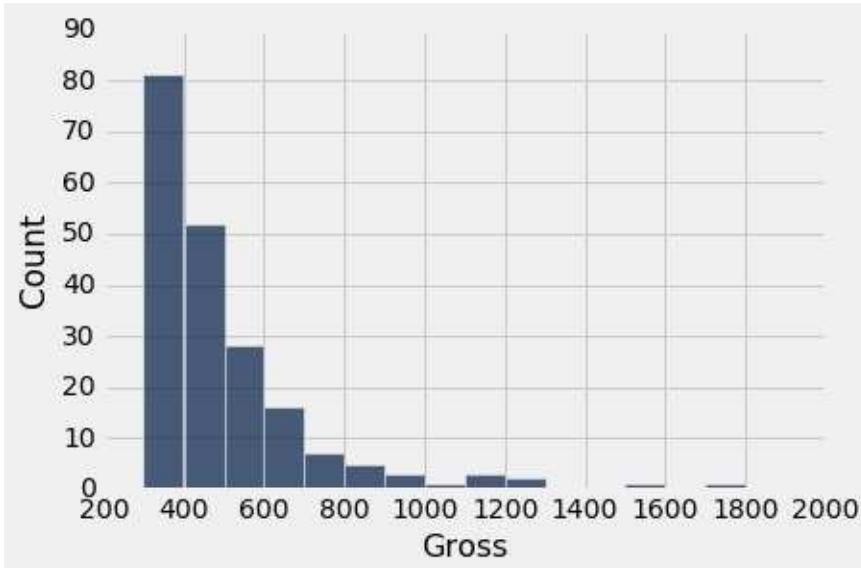
Although the ranges 300-400 and 400-600 have nearly identical counts, the former is twice as tall as the latter because it is only half as wide. The density of values in that range is twice as high. Histograms display in visual form where on the number line values are most concentrated.

```
millions.bin('Gross', bins=[300, 400, 600, 1500])
```

bin	Gross count
300	81
400	80
600	37
1500	0

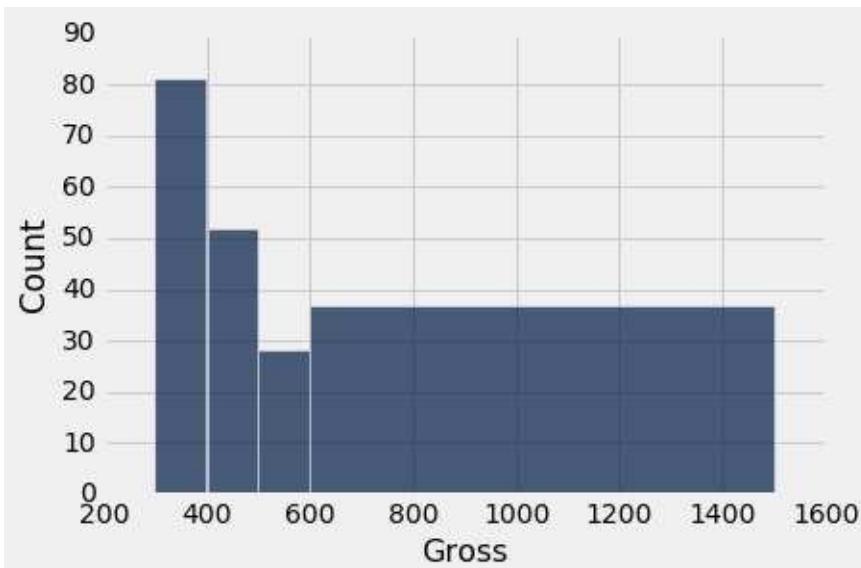
**Histogram of Counts.** It is possible to display counts directly in a chart, using the `normed=False` option of the `hist` method. The resulting chart has the same shape when bins all have equal widths, but the bar areas do not sum to 1.

```
millions.hist('Gross', bins=np.arange(300, 2001, 100), normed=False)
```



While the count scale is perhaps more natural to interpret than the density scale, the chart becomes highly misleading when bins have different widths. Below, it appears (due to the count scale) that high-grossing movies are quite common, when in fact we have seen that they are relatively rare.

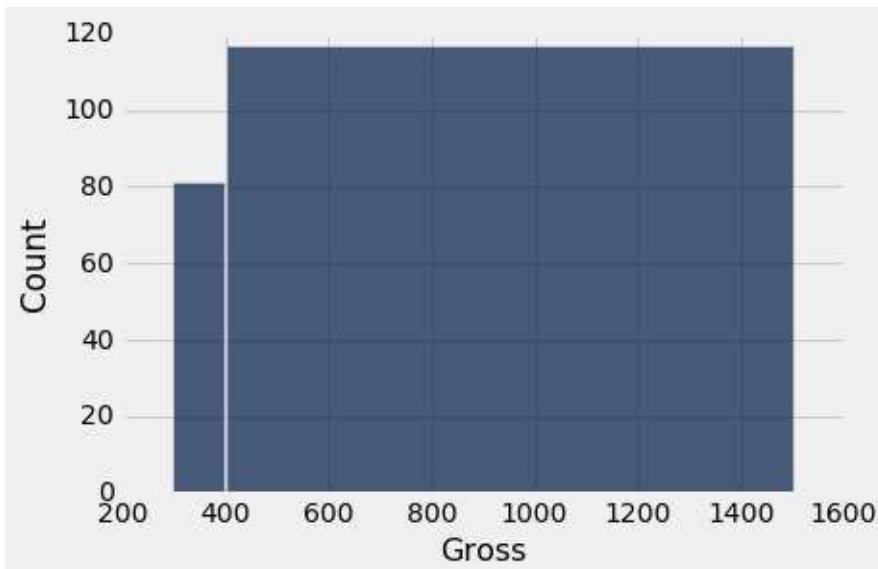
```
millions.hist('Gross', bins=[300, 400, 500, 600, 1500], normed=False)
```



Even though the method used is called *hist*, **the figure above is NOT A HISTOGRAM**. It gives the impression that movies are evenly distributed in the 300-600 range, but they are not. Moreover, it exaggerates the density of movies grossing more than 600 million. The

height of each bar is simply plotted the number of movies in the bin, *without accounting for the difference in the widths of the bins*. The picture becomes even more skewed if the last two bins are combined.

```
millions.hist('Gross', bins=[300, 400, 1500], normed=False)
```



In this count-based histogram, the shape of the distribution of movies is lost entirely.

### So what is a histogram?

The figure above shows that what the eye perceives as "big" is area, not just height. This is particularly important when the bins are of different widths.

That is why a histogram has two defining properties:

1. The bins are contiguous (though some might be empty) and are drawn to scale.
2. The **area** of each bar is proportional to the number of entries in the bin.

Property 2 is the key to drawing a histogram, and is usually achieved as follows:

$$\text{area of bar} = \text{proportion of entries in bin}$$

When drawn using this method, the histogram is said to be drawn *on the density scale*, and the total area of the bars is equal to 1.

To calculate the height of each bar, use the fact that the bar is a rectangle:

$$\text{area of bar} = \text{height of bar} \times \text{width of bin}$$

and so

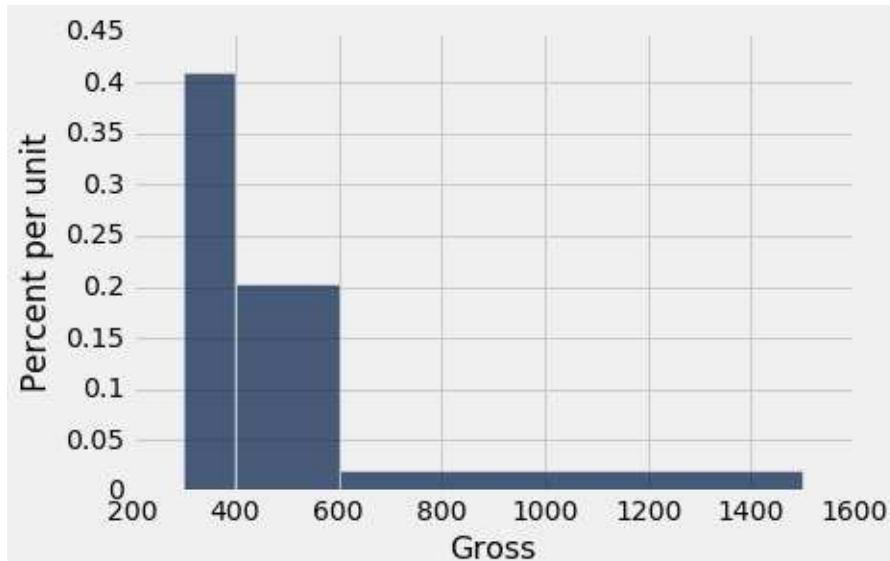
$$\begin{aligned}\text{height of bar} &= \frac{\text{area of bar}}{\text{width of bin}} \\ &= \frac{\text{proportion of entries in bin}}{\text{width of bin}}\end{aligned}$$

**The level of detail, and the flat tops of the bars**

Even though the density scale correctly represents proportions using area, some detail is lost by placing different values into the same bins.

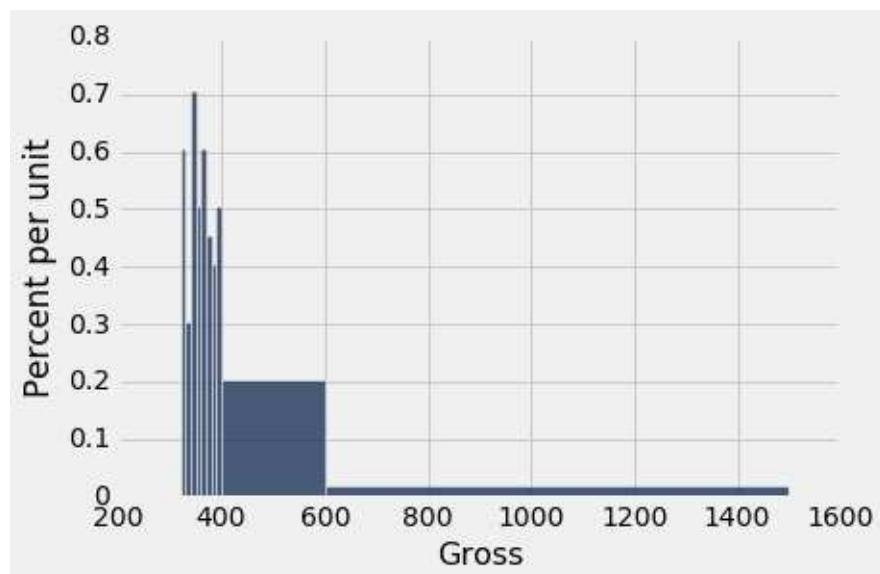
Take another look at the [300, 400) bin in the figure below. The flat top of the bar, at a level near 0.4%, hides the fact that the movies are somewhat unevenly distributed across the bin.

```
millions.hist('Gross', bins=[300, 400, 600, 1500])
```



To see this, let us split the [150, 200) bin into ten narrower bins of width 10 million dollars each:

```
millions.hist('Gross', bins=[300, 310, 320, 330, 340, 350, 360, 370])
```



Some of the skinny bars are taller than 0.4% and others are shorter. By putting a flat top at 0.4% over the whole bin, we are deciding to ignore the finer detail and use the flat level as a rough approximation. Often, though not always, this is sufficient for understanding the general shape of the distribution.

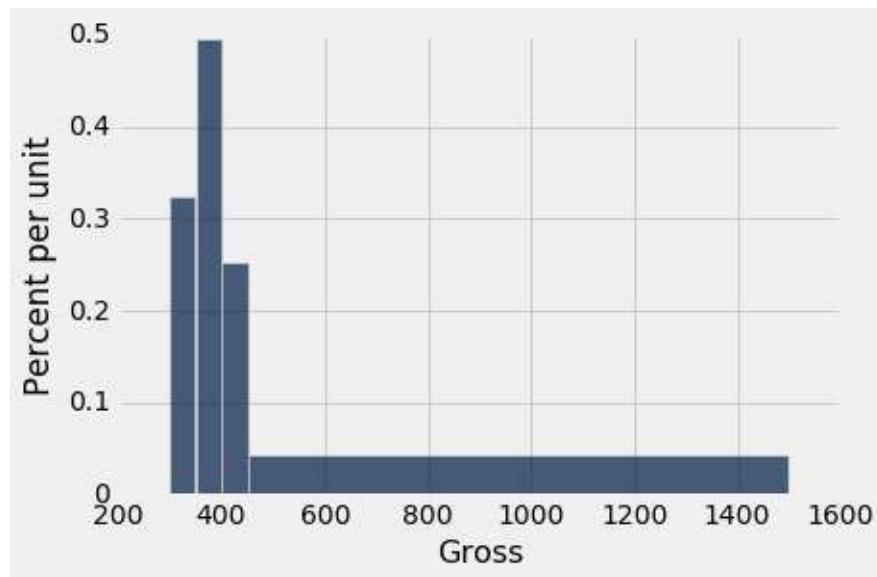
Notice that because we have the entire dataset, we can draw the histogram in as fine a level of detail as the data and our patience will allow. However, if you are looking at a histogram in a book or on a website, and you don't have access to the underlying dataset, then it becomes important to have a clear understanding of the "rough approximation" of the flat tops.

### The density scale

The height of each bar is a proportion divided by a bin width. Thus, for this dataset, the values on the vertical axis are "proportions per million dollars." To understand this better, look again at the [150, 200) bin. The bin is 50 million dollars wide. So we can think of it as consisting of 50 narrow bins that are each 1 million dollars wide. The bar's height of roughly "0.004 per million dollars" means that in each of those 50 skinny bins of width 1 million dollars, the proportion of movies is roughly 0.004.

Thus the height of a histogram bar is a proportion per unit on the horizontal axis, and can be thought of as the *density* of entries per unit width.

```
millions.hist('Gross', bins=[300, 350, 400, 450, 1500])
```

**Density Q&A**

Look again at the histogram, and this time compare the [400, 450) bin with the [450, 1500) bin.

**Q:** Which has more movies in it?

**A:** The [450, 1500) bin. It has 92 movies, compared with 25 movies in the [400, 450) bin.

**Q:** Then why is the [450, 1500) bar so much shorter than the [400, 450) bar?

**A:** Because height represents density per unit width, not the number of movies in the bin. The [450, 1500) bin has more movies than the [400, 450) bin, but it is also a whole lot wider. So the density is much lower.

```
millions.bin('Gross', bins=[300, 350, 400, 450, 1500])
```

bin	Gross count
300	32
350	49
400	25
450	92
1500	0

**Bar chart or histogram?**

Bar charts display the distributions of categorical variables. All the bars in a bar chart have the same width. The lengths (or heights, if the bars are drawn vertically) of the bars are proportional to the number of entries.

Histograms display the distributions of quantitative variables. The bars can have different widths. The areas of the bars are proportional to the number of entries.

### Multiple histograms

In all the examples in this section, we have drawn a single histogram. However, if a data table contains several columns, then *barh* and *hist* can be used to draw several charts at once. We will cover this feature in a later section.

# Chapter 3

## Sampling

Sampling is a powerful computational technique in data analysis. It allows us to investigate both issues of probability and statistics. *Probability* is the study of the outcomes of random processes; processes that include chance. *Statistics* is just the opposite: it is the study of random processes given observations of their outcomes. Put another way, probability focuses on what will be observed given what we know about the world. Statistics focuses on what we can infer about the world given what we have observed. Random Sampling is a central tool in both directions of inference.

# Sampling

## Interact

In this section, we will continue to use the `top_movies.csv` data set.

```
top = Table.read_table('top_movies.csv')
top.set_format([2, 3], NumberFormatter)
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906,723,418	906,723,400	2015
Avatar	Fox	760,507,625	846,120,800	2009
Titanic	Paramount	658,672,302	1,178,627,900	1997
Jurassic World	Universal	652,270,625	687,728,000	2015
Marvel's The Avengers	Buena Vista (Disney)	623,357,910	668,866,600	2012
The Dark Knight	Warner Bros.	534,858,444	647,761,600	2008
Star Wars: Episode I - The Phantom Menace	Fox	474,544,677	785,715,000	1999
Star Wars	Fox	460,998,007	1,549,640,500	1977
Avengers: Age of Ultron	Buena Vista (Disney)	459,005,868	465,684,200	2015
The Dark Knight Rises	Warner Bros.	448,139,099	500,961,700	2012

... (190 rows omitted)

## Sampling rows of a table

### Deterministic Samples

When you simply specify which elements of a set you want to choose, without any chances involved, you create a *deterministic sample*.

A deterministic sample from the rows of a table can be constructed using the `take` method. Its argument is a sequence of integers, and it returns a table containing the corresponding rows of the original table.

The code below returns a table consisting of the rows indexed 3, 18, and 100 in the table `top`. Since the original table is sorted by `rank`, which begins at 1, the zero-indexed rows always have a rank that is one greater than the index. However, the `take` method does not inspect the information in the rows to construct its sample.

```
top.take([3, 18, 100])
```

Title	Studio	Gross	Gross (Adjusted)	Year
Jurassic World	Universal	652,270,625	687,728,000	2015
Spider-Man	Sony	403,706,375	604,517,300	2002
Gone with the Wind	MGM	198,676,459	1,757,788,200	1939

We can select evenly spaced rows by calling `np.arange` and passing the result to `take`. In the example below, we start with the first row (index 0) of `top`, and choose every 40th row after that until we reach the end of the table.

```
top.take(np.arange(0, top.num_rows, 40))
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906,723,418	906,723,400	2015
Shrek the Third	Paramount/Dreamworks	322,719,944	408,090,600	2007
Bruce Almighty	Universal	242,829,261	350,350,700	2003
Three Men and a Baby	Buena Vista (Disney)	167,780,960	362,822,900	1987
Saturday Night Fever	Paramount	94,213,184	353,261,200	1977

## Probability Samples

Much of data science consists of making conclusions based on the data in random samples. Correctly interpreting analyses based on random samples requires data scientists to examine exactly what random samples are.

A *population* is the set of all elements from whom a sample will be drawn.

A *probability sample* is one for which it is possible to calculate, before the sample is drawn, the chance with which any subset of elements will enter the sample.

In a probability sample, all elements need not have the same chance of being chosen. For example, suppose you choose two people from a population that consists of three people A, B, and C, according to the following scheme:

- Person A is chosen with probability 1.
- One of Persons B or C is chosen according to the toss of a coin: if the coin lands heads, you choose B, and if it lands tails you choose C.

This is a probability sample of size 2. Here are the chances of entry for all non-empty subsets:

```
A: 1  
B: 1/2  
C: 1/2  
AB: 1/2  
AC: 1/2  
BC: 0  
ABC: 0
```

Person A has a higher chance of being selected than Persons B or C; indeed, Person A is certain to be selected. Since these differences are known and quantified, they can be taken into account when working with the sample.

To draw a probability sample, we need to be able to choose elements according to a process that involves chance. A basic tool for this purpose is a *random number generator*. There are several in Python. Here we will use one that is part of the module `random`, which in turn is part of the module `numpy`.

The method `randint`, when given two arguments `low` and `high`, returns an integer picked uniformly at random between `low` and `high`, including `low` but excluding `high`. Run the code below several times to see the variability in the integers that are returned.

```
np.random.randint(3, 8) # select once at random from 3, 4, 5, 6, 7
```

7

## A Systematic Sample

Imagine all the elements of the population listed in a sequence. One method of sampling starts by choosing a random position early in the list, and then evenly spaced positions after that. The sample consists of the elements in those positions. Such a sample is called a *systematic sample*.

Here we will choose a systematic sample of the rows of `top`. We will start by picking one of the first 10 rows at random, and then we will pick every 10th row after that.

```
"""Choose a random start among rows 0 through 9;
then take every 10th row."""
```

```
start = np.random.randint(0, 10)
top.take(np.arange(start, top.num_rows, 10))
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars	Fox	460,998,007	1,549,640,500	1977
The Hunger Games	Lionsgate	408,010,692	442,510,400	2012
The Passion of the Christ	NM	370,782,930	519,432,100	2004
Alice in Wonderland (2010)	Buena Vista (Disney)	334,191,110	365,718,600	2010
Star Wars: Episode II - Attack of the Clones	Fox	310,676,740	465,175,700	2002
The Sixth Sense	Buena Vista (Disney)	293,506,292	500,938,400	1999
The Hangover	Warner Bros.	277,322,503	323,343,300	2009
Raiders of the Lost Ark	Paramount	248,159,971	770,183,000	1981
The Lost World: Jurassic Park	Universal	229,086,679	434,216,600	1997
Austin Powers: The Spy Who Shagged Me	New Line	206,040,086	352,863,900	1999

... (10 rows omitted)

Run the code a few times to see how the output varies.

This systematic sample is a probability sample. In this scheme, all rows do have the same chance of being chosen. But that is not true of other subsets of the rows. Because the selected rows are evenly spaced, most subsets of rows have no chance of being chosen.

The only subsets that are possible are those that consist of rows all separated by multiples of 10. Any of those subsets is selected with chance 1/10.

### **Random sample with replacement**

Some of the simplest probability samples are formed by drawing repeatedly, uniformly at random, from the list of elements of the population. If the draws are made without changing the list between draws, the sample is called a *random sample with replacement*. You can imagine making the first draw at random, replacing the element drawn, and then drawing again.

In a random sample with replacement, each element in the population has the same chance of being drawn, and each can be drawn more than once in the sample.

If you want to draw a sample of people at random to get some information about a population, you might not want to sample with replacement – drawing the same person more than once can lead to a loss of information. But in data science, random samples with replacement arise in two major areas:

- studying probabilities by simulating tosses of a coin, rolls of a die, or gambling games
- creating new samples from a sample at hand

The second of these areas will be covered later in the course. For now, let us study some long-run properties of probabilities.

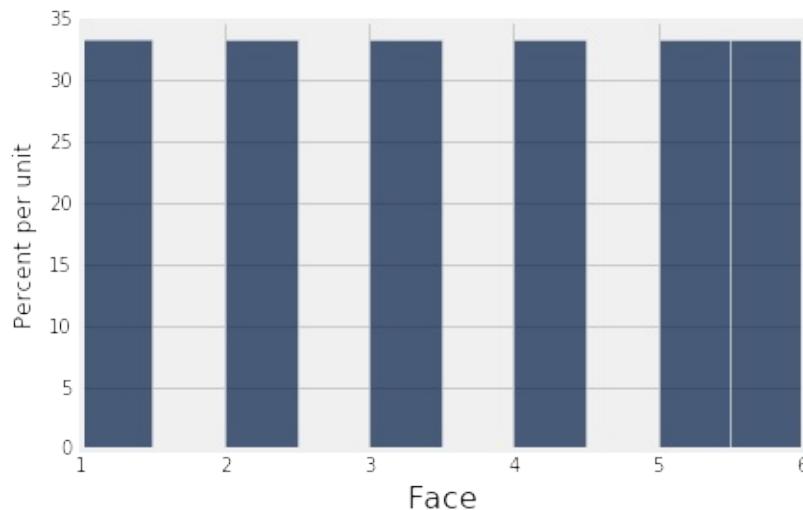
We will start with the table `die` which contains the numbers of spots on the faces of a die. All the numbers appear exactly once, as we are assuming that the die is fair.

```
die = Table().with_column('Face', [1, 2, 3, 4, 5, 6])
die
```

Face
1
2
3
4
5
6

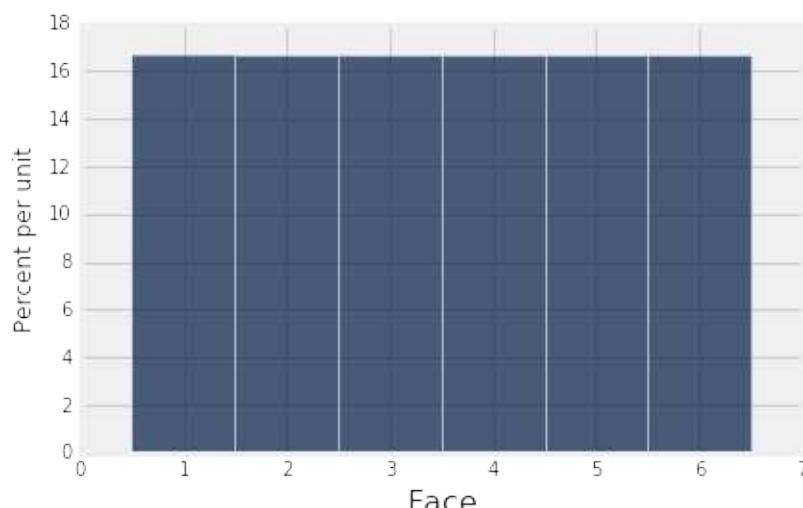
Drawing the histogram of this simple set of numbers yields an unsettling figure as `hist` makes a default choice of bins:

```
die.hist()
```



The numbers 1, 2, 3, 4, 5, 6 are integers, so the bins chosen by `hist` only have entries at the edges. In such a situation, it is a better idea to select bins so that they are centered on the integers. This is often true of histograms of data that are *discrete*, that is, variables whose successive values are separated by the same fixed amount. The real advantage of this method of bin selection will become more clear when we start imagining smooth curves drawn over histograms.

```
dice_bins = np.arange(0.5, 7, 1)
die.hist(bins=dice_bins)
```



Notice how each bin has width 1 and is centered on an integer. Notice also that because the width of each bin is 1, the height of each bar is  $\frac{1}{6}$ , the chance that the corresponding face appears.

The histogram shows the probability with which each face appears. It is called a *probability histogram* of the result of one roll of a die.

The histogram was drawn without rolling any dice or generating any random numbers. We will now use the computer to mimic actually rolling a die. The process of using a computer program to produce the results of a chance experiment is called *simulation*.

To roll the die, we will use a method called `sample`. This method returns a new table consisting of rows selected uniformly at random from a table. Its first argument is the number of rows to be returned. Its second argument is whether or not the sampling should be done with replacement.

The code below simulates 10 rolls of the die. As with all simulations of chance experiments, you should run the code several times and notice the variability in what is returned.

```
die.sample(10, with_replacement=True)
```

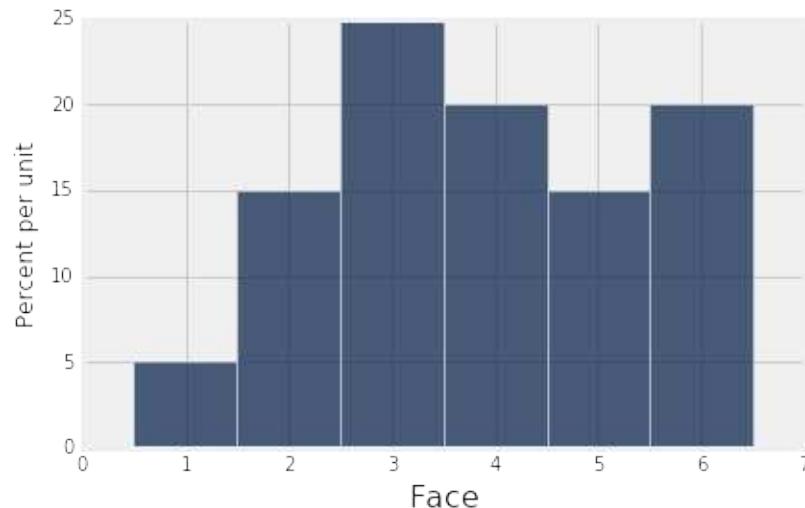
Face
5
4
2
3
6
5
1
1
6
2

We will now roll the die several times and draw the histogram of the observed results. The histogram of observed results is called an *empirical histogram*.

The results of rolling the die will all be integers in the range 1 through 6, so we will want to use the same bins as we used for the probability histogram. To avoid writing out the same bin argument every time we draw a histogram, let us define a function called `dice_hist` that will perform the task for us. The function will take one argument: the number `n` of dice to roll.

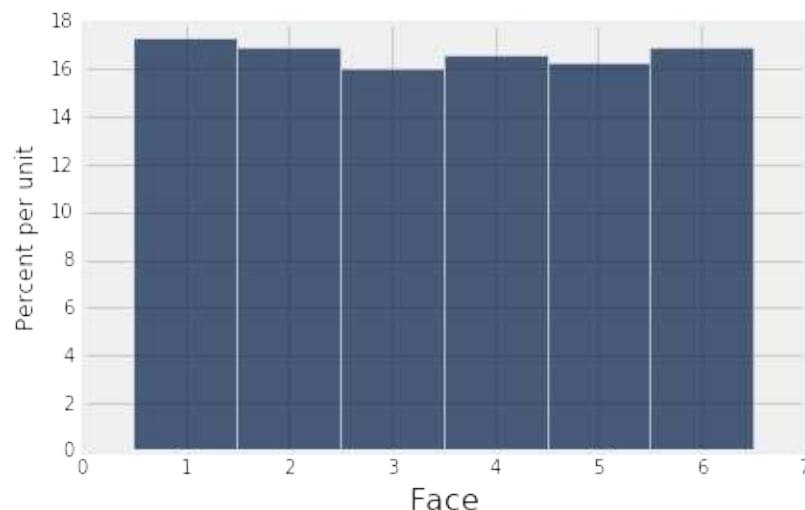
```
def dice_hist(n):
    rolls = die.sample(n, with_replacement=True)
    rolls.hist(bins=dice_bins)

dice_hist(20)
```



As we increase the number of rolls in the simulation, the proportions get closer to  $1/6$ .

```
dice_hist(10000)
```



The behavior we have observed is an instance of a general rule.

## The Law of Averages

If a chance experiment is repeated independently and under identical conditions, then, in the long run, the proportion of times that an event occurs gets closer and closer to the theoretical probability of the event.

For example, in the long run, the proportion of times the face with four spots appears gets closer and closer to 1/6.

Here "independently and under identical conditions" means that every repetition is performed in the same way regardless of the results of all the other repetitions.

## Convergence of empirical histograms

We have also observed that a random quantity (such as the number of spots on one roll of a die) is associated with two histograms:

- a probability histogram, that shows all the possible values of the quantity and all their chances
- an empirical histogram, created by simulating the random quantity repeatedly and drawing a histogram of the observed results

We have seen an example of the long-run behavior of empirical histograms:

As the number of repetitions increases, the empirical histogram of a random quantity looks more and more like the probability histogram.

## At the Roulette Table

Equipped with our new knowledge about the long-run behavior of chances, let us explore a gambling game. Betting on roulette is popular in gambling centers such as Las Vegas and Monte Carlo, and we will simulate one of the bets here.

The main randomizer in roulette in Nevada is a wheel that has 38 pockets on its rim. Two of the pockets are green, eighteen black, and eighteen red. The wheel is on a spindle, and there is a small ball on it. When the wheel is spun, the ball ricochets around and finally comes to rest in one of the pockets. That is declared to be the winning pocket.

You are allowed to bet on several pre-specified collections of pockets. If you bet on "red," you win if the ball comes to rest in one of the red pockets.

The bet even money. That is, it pays 1 to 1. To understand what that means, assume you are going to bet \$1 on "red." The first thing that happens, even before the wheel is spun, is that you have to hand over your \$1. If the ball lands in a green or black pocket, you never

see that dollar again. If the ball lands in a red pocket, you get your dollar back (to bring you back to even), plus another \$1 in winnings.

The table `wheel` represents the pockets of a Nevada roulette wheel. It has 38 rows labeled 1 through 38, one row per pocket.

```
pockets = np.arange(1, 39)
colors = ([['red', 'black'] * 5 + ['black', 'red']] * 4) * 2 + [['green']]
wheel = Table().with_columns([
    'pocket', pockets,
    'color', colors])
wheel
```



pocket	color
1	red
2	black
3	red
4	black
5	red
6	black
7	red
8	black
9	red
10	black
... (28 rows omitted)	

The function `bet_on_red` takes a numerical argument `x` and returns the net winnings on a \$1 bet on "red," provided `x` is the number of a pocket.

```
def bet_on_red(x):
    """The net winnings of betting on red for outcome x."""
    pockets = wheel.where('pocket', x)
    if pockets.column('color').item(0) == 'red':
        return 1
    else:
        return -1
```

```
bet_on_red(17)
```

```
-1
```

The function `spins` takes a numerical argument `n` and returns a new table consisting of `n` rows of `wheel` sampled at random with replacement. In other words, it simulates the results of `n` spins of the roulette wheel.

```
def spins(n):
    return wheel.sample(n, with_replacement=True)
```

We will create a table called `play` consisting of the results of 10 spins, and add a column that shows the net winnings on \$1 placed on "red." Recall that the `apply` method applies a function to each element in a column of a table.

```
outcomes = spins(500)
results = outcomes.with_column(
    'winnings', outcomes.apply(bet_on_red, 'pocket'))
results
```

pocket	color	winnings
13	black	-1
22	black	-1
38	green	-1
11	black	-1
29	black	-1
19	red	1
38	green	-1
37	green	-1
19	red	1
31	black	-1
... (490 rows omitted)		

And here is the net gain on all 500 bets:

```
sum(results.column('winnings'))
```

-32

Betting \$1 on red hundreds of times seems like a bad idea from a gambler's perspective. But from the casinos' perspective it is excellent. Casinos rely on large numbers of bets being placed. The payoff odds are set so that the more bets that are placed, the more money the casinos are likely to make, even though a few people are likely to go home with winnings.

### **Simple Random Sample – a Random Sample without Replacement**

A random sample without replacement is one in which elements are drawn from a list repeatedly, uniformly at random, at each stage deleting from the list the element that was drawn.

A random sample without replacement is also called a *simple random sample*. All elements of the population have the same chance of entering a simple random sample. All pairs have the same chance as each other, as do all triples, and so on.

The default action of `sample` is to draw without replacement. In card games, cards are almost always dealt without replacement. Let us use `sample` to deal cards from a deck.

A *standard deck* deck consists of 13 ranks of cards in each of four suits. The suits are called spades, clubs, diamonds, and hearts. The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Spades and clubs are black; diamonds and hearts are red. The Jacks, Queens, and Kings are called 'face cards.'

The table `deck` contains all 52 cards in a column labeled `cards`. The abbreviations are:

Ace: A      Jack: J      Queen: Q      King: K

Below, we use the `product` function to produce all possible pairs of a suit and a rank. The suits are represented by special characters used to print bridge and poker articles in newspapers.

```
from itertools import product

ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
suits = ['♠', '♥', '♦', '♣']
cards = product(ranks, suits)

deck = Table(['rank', 'suit']).with_rows(cards)
deck
```

A	♠
A	♥
A	♦
A	♣
2	♠
2	♥
2	♦
2	♣
3	♠
3	♥

rank	suit
A	♠
A	♥
A	♦
A	♣
2	♠
2	♥
2	♦
2	♣
3	♠
3	♥
4	♠
4	♥
4	♦
4	♣
5	♠
5	♥
5	♦
5	♣
6	♠
6	♥
6	♦
6	♣
7	♠
7	♥
7	♦
7	♣
8	♠
8	♥
8	♦
8	♣
9	♠
9	♥
9	♦
9	♣
10	♠
10	♥
10	♦
10	♣
J	♠
J	♥
J	♦
J	♣
Q	♠
Q	♥
Q	♦
Q	♣
K	♠
K	♥
K	♦
K	♣

... (42 rows omitted)

A *poker hand* is 5 cards dealt at random from the deck. The expression below deals a hand. Execute the cell a few times; how often are you dealt a pair?

```
deck.sample(5)
```

rank	suit
5	♥
4	♣
9	♥
3	♣
10	♣

A hand is a set of five cards, regardless of the order in which they appeared. For example, the hand  $9\spades 9\hearts Q\spades 7\spades 8\hearts$  is the same as the hand  $7\spades 9\spades 8\hearts 9\hearts Q\spades$ .

This fact can be used to show that simple random sampling can be thought of in two equivalent ways:

- drawing elements one by one at random without replacement
- randomly permuting (that is, shuffling) the whole list, and then pulling out a set of elements at the same time

# Iteration

[Interact](#)

## Repetition

It is often the case when programming that you will wish to repeat the same operation multiple times, perhaps with slightly different behavior each time. You could copy-paste the code 10 times, but that's tedious and prone to typos, and if you wanted to do it a thousand times (or a million times), forget it.

A better solution is to use a `for` statement to loop over the contents of a sequence. A `for` statement begins with the word `for`, followed by a name for the item in the sequence, followed by the word `in`, and ending with an expression that evaluates to a sequence. The indented body of the `for` statement is executed once *for each item in that sequence*.

```
for i in np.arange(5):
    print(i)
```

```
0
1
2
3
4
```

A typical use of a `for` statement is to build up a table by repeating a random computation many times and storing each result as a new row. The `append` method of a table takes a sequence and adds a new row. It's different from `with_row` because a new table is not created; instead, the original table is extended. The cell below draws 100 cards, but keeps only the aces.

```
aces = Table(['Rank', 'Suit'])
for i in np.arange(100):
    card = deck.row(np.random.randint(deck.num_rows))
    if card.item(0) == 'A':
        aces.append(card)

aces
```

Rank	Suit
A	♠
A	♣
A	♥
A	♦
A	♠
A	♦

This pattern can be used to track the results of repeated experiments. For example, perhaps we want to learn about the empirical properties of some randomly drawn poker hands. Below, we track whether the hand contains four-of-a-kind or five cards of the same suit (called a *flush*).

```
hands = Table(['Four-of-a-kind', 'Flush'])
for i in np.arange(10000):
    hand = deck.sample(5)
    four_of_a_kind = max(hand.group('rank').column('count')) == 4
    flush = max(hand.group('suit').column('count')) == 5
    hands.append([four_of_a_kind, flush])

hands
```

... (9990 rows omitted)

A `for` statement can also iterate over a sequence of labels. We can use this feature to summarize the results of our experiment. These are rare hands indeed!

```
for label in hands.labels:  
    success = np.count_nonzero(hands.column(label))  
    print('A', label, 'was drawn', success, 'of', hands.num_rows, '
```

A Four-of-a-kind was drawn 2 of 10000 times  
A Flush was drawn 22 of 10000 times

# Randomized response

Next, we'll look at a technique that was designed several decades ago to help conduct surveys of sensitive subjects. Researchers wanted to ask participants a few questions: Have you ever had an affair? Do you secretly think you are gay? Have you ever shoplifted? Have you ever sung a Justin Bieber song in the shower? They figured that some people might not respond honestly, because of the social stigma associated with answering "yes". So, they came up with a clever way to estimate the fraction of the population who are in the "yes" camp, without violating anyone's privacy.

Here's the idea. We'll instruct the respondent to roll a fair 6-sided die, secretly, where no one else can see it. If the die comes up 1, 2, 3, or 4, then respondent is supposed to answer honestly. If it comes up 5 or 6, the respondent is supposed to answer the *opposite* of what their true answer would be. But, they shouldn't reveal what came up on their die.

Notice how clever this is. Even if the person says "yes", that doesn't necessarily mean their true answer is "yes" -- they might very well have just rolled a 5 or 6. So the responses to the survey don't reveal any one individual's true answer. Yet, in aggregate, the responses give enough information that we can get a pretty good estimate of the fraction of people whose true answer is "yes".

Let's try a simulation, so we can see how this works. We'll write some code to perform this operation. First, a function to simulate rolling one die:

```
def roll_once():
    return np.random.randint(1, 7)
```

Now we'll use this to write a function to simulate how someone is supposed to respond to the survey. The argument to the function is their true answer (`True` or `False`); the function returns what they're supposed to tell the interview.

```
def respond(true_answer):
    if roll_once() >= 5:
        return not true_answer
    else:
        return true_answer
```

We can try it. Assume our true answer is 'no'; let's see what happens this time:

```
respond(False)
```

```
False
```

Of course, if you were to run it many times, you might get a different result each time. Below, we build a table of the responses for many responses when the true answer is always `False`.

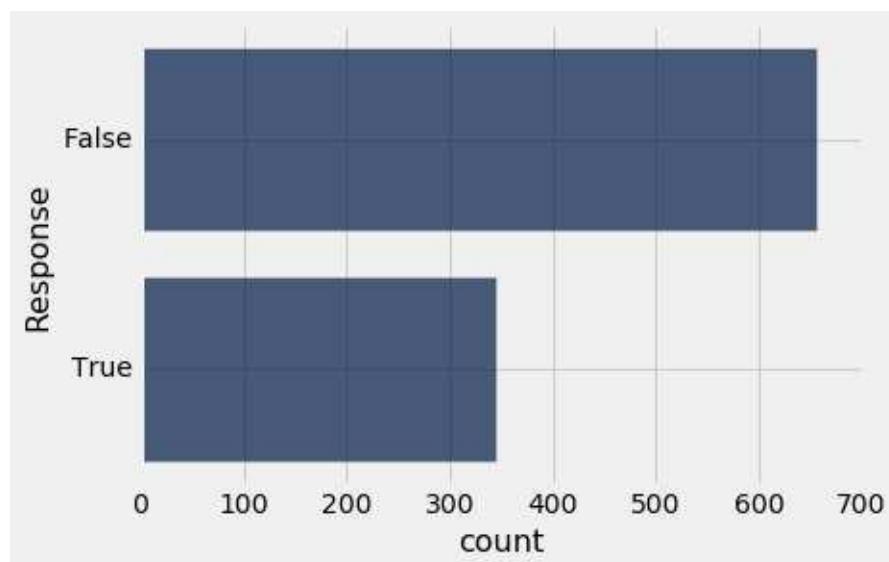
```
responses = Table(['Truth', 'Response'])
for i in np.arange(1000):
    responses.append([False, respond(False)])
responses
```

Truth	Response
False	False
False	True
False	False
False	True
False	False

... (990 rows omitted)

Let's build a bar chart and look at how many `True` and `False` responses we get.

```
responses.group('Response').barh('Response')
```



```
responses.where('Response', False).num_rows
```

656

```
responses.where('Response', True).num_rows
```

344

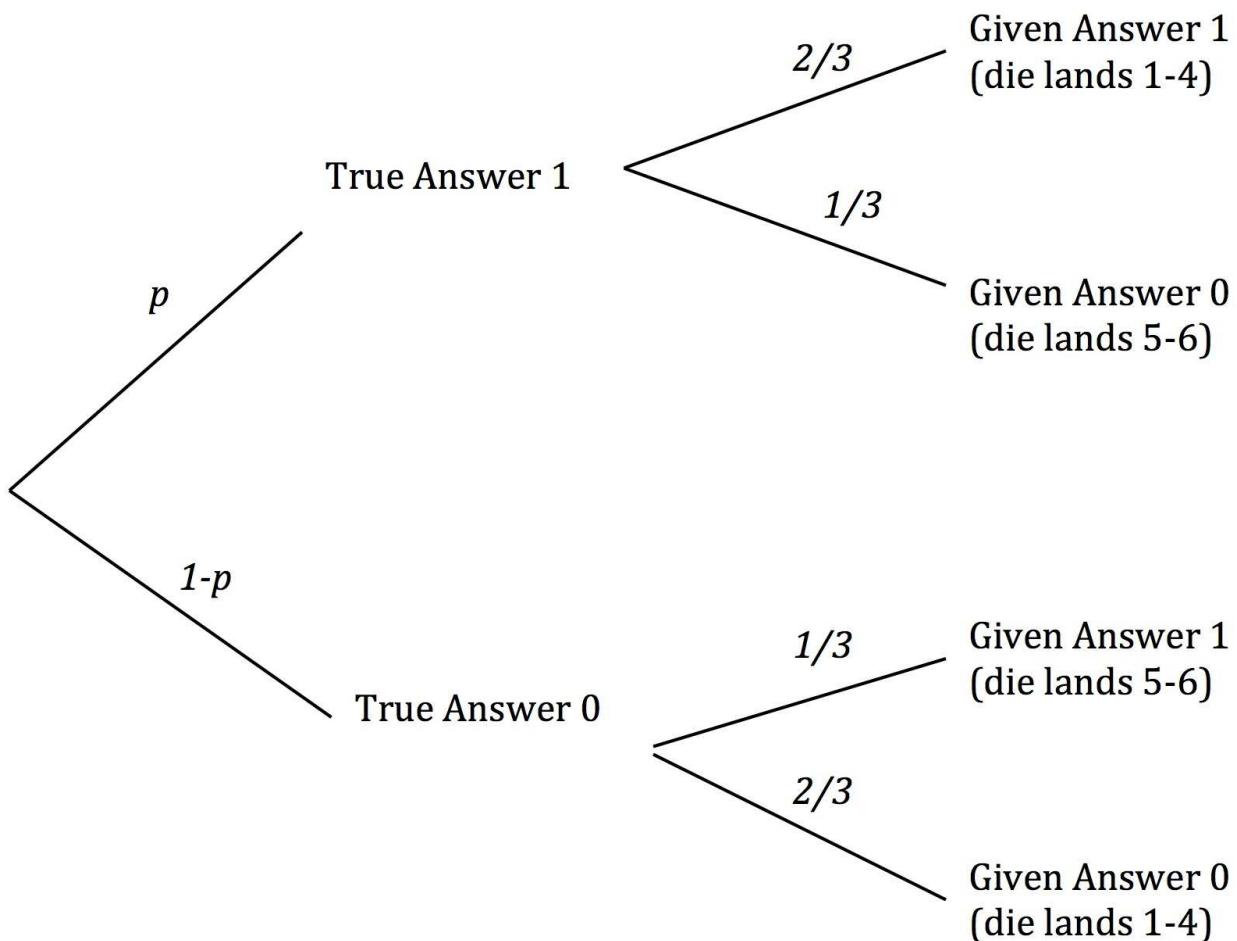
**Exercise for you:** If `N` out of 1000 responses are `True`, approximately what fraction of the population has truly sung a Justin Bieber song in the shower?

## Analysis¶

This method is called "randomized response". It is one way to poll people about sensitive subjects, while still protecting their privacy. You can see how it is a nice example of randomness at work.

It turns out that randomized response has beautiful generalizations. For instance, your Chrome web browser uses it to anonymously report feedback to Google, in a way that won't violate your privacy. That's all we'll say about it for this semester, but if you take an upper-division course, maybe you'll get to see some generalizations of this beautiful technique.

The steps in the randomized response survey can be visualized using a *tree diagram*. The diagram partitions all the survey respondents according to their true and answer and the answer that they eventually give. It also displays the proportions of respondents whose true answers are 1 ("True") and 0 ("False"), as well as the chances that determine the answers that they give. As in the code above, we have used  $p$  to denote the proportion whose true answer is 1.



The respondents who answer 1 split into two groups. The first group consists of the respondents whose true answer and given answers are both 1. If the number of respondents is large, the proportion in this group is likely to be about 2/3 of  $p$ . The second group consists of the respondents whose true answer is 0 and given answer is 1. This proportion in this group is likely to be about 1/3 of  $1-p$ .

We can observe  $p^*$ , the proportion of 1's among the given answers. Thus

$$p^* \approx \frac{2}{3} \times p + \frac{1}{3} \times (1 - p)$$

This allows us to solve for an approximate value of  $p$ :

$$p \approx 3p^* - 1$$

In this way we can use the observed proportion of 1's to "work backwards" and get an estimate of  $p$ , the proportion in which we are interested.

**Technical note.** It is worth noting the conditions under which this estimate is valid. The calculation of the proportions in the two groups whose given answer is 1 relies on *each of the groups* being large enough so that the Law of Averages allows us to make estimates about how their dice are going to land. This means that it is not only the total number of

respondents that has to be large – the number of respondents whose true answer is 1 has to be large, as does the number whose true answer is 0. For this to happen,  $p$  must be neither close to 0 nor close to 1. If the characteristic of interest is either extremely rare or extremely common in the population, the method of randomized response described in this example might not work well.

Let's try out this method on some real data. The chance of drawing a poker hand with no aces is

$$\frac{48}{52} \times \frac{47}{51} \times \frac{46}{50} \times \frac{45}{49} \times \frac{44}{48}$$

```
np.product(np.arange(48, 43, -1) / np.arange(52, 47, -1))
```

```
0.65884199833779666
```

It is quite embarrassing indeed to draw a hand with no aces. The table below contains one column for the truth of whether a hand has no aces and another for the randomized response.

```
ace_responses = Table(['Truth', 'Response'])
for i in np.arange(10000):
    hand = deck.sample(5)
    no_aces = hand.where('rank', 'A').num_rows == 0
    ace_responses.append([no_aces, respond(no_aces)])
ace_responses
```

Truth	Response
False	True
True	True
False	True
True	False
True	False
True	True
False	False
True	False
False	False
True	True

... (9990 rows omitted)

Using our derived formula, we can estimate what fraction of hands have no aces.

```
3 * np.count_nonzero(ace_responses.column('Response')) / 10000 - 1
```

```
0.6644000000000001
```

# Estimation

## Interact

In the previous section we saw that two kinds of histograms can be associated with a quantity generated by a random process.

- The **probability histogram** shows all the possible values of the quantity and the probabilities of all those values. The probabilities are calculated using math and the rules of chance.
- The **empirical histogram** is based on running the random process repeatedly and keeping track of the value of the quantity each time. It shows all the values of the quantity and the proportion of times each value was observed among all the repetitions.

We noted that by the Law of Averages, the empirical histogram is likely to resemble the probability histogram if the random process is repeated a large number of times.

This means that simulating random processes repeatedly is a way of approximating probability distributions *without figuring out the probabilities mathematically*. Thus computer simulations become a powerful tool in data science. They can help data scientists understand the properties of random quantities that would be complicated to analyze using math alone.

Here is an example of such a simulation.

## Estimating the number of enemy aircraft

In World War II, data analysts working for the Allies were tasked with estimating the number of German warplanes. The data included the serial numbers of the German planes that had been observed by Allied forces. These serial numbers gave the data analysts a way to come up with an answer.

To create an estimate of the total number of warplanes, the data analysts had to make some assumptions about the serial numbers. Here are two such assumptions, greatly simplified to make our calculations easier.

1. There are  $N$  planes, numbered  $1, 2, \dots, N$ .
2. The observed planes are drawn uniformly at random with replacement from the  $N$  planes.

The goal is to estimate the number  $N$ .

Suppose you observe some planes and note down their serial numbers. How might you use the data to guess the value of  $N$ ? A natural and straightforward method would be to simply use the **largest serial number observed**.

Let us see how well this method of estimation works. First, another simplification: Some historians now estimate that the German aircraft industry produced almost 100,000 warplanes of many different kinds. But here we will imagine just one kind. That makes Assumption 1 above easier to justify.

Suppose there are in fact  $N = 300$  planes of this kind, and that you observe 30 of them. We can construct a table called `serialno` that contains the serial numbers 1 through  $N$ . We can then sample 30 times with replacement (see Assumption 2) to get our sample of serial numbers. Our estimate is the maximum of these 30 numbers.

```
N = 300
serialno = Table().with_column('serial number', np.arange(1, N+1))
serialno
```

serial number
1
2
3
4
5
6
7
8
9
10
...

... (290 rows omitted)

```
max(serialno.sample(30, with_replacement=True).column(0))
```

296

As with all code involving random sampling, run it a few times to see the variation. You will observe that even with just 30 observations from among 300, the largest serial number is typically in the 250-300 range.

In principle, the largest serial number could be as small as 1, if you were unlucky enough to see Plane Number 1 all 30 times. And it could be as large as 300 if you observe Plane Number 300 at least once. But usually, it seems to be in the very high 200's. It appears that if you use the largest observed serial number as your estimate of the total, you will not be very far wrong.

Let us generate some data to see if we can confirm this. We will use *iteration* to repeat the sampling procedure numerous times, each time noting the largest serial number observed. These would be our estimates of  $N$  from all the numerous samples. We will then draw a histogram of all these estimates, and examine by how much they differ from  $N = 300$ .

In the code below, we will run 750 repetitions of the following process: Sample 30 times at random with replacement from 1 through 300 and note the largest number observed.

To do this, we will use a `for` loop. As you have seen before, we will start by setting up an empty table that will eventually hold all the estimates that are generated. As each estimate is the largest number in its sample, we will call this table `maxes`.

For each integer (called `i` in the code) in the range 0 through 749 (750 total), the `for` loop executes the code in the body of the loop. In this example, it generates a random sample of 30 serial numbers, computes the maximum value, and augments the rows of `maxes` with that value.

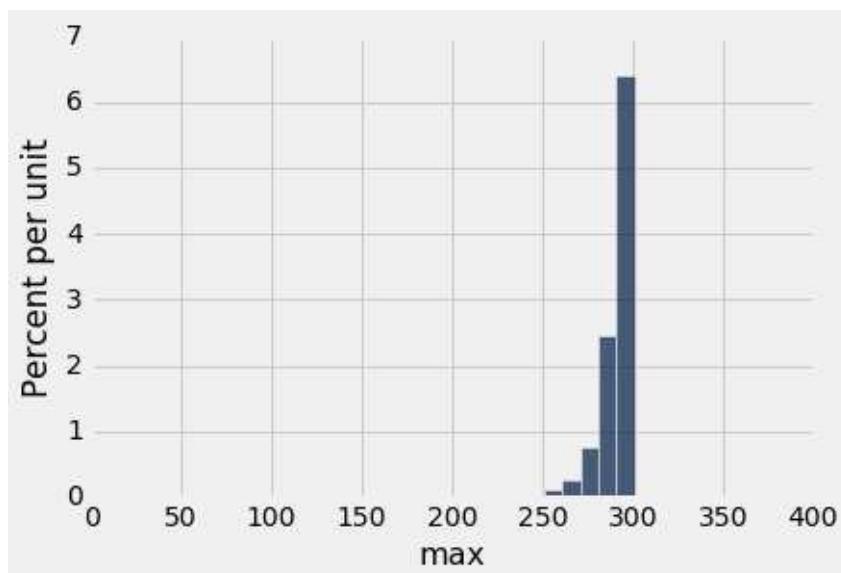
```
sample_size = 30
repetitions = 750

maxes = Table(['i', 'max'])
for i in np.arange(repetitions):
    sample = serialno.sample(sample_size, with_replacement=True)
    maxes.append([i, max(sample.column(0))])
maxes
```

i	max
0	300
1	300
2	291
3	300
4	297
5	295
6	283
7	296
8	299
9	273
... (740 rows omitted)	

Here is a histogram of the 750 estimates. As you can see, the estimates are all crowded up near 300, even though in theory they could be much smaller. The histogram indicates that as an estimate of the total number of planes, the largest serial number might be too low by about 10 to 25. But it is extremely unlikely to be underestimate the true number of planes by more than about 50.

```
every_ten = np.arange(1, N+100, 10)
maxes.hist('max', bins=every_ten)
```



## The empirical distribution of a statistic

In the example above, the largest serial number is called a *statistic* (singular!). A *statistic* is any number computed using the data in a sample.

The graph above is an empirical histogram. It displays the empirical or observed distribution of the statistic, based on 750 samples.

**The statistic could have been different.**

A fundamental consideration in using any statistic based on a random sample is that *the sample could have come out differently*, and therefore the statistic could have come out differently too.

**Just how different could the statistic have been?**

If you generate *all* of the possible samples, and compute the statistic for each of them, then you will have a clear picture of how different the statistic might have been. Indeed, you will have a complete enumeration of all the possible values of the statistic and all their probabilities. The resulting distribution is called the *probability distribution* of the statistic, and its histogram is the same as the probability histogram.

The probability distribution of a statistic is also called the *sampling distribution* of the statistic, because it is based on all of the possible samples.

The total number of possible samples is often very large. For example, the total number of possible sequences of 30 serial numbers that you could see if there were 300 aircraft is

300 \* \* 30

That's a lot of samples. Fortunately, we don't have to generate all of them. We know that the empirical histogram of the statistic, based on many but not all of the possible samples, is a good approximation to the probability histogram. The empirical distribution of a statistic gives a good idea of how different the statistic could be.

It is true that the probability distribution of a statistic contains more accurate information about the statistic than an empirical distribution does. But often, as in this example, the approximation provided by the empirical distribution is sufficient for data scientists to understand how much a statistic can vary. And empirical distributions are easier to compute. Therefore, data scientists often use empirical distributions instead of exact probability distributions when they are trying to understand random quantities.

### **Another Estimate.**

Here is an example to illustrate this point. Thus far, we have used the largest observed serial number as an estimate of the total number of planes. But there are other possible estimates, and we will now consider one of them.

The idea underlying this estimate is that the *average* of the observed serial numbers is likely be about halfway between 1 and  $N$ . Thus, if  $A$  is the average, then

$$A \approx \frac{N}{2} \text{ and so } N \approx 2A$$

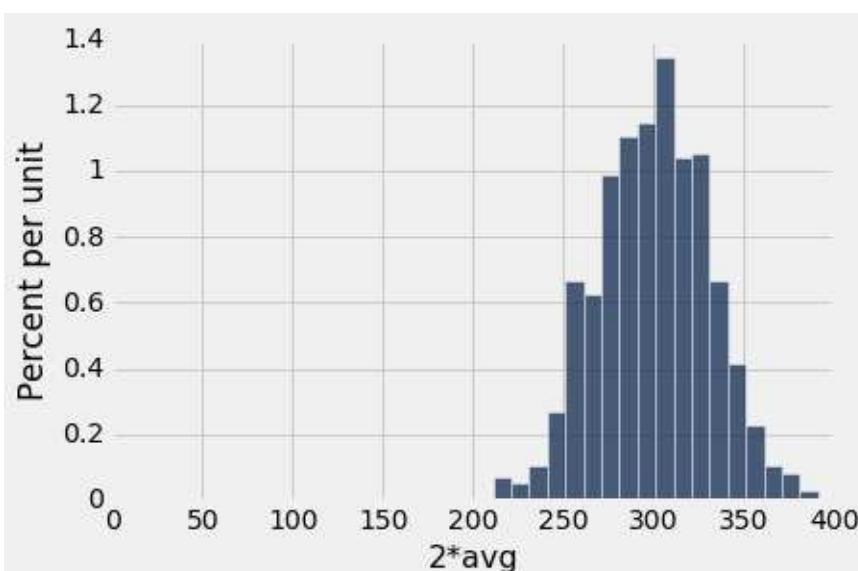
Thus a new statistic can be used to estimate the total number of planes: take the average of the observed serial numbers and double it.

How does this method of estimation compare with using the largest number observed? It is not easy to calculate the probability distribution of the new statistic. But as before, we can simulate it to get the probabilities approximately. Let's take a look at the empirical distribution based on repeated sampling. The number of repetitions is chosen to be the same as it was for the earlier estimate. This will allow us to compare the two empirical distributions.

```
new_est = Table(['i', '2*avg'])

for i in np.arange(repetitions):
    sample = serialno.sample(sample_size, with_replacement=True)
    new_est.append([i, 2 * np.average(sample.column(0))])

new_est.hist('2*avg', bins=every_ten)
```



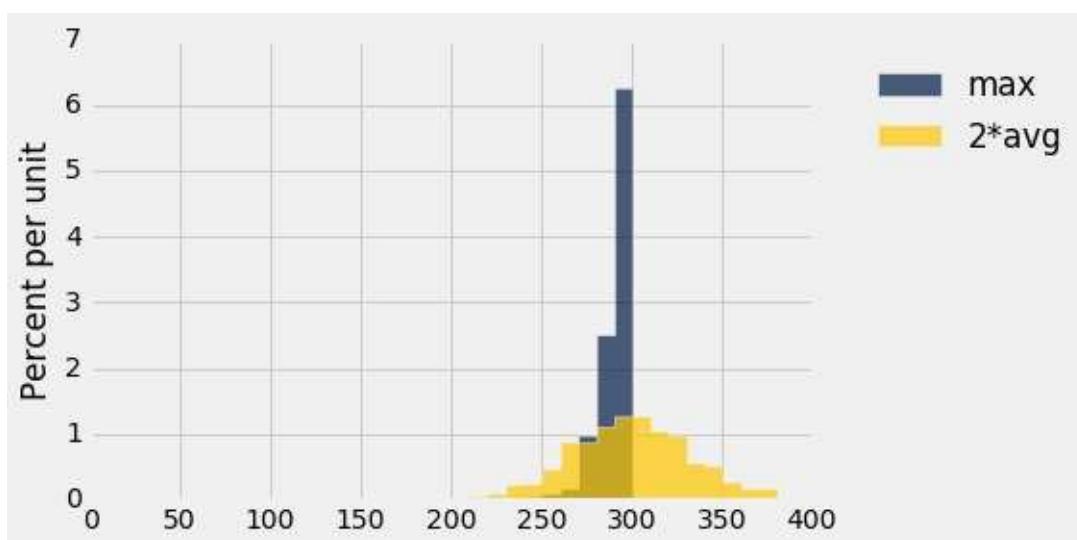
Notice that unlike the earlier estimate, this one can overestimate the number of planes. This will happen when the average of the observed serial numbers is closer to  $N$  than to 1.

For ease of comparison, let us run another set of 750 repetitions of the sampling process and calculate both estimates for each sample.

```
new_est = Table(['i', 'max', '2*avg'])

for i in np.arange(repetitions):
    sample = serialno.sample(sample_size, with_replacement=True)
    new_est.append([i, max(sample.column(0)), 2 * np.average(sample.column(0))])

new_est.select(['max', '2*avg']).hist(bins=every_ten)
```



You can see that the old method almost always underestimates; formally, we say that it is *biased*. But it has low variability, and is highly likely to be close to the true total number of planes.

The new method overestimates about as often as it underestimates, and thus is roughly *unbiased* on average in the long run. However, it is more variable than the old estimate, and thus is prone to larger absolute errors.

This is an instance of a *bias-variance tradeoff* that is not uncommon among competing estimates. Which estimate you decide to use will depend on the kinds of errors that matter the most to you. In the case of enemy warplanes, underestimating the total number might have grim consequences, in which case you might choose to use the more variable method that overestimates about half the time. On the other hand, if overestimation leads to needlessly high costs of guarding against planes that don't exist, you might be satisfied with the method that underestimates by a modest amount.

**Technical note:** In fact, twice the average is not unbiased. On average, it overestimates by exactly 1. For example, if  $n$  is 3, the average of draws from 1, 2, and 3 will be 2, and 2 times 2 is 4, which is one more than  $n$ . Twice the average minus 1 is an unbiased estimator of  $n$ .

# Center

## Interact

We have seen that studying the variability of an estimate involves questions such as:

- Where is the empirical histogram centered?
- About how far away from center could the values be?

In order to discuss estimates and their variability in greater detail, it will help to quantify some basic features of distributions.

## The mean of a list of numbers

In this course, we will use the words "average" and "mean" interchangeably.

**Definition:** The *average* or *mean* of a list of numbers is the sum of all the numbers in the list, divided by the number of entries in the list.

Here is the definition applied to a list of 20 numbers.

```
# The data: a list of numbers  
  
x_list = [4, 4, 4, 5, 5, 5, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3,  
  
# The average  
  
sum(x_list)/len(x_list)
```

3.15

The average of the list is 3.15.

You can think of taking the mean as an "equalizing" or "smoothing" operation. For example, imagine the entries in `x_list` being the amounts of money (in dollars) in the pockets of 20 different people. To get the mean, you first put all of the money into one big pot, and then divide it evenly among all 20 people. They had started out with different amounts of money in their pockets, but now each person has \$3.15, the average amount.

**Computation:** We could instead have found the mean by applying the function `np.mean` to the list `x_list`. Differences in rounding lead to answers that differ very slightly.

```
np.mean(x_list)
```

```
3.149999999999999
```

Or we could have placed the list in a table, and used the Table method `mean`:

```
x_table = Table().with_column('value', x_list)
x_table['value'].mean()
```

```
3.149999999999999
```

## The mean and the histogram

Another way to calculate the mean is to first arrange the list in a distribution table. The columns contain the distinct values in the list, the count of each value, and the corresponding proportion. The sum of the column `count` is 20, the number of entries in the list.

```
values = Table(['value', 'count']).with_rows([
    [2, 6],
    [3, 8],
    [4, 3],
    [5, 3]
])
counts = values.column('count')
x_dist = values.with_column('proportion', counts/sum(counts))

# the distribution table
x_dist
```

value	count	proportion
2	6	0.3
3	8	0.4
4	3	0.15
5	3	0.15

The sum of the numbers in the list can be calculated by multiplying the first two columns and adding up – four 2's, five 3's, and so on – and then dividing by 20:

```
sum(x_dist.column('value') * x_dist.column('count')) / sum(x_dist.c
```

```
3.149999999999999
```

This is the same as multiplying the first and third columns – the values and their proportions – and adding up:

```
sum(x_dist.column('value') * x_dist.column('proportion'))
```

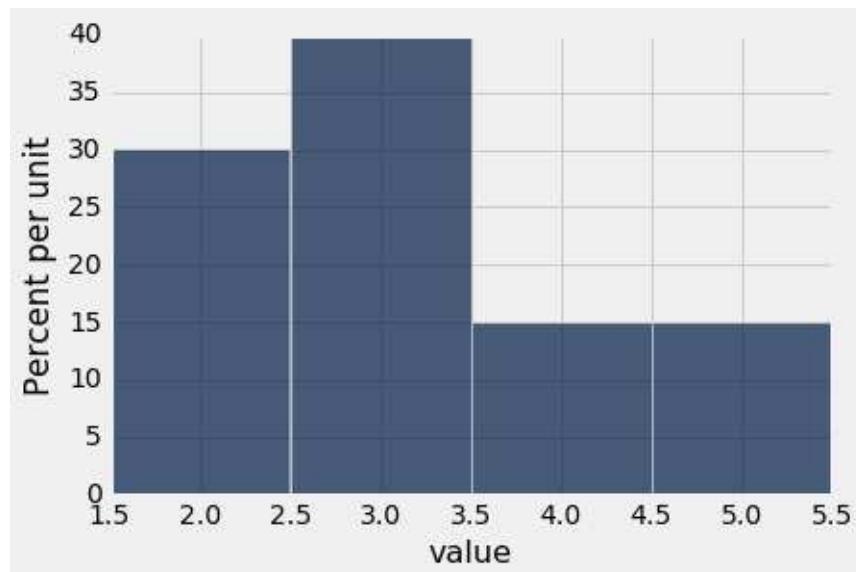
```
3.1500000000000004
```

We now have **another way to think about the mean**: The mean of a list is the *weighted* average of the *distinct* values, where the *weights are the proportions* in which those values appear.

Physically, this implies that **the average is the balance point of the histogram**. Here is the histogram of the distribution. Imagine that it is made out of cardboard and that you are trying to balance the cardboard figure at a point on the horizontal axis. The average of 3.15 is where the figure will balance.

To understand why that is, you will have to study some physics. Balance points, or centers of gravity, can be calculated in exactly the way we calculated the mean from the distribution table.

```
x_dist.select(['value', 'proportion']).hist(counts='value', bins=np
```



Notice that the mean of a list depends only on the distinct values and their proportions; you do not need to know how many entries there are in the list.

In other words, the average is a property of the histogram. If two lists have the same histogram, they will also have the same average.

## The mean and the median

If a student's score on a test is below average, does that imply that the student is in the bottom half of the class on that test?

Happily for the student, the answer is, "Not necessarily." The reason has to do with the relation between the average, which is the balance point of the histogram, and the median, which is the "half-way point" of the data.

Let's compare the balance point of the histogram to the point that has 50% of the area of the histogram on either side of it.

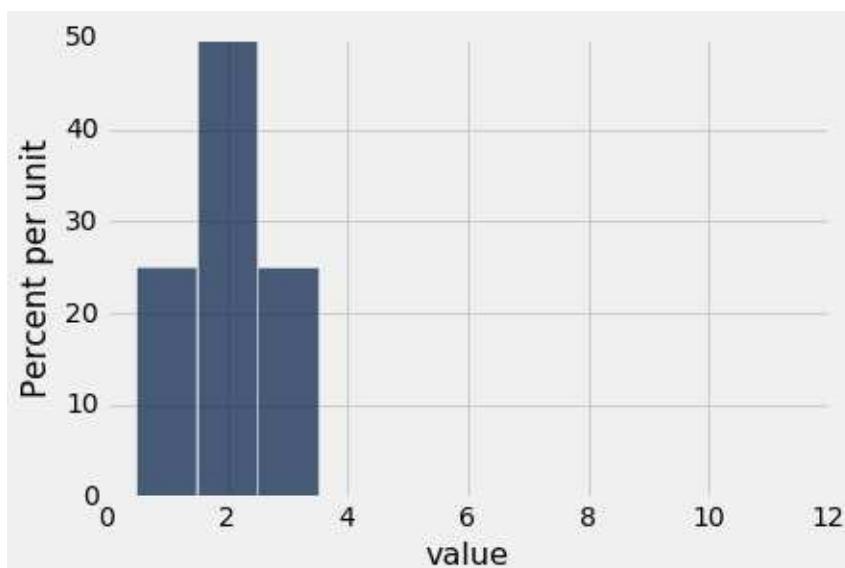
The relationship is easy to see in a simple example. Here is the list `1, 2, 2, 3`, represented as a distribution table called `sym` for "symmetric".

```
sym = Table().with_columns([
    'value', [1, 2, 2, 3],
    'dist', [0.25, 0.5, 0.25]
])
sym
```

value	dist
1	0.25
2	0.5
3	0.25

The histogram (or a calculation) shows that the average and median are both 2.

```
sym.hist(counts='value', bins=np.arange(0.5, 10.6, 1))
```



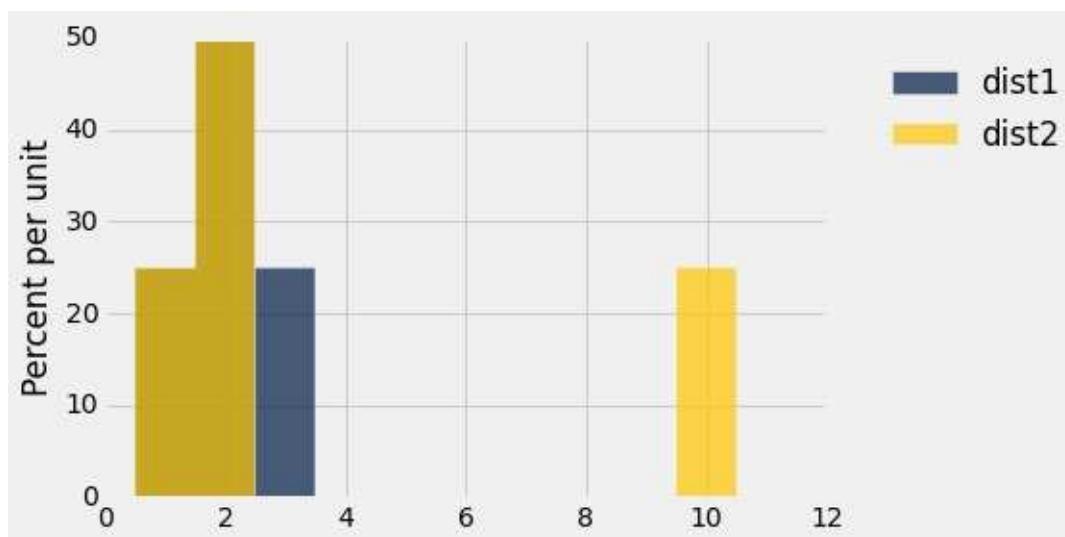
In general, for **symmetric distributions**, the mean and the median are equal.

What if the distribution is not symmetric? We will explore this by making a change in the histogram above: we will take the bar at the value 3 and slide it over to the value 10.

```
# Average versus median:  
# Balance point versus 50% point  
  
slide = Table().with_columns([  
    'value', [1,2,3,10],  
    'dist1',[0.25, 0.5, 0.25, 0],  
    'dist2',[0.25, 0.5, 0, 0.25]  
])  
slide
```

value	dist1	dist2
1	0.25	0.25
2	0.5	0.5
3	0.25	0
10	0	0.25

```
slide.hist(counts='value', bins=np.arange(0.5, 10.6, 1))
```



The blue histogram represents the original distribution; the gold histogram starts out the same as the blue at the left end, but its rightmost bar has slid over to the value 10. The brown part is where the two histograms overlap.

The median and mean of the blue distribution are both equal to 2. The median of the gold distribution is also equal to 2; the area on either side of 2 is still 50%, though the right half is distributed differently from the left.

But the mean of the gold distribution is not 2: the gold histogram would not balance at 2. The balance point has shifted to the right, to 3.75.

```
sum(slide['value']*slide['dist2'])
```

3.75

In the gold distribution, 3 out of 4 entries (75%) are below average. The student with a below average score can therefore take heart. He or she might be in the majority of the class.

**The mean, the median, and skewed distributions:** In general, if the histogram has a tail on one side (the formal term is "skewed"), then the mean is pulled away from the median in the direction of the tail.

**Example: Flight Delays.** The Bureau of Transportation Statistics provides data on departures and arrivals of flights in the United States. The table `ua` contains the departure delay times, in minutes, of about 37,500 United Airlines flights over the past few years. A negative delay means that the flight left earlier than scheduled.

```
ontime = Table.read_table('airline_ontime.csv')
ontime.show(3)
```

Carrier	Departure Delay	Arrival Delay
AA	-1	-1
AA	-1	-1
AA	-1	-1

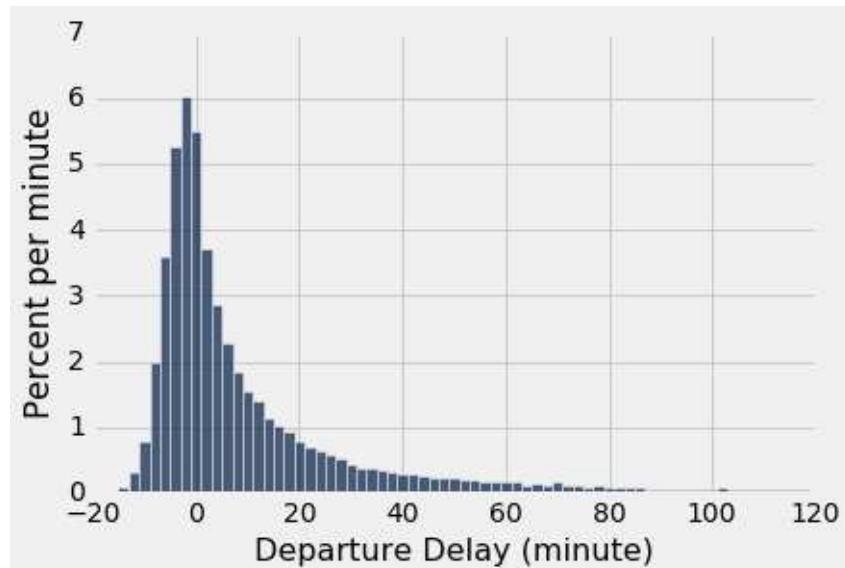
... (457010 rows omitted)

```
ua = ontime.where('Carrier', 'UA').select(1)
ua.show(3)
```

Departure Delay
-1
-1
-1

... (37360 rows omitted)

```
ua_bins = np.arange(-15, 121, 2)
ua.hist(bins=ua_bins, unit='minute')
```



The histogram of the delay times is right-skewed: it has a long right hand tail.

The mean gets pulled away from the median in the direction of the tail. So we expect the mean delay to be larger than the median, and that is indeed the case:

```
np.mean(ua['Departure Delay'])
```

```
13.885555228434548
```

```
np.median(ua['Departure Delay'])
```

```
2.0
```

# Spread

[Interact](#)

## Variability

### The Rough Size of Deviations from Average

As we have seen, inference based on test statistics must take into account the way in which the statistics vary across samples. It is therefore important to be able to quantify the variability in any list of numbers. One way is to create a measure of the difference between the values in the list and the average of the list.

We will start by defining such a measure in the context of a simple array of just four numbers .

```
numbers = np.array([1, 2, 2, 10])
numbers
```

```
array([ 1,  2,  2, 10])
```

The goal is to get a measure of roughly how far off the numbers in the list are from the average. To do this, we need the average, and all the deviations from the average. A "deviation from average" is just a value minus the average.

```
# Step 1. The average.

average = np.mean(numbers)
average
```

```
3.75
```

```
# Step 2. The deviations from average.

deviations = numbers - average
numbers_and_deviations = Table().with_columns([
    'value', numbers,
    'deviation from average', deviations
])
numbers_and_deviations
```

value	deviation from average
1	-2.75
2	-1.75
2	-1.75
10	6.25

Some of the deviations are negative; those correspond to values that are below average. Positive deviations correspond to above-average values.

To calculate roughly how big the deviations are, it is natural to compute the mean of the deviations. But something interesting happens when all the deviations are added together:

```
sum(deviations)
```

```
0.0
```

The positive deviations exactly cancel out the negative ones. This is true of all lists of numbers, no matter what the histogram of the list looks like: the sum of the deviations from average is zero.

Since the sum of the deviations is 0, the mean of the deviations will be 0 as well:

```
np.mean(deviations)
```

```
0.0
```

Because of this, the mean of the deviations is not a useful measure of the size of the deviations. What we really want to know is roughly how big the deviations are, regardless of whether they are positive or negative. So we need a way to eliminate the signs of the deviations.

There are two time-honored ways of losing signs: the absolute value, and the square. It turns out that taking the square constructs a measure with extremely powerful properties, some of which we will study in this course.

So let us eliminate the signs by squaring all the deviations. Then we will take the mean of the squares:

```
# Step 3. The squared deviations from average

squared_deviations = deviations ** 2
numbers_and_deviations.with_column('squared deviations from average')
```

value	deviation from average	squared deviations from average
1	-2.75	7.5625
2	-1.75	3.0625
2	-1.75	3.0625
10	6.25	39.0625

```
# Variance = the mean squared deviation from average

variance = np.mean(squared_deviations)
variance
```

13.1875

**Variance:** The mean squared deviation is called the *variance* of a sequence of values. While it does give us an idea of spread, it is not on the same scale as the original variable and its units are the square of the original. This makes interpretation very difficult. So we return to the original scale by taking the positive square root of the variance:

```
# Standard Deviation:      root mean squared deviation from average
# Steps of calculation:   5      4      3      2      1

sd = variance ** 0.5
sd
```

3.6314597615834874

## Standard Deviation¶

The quantity that we have just computed is called the *standard deviation* of the list, and is abbreviated as SD. It measures roughly how far the numbers on the list are from their average.

**Definition.** The SD of a list is defined as the *root mean square of deviations from average*. That's a mouthful. But read it from right to left and you have the sequence of steps in the calculation.

**Computation.** The `numpy` function `np.std` computes the SD of values in an array:

`np.std(numbers)`

3.6314597615834874

## Working with the SD¶

Let us now examine the SD in the context of a more interesting dataset. The table `nba` contains data on the players in the National Basketball Association (NBA) in 2013. For each player, the table records the position at which the player usually played, his height in inches, weight in pounds, and age in years.

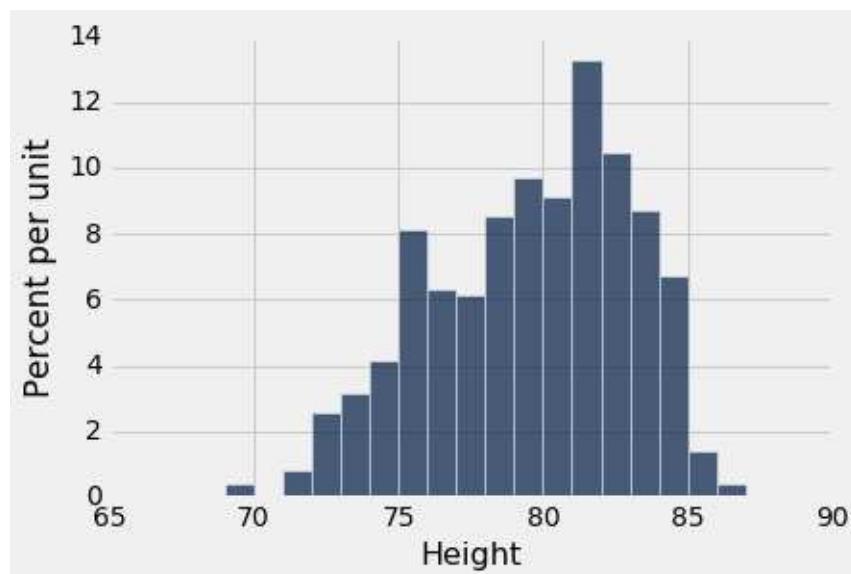
```
nba = Table.read_table("nba2013.csv")
nba
```

Name	Position	Height	Weight	Age in 2013
DeQuan Jones	Guard	80	221	23
Darius Miller	Guard	80	235	23
Trevor Ariza	Guard	80	210	28
James Jones	Guard	80	215	32
Wesley Johnson	Guard	79	215	26
Klay Thompson	Guard	79	205	23
Thabo Sefolosha	Guard	79	215	29
Chase Budinger	Guard	79	218	25
Kevin Martin	Guard	79	185	30
Evan Fournier	Guard	79	206	20

... (495 rows omitted)

Here is a histogram of the players' heights.

```
nba.select('Height').hist(bins=np.arange(68, 88, 1))
```



It is no surprise that NBA players are tall! Their average height is just over 79 inches (6'7"), about 10 inches taller than the average height of men in the United States.

```
mean_height = np.mean(nba.column('Height'))
mean_height
```

```
79.065346534653472
```

About far off are the players' heights from the average? This is measured by the SD of the heights, which is about 3.45 inches.

```
sd_height = np.std(nba.column('Height'))  
sd_height
```

```
3.4505971830275546
```

The towering center Hasheem Thabeet of the Oklahoma City Thunder was the tallest player at a height of 87 inches.

```
nba.sort('Height', descending=True).show(3)
```

Name	Position	Height	Weight	Age in 2013
Hasheem Thabeet	Center	87	263	26
Roy Hibbert	Center	86	278	26
Tyson Chandler	Center	85	235	30

... (502 rows omitted)

Thabeet was about 8 inches above the average height.

```
87 - mean_height
```

```
7.9346534653465284
```

That's a deviation from average, and it is about 2.3 times the standard deviation:

```
(87 - mean_height)/sd_height
```

```
2.2995015194397923
```

In other words, the height of the tallest player was about 2.3 SDs above average.

At 69 inches tall, Isaiah Thomas was one of the two shortest players in the NBA. His height was about 2.9 SDs below average.

```
nba.sort('Height').show(3)
```

Name	Position	Height	Weight	Age in 2013
Isaiah Thomas	Guard	69	185	24
Nate Robinson	Guard	69	180	29
John Lucas III	Guard	71	157	30

... (502 rows omitted)

```
(69 - mean_height)/sd_height
```

```
-2.9169868288775844
```

What we have observed is that the tallest and shortest players were both just a few SDs away from the average height. This is an example of why the SD is a useful measure of spread. No matter what the shape of the histogram, the average and the SD together tell you a lot about the data.

## First main reason for measuring spread by the SD

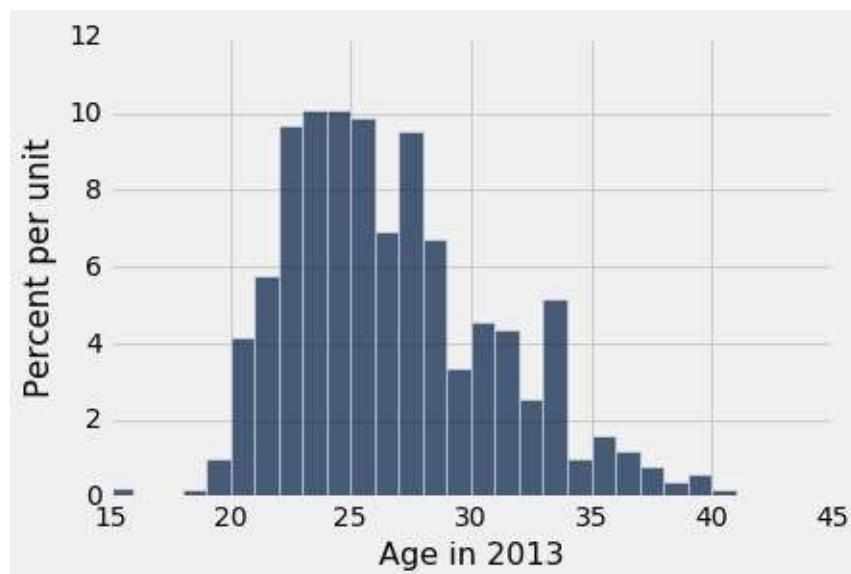
**Informal statement.** For all lists, the bulk of the entries are within the range "average  $\pm$  a few SDs".

Try to resist the desire to know exactly what fuzzy words like "bulk" and "few" mean. We will make them precise later in this section. For now, let us examine the statement in the context of some examples.

We have already seen that *all* of the heights of the NBA players were in the range "average  $\pm$  3 SDs".

What about the ages? Here is a histogram of the distribution. Yes, there was a 15-year-old player in the NBA in 2013.

```
nba.select('Age in 2013').hist(bins=np.arange(15, 45, 1))
```



Juwan Howard was the oldest player, at 40. His age was about 3.2 SDs above average.

```
nba.sort('Age in 2013', descending=True).show(3)
```

Name	Position	Height	Weight	Age in 2013
Juwan Howard	Forward	81	250	40
Marcus Camby	Center	83	235	39
Derek Fisher	Guard	73	210	39

... (502 rows omitted)

```
ages = nba['Age in 2013']
(40 - np.mean(ages))/np.std(ages)
```

3.1958482778922357

The youngest was 15-year-old Jarvis Varnado, who won the NBA Championship that year with the Miami Heat. His age was about 2.6 SDs below average.

```
nba.sort('Age in 2013').show(3)
```

Name	Position	Height	Weight	Age in 2013
Jarvis Varnado	Forward	81	230	15
Giannis Antetokounmpo	Forward	81	205	18
Sergey Karasev	Guard	79	197	19

... (502 rows omitted)

```
(15 - np.mean(ages))/np.std(ages)
```

```
-2.5895811038670811
```

What we have observed for the heights and ages is true in great generality. For all lists, the bulk of the entries are no more than 2 or 3 SDs away from the average.

These rough statements about deviations from average are made precise by the following result.

## Chebychev's inequality

For all lists, and all positive numbers  $z$ , the proportion of entries that are in the range "average  $\pm z$  SDs" is at least  $1 - \frac{1}{z^2}$ .

What makes this result powerful is that it is true for all lists – all distributions, no matter how irregular.

Specifically, Chebychev's inequality says that for every list:

- the proportion in the range "average  $\pm 2$  SDs" is at least  $1 - 1/4 = 0.75$
- the proportion in the range "average  $\pm 3$  SDs" is at least  $1 - 1/9 \approx 0.89$
- the proportion in the range "average  $\pm 4.5$  SDs" is at least  $1 - 1/4.5^2 \approx 0.95$

Chebychev's Inequality gives a lower bound, not an exact answer or an approximation. For example, the percent of entries in the range "average  $\pm 2$  SDs" might be quite a bit larger than 75%. But it cannot be smaller.

## Standard units

In the calculations above,  $z$  measures *standard units*, the number of standard deviations above average.

Some values of standard units are negative, corresponding to original values that are below average. Other values of standard units are positive. But no matter what the distribution of the list looks like, Chebychev's Inequality says that standard units will typically be in the (-5, 5) range.

To convert a value to standard units, first find how far it is from average, and then compare that deviation with the standard deviation.

$$z = \frac{\text{value} - \text{average}}{\text{SD}}$$

As we will see, standard units are frequently used in data analysis. So it is useful to define a function that converts an array of numbers to standard units.

```
def standard_units(any_numbers):
    "Convert any array of numbers to standard units."
    return (any_numbers - np.mean(any_numbers))/np.std(any_numbers)
```

As we saw in an earlier section, the table `ua` contains a column `Departure Delay` consisting of the departure delay times, in minutes, of over 37,000 United Airlines flights. We can create a new column called `Delay_su` by applying the function `standard_units` to the column of delay times. This allows us to see all the delay times in minutes as well as their corresponding values in standard units.

```
ua.append_column('z', standard_units(ua.column('Departure Delay')))
ua
```

... (37353 rows omitted)

Something rather alarming happens when we sort the delay times from highest to lowest. The standard units are extremely high!

```
ua.sort(0, descending=True)
```

Departure Delay	z
886	22.9631
739	19.0925
678	17.4864
595	15.301
566	14.5374
558	14.3267
543	13.9318
541	13.8791
537	13.7738
513	13.1419

... (37353 rows omitted)

What this shows is that it is possible for data to be many SDs above average (and for flights to be delayed by almost 15 hours). The highest value is more than 22 in standard units.

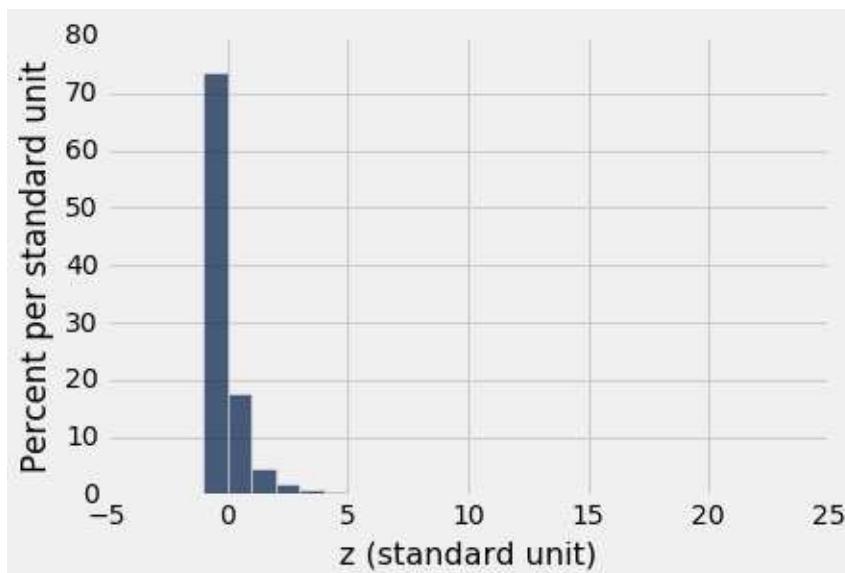
However, the proportion of these extreme values is tiny, and the bounds in Chebychev's inequality still hold true. For example, let us calculate the percent of delay times that are in the range "average  $\pm 2$  SDs". This is the same as the percent of times for which the standard units are in the range (-2, 2). That is just about 96%, as computed below, consistent with Chebychev's bound of "at least 75%".

```
within_2_sd = np.logical_and(ua.column('z') > -2, ua.column('z') <
ua.where(within_2_sd).num_rows/ua.num_rows
```

```
0.960522441988063
```

The histogram of delay times is shown below, with the horizontal axis in standard units. By the table above, the right hand tail continues all the way out to 22 standard units (886 minutes). The area of the histogram outside the range  $z = -2$  to  $z = 2$  is about 4%, put together in tiny little bits that are mostly invisible in a histogram.

```
ua.hist('z', bins=np.arange(-5, 23, 1), unit='standard unit')
```



# The Normal Distribution

## Interact

We know that the mean is the balance point of the histogram. Unlike the mean, the SD is usually not easy to identify by looking at the histogram.

However, there is one shape of distribution for which the SD is almost as clearly identifiable as the mean. This section examines that shape, as it appears frequently in probability histograms and also in many histograms of data.

## The SD and the bell curve

The table `births` contains data on over 1,000 babies born at a hospital in the Bay Area. The columns contain several variables: the baby's birth weight in ounces; the number of gestational days; the mother's age in years, height in inches, and pregnancy weight in pounds; and a record of whether she smoked or not.

```
births = Table.read_table('baby.csv')
births
```

Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
120	284	27	62	100	False
113	282	33	64	135	False
128	279	28	64	115	True
108	282	23	67	125	True
136	286	25	62	93	False
138	244	33	62	178	False
132	245	23	65	140	False
120	289	25	62	125	False
143	299	30	66	136	True
140	351	27	68	120	False

... (1164 rows omitted)

The mothers' heights have a mean of 64 inches and an SD of 2.5 inches. Unlike the heights of the basketball players, the mothers' heights are distributed fairly symmetrically about the mean in a bell-shaped curve.

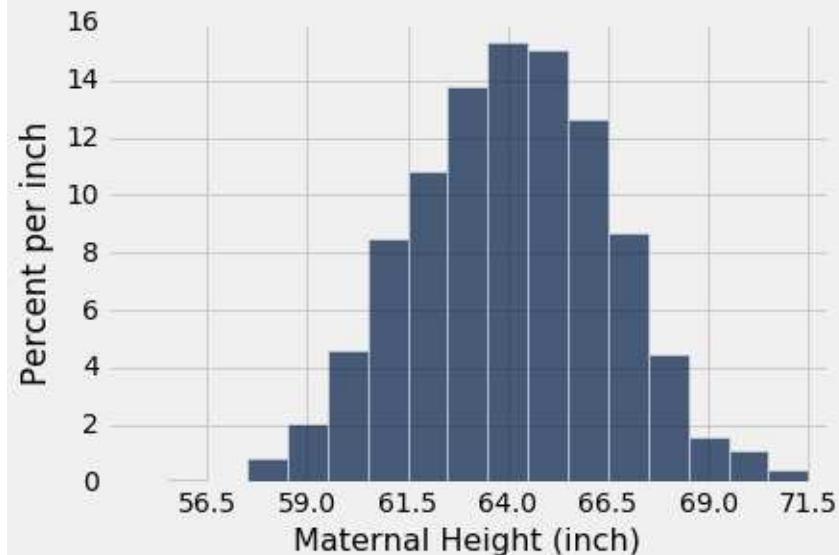
```
heights = births.column('Maternal Height')
mean_height = np.round(np.mean(heights), 1)
mean_height
```

```
64.0
```

```
sd_height = np.round(np.std(heights), 1)
sd_height
```

```
2.5
```

```
births.hist('Maternal Height', bins=np.arange(55.5, 72.5, 1), unit=
positions = np.arange(-3, 3.1, 1)*sd_height + mean_height
_ = plots.xticks(positions)
```



The last two lines of code in the cell above change the labeling of the horizontal axis. Now, the labels correspond to "average  $\pm z$  SDs" for  $z = 0, \pm 1, \pm 2$ , and  $\pm 3$ . Because of the shape of the distribution, the "center" has an unambiguous meaning and is clearly visible at 64.

To see how the SD is related to the curve, start at the top of the curve and look towards the right. Notice that there is a place where the curve changes from looking like an "upside-down cup" to a "right-way-up cup"; formally, the curve has a point of inflection. That point is one SD above average. It is the point  $z = 1$ , which is "average plus 1 SD" = 66.5 inches.

Symmetrically on the left-hand side of the mean, the point of inflection is at  $z = -1$ , that is, "average minus 1 SD" = 61.5 inches.

**How to spot the SD on a bell-shaped curve:** For bell-shaped distributions, the SD is the distance between the mean and the points of inflection on either side.

## Bell-shaped probability histograms

The reason for studying bell-shaped curves is not just that they appear as some distributions of data, but that they appear as approximations to probability histograms of sample averages.

We have seen this phenomenon already, in the example about estimating the total number of aircraft. Recall that we simulated random samples of size 30 drawn with replacement from the serial numbers 1, 2, 3, ..., 300, and computed two different quantities: the largest of the 30 numbers observed, and twice the average of the 30 numbers. Here are the generated histograms.

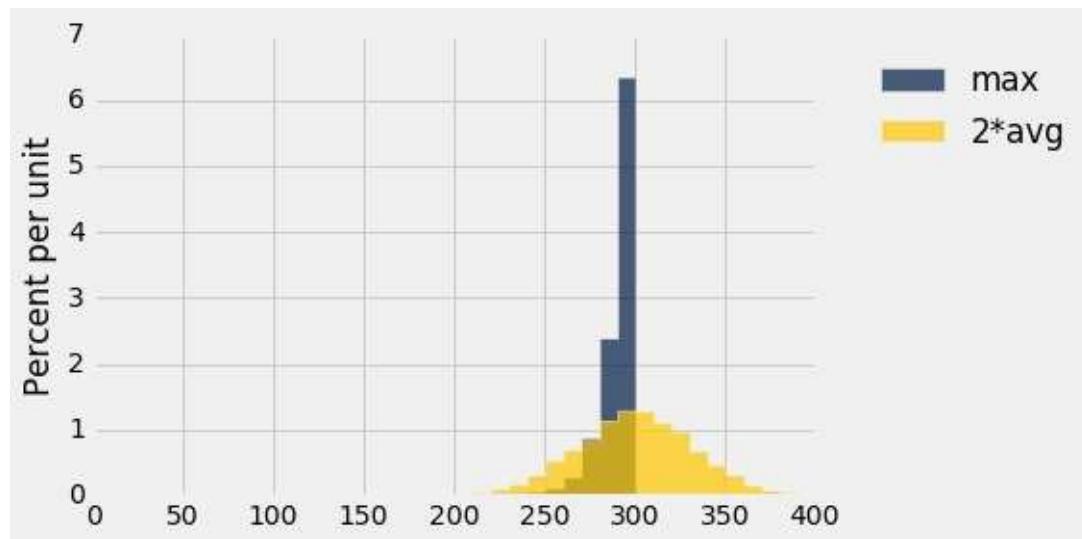
```
N = 300
sample_size = 30
repetitions = 5000

serialno = Table().with_column('serial number', np.arange(1, N+1))

new_est = Table(['i', 'max', '2*avg'])

for i in np.arange(repetitions):
    sample = serialno.sample(sample_size, with_replacement=True)
    new_est.append([i, max(sample.column(0)), 2 * np.average(sample.column(0))])

every_ten = np.arange(1, N+100, 10)
new_est.select(['max', '2*avg']).hist(bins=every_ten)
```



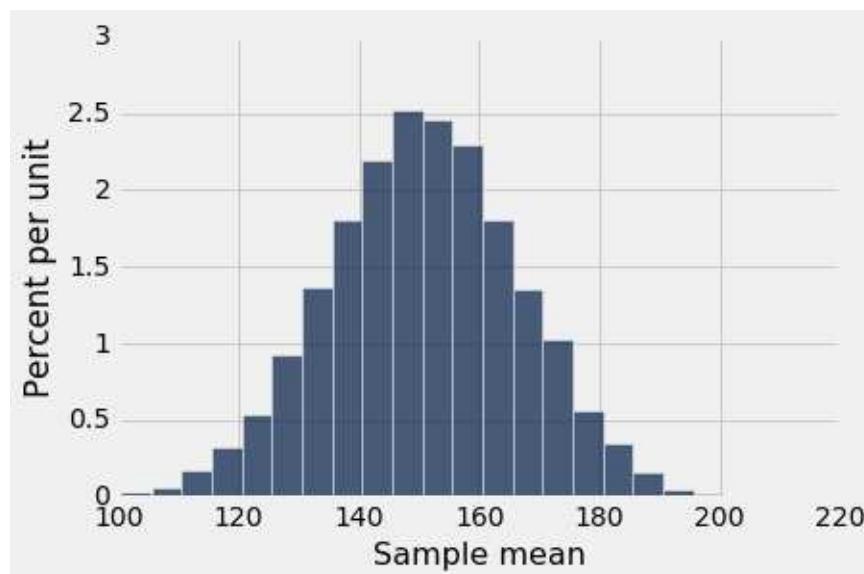
We observed that the distribution of "twice the sample average" is roughly symmetric; now let's also observe that it is roughly bell-shaped.

If the distribution of "twice the average" is bell-shaped, so is the distribution of the average. Here is the distribution of the average of a sample of size 30 drawn from the numbers 1, 2, 3, ..., 300. The number of replications of the sampling procedure is large, so it is safe to assume that the empirical histogram resembles the probability histogram of the sample average.

```
sample_means = Table(['Sample mean'])

for i in np.arange(repetitions):
    sample = serialno.sample(sample_size, with_replacement=True)
    sample_means.append([np.average(sample.column(0))])

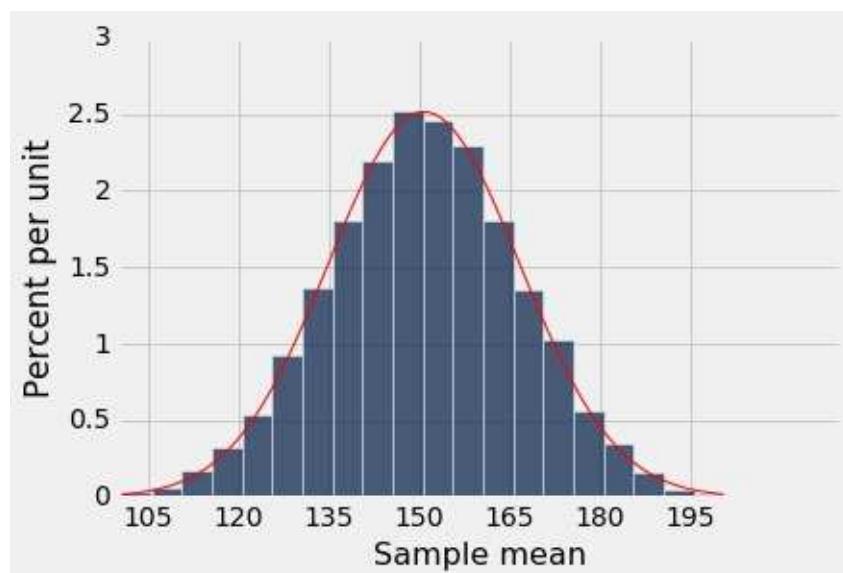
every_five = np.arange(100.5, 200.6, 5)
sample_means.hist(bins=every_five)
```



We can draw this curve with a red bell-shaped curve superposed. This curve is generated from the mean and standard deviation of the sample means. Don't worry about the code that draws the curve; just focus on the diagram itself.

```
# Import a module of standard statistical functions
from scipy import stats

sample_means.hist(bins=every_five)
means = sample_means['Sample mean']
m = np.mean(means)
s = np.std(means)
x = np.arange(100.5, 200.6, 1)
plots.plot(x, stats.norm.pdf(x, m, s), color='r', lw=1)
positions = np.arange(-3, 3.1, 1)*15 + 150
_ = plots.xticks(positions, [int(y) for y in positions])
```



Since all the numbers drawn are between 1 and 300, we expect the sample mean to be around 150, which is where the histogram is centered.

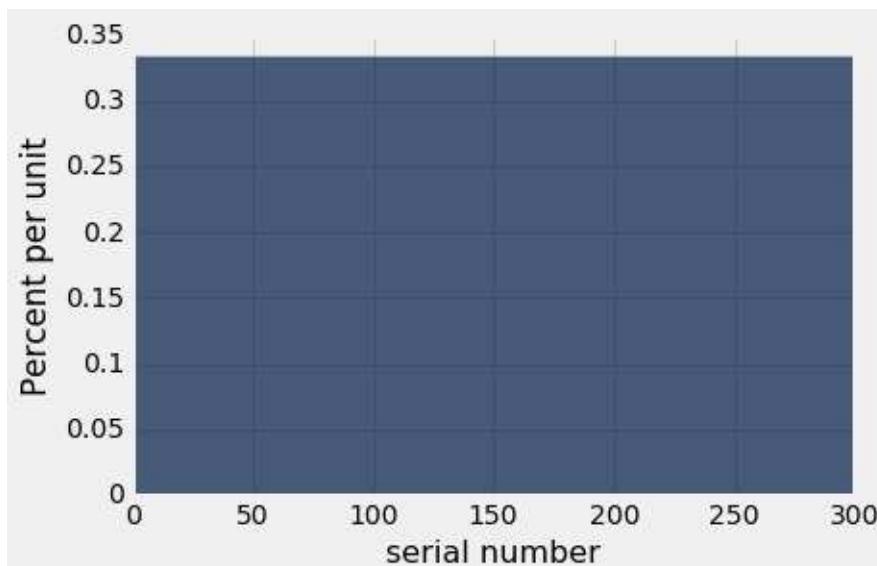
Can you guess what the SD is? Run your eye along the curve till you feel you are close to the point of inflection to the right of center. If you guessed that the point is at about 165, you're correct – it's between 165 and 166 and symmetrically between 134 and 135 to the left of center.

## The probability histogram of a random sample average [T](#)

What is notable about this is not just that we can spot the SD, but that the probability distribution of the statistic is bell-shaped in the first place. There is a remarkable piece of mathematics called the **Central Limit Theorem** that shows that the distribution of the average of a large random sample will be roughly normal, *no matter what the distribution of the population from which the sample is drawn*.

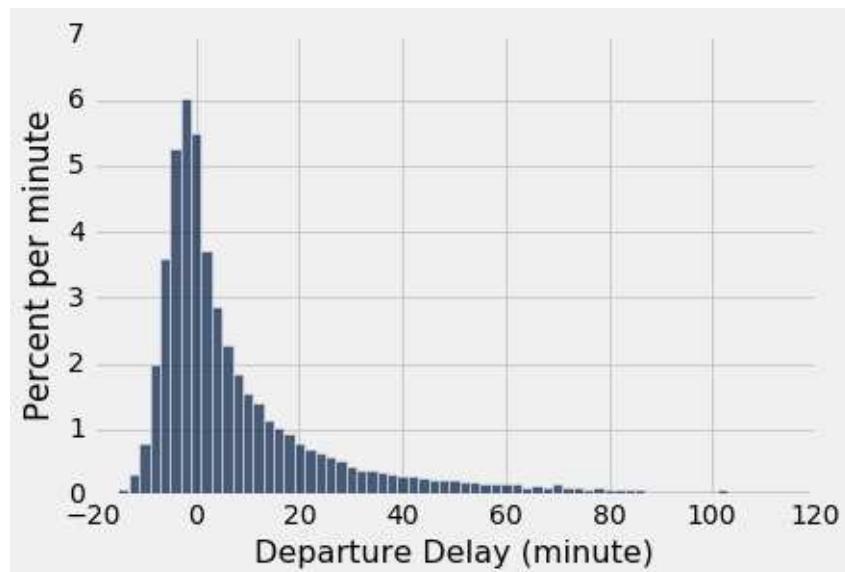
In our example about serial numbers, each of the 300 serial numbers is equally likely to be drawn each time, so the distribution of the population is uniform and the probability histogram looks like a brick.

```
serialno.hist(bins=np.arange(0.5, 300.5, 1))
```



Let's test out the Central Limit Theorem by drawing samples from quite a different distribution. Here is the histogram of over 37,000 flight delay times in the table `ua`. It looks nothing like the uniform distribution above.

```
plotrange = np.arange(-15, 121, 2)
ua.hist(bins=plotrange, unit='minute')
```



We will now look at the probability distribution of the average of 625 flight times drawn at random from this population. The draws will be made with replacement, as they were in the example about serial numbers.

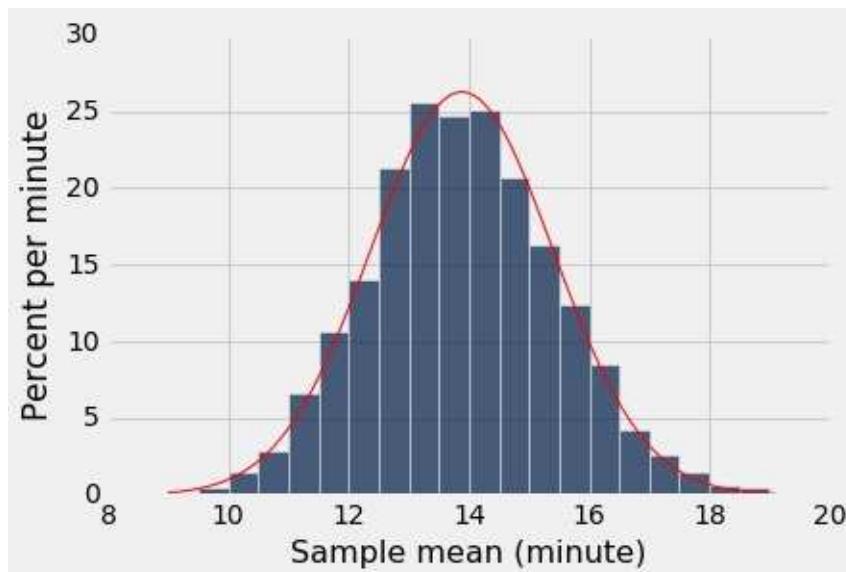
```
sample_size = 625

sample_means = Table(['Sample mean'])

for i in np.arange(repetitions):
    sample = ua.sample(sample_size, with_replacement=True)
    sample_means.append([np.average(sample.column(0))])

sample_means.hist(bins=np.arange(9, 19.1, 0.5), unit='minute')

# Draw a red bell-shaped curve from the mean and SD of the sample means
means = sample_means['Sample mean']
m = np.mean(means)
s = np.std(means)
x = np.arange(9, 19.1, 0.1)
_ = plots.plot(x, stats.norm.pdf(x, np.mean(ua[0]), np.std(ua[0])/2))
```



The probabilities for the sample average are roughly bell-shaped, consistent with the Central Limit Theorem.

The average delay time of all the flights in the population is about 14 minutes, which is where the histogram is centered. To spot the SD, it will help to note that the bars are half a minute wide. By eye, the points of inflection look to be about three bars away from the center. So we would guess that the SD is about 1.5 minutes, and in fact that is not far from the exact value.

What is the exact value? We will answer that question later in the term. Remarkably, it can be calculated easily based on the sample size and the SD of the population, without doing any simulations at all.

For now, here is the main point to note.

## Second main reason for using the SD to measure spread

For a large random sample, the probability distribution of the sample average will be roughly bell-shaped, with a mean and SD that can be easily identified, no matter what the distribution of the population from which the sample is drawn.

## The standard normal curve

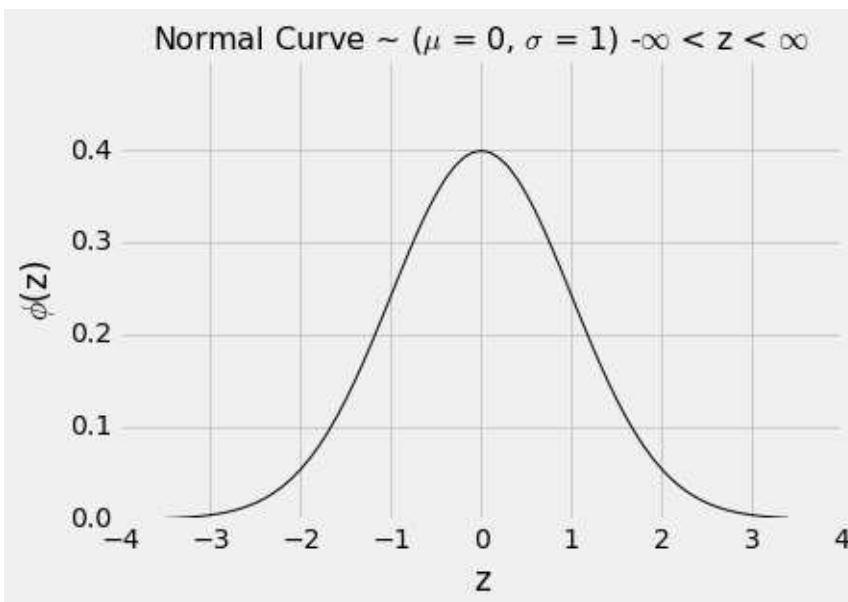
The bell-shaped curves above look essentially all the same apart from the labels that we have placed on the horizontal axes. Indeed, there is really just one basic curve from which all of these curves can be drawn just by relabeling the axes appropriately.

To draw that basic curve, we will use the units into which we can convert every list: standard units. The resulting curve is therefore called the *standard normal curve*.

The standard normal curve has an impressive equation. But for now, it is best to think of it as a smoothed version of a histogram of a variable that has been measured in standard units.

$$\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2}, \quad -\infty < z < \infty$$

```
# The standard normal probability density function (pdf)
plot_normal_cdf()
```



As always when you examine a new histogram, start by looking at the horizontal axis. On the horizontal axis of the standard normal curve, the values are standard units. Here are some properties of the curve. Some are apparent by observation, and others require a considerable amount of mathematics to establish.

- The total area under the curve is 1. So you can think of it as a histogram drawn to the density scale.
- The curve is symmetric about 0. So if a variable has this distribution, its mean and median are both 0.
- The points of inflection of the curve are at -1 and +1.
- If a variable has this distribution, its SD is 1. The normal curve is one of the very few distributions that has an SD so clearly identifiable on the histogram.

Since we are thinking of the curve as a smoothed histogram, we will want to represent proportions of the total amount of data by areas under the curve. Areas under smooth curves are often found by calculus, using a method called integration. It is a fact of mathematics, however, that the standard normal curve cannot be integrated in any of the

usual ways of calculus. Therefore, areas under the curve have to be approximated. That is why almost all statistics textbooks carry tables of areas under the normal curve. It is also why all statistical systems, including the `stats` module of Python, include methods that provide excellent approximations to those areas.

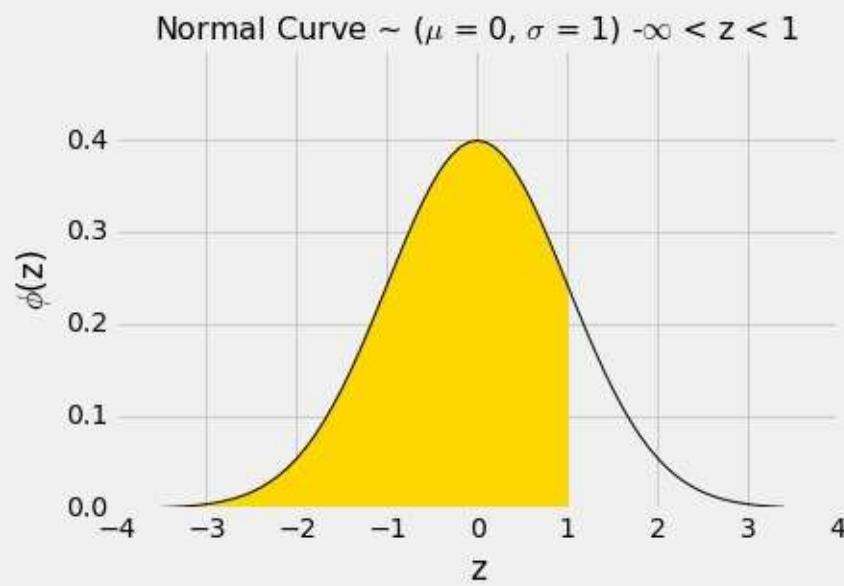
## Areas under the normal curve

### The standard normal cumulative distribution function (cdf)

The fundamental function for finding areas under the normal curve is `stats.norm.cdf`. It takes a numerical argument and returns all the area under the curve to the left of that number. Formally, it is called the "cumulative distribution function" of the standard normal curve. That rather unwieldy mouthful is abbreviated as cdf.

Let us use this function to find the area to the left of  $z = 1$  under the standard normal curve.

```
# Area under the standard normal curve, below 1
plot_normal_cdf(1)
```



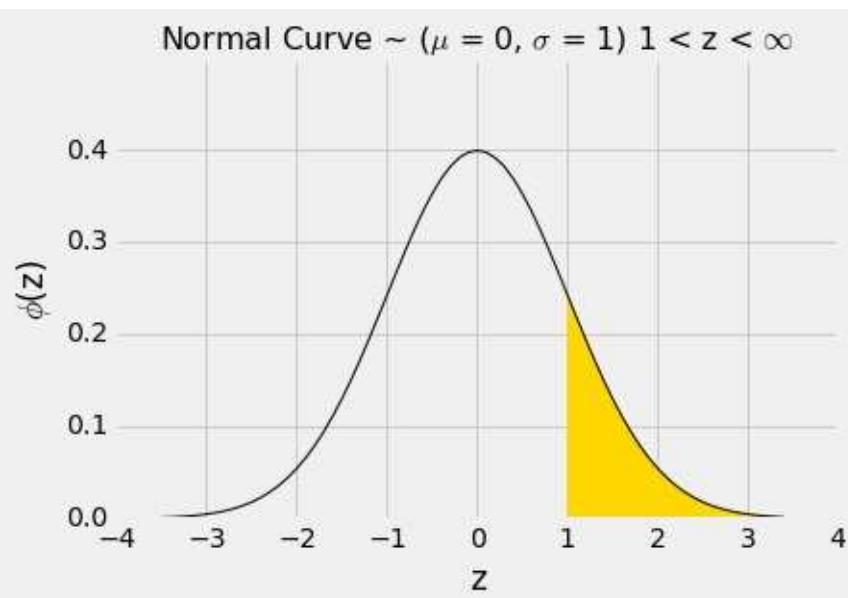
```
stats.norm.cdf(1)
```

```
0.84134474606854293
```

That's about 84%. We can now use the symmetry of the curve and the fact that the total area under the curve is 1 to find other areas.

For example, the area to the right of  $z = 1$  is about  $100\% - 84\% = 16\%$ .

```
# Area under the standard normal curve, above 1  
plot_normal_cdf(lbound=1)
```

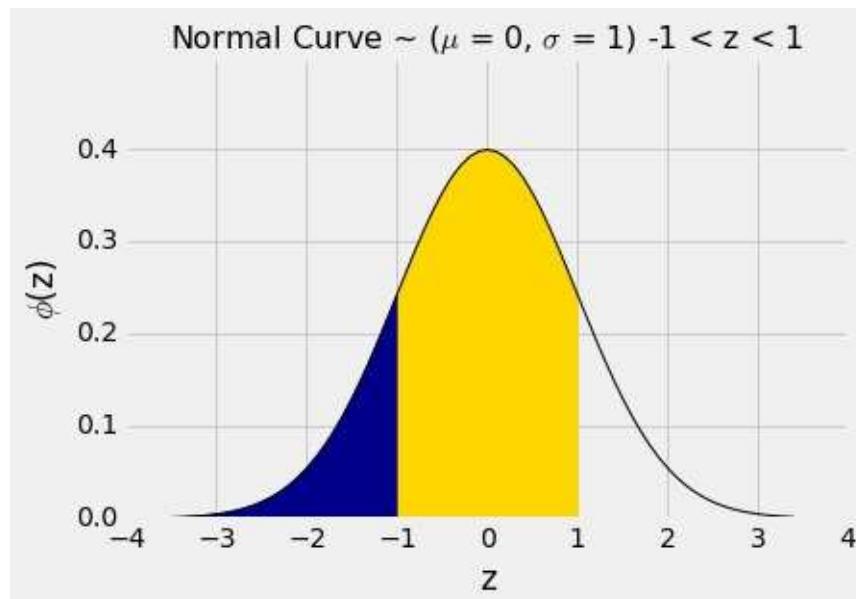


```
1 - stats.norm.cdf(1)
```

```
0.15865525393145707
```

The area between  $z = -1$  and  $z = 1$  can be computed in several different ways.

```
# Area under the standard normal curve, between -1 and 1  
plot_normal_cdf(1, lbound=-1)
```



For example, we can calculate the area as "100% - two equal tails", which works out to roughly  $100\% - 2 \times 16\% = 68\%$ .

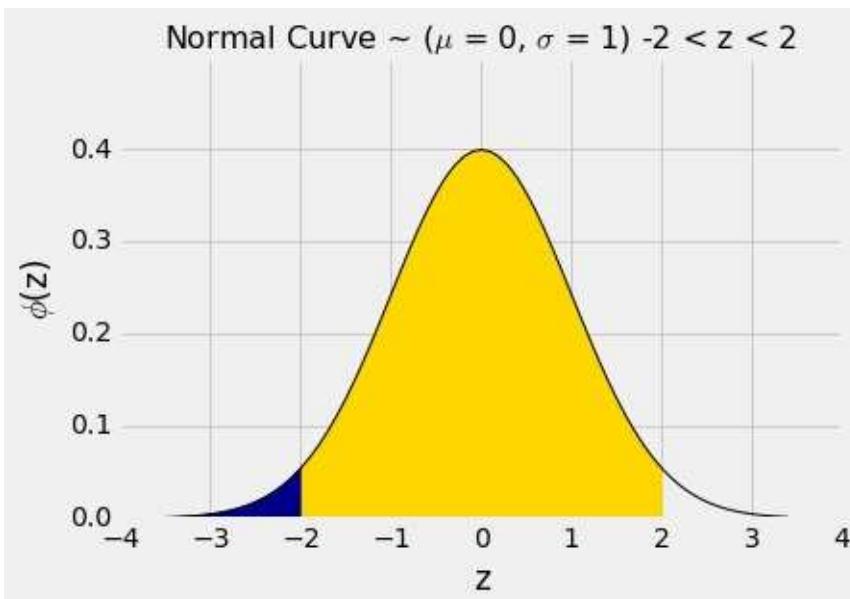
Or we could note that the area between  $z = 1$  and  $z = -1$  is equal to all the area to the left of  $z = 1$ , minus all the area to the left of  $z = -1$ .

```
stats.norm.cdf(1) - stats.norm.cdf(-1)
```

```
0.68268949213708585
```

By a similar calculation, we see that the area between  $-2$  and  $2$  is about 95%.

```
# Area under the standard normal curve, between -2 and 2
plot_normal_cdf(2, lbound=-2)
```



```
stats.norm.cdf(2) - stats.norm.cdf(-2)
```

```
0.95449973610364158
```

In other words, if a histogram is roughly bell shaped, the proportion of data in the range "average  $\pm 2$  SDs" is about 95%.

That is quite a bit more than Chebychev's lower bound of 75%. Chebychev's bound is weaker because it has to work for all distributions. If we know that a distribution is normal, we have good approximations to the proportions, not just bounds.

The table below compares what we know about all distributions and about normal distributions. Notice that Chebychev's bound is not very illuminating when  $z = 1$ .

Percent in Range	All Distributions	Normal Distribution
average $\pm 1$ SD	at least 0%	about 68%
average $\pm 2$ SDs	at least 75%	about 95%
average $\pm 3$ SDs	at least 88.888...%	about 99.73%

# Exploration: Privacy

Section author: [David Wagner](#)

[Interact](#)

## License plates

We're going to look at some data collected by the Oakland Police Department. They have automated license plate readers on their police cars, and they've built up a database of license plates that they've seen -- and where and when they saw each one.

## Data collection

First, we'll gather the data. It turns out the data is publicly available on the Oakland public records site. I downloaded it and combined it into a single CSV file by myself before lecture.

```
lprs = Table.read_table('./all-lprs.csv.gz', compression='gzip', se  
lprs.labels  
('red_VRM', 'red_Timestamp', 'Location')
```

Let's start by renaming some columns, and then take a look at it.

```
lprs.relabel('red_VRM', 'Plate')  
lprs.relabel('red_Timestamp', 'Timestamp')  
lprs
```

Plate	Timestamp	Location
1275226	01/19/2011 02:06:00 AM	(37.798304999999999, -122.27574799999999)
27529C	01/19/2011 02:06:00 AM	(37.798304999999999, -122.27574799999999)
1158423	01/19/2011 02:06:00 AM	(37.798304999999999, -122.27574799999999)
1273718	01/19/2011 02:06:00 AM	(37.798304999999999, -122.27574799999999)
1077682	01/19/2011 02:06:00 AM	(37.798304999999999, -122.27574799999999)
1214195	01/19/2011 02:06:00 AM	(37.798281000000003, -122.27575299999999)
1062420	01/19/2011 02:06:00 AM	(37.79833, -122.27574300000001)
1319726	01/19/2011 02:05:00 AM	(37.798475000000003, -122.27571500000001)
1214196	01/19/2011 02:05:00 AM	(37.798499999999997, -122.27571)
75227	01/19/2011 02:05:00 AM	(37.798596000000003, -122.27569)

... (2742091 rows omitted)

Phew, that's a lot of data: we can see about 2.7 million license plate reads here.

Let's start by seeing what can be learned about someone, using this data -- assuming you know their license plate.

## Stalking Jean Quan

As a warmup, we'll take a look at ex-Mayor Jean Quan's car, and where it has been seen. Her license plate number is 6FCH845. (How did I learn that? Turns out she was in the news for getting \$1000 of parking tickets, and [the news article](#) included a picture of her car, with the license plate visible. You'd be amazed by what's out there on the Internet...)

```
lprs.where('Plate', '6FCH845')
```

Plate	Timestamp	Location
6FCH845	11/01/2012 09:04:00 AM	(37.79871, -122.276221)
6FCH845	10/24/2012 11:15:00 AM	(37.799695, -122.274868)
6FCH845	10/24/2012 11:01:00 AM	(37.799693, -122.274806)
6FCH845	10/24/2012 10:20:00 AM	(37.799735, -122.274893)
6FCH845	05/08/2014 07:30:00 PM	(37.797558, -122.26935)
6FCH845	12/31/2013 10:09:00 AM	(37.807556, -122.278485)

OK, so her car shows up 6 times in this data set. However, it's hard to make sense of those coordinates. I don't know about you, but I can't read GPS so well.

So, let's work out a way to show where her car has been seen on a map. We'll need to extract the latitude and longitude, as the data isn't quite in the format that the mapping software expects: the mapping software expects the latitude to be in one column and the longitude in another. Let's write some Python code to do that, by splitting the Location string into two pieces: the stuff before the comma (the latitude) and the stuff after (the longitude).

```
def get_latitude(s):
    before, after = s.split(',')           # Break it into two parts
    lat_string = before.replace('(', '')   # Get rid of the annoying
    return float(lat_string)              # Convert the string to a

def get_longitude(s):
    before, after = s.split(',')          # Break it into two parts
    long_string = after.replace(')', '').strip() # Get rid of the ')'
    return float(long_string)             # Convert the string to a float
```

Let's test it to make sure it works correctly.

```
get_latitude('(37.797558, -122.26935)')
```

```
37.797558
```

```
get_longitude('(37.797558, -122.26935)')
```

```
-122.26935
```

Good, now we're ready to add these as extra columns to the table.

```
lprs = lprs.with_columns([
    'Latitude', lprs.apply(get_latitude, 'L'),
    'Longitude', lprs.apply(get_longitude, 'L')
])
lprs
```

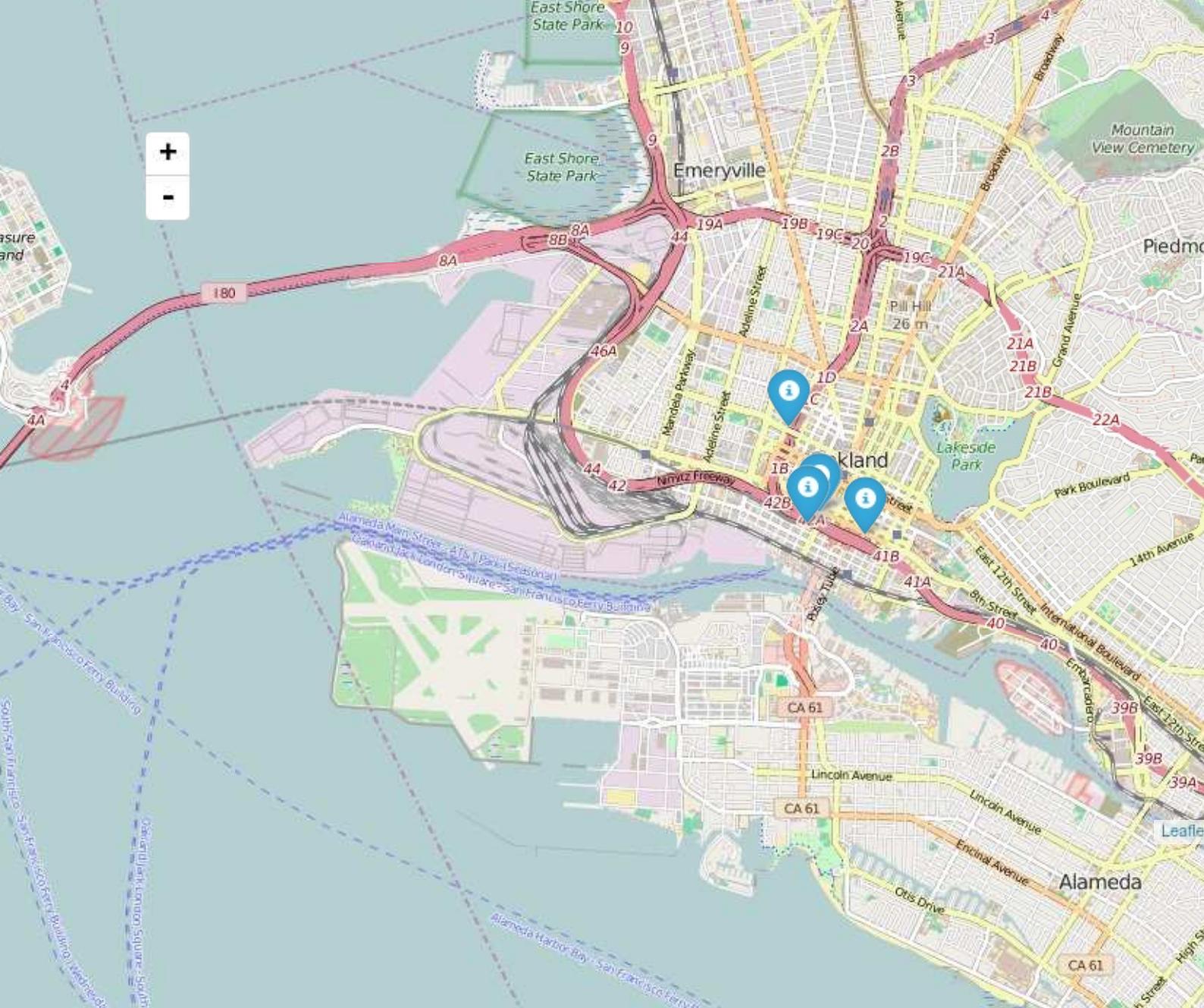
Plate	Timestamp	Location		
1275226	01/19/2011 02:06:00 AM	(37.79830499999999, -122.2757479999999)		
27529C	01/19/2011 02:06:00 AM	(37.79830499999999, -122.2757479999999)		
1158423	01/19/2011 02:06:00 AM	(37.79830499999999, -122.2757479999999)		
1273718	01/19/2011 02:06:00 AM	(37.79830499999999, -122.2757479999999)		
1077682	01/19/2011 02:06:00 AM	(37.79830499999999, -122.2757479999999)		
1214195	01/19/2011 02:06:00 AM	(37.798281000000003, -122.2757529999999)		
1062420	01/19/2011 02:06:00 AM	(37.79833, -122.27574300000001)	37.7983	-122.276
1319726	01/19/2011 02:05:00 AM	(37.798475000000003, -122.27571500000001)	37.7985	-122.276
1214196	01/19/2011 02:05:00 AM	(37.79849999999997, -122.27571)	37.7985	-122.276
75227	01/19/2011 02:05:00 AM	(37.798596000000003, -122.27569)	37.7986	-122.276

... (2742091 rows omitted)

And at last, we can draw a map with a marker everywhere that her car has been seen.

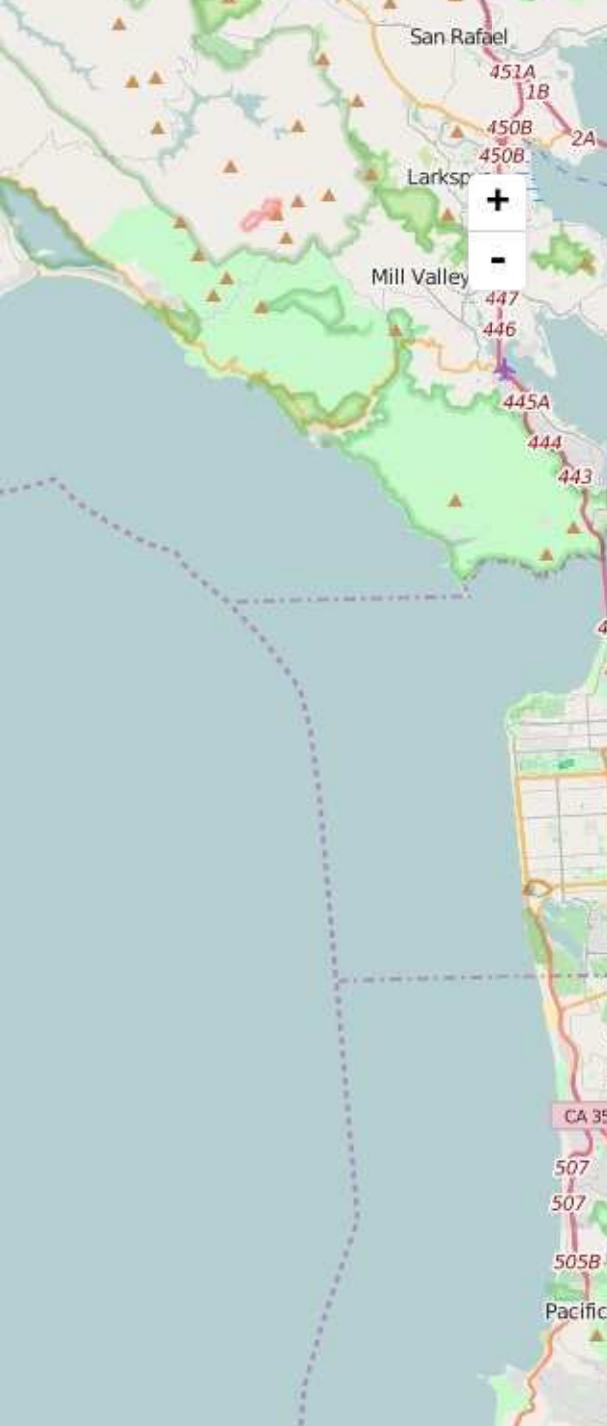
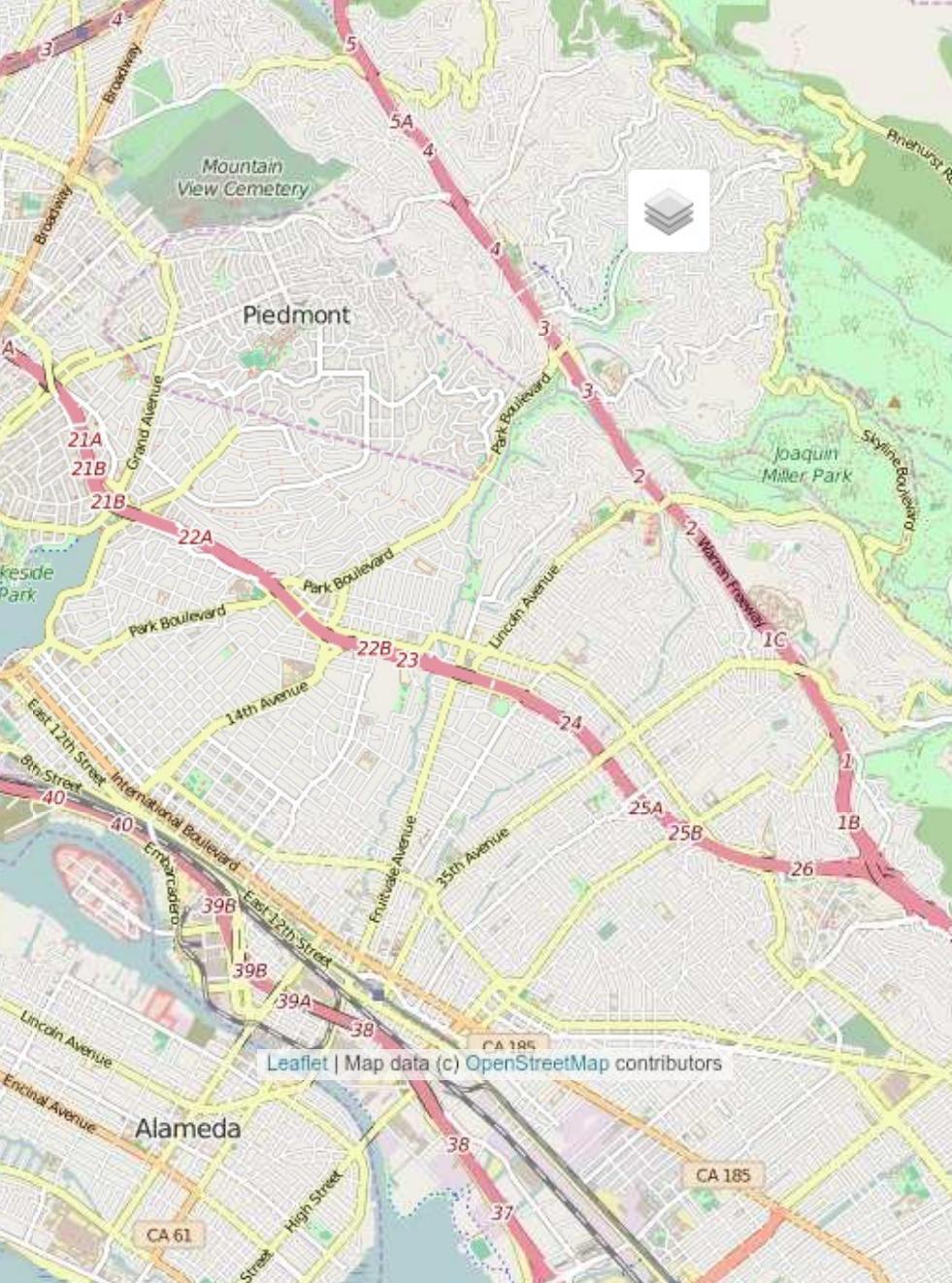
```
jean_quan = lprs.where('Plate', '6FCH845').select(['Latitude', 'Longitude'])
Marker.map_table(jean_quan)
```

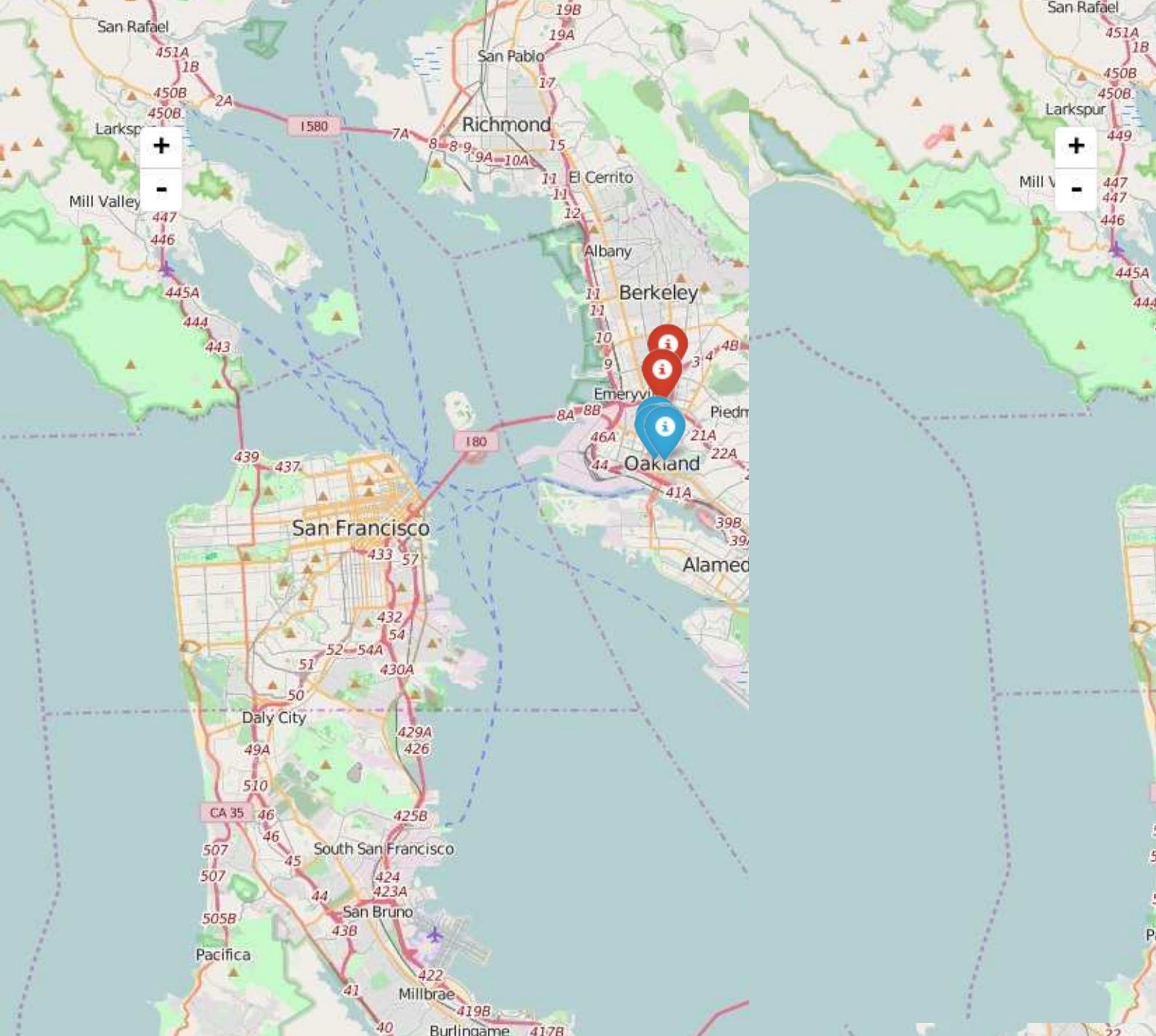


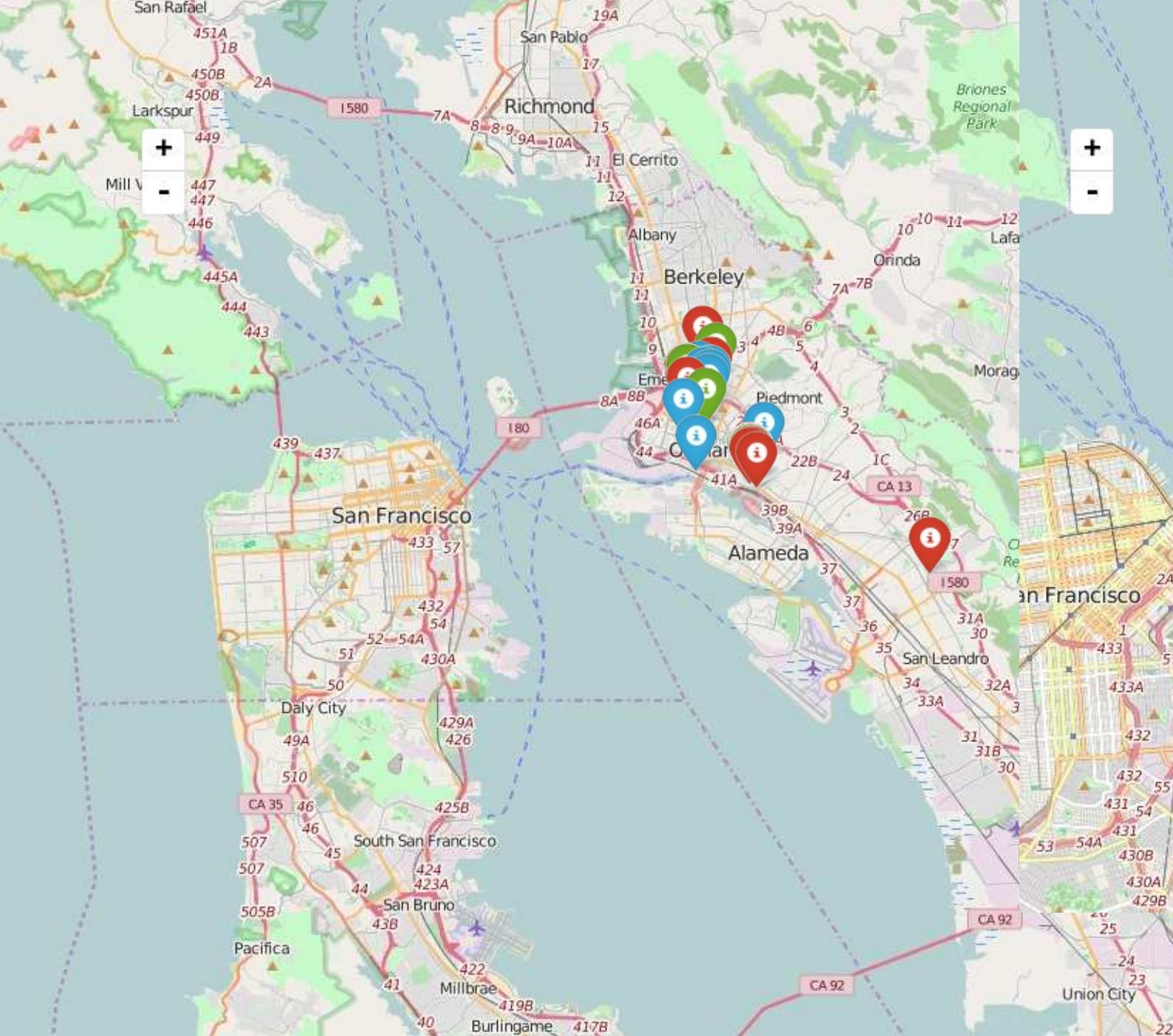


## Poking around

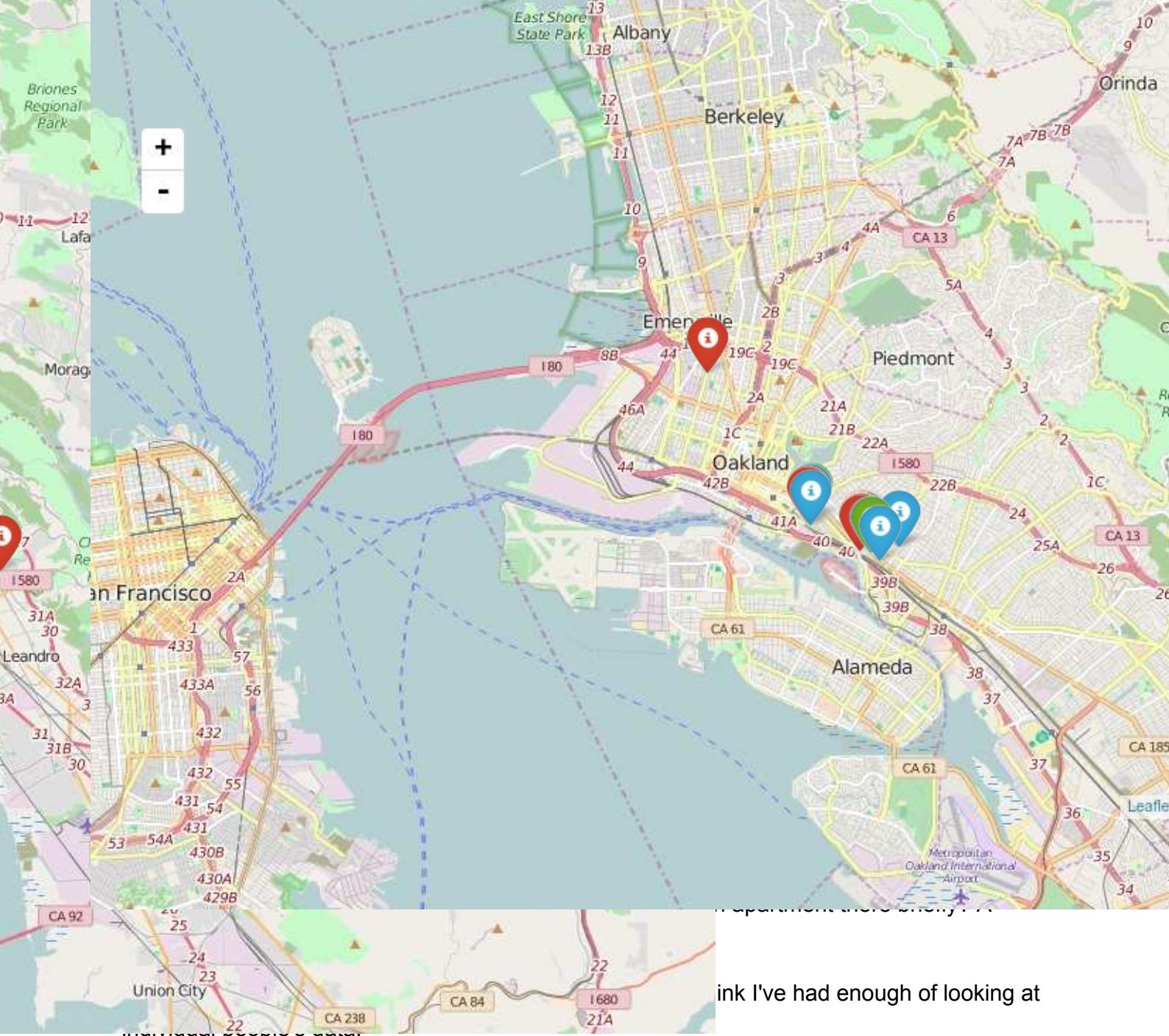
Let's try another. And let's see if we can make the map a little more fancy. It'd be nice to distinguish between license plate reads that are seen during the daytime (on a weekday), vs the evening (on a weekday), vs on a weekend. So we'll color-code the markers. To do this, we'll write some Python code to analyze the Timestamp and choose an appropriate color.







```
map_plate('6UZA652')
```

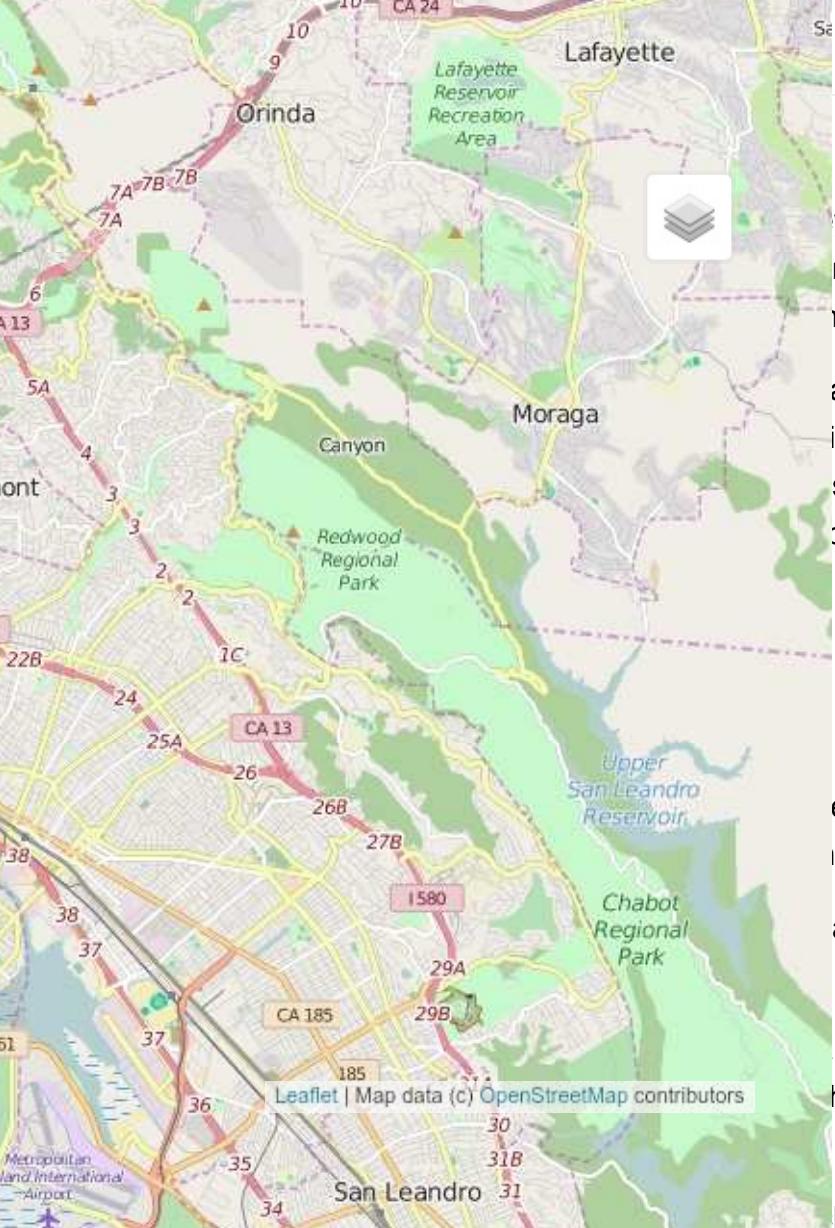


ink I've had enough of looking at

## Inference

As we can see, this kind of data can potentially reveal a fair bit about people. Someone with access to the data can draw inferences. Take a moment to think about what someone might be able to infer from this kind of data.

As we've seen here, it's not too hard to make a pretty good guess at roughly where some lives, from this kind of information: their car is probably parked near their home most nights. Also, it will often be possible to guess where someone works: if they commute into work by car, then on weekdays during business hours, their car is probably parked near their office, so we'll see a clear cluster that indicates where they work.



Unfortunately, this is often harder than it sounds.

4. Engage with stakeholders. Provide transparency, to try to avoid people being taken by surprise. Give individuals a way to see what data has been collected about them. Give people a way to opt out and have their data be deleted, if they wish. Engage in a discussion about values, and tell people what steps you are taking to protect them from unwanted consequences.

This only scratches the surface of the subject. My main goal in this lecture was to make you aware of privacy concerns, so that if you are ever a steward of a large data set, you can think about how to protect people's data and use it responsibly.

it might also be possible to get a sense of (e.g., how much time do you spend at the park?). And in some cases, this data can reveal sensitive information about relationships and spending nights at bars. It's important to be aware of the sensitive stuff.

Data that's collected for one purpose can often reveal a lot more. It can allow the wrong people to learn things about things that people would prefer to keep private. "Big data", if we're not careful, privacy can be compromised.

What can be done about this? That's a lengthy subject, but here are some simple strategies that data owners can take:

- Limit who has access to the data at they need, and delete it after it's not needed.

• Perhaps only a handful of trusted insiders need access to the data. Limit who has access to the data so only they have access to it. One way to do this is to de-identify the data, which means removing information that could be used to identify the individual who it is about.

# Chapter 4

## Prediction

# Correlation

[Interact](#)

## The relation between two variables

In the previous sections, we developed several tools that help us describe the distribution of a single variable. Data science also helps us understand how multiple variables are related to each other. This allows us to predict the value of one variable given the values of others, and to get a sense of the amount of error in the prediction.

A good way to start exploring the relation between two variables is by visualization. A graph called a *scatter diagram* can be used to plot the value of one variable against values of another. Let us start by looking at some scatter diagrams and then move on to quantifying some of the features that we see.

The table `hybrid` contains data on hybrid passenger cars sold in the United States from 1997 to 2013. The data were obtained from the online data archive of [Prof. Larry Winner](#) of the University of Florida. The columns include the model of the car, year of manufacture, the MSRP (manufacturer's suggested retail price) in 2013 dollars, the acceleration rate in km per hour per second, fuel economy in miles per gallon, and the model's class.

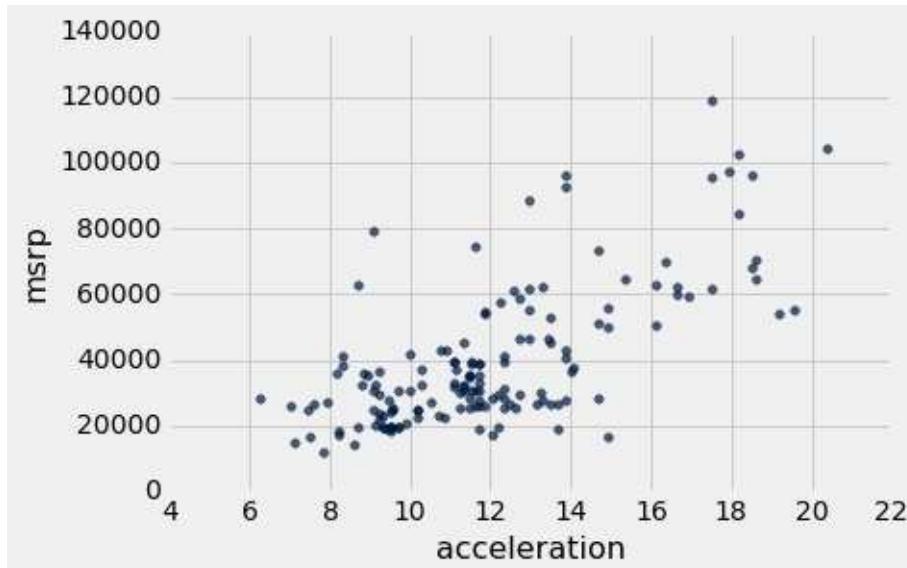
hybrid

vehicle	year	msrp	acceleration	mpg	class
Prius (1st Gen)	1997	24509.7	7.46	41.26	Compact
Tino	2000	35355	8.2	54.1	Compact
Prius (2nd Gen)	2000	26832.2	7.97	45.23	Compact
Insight	2000	18936.4	9.52	53	Two Seater
Civic (1st Gen)	2001	25833.4	7.04	47.04	Compact
Insight	2001	19036.7	9.52	53	Two Seater
Insight	2002	19137	9.71	53	Two Seater
Alphard	2003	38084.8	8.33	40.46	Minivan
Insight	2003	19137	9.52	53	Two Seater
Civic	2003	14071.9	8.62	41	Compact

... (143 rows omitted)

The Table method `scatter` can be used to plot `acceleration` on the horizontal axis (the first argument) and `msrp` on the vertical (the second argument). There are 153 points in the scatter, one for each car in the table.

```
hybrid.scatter('acceleration', 'msrp')
```

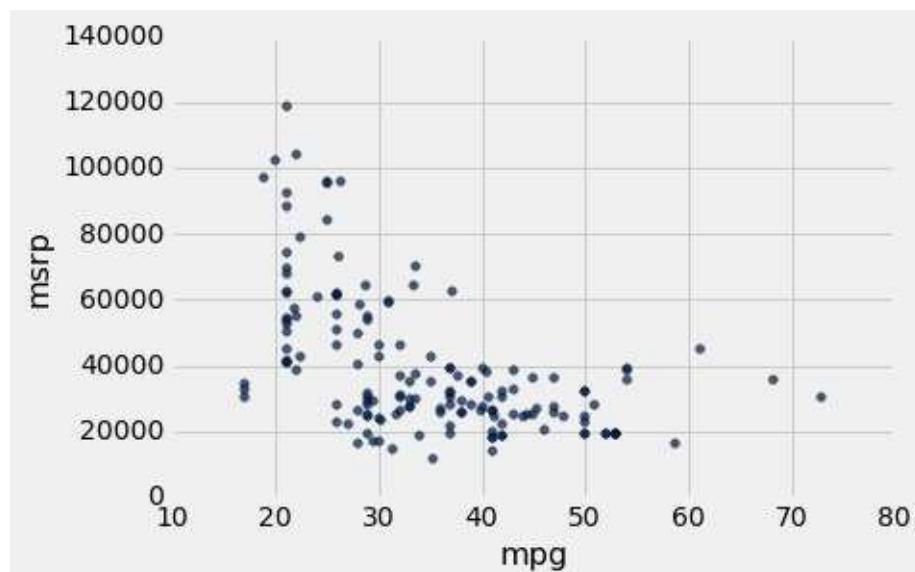


The scatter of points is sloping upwards, indicating that cars with greater acceleration tended to cost more, on average; conversely, the cars that cost more tended to have greater acceleration on average.

This is an example of *positive association*: above-average values of one variable tend to be associated with above-average values of the other.

The scatter diagram of MSRP (vertical axis) versus mileage (horizontal axis) shows a *negative association*. The scatter has a clear downward trend. Hybrid cars with higher mileage tended to cost less, on average. This seems surprising till you consider that cars that accelerate fast tend to be less fuel efficient and have lower mileage. As the previous scatter showed, those were also the cars that tended to cost more.

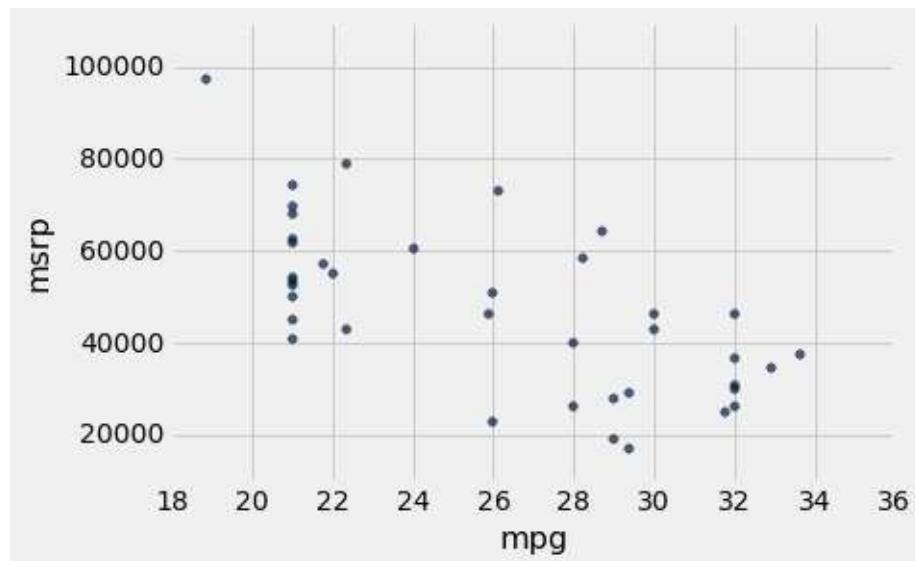
```
hybrid.scatter('mpg', 'msrp')
```



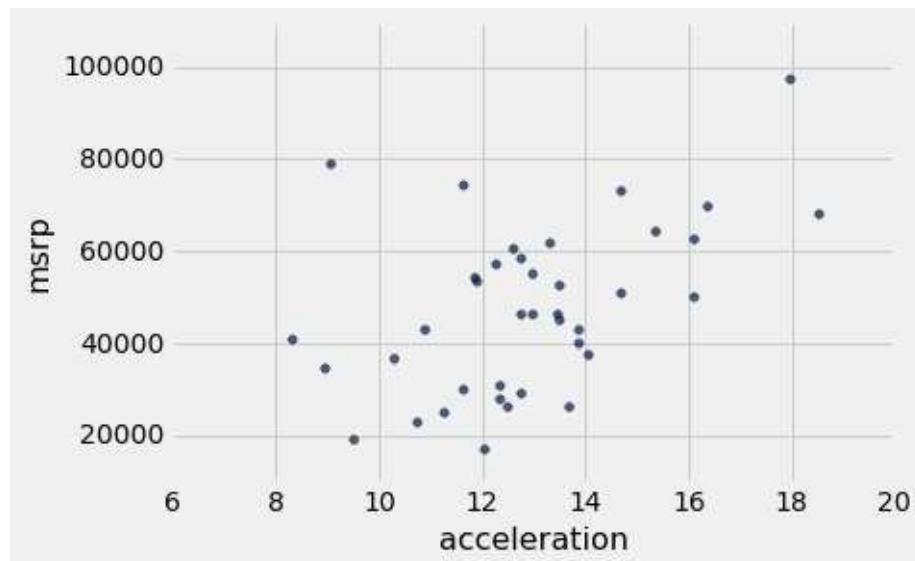
Along with the negative association, the scatter diagram of price versus efficiency shows a non-linear relation between the two variables. The points appear to be clustered around a curve.

If we restrict the data just to the SUV class, however, the association between price and efficiency is still negative but the relation appears to be more linear. The relation between the price and acceleration of SUV's also shows a linear trend, but with a positive slope.

```
suv = hybrid.where('class', 'SUV')
suv.scatter('mpg', 'msrp')
```



```
suv.scatter('acceleration', 'msrp')
```

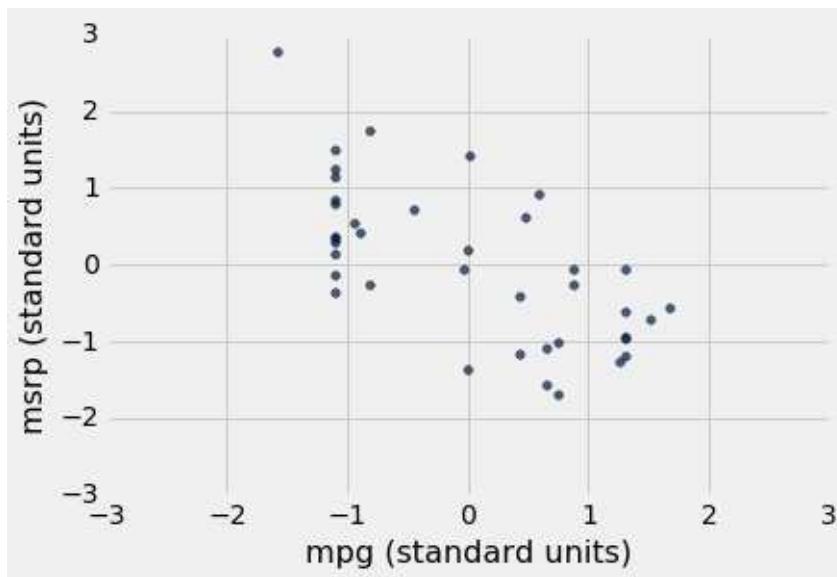


You will have noticed that we can derive useful information from the general orientation and shape of a scatter diagram even without paying attention to the units in which the variables were measured.

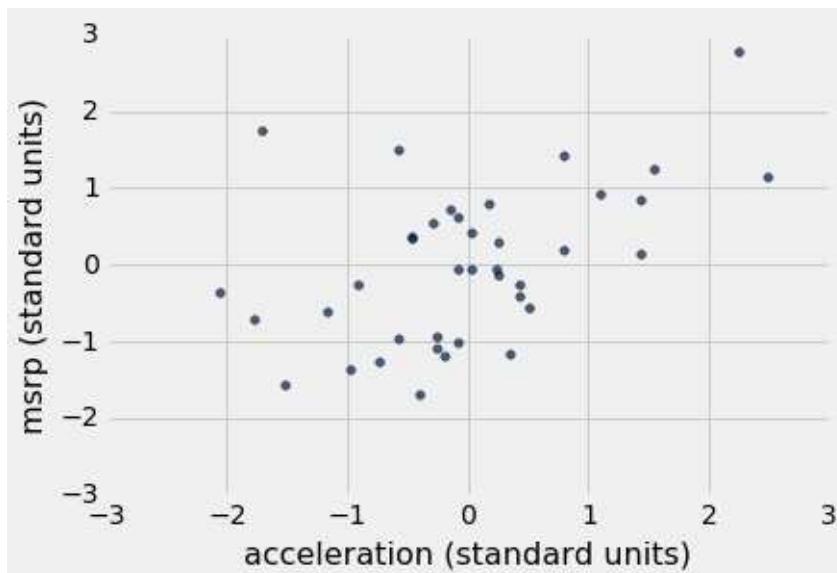
Indeed, we could plot all the variables in standard units and the plots would look the same. This gives us a way to compare the degree of linearity in two scatter diagrams.

Here are the two scatter diagrams for SUVs, with all the variables measured in standard units.

```
Table().with_columns([
    'mpg (standard units)', standard_units(suv.column('mpg')),
    'msrp (standard units)', standard_units(suv.column('msrp'))
]).scatter(0, 1)
plots.xlim([-3, 3])
plots.ylim([-3, 3])
None
```



```
Table().with_columns([
    'acceleration (standard units)', standard_units(suv.column('acc
    'msrp (standard units)', standard_units(suv.column('msr
]).scatter(0, 1)
plots.xlim([-3, 3])
plots.ylim([-3, 3])
None
```



The associations that we see in these figures are the same as those we saw before. Also, because the two scatter diagrams are drawn on exactly the same scale, we can see that the linear relation in the second diagram is a little more fuzzy than in the first.

We will now define a measure that uses standard units to quantify the kinds of association that we have seen.

# The correlation coefficient

The *correlation coefficient* measures the strength of the linear relationship between two variables. Graphically, it measures how clustered the scatter diagram is around a straight line.

The term *correlation coefficient* isn't easy to say, so it is usually shortened to *correlation* and denoted by  $r$ .

Here are some mathematical facts about  $r$  that we will observe by simulation.

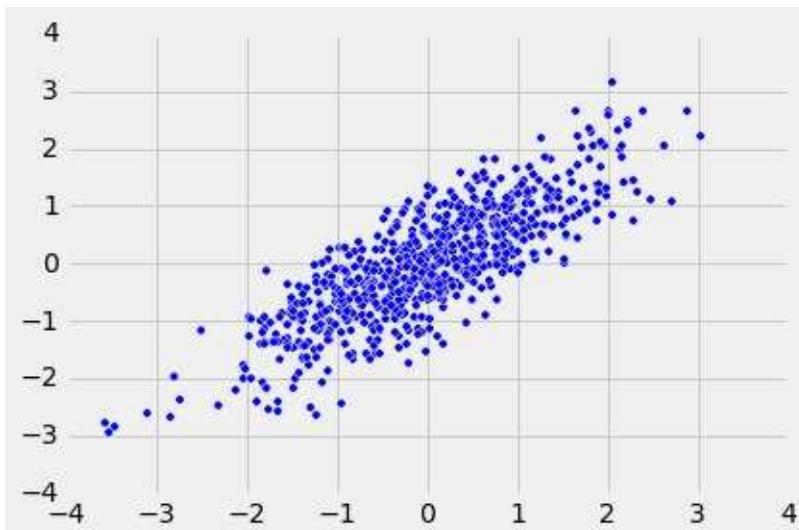
- The correlation coefficient  $r$  is a number between  $-1$  and  $1$ .
- $r$  measures the extent to which the scatter plot clusters around a straight line.
- $r = 1$  if the scatter diagram is a perfect straight line sloping upwards, and  $r = -1$  if the scatter diagram is a perfect straight line sloping downwards.

The function `r_scatter` takes a value of  $r$  as its argument and simulates a scatter plot with a correlation very close to  $r$ . Because of randomness in the simulation, the correlation is not expected to be exactly equal to  $r$ .

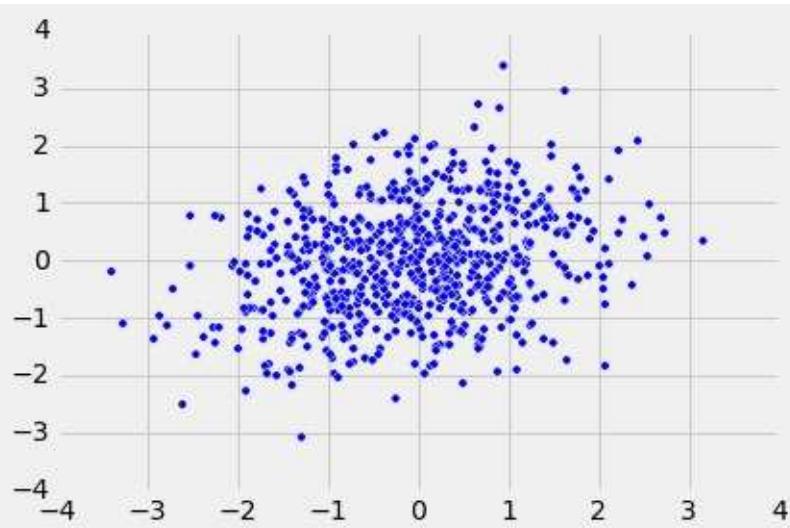
Call `r_scatter` a few times, with different values of  $r$  as the argument, and see how the football changes. Positive  $r$  corresponds to positive association: above-average values of one variable are associated with above-average values of the other, and the scatter plot slopes upwards.

When  $r = 1$  the scatter plot is perfectly linear and slopes upward. When  $r = -1$ , the scatter plot is perfectly linear and slopes downward. When  $r = 0$ , the scatter plot is a formless cloud around the horizontal axis, and the variables are said to be *uncorrelated*.

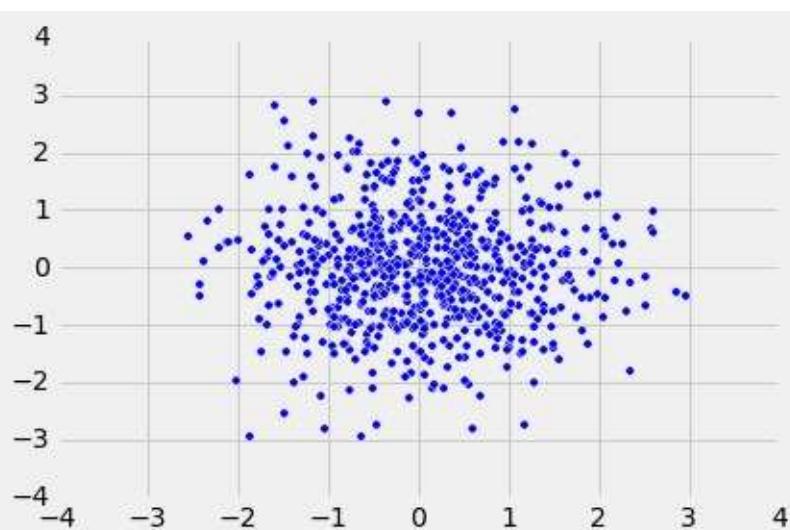
```
r_scatter(0.8)
```



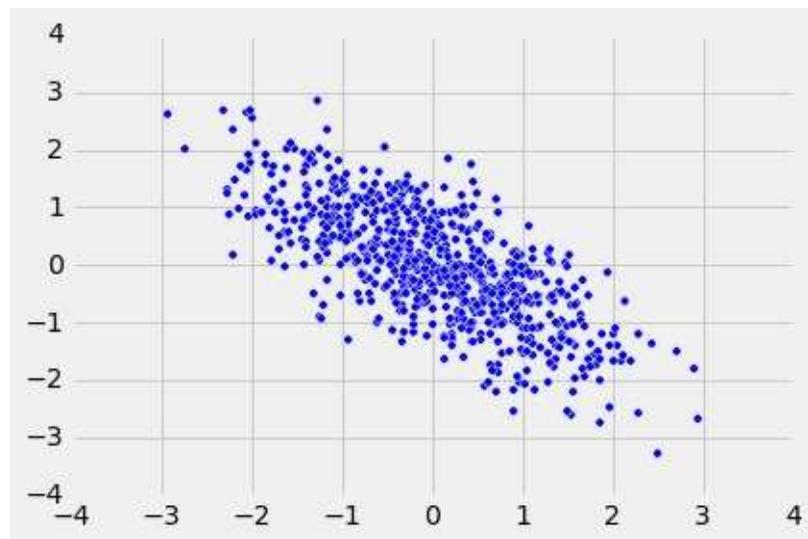
```
r_scatter(0.25)
```



```
r_scatter(0)
```



```
r_scatter(-0.7)
```



## Calculating $r$

The formula for  $r$  is not apparent from our observations so far, and it has a mathematical basis that is outside the scope of this class. However, the calculation is straightforward and helps us understand several of the properties of  $r$ .

### Formula for $r$ :

- $r$  is the average of the products of the two variables, when both variables are measured in standard units.

Here are the steps in the calculation. We will apply the steps to a simple table of values of  $x$  and  $y$ .

```

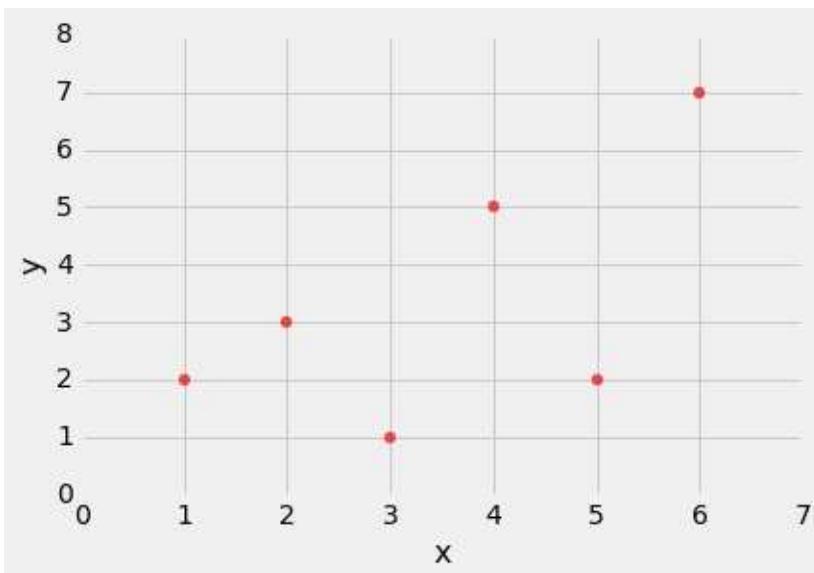
x = np.arange(1, 7, 1)
y = [2, 3, 1, 5, 2, 7]
t = Table().with_columns([
    'x', x,
    'y', y
])
t

```

x	y
1	2
2	3
3	1
4	5
5	2
6	7

Based on the scatter diagram, we expect that  $r$  will be positive but not equal to 1.

```
t.scatter(0, 1, s=30, color='red')
```



**Step 1.** Convert each variable to standard units.

```
t_su = t.with_columns([
    'x (standard units)', standard_units(x),
    'y (standard units)', standard_units(y)
])
t_su
```

x	y	x (standard units)	y (standard units)
1	2	-1.46385	-0.648886
2	3	-0.87831	-0.162221
3	1	-0.29277	-1.13555
4	5	0.29277	0.811107
5	2	0.87831	-0.648886
6	7	1.46385	1.78444

**Step 2.** Multiply each pair of standard units.

```
t_product = t_su.with_column('product of standard units', t_su.colu
t_product
```

x	y	x (standard units)	y (standard units)	product of standard units
1	2	-1.46385	-0.648886	0.949871
2	3	-0.87831	-0.162221	0.142481
3	1	-0.29277	-1.13555	0.332455
4	5	0.29277	0.811107	0.237468
5	2	0.87831	-0.648886	-0.569923
6	7	1.46385	1.78444	2.61215

**Step 3.**  $r$  is the average of the products computed in Step 2.

```
# r is the average of the products of standard units
r = np.mean(t_product.column(4))
r
```

0.61741639718977093

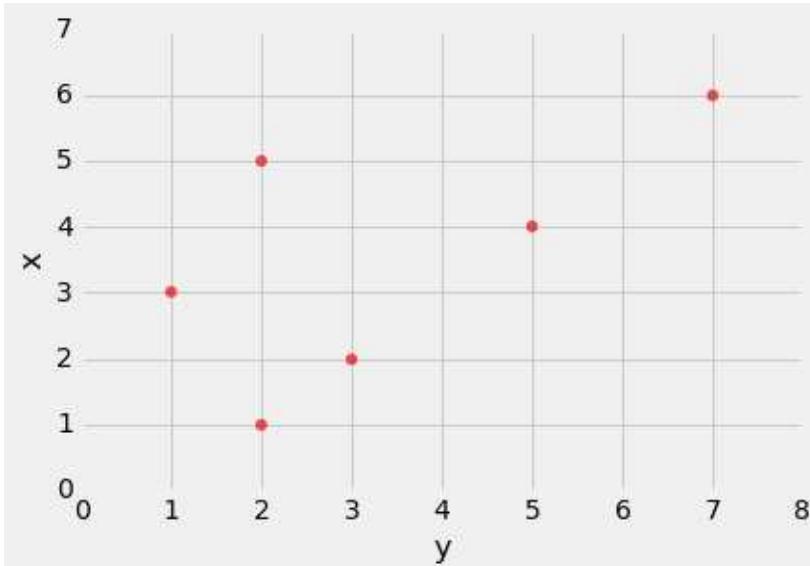
As expected,  $r$  is positive but not equal to 1.

## Properties of $r$

The calculation shows that:

- $r$  is a pure number. It has no units. This is because  $r$  is based on standard units.
- $r$  is unaffected by changing the units on either axis. This too is because  $r$  is based on standard units.
- $r$  is unaffected by switching the axes. Algebraically, this is because the product of standard units does not depend on which variable is called  $x$  and which  $y$ . Geometrically, switching axes reflects the scatter plot about the line  $y = x$ , but does not change the amount of clustering nor the sign of the association.

```
t.scatter('y', 'x', s=30, color='red')
```



We can define a function `correlation` to compute  $r$ , based on the formula that we used above. The arguments are a table and two labels of columns in the table. The function returns the mean of the products of those column values in standard units, which is  $r$ .

```
def correlation(t, x, y):
    return np.mean(standard_units(t.column(x))*standard_units(t.col
```

Let's call the function on the `x` and `y` columns of `t`. The function returns the same answer to the correlation between  $x$  and  $y$  as we got by direct application of the formula for  $r$ .

```
correlation(t, 'x', 'y')
```

```
0.61741639718977093
```

Calling `correlation` on columns of the table `suv` gives us the correlation between price and mileage as well as the correlation between price and acceleration.

```
correlation(suv, 'mpg', 'msrp')
```

```
-0.6667143635709919
```

```
correlation(suv, 'acceleration', 'msrp')
```

```
0.48699799279959155
```

These values confirm what we had observed:

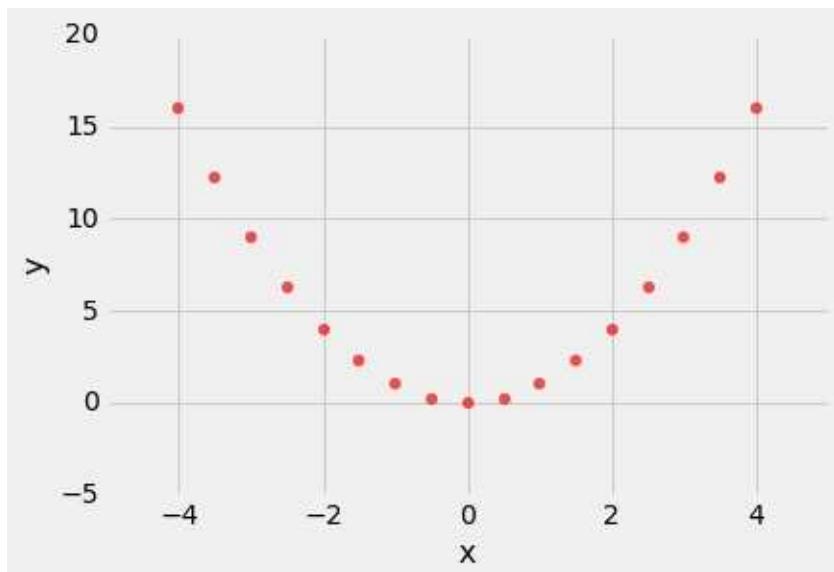
- There is a negative association between price and efficiency, whereas the association between price and acceleration is positive.
- The linear relation between price and acceleration is a little weaker (correlation about 0.5) than between price and mileage (correlation about -0.67).

## Care in Using Correlation

Correlation is a simple and powerful concept, but it is sometimes misused. Before using `r`, it is important to be aware of what correlation does and does not measure.

- Correlation only measures association. Correlation does not imply causation. Though the correlation between the weight and the math ability of children in a school district may be positive, that does not mean that doing math makes children heavier or that putting on weight improves the children's math skills. Age is a confounding variable: older children are both heavier and better at math than younger children, on average.
- Correlation measures **linear** association. Variables that have strong non-linear association might have very low correlation. Here is an example of variables that have a perfect quadratic relation  $y = x^2$  but have correlation equal to 0.

```
nonlinear = Table().with_columns([
    'x', np.arange(-4, 4.1, 0.5),
    'y', np.arange(-4, 4.1, 0.5)**2
])
nonlinear.scatter('x', 'y', s=30, color='r')
```

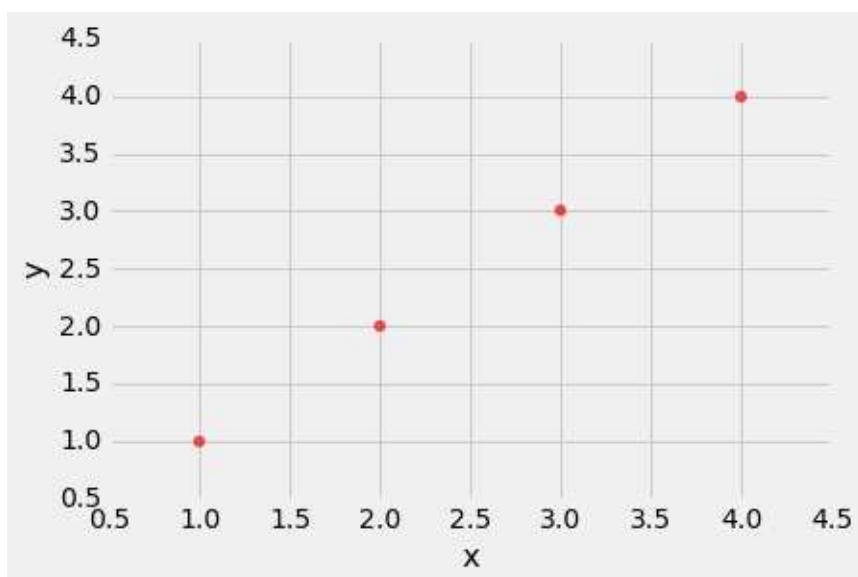


```
correlation(nonlinear, 'x', 'y')
```

```
0.0
```

- Outliers can have a big effect on correlation. Here is an example where a scatter plot for which  $r$  is equal to 1 is turned into a plot for which  $r$  is equal to 0, by the addition of just one outlying point.

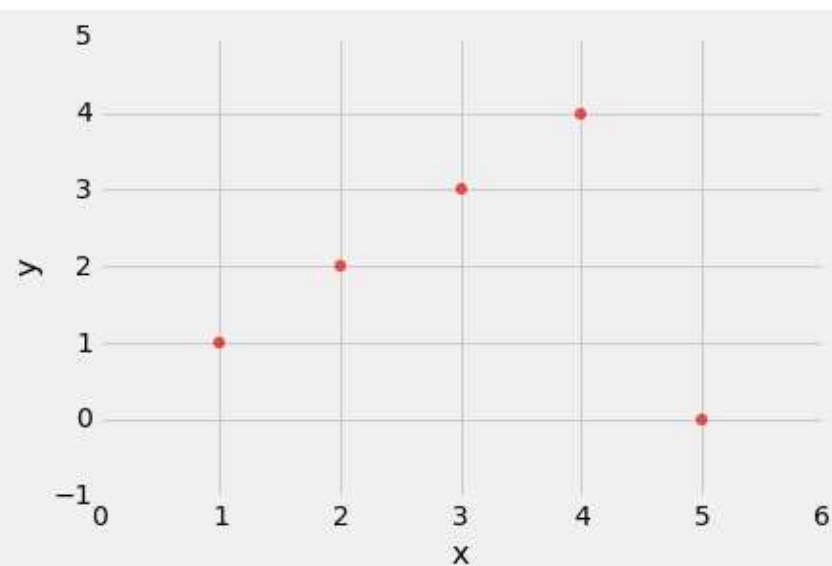
```
line = Table().with_columns([
    'x', [1, 2, 3, 4],
    'y', [1, 2, 3, 4]
])
line.scatter('x', 'y', s=30, color='r')
```



```
correlation(line, 'x', 'y')
```

1.0

```
outlier = Table().with_columns([
    'x', [1, 2, 3, 4, 5],
    'y', [1, 2, 3, 4, 0]
])
outlier.scatter('x', 'y', s=30, color='r')
```



```
correlation(outlier, 'x', 'y')
```

0.0

- Correlations based on aggregated data can be misleading. As an example, here are data on the Critical Reading and Math SAT scores in 2014. There is one point for each of the 50 states and one for Washington, D.C. The column `Participation Rate` contains the percent of high school seniors who took the test. The next three columns show the average score in the state on each portion of the test, and the final column is the average of the total scores on the test.

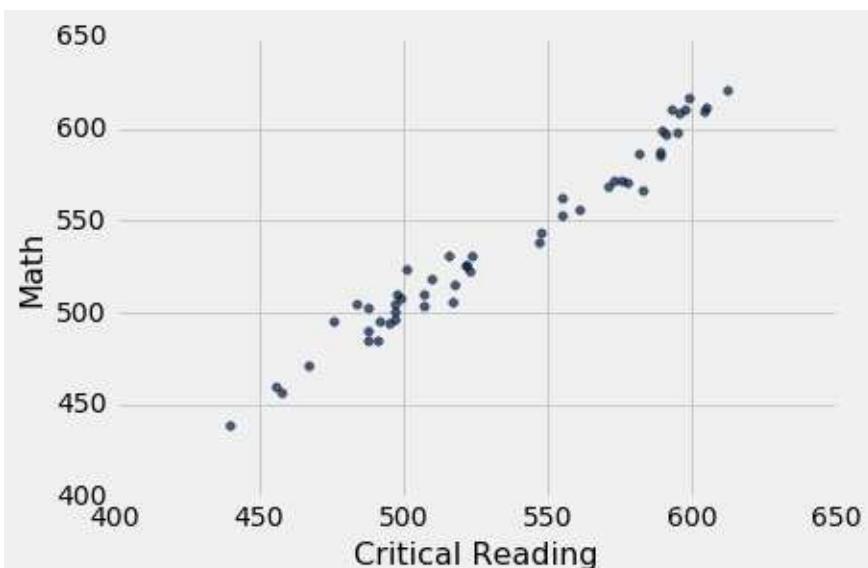
```
sat2014 = Table.read_table('sat2014.csv').sort('State')
sat2014
```

State	Participation Rate	Critical Reading	Math	Writing	Combined
Alabama	6.7	547	538	532	1617
Alaska	54.2	507	503	475	1485
Arizona	36.4	522	525	500	1547
Arkansas	4.2	573	571	554	1698
California	60.3	498	510	496	1504
Colorado	14.3	582	586	567	1735
Connecticut	88.4	507	510	508	1525
Delaware	100	456	459	444	1359
District of Columbia	100	440	438	431	1309
Florida	72.2	491	485	472	1448

... (41 rows omitted)

The scatter diagram of Math scores versus Critical Reading scores is very tightly clustered around a straight line; the correlation is close to 0.985.

```
sat2014.scatter('Critical Reading', 'Math')
```



```
correlation(sat2014, 'Critical Reading', 'Math')
```

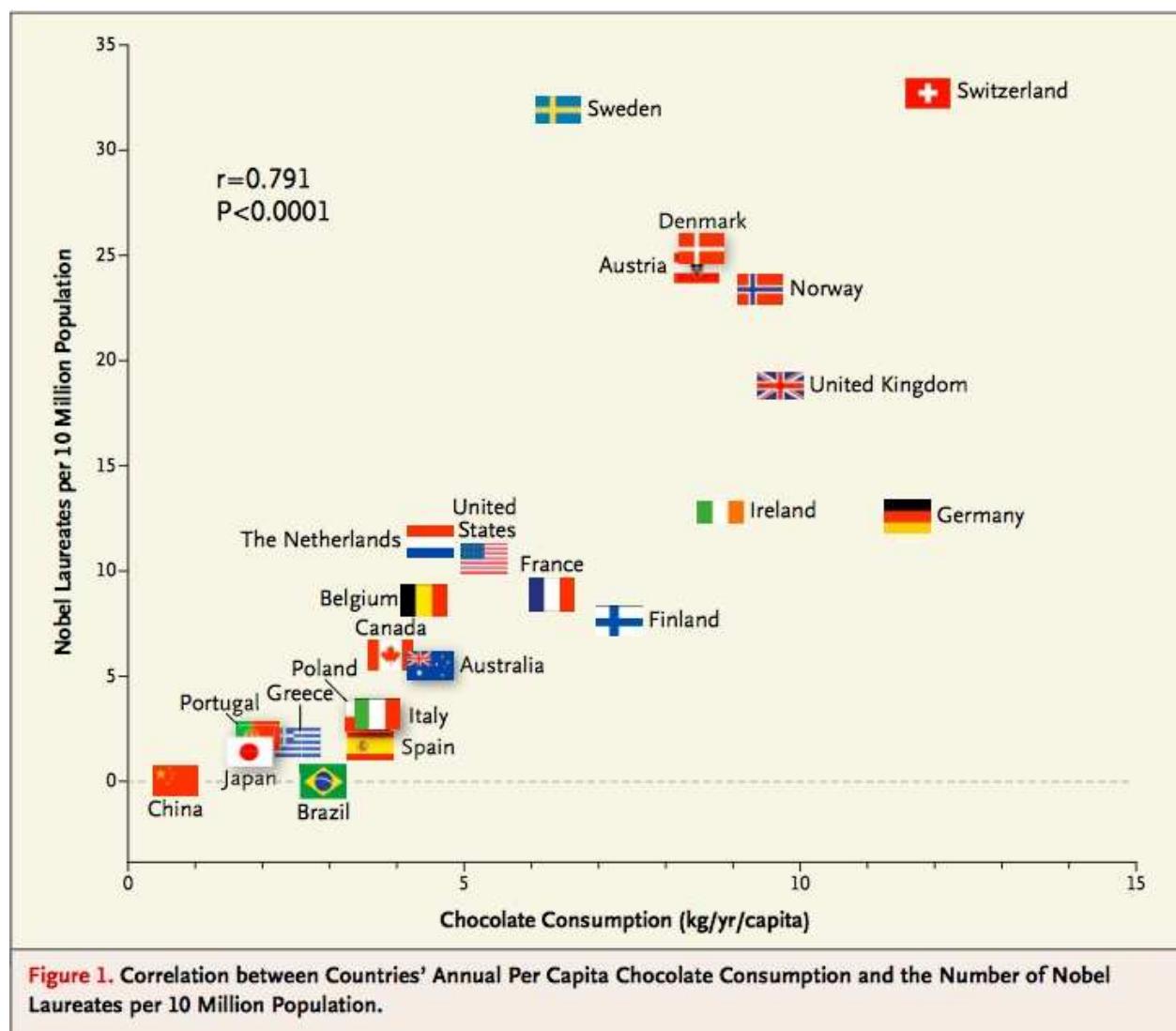
```
0.98475584110674341
```

It is important to note that this does not reflect the strength of the relation between the Math and Critical Reading scores of students. States don't take tests – students do. The data in the table have been created by lumping all the students in each state into a single point at the average values of the two variables in that state. But not all students in the state will be at that point, as students vary in their performance. If you plot a point for each student instead of just one for each state, there will be a cloud of points around each point in the figure above. The overall picture will be more fuzzy. The correlation between the Math and Critical Reading scores of the students will be lower than the value calculated based on state averages.

Correlations based on aggregates and averages are called *ecological correlations* and are frequently reported. As we have just seen, they must be interpreted with care.

## Serious or tongue-in-cheek?

In 2012, a [paper](#) in the respected New England Journal of Medicine examined the relation between chocolate consumption and Nobel Prizes in a group of countries. The [Scientific American](#) responded seriously; [others](#) were more relaxed. The paper included the following graph:



# Regression

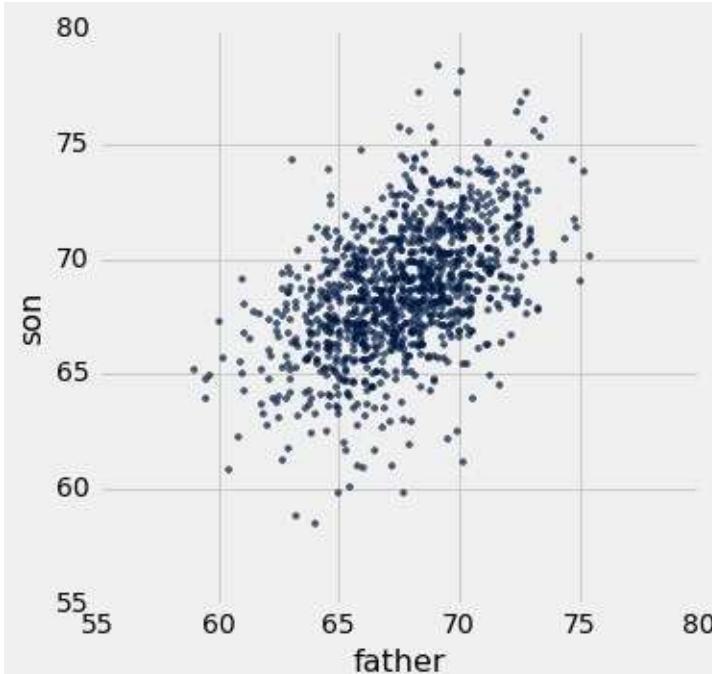
[Interact](#)

## The regression line

The concepts of correlation and the "best" straight line through a scatter plot were developed in the late 1800's and early 1900's. The pioneers in the field were Sir Francis Galton, who was a cousin of Charles Darwin, and Galton's protégé Karl Pearson. Galton was interested in eugenics, and was a meticulous observer of the physical traits of parents and their offspring. Pearson, who had greater expertise than Galton in mathematics, helped turn those observations into the foundations of mathematical statistics.

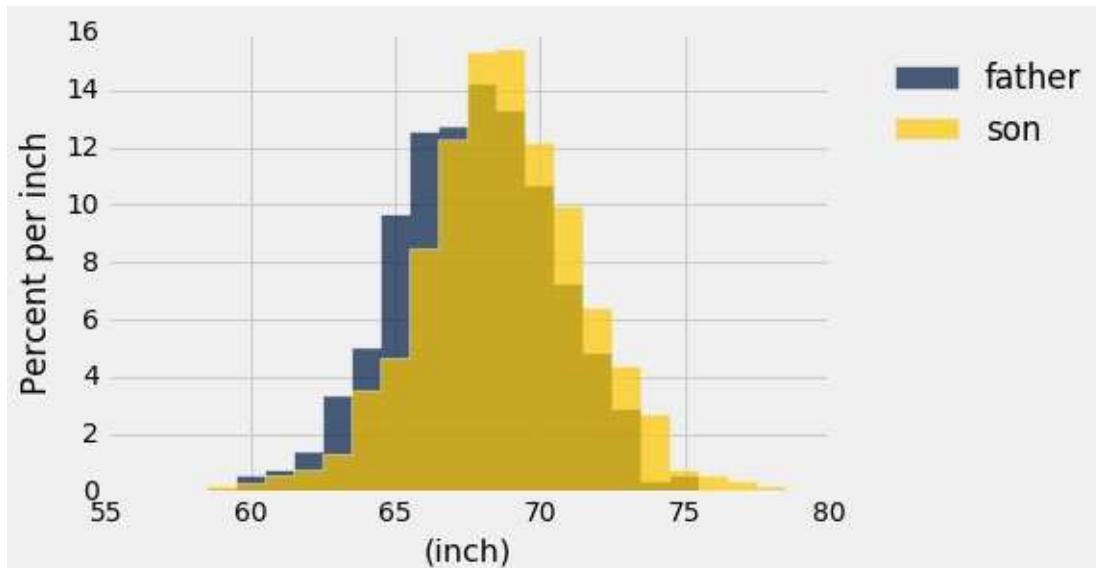
The scatter plot below is of a famous dataset collected by Pearson and his colleagues in the early 1900's. It consists of the heights, in inches, of 1,078 pairs of fathers and sons. The `scatter` method of a table takes an optional argument `s` to set the size of the dots.

```
heights = Table.read_table('heights.csv')
heights.scatter('father', 'son', s=10)
```



Notice the familiar football shape with a dense center and a few points on the perimeter. This shape results from two bell-shaped distributions that are correlated. The heights of both fathers and sons have a bell-shaped distribution.

```
heights.hist(bins=np.arange(55.5, 80, 1), unit='inch')
```



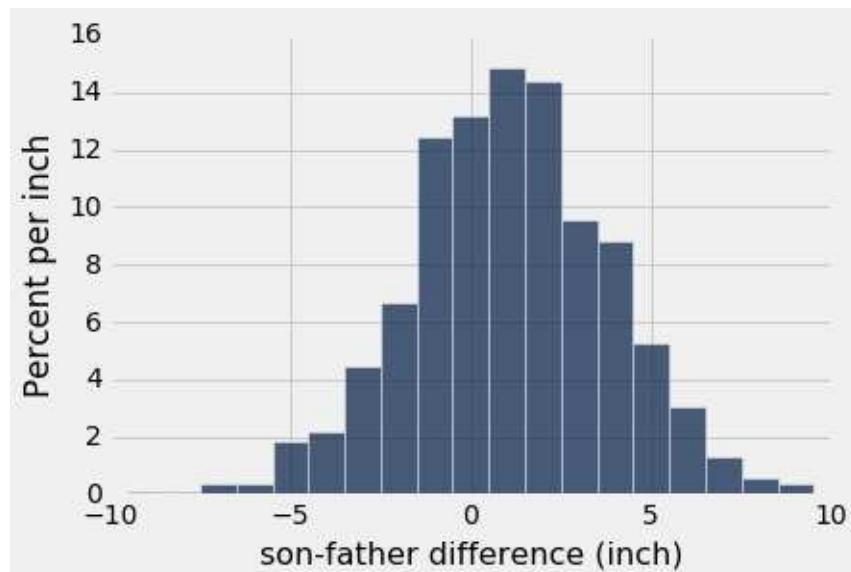
The average height of sons is about an inch taller than the average of fathers.

```
np.mean(heights.column('son')) - np.mean(heights.column('father'))
```

```
0.99740259740261195
```

The difference in height between a son and a father varies as well, with the bulk of the distribution between -5 inches and +7 inches.

```
diffs = Table().with_column('son-father difference',
                            heights.column('son') - heights.column('father'))
diffs.hist(bins=np.arange(-9.5, 10, 1), unit='inch')
```



The correlation between the heights of the fathers and sons is about 0.5.

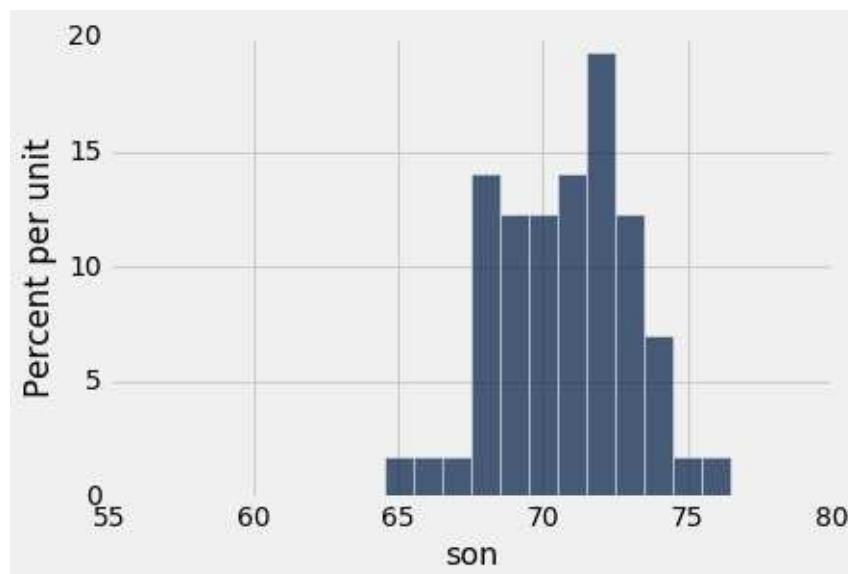
```
r = correlation(heights, 'father', 'son')
r
```

```
0.50116268080759108
```

## The regression effect

For fathers around 72 inches in height, we might expect their sons to be tall as well. The histogram below shows the height of all sons of 72-inch fathers.

```
six_foot_fathers = heights.where(np.round(heights.column('father')) == 72)
six_foot_fathers.hist('son', bins=np.arange(55.5, 80, 1))
```



Most (68%) of the sons of these 72-inch fathers are *less than* 72 inches tall, even though sons are an inch taller than fathers on average!

```
np.count_nonzero(six_foot_fathers.column('son') < 72) / six_foot_fa
```

◀

▶

0.6842105263157895

In fact, the average height of a son of a 72-inch father is less than 71 inches. The sons of tall fathers are simply not as tall in this sample.

```
np.mean(six_foot_fathers.column('son'))
```

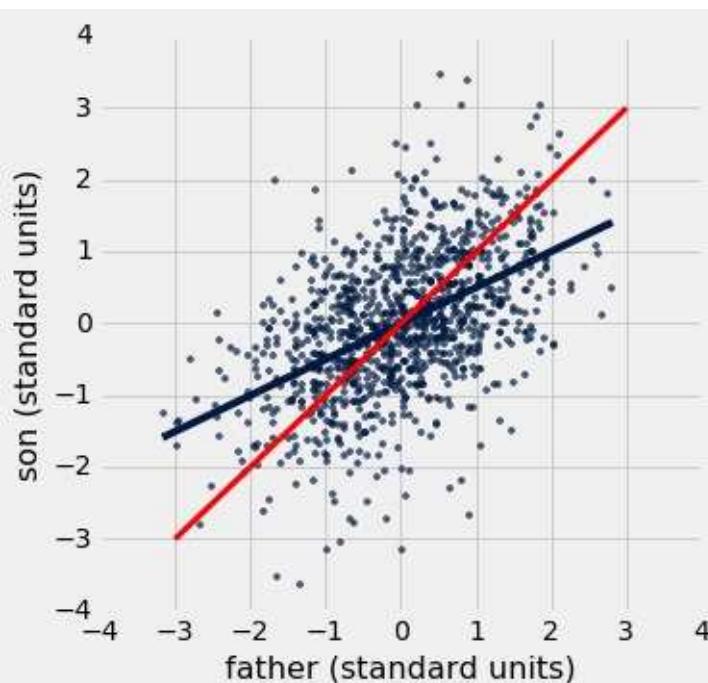
70.728070175438603

This fact was noticed by Galton, who had been hoping that exceptionally tall fathers would have sons who were just as exceptionally tall. However, the data were clear, and Galton realized that the tall fathers have sons who are not quite as exceptionally tall, on average. Frustrated, Galton called this phenomenon "regression to mediocrity."

Galton also noticed that exceptionally short fathers had sons who were somewhat taller relative to their generation, on average. In general, individuals who are away from average on one variable are expected to be not quite as far away from average on the other. This is called the *regression effect*.

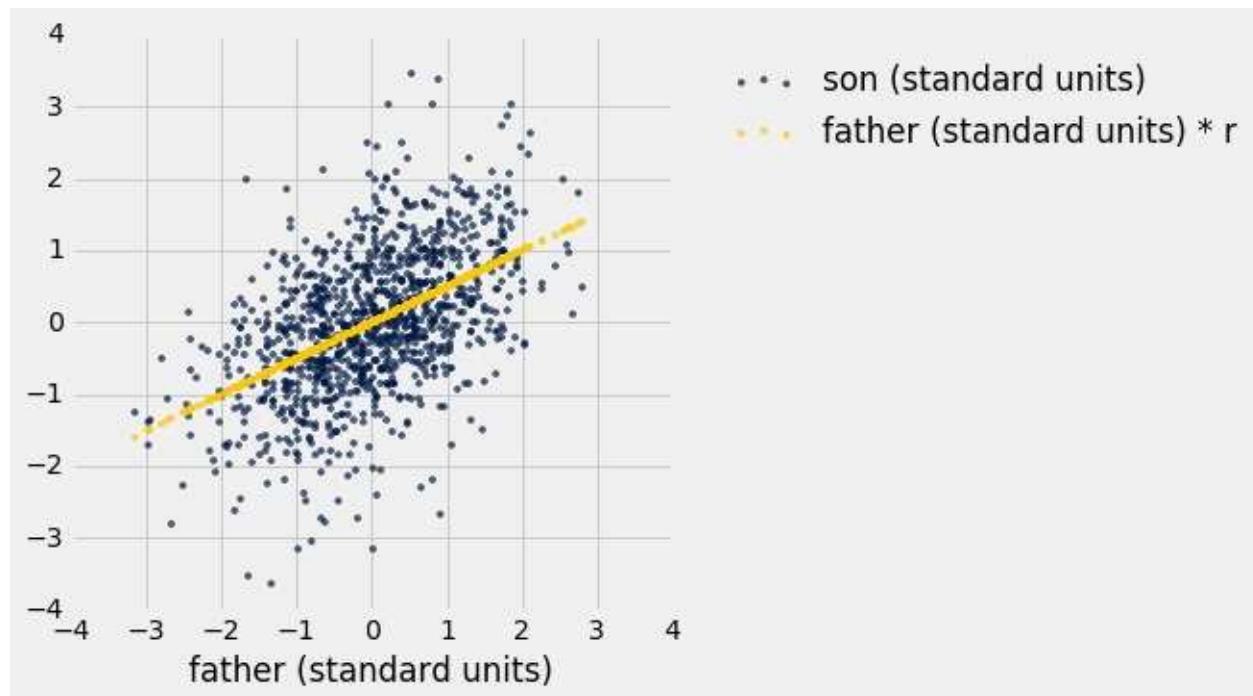
The figure below shows the scatter plot of the data when both variables are measured in standard units. The red line shows equal standard units and has a slope of 1, but it does not match the angle of the cloud of points. The blue line, which does follow the angle of the cloud, is called the *regression line*, named for the "regression to mediocrity" it predicts.

```
heights_standard = Table().with_columns([
    'father (standard units)', standard_units(heights['father'])
    'son (standard units)', standard_units(heights['son'])
])
heights_standard.scatter(0, fit_line=True, s=10)
_ = plots.plot([-3, 3], [-3, 3], color='r', lw=3)
```



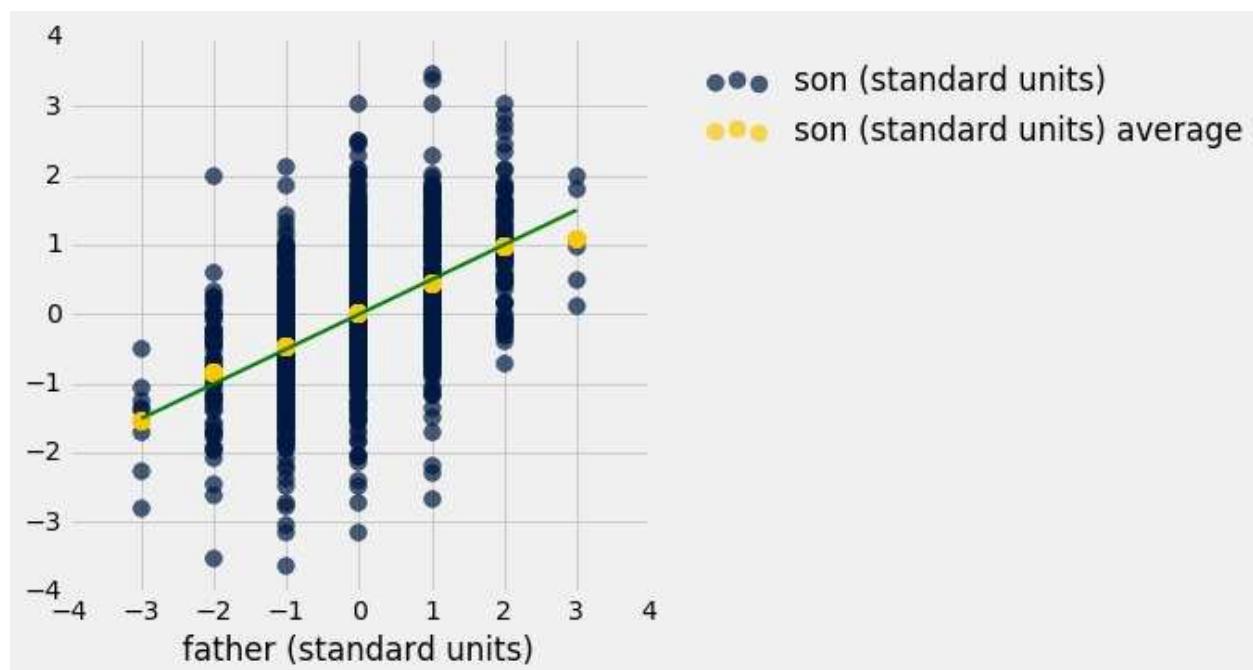
The `scatter` method of a `Table` draws this regression line for us when called with `fit_line=True`. This line passes through the result of multiplying father heights (in standard units) by  $r$ .

```
father_standard = heights_standard.column(0)
heights_standard.with_column('father (standard units) * r', father_
```



Another interpretation of this line is that it passes through average values for slices of the population of sons. To see this relationship, we can round the fathers each to the nearest unit, then average the heights of all sons associated with these rounded values. The green line is the regression line for these data, and it passes close to all of the yellow points, which are mean heights of sons (in standard units).

```
rounded = heights_standard.with_column('father (standard units)', r
rounded.join('father (standard units)', rounded.groupby(0, np.average
_ = plots.plot([-3, 3], [-3 * r, 3 * r], color='g', lw=2)
```



Karl Pearson used the observation of the regression effect in the data above, as well as in other data provided by Galton, to develop the formal calculation of the correlation coefficient  $r$ . That is why  $r$  is sometimes called *Pearson's correlation*.

## The regression line in original units

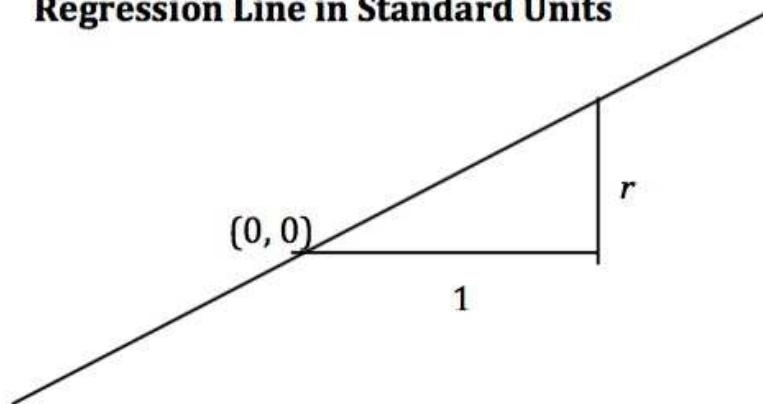
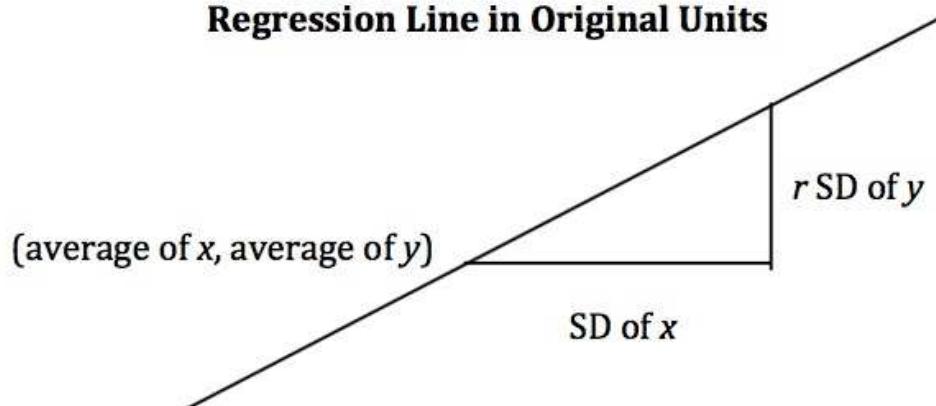
As we saw in the last section for football shaped scatter plots, when the variables  $x$  and  $y$  are measured in standard units, the best straight line for estimating  $y$  based on  $x$  has slope  $r$  and passes through the origin. Thus the equation of the regression line can be written as:

$$(y \text{ in standard units}) = r \times (x \text{ in standard units})$$

That is,

$$\frac{\text{estimate of } y - \text{ average of } y}{\text{SD of } y} = r$$
$$\times \frac{\text{the given } x - \text{ average of } x}{\text{SD of } x}$$

The equation can be converted into the original units of the data, either by rearranging this equation algebraically, or by labeling some important features of the line both in standard units and in the original units.

**Regression Line in Standard Units****Regression Line in Original Units****Calculation of the slope and intercept** 

The regression line is also commonly expressed as a slope and an intercept, where an estimate for  $y$  is computed from an  $x$  value using the equation

$$y = \text{slope} \times x + \text{intercept}$$

The calculations of the slope and intercept of the regression line can be derived from the equation above.

```

def slope(table, x, y):
    r = correlation(table, x, y)
    return r * np.std(table.column(y))/np.std(table.column(x))

def intercept(table, x, y):
    a = slope(table, x, y)
    return np.mean(table.column(y)) - a * np.mean(table.column(x))

[slope(heights, 'father', 'son'), intercept(heights, 'father', 'son')]

```

[0.51400591254559247, 33.892800540661682]

It is worth noting that the intercept of approximately 33.89 inches is *not* intended as an estimate of the height of a son whose father is 0 inches tall. There is no such son and no such father. The intercept is merely a geometric or algebraic quantity that helps define the line. In general, it is not a good idea to *extrapolate*, that is, to make estimates outside the range of the available data. It is certainly not a good idea to extrapolate as far away as 0 is from the heights of the fathers in the study.

It is also worth noting that the slope *is not* `r`! Instead, it is `r` multiplied by the ratio of standard deviations.

```
correlation(heights, 'father', 'son')
```

0.50116268080759108

## Fitted values

We can also estimate every son in the data using the slope and intercept. The estimated values of  $y$  are called the *fitted values*. They all lie on a straight line. To calculate them, take a son's height, multiply it by the slope of the regression line, and add the intercept. In other words, calculate the height of the regression line at the given value of  $x$ .

```

def fit(table, x, y):
    """Return the height of the regression line at each x value."""
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * table.column(x) + b

fitted = heights.with_column('son (fitted)', fit(heights, 'father',
fitted

```

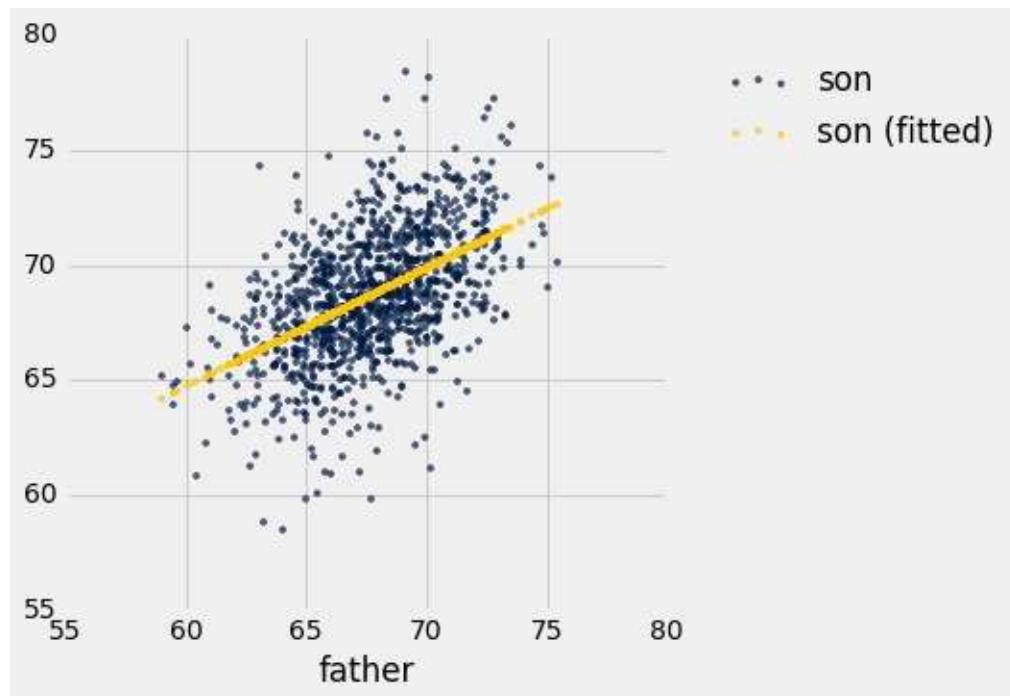


father	son	son (fitted)
65	59.8	67.3032
63.3	63.2	66.4294
65	63.3	67.3032
65.8	62.8	67.7144
61.1	64.3	65.2986
63	64.2	66.2752
65.4	64.1	67.5088
64.7	64	67.149
66.1	64.6	67.8686
67	64	68.3312

... (1068 rows omitted)

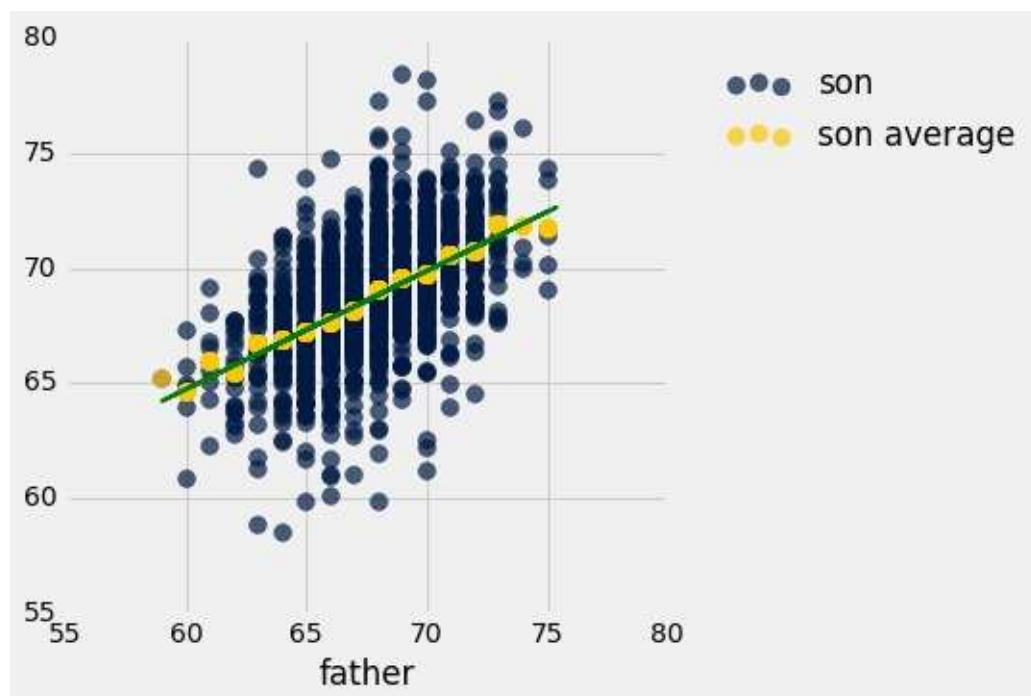
In original units, we can see that these fitted values fall on a line.

```
fitted.scatter(0, s=10)
```



Moreover, this line (in green below), passes near the average heights of sons for slices of the population. In this case, we round the heights of fathers to the nearest inch to construct the slices.

```
rounded = heights.with_column('father', np.round(heights.column('father'))).group_by('father').average().set_index('father')
_ = plots.plot(father, son, color='darkblue', size=10)
_ = plots.plot(father, son_average, color='darkgreen', size=10)
```





# Prediction

## Interact

In the last section, we developed the concepts of correlation and regression as ways to describe data. We will now see how these concepts can become powerful tools for prediction, when used appropriately. In particular, given paired observations of two quantities and some additional observations of only the first quantity, we can infer typical values for the second quantity.

## Assumptions of randomness: a "regression model"

Prediction is possible if we believe that a scatter plot reflects the underlying relation between the two variables being plotted, but does not specify the relation completely. For example, a scatter plot of the heights of fathers and sons shows us the precise relation between the two variables in one particular sample of men; but we might wonder whether that relation holds true, or almost true, among all fathers and sons in the population from which the sample was drawn, or indeed among fathers and sons in general.

As always, inferential thinking begins with a careful examination of the assumptions about the data. Sets of assumptions are known as *models*. Sets of assumptions about randomness in roughly linear scatter plots are called *regression models*.

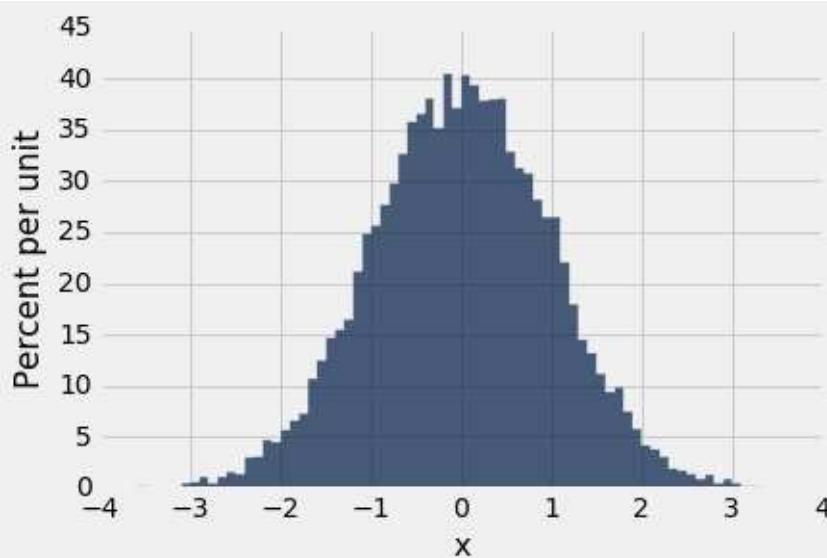
In brief, such models say that the underlying relation between the two variables is perfectly linear; this straight line is the *signal* that we would like to identify. However, we are not able to see the line clearly, because there is additional variability in the data beyond the relation. What we see are points that are scattered around the line. For each of the points, the linear signal that relates the two variables has been combined with some other source of variability that has nothing to do with the relation at all. This *random noise* obscures the linear signal, and it is our inferential goal to identify the signal despite the noise.

In order to make these statements more precise, we will use the `np.random.normal` function, which generates samples from a normal distribution with 0 mean and 1 standard deviation. These samples correspond to a bell-shaped distribution expressed in standard units.

```
np.random.normal()
```

```
-0.6423537870160524
```

```
samples = Table('x')
for i in np.arange(10000):
    samples.append([np.random.normal()])
samples.hist(0, bins=np.arange(-4, 4, 0.1))
```



The regression model specifies that the points in a scatter plot, measured in standard units, are generated at random as follows.

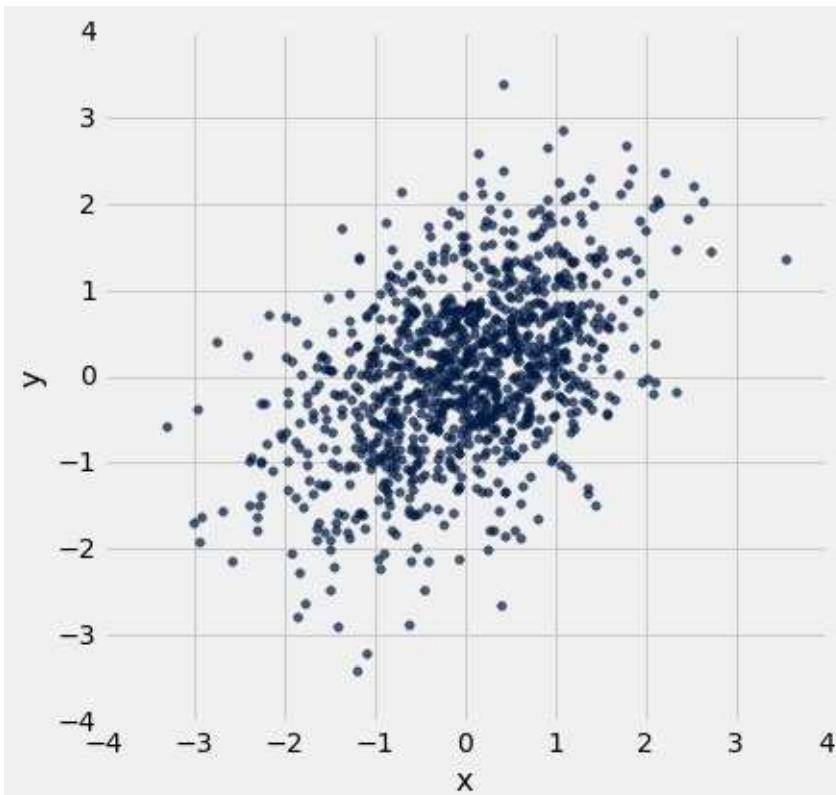
- The  $x$  are sampled from the normal distribution.
- For each  $x$ , its corresponding  $y$  is sampled using the `signal_and_noise` function below, which takes  $x$  and the correlation coefficient  $r$ .

```
def signal_and_noise(x, r):
    return r * x + np.random.normal() * (1-r**2)**0.5
```

For example, if we choose  $r$  to be a half, then the following scatter diagram might result.

```
def regression_model(r, sample_size):
    pairs = Table(['x', 'y'])
    for i in np.arange(sample_size):
        x = np.random.normal()
        y = signal_and_noise(x, r)
        pairs.append([x, y])
    return pairs

regression_model(0.5, 1000).scatter('x', 'y')
```



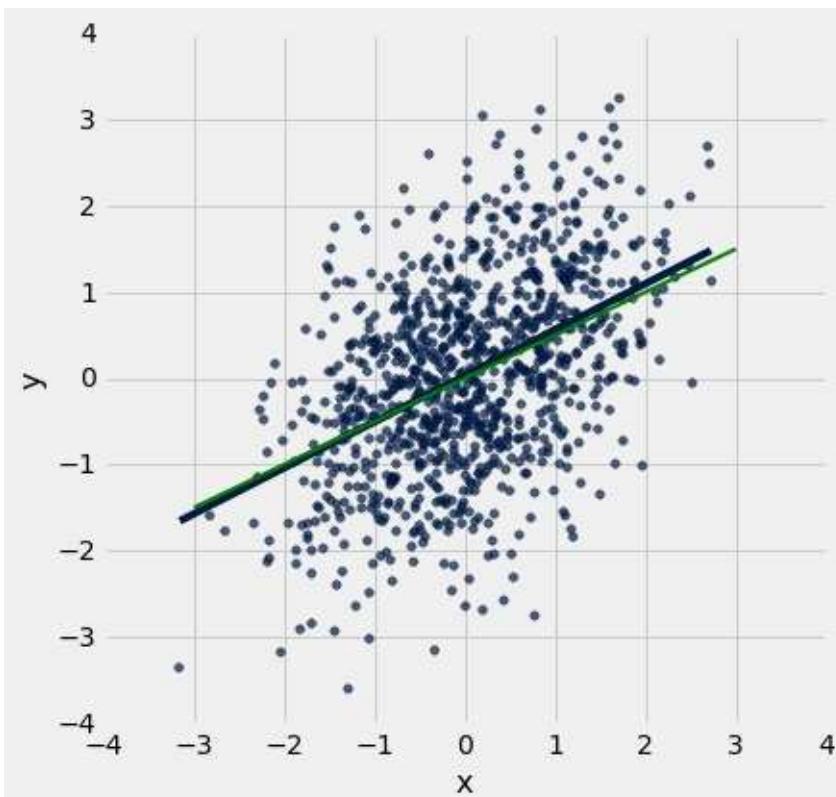
Based on this scatter plot, how should we estimate the true line? The best line that we can put through a scatter plot is the regression line. So the regression line is a natural estimate of the true line.

The simulation below draws both the true line used to generate the data (green) and the regression line (blue) found by analyzing the pairs that were generated.

```
def compare(true_r, sample_size):
    pairs = regression_model(true_r, sample_size)
    estimated_r = correlation(pairs, 'x', 'y')
    pairs.scatter('x', 'y', fit_line=True)
    plt.plot([-3, 3], [-3 * true_r, 3 * true_r], color='g', lw=2)
    print("The true r is ", true_r, " and the estimated r is ", est

compare(0.5, 1000)
```

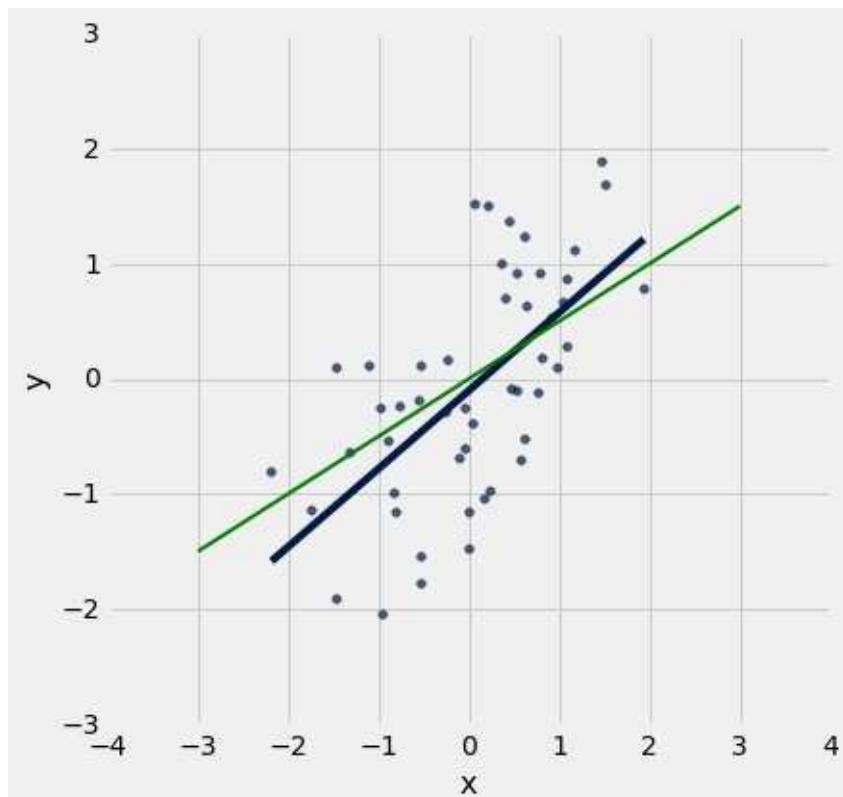
```
The true r is  0.5  and the estimated r is  0.459672161067
```



With 1000 samples, we see that the true line and the regression line are quite similar. With fewer samples, the regression estimate of the true line that generated the data may be quite different.

```
compare(0.5, 50)
```

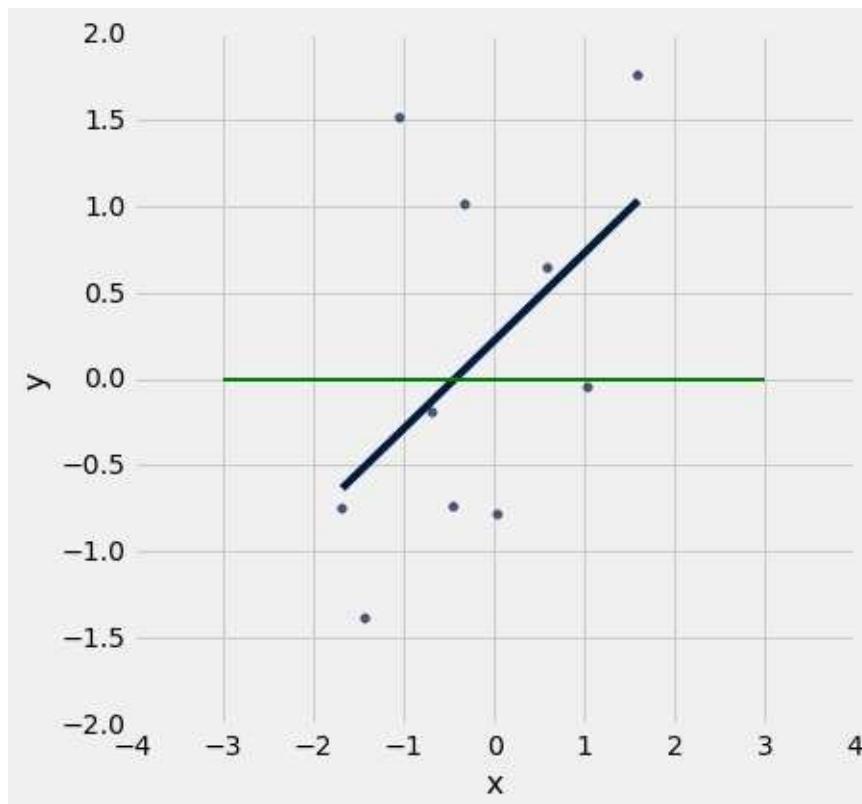
```
The true r is  0.5  and the estimated r is  0.627329170668
```



With very few samples, the regression line may have a clear positive or negative slope, even when the true relationship has a correlation of 0.

```
compare(0, 10)
```

```
The true r is 0 and the estimated r is 0.501582566452
```



With real data, we will never observe the true line. In fact, regression is often applied when we are uncertain whether the true relation between two quantities is linear at all. What the simulation shows that if the regression model looks plausible, and we have a large sample, then regression line is a good approximation to the true line.

## Predictions

Here is an example where regression model can be used to make predictions.

The data are a subset of the information gathered in a randomized controlled trial about treatments for Hodgkin's disease. Hodgkin's disease is a cancer that typically affects young people. The disease is curable but the treatment is very harsh. The purpose of the trial was to come up with dosage that would cure the cancer but minimize the adverse effects on the patients.

This table `hodgkins` contains data on the effect that the treatment had on the lungs of 22 patients. The columns are:

- Height in cm
- A measure of radiation to the mantle (neck, chest, under arms)
- A measure of chemotherapy
- A score of the health of the lungs at baseline, that is, at the start of the treatment; higher scores correspond to more healthy lungs
- The same score of the health of the lungs, 15 months after treatment

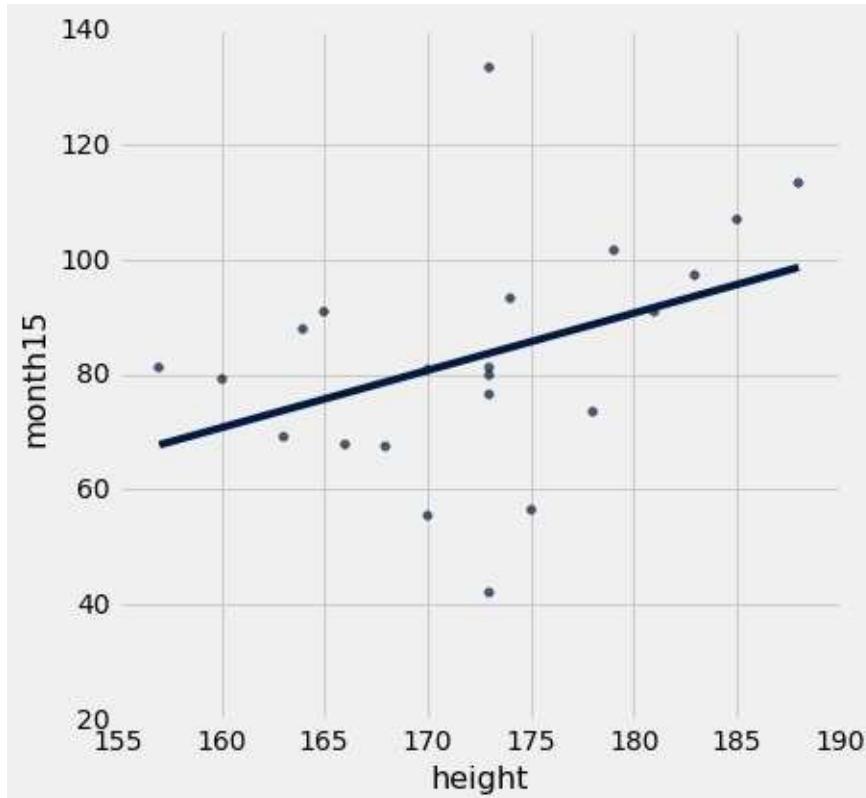
```
hodgkins = Table.read_table('hodgkins.csv')
hodgkins.show()
```

height	rad	chemo	base	month15
164	679	180	160.57	87.77
168	311	180	98.24	67.62
173	388	239	129.04	133.33
157	370	168	85.41	81.28
160	468	151	67.94	79.26
170	341	96	150.51	80.97
163	453	134	129.88	69.24
175	529	264	87.45	56.48
185	392	240	149.84	106.99
178	479	216	92.24	73.43
179	376	160	117.43	101.61
181	539	196	129.75	90.78
173	217	204	97.59	76.38
166	456	192	81.29	67.66
170	252	150	98.29	55.51
165	622	162	118.98	90.92
173	305	213	103.17	79.74
174	566	198	94.97	93.08
173	322	119	85	41.96
173	270	160	115.02	81.12
183	259	241	125.02	97.18
188	238	252	137.43	113.2

It is evident that the patients' lungs were less healthy 15 months after the treatment than at baseline. At 36 months, they did recover most of their lung function, but those data are not part of this table.

The scatter plot below shows that taller patients had higher scores at 15 months.

```
hodgkins.scatter('height', 'month15', fit_line=True)
```



The scatter plot looks roughly linear. Let us assume that the regression model holds. Under that assumption, the regression line can be used to predict the 15-month score of a new patient, based on the patient's height. The prediction will be good provided the assumptions of the regression model are justified for these data, and provided the new patient is similar to those in the study.

The function `regress` gives us the slope and intercept of the regression line. To predict the 15-month score of a new patient whose height is  $x$  cm, we use the following calculation:

Predicted 15-month score of a patient who is  $x$  inches tall = slope  $\times x +$  intercept

The predicted 15-month score of a patient who is 173 cm tall is 83.66 cm, and that of a patient who is 163 cm tall is 73.69 cm.

```
# slope and intercept
a = slope(hodgkins, 'height', 'month15')
b = intercept(hodgkins, 'height', 'month15')
```

```
# New patient, 173 cm tall
# Predicted 15-month score:

a * 173 + b
```

83.65699801460444

```
# New patient, 163 cm tall
# Predicted 15-month score:

a * 163 + b
```

73.694360467072741

The logic of regression prediction can be wrapped in a function `predict` that takes a table, the two columns of interest, and some new value for `x`. It predicts the average value for `y`.

```
def predict(t, x, y, new_x_value):
    a = slope(t, x, y)
    b = intercept(t, x, y)
    return a * new_x_value + b

predict(hodgkins, 'height', 'month15', 163)
```

73.694360467072741

## Variability of the Prediction¶

As data scientists working under the regression model, we must always remain aware that a sample might have been different. Had the sample been different, the regression line would have been different too, and so would our prediction.

To understand how much a prediction might vary, we can select different samples from the data we have. Below, we select only 15 of the 22 rows of the `hodgkins` table at random without replacement, then makes a prediction.

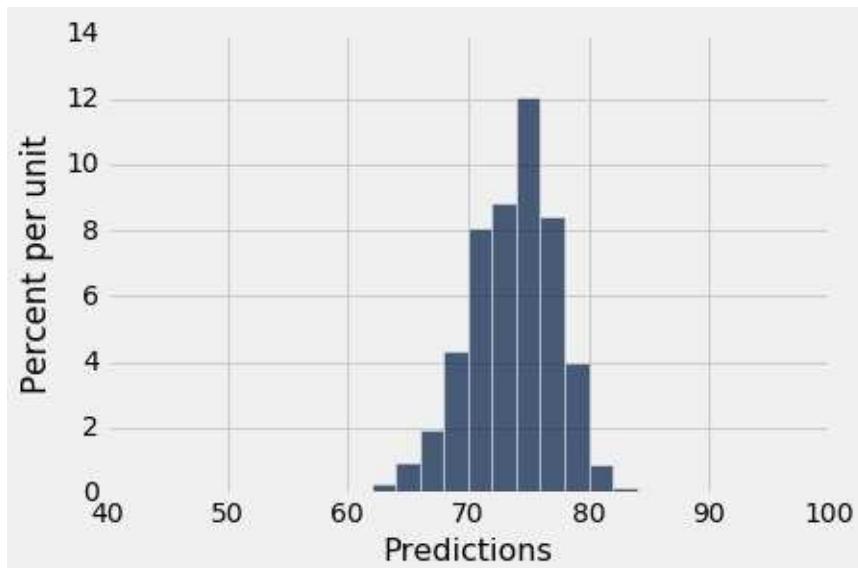
```
predict(hodgkins.sample(15), 'height', 'month15', 163)
```

```
71.653856125189805
```

The prediction is different, but how much might we expect it to differ? From a single sample, it is not possible to answer this question. The simulation below draws 1000 samples of 15 rows each, then draws a histogram of the predicted values using each sample.

```
def sample_predict(new_x_value):
    predictions = Table(['Predictions'])
    for i in np.arange(1000):
        predicted = predict(hodgkins.sample(15), 'height', 'month15', new_x_value)
        predictions.append([predicted])
    predictions.hist(0, bins=np.arange(40, 96, 2))

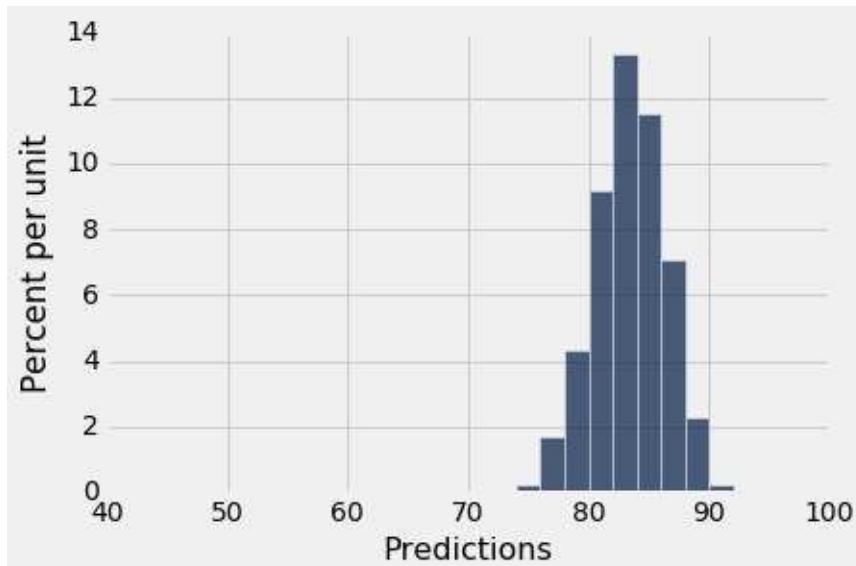
sample_predict(163)
```



Using only 15 rows instead of all 22, the prediction could have come out differently from 73.69. This histogram shows how different the prediction might be. Almost all predictions fall between 64 and 82, but a few are outside of this range.

For an  $x$  value that is closer to the mean of all heights, the predictions from 15 rows are more tightly clustered.

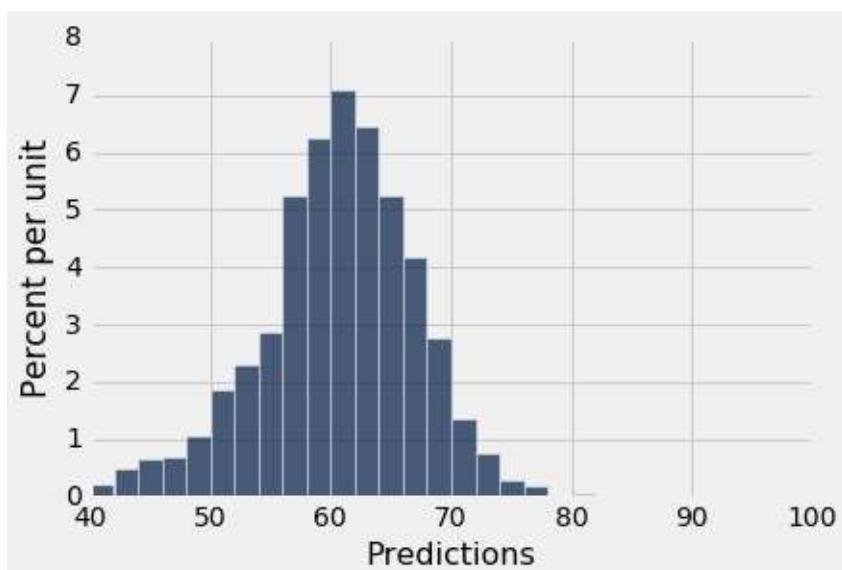
```
sample_predict(173)
```



In this histogram of the predictions for a height of 173, we see that almost all predictions fall between 76 and 90. The narrow range 80 to 86 accounts for nearly 70% of the estimates.

Predictions of values outside the typical observations of height vary wildly. A data scientist is rarely justified in extrapolating a linear trend beyond the range of values observed.

```
sample_predict(150)
```



# Higher-Order Functions

## Interact

The functions we have defined so far compute their outputs directly from their inputs. Their return values can be computed from their arguments. However, some functions require additional data to determine their return values.

Let's start with a simple example. Consider the following three functions that all scale their input by a constant factor.

```
def double(x):
    return 2 * x

def triple(x):
    return 3 * x

def halve(x):
    return 0.5 * x

[double(4), triple(4), halve(4), double(5), triple(5), halve(5)]
```

```
[8, 12, 2.0, 10, 15, 2.5]
```

Rather than defining each of these functions with a separate `def` statement, it is possible to generate each of them dynamically by passing in a constant to a generic `scale_by` function.

```
def scale_by(a):
    def scale(x):
        return a * x
    return scale

double = scale_by(2)
triple = scale_by(3)
halve = scale_by(0.5)

[double(4), triple(4), halve(4), double(5), triple(5), halve(5)]
```

```
[8, 12, 2.0, 10, 15, 2.5]
```

The body of the `scale_by` function actually defines a new function, called `scale`, and then returns it. The return value of `scale_by` is a function that takes a number `x` and multiplies it by `a`. Because `scale_by` is a function that returns a function, it is called a *higher-order function*.

The purpose of higher-order functions is to create functions that combine data with behavior. Above, the `double` function is created by combining the data value `2` with the behavior of scaling by a constant.

A `def` statement within another `def` statement behaves in the same way as any other `def` statement *except* that:

1. The name of the inner function is only accessible within the outer function's body. For example, it is not possible to use the name `scale` once `scale_by` has returned.
2. The body of the inner function can refer to the arguments and names within the outer function. So, for example, the body of the inner `scale` function can refer to `a`, which is an argument of the outer function `scale_by`.

The final line of the outer `scale_by` function is `return scale`. This line returns the *function* that was just defined. The purpose of returning a function is to give it a name. The line

```
double = scale_by(2)
```

gives the name `double` to the particular instance of the `scale` function that has `a` assigned to the number 2. Notice that it is possible to keep multiple instances of the `scale` function around at once, all with different names such as `double` and `triple`, and Python keeps track of which value for `a` to use each time one of these functions is called.

## Example: Standard Units

We have seen how to convert a list of numbers to standard units by subtracting the mean and dividing by the standard deviation.

```
def standard_units(any_numbers):
    "Convert any array of numbers to standard units."
    return (any_numbers - np.mean(any_numbers))/np.std(any_numbers)
```

This function can convert any list of numbers to standard units.

```
observations = [3, 4, 2, 4, 3, 5, 1, 6]
np.round(standard_units(observations), 3)
```

```
array([-0.333,  0.333, -1.     ,  0.333, -0.333,  1.     , -1.667,  1.667])
```

How would we answer the question, "what is the number 8 (in original units) converted to standard units?"

Adding 8 to the list of `observations` and re-computing `standard_units` does *not* answer this question, because adding an observation changes the scale!

```
observations_with_eight = [3, 4, 2, 4, 3, 5, 1, 6, 8]
standard_units(observations_with_eight)
```

```
array([-0.5,  0. , -1. ,  0. , -0.5,  0.5, -1.5,  1. ,  2. ])
```

If instead we want to maintain the original scale, we need a function that remembers the original mean and standard deviation. This scenario, where both data and behavior are required to arrive at the answer, is a natural fit for a higher-order function.

```
def converter_to_standard_units(any_numbers):
    """Return a function that converts to the standard units of any
    mean = np.mean(any_numbers)
    sd = np.std(any_numbers)
    def convert(a_number_in_original_units):
        return (a_number_in_original_units - mean) / sd
    return convert
```

Now, we can use the original list of numbers to create the conversion function. The mean and standard deviation are stored within the function that is returned, called `to_su` below. What's more, these numbers are only computed once, no matter how many times we call the function.

```
observations = [3, 4, 2, 4, 3, 5, 1, 6]
to_su = converter_to_standard_units(observations)
[to_su(2), to_su(3), to_su(8)]
```

```
[-1.0, -0.3333333333333331, 3.0]
```

A complementary function that returns a converter from standard units to original units involves multiplying by the standard deviation, then adding the mean.

```
def converter_from_standard_units(any_numbers):
    """Return a function that converts from the standard units of a
    mean = np.mean(any_numbers)
    sd = np.std(any_numbers)
    def convert(in_standard_units):
        return in_standard_units * sd + mean
    return convert

from_su = converter_from_standard_units(observations)
from_su(3)
```

```
8.0
```

It should be the case that any number converted to standard units and back, using the same list to create both functions.

```
from_su(to_su(11))
```

```
11.0
```

## Example: Prediction

Previously, in order to compute the fitted values for a regression, we first computed the slope and intercept of the regression line. As an alternate implementation, for each  $x$ , we can convert  $x$  to standard units from the original units of  $x$  values, then multiply by  $r$ , then

convert the result to the original units of `y` values. We can write down this process as a higher-order function.

```
def regression_estimator(table, x, y):
    """Return an estimator for y as a function of x."""
    to_su = converter_to_standard_units(table.column(x))
    from_su = converter_from_standard_units(table.column(y))
    r = correlation(table, x, y)
    def estimate(any_x):
        return from_su(r * to_su(any_x))
    return estimate
```

The following dataset of mothers and babies was collected from the Kaiser Hospital in Oakland, California.

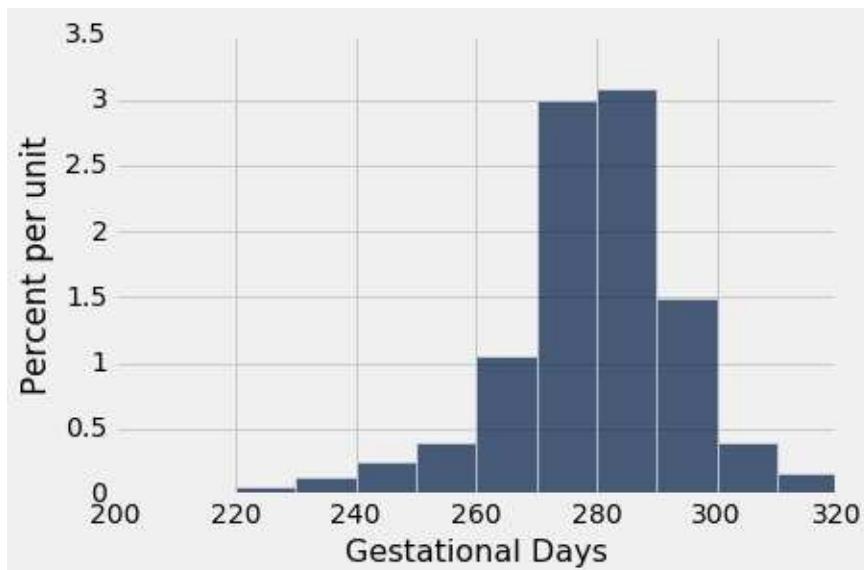
```
baby = Table.read_table('baby.csv')
baby
```

Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
120	284	27	62	100	False
113	282	33	64	135	False
128	279	28	64	115	True
108	282	23	67	125	True
136	286	25	62	93	False
138	244	33	62	178	False
132	245	23	65	140	False
120	289	25	62	125	False
143	299	30	66	136	True
140	351	27	68	120	False

... (1164 rows omitted)

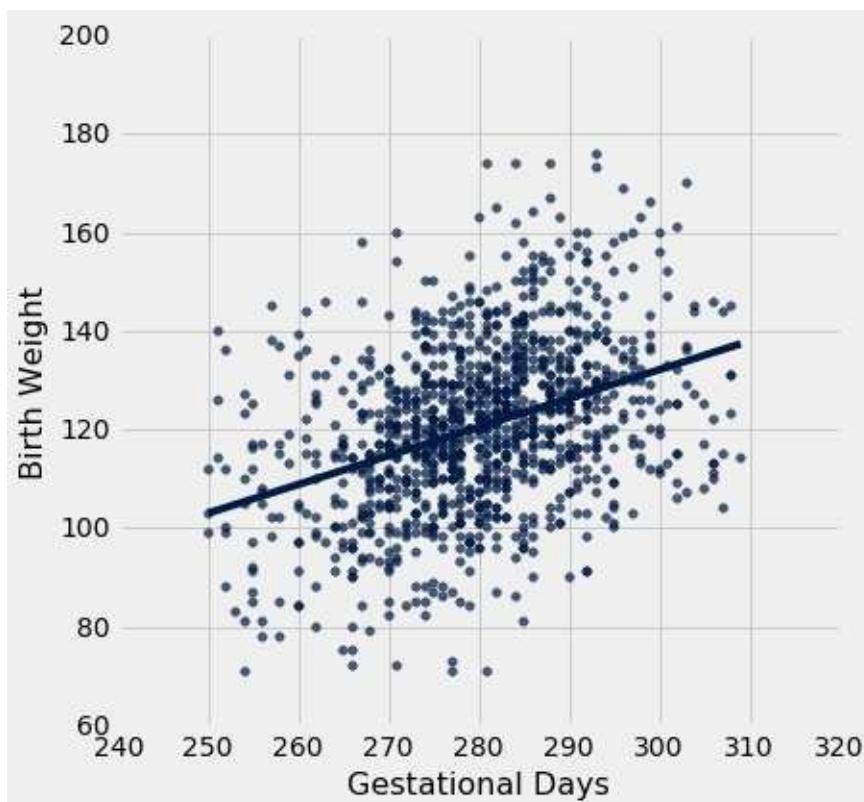
Gestational days are the number of days that the mother is pregnant. Most pregnancies last between 250 and 310 days.

```
baby.hist(1, bins=np.arange(200, 330, 10))
```



For this collection of typical births, gestational days and birth weight appear to be linearly related.

```
typical = baby.where(np.logical_and(baby.column(1) >= 250, baby.column(2) >= 60))
typical.scatter(1, 0, fit_line=True)
```



The function `average_weight` returns the average weight of babies born after a certain number of gestational days, using the regression line to predict this value. This function is defined within `regression_estimator` and returned. The returned value is bound to the name `average_weight` using an assignment statement.

```
average_weight = regression_estimator(typical, 1, 0)
```

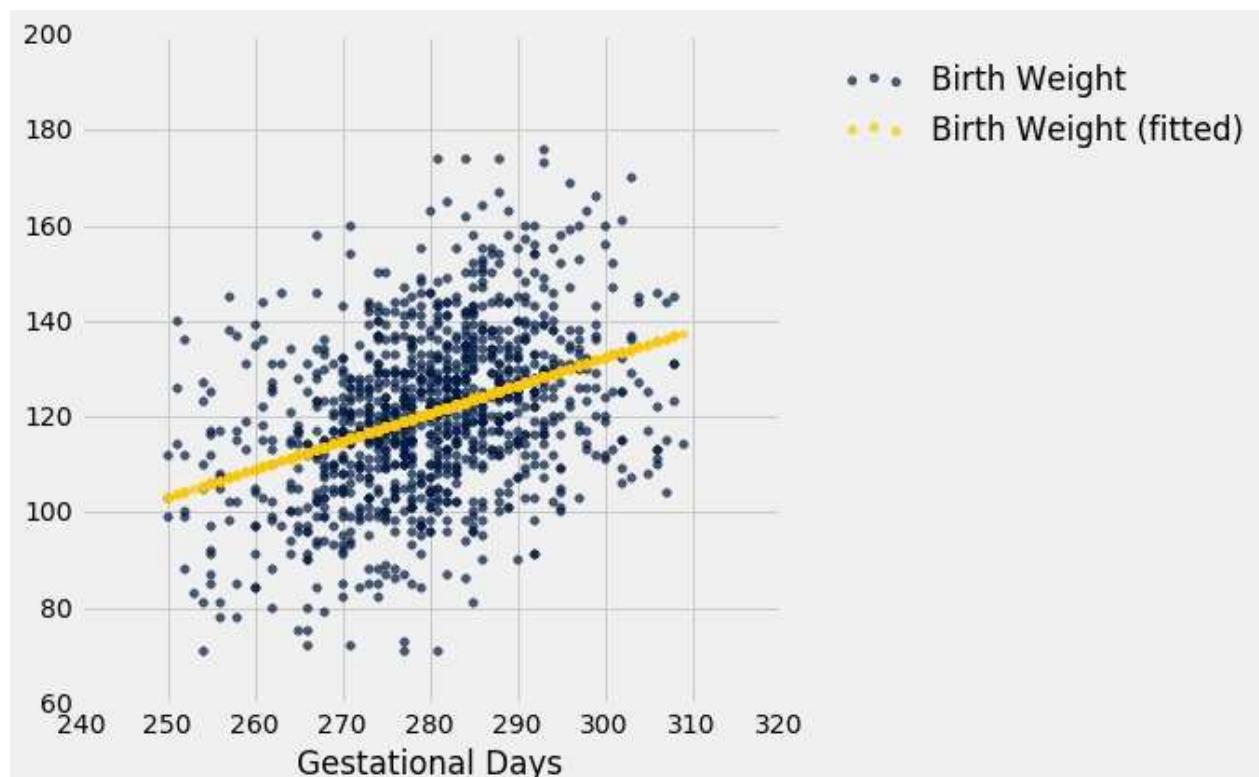
The `average_weight` function contains all the information needed to make birth weight predictions, but when it is called, it takes only a single argument: the gestational days. We avoid the need to pass the table and column labels into the `average_weight` function each time that we call it because those values were passed to `regression_estimator`, which defined the `average_weight` function in terms of those values.

```
average_weight(290)
```

```
126.33785752202166
```

This function is quite flexible. For instance, we can compute fitted values for all gestational days by applying it to the column of gestational days.

```
fitted = typical.apply(average_weight, 1)
typical.select([1, 0]).with_column('Birth Weight (fitted)', fitted)
```



Higher-order functions appear often in the context of prediction because estimating a value often involves both data to inform the prediction and a procedure that makes the prediction itself.

# Errors

[Interact](#)

## Error in the regression estimate

Though the average residual is 0, each individual residual is not. Some residuals might be quite far from 0. To get a sense of the amount of error in the regression estimate, we will start with a graphical description of the sense in which the regression line is the "best".

Our example is a dataset that has one point for every chapter of the novel "Little Women." The goal is to estimate the number of characters (that is, letters, punctuation marks, and so on) based on the number of periods. Recall that we attempted to do this in the very first lecture of this course.

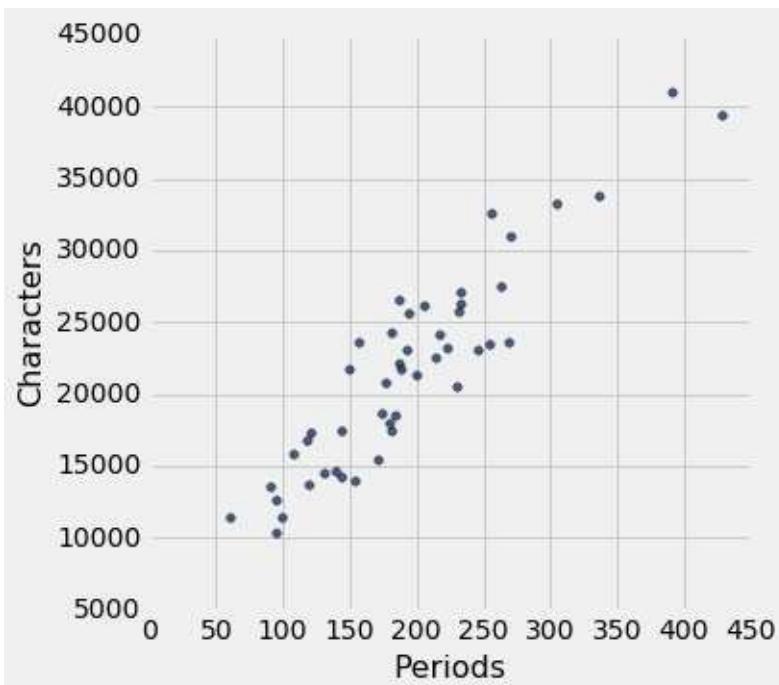
```
little_women = Table.read_table('little_women.csv')
little_women.show(3)
```

Characters	Periods
21759	189
22148	188
20558	231

... (44 rows omitted)

```
# One point for each chapter
# Horizontal axis: number of periods
# Vertical axis: number of characters (as in a, b, ", ?, etc; not p)

little_women.scatter('Periods', 'Characters')
```



```
correlation(little_women, 'Periods', 'Characters')
```

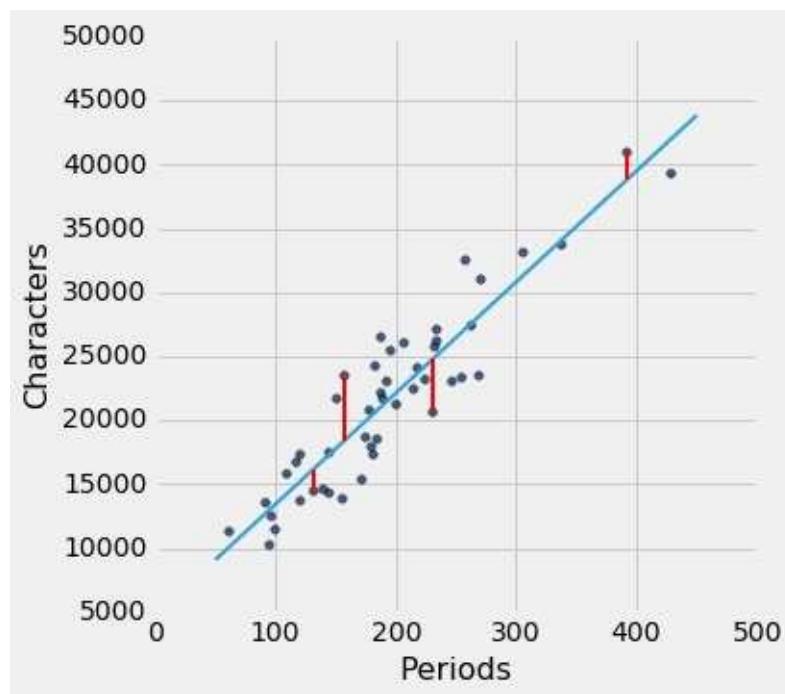
```
0.92295768958548163
```

The scatter plot is remarkably close to linear, and the correlation is more than 0.92.

The figure below shows the scatter plot and regression line, with four of the errors marked in red.

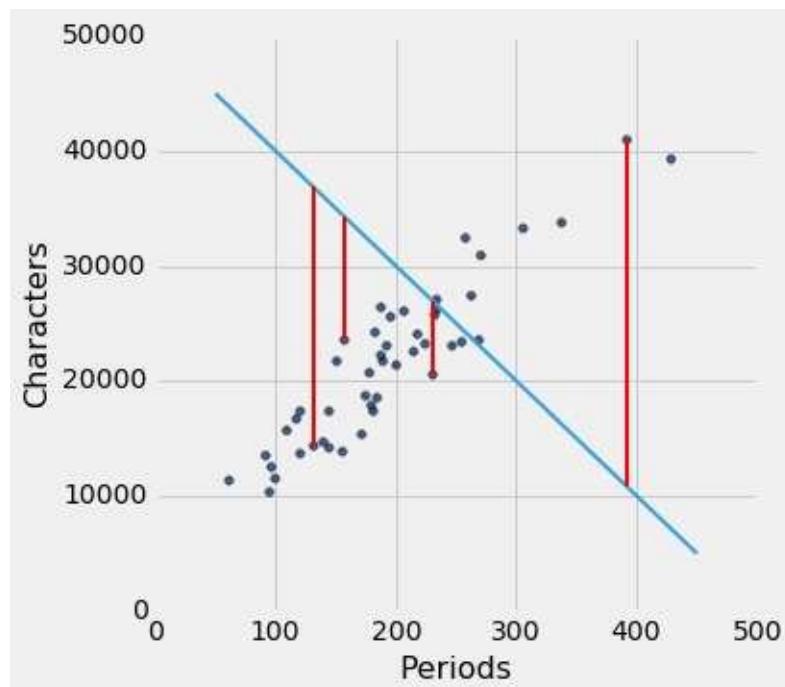
```
# Residuals: Deviations from the regression line
lw_slope = slope(little_women, 'Periods', 'Characters')
lw_intercept = intercept(little_women, 'Periods', 'Characters')
print('Slope:      ', np.round(lw_slope))
print('Intercept:', np.round(lw_intercept))
lw_errors(lw_slope, lw_intercept)
```

```
Slope:      87.0
Intercept: 4745.0
```



Had we used a different line to create our estimates, the errors would have been different. The picture below shows how big the errors would be if we were to use a particularly silly line for estimation.

```
# Errors: Deviations from a different line
lw_errors(-100, 50000)
```



Below is a line that we have used before without saying that we were using a line to create estimates. It is the horizontal line at the value "average of  $y$ ." Suppose you were asked to estimate  $y$  and were *not told the value of  $x$* ; then you would use the average of  $y$  as your

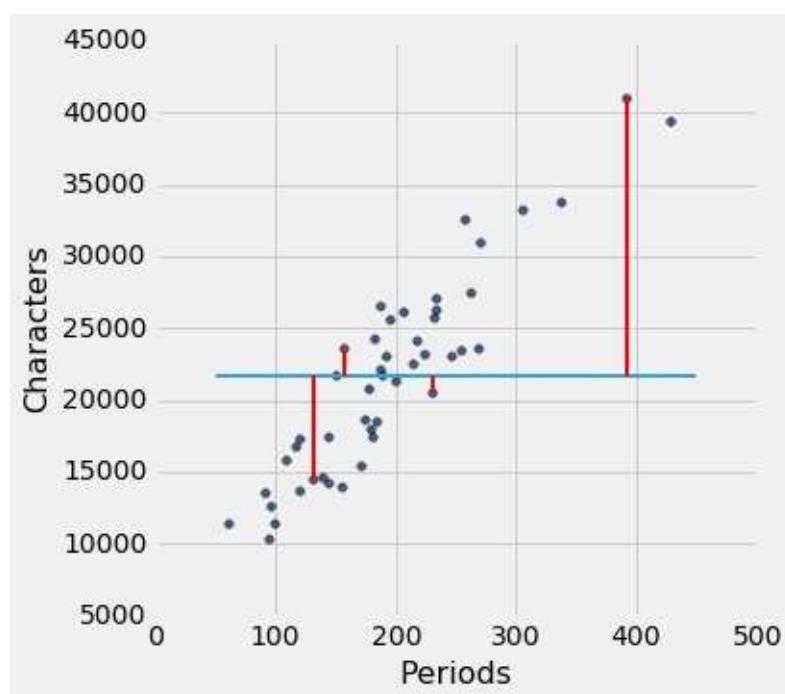
estimate, regardless of the chapter. In other words, you would use the flat line below.

Each error that you would make would then be a deviation from average. The rough size of these deviations is the SD of  $y$ .

In summary, if we use the flat line at the average of  $y$  to make our estimates, the estimates will be off by the SD of  $y$ .

```
# Errors: Deviations from the flat line at the average of y

characters_average = np.mean(little_women.column('Characters'))
lw_errors(0, characters_average)
```



## Least Squares

If you use any arbitrary line as your line of estimates, then some of your errors are likely to be positive and others negative. To avoid cancellation when measuring the rough size of the errors, we take the mean of the squared errors rather than the mean of the errors themselves. This is exactly analogous to our reason for looking at squared deviations from average, when we were learning how to calculate the SD.

The mean squared error of estimation using a straight line is a measure of roughly how big the squared errors are; taking the square root yields the root mean square error, which is in the same units as  $y$ .

Here is a remarkable fact of mathematics: the regression line minimizes the mean squared error of estimation (and hence also the root mean squared error) among all straight lines. That is why the regression line is sometimes called the "least squares line."

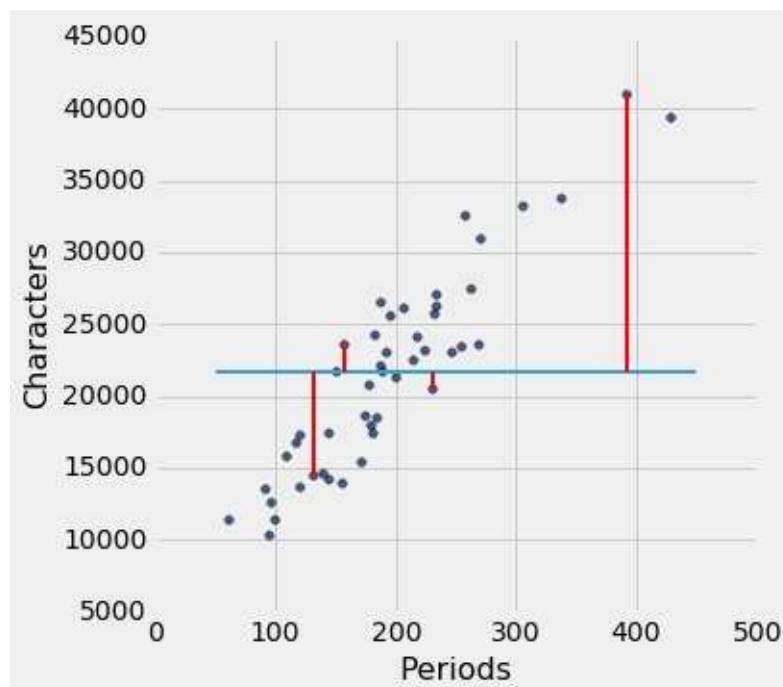
### Computing the "best" line.

- To get estimates of  $y$  based on  $x$ , you can use any line you want.
- Every line has a mean squared error of estimation.
- "Better" lines have smaller errors.
- **The regression line is the unique straight line that minimizes the mean squared error of estimation among all straight lines.**

We can define a function to compute the root mean squared error of any line through the Little Women scatter diagram.

```
def lw_rmse(slope, intercept):  
    lw_errors(slope, intercept)  
    x = little_women.column('Periods')  
    y = little_women.column('Characters')  
    fitted = slope * x + intercept  
    mse = np.average((y - fitted) ** 2)  
    print("Root mean squared error:", mse ** 0.5)  
  
lw_rmse(0, characters_average)
```

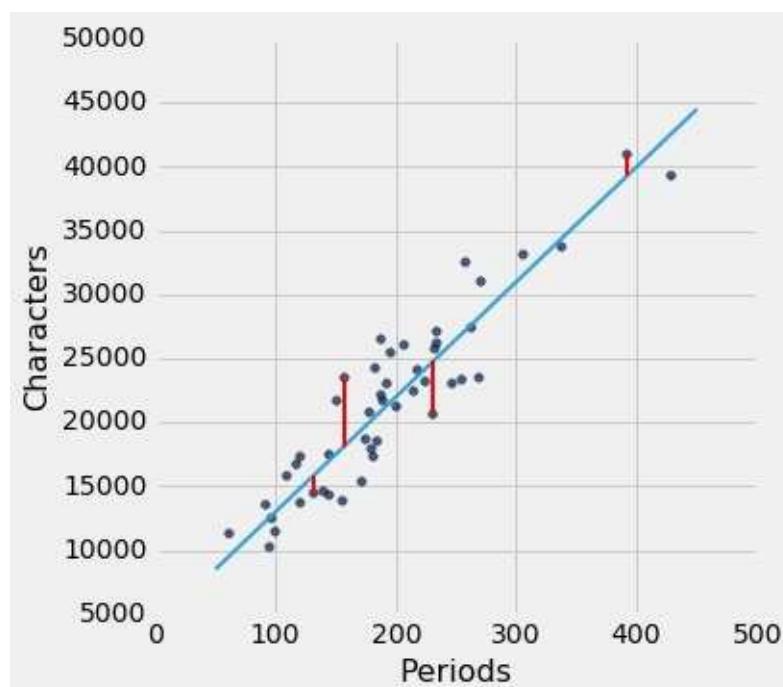
```
Root mean squared error: 7019.17593405
```



The error of the regression line is indeed much smaller if we choose a slope and intercept near the regression line.

```
lw_rmse(90, 4000)
```

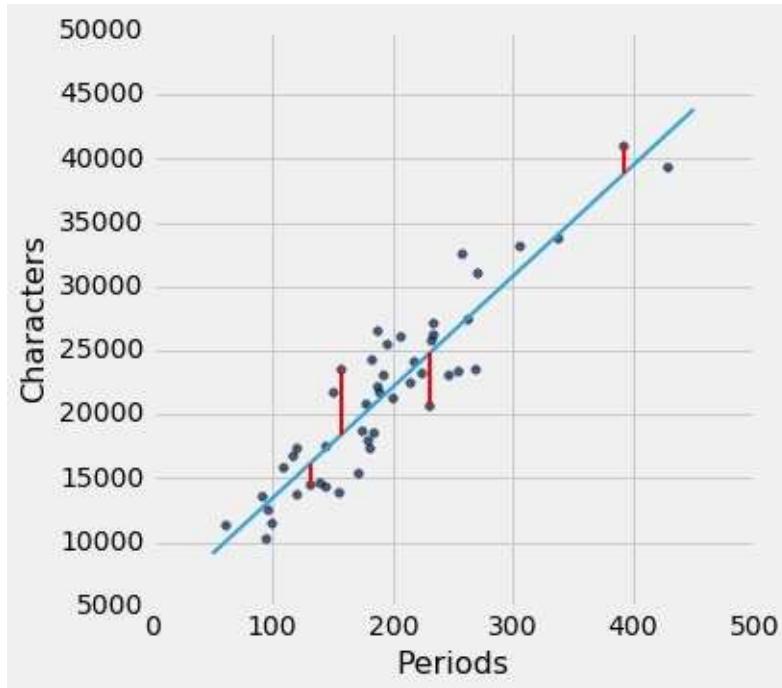
Root mean squared error: 2715.53910638



But the minimum is achieved using the regression line itself.

```
lw_rmse(lw_slope, lw_intercept)
```

```
Root mean squared error: 2701.69078531
```



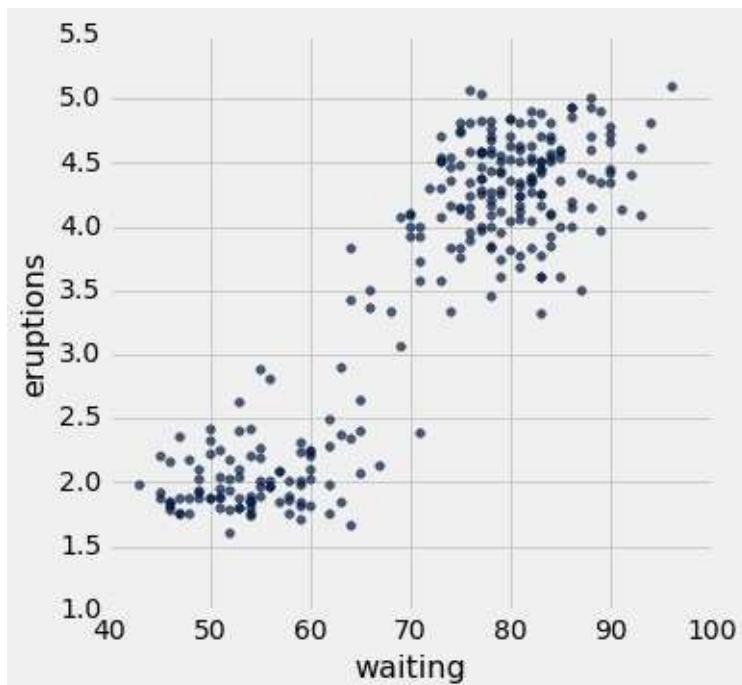
## Numerical Optimization

We can also define `mean_squared_error` for an arbitrary data set. We'll use a higher-order function so that we can try many different lines on the same data set, simply by passing in their slope and intercept.

```
def mean_squared_error(table, x, y):
    def for_line(slope, intercept):
        fitted = (slope * table.column(x) + intercept)
        return np.average((table.column(y) - fitted) ** 2)
    return for_line
```

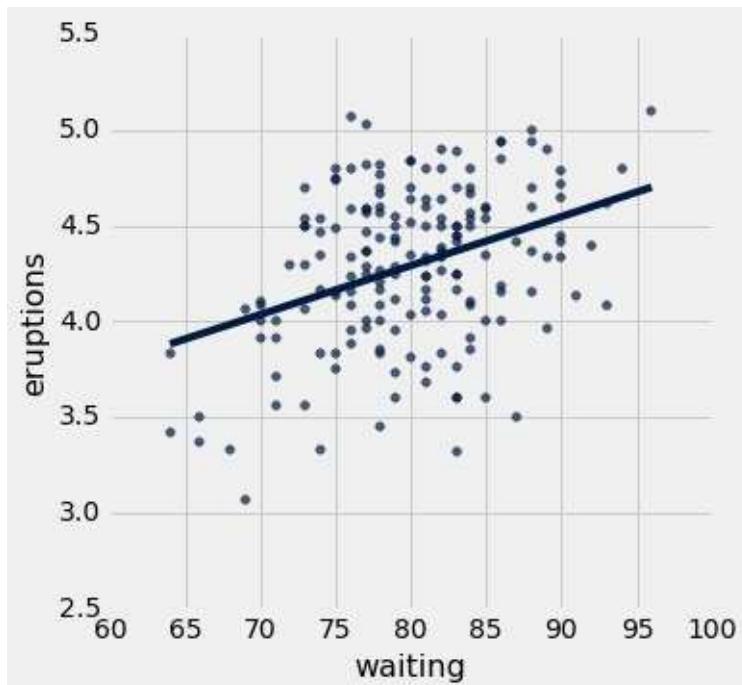
The [Old Faithful](#) geyser in Yellowstone national park erupts regularly, but the *waiting* time between eruptions (in seconds) and the duration of the *eruptions* in seconds do vary.

```
faithful = Table.read_table('faithful.csv')
faithful.scatter(1, 0)
```



It appears that there are two types of eruptions, short and long. The eruptions above 3 seconds appear correlated with waiting time.

```
long = faithful.where(faithful.column('eruptions') > 3)
long.scatter(1, 0, fit_line=True)
```



The `rmse_geyser` function takes a slope and an intercept and returns the root mean squared error of a linear predictor of eruption size from the waiting time.

```
mse_long = mean_squared_error(long, 1, 0)
mse_long(0, 4) ** 0.5
```

```
0.50268461001194098
```

If we experiment with different values, we can find a low-error slope and intercept through trial and error.

```
mse_long(0.01, 3.5) ** 0.5
```

```
0.39143872353883735
```

```
mse_long(0.02, 2.7) ** 0.5
```

```
0.38168519564089542
```

The `minimize` function can be used to find the arguments of a function for which the function returns its minimum value. Python uses a similar trial-and-error approach, following the changes that lead to incrementally lower output values.

```
a, b = minimize(mse_long)
```

The root mean squared error of the minimal slope and intercept is smaller than any of the values we considered before.

```
mse_long(a, b) ** 0.5
```

```
0.38014612988504093
```

In fact, these values `a` and `b` are the same as the values returned by the `slope` and `intercept` functions we developed based on the correlation coefficient. We see small deviations due to the inexact nature of `minimize`, but the values are essentially the same.

```
print("slope:    ", slope(long, 1, 0))
print("a:        ", a)
print("intercept:", intercept(long, 1, 0))
print("b:        ", b)
```

```
slope:    0.0255508940715
a:        0.0255496
intercept: 2.24752334164
b:        2.247629
```

Therefore, we have found not only that the regression line minimizes mean squared error, but also that minimizing mean squared error gives us the regression line. The regression line is the only line that minimizes mean squared error.

## Residuals

The amount of error in each of these regression estimates is the difference between the son's height and its estimate. These errors are called *residuals*. Some residuals are positive. These correspond to points that are above the regression line – points for which the regression line under-estimates  $y$ . Negative residuals correspond to the line over-estimating values of  $y$ .

```
waiting = long.column('waiting')
eruptions = long.column('eruptions')
fitted = a * waiting + b
res = long.with_columns([
    'eruptions (fitted)', fitted,
    'residuals', eruptions - fitted
])
res
```

eruptions	waiting	eruptions (fitted)	residuals
3.6	79	4.26605	-0.666047
3.333	74	4.1383	-0.805299
4.533	85	4.41934	0.113655
4.7	88	4.49599	0.204006
3.6	85	4.41934	-0.819345
4.35	85	4.41934	-0.069345
3.917	84	4.3938	-0.476795
4.2	78	4.2405	-0.0404978
4.7	83	4.36825	0.331754
4.8	84	4.3938	0.406205

... (165 rows omitted)

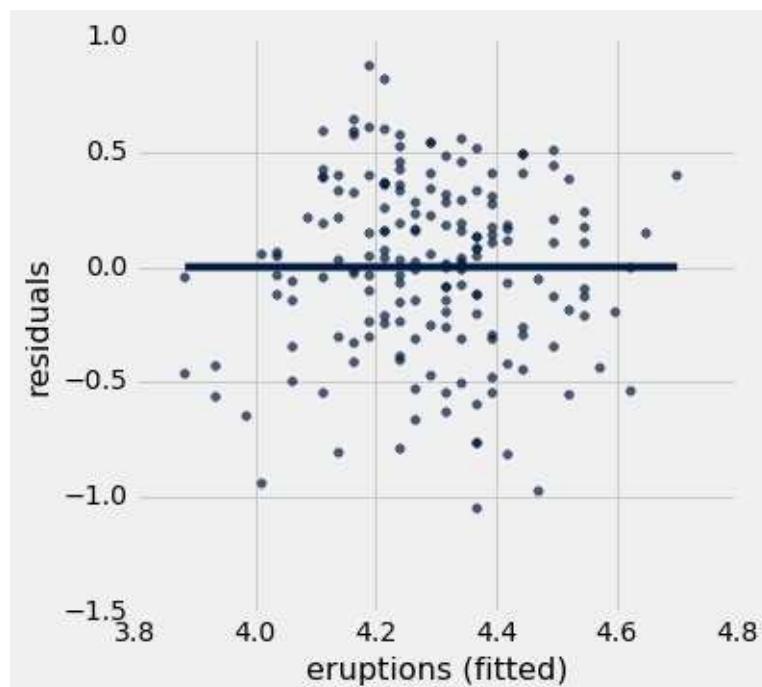
As with deviations from average, the positive and negative residuals exactly cancel each other out. So the average (and sum) of the residuals is 0. Because we found  $a$  and  $b$  by numerically minimizing the root mean squared error rather than computing them exactly, the sum of residuals is slightly different from zero in this case.

```
sum(res.column('residuals'))
```

```
-0.00037579999996140145
```

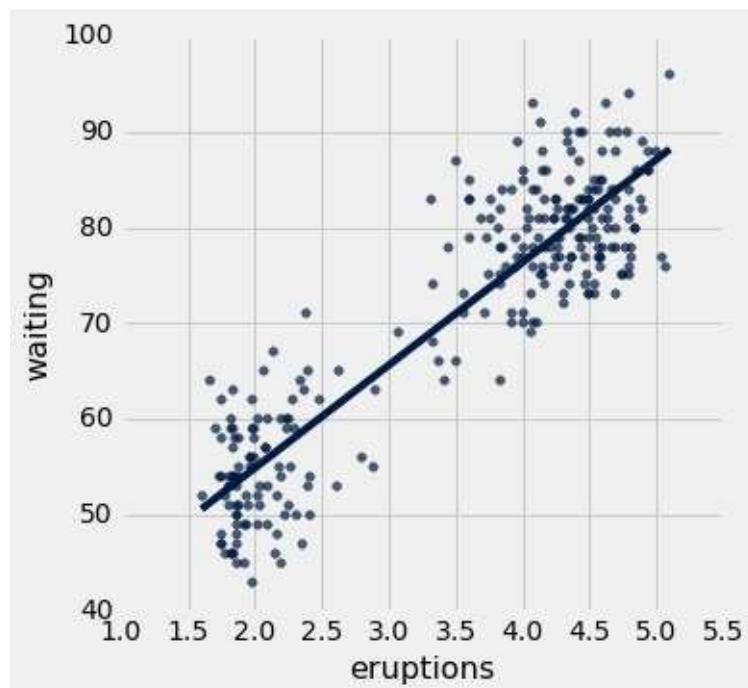
A residual plot is a scatter plot of the residuals versus the fitted values. The residual plot of a good regression looks like the one below: a formless cloud with no pattern, centered around the horizontal axis. It shows that there is no discernible non-linear pattern in the original scatter plot.

```
res.scatter(2, 3, fit_line=True)
```



By contrast, suppose we had attempted to fit a regression line to the entire set of eruptions of Old Faithful in the original dataset.

```
faithful.scatter(0, 1, fit_line=True)
```



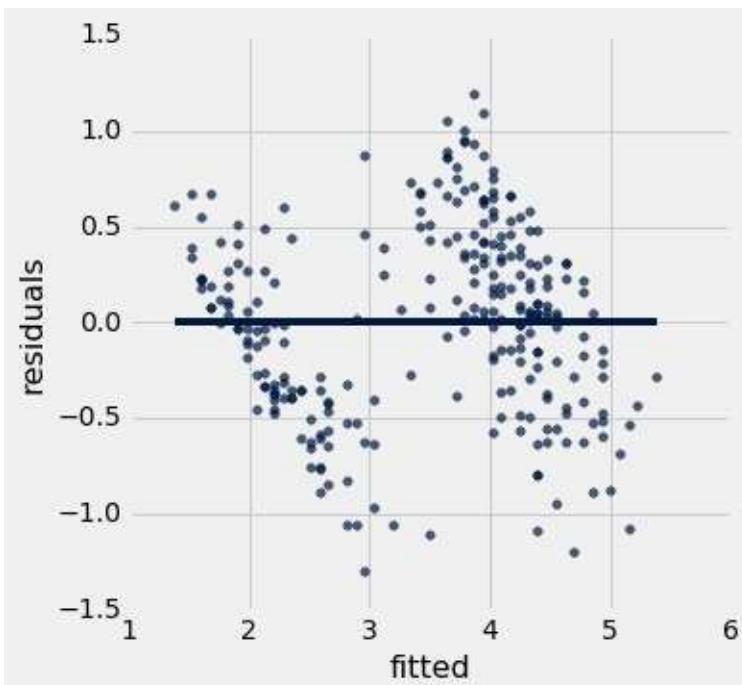
This line does not pass through the average value of vertical strips. We may be able to see this fact from the scatter alone, but it is dramatically clear by visualizing the residual scatter.

```

def residual_plot(table, x, y):
    fitted = fit(table, x, y)
    residuals = table.column(y) - fitted
    res_table = Table().with_columns([
        'fitted', fitted,
        'residuals', residuals])
    res_table.scatter(0, 1, fit_line=True)

residual_plot(faithful, 1, 0)

```



The diagonal stripes show a non-linear pattern in the data. In this case, we would not expect the regression line to provide low-error estimates.

## Example: SAT Scores

Residual plots can be useful for spotting non-linearity in the data, or other features that weaken the regression analysis. For example, consider the SAT data of the previous section, and suppose you try to estimate the `Combined` score based on `Participation Rate`.

```

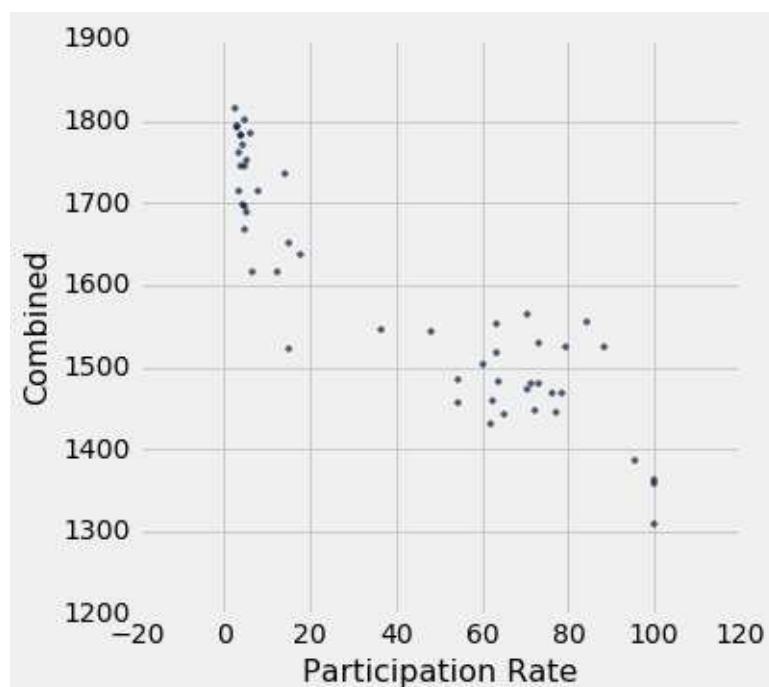
sat = Table.read_table('sat2014.csv')
sat

```

State	Participation Rate	Critical Reading	Math	Writing	Combined
North Dakota	2.3	612	620	584	1816
Illinois	4.6	599	616	587	1802
Iowa	3.1	605	611	578	1794
South Dakota	2.9	604	609	579	1792
Minnesota	5.9	598	610	578	1786
Michigan	3.8	593	610	581	1784
Wisconsin	3.9	596	608	578	1782
Missouri	4.2	595	597	579	1771
Wyoming	3.3	590	599	573	1762
Kansas	5.3	591	596	566	1753

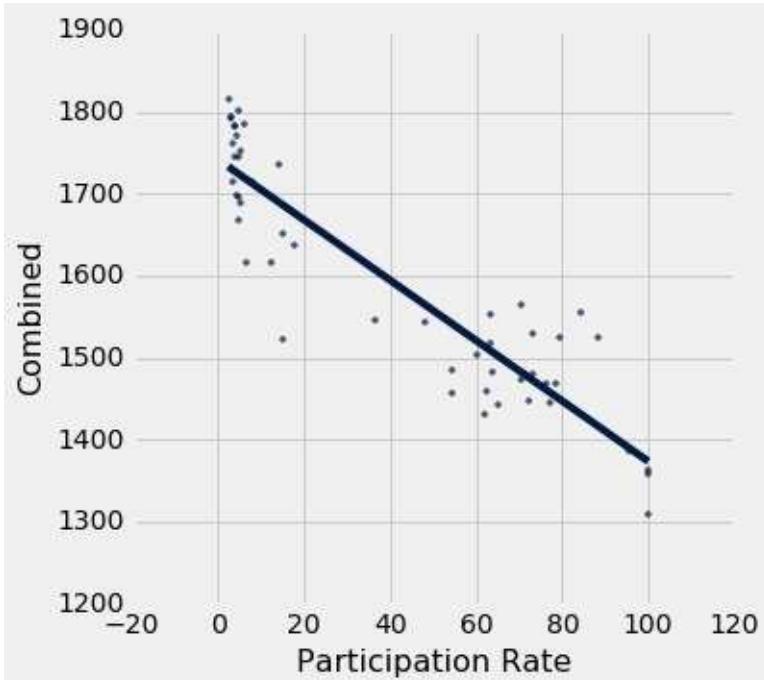
... (41 rows omitted)

```
sat.scatter('Participation Rate', 'Combined', s=8)
```



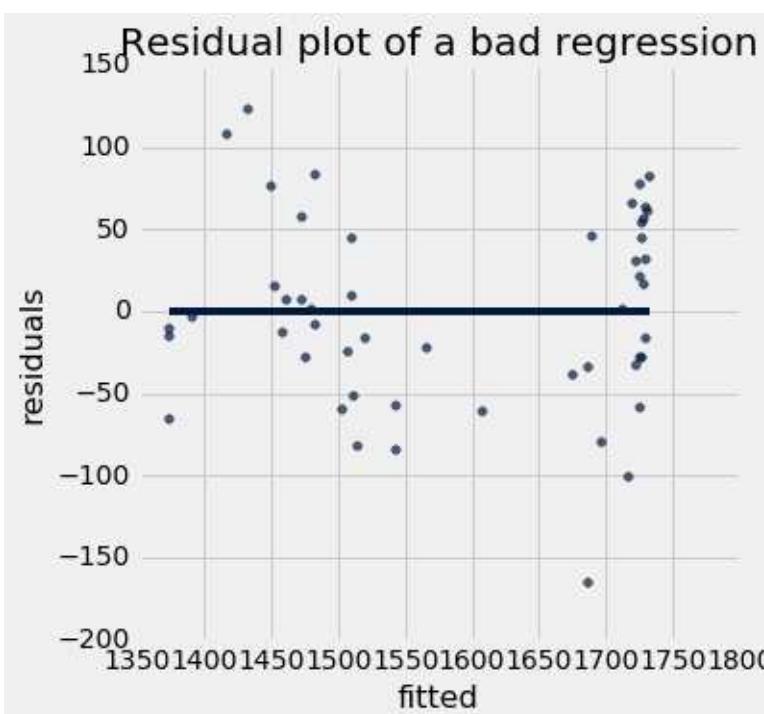
The relation between the variables is clearly non-linear, but you might be tempted to fit a straight line anyway, especially if you never looked at a scatter diagram of the data.

```
sat.scatter('Participation Rate', 'Combined', s=8, fit_line=True)
```



The points in the scatter plot start out above the regression line, then are consistently below the line, then above, then below. This pattern of non-linearity is more clearly visible in the residual plot.

```
residual_plot(sat, 'Participation Rate', 'Combined')
_= plt.title('Residual plot of a bad regression')
```



This residual plot is not a formless cloud; it shows a non-linear pattern, and is a signal that linear regression should not have been used for these data.

## The Size of Residuals

We will conclude by observing several relationships between quantities we have already identified. We'll illustrate the relationships using the `long` eruptions of Old Faithful, but they indeed hold for any collection of paired observations.

**Fact 1:** The root mean squared error of a line that has zero slope and an intercept at the average of `y` is the standard deviation of `y`.

```
mse_long(0, np.average(eruptions)) ** 0.5
```

```
0.40967604587437784
```

```
np.std(eruptions)
```

```
0.40967604587437784
```

By contrast, the standard deviation of fitted values is smaller.

```
eruptions_fitted = fit(long, 'waiting', 'eruptions')
np.std(eruptions_fitted)
```

```
0.15271994814407569
```

**Fact 2:** The ratio of the standard deviations of fitted values to `y` values is  $|r|$ .

```
r = correlation(long, 'waiting', 'eruptions')
print('r:      ', r)
print('ratio: ', np.std(eruptions_fitted) / np.std(eruptions))
```

```
r:      0.372782225571
ratio: 0.372782225571
```

Notice the absolute value of  $r$  in the formula above. For the heights of fathers and sons, the correlation is positive and so there is no difference between using  $r$  and using its absolute value. However, the result is true for variables that have negative correlation as well, provided we are careful to use the absolute value of  $r$  instead of  $r$ .

The regression line does a better job of estimating eruption lengths than a zero-slope line. Thus, the rough size of the errors made using the regression line must be smaller than that using a flat line. In other words, the SD of the residuals must be smaller than the overall SD of  $y$ .

```
eruptions_residuals = eruptions - eruptions_fitted  
np.std(eruptions_residuals)
```

```
0.38014612980028634
```

**Fact 3:** The SD of the residuals is  $\sqrt{1 - r^2}$  times the SD of  $y$ .

```
np.std(eruptions) * (1 - r**2) ** 0.5
```

```
0.38014612980028639
```

**Fact 4:** The variance of the fitted values plus the variance of the residuals gives the variance of  $y$ .

```
np.std(eruptions_fitted) ** 2 + np.std(eruptions_residuals) ** 2
```

```
0.16783446256326531
```

```
np.std(eruptions) ** 2
```

```
0.16783446256326534
```

# Multiple Regression

## Interact

Now that we have explored ways to use multiple features to predict a categorical variable, it is natural to study ways of using multiple predictor variables to predict a quantitative variable. A commonly used method to do this is called *multiple regression*.

We will start with an example to review some fundamental aspects of *simple regression*, that is, regression based on one predictor variable.

## Example: Simple Regression

Suppose that our goal is to use regression to estimate the height of a basset hound based on its weight, using a sample that looks consistent with the regression model. Suppose the observed correlation  $r$  is 0.5, and that the summary statistics for the two variables are as in the table below:

	average	SD
height	14 inches	2 inches
weight	50 pounds	5 pounds

To calculate the equation of the regression line, we need the slope and the intercept.

$$\begin{aligned} \text{slope} &= \frac{r \cdot \text{SD of } y}{\text{SD of } x} = \frac{0.5 \cdot 2 \text{ inches}}{5 \text{ pounds}} \\ &= 0.2 \text{ inches per pound} \end{aligned}$$

$$\begin{aligned} \text{intercept} &= \text{average of } y - \text{slope} \cdot \text{average of } x \\ &= 14 \text{ inches} - 0.2 \text{ inches per pound} \cdot 50 \text{ pounds} \\ &= 4 \text{ inches} \end{aligned}$$

The equation of the regression line allows us to calculate the estimated height, in inches, based on a given weight in pounds:

$$\text{estimated height} = 0.2 \cdot \text{given weight} + 4$$

The slope of the line is measures the increase in the estimated height per unit increase in weight. The slope is positive, and it is important to note that this does not mean that we think basset hounds get taller if they put on weight. The slope reflects the difference in the average heights of two groups of dogs that are 1 pound apart in weight. Specifically,

consider a group of dogs whose weight is  $w$  pounds, and the group whose weight is  $w + 1$  pounds. The second group is estimated to be 0.2 inches taller, on average. This is true for all values of  $w$  in the sample.

In general, the slope of the regression line can be interpreted as the average increase in  $y$  per unit increase in  $x$ . Note that if the slope is negative, then for every unit increase in  $x$ , the average of  $y$  decreases.

## Multiple Predictors

In multiple regression, more than one predictor variable is used to estimate  $y$ . For example, a Dartmouth study of undergraduates collected information about their use of an online course forum and their GPA. We might want to predict a student's GPA based on the number of days that they visit the course forum and how many times they answer someone else's question. Then the multiple regression model would involve two slopes, an intercept, and random errors as before:

$$\text{GPA} = \text{slope}_d * \text{days} + \text{slope}_a * \text{answers} + \text{intercept} \\ + \text{random error}$$

Our goal would be to find the estimated GPA using the best estimates of the two slopes and the intercept; as before, the "best" estimates are those that minimize the mean squared error of estimation.

To start off, we will investigate the data set, which is [described online](#). Each row represents a student. In addition to the student's GPA, the row tallies

- `days_online` : The number of days on which the student viewed the online forum
- `views` : The number of posts viewed
- `contributions` : The number of contributions, including posts and follow-up discussions
- `questions` : The number of questions posted
- `notes` : The number of notes posted
- `answers` : The number of answers posted

```
grades = Table.read_table('grades_and_piazza.csv')
grades
```

GPA	days online	views	contributions	questions	notes	answers
2.863	29	299	5	1	1	0
3.505	57	299	0	0	0	0
3.029	27	101	1	1	0	0
3.679	67	301	1	0	0	0
3.474	43	201	12	1	0	0
3.705	67	308	45	22	0	5
3.806	36	171	20	4	3	4
3.667	82	300	26	11	0	3
3.245	44	127	6	1	1	0
3.293	35	259	16	13	1	0
... (20 rows omitted)						

## Correlation Matrix

Perhaps we wish to predict GPA based on forum usage. A natural first step is to see which variables are correlated with the GPA. Here is the correlation matrix. The `to_df` method generates a Pandas dataframe containing the same data as the table, and its `corr` method generates a matrix of correlations for each pair of columns.

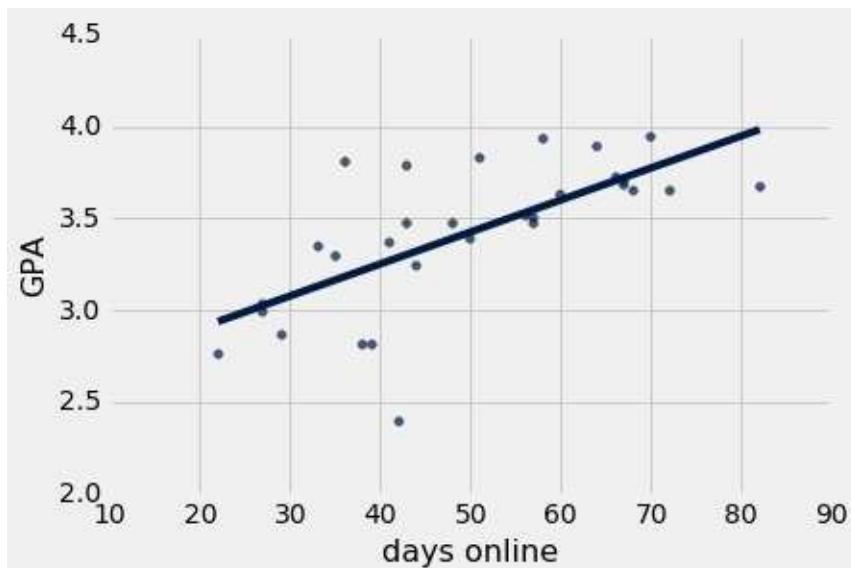
All forum usage variables are correlated with GPA, but the `notes` correlation has a very small magnitude.

```
grades.to_df().corr()
```

	GPA	days online	views	contributions	questions
GPA	1.000000	0.684905	0.444175	0.427897	0.409212
days online	0.684905	1.000000	0.654557	0.448319	0.435269
views	0.444175	0.654557	1.000000	0.426406	0.361002
contributions	0.427897	0.448319	0.426406	1.000000	0.857981
questions	0.409212	0.435269	0.361002	0.857981	1.000000
notes	-0.160604	-0.230839	0.065627	0.295661	-0.006365
answers	0.440382	0.502810	0.365010	0.702679	0.515661

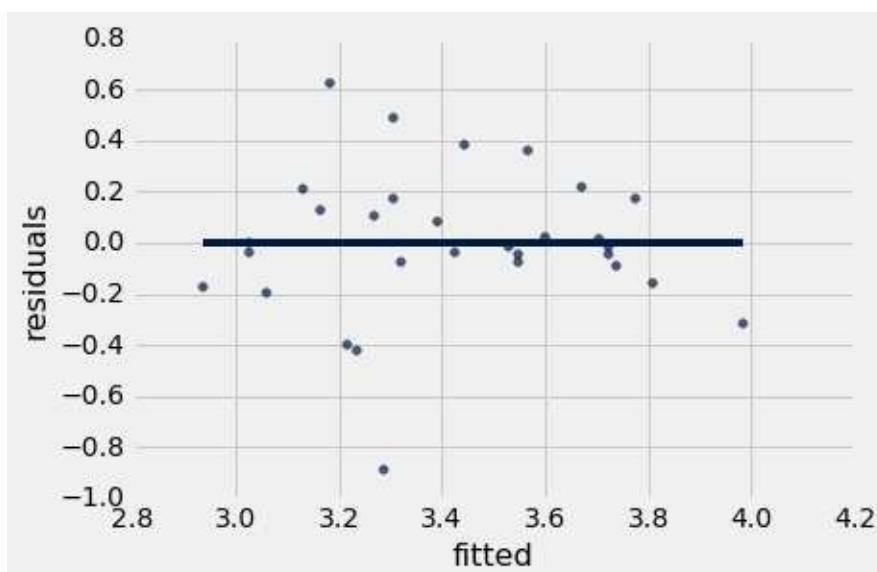
To start off, let us perform the simple regression of `GPA` on just `days online`, the count with the strongest correlation with an `r` of 0.68. Here is the scatter diagram and regression line.

```
grades.scatter('days online', 'GPA', fit_line=True)
```



We can immediately see that the relation is not entirely linear, and a residual plot highlights this fact. One issue is that the GPA is never above 4.0, so the model assumption that `y` values are bell shaped does not hold in this case.

```
residual_plot(grades, 'days online', 'GPA')
```



Nonetheless, the regression line does capture some of the pattern in the data, and so we will continue to work with it, noting that our predictions may not be entirely justified by the regression model.

The root mean squared error is a bit above 1/4 of a GPA point when predicting GPA from only the days online.

```
grades_days_mse = mean_squared_error(grades, 'days online', 'GPA')
a, b = minimize(grades_days_mse)
grades_days_mse(a, b) ** 0.5
```

```
0.28494512878662448
```

## Two Predictors¶

Perhaps more information can improve our prediction. The number of `answers` is also correlated with GPA. When using it alone as a predictor, the root mean squared error is greater because the correlation has lower magnitude than `days online`.

```
grades_answers_mse = mean_squared_error(grades, 'answers', 'GPA')
a, b = minimize(grades_answers_mse)
grades_answers_mse(a, b) ** 0.5
```

```
0.35110518920562062
```

However, the two variables can be used together. First, we define the mean squared error in terms of a line that has two slopes, one for each variable, as well as an intercept.

```
def grades_days_answers_mse(a_days, a_answers, b):
    fitted = a_days * grades.column('days online') + \
             a_answers * grades.column('answers') + \
             b
    y = grades.column('GPA')
    return np.average((y - fitted) ** 2)

minimize(grades_days_answers_mse)
```

```
array([ 0.0157683,  0.0301254,  2.6031789])
```

The `minimize` function returns three values, the slope for `days_online`, the slope for `answers`, and the intercept. Together, these three values describe a linear prediction function. For example, someone who spent 50 days online and answered 5 questions is predicted to have a GPA of about 3.54.

```
np.round(0.0157683 * 50 + 0.0301254 * 5 + 2.6031789, 2)
```

```
3.54
```

The root mean squared error of this predictor is a bit smaller than that of either single-variable predictor we had before!

```
a_days, a_answers, b = minimize(grades_days_answers_mse)  
grades_days_answers_mse(a_days, a_answers, b) ** 0.5
```

```
0.28161529539241298
```

## Combining all variables

We can combine all information about the course forum in order to attempt to find an even better predictor.

```

def fit_grades(a_days, a_views, a_contributions, a_questions, a_notes):
    return a_days * grades.column('days online') + \
           a_views * grades.column('views') + \
           a_contributions * grades.column('contributions') + \
           a_questions * grades.column('questions') + \
           a_notes * grades.column('notes') + \
           a_answers * grades.column('answers') + \
           b

def grades_all_mse(a_days, a_views, a_contributions, a_questions, a_answers):
    fitted = fit_grades(a_days, a_views, a_contributions, a_questions, a_answers)
    y = grades.column('GPA')
    return np.average((y - fitted) ** 2)

minimize(grades_all_mse)

array([ 1.43703000e-02, -8.15000000e-05,  6.50720000e-03,
       -1.72780000e-03, -4.04014000e-02,  1.59645000e-02,
       2.65375470e+00])

```

Using all of this information, we have constructed a predictor with an even smaller root mean squared error. The `*` below passes all of the return values of the `minimize` function as arguments to `grades_all_mse`.

```
grades_all_mse(*minimize(grades_all_mse)) ** 0.5
```

```
0.27783237243108722
```

All of this information did not improve our predictor very much. The root mean squared error decreased from 2.82 for the best single-variable predictor to 2.78 when using all of the available information.

### **Definition of $R^2$ , consistent with our old $r^2$**

When we studied simple regression, we had noted that

$$|r| = \frac{\text{SD of fitted values of } y}{\text{SD of observed values of } y}$$

Let us use our old functions to compute the fitted values and confirm that this is true for our example:

```
fitted = fit(grades, 'days online', 'GPA')
np.std(fitted) / np.std(grades.column('GPA'))
```

```
0.68490480299828194
```

Because variance is the square of the standard deviation, we can say that

$$0.469 = r^2 = \frac{\text{variance of fitted values of } y}{\text{variance of observed values of } y}$$

Notice that this way of thinking about  $r^2$  involves only the estimated values and the observed values, *not the number of predictor variables*. Therefore, it motivates the definition of *multiple  $R^2$* :

$$R^2 = \frac{\text{variance of fitted values of } y}{\text{variance of observed values of } y}$$

It is a fact of mathematics that this quantity is always between 0 and 1. With multiple predictor variables, there is no clear interpretation of a sign attached to the square root of  $R^2$ . Some of the predictors might be positively associated with  $y$ , others negatively. An overall measure of the fit is provided by  $R^2$ .

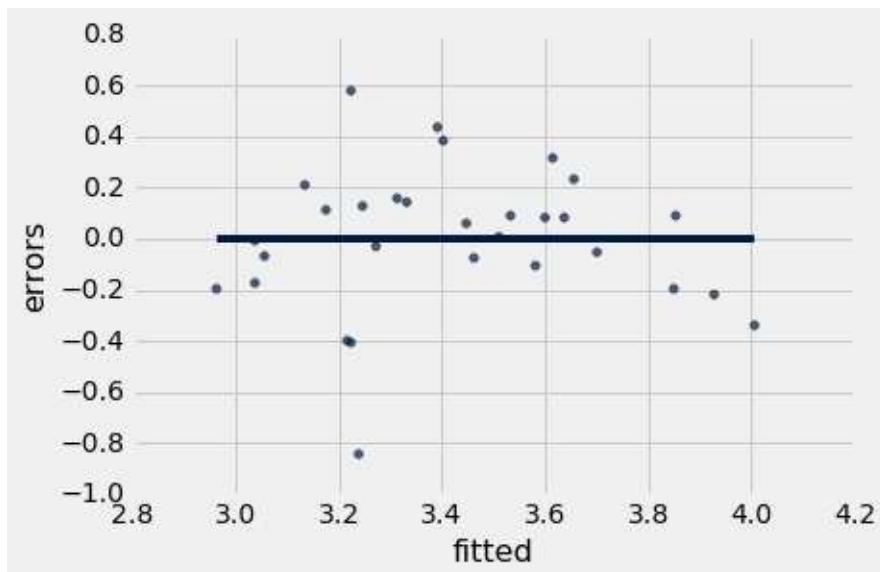
```
fitted = fit_grades(*minimize(grades_all_mse))
np.var(fitted) / np.var(grades.column('GPA'))
```

```
0.49525855565521787
```

It is not always wise to maximize  $R^2$  by including many variables, but we will address this concern later in the text.

We can also view the residual plot of multiple regression, just as before. The horizontal axis shows fitted values and the vertical axis shows errors. Again, there is some structure to this plot, so perhaps the relation between these variables is not entirely linear. In this case, the structure is minor enough that the regression model is a reasonable choice of predictor.

```
Table().with_columns([
    'fitted', fitted,
    'errors', grades.column('GPA') - fitted
]).scatter(0, 1, fit_line=True)
```



# Classification

Section author: [David Wagner](#)

## Interact

This section will discuss machine learning. Machine learning is a class of techniques for automatically finding patterns in data and using it to draw inferences or make predictions. We're going to focus on a particular kind of machine learning, namely, *classification*.

Classification is about learning how to make predictions from past examples: we're given some examples where we have been told what the correct prediction was, and we want to learn from those examples how to make good predictions in the future. Here are a few applications where classification is used in practice:

- For each order Amazon receives, Amazon would like to predict: *is this order fraudulent?* They have some information about each order (e.g., its total value, whether the order is being shipped to an address this customer has used before, whether the shipping address is the same as the credit card holder's billing address). They have lots of data on past orders, and they know whether which of those past orders were fraudulent and which weren't. They want to learn patterns that will help them predict, as new orders arrive, whether those new orders are fraudulent.
- Online dating sites would like to predict: *are these two people compatible?* Will they hit it off? They have lots of data on which matches they've suggested to their customers in the past, and they have some idea which ones were successful. As new customers sign up, they'd like to predict make predictions about who might be a good match for them.
- Doctors would like to know: *does this patient have cancer?* Based on the measurements from some lab test, they'd like to be able to predict whether the particular patient has cancer. They have lots of data on past patients, including their lab measurements and whether they ultimately developed cancer, and from that, they'd like to try to infer what measurements tend to be characteristic of cancer (or non-cancer) so they can diagnose future patients accurately.
- Politicians would like to predict: *are you going to vote for them?* This will help them focus fundraising efforts on people who are likely to support them, and focus get-out-the-vote efforts on voters who will vote for them. Public databases and commercial databases have a lot of information about most people: e.g., whether they own a home or rent; whether they live in a rich neighborhood or poor neighborhood; their interests and hobbies; their shopping habits; and so on. And political campaigns have surveyed

some voters and found out who they plan to vote for, so they have some examples where the correct answer is known. From this data, the campaigns would like to find patterns that will help them make predictions about all other potential voters.

All of these are classification tasks. Notice that in each of these examples, the prediction is a yes/no question -- we call this *binary classification*, because there are only two possible predictions. In a classification task, we have a bunch of *observations*. Each observation represents a single individual or a single situation where we'd like to make a prediction. Each observation has multiple *attributes*, which are known (e.g., the total value of the order; voter's annual salary; and so on). Also, each observation has a *class*, which is the answer to the question we care about (e.g., yes or no; fraudulent or not; etc.).

For instance, with the Amazon example, each order corresponds to a single observation. Each observation has several attributes (e.g., the total value of the order, whether the order is being shipped to an address this customer has used before, and so on). The class of the observation is either 0 or 1, where 0 means that the order is not fraudulent and 1 means that the order is fraudulent. Given the attributes of some new order, we are trying to predict its class.

Classification requires data. It involves looking for patterns, and to find patterns, you need data. That's where the data science comes in. In particular, we're going to assume that we have access to *training data*: a bunch of observations, where we know the class of each observation. The collection of these pre-classified observations is also called a training set. A classification algorithm is going to analyze the training set, and then come up with a classifier: an algorithm for predicting the class of future observations.

Note that classifiers do not need to be perfect to be useful. They can be useful even if their accuracy is less than 100%. For instance, if the online dating site occasionally makes a bad recommendation, that's OK; their customers already expect to have to meet many people before they'll find someone they hit it off with. Of course, you don't want the classifier to make too many errors -- but it doesn't have to get the right answer every single time.

## Chronic kidney disease

Let's work through an example. We're going to work with a data set that was collected to help doctors diagnose chronic kidney disease (CKD). Each row in the data set represents a single patient who was treated in the past and whose diagnosis is known. For each patient, we have a bunch of measurements from a blood test. We'd like to find which measurements are most useful for diagnosing CKD, and develop a way to classify future patients as "has CKD" or "doesn't have CKD" based on their blood test results.

Let's load the data set into a table and look at it.

```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose Random',
    ckd
```

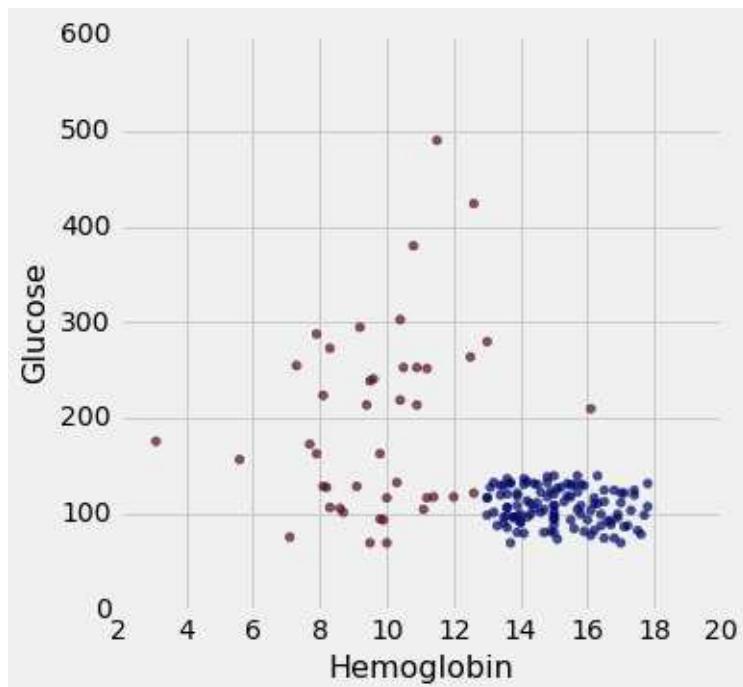
Age	Blood Pressure	Specific Gravity	Albumin	Sugar	Red Blood Cells	Pus Cell	Pus Cell
48	70	1.005	4	0	normal	abnormal	present
53	90	1.02	2	0	abnormal	abnormal	present
63	70	1.01	3	0	abnormal	abnormal	present
68	80	1.01	3	2	normal	abnormal	present
61	80	1.015	2	0	abnormal	abnormal	notpresent
48	80	1.025	4	0	normal	abnormal	notpresent
69	70	1.01	3	4	normal	abnormal	notpresent
73	70	1.005	0	0	normal	normal	notpresent
73	80	1.02	2	0	abnormal	abnormal	notpresent
46	60	1.01	1	0	normal	normal	notpresent

... (148 rows omitted)

We have data on 158 patients. There are an awful lot of attributes here. The column labelled "Class" indicates whether the patient was diagnosed with CKD: 1 means they have CKD, 0 means they do not have CKD.

Let's look at two columns in particular: the hemoglobin level (in the patient's blood), and the blood glucose level (at a random time in the day; without fasting specially for the blood test). We'll draw a scatter plot, to make it easy to visualize this. Red dots are patients with CKD; blue dots are patients without CKD. What test results seem to indicate CKD?

```
ckd.scatter('Hemoglobin', 'Glucose', c=ckd.column('Class'))
```



Suppose Alice is a new patient who is not in the data set. If I tell you Alice's hemoglobin level and blood glucose level, could you predict whether she has CKD? It sure looks like it! You can see a very clear pattern here: points in the lower-right tend to represent people who don't have CKD, and the rest tend to be folks with CKD. To a human, the pattern is obvious. But how can we program a computer to automatically detect patterns such as this one?

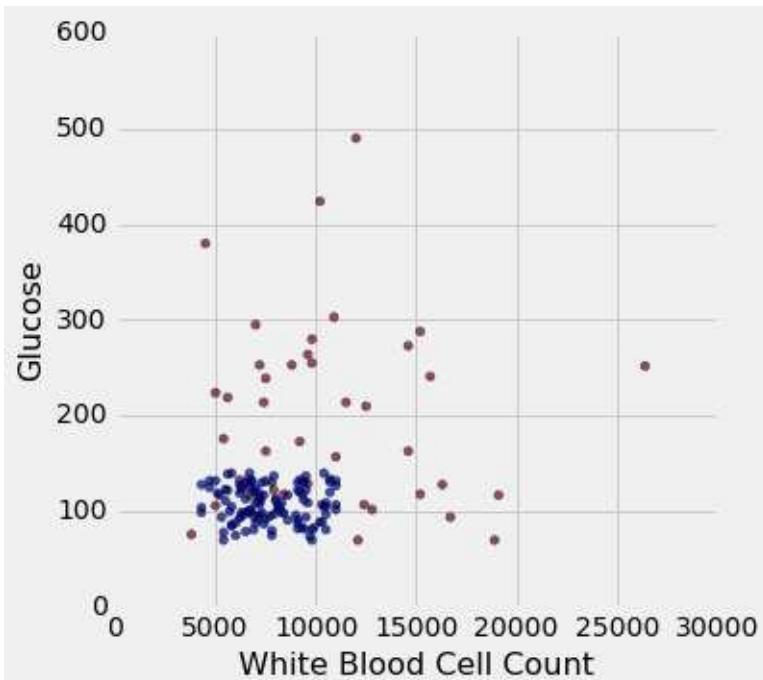
Well, there are lots of kinds of patterns one might look for, and lots of algorithms for classification. But I'm going to tell you about one that turns out to be surprisingly effective. It is called *nearest neighbor classification*. Here's the idea. If we have Alice's hemoglobin and glucose numbers, we can put her somewhere on this scatterplot; the hemoglobin is her x-coordinate, and the glucose is her y-coordinate. Now, to predict whether she has CKD or not, we find the nearest point in the scatterplot and check whether it is red or blue; we predict that Alice should receive the same diagnosis as that patient.

In other words, to classify Alice as CKD or not, we find the patient in the training set who is "nearest" to Alice, and then use that patient's diagnosis as our prediction for Alice. The intuition is that if two points are near each other in the scatterplot, then the corresponding measurements are pretty similar, so we might expect them to receive the same diagnosis (more likely than not). We don't know Alice's diagnosis, but we do know the diagnosis of all the patients in the training set, so we find the patient in the training set who is most similar to Alice, and use that patient's diagnosis to predict Alice's diagnosis.

The scatterplot suggests that this *nearest neighbor classifier* should be pretty accurate. Points in the lower-right will tend to receive a "no CKD" diagnosis, as their nearest neighbor will be a blue point. The rest of the points will tend to receive a "CKD" diagnosis, as their nearest neighbor will be a red point. So the nearest neighbor strategy seems to capture our intuition pretty well, for this example.

However, the separation between the two classes won't always be quite so clean. For instance, suppose that instead of hemoglobin levels we were to look at white blood cell count. Look at what happens:

```
ckd.scatter('White Blood Cell Count', 'Glucose', c=ckd.column('Class'))
```



As you can see, non-CKD individuals are all clustered in the lower-left. Most of the patients with CKD are above or to the right of that cluster... but not all. There are some patients with CKD who are in the lower left of the above figure (as indicated by the handful of red dots scattered among the blue cluster). What this means is that you can't tell for certain whether someone has CKD from just these two blood test measurements.

If we are given Alice's glucose level and white blood cell count, can we predict whether she has CKD? Yes, we can make a prediction, but we shouldn't expect it to be 100% accurate. Intuitively, it seems like there's a natural strategy for predicting: plot where Alice lands in the scatterplot; if she is in the lower-left, predict that she doesn't have CKD, otherwise predict she has CKD. This isn't perfect -- our predictions will sometimes be wrong. (Take a minute and think it through: for which patients will it make a mistake?) As the scatterplot above indicates, sometimes people with CKD have glucose and white blood cell levels that look identical to those of someone without CKD, so any classifier is inevitably going to make the wrong prediction for them.

Can we automate this on a computer? Well, the nearest neighbor classifier would be a reasonable choice here too. Take a minute and think it through: how will its predictions compare to those from the intuitive strategy above? When will they differ? Its predictions will be pretty similar to our intuitive strategy, but occasionally it will make a different prediction. In

particular, if Alice's blood test results happen to put her right near one of the red dots in the lower-left, the intuitive strategy would predict "not CKD", whereas the nearest neighbor classifier will predict "CKD".

There is a simple generalization of the nearest neighbor classifier that fixes this anomaly. It is called the *k-nearest neighbor classifier*. To predict Alice's diagnosis, rather than looking at just the one neighbor closest to her, we can look at the 3 points that are closest to her, and use the diagnosis for each of those 3 points to predict Alice's diagnosis. In particular, we'll use the majority value among those 3 diagnoses as our prediction for Alice's diagnosis. Of course, there's nothing special about the number 3: we could use 4, or 5, or more. (It's often convenient to pick an odd number, so that we don't have to deal with ties.) In general, we pick a number  $k$ , and our predicted diagnosis for Alice is based on the  $k$  patients in the training set who are closest to Alice. Intuitively, these are the  $k$  patients whose blood test results were most similar to Alice, so it seems reasonable to use their diagnoses to predict Alice's diagnosis.

The  $k$ -nearest neighbor classifier will now behave just like our intuitive strategy above.

## Decision boundary

Sometimes a helpful way to visualize a classifier is to map the region of space where the classifier would predict 'CKD', and the region of space where it would predict 'not CKD'. We end up with some boundary between the two, where points on one side of the boundary will be classified 'CKD' and points on the other side will be classified 'not CKD'. This boundary is called the *decision boundary*. Each different classifier will have a different decision boundary; the decision boundary is just a way to visualize what criteria the classifier is using to classify points.

## Banknote authentication

Let's do another example. This time we'll look at predicting whether a banknote (e.g., a \$20 bill) is counterfeit or legitimate. Researchers have put together a data set for us, based on photographs of many individual banknotes: some counterfeit, some legitimate. They computed a few numbers from each image, using techniques that we won't worry about for this course. So, for each banknote, we know a few numbers that were computed from a photograph of it as well as its class (whether it is counterfeit or not). Let's load it into a table and take a look.

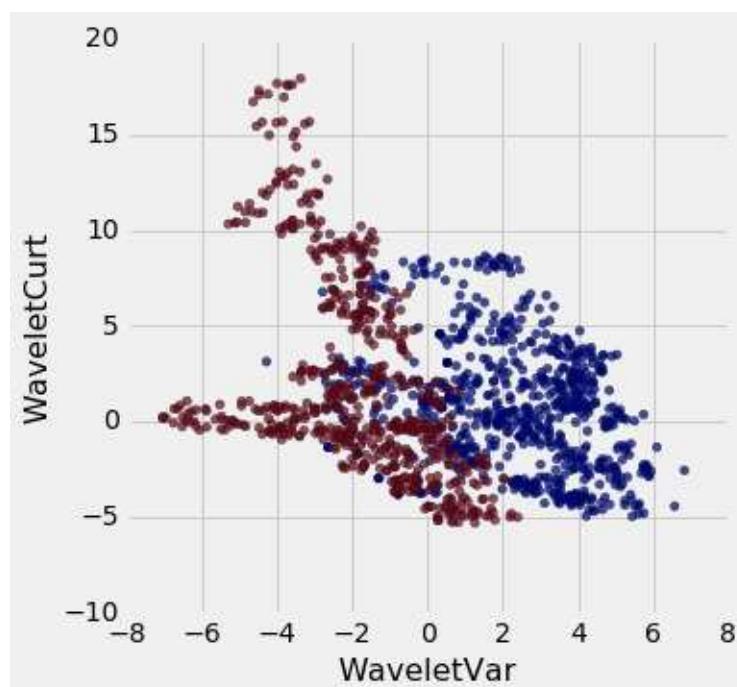
```
banknotes = Table.read_table('banknote.csv')
banknotes
```

WaveletVar	WaveletSkew	WaveletCurt	Entropy	Class
3.6216	8.6661	-2.8073	-0.44699	0
4.5459	8.1674	-2.4586	-1.4621	0
3.866	-2.6383	1.9242	0.10645	0
3.4566	9.5228	-4.0112	-3.5944	0
0.32924	-4.4552	4.5718	-0.9888	0
4.3684	9.6718	-3.9606	-3.1625	0
3.5912	3.0129	0.72888	0.56421	0
2.0922	-6.81	8.4636	-0.60216	0
3.2032	5.7588	-0.75345	-0.61251	0
1.5356	9.1772	-2.2718	-0.73535	0

... (1362 rows omitted)

Let's look at whether the first two numbers tell us anything about whether the banknote is counterfeit or not. Here's a scatterplot:

```
banknotes.scatter('WaveletVar', 'WaveletCurt', c=banknotes.column('
```



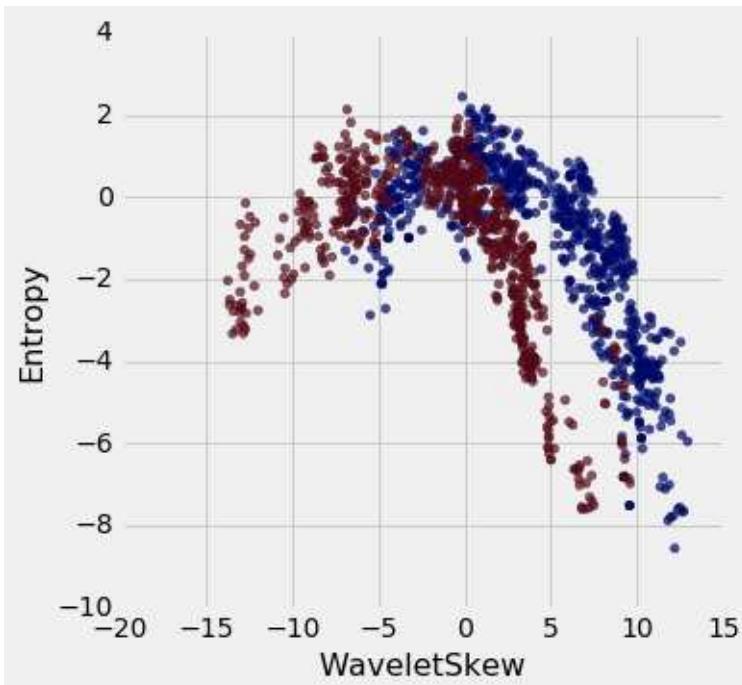
Pretty interesting! Those two measurements do seem helpful for predicting whether the banknote is counterfeit or not. However, in this example you can now see that there is some overlap between the blue cluster and the red cluster. This indicates that there will be some

images where it's hard to tell whether the banknote is legitimate based on just these two numbers. Still, you could use a  $k$ -nearest neighbor classifier to predict the legitimacy of a banknote.

Take a minute and think it through: Suppose we used  $k = 11$  (say). What parts of the plot would the classifier get right, and what parts would it make errors on? What would the decision boundary look like?

The patterns that show up in the data can get pretty wild. For instance, here's what we'd get if used a different pair of measurements from the images:

```
banknotes.scatter('WaveletSkew', 'Entropy', c=banknotes.column('Class'))
```



There does seem to be a pattern, but it's a pretty complex one. Nonetheless, the  $k$ -nearest neighbors classifier can still be used and will effectively "discover" patterns out of this. This illustrates how powerful machine learning can be: it can effectively take advantage of even patterns that we would not have anticipated, or that we would have thought to "program into" the computer.

## Multiple attributes

So far I've been assuming that we have exactly 2 attributes that we can use to help us make our prediction. What if we have more than 2? For instance, what if we have 3 attributes?

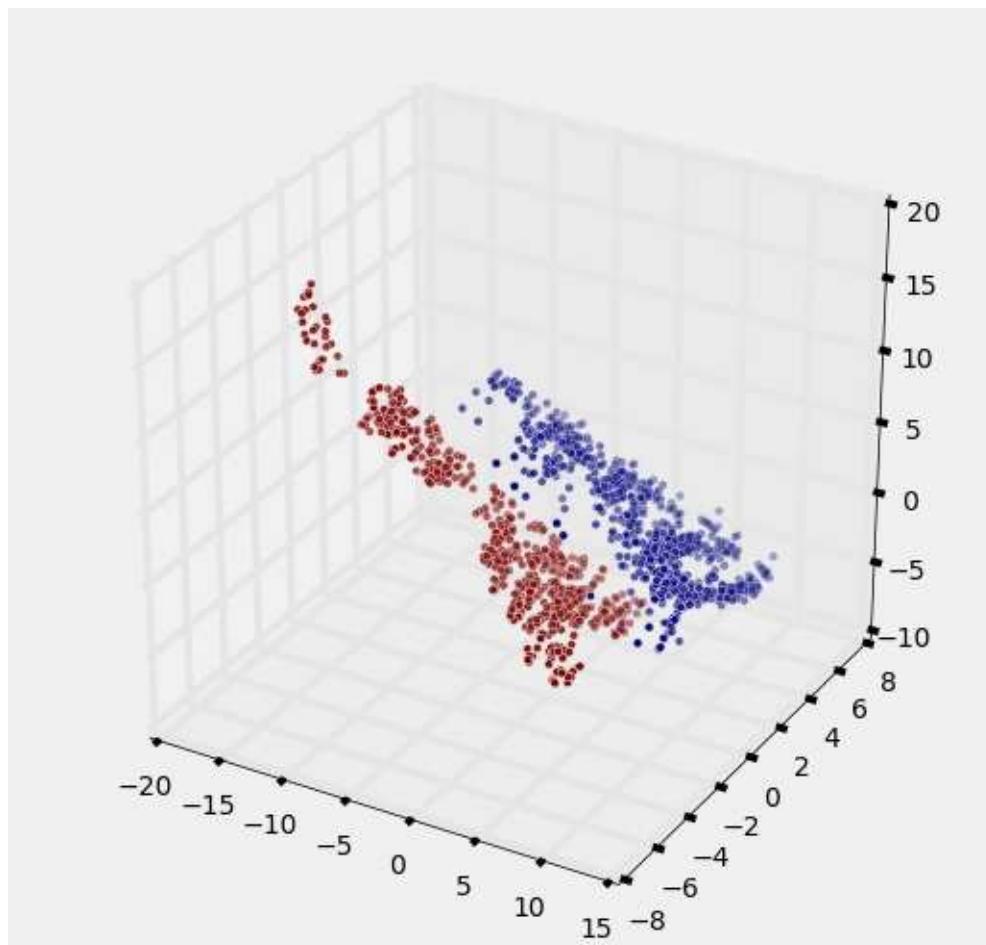
Here's the cool part: you can use the same ideas for this case, too. All you have to do is make a 3-dimensional scatterplot, instead of a 2-dimensional plot. You can still use the  $k$ -nearest neighbors classifier, but now computing distances in 3 dimensions instead of just 2. It just works. Very cool!

In fact, there's nothing special about 2 or 3. If you have 4 attributes, you can use the  $k$ -nearest neighbors classifier in 4 dimensions. 5 attributes? Work in 5-dimensional space. And no need to stop there! This all works for arbitrarily many attributes; you just work in a very high dimensional space. It gets wicked-impossible to visualize, but that's OK. The computer algorithm generalizes very nicely: all you need is the ability to compute the distance, and that's not hard. Mind-blowing stuff!

For instance, let's see what happens if we try to predict whether a banknote is counterfeit or not using 3 of the measurements, instead of just 2. Here's what you get:

```
ax = plt.figure(figsize=(8,8)).add_subplot(111, projection='3d')
ax.scatter(banknotes.column('WaveletSkew'),
           banknotes.column('WaveletVar'),
           banknotes.column('WaveletCurt'),
           c=banknotes.column('Class'))
```

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x1093bdef0>
```



Awesome! With just 2 attributes, there was some overlap between the two clusters (which means that the classifier was bound to make some mistakes for pointers in the overlap). But when we use these 3 attributes, the two clusters have almost no overlap. In other words, a classifier that uses these 3 attributes will be more accurate than one that only uses the 2 attributes.

This is a general phenomenon in classification. Each attribute can potentially give you new information, so more attributes sometimes helps you build a better classifier. Of course, the cost is that now we have to gather more information to measure the value of each attribute, but this cost may be well worth it if it significantly improves the accuracy of our classifier.

To sum up: you now know how to use  $k$ -nearest neighbor classification to predict the answer to a yes/no question, based on the values of some attributes, assuming you have a training set with examples where the correct prediction is known. The general roadmap is this:

1. identify some attributes that you think might help you predict the answer to the question;
2. gather a training set of examples where you know the values of the attributes as well as the correct prediction;
3. to make predictions in the future, measure the value of the attributes and then use  $k$ -nearest neighbor classification to predict the answer to the question.

## Breast cancer diagnosis

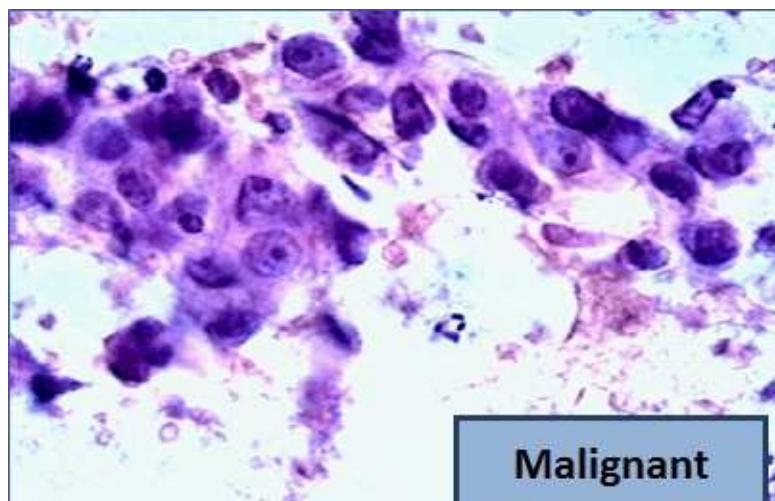
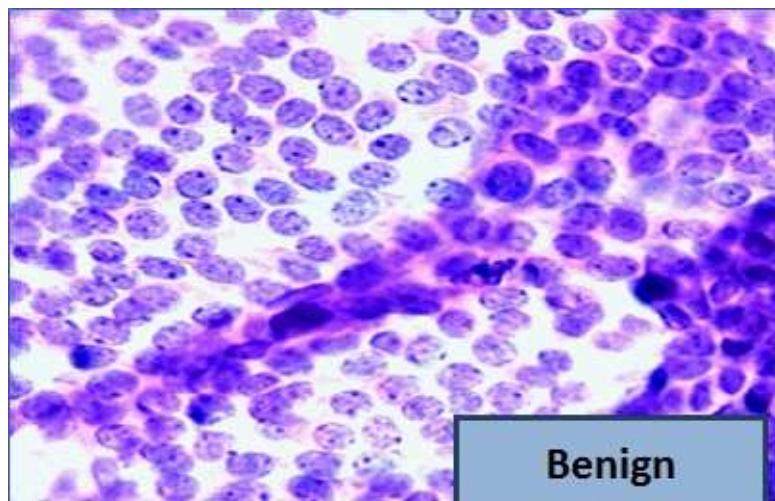
Now I want to do a more extended example based on diagnosing breast cancer. I was inspired by Brittany Wenger, who won the Google national science fair three years ago as a 17-year old high school student. Here's Brittany:



Brittany's science fair project was to build a classification algorithm to diagnose breast cancer. She won grand prize for building an algorithm whose accuracy was almost 99%.

Let's see how well we can do, with the ideas we've learned in this course.

So, let me tell you a little bit about the data set. Basically, if a woman has a lump in her breast, the doctors may want to take a biopsy to see if it is cancerous. There are several different procedures for doing that. Brittany focused on fine needle aspiration (FNA), because it is less invasive than the alternatives. The doctor gets a sample of the mass, puts it under a microscope, takes a picture, and a trained lab tech analyzes the picture to determine whether it is cancer or not. We get a picture like one of the following:



Unfortunately, distinguishing between benign vs malignant can be tricky. So, researchers have studied using machine learning to help with this task. The idea is that we'll ask the lab tech to analyze the image and compute various attributes: things like the typical size of a cell, how much variation there is among the cell sizes, and so on. Then, we'll try to use this information to predict (classify) whether the sample is malignant or not. We have a training set of past samples from women where the correct diagnosis is known, and we'll hope that our machine learning algorithm can use those to learn how to predict the diagnosis for future samples.

We end up with the following data set. For the "Class" column, 1 means malignant (cancer); 0 means benign (not cancer).

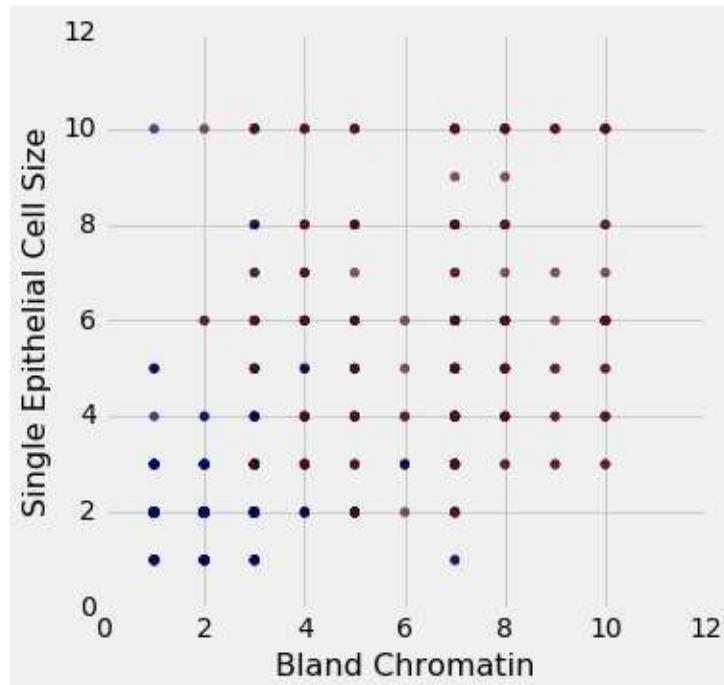
```
patients = Table.read_table('breast-cancer.csv').drop('ID')
patients
```

Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin
5	1	1	1	2	1	3
5	4	4	5	7	10	3
3	1	1	1	2	2	3
6	8	8	1	3	4	3
4	1	1	3	2	1	3
8	10	10	8	7	10	9
1	1	1	1	2	10	3
2	1	2	1	2	1	3
2	1	1	1	2	1	1
4	2	1	1	2	1	2

... (673 rows omitted)

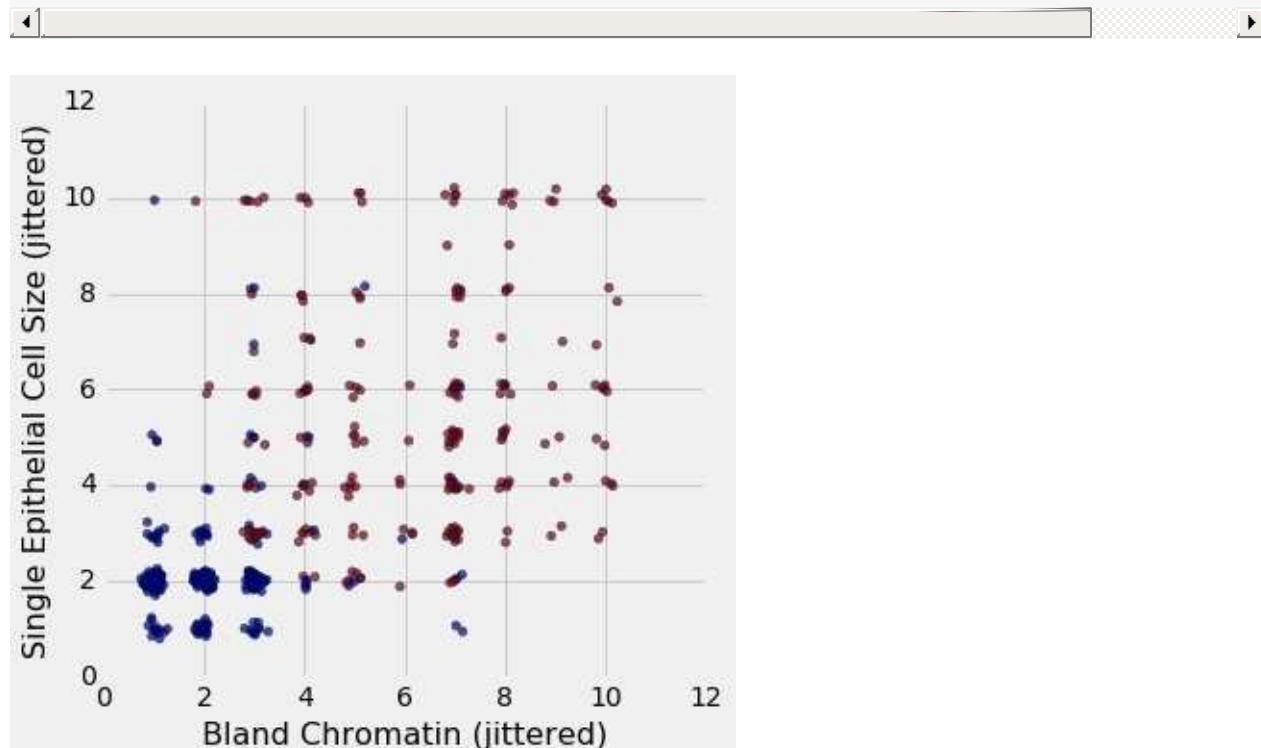
So we have 9 different attributes. I don't know how to make a 9-dimensional scatterplot of all of them, so I'm going to pick two and plot them:

```
patients.scatter('Bland Chromatin', 'Single Epithelial Cell Size',
```



Oops. That plot is utterly misleading, because there are a bunch of points that have identical values for both the x- and y-coordinates. To make it easier to see all the data points, I'm going to add a little bit of random jitter to the x- and y-values. Here's how that looks:

```
def randomize_column(a):
    return a + np.random.normal(0.0, 0.09, size=len(a))
Table().with_columns([
    'Bland Chromatin (jittered)',
    randomize_column(patients.column('Bland Chromatin')),
    'Single Epithelial Cell Size (jittered)',
    randomize_column(patients.column('Single Epithelial Cell Si
]).scatter(0, 1, c=patients.column('Class'))
```



For instance, you can see there are lots of samples with chromatin = 2 and epithelial cell size = 2; all non-cancerous.

Keep in mind that the jittering is just for visualization purposes, to make it easier to get a feeling for the data. When we want to work with the data, we'll use the original (unjittered) data.

## Applying the k-nearest neighbor classifier to breast cancer diagnosis

We've got a data set. Let's try out the  $k$ -nearest neighbor classifier and see how it does. This is going to be great.

We're going to need an implementation of the  $k$ -nearest neighbor classifier. In practice you would probably use an existing library, but it's simple enough that I'm going to implement it myself.

The first thing we need is a way to compute the distance between two points. How do we do this? In 2-dimensional space, it's pretty easy. If we have a point at coordinates  $(x_0, y_0)$  and another at  $(x_1, y_1)$ , the distance between them is

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}.$$

(Where did this come from? It comes from the Pythagorean theorem: we have a right triangle with side lengths  $x_0 - x_1$  and  $y_0 - y_1$ , and we want to find the length of the diagonal.)

In 3-dimensional space, the formula is

$$D = \sqrt{x_0^2 + x_1^2 + y_0^2 + y_1^2 + z_0^2 + z_1^2}.$$

In  $k$ -dimensional space, things are a bit harder to visualize, but I think you can see how the formula generalized: we sum up the squares of the differences between each individual coordinate, and then take the square root of that. Let's implement a function to compute this distance function for us:

```
def distance(pt1, pt2):
    total = 0
    for i in np.arange(len(pt1)):
        total = total + (pt1.item(i) - pt2.item(i))**2
    return math.sqrt(total)
```

Next, we're going to write some code to implement the classifier. The input is a patient  $p$  who we want to diagnose. The classifier works by finding the  $k$  nearest neighbors of  $p$  from the training set. So, our approach will go like this:

1. Find the closest  $k$  neighbors of  $p$ , i.e., the  $k$  patients from the training set that are most similar to  $p$ .
2. Look at the diagnoses of those  $k$  neighbors, and take the majority vote to find the most-common diagnosis. Use that as our predicted diagnosis for  $p$ .

So that will guide the structure of our Python code.

To implement the first step, we will compute the distance from each patient in the training set to `p`, sort them by distance, and take the  $k$  closest patients in the training set. The code will make a copy of the table, compute the distance from each patient to `p`, add a new column to the table with those distances, and then sort the table by distance and take the first  $k$  rows. That leads to the following Python code:

```
def closest(training, p, k):
    ...

def majority(topkclasses):
    ...

def classify(training, p, k):
    kclosest = closest(training, p, k)
    kclosest.classes = kclosest.select('Class')
    return majority(kclosest)
```

```

def computetablewithdists(training, p):
    dists = np.zeros(training.num_rows)
    attributes = training.drop('Class')
    for i in np.arange(training.num_rows):
        dists[i] = distance(attributes.row(i), p)
    return training.with_column('Distance', dists)

def closest(training, p, k):
    withdists = computetablewithdists(training, p)
    sortedbydist = withdists.sort('Distance')
    topk = sortedbydist.take(np.arange(k))
    return topk

def majority(topkclasses):
    if topkclasses.where('Class', 1).num_rows > topkclasses.where('
        return 1
    else:
        return 0

def classify(training, p, k):
    closestk = closest(training, p, k)
    topkclasses = closestk.select('Class')
    return majority(topkclasses)

```

Let's see how this works, with our data set. We'll take patient 12 and imagine we're going to try to diagnose them:

```
patients.take(12)
```

Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Blar Chrron
5	3	3	3	2	3	4

We can pull out just their attributes (excluding the class), like this:

```
patients.drop('Class').row(12)
```

```
Row(Clump Thickness=5, Uniformity of Cell Size=3, Uniformity of Cel
```

Let's take  $k = 5$ . We can find the 5 nearest neighbors:

```
closest(patients, patients.drop('Class').row(12), 5)
```

Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Blar Chrom
5	3	3	3	2	3	4
5	3	3	4	2	4	3
5	1	3	3	2	2	2
5	2	2	2	2	2	3
5	3	3	1	3	3	3

3 out of the 5 nearest neighbors have class 1, so the majority is 1 (has cancer) -- and that is the output of our classifier for this patient:

```
classify(patients, patients.drop('Class').row(12), 5)
```

1

Awesome! We now have a classification algorithm for diagnosing whether a patient has breast cancer or not, based on the measurements from the lab. Are we done? Shall we give this to doctors to use?

Hold on: we're not done yet. There's an obvious question to answer, before we start using this in practice:

*How accurate is this method, at diagnosing breast cancer?*

And that raises a more fundamental issue. How can we measure the accuracy of a classification algorithm?

## Measuring accuracy of a classifier

We've got a classifier, and we'd like to determine how accurate it will be. How can we measure that?

**Try it out.** One natural idea is to just try it on patients for a year, keep records on it, and see how accurate it is. However, this has some disadvantages: (a) we're trying something on patients without knowing how accurate it is, which might be unethical; (b) we have to wait a year to find out whether our classifier is any good. If it's not good enough and we get an idea for an improvement, we'll have to wait another year to find out whether our improvement was better.

**Get some more data.** We could try to get some more data from other patients whose diagnosis is known, and measure how accurate our classifier's predictions are on those additional patients. We can compare what the classifier outputs against what we know to be true.

**Use the data we already have.** Another natural idea is to re-use the data we already have: we have a training set that we used to train our classifier, so we could just run our classifier on every patient in the data set and compare what it outputs to what we know to be true. This is sometimes known as testing the classifier on your training set.

How should we choose among these options? Are they all equally good?

It turns out that the third option, testing the classifier on our training set, is fundamentally flawed. It might sound attractive, but it gives misleading results: it will over-estimate the accuracy of the classifier (it will make us think the classifier is more accurate than it really is). Intuitively, the problem is that what we really want to know is how well the classifier has done at "generalizing" beyond the specific examples in the training set; but if we test it on patients from the training set, then we haven't learned anything about how well it would generalize to other patients.

This is subtle, so it might be helpful to try an example. Let's try a thought experiment. Let's focus on the 1-nearest neighbor classifier ( $k = 1$ ). Suppose you trained the 1-nearest neighbor classifier on data from all 683 patients in the data set, and then you tested it on those same 683 patients. How many would it get right? Think it through and see if you can work out what will happen. That's right! The classifier will get the right answer for all 683 patients. Suppose we apply the classifier to a patient from the training set, say Alice. The classifier will look for the nearest neighbor (the most similar patient from the training set), and the nearest neighbor will turn out to be Alice herself (the distance from any point to itself is zero). Thus, the classifier will produce the right diagnosis for Alice. The same reasoning applies to every other patient in the training set.

So, if we test the 1-nearest neighbor classifier on the training set, the accuracy will always be 100%: absolutely perfect. This is true no matter whether there are actually any patterns in the data. But the 100% is a total lie. When you apply the classifier to other patients who were not in the training set, the accuracy could be far worse.

In other words, testing on the training tells you nothing about how accurate the 1-nearest neighbor classifier will be. This illustrates why testing on the training set is so flawed. This flaw is pretty blatant when you use the 1-nearest neighbor classifier, but don't think that with some other classifier you'd be immune to this problem -- the problem is fundamental and applies no matter what classifier you use. Testing on the training set gives you a biased estimate of the classifier's accuracy. For these reasons, you should never test on the training set.

So what *should* you do, instead? Is there a more principled approach?

It turns out there is. The approach comes down to: get more data. More specifically, the right solution is to use one data set for training, and a different data set for testing, with no overlap between the two data sets. We call these a *training set* and a *test set*.

Where do we get these two data sets from? Typically, we'll start out with some data, e.g., the data set on 683 patients, and before we do anything else with it, we'll split it up into a training set and a test set. We might put 50% of the data into the training set and the other 50% into the test set. Basically, we are setting aside some data for later use, so we can use it to measure the accuracy of our classifier. Sometimes people will call the data that you set aside for testing a *hold-out set*, and they'll call this strategy for estimating accuracy the *hold-out method*.

Note that this approach requires great discipline. Before you start applying machine learning methods, you have to take some of your data and set it aside for testing. You must avoid using the test set for developing your classifier: you shouldn't use it to help train your classifier or tweak its settings or for brainstorming ways to improve your classifier. Instead, you should use it only once, at the very end, after you've finalized your classifier, when you want an unbiased estimate of its accuracy.

## The effectiveness of our classifier, for breast cancer

OK, so let's apply the hold-out method to evaluate the effectiveness of the  $k$ -nearest neighbor classifier for breast cancer diagnosis. The data set has 683 patients, so we'll randomly permute the data set and put 342 of them in the training set and the remaining 341 in the test set.

```
patients = patients.sample(683) # Randomly permute the rows
trainset = patients.take(range(342))
testset = patients.take(range(342, 683))
```

We'll train the classifier using the 342 patients in the training set, and evaluate how well it performs on the test set. To make our lives easier, we'll write a function to evaluate a classifier on every patient in the test set:

```
def evaluate_accuracy(training, test, k):
    testattrs = test.drop('Class')
    numcorrect = 0
    for i in range(test.num_rows):
        # Run the classifier on the ith patient in the test set
        c = classify(training, testattrs.rows[i], k)
        # Was the classifier's prediction correct?
        if c == test.column('Class').item(i):
            numcorrect = numcorrect + 1
    return numcorrect / test.num_rows
```

Now for the grand reveal -- let's see how we did. We'll arbitrarily use  $k = 5$ .

```
evaluate_accuracy(trainset, testset, 5)
```

```
0.967741935483871
```

About 96% accuracy. Not bad! Pretty darn good for such a simple technique.

As a footnote, you might have noticed that Brittany Wenger did even better. What techniques did she use? One key innovation is that she incorporated a confidence score into her results: her algorithm had a way to determine when it was not able to make a confident prediction, and for those patients, it didn't even try to predict their diagnosis. Her algorithm was 99% accurate on the patients where it made a prediction -- so that extension seemed to help quite a bit.

## Important takeaways

Here are a few lessons we want you to learn from this.

First, machine learning is powerful. If you had to try to write code to make a diagnosis without knowing about machine learning, you might spend a lot of time by trial-and-error trying to come up with some complicated set of rules that seem to work, and the result might not be very accurate. The  $k$ -nearest neighbors algorithm automates the entire task for you. And machine learning often lets them make predictions far more accurately than anything you'd come up with by trial-and-error.

Second, you can do it. Yes, you. You can use machine learning in your own work to make predictions based on data. You now know enough to start applying these ideas to new data sets and help others make useful predictions. The techniques are very powerful, but you don't have to have a Ph.D. in statistics to use them.

Third, be careful about how to evaluate accuracy. Use a hold-out set.

There's lots more one can say about machine learning: how to choose attributes, how to choose  $k$  or other parameters, what other classification methods are available, how to solve more complex prediction tasks, and lots more. In this course, we've barely even scratched the surface. If you enjoyed this material, you might enjoy continuing your studies in statistics and computer science; courses like Stats 132 and 154 and CS 188 and 189 go into a lot more depth.

# Features

Section author: [David Wagner](#)

[Interact](#)

## Building a model

So far, we have talked about *prediction*, where the purpose of learning is to be able to predict the class of new instances. I'm now going to switch to *model building*, where the goal is to learn a model of how the class depends upon the attributes.

One place where model building is useful is for science: e.g., which genes influence whether you become diabetic? This is interesting and useful in its right (apart from any applications to predicting whether a particular individual will become diabetic), because it can potentially help us understand the workings of our body.

Another place where model building is useful is for control: e.g., what should I change about my advertisement to get more people to click on it? How should I change the profile picture I use on an online dating site, to get more people to "swipe right"? Which attributes make the biggest difference to whether people click/swipe? Our goal is to determine which attributes to change, to have the biggest possible effect on something we care about.

We already know how to build a classifier, given a training set. Let's see how to use that as a building block to help us solve these problems.

How do we figure out which attributes have the biggest influence on the output? Take a moment and see what you can come up with.

## Feature selection

Background: attributes are also called *features*, in the machine learning literature.

Our goal is to find a subset of features that are most relevant to the output. The way we'll formalize this is to identify a subset of features that, when we train a classifier using just those features, gives the highest possible accuracy at prediction.

Intuitively, if we get 90% accuracy using all the features and 88% accuracy using just three of the features (for example), then it stands to reason that those three features are probably the most relevant, and they capture most of the information that affects or determines the

output.

With this insight, our problem becomes:

Find the subset of  $\ell$  features that gives the best possible accuracy (when we use only those  $\ell$  features for prediction).

This is a feature selection problem. There are many possible approaches to feature selection. One simple one is to try all possible ways of choosing  $\ell$  of the features, and evaluate the accuracy of each. However, this can be very slow, because there are so many ways to choose a subset of  $\ell$  features.

Therefore, we'll consider a more efficient procedure that often works reasonably well in practice. It is known as greedy feature selection. Here's how it works.

1. Suppose there are  $d$  features. Try each on its own, to see how much accuracy we can get using a classifier trained with just that one feature. Keep the best feature.
2. Now we have one feature. Try remaining  $d - 1$  features, to see which is the best one to add to it (i.e., we are now training a classifier with just 2 features: the best feature picked in step 1, plus one more). Keep the one that best improves accuracy. Now we have 2 features.
3. Repeat. At each stage, we try all possibilities for how to add one more feature to the feature subset we've already picked, and we keep the one that best improves accuracy.

Let's implement it and try it on some examples!

## Code for k-NN

First, some code from last time, to implement  $k$ -nearest neighbors.

```
def distance(pt1, pt2):
    tot = 0
    for i in range(len(pt1)):
        tot = tot + (pt1[i] - pt2[i])**2
    return math.sqrt(tot)
```

```

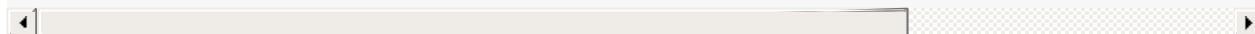
def computetablewithdists(training, p):
    dists = np.zeros(training.num_rows)
    attributes = training.drop('Class').rows
    for i in range(training.num_rows):
        dists[i] = distance(attributes[i], p)
    withdists = training.copy()
    withdists.append_column('Distance', dists)
    return withdists

def closest(training, p, k):
    withdists = computetablewithdists(training, p)
    sortedbydist = withdists.sort('Distance')
    topk = sortedbydist.take(range(k))
    return topk

def majority(topkclasses):
    if topkclasses.where('Class', 1).num_rows > topkclasses.where('
        return 1
    else:
        return 0

def classify(training, p, k):
    closestk = closest(training, p, k)
    topkclasses = closestk.select('Class')
    return majority(topkclasses)

```



```

def evaluate_accuracy(training, valid, k):
    validattrs = valid.drop('Class')
    numcorrect = 0
    for i in range(valid.num_rows):
        # Run the classifier on the ith patient in the test set
        c = classify(training, validattrs.rows[i], k)
        # Was the classifier's prediction correct?
        if c == valid['Class'][i]:
            numcorrect = numcorrect + 1
    return numcorrect / valid.num_rows

```

## Code for feature selection

Now we'll implement the feature selection algorithm. First, a subroutine to evaluate the accuracy when using a particular subset of features:

```
def evaluate_features(training, valid, features, k):
    tr = training.select(['Class']+features)
    va = valid.select(['Class']+features)
    return evaluate_accuracy(tr, va, k)
```

Next, we'll implement a subroutine that, given a current subset of features, tries all possible ways to add one more feature to the subset, and evaluates the accuracy of each candidate. This returns a table that summarizes the accuracy of each option it examined.

```
def try_one_more_feature(training, valid, baseattrs, k):
    results = Table(['Attribute', 'Accuracy'])
    for attr in training.drop(['Class']+baseattrs).labels:
        acc = evaluate_features(training, valid, [attr]+baseattrs,
                               results.append((attr, acc)))
    return results.sort('Accuracy', descending=True)
```

Finally, we'll implement the greedy feature selection algorithm, using the above subroutines. For our own purposes of understanding what's going on, I'm going to have it print out, at each iteration, all features it considered and the accuracy it got with each.

```

def select_features(training, valid, k, maxfeatures=3):
    results = Table(['NumAttrs', 'Attributes', 'Accuracy'])
    curattrs = []
    iters = min(maxfeatures, len(training.labels)-1)
    while len(curattrs) < iters:
        print('== Computing best feature to add to '+str(curattrs))
        # Try all ways of adding just one more feature to curattrs
        r = try_one_more_feature(training, valid, curattrs, k)
        r.show()
        print()
        # Take the single best feature and add it to curattrs
        attr = r['Attribute'][0]
        acc = r['Accuracy'][0]
        curattrs.append(attr)
        results.append((len(curattrs), ', '.join(curattrs), acc))
    return results

```



## Example: Tree Cover

Now let's try it out on an example. I'm working with a data set gathered by the US Forestry service. They visited thousands of wilderness locations and recorded various characteristics of the soil and land. They also recorded what kind of tree was growing predominantly on that land. Focusing only on areas where the tree cover was either Spruce or Lodgepole Pine, let's see if we can figure out which characteristics have the greatest effect on whether the predominant tree cover is Spruce or Lodgepole Pine.

There are 500,000 records in this data set -- more than I can analyze with the software we're using. So, I'll pick a random sample of just a fraction of these records, to let us do some experiments that will complete in a reasonable amount of time.

```

all_trees = Table.read_table('treecover2.csv.gz', sep=', ')
training = all_trees.take(range(0, 1000))
validation = all_trees.take(range(1000, 1500))
test = all_trees.take(range(1500, 2000))

```

```
training.show(2)
```

Elevation	Aspect	Slope	HorizDistToWater	VertDistToWater	HorizDistToWater
2804	139	9	268	65	3180
2785	155	18	242	118	3090

... (998 rows omitted)

Let's start by figuring out how accurate a classifier will be, if trained using this data. I'm going to arbitrarily use  $k = 15$  for the  $k$ -nearest neighbor classifier.

```
evaluate_accuracy(training, validation, 15)
```

```
0.572
```

Now we'll apply feature selection. I wonder which characteristics have the biggest influence on whether Spruce vs Lodgepole Pine grows? We'll look for the best 4 features.

```
best_features = select_features(training, validation, 15)
```

```
== Computing best feature to add to []
```

<b>Attribute</b>	<b>Accuracy</b>
Elevation	0.762
HorizDistToRoad	0.562
HorizDistToFire	0.534
Hillshade9am	0.518
Hillshade3pm	0.516
Slope	0.516
Area4	0.514
Area3	0.514
Area2	0.514
Area1	0.514
VertDistToWater	0.512
HillshadeNoon	0.51
Aspect	0.51
HorizDistToWater	0.502

```
-- Computing best feature to add to ['Elevation']
```

<b>Attribute</b>	<b>Accuracy</b>
HorizDistToWater	0.782
Hillshade9am	0.776
Slope	0.77
Hillshade3pm	0.768
Area4	0.762
Area3	0.762
Area2	0.762
Area1	0.762
VertDistToWater	0.762
Aspect	0.75
HillshadeNoon	0.748
HorizDistToRoad	0.73
HorizDistToFire	0.664

```
== Computing best feature to add to ['Elevation', 'HorizDistToWater']
```

Attribute	Accuracy
Hillshade3pm	0.792
HillshadeNoon	0.792
Slope	0.784
Aspect	0.784
Area4	0.782
Area3	0.782
Area2	0.782
Area1	0.782
Hillshade9am	0.78
VertDistToWater	0.776
HorizDistToRoad	0.708
HorizDistToFire	0.64

```
best_features
```

NumAttrs	Attributes	Accuracy
1	Elevation	0.762
2	Elevation, HorizDistToWater	0.782
3	Elevation, HorizDistToWater, Hillshade3pm	0.792

As we can see, Elevation looks like far and away the most discriminative feature. This suggests that this characteristic might play a large role in the biology of which tree grows best, and thus might tell us something about the science.

What about the horizontal distance to water and the amount of shade at 3pm? It looks like they might also be predictive, and thus might play a role in the biology as well -- assuming the apparent improvement in accuracy is real, and we're not just fooling ourselves.

## Hold-out sets: Training, Validation, Testing

Suppose we built a predictor using just the best two features, Elevation and HorizDistToWater. How accurate would we expect it to be, on the entire population of locations? 78.2% accurate? more? less? Why?

Well, at this point, it's hard to tell. It's the same issue we mentioned earlier about fooling ourselves. We've tried multiple different approaches, and taken the best; if we then evaluate it on the same data set we used to select which is best, we will get a biased numbers -- it might not be an accurate estimate of the true accuracy.

Why? Our data set is noisy. We've looked for correlations, and kept the association that had the highest correlation in our training set. But was that a real relationship, or was it just noise? If you pick a single attribute and measure its accuracy on a sample, we'd expect this to be a reasonable approximation to its accuracy on the entire population, but with some random error. If we look at 100 possible combinations and choose the best, it's possible we found one whose accuracy on the entire population is indeed large -- or it's possible we were just selecting the one whose error term happened to be the largest of the 100.

For these reasons, we can't expect the accuracy numbers we've computed above to necessarily be a good, unbiased measure of the accuracy on the entire population.

The way to get an unbiased estimate of accuracy is the same as last lecture: get some more data; or set some aside in the beginning so we have more when we need it. In this case, I set aside two extra chunks of data, a *validation* data set and a *test* data set. I used the validation set to select a few best features. Now we're going to measure the performance of this on the test set, just to see what happens.

```
evaluate_features(training, validation, ['Elevation'], 15)
```

```
0.762
```

```
evaluate_features(training, test, ['Elevation'], 15)
```

```
0.712
```

```
evaluate_features(training, validation, ['Elevation', 'HorizDistToW
```

```
0.782
```

```
evaluate_features(training, test, ['Elevation', 'HorizDistToWater'])
```

0.742

Why do you think we see this difference?

To get a better feeling for this, we can look at all of our different features, and compare its accuracy on the training set vs its accuracy on the validation set, to see how much random error there is in these accuracy figures. If there was no error, then both accuracy numbers would always be the same, but as we'll see, in practice there is some error, because computing the accuracy on a sample is only an approximation to the accuracy on the entire population.

```
accuracies = Table(['Validation Accuracy', 'Accuracy on another sample'])
another_sample = all_trees.take(range(2000, 2500))
for feature in training.drop('Class').labels:
    x = evaluate_features(training, validation, [feature], 15)
    y = evaluate_features(training, another_sample, [feature], 15)
    accuracies.append((x,y))
accuracies.sort('Validation Accuracy')
```

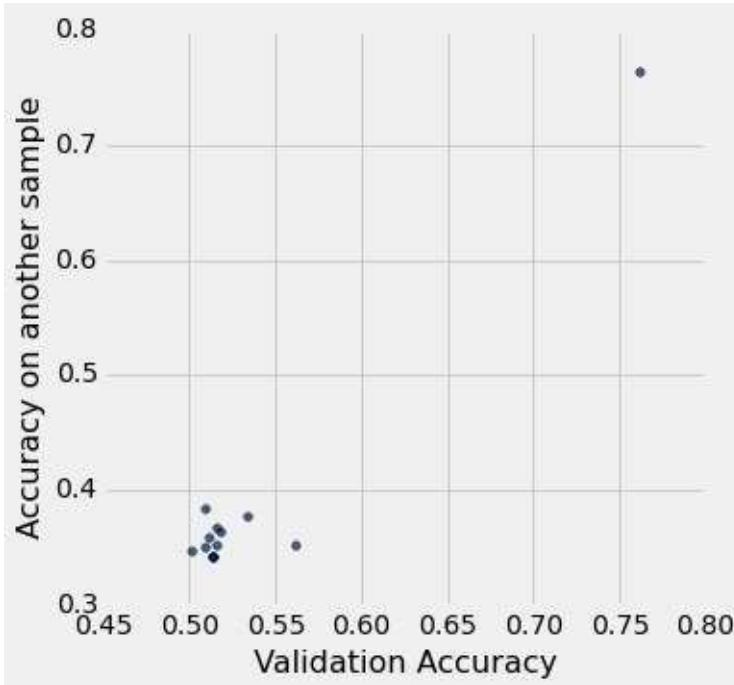
1 | ▶

Validation Accuracy	Accuracy on another sample
0.502	0.346
0.51	0.384
0.51	0.35
0.512	0.358
0.514	0.342
0.514	0.342
0.514	0.342
0.514	0.342
0.516	0.366
0.516	0.352

... (4 rows omitted)

```
accuracies.scatter('Validation Accuracy')
```

```
/opt/conda/lib/python3.4/site-packages/matplotlib/collections.py:59
    if self._edgecolors == str('face'):
```



## Thought Questions

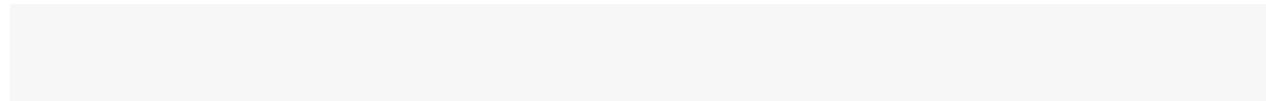
Suppose that the top two attributes had been Elevation and HorizDistToRoad. Interpret this for me. What might this mean for the biology of trees? One possible explanation is that the distance to the nearest road affects what kind of tree grows; can you give any other possible explanations?

Once we know the top two attributes are Elevation and HorizDistToWater, suppose we next wanted to know *how* they affect what kind of tree grows: e.g., does high elevation tend to favor spruce, or does it favor lodgepole pine? How would you go about answering these kinds of questions?

The scientists also gathered some more data that I left out, for simplicity: for each location, they also gathered what kind of soil it has, out of 40 different types. The original data set had a column for soil type, with numbers from 1-40 indicating which of the 40 types of soil was present. Suppose I wanted to include this among the other characteristics. What would go wrong, and how could I fix it up?

For this example we picked  $k = 15$  arbitrarily. Suppose we wanted to pick the best value of  $k$  -- the one that gives the best accuracy. How could we go about doing that? What are the pitfalls, and how could they be addressed?

Suppose I wanted to use feature selection to help me adjust my online dating profile picture to get the most responses. There are some characteristics I can't change (such as how handsome I am), and some I can (such as whether I smile or not). How would I adjust the feature selection algorithm above to account for this?



# **Chapter 5**

## **Inference**

# Confidence Intervals

## Interact

Whenever we analyze a random sample of a population, our goal is to draw robust conclusions about the whole population, despite the fact that we can measure only part of it. We must guard against being misled by the inevitable random variations in a sample. It is possible that a sample has very different characteristics than the population itself, but those samples are unlikely. It is much more typical that a random sample, especially a large simple random sample, is representative of the population. More importantly, the chance that a sample resembles the population is something that we know in advance, based on how we selected our random sample. Statistical inference is the practice of using this information to make precise statements about a population based on a random sample.

## Statistics and Parameters

The [Bay Area Bike Share](#) service published a [dataset](#) describing every bicycle rental from September 2014 to August 2015 in their system. This set of all trips is a population, and fortunately we have the data for each and every trip rather than just a sample. We'll focus only on the *free trips*, which are trips that last less than 1800 seconds (half an hour).

```
free_trips = Table.read_table("trip.csv").where('Duration', are.below_or_equal_to(1800))
```

Start Station	End Station	Duration
Harry Bridges Plaza (Ferry Building)	San Francisco Caltrain (Townsend at 4th)	765
San Antonio Shopping Center	Mountain View City Hall	1036
Post at Kearny	2nd at South Park	307
San Jose City Hall	San Salvador at 1st	409
Embarcadero at Folsom	Embarcadero at Sansome	789
Yerba Buena Center of the Arts (3rd @ Howard)	San Francisco Caltrain (Townsend at 4th)	293
Embarcadero at Folsom	Embarcadero at Sansome	896
Embarcadero at Sansome	Steuart at Market	255
Beale at Market	Temporary Transbay Terminal (Howard at Beale)	126
Post at Kearny	South Van Ness at Market	932

... (338333 rows omitted)

A quantity measured for a whole population is called a *parameter*. For example, the average duration for any free trip between September 2014 to August 2015 can be computed exactly from these data: 550.00.

```
np.average(free_trips.column('Duration'))
```

```
550.00143345658103
```

Although we have information for the full population, we will investigate what we can learn from a simple random sample. Below, we sample 100 trips without replacement from `free_trips` not just once, but 40 different times. All 4000 trips are stored in a table called `samples`.

```
n = 100
samples = Table(['Sample #', 'Duration'])
for i in np.arange(40):
    for trip in free_trips.sample(n).rows:
        samples.append([i, trip.item('Duration')])
samples
```

Sample #	Duration
0	557
0	491
0	877
0	279
0	339
0	542
0	715
0	255
0	1543
0	194

... (3990 rows omitted)

A quantity measured for a sample is called a *statistic*. For example, the average duration of a free trip in a sample is a statistic, and we find that there is some variation across samples.

```
for i in np.arange(3):
    sample_average = np.average(samples.where('Sample #', i).column)
    print("Average duration in sample", i, "is", sample_average)
```

```
Average duration in sample 0 is 547.29
Average duration in sample 1 is 500.6
Average duration in sample 2 is 660.16
```

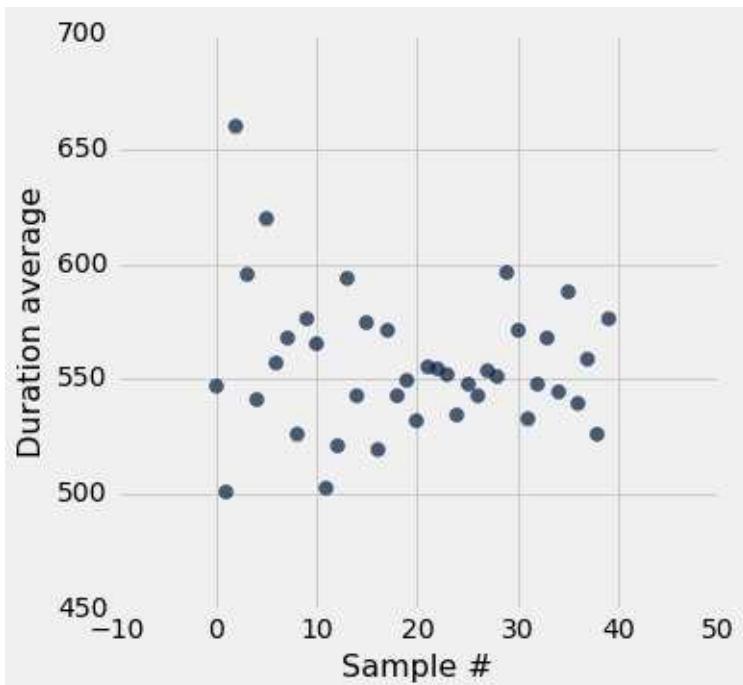
Often, the goal of analysis is to estimate parameters from statistics. Proper statistical inference involves more than just picking an estimate: we should also say something about how confident we should be in that estimate. A single guess is rarely sufficient to make a robust conclusion about a population. We also need to quantify in some way how precise we believe that guess to be.

Already we can see from the example above that the average duration of a sample is around 550, the parameter. Our goal is to quantify this notion of *around*, and one way to do so is to state a range of values.

## Error in Estimates

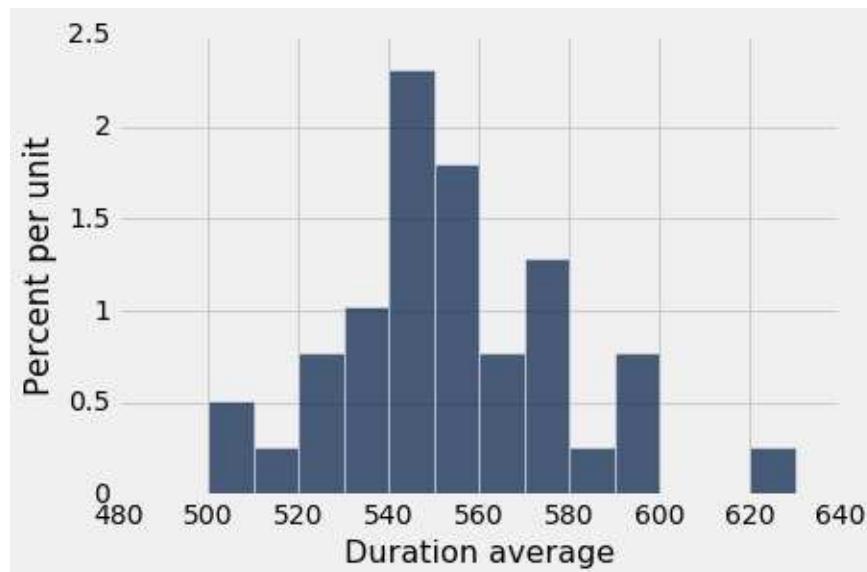
We can learn a great deal about the behavior of a statistic by computing it for many samples. In the scatter diagram below, the order in which the statistic was computed appears on the horizontal axis, and the value of the average duration for a sample of  $n$  trips appears on the vertical axis.

```
samples.groupby(0, np.average).scatter(0, 1, s=50)
```



The sample number doesn't tell us anything beyond when the sample was selected, and we can see from this unstructured cloud of points that one sample does not affect the next in any consistent way. The sample averages are indeed all around 550. Some are above, and some are below.

```
samples.groupby(0, np.average).hist(1, bins=np.arange(480, 641, 10))
```



There are 40 averages for 40 samples. If we find an interval that contains the middle 38 out of 40, it will contain 95% of the samples. One such interval is bounded by the second smallest and the second largest sample averages among the 40 samples.

```
averages = np.sort(samples.groupby(0, np.average).column(1))
lower = averages.item(1)
upper = averages.item(38)
print('Empirically, 95% of sample averages fell within the interval
```

```
Empirically, 95% of sample averages fell within the interval 502.38
```

This statement gives us our first notion of a margin of error. If this statement were to hold up not just for our 40 samples, but for all samples, then a reasonable strategy for estimating the true population average would be to draw a 100-trip sample, compute its sample average, and guess that the true average was within `max_deviation` of this quantity. We would be right at least 95% of the time.

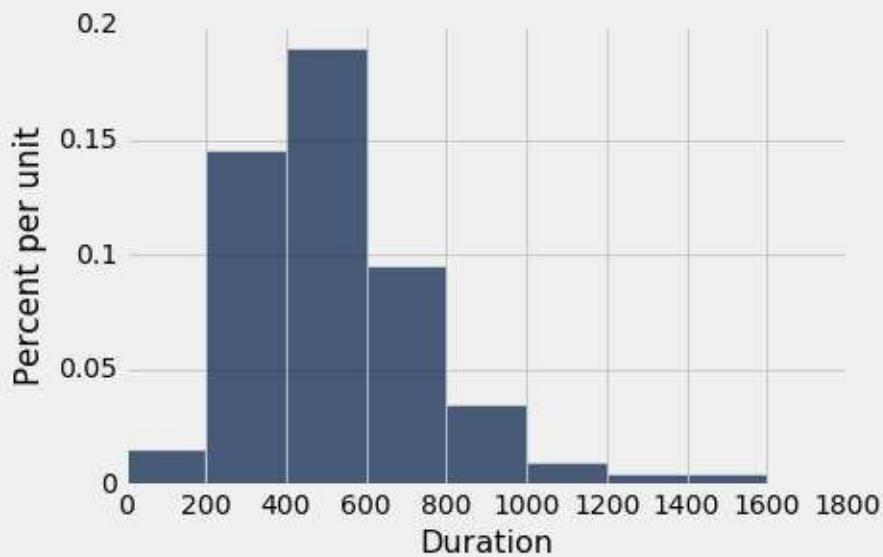
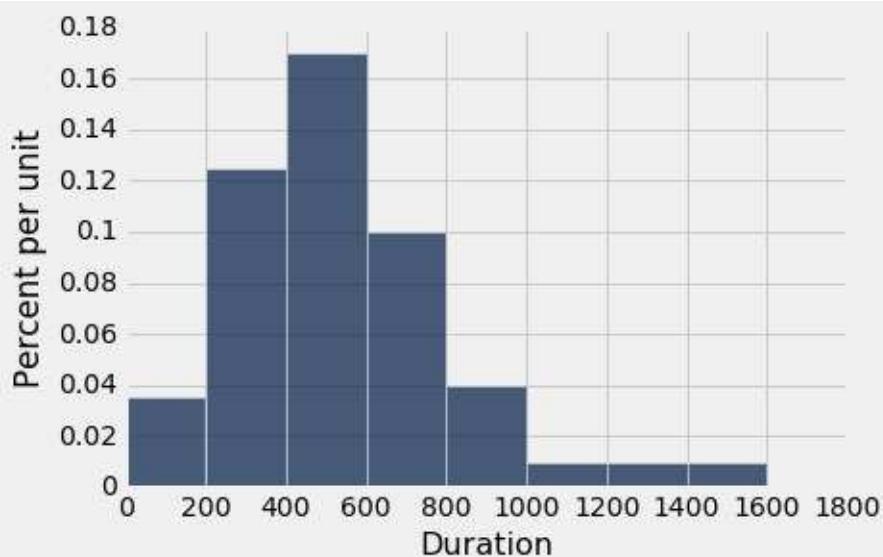
The problem with this approach is that finding the `max_deviation` required drawing many different samples, and we would like to be able to say something similar after having collected only one 100-trip sample.

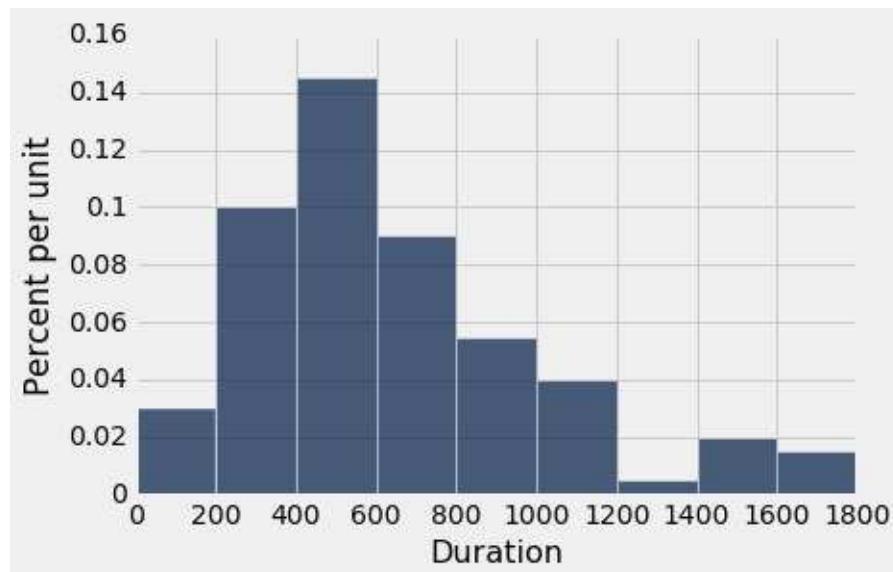
## Variability Within Samples

By contrast, The individual durations in the samples themselves are not closely clustered around 550. Instead, they span the whole range from 0 to 1800. The durations for the first three 100-item samples are visualized below. All have somewhat similar shapes; they were

drawn from the same population.

```
for i in np.arange(3):
    samples.where('Sample #', i).hist(1, bins=np.arange(0, 1801, 200))
```





We have now observed two different ways in which the variation of a random sample can be observed: the statistics computed from those samples vary in magnitude, and the sampled quantities themselves vary in distribution. That is, there is variability *across samples*, which we see by comparing sample averages, and there is variability *within each sample*, which we see in the duration histograms above.

## Percentiles

The 80th percentile of a collection of values is the smallest value in the collection that is at least as large as 80% of the values in the collection. Likewise, the 40th percentile is at least as large as 40% of the values.

For example, let's consider the sizes of the five largest continents: Africa, Antarctica, Asia, North America, and South America, rounded to the nearest million square miles.

```
sizes = [12, 17, 6, 9, 7]
```

The 20th percentile is the smallest value that is at least as large as 20% or one fifth of the elements.

```
percentile(20, sizes)
```

6

The 60th percentile is at least as large as 60% or three fifths of the elements.

```
percentile(60, sizes)
```

```
9
```

To find the element corresponding to a percentile  $p$ , sort the elements and take the  $k$ th element, where  $k = \frac{p}{100} \times n$  where  $n$  is the size of the collection. In general, any percentile from 0 to 100 can be computed for a collection of values, and it is always an element of the collection. If  $k$  is not an integer, round it up to find the percentile element. For example, the 90th percentile of a five element set is the fifth element, because  $\frac{90}{100} \times 5$  is 4.5, which rounds up to 5.

```
percentile(90, sizes)
```

```
17
```

When the `percentile` function is called with a single argument, it returns a function that computes percentiles of collections.

```
tenth = percentile(10)
tenth(sizes)
```

```
6
```

```
tenth(np.arange(50))
```

```
4
```

```
tenth(np.arange(5000))
```

```
499
```

# Confidence Intervals

Inference begins with a population: a collection that you can describe but often cannot measure directly. One thing we can often do with a population is to draw a simple random sample. By measuring a statistic of the sample, we may hope to estimate a parameter of the population. In this process, the population is not random, nor are its parameters. The sample is random, and so is its statistic. The whole process of estimation therefore gives a random result. It starts from a population (fixed), draws a sample (random), computes a statistic (random), and then estimates the parameter from the statistic (random). By calling the process random, we mean that the final estimate could have been different because the sample could have been different.

A confidence interval is one notion of a margin of error around an estimate. The margin of error acknowledges that the estimation process could have been misleading because of the variation in the sample. It gives a range that might contain the original parameter, along with a percentage of confidence. For example, a 95% confidence interval is one in which we would expect the interval to contain the true parameter for 95% of random samples. For any particular sample, any particular estimate, and any particular interval, the parameter is either contained or not; we never know for sure. The purpose of the confidence interval is to quantify how often our estimation process gives us a range containing the truth.

There are many different ways to compute a confidence interval. The challenge in computing a confidence interval from a single sample is that we don't know the population, and so we don't know how its samples will vary. One way to make progress despite this limited information is to assume that the variability *within the sample* is similar to the variability within the population.

## Resampling from a Sample

Even though we happen to have the whole population of `free_trips`, let's imagine for a moment that we have only a single 100-trip sample from this population. That sample has 100 durations. Our goal now is to learn what we can about the population using just this sample.

```
sample = free_trips.sample(n)
sample
```

Start Station	End Station	Duration
Grant Avenue at Columbus Avenue	San Francisco Caltrain 2 (330 Townsend)	877
Embarcadero at Folsom	San Francisco Caltrain (Townsend at 4th)	597
Beale at Market	Beale at Market	67
2nd at Folsom	Washington at Kearny	714
Clay at Battery	Embarcadero at Sansome	510
South Van Ness at Market	Powell Street BART	494
San Jose Diridon Caltrain Station	MLK Library	530
Mechanics Plaza (Market at Battery)	Clay at Battery	286
South Van Ness at Market	Civic Center BART (7th at Market)	172
Broadway St at Battery St	5th at Howard	551

... (90 rows omitted)

```
average = np.average(sample.column('Duration'))
average
```

518.5599999999995

A technique called *bootstrap resampling* uses the variability of the sample as a proxy for the variability of the population. Instead of drawing many samples from the population, we will draw many samples from the sample. Since the original sample and the re-sampled sample have the same size, we must draw with replacement in order to see any variability at all.

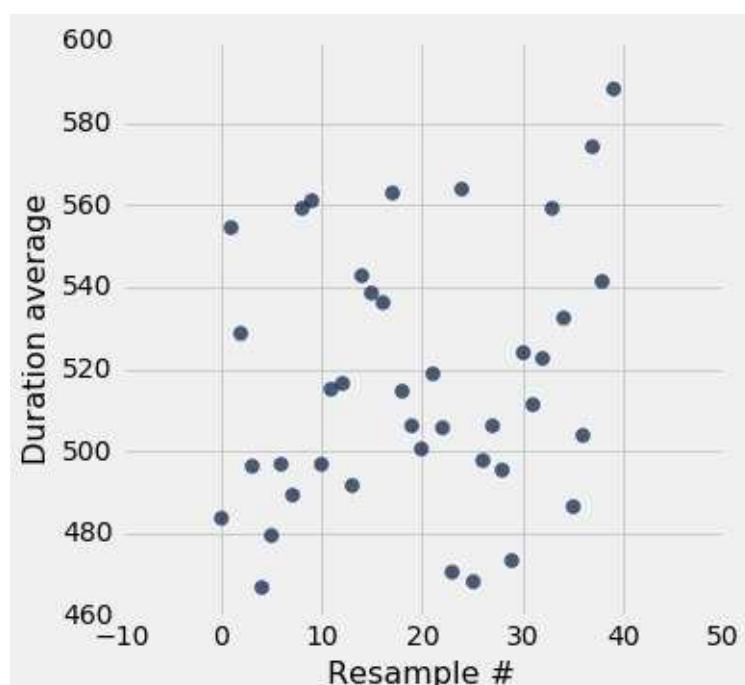
```
resamples = Table(['Resample #', 'Duration'])
for i in np.arange(40):
    resample = sample.sample(n, with_replacement=True)
    for trip in resample.rows:
        resamples.append([i, trip.item('Duration')])
resamples
```

Resample #	Duration
0	329
0	252
0	252
0	685
0	434
0	396
0	434
0	585
0	394
0	202

... (3990 rows omitted)

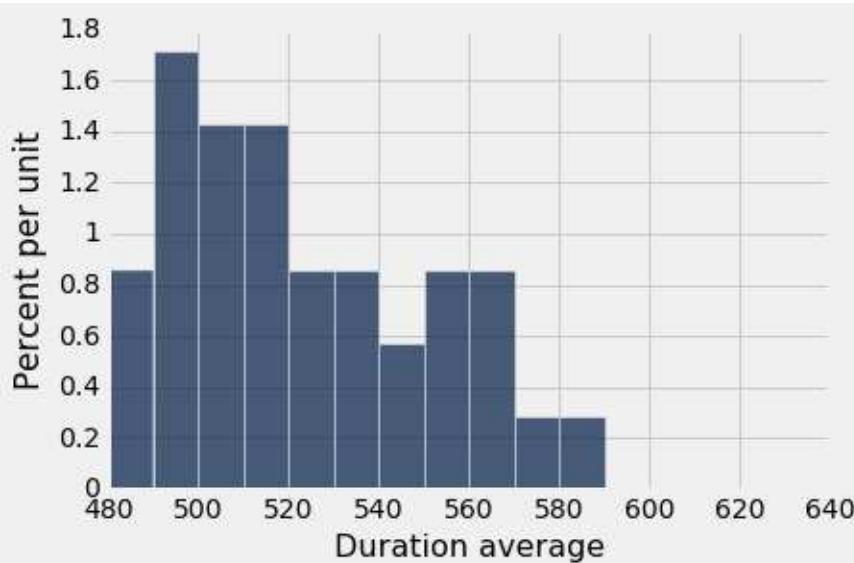
Using the `resamples`, we can investigate the variability of the corresponding sample averages.

```
resamples.groupby(0, np.average).scatter(0, 1, s=50)
```



These sample averages are not necessarily clustered around the true average 550. Instead, they will be clustered around the average of the sample from which they were re-sampled.

```
resamples.groupby(0, np.average).hist(1, bins=np.arange(480, 641, 10))
```



Although the center of this distribution is not the same as the true population average, its variability is similar.

## Bootstrap Confidence Interval

A technique called *bootstrap resampling* estimates a confidence interval using resampling. To find a confidence interval, we must estimate how much the sample statistic deviates from the population parameter. The bootstrap assumes that a resampled statistic will deviate from the sample statistic in approximately the same way that the sample statistic deviates from the parameter.

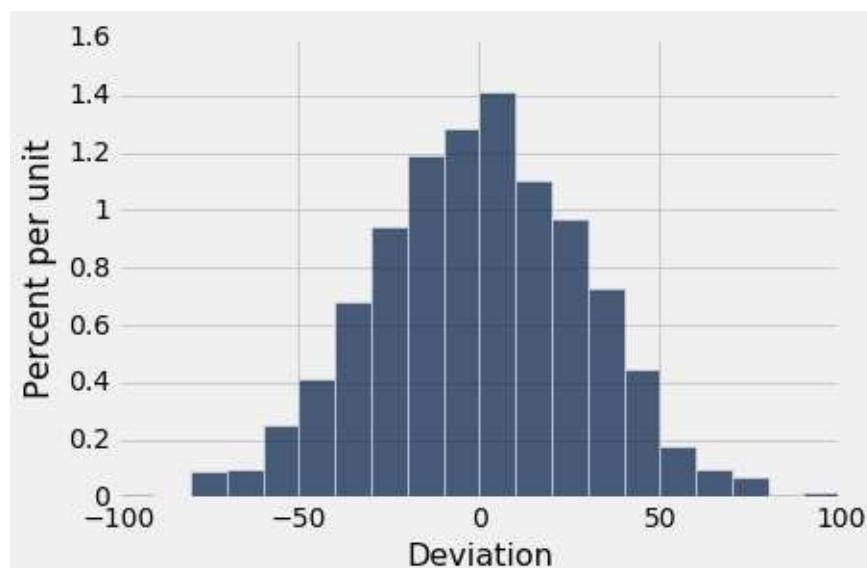
To carry out the technique, we first compute the deviations from the sample statistic (in this case, the sample average) for many resampled samples. The number is arbitrary, and it doesn't require any collecting of new data; 1000 or more is typical.

```
deviations = Table(['Resample #', 'Deviation'])
for i in np.arange(1000):
    resample = sample.sample(n, with_replacement=True)
    dev = average - np.average(resample.column(2))
    deviations.append([i, dev])
deviations
```

Resample #	Deviation
0	30.95
1	41.16
2	3.37
3	2.54
4	-22.87
5	27.63
6	49.93
7	11.91
8	26.32
9	-5.97
... (990 rows omitted)	

A deviation of zero indicates that the resampled statistic is the same as the sampled statistic. That's evidence that the sample statistic is perhaps the same as the population parameter. A positive deviation is evidence that the sample statistic may be greater than the parameter. As we can see, for averages, the distribution of deviations is roughly symmetric.

```
deviations.hist(1, bins=np.arange(-100, 101, 10))
```



A 95% confidence interval is computed based on the middle 95% of this distribution of deviations. The max and min deviations are computed by the 97.5th percentile and 2.5th percentile respectively, because the span between these contains 95% of the distribution.

```
lower = average - percentile(97.5, deviations.column(1))
lower
```

```
461.93000000000001
```

```
upper = average - percentile(2.5, deviations.column(1))
upper
```

```
576.2999999999995
```

Now, we have not only an estimate, but an interval around it that expresses our uncertainty about the value of the parameter.

```
print('A 95% confidence interval around', average, 'is', lower, 'to', upper)
```

```
A 95% confidence interval around 518.56 is 461.93 to 576.3
```

And behold, the interval happens to contain the true parameter 550. Normally, after generating a confidence interval, there is no way to check that it is correct, because to do so requires access to the entire population. In this case, we started with the whole population in a single table, and so we are able to check.

The following function encapsulates the entire process of producing a confidence interval for the population average using a single sample. Each time it is run, even for the same sample, the interval will be slightly different because of the random variation of resampling. However, the upper and lower bounds generated by 2000 samples for this problem are typically quite similar for multiple calls.

```
def average_ci(sample, label):
    deviations = Table(['Resample #', 'Deviation'])
    n = sample.num_rows
    average = np.average(sample.column(label))
    for i in np.arange(1000):
        resample = sample.sample(n, with_replacement=True)
        dev = np.average(resample.column(label)) - average
        deviations.append([i, dev])
    return (average - percentile(97.5, deviations.column(1)),
            average - percentile(2.5, deviations.column(1)))

average_ci(sample, 'Duration')
```

```
(457.4399999999994, 572.6399999999987)
```

```
average_ci(sample, 'Duration')
```

```
(458.739999999999, 575.5099999999988)
```

However, if we draw an entirely new sample, then the confidence interval will change.

```
average_ci(free_trips.sample(n), 'Duration')
```

```
(530.1800000000006, 661.2100000000004)
```

The interval is about the same width, and that's typical of multiple samples of the same size drawn from the same population. However, if a much larger sample is drawn, then the confidence interval that results is typically going to be much smaller. We're not losing confidence in this case; we're just using more evidence to perform inference, and so we have less uncertainty about the full population.

```
average_ci(free_trips.sample(16 * n), 'Duration')
```

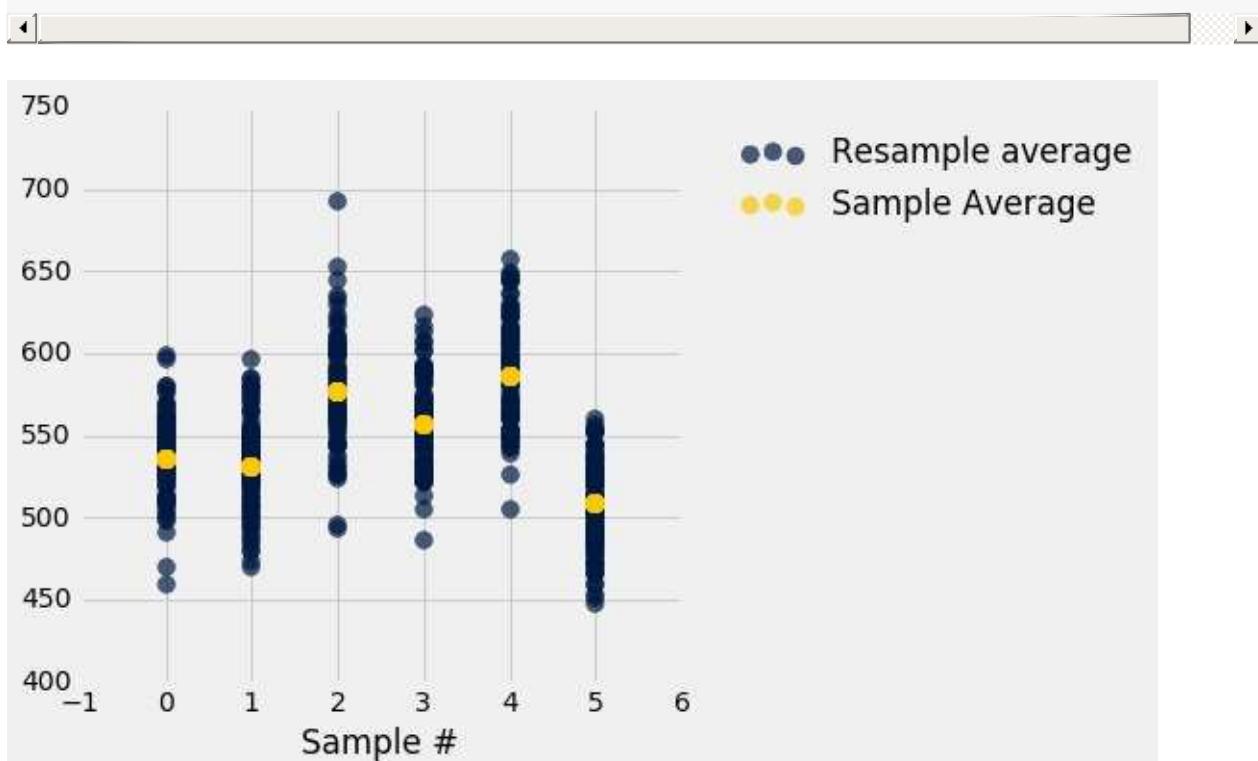
```
(527.18937499999993, 555.11437499999988)
```

## Evaluating Confidence Intervals

When given only a sample, it is not possible to check whether the parameter was estimated correctly. However, in the case of the Bay Area Bike Share, we have not only the sample, but many samples and the whole population. Therefore, we can check how often the confidence intervals we compute in this way do in fact contain the true parameter.

Before measuring the accuracy of our technique, we can visualize the result of resampling each sample. The following scatter diagram has 6 vertical lines for 6 samples, and each line consists of 100 points that represent the statistics from resamples for that sample.

```
samples = Table(['Sample #', 'Resample average', 'Sample Average'])
for i in np.arange(6):
    sample = free_trips.sample(n)
    average = np.average(sample.column('Duration'))
    for j in np.arange(100):
        resample = sample.sample(n, with_replacement=True)
        resample_average = np.average(resample.column('Duration'))
        samples.append([i, resample_average, average])
samples.scatter(0, s=80)
```



We can see from the plot above that for almost every sample, there are some resampled averages above 550 and some below. When we use these resampled averages to compute confidence intervals, many of those intervals contain 550 as a result.

Below, we draw 100 more samples and show the upper and lower bounds of the confidence intervals computed from each one.

```
intervals = Table(['Sample #', 'Lower', 'Estimate', 'Upper'])
for i in np.arange(100):
    sample = free_trips.sample(n)
    average = np.average(sample.column('Duration'))
    lower, upper = average_ci(sample, 'Duration')
    intervals.append([i, lower, average, upper])
```

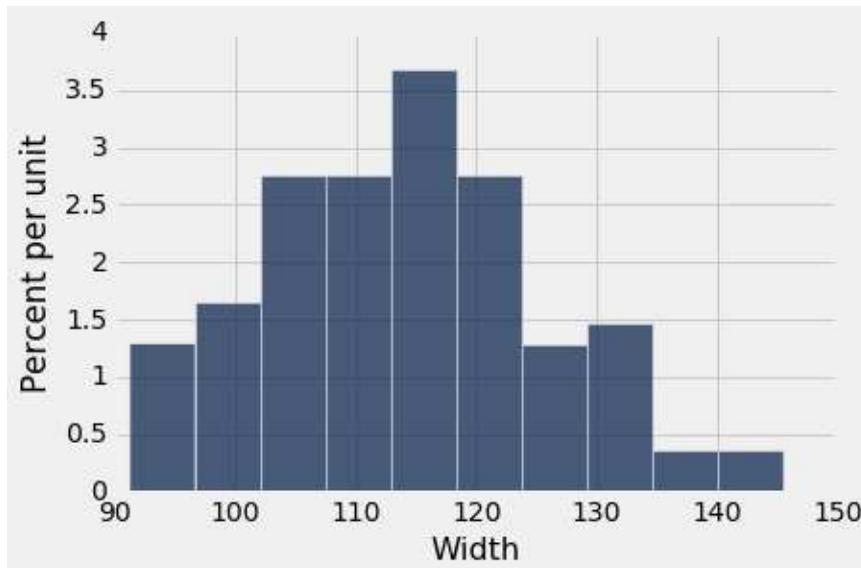
We can compute precisely which intervals contain the truth, and we should find that about 95 of them do.

```
correct = np.logical_and(intervals.column('Lower') <= 550, intervals.column('Upper') >= 550)
np.count_nonzero(correct)
```

97

Each confidence interval is generated from only a single sample of 100 trips. As a result, the width of the interval varies quite a bit as well. Compared to the interval width we computed originally (which required multiple samples), we see that some of these intervals are much more narrow and some are much wider.

```
Table().with_column('Width', intervals.column('Upper')-intervals.column('Lower'))
```



Given a single sample, it is impossible to know whether you have correctly estimated the parameter or not. However, you can infer a precise notion of confidence by following a process based on random sampling.

# Distance Between Distributions

## Interact

By now we have seen many examples of situations in which one distribution appears to be quite close to another, such as a population and its sample. The goal of this section is to quantify the difference between two distributions. This will allow our analyses to be based on more than the assessments that we are able to make by eye.

For this, we need a measure of the distance between two distributions. We will develop such a measure in the context of a [study](#) conducted in 2010 by the American Civil Liberties Union (ACLU) of Northern California.

The focus of the study was the racial composition of jury panels in Alameda County. A jury panel is a group of people chosen to be prospective jurors; the final trial jury is selected from among them. Jury panels can consist of a few dozen people or several thousand, depending on the trial. By law, a jury panel is supposed to be representative of the community in which the trial is taking place. Section 197 of California's Code of Civil Procedure says, "All persons selected for jury service shall be selected at random, from a source or sources inclusive of a representative cross section of the population of the area served by the court."

The final jury is selected from the panel by deliberate inclusion or exclusion: the law allows potential jurors to be excused for medical reasons; lawyers on both sides may strike a certain number of potential jurors from the list in what are called "peremptory challenges"; the trial judge might make a selection based on questionnaires filled out by the panel; and so on. But the initial panel is supposed to resemble a random sample of the population of eligible jurors.

The ACLU of Northern California compiled data on the racial composition of the jury panels in 11 felony trials in Alameda County in the years 2009 and 2010. In those panels, the total number of people who reported for jury service was 1453. The ACLU gathered demographic data on all of these prospective jurors, and compared those data with the composition of all eligible jurors in the county.

The data are tabulated below in a table called `jury`. For each race, the first value is the percentage of all eligible juror candidates of that race. The second value is the percentage of people of that race among those who appeared for the process of selection into the jury.

```

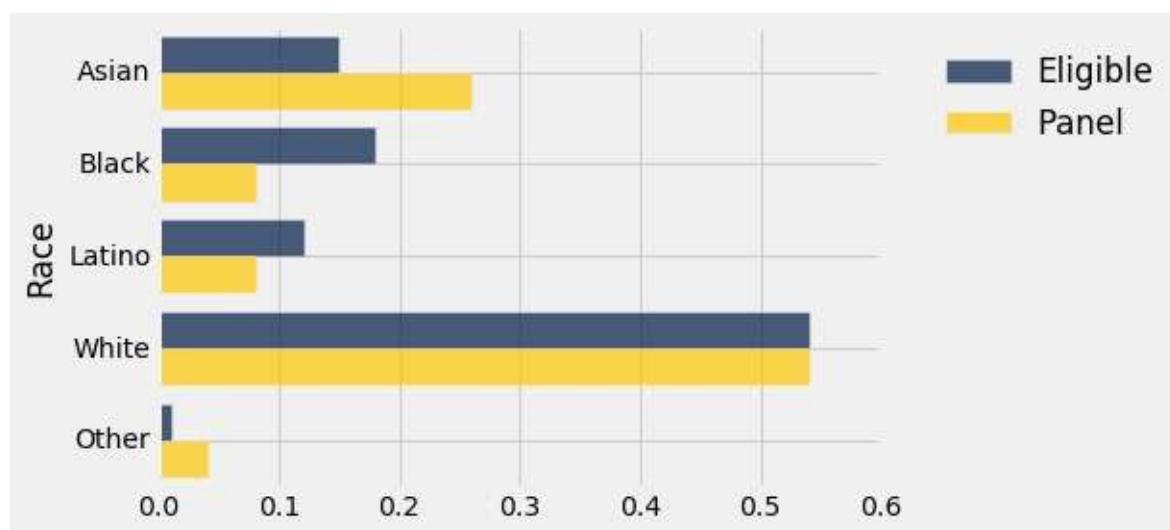
jury = Table(["Race", "Eligible", "Panel"]).with_rows([
    ["Asian", 0.15, 0.26],
    ["Black", 0.18, 0.08],
    ["Latino", 0.12, 0.08],
    ["White", 0.54, 0.54],
    ["Other", 0.01, 0.04],
])
jury.set_format([1, 2], PercentFormatter(0))

```

Race	Eligible	Panel
Asian	15%	26%
Black	18%	8%
Latino	12%	8%
White	54%	54%
Other	1%	4%

Some races are overrepresented and some are underrepresented on the jury panels in the study. A bar graph is helpful for visualizing the differences.

```
jury.bahr('Race')
```



## Total Variation Distance

To measure the difference between the two distributions, we will compute a quantity called the *total variation distance* between them. To compute the total variation distance, take the difference between the two proportions in each category, add up the absolute values of all the differences, and then divide the sum by 2.

Here are the differences and the absolute differences:

```
jury.append_column("Difference", jury.column("Panel") - jury.column("Eligible"))
jury.append_column("Abs. Difference", np.abs(jury.column("Difference")))
jury.set_format([3, 4], PercentFormatter(0))
```

Race	Eligible	Panel	Difference	Abs. Difference
Asian	15%	26%	11%	11%
Black	18%	8%	-10%	10%
Latino	12%	8%	-4%	4%
White	54%	54%	0%	0%
Other	1%	4%	3%	3%

And here is the sum of the absolute differences, divided by 2:

```
sum(jury.column(3)) / 2
```

```
1.3877787807814457e-17
```

The total variation distance between the distribution of eligible jurors and the distribution of the panels is 0.14. Before we examine the numerical value, let us examine the reasons behind the steps in the calculation.

It is quite natural to compute the difference between the proportions in each category. Now take a look at the column `diff` and notice that the sum of its entries is 0: the positive entries add up to 0.14, exactly canceling the total of the negative entries which is -0.14. The proportions in each of the two columns `Panel` and `Eligible` add up to 1, and so the give-and-take between their entries must add up to 0.

To avoid the cancellation, we drop the negative signs and then add all the entries. But this gives us two times the total of the positive entries (equivalently, two times the total of the negative entries, with the sign removed). So we divide the sum by 2.

We would have obtained the same result by just adding the positive differences. But our method of including all the absolute differences eliminates the need to keep track of which differences are positive and which are not.

The following function computes the total variation distance between two columns of a table.

```
def total_variation_distance(column, other):
    return sum(np.abs(column - other)) / 2

def table_tvd(table, label, other):
    return total_variation_distance(table.column(label), table.column(other))

table_tvd(jury, 'Eligible', 'Panel')
```

0.14000000000000001

## Are the panels representative of the population?

We will now turn to the numerical value of the total variation distance between the eligible jurors and the panel. How can we interpret the distance of 0.14? To answer this, let us recall that the panels are supposed to be selected at random. It will therefore be informative to compare the value of 0.14 with the total variation distance between the eligible jurors and a randomly selected panel.

To do this, we will employ our skills at simulation. There were 1453 prospective jurors in the panels in the study. So let us take a random sample of size 1453 from the population of eligible jurors.

**[Technical note.]** Random samples of prospective jurors would be selected without replacement. However, when the size of a sample is small relative to the size of the population, sampling without replacement resembles sampling with replacement; the proportions in the population don't change much between draws. The population of eligible jurors in Alameda County is over a million, and compared to that, a sample size of about 1500 is quite small. We will therefore sample with replacement.]

The `np.random.multinomial` function draws a random sample uniformly with replacement from a population whose distribution is categorical. Specifically, `np.random.multinomial` takes as its input a sample size and an array consisting of the probabilities of choosing

different categories. It returns the count of the number of times each category was chosen.

```
sample_size = 1453

random_panel = np.random.multinomial(sample_size, jury["Eligible"])
random_panel
```

```
array([208, 265, 166, 805,     9])
```

```
sum(random_panel)
```

```
1453
```

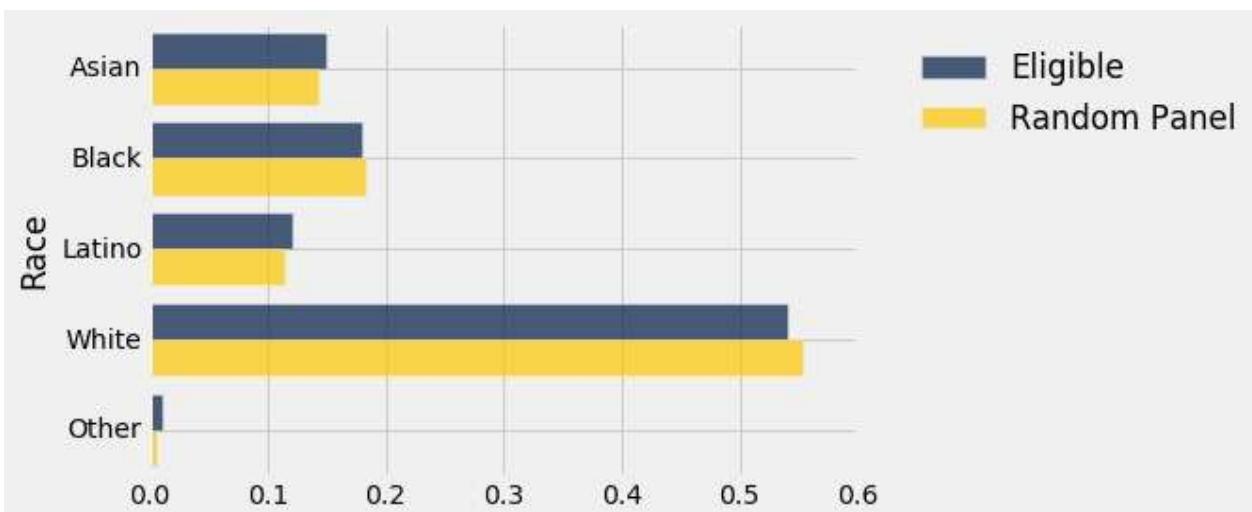
We can compare this distribution with the distribution of eligible jurors, by converting the counts to proportions. To do this, we will divide the counts by the sample size. It is clear from the results that the two distributions are quite similar.

```
sampled = jury.select(['Race', 'Eligible', 'Panel']).with_column(
    'Random Panel', random_panel / sample_size)
sampled.set_format(3, PercentFormatter(0))
```

Race	Eligible	Panel	Random Panel
Asian	15%	26%	14%
Black	18%	8%	18%
Latino	12%	8%	11%
White	54%	54%	55%
Other	1%	4%	1%

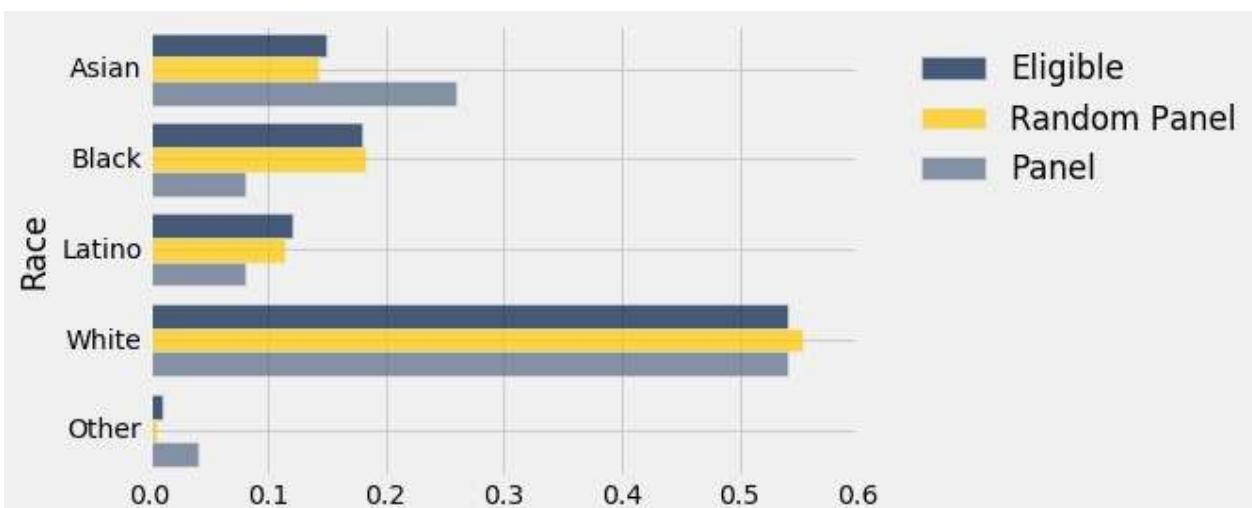
As always, it helps to visualize. The population and sample are quite similar.

```
sampled.barch('Race', [1, 3])
```



When we compare also to the panel, the difference is stark.

```
sampled.bah('Race', [1, 3, 2])
```



The total variation distance between the population distribution and the sample distribution is quite small.

```
table_tvd(sampled, 'Eligible', 'Random Panel')
```

```
0.016407432897453545
```

Comparing this to the distance of 0.14 for the panel quantifies our observation that the random sample is close to the distribution of eligible jurors, and the panel is relatively far from the distribution of eligible jurors.

However, the distance between the random sample and the eligible jurors depends on the sample; sampling again might give a different result.

## How do random samples differ from the population?

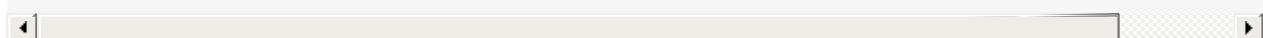
The total variation distance between the distribution of the random sample and the distribution of the eligible jurors is the statistic that we are using to measure the distance between the two distributions. By repeating the process of sampling, we can measure how much the statistic varies across different random samples. The code below computes the empirical distribution of the statistic based on a large number of replications of the sampling process.

```
# Compute empirical distribution of TVDs

sample_size = 1453
repetitions = 1000
eligible = jury.column('Eligible')
tvds = Table(["TVD from a random sample"])

for i in np.arange(repetitions):
    sample = np.random.multinomial(sample_size, eligible) / sample_
    tvd = sum(abs(sample - eligible)) / 2
    tvds.append([tvd])

tvds
```



**TVD from a random sample**

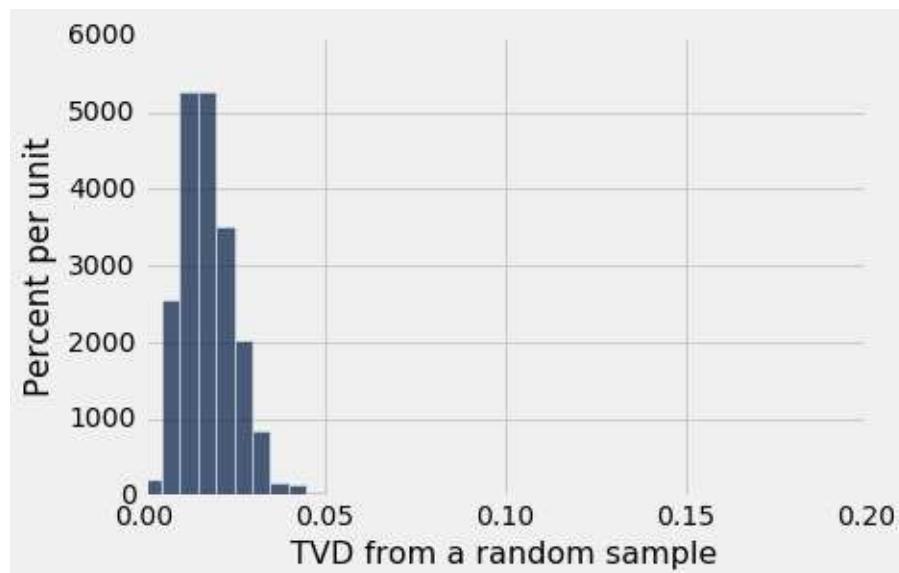
0.0207571
0.0286098
0.0201721
0.0308672
0.0185134
0.0102546
0.0119133
0.0346731
0.0271576
0.0216862

... (990 rows omitted)

Each row of the column above contains the total variation distance between a random sample and the population of eligible jurors.

The histogram of this column shows that drawing 1453 jurors at random from the pool of eligible candidates results in a distribution that rarely deviates from the eligible jurors' race distribution by more than 0.05.

```
tvds.hist(bins=np.arange(0, 0.2, 0.005))
```



## How do the panels compare to random samples?¶

The panels in the study, however, were not quite so similar to the eligible population. The total variation distance between the panels and the population was 0.14, which is far out in the tail of the histogram above. It does not look like a typical distance between a random sample and the eligible population.

Our analysis supports the ACLU's conclusion that the panels were not representative of the population. The ACLU report discusses several reasons for the discrepancies. For example, some minority groups were underrepresented on the records of voter registration and of the Department of Motor Vehicles, the two main sources from which jurors are selected; at the time of the study, the county did not have an effective process for following up on prospective jurors who had been called but had failed to appear; and so on. Whatever the reasons, it seems clear that the composition of the jury panels was different from what we would have expected in a random sample.

## A Classical Interlude: the Chi-Squared Statistic¶

"Do the data look like a random sample?" is a question that has been asked in many contexts for many years. In classical data analysis, the statistic most commonly used in answering this question is called the  $\chi^2$  ("chi squared") statistic, and is calculated as follows:

**Step 1.** For each category, define the "expected count" as follows:

$$\text{expected count} = \frac{\text{sample size}}{\text{proportion in population}}$$

This is the count that you would expect in that category, in a randomly chosen sample.

**Step 2.** For each category, compute

$$\frac{(\text{observed count} - \text{expected count})^2}{\text{expected count}}$$

**Step 3.** Add up all the numbers computed in Step 2.

A little algebra shows that this is equivalent to:

**Alternative Method, Step 1.** For each category, compute

$$\frac{(\text{sample proportion} - \text{population proportion})^2}{\text{population proportion}}$$

**Alternative Method, Step 2.** Add up all the numbers you computed in the step above, and multiply the sum by the sample size.

It makes sense that the statistic should be based on the difference between proportions in each category, just as the total variation distance is. The remaining steps of the method are not very easy to explain without getting deeper into mathematics.

The reason for choosing this statistic over any other was that statisticians were able to come up a formula for its approximate probability distribution when the sample size is large. The distribution has an impressive name: it is called the  $\chi^2$  *distribution with degrees of freedom equal to 4* (one fewer than the number of categories). Data analysts would compute the  $\chi^2$  statistic for their sample, and then compare it with tabulated numerical values of the  $\chi^2$  distributions.

### Simulating the $\chi^2$ statistic.

The  $\chi^2$  statistic is just a statistic like any other. We can use the computer to simulate its behavior even if we have not studied all the underlying mathematics.

For the panels in the ACLU study, the  $\chi^2$  statistic is about 348:

```
sum(((jury["Panel"] - jury["Eligible"])**2)/jury["Eligible"])*sample_size
```

348.07422222222226

To generate the empirical distribution of the statistic, all we need is a small modification of the code we used for simulating total variation distance:

```
# Compute empirical distribution of chi-squared statistic

classical_from_sample = Table(["'Chi-squared' statistic, from a random sample"])
for i in np.arange(repetitions):
    sample = np.random.multinomial(sample_size, eligible)/sample_size
    chisq = sum(((sample - eligible)**2)/eligible)*sample_size
    classical_from_sample.append([chisq])

classical_from_sample
```

### 'Chi-squared' statistic, from a random sample

5.29816

1.83597

5.03969

4.3795

10.738

4.74311

4.15047

5.83918

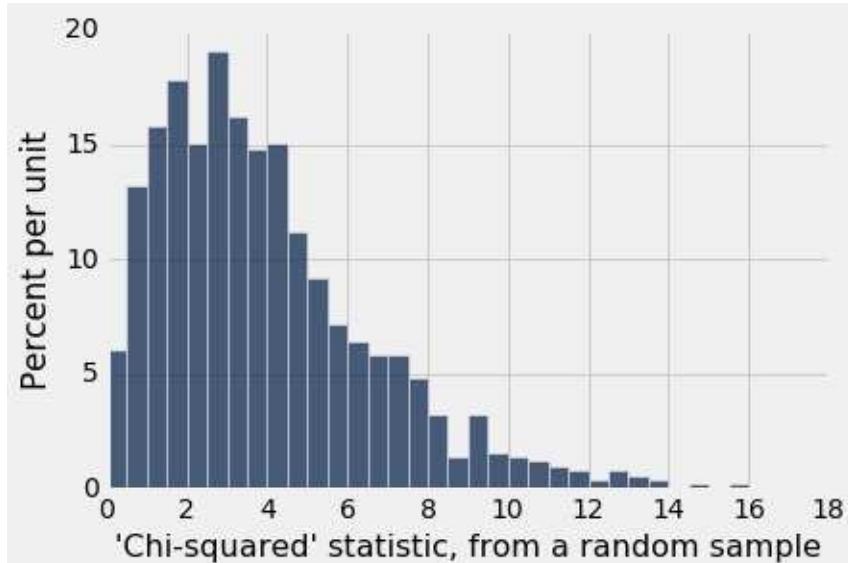
9.6604

0.127756

... (990 rows omitted)

Here is a histogram of the empirical distribution of the  $\chi^2$  statistic. The simulated values of  $\chi^2$  based on random samples are considerably smaller than the value of 348 that we got from the jury panels.

```
classical_from_sample.hist(bins=np.arange(0, 18, 0.5), normed=True)
```



In fact, the long run average value of  $\chi^2$  statistics is equal to the degrees of freedom. And indeed, the average of the simulated values of  $\chi^2$  is close to 4, the degrees of freedom in this case.

```
np.mean(classical_from_sample.column(0))
```

3.978631566873136

In this situation, random samples are expected to produce a  $\chi^2$  statistic of about 4. The panels, on the other hand, produced a  $\chi^2$  statistic of 348. This is yet more confirmation of the conclusion that the panels do not resemble a random sample from the population of eligible jurors.

# Hypothesis Testing

## Interact

Our investigation of jury panels is an example of *hypothesis testing*. We wished to know the answer to a yes-or-no question about the world, and we reasoned about random samples in order to answer the question. In this section, we formalize the process.

## Sampling from a Distribution

The rows of a table describe individual elements of a collection of data. In the previous section, we worked with a table that described a distribution over categories. Tables that describe the proportions or counts of different categories in a sample or population arise often in data analysis as a summary of a data collection process. The `sample_from_distribution` method draws a sample of counts for the categories. Its implementation uses the same `np.random.multinomial` method from the previous section.

The table below describes the proportion of the eligible potential jurors in Alameda County (estimated from 2000 census data and other sources) for broad categories of race and ethnic background. This table was compiled by Professor Weeks, a demographer at San Diego State University, for the Alameda County trial *People v. Stuart Alexander* in 2002.

```
population = Table(["Race", "Eligible"]).with_rows([
    ["Asian", 0.15],
    ["Black", 0.18],
    ["Latino", 0.12],
    ["White", 0.54],
    ["Other", 0.01],
])
population
```

Race	Eligible
Asian	0.15
Black	0.18
Latino	0.12
White	0.54
Other	0.01

The `sample_from_distribution` method takes a number of samples and a column index or label. It returns a table in which the distribution column has been replaced by category counts of the sample.

```
sample_size = 1453
population.sample_from_distribution("Eligible", sample_size)
```

Race	Eligible	Eligible sample
Asian	0.15	233
Black	0.18	245
Latino	0.12	177
White	0.54	787
Other	0.01	11

Each count is selected randomly in such a way that the chance of each category is the corresponding proportion in the original "Eligible" column. Sampling again will give different counts, but again the "white" category is much more common than "other" because of its much higher proportion.

```
population.sample_from_distribution("Eligible", sample_size)
```

Race	Eligible	Eligible sample
Asian	0.15	227
Black	0.18	247
Latino	0.12	155
White	0.54	819
Other	0.01	5

The option `proportions=True` draws sample counts and then divides each count by the total number of samples. The result is another distribution; one based on a sample.

```
sample = population.sample_from_distribution("Eligible", sample_size)
sample
```

Race	Eligible	Eligible sample
Asian	0.15	0.140399
Black	0.18	0.189264
Latino	0.12	0.125258
White	0.54	0.532691
Other	0.01	0.0123882

This ut sample can also be used to generate a sample. The sample of a sample is called a *resampled sample* or a *bootstrap sample*. It is not a random sample of the original distribution, but shares many characteristics with such a sample.

```
sample.sample_from_distribution('Eligible sample', sample_size)
```

Race	Eligible	Eligible sample	Eligible sample sample
Asian	0.15	0.140399	204
Black	0.18	0.189264	258
Latino	0.12	0.125258	163
White	0.54	0.532691	810
Other	0.01	0.0123882	18

## Empirical Distributions

An *empirical distribution* of a statistic is a table generated by computing the statistic for many samples. The previous section used empirical distributions to reason about whether or not a sample had a statistic that was typical of a random sample. The following function computes the empirical histogram of a statistic, which is expressed as a function that takes a sample.

```
def empirical_distribution(table, label, sample_size, k, f):
    stats = Table(['Sample #', 'Statistic'])
    for i in np.arange(k):
        sample = table.sample_from_distribution(label, sample_size)
        statistic = f(sample)
        stats.append([i, statistic])
    return stats
```

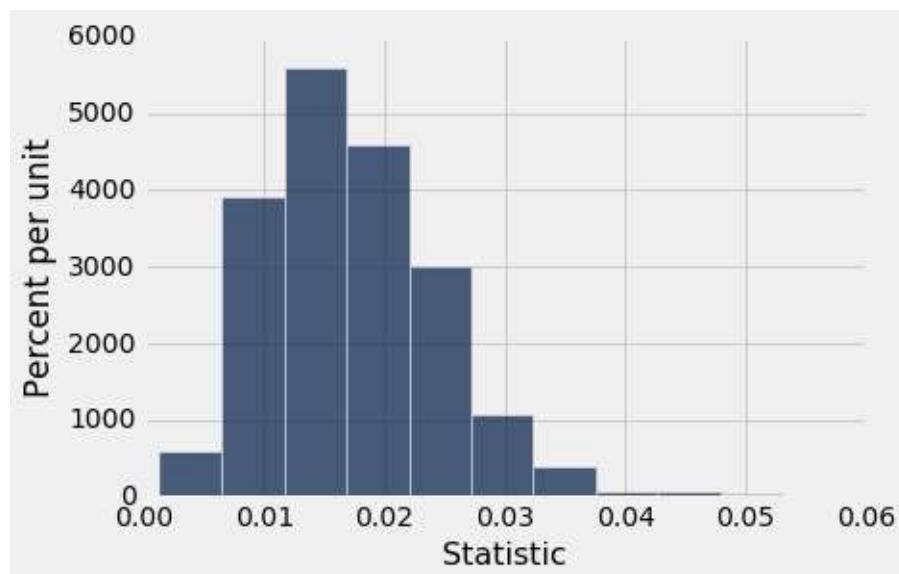
This function can be used to compute an empirical distribution of the total variation distance from the population distribution.

```
def tvd_from_eligible(sample):
    counts = sample.column('Eligible sample')
    distribution = counts / sum(counts)
    return total_variation_distance(population.column('Eligible'),
                                    empirical_distribution(population, 'Eligible', sample_size, 1000, t
```

Sample #	Statistic
0	0.0248796
1	0.0195664
2	0.00401927
3	0.0165864
4	0.0207502
5	0.013042
6	0.0297041
7	0.0153544
8	0.00830695
9	0.014384
... (990 rows omitted)	

The empirical distribution can be visualized using a histogram.

```
empirical_distribution(population, 'Eligible', sample_size, 1000, t
```



## U.S. Supreme Court, 1965: Swain vs. Alabama [¶](#)

In the early 1960's, in Talladega County in Alabama, a black man called Robert Swain was convicted of raping a white woman and was sentenced to death. He appealed his sentence, citing among other factors the all-white jury. At the time, only men aged 21 or older were allowed to serve on juries in Talladega County. In the county, 26% of the eligible jurors were black, but there were only 8 black men among the 100 selected for the jury panel in Swain's trial. No black man was selected for the trial jury.

In 1965, the Supreme Court of the United States denied Swain's appeal. In its ruling, the Court wrote "... the overall percentage disparity has been small and reflects no studied attempt to include or exclude a specified number of [black men]."

Let us use the methods we have developed to examine the disparity between 8 out of 100 and 26 out of 100 black men in a panel of 100 drawn at random from among the eligible jurors.

```
alabama = Table(['Race', 'Eligible']).with_rows([
    ["Black", 0.26],
    ["Other", 0.74]
])
alabama.set_format(1, PercentFormatter(0))
```

Race	Eligible
Black	26%
Other	74%

As our test statistic, we will use the number of black men in a random sample of size 100. Here is an empirical distribution of this statistic.

```
def value_for(category, category_label, value_label):
    def in_table(sample):
        return sample.where(category_label, category).column(value_label)
    return in_table

num_black = value_for('Black', 'Race', 'Eligible sample')

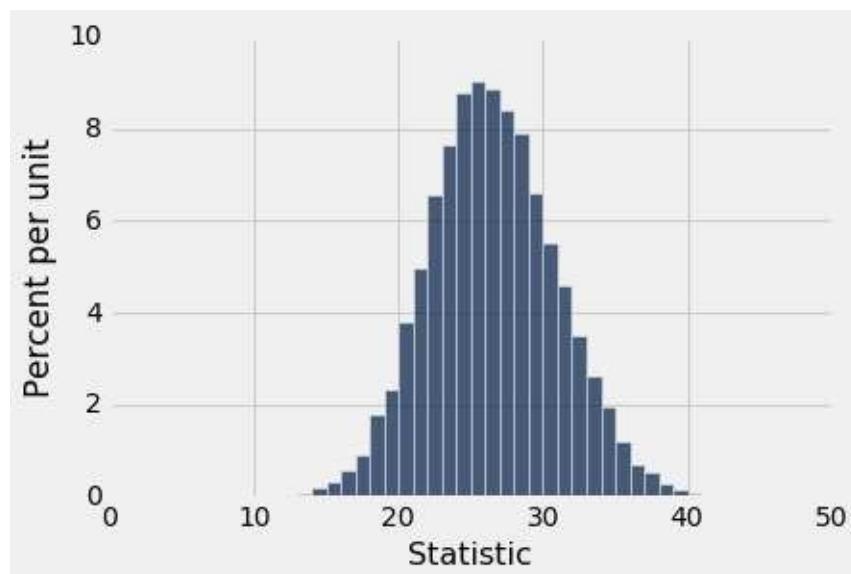
black_in_sample = empirical_distribution(alabama, 'Eligible', 100,
                                         num_black)
```

Sample #	Statistic
0	27
1	31
2	28
3	27
4	24
5	23
6	29
7	28
8	16
9	25

... (9990 rows omitted)

The numbers of black men in the first 10 repetitions were quite a bit larger than 8. The empirical histogram below shows the distribution of the test statistic over all the repetitions.

```
black_in_sample.hist(1, bins=np.arange(0, 50, 1))
```



If the 100 men in Swain's jury panel had been chosen at random, it would have been extremely unlikely for the number of black men on the panel to be as small as 8. We must conclude that the percentage disparity was larger than the disparity expected due to chance variation.

## Method and Terminology of Statistical Tests of Hypotheses

We have developed some of the fundamental concepts of statistical tests of hypotheses, in the context of examples about jury selection. Using statistical tests as a way of making decisions is standard in many fields and has a standard terminology. Here is the sequence of the steps in most statistical tests, along with some terminology and examples.

### STEP 1: THE HYPOTHESES

All statistical tests attempt to choose between two views of how the data were generated. These two views are called *hypotheses*.

**The null hypothesis.** This says that the data were generated at random under clearly specified assumptions that make it possible to compute chances. The word "null" reinforces the idea that if the data look different from what the null hypothesis predicts, the difference is due to nothing but chance.

In both of our examples about jury selection, the null hypothesis is that the panels were selected at random from the population of eligible jurors. Though the racial composition of the panels was different from that of the populations of eligible jurors, there was no reason for the difference other than chance variation.

**The alternative hypothesis.** This says that some reason other than chance made the data differ from what was predicted by the null hypothesis. Informally, the alternative hypothesis says that the observed difference is "real."

In both of our examples about jury selection, the alternative hypothesis is that the panels were not selected at random. Something other than chance led to the differences between the racial composition of the panels and the racial composition of the populations of eligible jurors.

## STEP 2: THE TEST STATISTIC

In order to decide between the two hypothesis, we must choose a statistic upon which we will base our decision. This is called the **test statistic**.

In the example about jury panels in Alameda County, the test statistic we used was the total variation distance between the racial distributions in the panels and in the population of eligible jurors. In the example about Swain versus the State of Alabama, the test statistic was the number of black men on the jury panel.

Calculating the observed value of the test statistic is often the first computational step in a statistical test. In the case of jury panels in Alameda County, the observed value of the total variation distance between the distributions in the panels and the population was 0.14. In the example about Swain, the number of black men on his jury panel was 8.

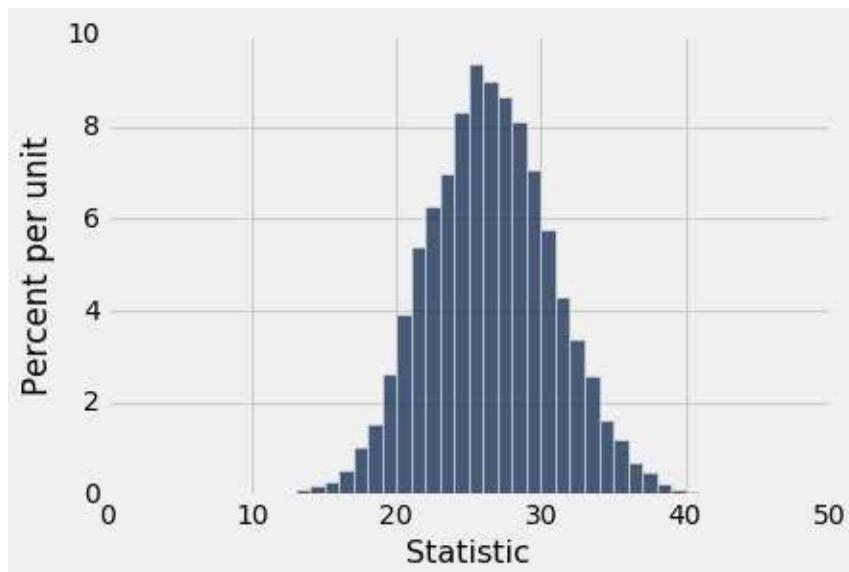
## STEP 3: THE PROBABILITY DISTRIBUTION OF THE TEST STATISTIC, UNDER THE NULL HYPOTHESIS

This step sets aside the observed value of the test statistic, and instead focuses on *what the value might be if the null hypothesis were true*. Under the null hypothesis, the sample could have come out differently due to chance. So the test statistic could have come out differently. This step consists of figuring out all possible values of the test statistic and all their probabilities, under the null hypothesis of randomness.

In other words, in this step we calculate the probability distribution of the test statistic pretending that the null hypothesis is true. For many test statistics, this can be a daunting task both mathematically and computationally. Therefore, we approximate the probability distribution of the test statistic by the empirical distribution of the statistic based on a large number of repetitions of the sampling procedure.

This was the empirical distribution of the test statistic – the number of black men on the jury panel – in the example about Swain and the Supreme Court:

```
black_in_sample.hist(1, bins=np.arange(0, 50, 1))
```



#### STEP 4. THE CONCLUSION OF THE TEST

The choice between the null and alternative hypotheses depends on the comparison between the results of Steps 2 and 3: the observed test statistic and its distribution as predicted by the null hypothesis.

If the two are consistent with each other, then the observed test statistic is in line with what the null hypothesis predicts. In other words, the test does not point towards the alternative hypothesis; the null hypothesis is better supported by the data.

But if the two are not consistent with each other, as is the case in both of our examples about jury panels, then the data do not support the null hypothesis. In both our examples we concluded that the jury panels were not selected at random. Something other than chance affected their composition.

**How is "consistent" defined?** Whether the observed test statistic is consistent with its predicted distribution under the null hypothesis is a matter of judgment. We recommend that you provide your judgment along with the value of the test statistic and a graph of its predicted distribution. That will allow your reader to make his or her own judgment about whether the two are consistent.

If you do not want to make your own judgment, there are conventions that you can follow. These conventions are based on what is called the **observed significance level** or *P-value* for short. The P-value is a chance computed using the probability distribution of the test statistic, and can be approximated by using the empirical distribution in Step 3.

**Practical note on P-values and conventions.** Place the observed test statistic on the horizontal axis of the histogram, and find the proportion in the tail starting at that point. That's the P-value.

If the P-value is small, the data support the alternative hypothesis. The conventions for what is "small":

- If the P-value is less than 5%, the result is called "statistically significant."
- If the P-value is even smaller – less than 1% – the result is called "highly statistically significant."

**More formal definition of P-value.** The P-value is the chance, under the null hypothesis, that the test statistic is equal to the value that was observed or is even further in the direction of the alternative.

The P-value is based on comparing the observed test statistic and what the null hypothesis predicts. A small P-value implies that under the null hypothesis, the value of the test statistic is unlikely to be near the one we observed. When a hypothesis and the data are not in accord, the hypothesis has to go. That is why we reject the null hypothesis if the P-value is small.

The P-value for the null hypothesis that Swain's jury panel was selected at random from the population of Talladega is approximated using the empirical distribution as follows:

```
np.count_nonzero(black_in_sample.column(1) <= 8) / black_in_sample.
```

```
0.0
```

## HISTORICAL NOTE ON THE CONVENTIONS

The determination of statistical significance, as defined above, has become standard in statistical analyses in all fields of application. When a convention is so universally followed, it is interesting to examine how it arose.

The method of statistical testing – choosing between hypotheses based on data in random samples – was developed by Sir Ronald Fisher in the early 20th century. Sir Ronald might have set the convention for statistical significance somewhat unwittingly, in the following statement in his 1925 book *Statistical Methods for Research Workers*. About the 5% level, he wrote, "It is convenient to take this point as a limit in judging whether a deviation is to be considered significant or not."

What was "convenient" for Sir Ronald became a cutoff that has acquired the status of a universal constant. No matter that Sir Ronald himself made the point that the value was his personal choice from among many: in an article in 1926, he wrote, "If one in twenty does not seem high enough odds, we may, if we prefer it draw the line at one in fifty (the 2 percent point), or one in a hundred (the 1 percent point). Personally, the author prefers to set a low standard of significance at the 5 percent point ..."

Fisher knew that "low" is a matter of judgment and has no unique definition. We suggest that you follow his excellent example. Provide your data, make your judgment, and explain why you made it.

# Permutation

[Interact](#)

## Comparing Two Groups

In the examples above, we investigated whether a sample appears to be chosen randomly from an underlying population. We did this by comparing the distribution of the sample with the distribution of the population. A similar line of reasoning can be used to compare the distributions of two samples. In particular, we can investigate whether or not two samples appear to be drawn from the same underlying distribution.

### Example: Married Couples and Unmarried Partners

Our next example is based on a study conducted in 2010 under the auspices of the National Center for Family and Marriage Research.

In the United States, the proportion of couples who live together but are not married has been rising in recent decades. The study involved a national random sample of over 1,000 heterosexual couples who were either married or "cohabiting partners" – living together but unmarried. One of the goals of the study was to compare the attitudes and experiences of the married and unmarried couples.

The table below shows a subset of the data collected in the study. Each row corresponds to one person. The variables that we will examine in this section are:

- Marital Status: married or unmarried
- Employment Status: one of several categories described below
- Gender
- Age: Age in years

```
columns = ['Marital Status', 'Employment Status', 'Gender', 'Age',
couples = Table.read_table('couples.csv').select(columns)
couples
```

Marital Status	Employment Status	Gender	Age
married	working as paid employee	male	51
married	working as paid employee	female	53
married	working as paid employee	male	57
married	working as paid employee	female	57
married	working as paid employee	male	60
married	working as paid employee	female	57
married	working, self-employed	male	62
married	working as paid employee	female	59
married	not working - other	male	53
married	not working - retired	female	61

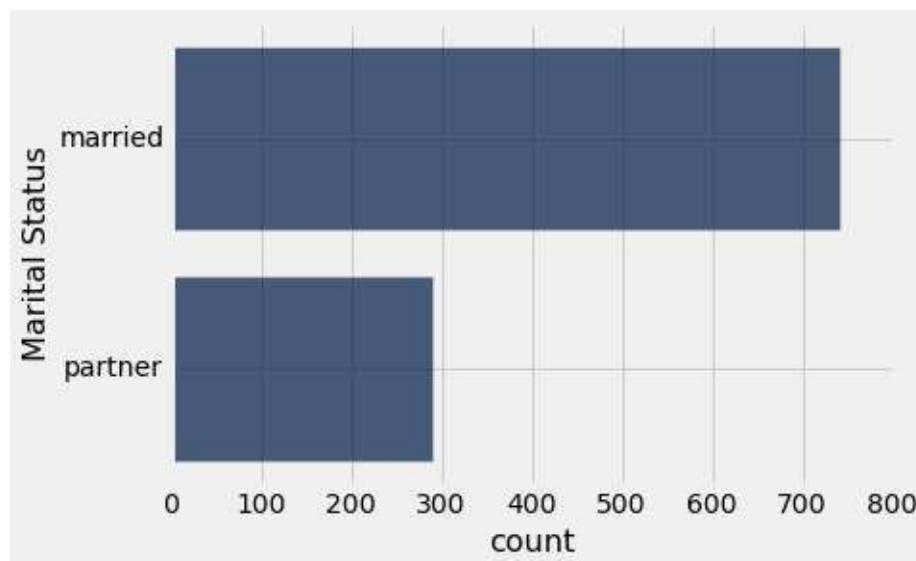
... (2056 rows omitted)

Let us consider just the males first. There are 742 married couples and 292 unmarried couples, and all couples in this study had one male and one female, making 1,034 males in all.

```
# Separate tables for married and cohabiting unmarried couples:

married_men = couples.where('Gender', 'male').where('Marital Status'
partnered_men = couples.where('Gender', 'male').where('Marital Stat

# Let's see how many married and unmarried people there are:
couples.where('Gender', 'male').group('Marital Status').barh("Marit
```



Societal norms have changed over the decades, and there has been a gradual acceptance of couples living together without being married. Thus it is natural to expect that unmarried couples will in general consist of younger people than married couples.

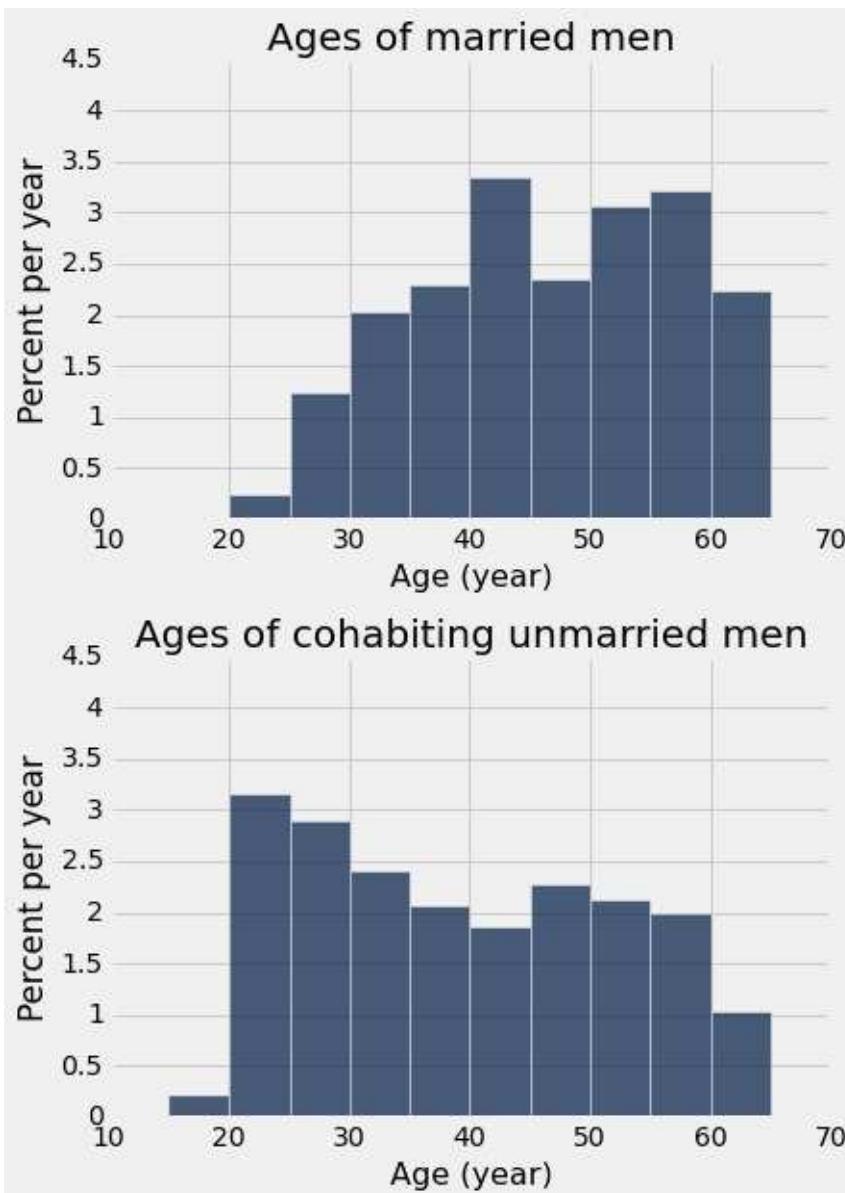
The histograms of the ages of the married and unmarried men show that this is indeed the case. We will draw these histograms and compare them. In order to compare two histograms, both should be drawn to the same scale. Let us write a function that does this for us.

```
def plot_age(table, subject):
    """
    Draws a histogram of the Age column in the given table.

    table should be a Table with a column of people's ages called A
    subject should be a string -- the name of the group we're displaying
    like "married men".
    """

    # Draw a histogram of ages running from 15 years to 70 years
    table.hist('Age', bins=np.arange(15, 71, 5), unit='year')
    # Set the lower and upper bounds of the vertical axis so that
    # the plots we make are all comparable.
    plt.ylim(0, 0.045)
    plt.title("Ages of " + subject)
```

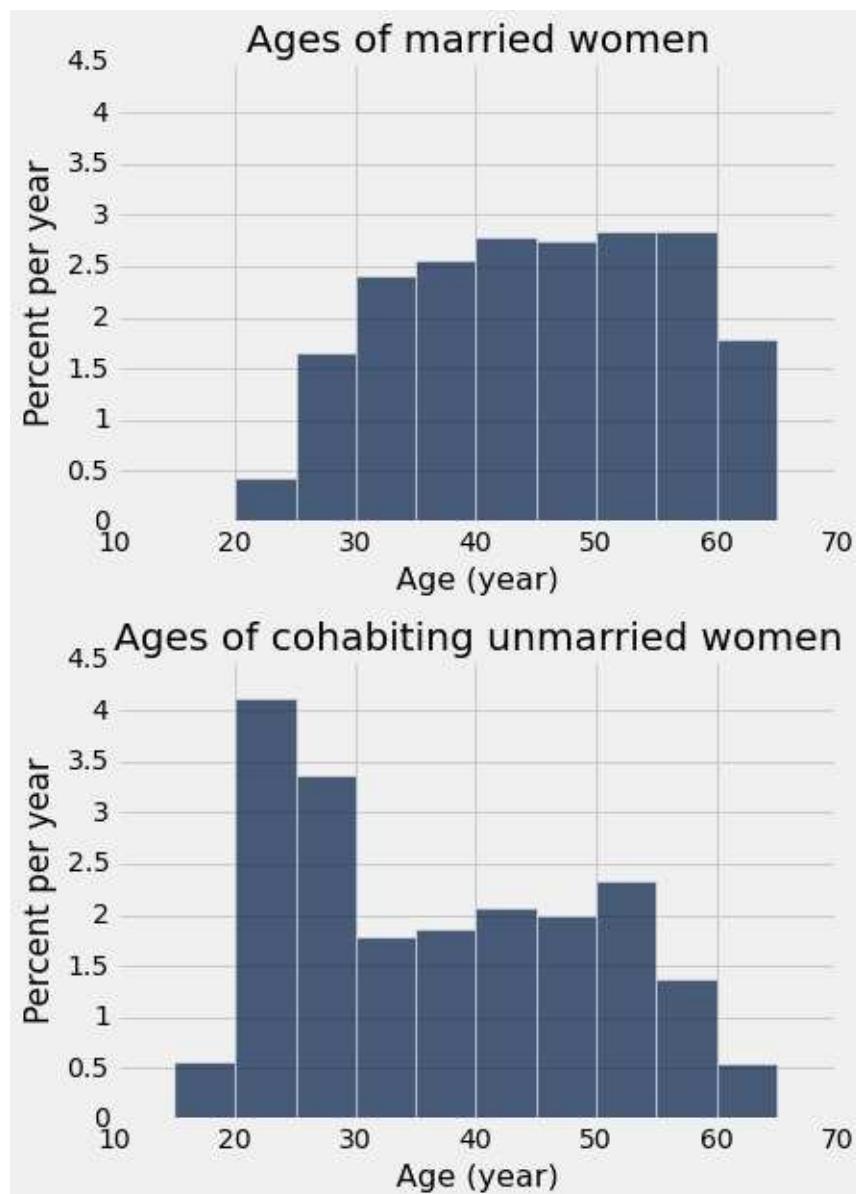
```
# Ages of men:  
plot_age(married_men, "married men")  
plot_age(partnered_men, "cohabiting unmarried men")
```



The difference is even more marked when we compare the married and unmarried women.

```
married_women = couples.where('Gender', 'female').where('Marital Status', 'Married')  
partnered_women = couples.where('Gender', 'female').where('Marital Status', 'Cohabiting or Unmarried')

# Ages of women:  
plot_age(married_women, "married women")  
plot_age(partnered_women, "cohabiting unmarried women")
```



The histograms show that the married men in the sample are in general older than unmarried cohabiting men. Married women are in general older than unmarried women. These observations are consistent with what we had predicted based on changing social norms.

If married couples are in general older, they might differ from unmarried couples in other ways as well. Let us compare the employment status of the married and unmarried men in the sample.

The table below shows the marital status and employment status of each man in the sample.

```
males = couples.where('Gender', 'male').select(['Marital Status', 'Employment Status'])
males.show(5)
```

Marital Status	Employment Status
married	working as paid employee
married	working as paid employee
married	working as paid employee
married	working, self-employed
married	not working - other

... (1028 rows omitted)

## Contingency Tables

To investigate the association between employment and marriage, we would like to be able to ask questions like, "How many married men are retired?"

Recall that the method `pivot` lets us do exactly that. It *cross-classifies* each man according to the two variables – marital status and employment status. Its output is a *contingency table* that contains the counts in each pair of categories.

```
employment = males.pivot('Marital Status', 'Employment Status')
employment
```

Employment Status	married	partner
not working - disabled	44	20
not working - looking for work	28	33
not working - on a temporary layoff from a job	15	8
not working - other	16	9
not working - retired	44	4
working as paid employee	513	170
working, self-employed	82	47

The arguments of `pivot` are the labels of the two columns corresponding to the variables we are studying. Categories of the first argument appear as columns; categories of the second argument are the rows. Each cell of the table contains the number of men in a pair of categories – a particular employment status and a particular marital status.

The table shows that regardless of marital status, the men in the sample are most likely to be working as paid employees. But it is quite hard to compare the entire distributions based on this table, because the numbers of married and unmarried men in the sample are not the

same. There are 742 married men but only 291 unmarried ones.

```
employment.drop(0).sum()
```

<b>married</b>	<b>partner</b>
742	291

To adjust for this difference in total numbers, we will convert the counts into proportions, by dividing all the `married` counts by 742 and all the `partner` counts by 291.

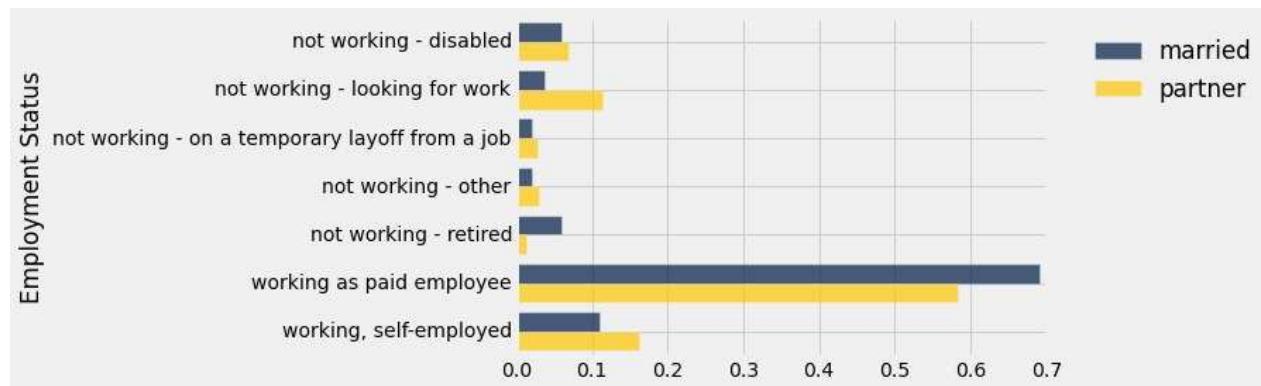
```
proportions = Table().with_columns([
    "Employment Status", cc.column("Employment Status"),
    "married", employment.column('married')/sum(employment.colu
    "partner", employment.column('partner')/sum(employment.colu
])
proportions.show()
```

<b>Employment Status</b>	<b>married</b>	<b>partner</b>
not working - disabled	0.0592992	0.0687285
not working - looking for work	0.0377358	0.113402
not working - on a temporary layoff from a job	0.0202156	0.0274914
not working - other	0.0215633	0.0309278
not working - retired	0.0592992	0.0137457
working as paid employee	0.691375	0.584192
working, self-employed	0.110512	0.161512

The `married` column of this table shows the distribution of employment status of the married men in the sample. For example, among married men, the proportion who are retired is about 0.059. The `partner` column shows the distribution of the employment status of the unmarried men in the sample. Among unmarried men, the proportion who are retired is about 0.014.

The two distributions look different from each other in other ways too, as can be seen more clearly in the bar graphs below. It appears that a larger proportion of the married men in the sample work as paid employees, whereas a larger proportion of the unmarried men are not working but are looking for work.

```
proportions.bah('Employment Status')
```



The distributions of employment status of the men in the two groups – married and unmarried – is clearly different in the sample.

## Are the two distributions different in the population?

This raises the question of whether the difference is due to randomness in the sampling, or whether the distributions of employment status are indeed different for married and unmarried cohabiting men in the U.S. Remember that the data that we have are from a sample of just 1,033 couples; we do not know the distribution of employment status of married or unmarried cohabiting men in the entire country.

We can answer the question by performing a statistical test of hypotheses. Let us use the terminology that we developed for this in the previous section.

**Null hypothesis.** In the United States, the distribution of employment status among married men is the same as among unmarried men who live with their partners.

Another way of saying this is that employment status and marital status are *independent* or *not associated*.

If the null hypothesis were true, then the difference that we have observed in the sample would be just due to chance.

**Alternative hypothesis.** In the United States, the distributions of the employment status of the two groups of men are different. In other words, employment status and marital status are associated in some way.

As our **test statistic**, we will use the total variation distance between two distributions.

The observed value of the test statistic is about 0.15:

```
# TVD between the two distributions in the sample
married = proportions.column('married')
partner = proportions.column('partner')
observed_tvd = 0.5*sum(abs(married - partner))
observed_tvd
```

```
0.15273571011754242
```

## Random Permutations

In order to compare this observed value of the total variation distance with what is predicted by the null hypothesis, we need to know how the total variation distance would vary across all possible random samples if employment status and marital status were not related.

This is quite daunting to derive by mathematics, but let us see if we can get a good approximation by simulation.

With just one sample at hand, and no further knowledge of the distribution of employment status among men in the United States, how can we go about replicating the sampling procedure? The key is to note that *if* marital status and employment status were not connected in any way, then we could replicate the sampling process by replacing each man's employment status by a randomly picked employment status from among all the men, married and unmarried.

Doing this for all the men is equivalent to randomly rearranging the entire column containing employment status, while leaving the marital status column unchanged. Such a rearrangement is called a *random permutation*.

Thus, under the null hypothesis, we can replicate the sampling process by assigning to each man an employment status chosen at random without replacement from the entries in the column `Employment Status`. We can do the replication by simply permuting the entire `Employment Status` column and leaving everything else unchanged.

Let's implement this plan. First, we will shuffle the column `empl_status` using the `sample` method, which just shuffles all the rows when provided with no arguments.

```
# Randomly permute the employment status of all men

shuffled = males.select('Employment Status').sample()
shuffled
```

<b>Employment Status</b>
working, self-employed
working, self-employed
working as paid employee
working as paid employee
working, self-employed
working as paid employee
not working - disabled
working as paid employee
working as paid employee
working as paid employee
... (1023 rows omitted)

The first two columns of the table below are taken from the original sample. The third has been created by randomly permuting the original Employment Status column.

```
# Construct a table in which employment status has been shuffled

males_with_shuffled_empl = Table().with_columns([
    "Marital Status", males.column('Marital Status'),
    "Employment Status", males.column('Employment Status'),
    "Employment Status (shuffled)", shuffled.column('Employment Status')
])
males_with_shuffled_empl
```

Marital Status	Employment Status	Employment Status (shuffled)
married	working as paid employee	working, self-employed
married	working as paid employee	working, self-employed
married	working as paid employee	working as paid employee
married	working, self-employed	working as paid employee
married	not working - other	working, self-employed
married	not working - on a temporary layoff from a job	working as paid employee
married	not working - disabled	not working - disabled
married	working as paid employee	working as paid employee
married	working as paid employee	working as paid employee
married	not working - retired	working as paid employee

... (1023 rows omitted)

Once again, the `pivot` method computes the contingency table, which allows us to calculate the total variation distance between the distributions of the two groups of men after their employment status has been shuffled.

```
employment_shuffled = males_with_shuffled_empl.pivot('Marital Status')
employment_shuffled
```

Employment Status (shuffled)	married	partner
not working - disabled	48	16
not working - looking for work	44	17
not working - on a temporary layoff from a job	16	7
not working - other	18	7
not working - retired	39	9
working as paid employee	489	194
working, self-employed	88	41

```
# TVD between the two distributions in the contingency table above
e_s = employment_shuffled
married = e_s.column('married')/sum(e_s.column('married')))
partner = e_s.column('partner')/sum(e_s.column('partner')))
0.5*sum(abs(married - partner))
```

```
0.032423745611841297
```

This total variation distance was computed based on the null hypothesis that the distributions of employment status for the two groups of men are the same. You can see that it is noticeably smaller than the observed value of the total variation distance (0.15) between the two groups in our original sample.

## A Permutation Test

Could this just be due to chance variation? We will only know if we run many more replications, by randomly permuting the `Employment status` column repeatedly. This method of testing is known as a **permutation test**.

```
# Put it all together in a for loop to perform a permutation test

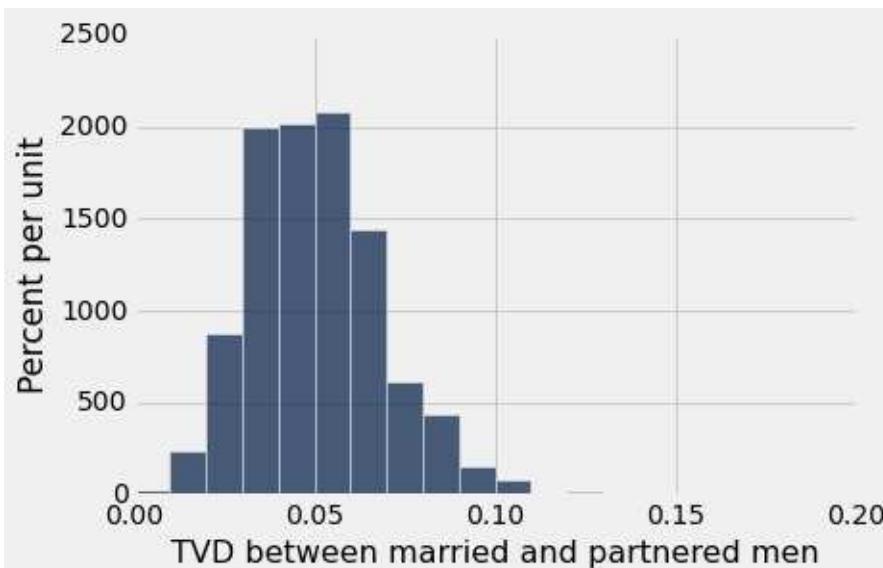
repetitions = 500

tvds = Table().with_column("TVD between married and partnered men",

for i in np.arange(repetitions):
    # Construct a permuted table
    shuffled = males.select('Employment Status').sample()
    combined = Table().with_columns([
        "Marital Status", males.column('Marital Status'),
        "Employment Status", shuffled.column('Employment Status')
    ])
    employment_shuffled = combined.pivot('Marital Status', 'Employm

    # Compute TVD
    e_s = employment_shuffled
    married = e_s.column('married')/sum(e_s.column('married'))
    partner = e_s.column('partner')/sum(e_s.column('partner'))
    permutation_tvd = 0.5*sum(abs(married - partner))
    tvds.append([permutation_tvd])

tvds.hist(bins=np.arange(0, 0.2, 0.01))
```



The figure above is the **empirical distribution of the total variation distance** between the distributions of the employment status of married and unmarried men, under the null hypothesis.

The observed test statistic of 0.15 is quite far in the tail, and so the chance of observing such an extreme value under the null hypothesis is close to 0.

As before, this chance is called an empirical P-value. The P-value is the chance that our test statistic (TVD) would come out at least as extreme as the observed value (in this case 0.15 or greater) under the null hypothesis.

## Conclusion of the test

Our empirical estimate based on repeated sampling gives us all the information we need for drawing conclusions from the data: the observed statistic is very unlikely under the null hypothesis.

The low P-value constitutes **evidence in favor of the alternative hypothesis**. The data support the hypothesis that in the United States, the distribution of the employment status of married men is not the same as that of unmarried men who live with their partners.

We have just completed our first *permutation test*. Permutation tests are quite common in practice because they make very few assumptions about the underlying population and are straightforward to perform and interpret.

### Note about the approximate P-value

Our simulation gives us an approximate empirical P-value, because it is based on just 500 random permutations instead of all the possible random permutations. We can compute this empirical P-value directly, without drawing the histogram:

```
empirical_p_value = np.count_nonzero(tvds.column(0) >= observed_tvc  
empirical_p_value
```

```
0 . 0
```

Computing the exact P-value would require us to consider all possible outcomes of shuffling (which is very large) instead of 500 random shuffles. If we had performed all the random shuffles, there would have been a few with more extreme TVDs. The true P-value is greater than zero, but not by much.

## Generalizing Our Hypothesis Test

The example above includes a substantial amount of code in order to investigate the relationship between two characteristics (marital status and employment status) for a particular subset of the surveyed population (males). Suppose we would like to investigate different characteristics or a different population. How can we reuse the code we have written so far in order to explore more relationships?

When you are about to copy your code, you should think, "Maybe I should write some functions."

What functions to write? A good way to make this decision is to think about what you have to compute repeatedly.

In our example, the total variation distance is computed over and over again. So we will begin with a generalized computation of total variation distance between the distribution of any column of values (such as employment status) when separated into any two conditions (such as marital status) for a collection of data described by any table. Our implementation includes the same statements as we used above, but uses generic names that are specified by the final function call.

```
# TVD between the distributions of values under any two conditions

def tvd(t, conditions, values):
    """Compute the total variation distance
    between proportions of values under two conditions.

    t          (Table) -- a table
    conditions (str)   -- a column label in t; should have only two
    values      (str)   -- a column label in t
    """
    e = t.pivot(conditions, values)
    a = e.column(1)
    b = e.column(2)
    return 0.5*sum(abs(a/sum(a) - b/sum(b)))

tvd(males, 'Marital Status', 'Employment Status')
```

0.15273571011754242

Next, we can write a function that performs a permutation test using this `tvd` function to compute the same statistic on shuffled variants of any table. It's worth reading through this implementation to understand its details.

```
def permutation_tvd(original, conditions, values):
    """
    Perform a permutation test of whether
    the distribution of values for two conditions
    is the same in the population,
    using the total variation distance between two distributions
    as the test statistic.

    original is a Table with two columns. The value of the argument
    conditions is the name of one column, and the value of the argument
    values is the name of the other column. The conditions table should
    have only 2 possible values corresponding to 2 categories in the
    data.

    The values column is shuffled many times, and the data are grouped
    according to the conditions column. The total variation distance
    between the proportions values in the 2 categories is computed.

    Then we draw a histogram of all those TV distances. This shows
    what the TVD between the values of the two distributions would
    look like if the values were independent of the conditions.

    """
    repetitions = 500
    stats = []

    for i in np.arange(repetitions):
        shuffled = original.sample()
        combined = Table().with_columns([
            conditions, original.column(conditions),
            values, shuffled.column(values)
        ])
        stats.append(tvd(combined, conditions, values))

    observation = tvd(original, conditions, values)
    p_value = np.count_nonzero(stats >= observation) / repetitions
```

```

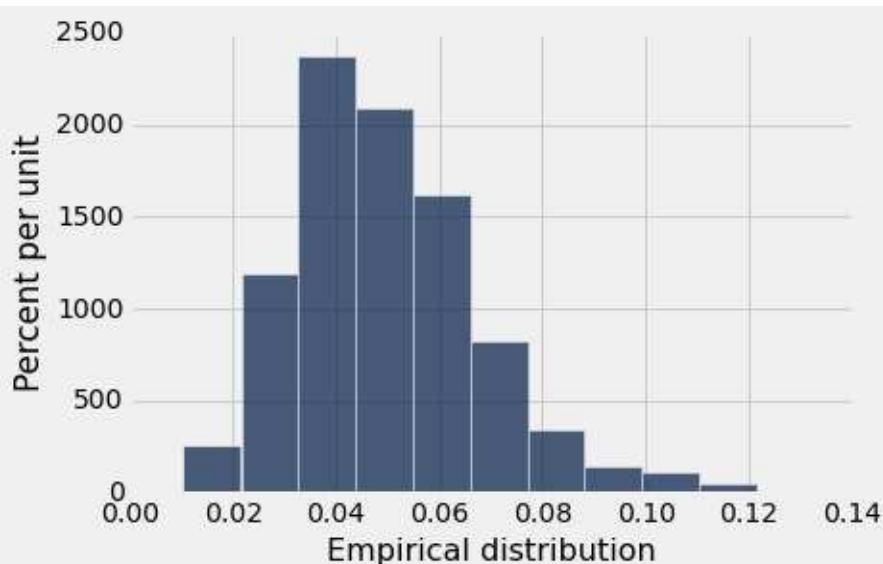
print("Observation:", observation)
print("Empirical P-value:", p_value)
Table([stats], ['Empirical distribution']).hist()

```

```
permutation_tvd(males, 'Marital Status', 'Employment Status')
```

Observation: 0.152735710118

Empirical P-value: 0.0



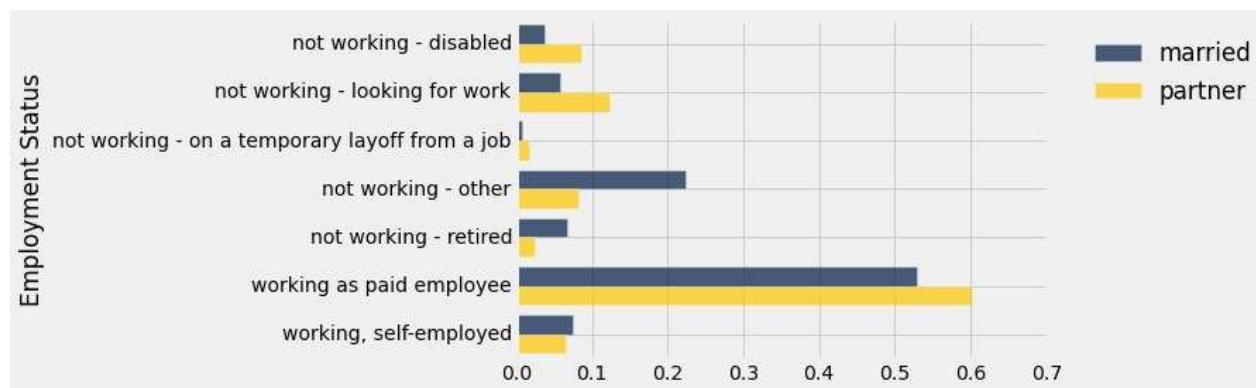
Now that we have generalized our permutation test, we can apply it to other hypotheses. For example, we can compare the distribution over the employment status of women, grouping them by their marital status. In the case of men we found a difference, but what about with women? First, we can visualize the two distributions.

```

def compare_bar(t, conditions, values):
    """Bargraphs of distributions of values for each of two conditions
    e = t.pivot(conditions, values)
    for label in e.drop(0).labels:
        # Convert each column of counts into proportions
        e.append_column(label, e.column(label)/sum(e.column(label)))
    e.bahr(values)

    compare_bar(couples.where('Gender', 'female'), 'Marital Status', 'E

```



A glance at the figure shows that the two distributions are different in the sample. The difference in the category "Not working – other" is particularly striking: about 22% of the married women are in this category, compared to only about 8% of the unmarried women. There are several reasons for this difference. For example, the percent of homemakers is greater among married women than among unmarried women, possibly because married women are more likely to be "stay-at-home" mothers of young children. The difference could also be generational: as we saw earlier, the married couples are older than the unmarried partners, and older women are less likely to be in the workforce than younger women.

While we can see that the distributions are different in the sample, we are not really interested in the sample for its own sake. We are examining the sample because it is likely to reflect the population. So, as before, we will use the sample to try to answer a question about something unknown: the distributions of employment status of married and unmarried cohabiting women *in the United States*. That is the population from which the sample was drawn.

We have to consider the possibility that the observed difference in the sample could simply be the result of chance variation. Remember that our data are only from a random sample of couples. We do not have data for all the couples in the United States.

**Null hypothesis:** In the U.S., the distribution of employment status is the same for married women as for unmarried women living with their partners. The difference in the sample is due to chance.

**Alternative hypothesis:** In the U.S., the distributions of employment status among married and unmarried cohabiting women are different.

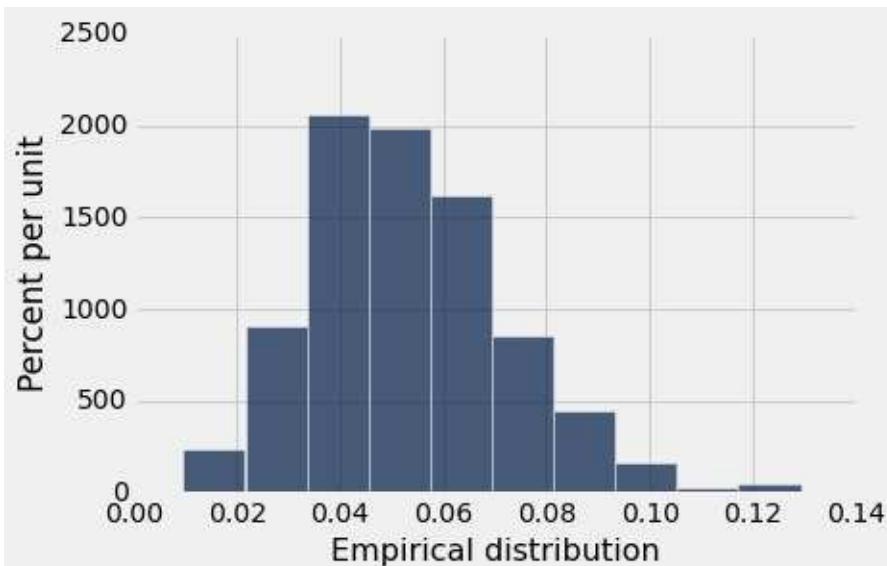
## Another permutation test to compare distributions

We can test these hypotheses just as we did for men, by using the function `permutation_tvd` that we defined for this purpose.

```
permutation_tvd(couples.where('Gender', 'female'), 'Marital Status')
```

```
Observation: 0.194755513565
```

```
Empirical P-value: 0.0
```



As for the males, the empirical P-value is 0 based on a large number of repetitions. So the exact P-value is close to 0, which is evidence in favor of the alternative hypothesis. The data support the hypothesis that for women in the United States, employment status is associated with whether they are married or unmarried and living with their partners.

## Another example

Are gender and employment status independent in the population? We are now in a position to test this quite swiftly:

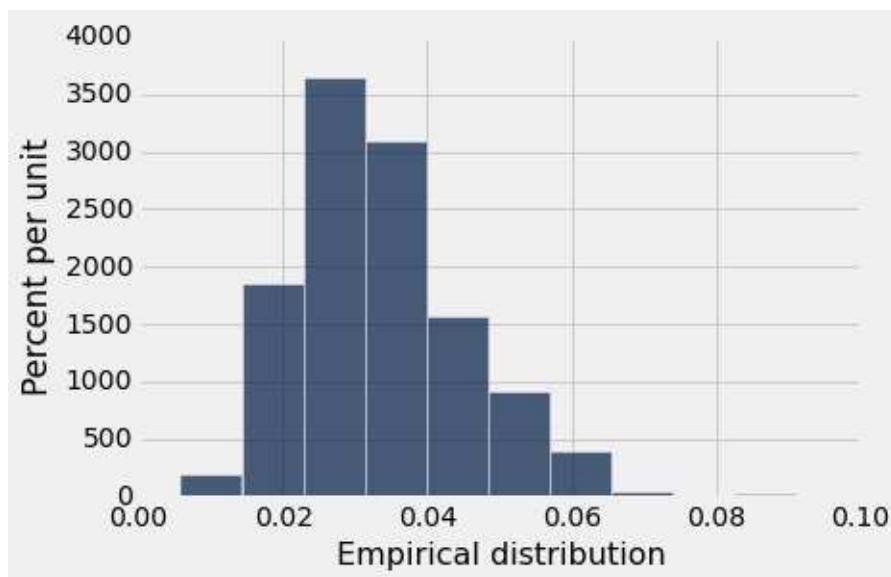
**Null hypothesis.** Among married and unmarried cohabiting individuals in the United States, gender is independent of employment status.

**Alternative hypothesis.** Among married and unmarried cohabiting people in the United States, gender and employment status are related.

```
permutation_tvd(couples, 'Gender', 'Employment Status')
```

```
Observation: 0.185866408519
```

```
Empirical P-value: 0.0
```



The conclusion of the test is that gender and employment status are not independent in the population. This is no surprise; for example, because of societal norms, older women were less likely to have gone into the workforce than men.

## Deflategate: Permutation Tests and Quantitative Variables

On January 18, 2015, the Indianapolis Colts and the New England Patriots played the American Football Conference (AFC) championship game to determine which of those teams would play in the Super Bowl. After the game, there were allegations that the Patriots' footballs had not been inflated as much as the regulations required; they were softer. This could be an advantage, as softer balls might be easier to catch.

For several weeks, the world of American football was consumed by accusations, denials, theories, and suspicions: the press labeled the topic Deflategate, after the Watergate political scandal of the 1970's. The National Football League (NFL) commissioned an independent analysis. In this example, we will perform our own analysis of the data.

Pressure is often measured in pounds per square inch (psi). NFL rules stipulate that game balls must be inflated to have pressures in the range 12.5 psi and 13.5 psi. Each team plays with 12 balls. Teams have the responsibility of maintaining the pressure in their own footballs, but game officials inspect the balls. Before the start of the AFC game, all the Patriots' balls were at about 12.5 psi. Most of the Colts' balls were at about 13.0 psi. However, these pre-game data were not recorded.

During the second quarter, the Colts intercepted a Patriots ball. On the sidelines, they measured the pressure of the ball and determined that it was below the 12.5 psi threshold. Promptly, they informed officials.

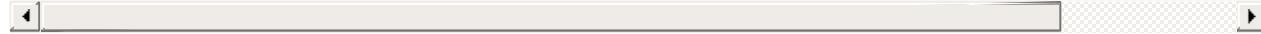
At half-time, all the game balls were collected for inspection. Two officials, Clete Blakeman and Dyrol Prioleau, measured the pressure in each of the balls. Here are the data; pressure is measured in psi. The Patriots ball that had been intercepted by the Colts was not inspected at half-time. Nor were most of the Colts' balls – the officials simply ran out of time and had to relinquish the balls for the start of second half play.

```
football = Table.read_table('football.csv')
football.show()
```

<b>Team</b>	<b>Ball</b>	<b>Blakeman</b>	<b>Prioleau</b>
0	Patriots 1	11.5	11.8
0	Patriots 2	10.85	11.2
0	Patriots 3	11.15	11.5
0	Patriots 4	10.7	11
0	Patriots 5	11.1	11.45
0	Patriots 6	11.6	11.95
0	Patriots 7	11.85	12.3
0	Patriots 8	11.1	11.55
0	Patriots 9	10.95	11.35
0	Patriots 10	10.5	10.9
0	Patriots 11	10.9	11.35
1	Colts 1	12.7	12.35
1	Colts 2	12.75	12.3
1	Colts 3	12.5	12.95
1	Colts 4	12.55	12.15

For each of the 15 balls that were inspected, the two officials got different results. It is not uncommon that repeated measurements on the same object yield different results, especially when the measurements are performed by different people. So we will assign to each the ball the average of the two measurements made on that ball.

```
football = football.with_column(
    'Combined', (football.column('Blakeman')+football.column('Prioleau'))
)
football.show()
```



<b>Team</b>	<b>Ball</b>	<b>Blakeman</b>	<b>Prioleau</b>	<b>Combined</b>
0	Patriots 1	11.5	11.8	11.65
0	Patriots 2	10.85	11.2	11.025
0	Patriots 3	11.15	11.5	11.325
0	Patriots 4	10.7	11	10.85
0	Patriots 5	11.1	11.45	11.275
0	Patriots 6	11.6	11.95	11.775
0	Patriots 7	11.85	12.3	12.075
0	Patriots 8	11.1	11.55	11.325
0	Patriots 9	10.95	11.35	11.15
0	Patriots 10	10.5	10.9	10.7
0	Patriots 11	10.9	11.35	11.125
1	Colts 1	12.7	12.35	12.525
1	Colts 2	12.75	12.3	12.525
1	Colts 3	12.5	12.95	12.725
1	Colts 4	12.55	12.15	12.35

At a glance, it seems apparent that the Patriots' footballs were at a lower pressure than the Colts' balls. Because some deflation is normal during the course of a game, the independent analysts decided to calculate the drop in pressure from the start of the game. Recall that the Patriots' balls had all started out at about 12.5 psi, and the Colts' balls at about 13.0 psi. Therefore the drop in pressure for the Patriots' balls was computed as 12.5 minus the pressure at half-time, and the drop in pressure for the Colts' balls was 13.0 minus the pressure at half-time.

```
football = football.with_column(
    'Drop', np.array([12.5]*11 + [13.0]*4) - football.column('Combined'))
football.show()
```



Team	Ball	Blakeman	Prieleau	Combined	Drop
0	Patriots 1	11.5	11.8	11.65	0.85
0	Patriots 2	10.85	11.2	11.025	1.475
0	Patriots 3	11.15	11.5	11.325	1.175
0	Patriots 4	10.7	11	10.85	1.65
0	Patriots 5	11.1	11.45	11.275	1.225
0	Patriots 6	11.6	11.95	11.775	0.725
0	Patriots 7	11.85	12.3	12.075	0.425
0	Patriots 8	11.1	11.55	11.325	1.175
0	Patriots 9	10.95	11.35	11.15	1.35
0	Patriots 10	10.5	10.9	10.7	1.8
0	Patriots 11	10.9	11.35	11.125	1.375
1	Colts 1	12.7	12.35	12.525	0.475
1	Colts 2	12.75	12.3	12.525	0.475
1	Colts 3	12.5	12.95	12.725	0.275
1	Colts 4	12.55	12.15	12.35	0.65

It is apparent that the drop was larger, on average, for the Patriots' footballs. But could the difference be just due to chance?

To answer this, we must first examine how chance might enter the analysis. This is not a situation in which there is a random sample of data from a large population. It is also not clear how to create a justifiable abstract chance model, as the balls were all different, inflated by different people, and maintained under different conditions.

One way to introduce chances is to ask whether the drops in pressures of the 11 Patriots balls and the 4 Colts balls resemble a random permutation of the 15 drops. Then the 4 Colts drops would be a simple random sample of all 15 drops. This gives us a null hypothesis that we can test using random permutations.

**Null hypothesis.** The drops in the pressures of the 4 Colts balls are like a random sample (without replacement) from all 15 drops.

## A new test statistic

The data are quantitative, so we cannot compare the two distributions category by category using the total variation distance. If we try to bin the data in order to use the TVD, the choice of bins can have a noticeable effect on the statistic. So instead, we will work with a simple statistic based on means. We will just compare the average drops in the two groups.

The observed difference between the average drops in the two groups was about 0.7335 psi.

```
patriots = football.where('Team', 0).column('Drop')
colts = football.where('Team', 1).column('Drop')
observed_difference = np.mean(patriots) - np.mean(colts)
observed_difference
```

```
0.73352272727272805
```

Now the question becomes: If we took a random permutation of the 15 drops, how likely is it that the difference in the means of the first 11 and the last 4 would be at least as large as the difference observed by the officials?

To answer this, we will randomly permute the 15 drops, assign the first 11 permuted values to the Patriots and the last 4 to the Colts. Then we will find the difference in the means of the two permuted groups.

```
drops_shuffled = football.sample().column('Drop')
patriots_shuffled = drops_shuffled[:10]
colts_shuffled = drops_shuffled[11:]
shuffled_difference = np.mean(patriots_shuffled) - np.mean(colts_shuffled)
shuffled_difference
```

```
0.01000000000000675
```

This is different from the observed value we calculated earlier. But to get a better sense of the variability under random sampling we must repeat the process many times. Let us try making 5000 repetitions and drawing a histogram of the 5000 differences between means.

```

repetitions = 5000

test_stats = []
for i in range(repetitions):
    drops_shuffled = football.sample().column('Drop')
    patriots_shuffled = drops_shuffled[:10]
    colts_shuffled = drops_shuffled[11:]
    shuffled_difference = np.mean(patriots_shuffled) - np.mean(colts_shuffled)
    test_stats.append(shuffled_difference)

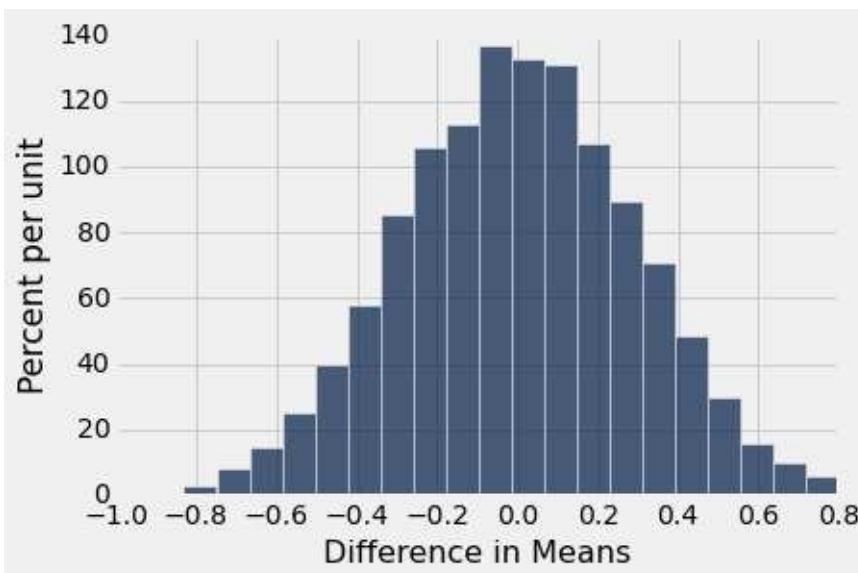
observation = np.mean(patriots) - np.mean(colts)
emp_p_value = np.count_nonzero(test_stats >= observation) / repetitions

differences = Table().with_column('Difference in Means', test_stats)
differences.hist(bins=20)
print("Observation:", observation)
print("Empirical P-value:", emp_p_value)

```

Observation: 0.733522727273

Empirical P-value: 0.0028



The observed difference was roughly 0.7335 psi. According to the empirical distribution above, there is a very small chance that a random permutation would yield a difference that large. So the data support the conclusion that the two groups of pressures were not like a random permutation of all 15 pressures.

The independent investigative team analyzed the data in several different ways, taking into account the laws of physics. The final report said,

"[T]he average pressure drop of the Patriots game balls exceeded the average pressure drop of the Colts balls by 0.45 to 1.02 psi, depending on various possible assumptions regarding the gauges used, and assuming an initial pressure of 12.5 psi for the Patriots balls and 13.0 for the Colts balls."

-- *Investigative report commissioned by the NFL regarding the AFC Championship game on January 18, 2015*

Our analysis shows an average pressure drop of about 0.73 psi, which is consistent with the official analysis.

The all-important question in the football world was whether the excess drop of pressure in the Patriots' footballs was deliberate. To that question, the data have no answer. If you are curious about the answer given by the investigators, here is the [full report](#).

# A/B Testing

[Interact](#)

## Using the Bootstrap Method to Test Hypotheses

We have used random permutations to see whether two samples are drawn from the same underlying distribution. The bootstrap method can also be used in such statistical tests of hypotheses.

The examples in this section are based on data on a random sample of 1,174 pairs of mothers and their newborn infants. The table `baby` contains the data. Each row represents a mother-baby pair. The variables are:

- the baby's birth weight in ounces
- the number of gestational days
- the mother's age in completed years
- the mother's height in inches
- the mother's pregnancy weight in pounds
- whether or not the mother was a smoker

```
baby = Table.read_table('baby.csv')  
baby
```

Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
120	284	27	62	100	False
113	282	33	64	135	False
128	279	28	64	115	True
108	282	23	67	125	True
136	286	25	62	93	False
138	244	33	62	178	False
132	245	23	65	140	False
120	289	25	62	125	False
143	299	30	66	136	True
140	351	27	68	120	False

... (1164 rows omitted)

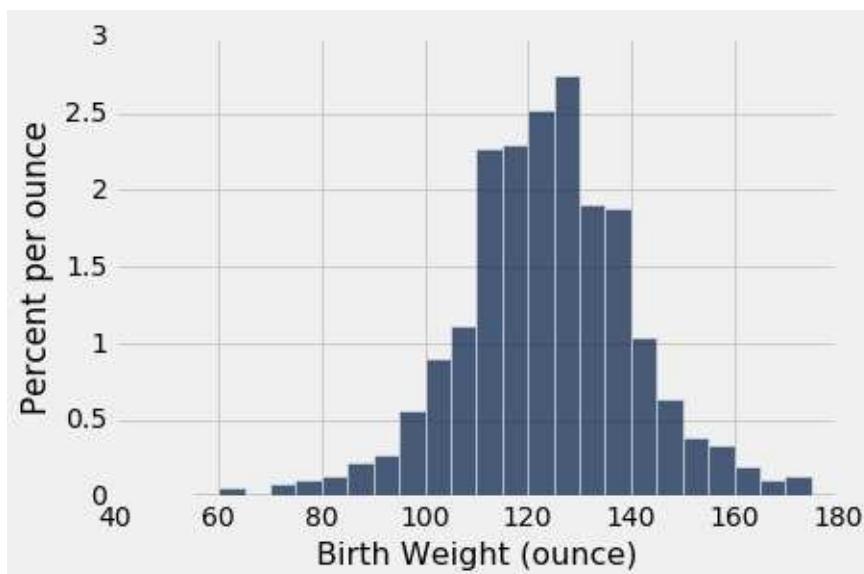
Suppose we want to compare the mothers who smoke and the mothers who are non-smokers. Do they differ in any way other than smoking? A key variable of interest is the birth weight of their babies. To study this variable, we begin by noting that there are 715 non-smokers among the women in the sample, and 459 smokers.

```
baby.group('Maternal Smoker')
```

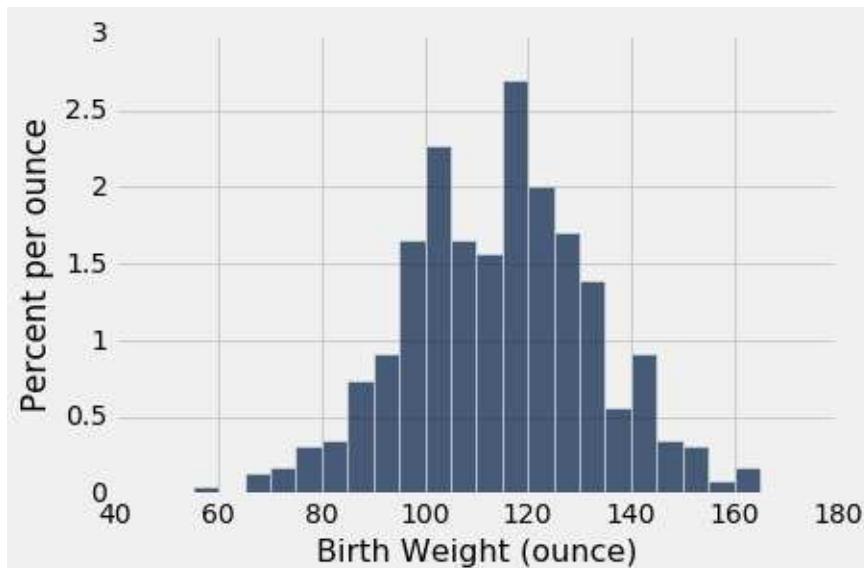
Maternal Smoker	count
False	715
True	459

The first histogram below displays the distribution of birth weights of the babies of the non-smokers in the sample. The second displays the birth weights of the babies of the smokers.

```
baby.where('Maternal Smoker', False).hist('Birth Weight', bins=np.arange(55, 455, 10))
```



```
baby.where('Maternal Smoker', True).hist('Birth Weight', bins=np.arange(40, 180, 10))
```



Both distributions are approximately bell shaped and centered near 120 ounces. The distributions are not identical, of course, which raises the question of whether the difference reflects just chance variation or a difference in the distributions in the population.

This question can be answered by a test of the null and alternative hypotheses below. Because we are testing for the equality of two distributions, one for Category A (mothers who don't smoke) and the other for Category B (mothers who smoke), the method is known rather unimaginatively as *A/B testing*.

**Null hypothesis.** In the population, the distribution of birth weights of babies is the same for mothers who don't smoke as for mothers who do. The difference in the sample is due to chance.

**Alternative hypothesis.** The two distributions are different in the population.

**Test statistic.** Birth weight is a quantitative variable, so it is reasonable to use the difference between means as the test statistic. There are other reasonable test statistics as well; the difference between means is just one simple choice.

The observed difference between the means of the two groups in the sample is about 9.27 ounces.

```
nonsmokers_mean = np.mean(baby.where('Maternal Smoker', False).column('nonsmokers'))  
smokers_mean = np.mean(baby.where('Maternal Smoker', True).column('smokers'))  
nonsmokers_mean - smokers_mean
```

```
9.2661425720249184
```

## Implementing the bootstrap method¶

To see whether such a difference could have arisen due to chance under the null hypothesis, we could use a permutation test just as we did in the previous section. An alternative is to use the bootstrap, which we will do here.

Under the null hypothesis, the distributions of birth weights are the same for babies of women who smoke and for babies of those who don't. To see how much the distributions could vary due to chance alone, we will use the bootstrap method and draw new samples at random *with* replacement from the entire set of birth weights. The only difference between this and the permutation test of the previous section is that random permutations are based on sampling *without* replacement.

We will perform 10,000 repetitions of the bootstrap process. There are 1,174 babies, so in each repetition of the bootstrap process we draw 1,174 times at random with replacement. Of these 1,174 draws, 715 are assigned to the non-smoking mothers and the remaining 459 to the mothers who smoke.

The code starts by sorting the rows so that the rows corresponding to the 715 non-smokers appear first. This eliminates the need to use conditional expressions to identify the rows corresponding to smokers and non-smokers in each replication of the sampling process.

```
"""Bootstrap test for the difference in mean birthweight
Category A: non-smoker      Category B: smoker"""

sorted_by_smoking = baby.select(['Maternal Smoker', 'Birth Weight'])

# calculate the observed difference in means
meanA = np.mean(sorted_by_smoking.column('Birth Weight')[:715])
meanB = np.mean(sorted_by_smoking.column('Birth Weight')[715:])
observed_difference = meanA - meanB

repetitions=10000
diffs = []

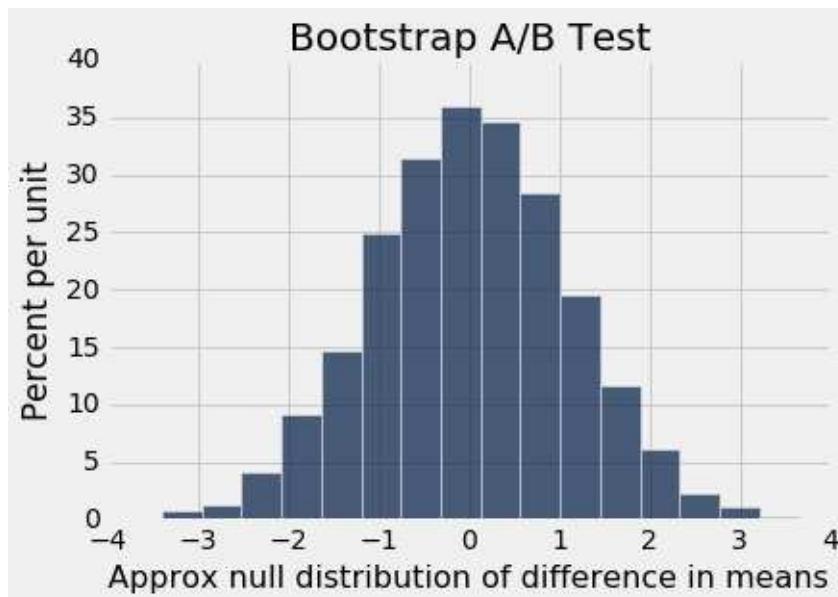
for i in range(repetitions):

    # sample WITH replacement, same number as original sample size
    resample = sorted_by_smoking.sample(with_replacement=True)

    # Compute the difference of the means of the resampled values,
    diff_means = np.mean(resample.column('Birth Weight')[:715]) - r
    diffs.append([diff_means])

# Display results
differences = Table().with_column('diff_in_means', diffs)
differences.hist(bins=20)
plots.xlabel('Approx null distribution of difference in means')
plots.title('Bootstrap A/B Test')
plots.xlim(-4, 4)
plots.ylim(0, 0.4)
print('Observed difference in means: ', observed_difference)
```

Observed difference in means: 9.26614257202



The figure shows that under the null hypothesis of equal distributions in the population, the bootstrap empirical distribution of the difference between the sample means of the two groups is roughly bell shaped, centered at 0, stretching from about **-4** ounces to **4** ounces. The observed difference in the original sample is about 9.27 ounces, which is inconsistent with this distribution. So the conclusion of the test is that in the population, the distributions of birth weights of the babies of non-smokers and smokers are different.

## Bootstrap A/B testing

Let us define a function `bootstrap_AB_test` that performs an A/B test using the bootstrap method and the difference in sample means as the test statistic. The null hypothesis is that the two underlying distributions in the population are equal; the alternative is that they are not.

The arguments of the function are:

- the name of the table that contains the data in the original sample
- the label of the column containing the code 0 for Category A and 1 for Category B
- the label of the column containing the values of the response variable (that is, the variable whose distribution is of interest)
- the number of repetitions of the resampling procedure

The function returns the observed difference in means, the bootstrap empirical distribution of the difference in means, and the bootstrap empirical P-value. Because the alternative simply says that the two underlying distributions are different, the P-value is computed as the proportion of sampled differences that are at least as large in absolute value as the absolute value of the observed difference.

```
"""Bootstrap A/B test for the difference in the mean response
Assumes A=0, B=1"""

def bootstrap_AB_test(table, categories, values, repetitions):

    # Sort the sample table according to the A/B column;
    # then select only the column of effects.
    response = table.sort(categories).select(values)

    # Find the number of entries in Category A.
    n_A = table.where(categories, 0).num_rows

    # Calculate the observed value of the test statistic.
    meanA = np.mean(response.column(values)[:n_A])
    meanB = np.mean(response.column(values)[n_A:])
    observed_difference = meanA - meanB

    # Run the bootstrap procedure and get a list of resampled differences
    diffs = []
    for i in range(repetitions):
        resample = response.sample(with_replacement=True)
        d = np.mean(resample.column(values)[:n_A]) - np.mean(resample.column(values)[n_A:])
        diffs.append([d])

    # Compute the bootstrap empirical P-value
    diff_array = np.array(diffs)
    empirical_p = np.count_nonzero(abs(diff_array) >= abs(observed_difference))

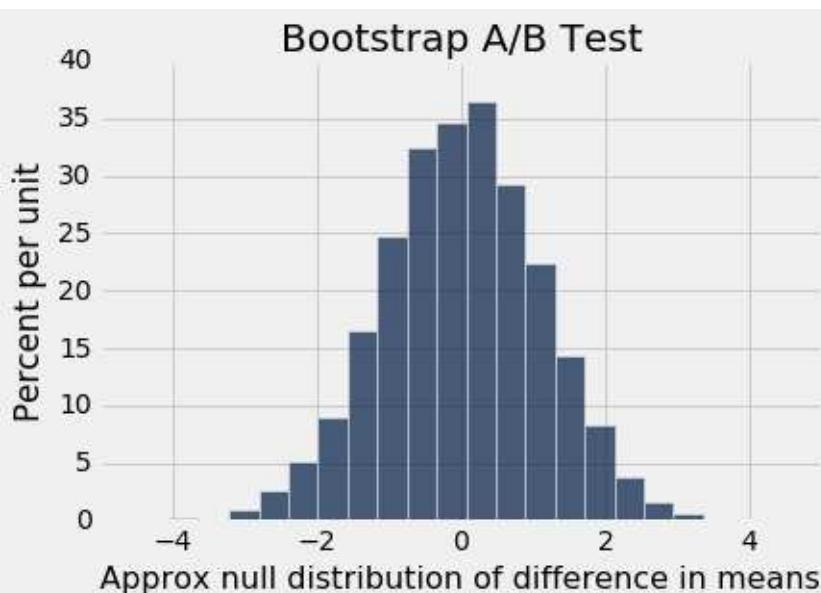
    # Display results
    differences = Table().with_column('diff_in_means', diffs)
    differences.hist(bins=20, normed=True)
    plots.xlabel('Approx null distribution of difference in means')
    plots.title('Bootstrap A/B Test')
    print("Observed difference in means: ", observed_difference)
    print("Bootstrap empirical P-value: ", empirical_p)
```

We can now use the function `bootstrap_AB_test` to compare the smokers and non-smokers with respect to several different response variables. The tests show a statistically significant difference between the two groups in birth weight (as shown earlier), gestational days,

maternal age, and maternal pregnancy weight. It comes as no surprise that the two groups do not differ significantly in their mean heights.

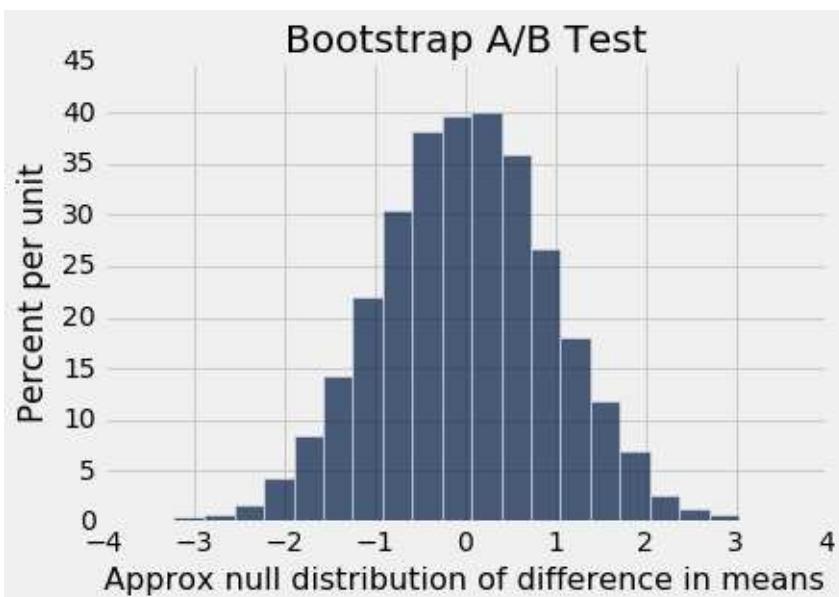
```
bootstrap_AB_test(baby, 'Maternal Smoker', 'Birth Weight', 10000)
```

```
Observed difference in means: 9.26614257202
Bootstrap empirical P-value: 0.0
```



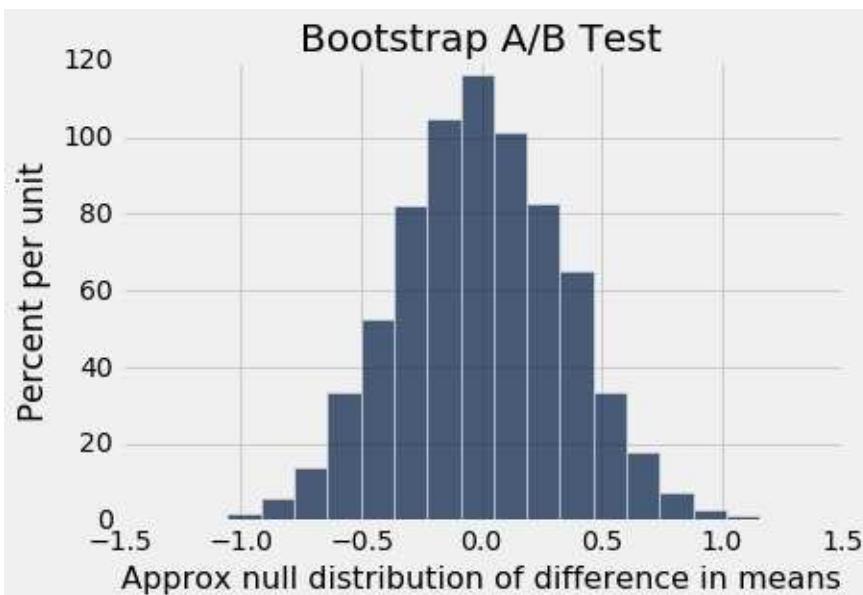
```
bootstrap_AB_test(baby, 'Maternal Smoker', 'Gestational Days', 1000)
```

```
Observed difference in means: 1.97652238829
Bootstrap empirical P-value: 0.0367
```



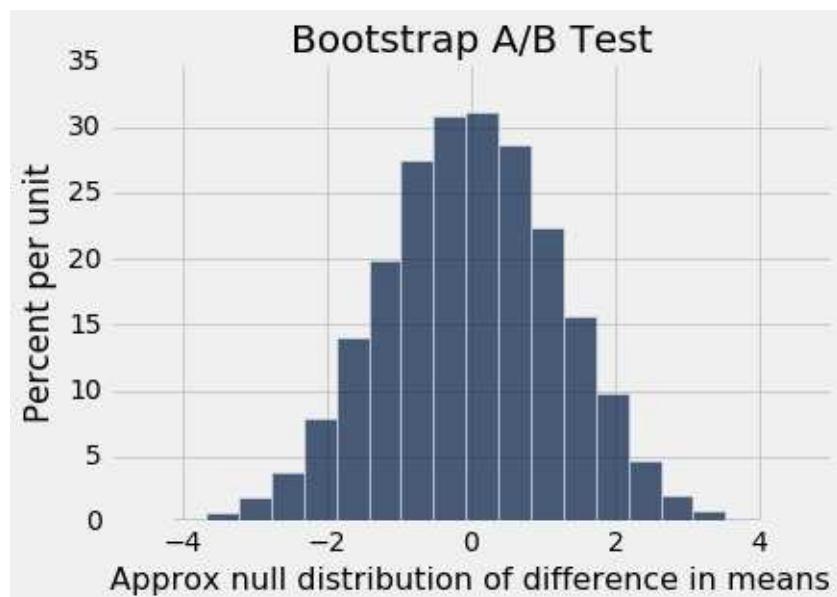
```
bootstrap_AB_test(baby, 'Maternal Smoker', 'Maternal Age', 10000)
```

Observed difference in means: 0.80767250179  
 Bootstrap empirical P-value: 0.0192



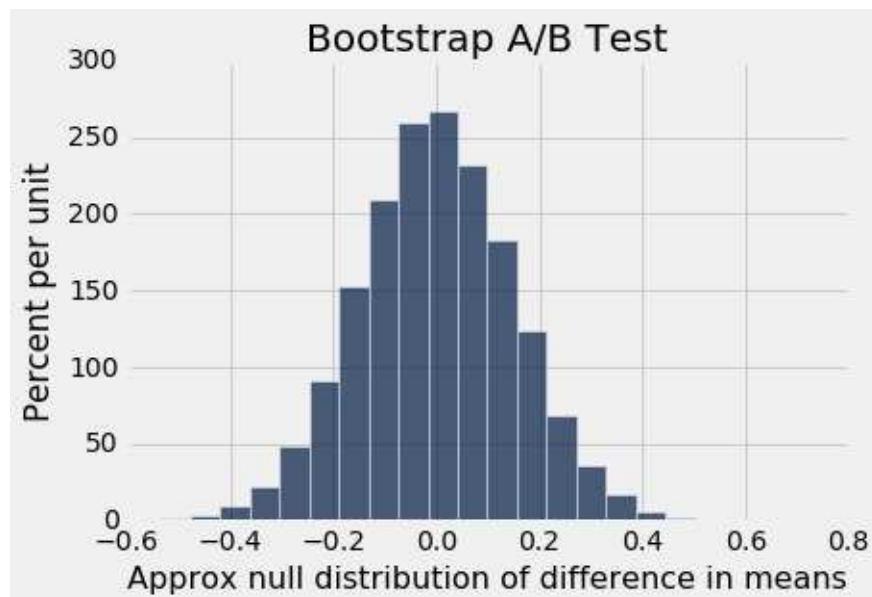
```
bootstrap_AB_test(baby, 'Maternal Smoker', 'Maternal Pregnancy Weight', 10000)
```

Observed difference in means: 2.56033030151  
 Bootstrap empirical P-value: 0.0374



```
bootstrap_AB_test(baby, 'Maternal Smoker', 'Maternal Height', 10000)
```

```
Observed difference in means: -0.0905891494127  
Bootstrap empirical P-value: 0.5411
```



# Regression Inference

[Interact](#)

## Regression: a Review

Earlier in the course, we developed the concepts of correlation and regression as ways to describe relations between quantitative variables. We will now see how these concepts can become powerful tools for inference, when used appropriately.

First, let us briefly review the main ideas of correlation and regression, along with code for calculations.

It is often convenient to measure variables in **standard units**. When you convert a value to standard units, you are measuring how many standard deviations above average the value is.

The function `standard_units` takes an array as its argument and returns the array converted to standard units.

```
def standard_units(x):
    return (x - np.mean(x))/np.std(x)
```

The **correlation** between two variables measures the degree to which a scatter plot of the two variables is clustered around a straight line. It has no units and is a number between -1 and 1. It is calculated as the average of the product of the two variables, when both variables are measured in standard units.

The function `correlation` takes three arguments – the name of a table and the labels of two columns of the table – and returns the correlation between the two columns.

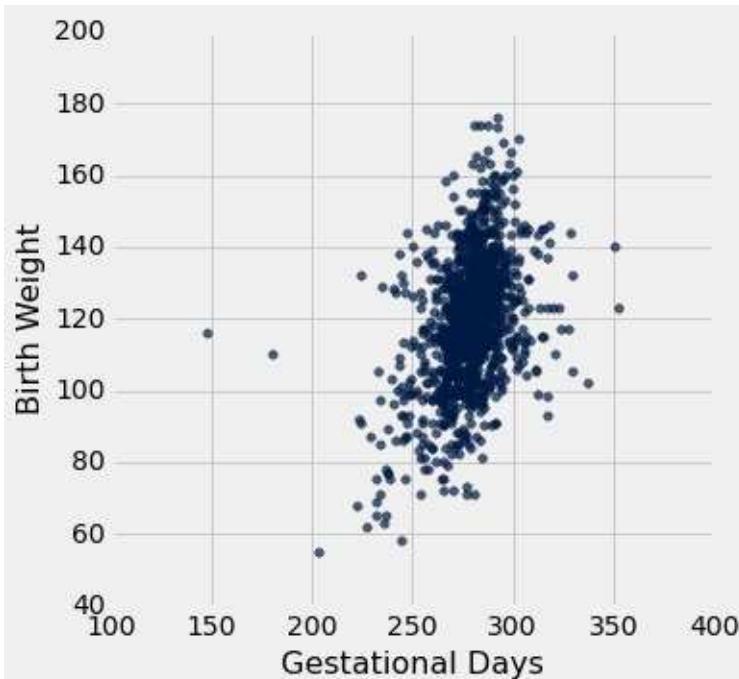
```
def correlation(table, x, y):
    x_in_standard_units = standard_units(table.column(x))
    y_in_standard_units = standard_units(table.column(y))
    return np.mean(x_in_standard_units * y_in_standard_units)
```

Here is the scatter plot of birth weight (the *y*-variable) versus gestational days (the *x*-variable) of the babies in the table `baby`. The plot shows a positive association, and `correlation` returns a value of just over 0.4.

```
correlation(baby, 'Gestational Days', 'Birth Weight')
```

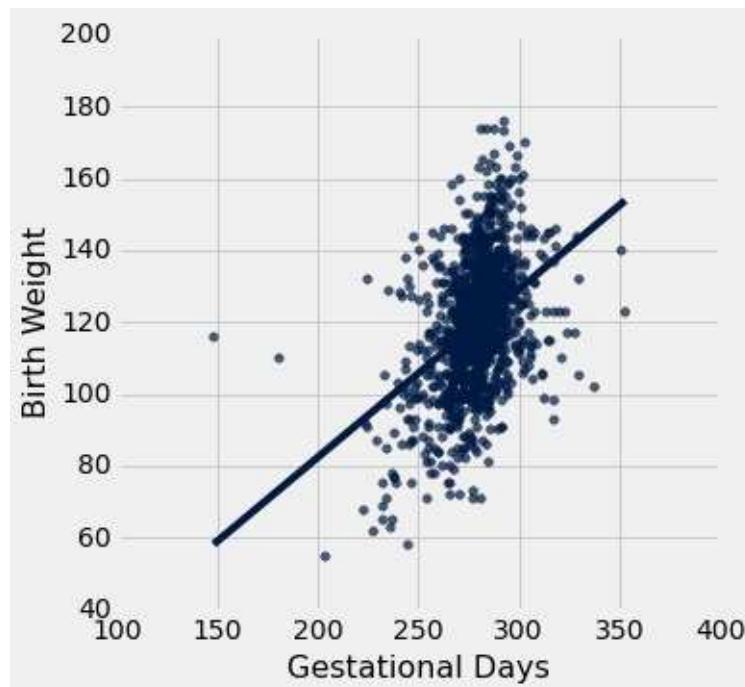
```
0.40754279338885108
```

```
baby.scatter('Gestational Days', 'Birth Weight')
```



When `scatter` is called with the option `fit_line=True`, the regression line is drawn on the scatter plot.

```
baby.scatter('Gestational Days', 'Birth Weight', fit_line=True)
```



The **regression line** is the best among all straight lines that could be used to estimate  $y$  based on  $x$ , in the sense that it minimizes the mean squared error of estimation.

The **slope of the regression line** is given by:

$$\text{slope} = r \times \frac{\text{SD of } y}{\text{SD of } x}$$

where  $r$  is the correlation.

The **intercept of the regression line** is given by

$$\text{intercept} = \text{average of } y - \text{slope} \times \text{average of } x$$

The functions `slope` and `intercept` calculate these two quantities. Each takes the same three arguments as `correlation`: the name of the table and the labels of columns  $x$  and  $y$ , in that order.

```
def slope(table, x, y):
    r = correlation(table, x, y)
    return r * np.std(table.column(y))/np.std(table.column(x))
```

```
def intercept(table, x, y):
    a = slope(table, x, y)
    return np.mean(table.column(y)) - a * np.mean(table.column(x))
```

We can now calculate the slope and the intercept of the regression line drawn above. The slope is about 0.47 ounces per gestational day.

```
slope(baby, 'Gestational Days', 'Birth Weight')
```

```
0.46655687694921522
```

The intercept is about -10.75 ounces.

```
intercept(baby, 'Gestational Days', 'Birth Weight')
```

```
-10.754138914450252
```

Thus the **equation of the regression line** for estimating birth weight based on gestational days is:

$$\text{estimate of birth weight} = 0.47 \times \text{gestational days} \\ - 10.75$$

The **fitted value** at a given value of  $x$  is the estimate of  $y$  based on that value of  $x$ . In other words, the fitted value at a given value of  $x$  is the height of the regression line at that  $x$ .

The function `fitted_value` computes this height. Like the functions `correlation`, `slope`, and `intercept`, its arguments include the name of the table and the labels of the  $x$  and  $y$  columns. But it also requires a fourth argument, which is the value of  $x$  at which the estimate will be made.

```
def fitted_value(table, x, y, given_x):
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * given_x + b
```

The fitted value at 300 gestational days is about 129.2 ounces. For a pregnancy that has length 300 gestational days, our estimate for the baby's weight is about 129.2 ounces.

```
fitted_value(baby, 'Gestational Days', 'Birth Weight', 300)
```

```
129.2129241703143
```

The function `fit` returns an array consisting of the fitted values at all of the values of  $x$  in the table.

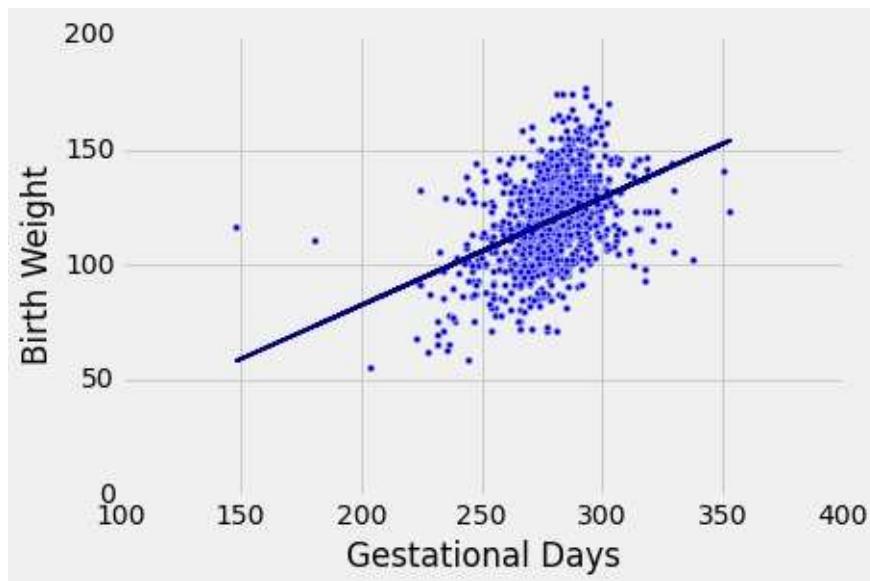
```
def fit(table, x, y):
    a = slope(table, x, y)
    b = intercept(table, x, y)
    return a * table.column(x) + b
```

As we know, the Table method `scatter` can be used to draw a scatter plot with a regression line through it. Here is another way, using `fit`. The function `scatter_fit` takes the same three arguments as `fit` and returns a scatter diagram along with the straight line of fitted values.

```
def scatter_fit(table, x, y):
    plots.scatter(table.column(x), table.column(y), s=15)
    plots.plot(table.column(x), fit(table, x, y), lw=2, color='darkred')
    plots.xlabel(x)
    plots.ylabel(y)

scatter_fit(baby, 'Gestational Days', 'Birth Weight')

# The next two lines just make the plot easier to read
plots.ylim([0, 200])
None
```



## Assumptions of randomness: a "regression model"

Thus far in this section, our analysis has been purely descriptive. But recall that the data are a random sample of all the births in a system of hospitals. What do the data say about the whole population of births? What could they say about a new birth at one of the hospitals?

Questions of inference may arise if we believe that a scatter plot reflects the underlying relation between the two variables being plotted but does not specify the relation completely. For example, a scatter plot of birth weight versus gestational days shows us the precise relation between the two variables in our sample; but we might wonder whether that relation holds true, or almost true, for all babies in the population from which the sample was drawn, or indeed among babies in general.

As always, inferential thinking begins with a careful examination of the assumptions about the data. Sets of assumptions are known as *models*. Sets of assumptions about randomness in roughly linear scatter plots are called *regression models*.

In brief, such models say that the underlying relation between the two variables is perfectly linear; this straight line is the *signal* that we would like to identify. However, we are not able to see the line clearly. What we see are points that are scattered around the line. In each of the points, the signal has been contaminated by *random noise*. Our inferential goal, therefore, is to separate the signal from the noise.

In greater detail, the regression model specifies that the points in the scatter plot are generated at random as follows.

- The relation between  $x$  and  $y$  is perfectly linear. We cannot see this "true line" but

Tyche can. She is the Goddess of Chance.

- Tyche creates the scatter plot by taking points on the line and pushing them off the line vertically, either above or below, as follows:
  - For each  $x$ , Tyche finds the corresponding point on the true line, and then adds an error.
  - The errors are drawn at random with replacement from a population of errors that has a normal distribution with mean 0.
  - Tyche creates a point whose horizontal coordinate is  $x$  and whose vertical coordinate is "the height of the true line at  $x$ , plus the error".
- Finally, Tyche erases the true line from the scatter, and shows us just the scatter plot of her points.

Based on this scatter plot, how should we estimate the true line? The best line that we can put through a scatter plot is the regression line. So the regression line is a natural estimate of the true line.

The simulation below shows how close the regression line is to the true line. The first panel shows how Tyche generates the scatter plot from the true line; the second show the scatter plot that we see; the third shows the regression line through the plot; and the fourth shows both the regression line and the true line.

To run the simulation, call the function `draw_and_compare` with three arguments: the slope of the true line, the intercept of the true line, and the sample size.

Run the simulation a few times, with different values for the slope and intercept of the true line, and varying sample sizes. Because all the points are generated according to the model, you will see that the regression line is a good estimate of the true line if the sample size is moderately large.

```

def draw_and_compare(true_slope, true_int, sample_size):
    x = np.random.normal(50, 5, sample_size)
    xlims = np.array([np.min(x), np.max(x)])
    eps = np.random.normal(0, 6, sample_size)
    y = (true_slope*x + true_int) + eps
    tyche = Table([x,y],['x','y'])

    plots.figure(figsize=(6, 16))
    plots.subplot(4, 1, 1)
    plots.scatter(tyche['x'], tyche['y'], s=15)
    plots.plot(xlims, true_slope*xlims + true_int, lw=2, color='green')
    plots.title('What Tyche draws')

    plots.subplot(4, 1, 2)
    plots.scatter(tyche['x'],tyche['y'], s=15)
    plots.title('What we get to see')

    plots.subplot(4, 1, 3)
    scatter_fit(tyche, 'x', 'y')
    plots.xlabel("")
    plots.ylabel("")
    plots.title('Regression line: our estimate of true line')

    plots.subplot(4, 1, 4)
    scatter_fit(tyche, 'x', 'y')
    xlims = np.array([np.min(tyche['x']), np.max(tyche['x'])])
    plots.plot(xlims, true_slope*xlims + true_int, lw=2, color='green')
    plots.title("Regression line and true line")

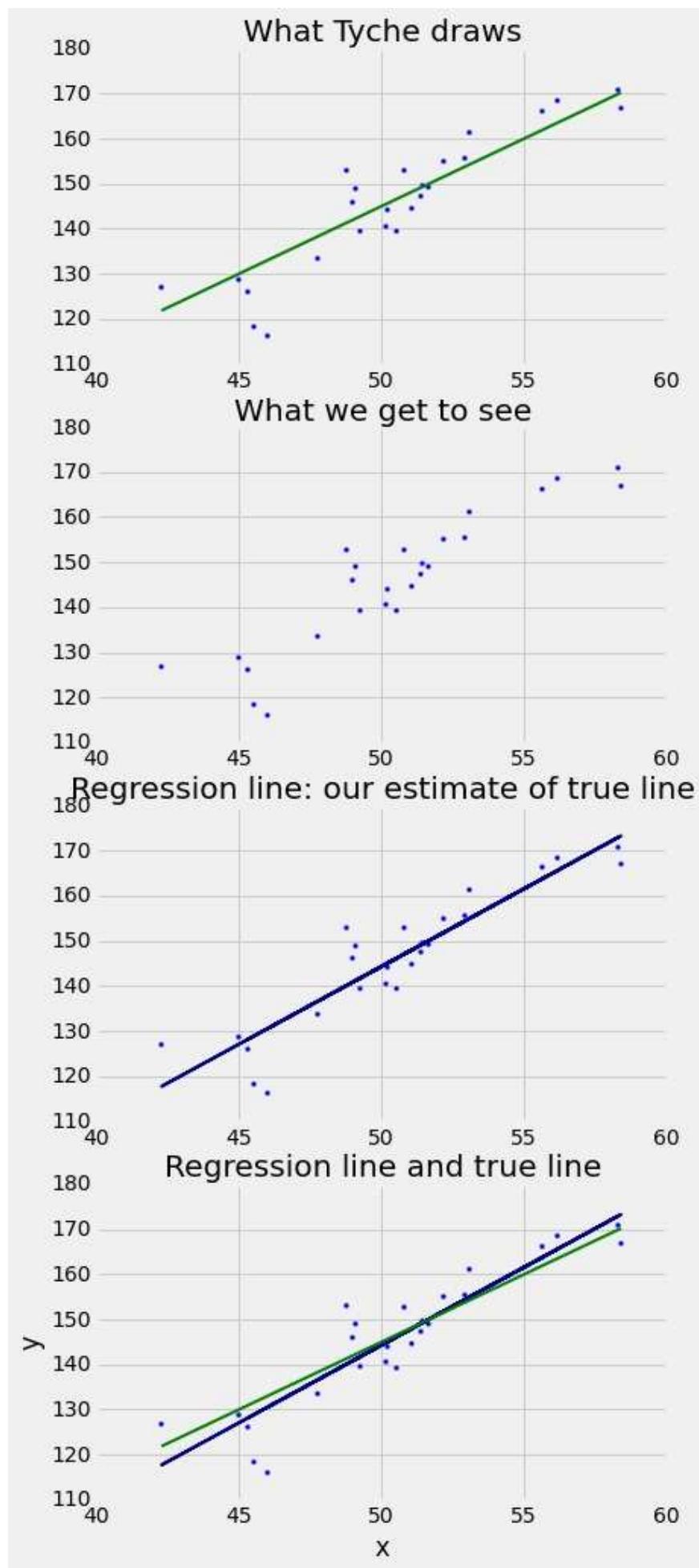
```

```

# Tyche's true line,
# the points she creates,
# and our estimate of the true line.
# Arguments: true slope, true intercept, number of points

draw_and_compare(3, -5, 25)

```



# Prediction using Regression

In reality, of course, we are not Tyche, and we will never see the true line. What the simulation shows that if the regression model looks plausible, and if we have a large sample, then the regression line is a good approximation to the true line.

The scatter diagram of birth weights versus gestational days looks roughly linear. Let us assume that the regression model holds. Suppose now that at the hospital there is a new baby who has 300 gestational days. We can use the regression line to predict the birth weight of this baby.

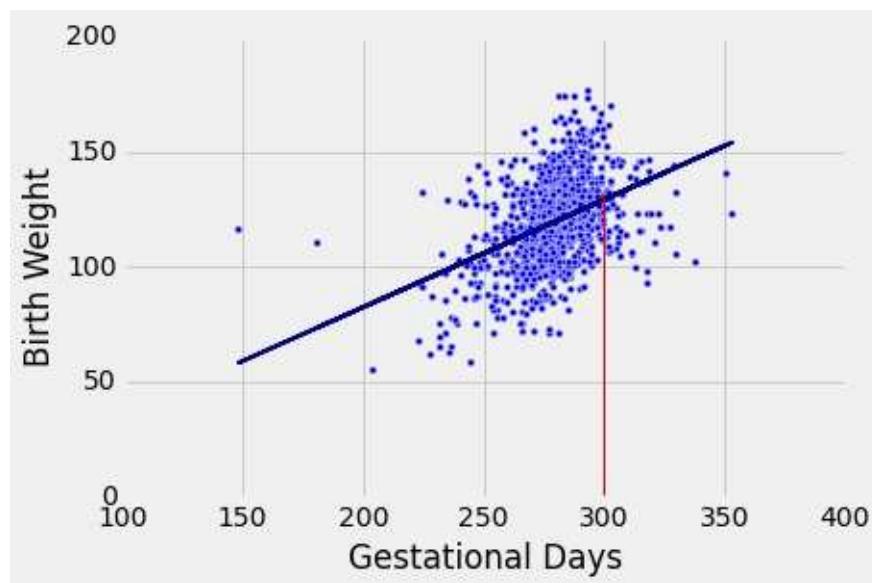
As we saw earlier, the fitted value at 300 gestational days was about 129.2 ounces. That is our prediction for the birth weight of the new baby.

```
fit_300 = fitted_value(baby, 'Gestational Days', 'Birth Weight', 300)
fit_300
```

```
129.2129241703143
```

The figure below shows where the prediction lies on the regression line. The red line is at  $x = 300$ .

```
scatter_fit(baby, 'Gestational Days', 'Birth Weight')
plots.scatter(300, fit_300, color='red', s=20)
plots.plot([300, 300], [0, fit_300], color='red', lw=1)
plots.ylim([0, 200])
None
```



## The Variability of the Prediction

We have developed a method for predicting a new baby's birthweight based on the number of gestational days. But as data scientists, we know that the sample might have been different. Had the sample been different, the regression line would have been different too, and so would our prediction. To see how good our prediction is, we must get a sense of how variable the prediction can be.

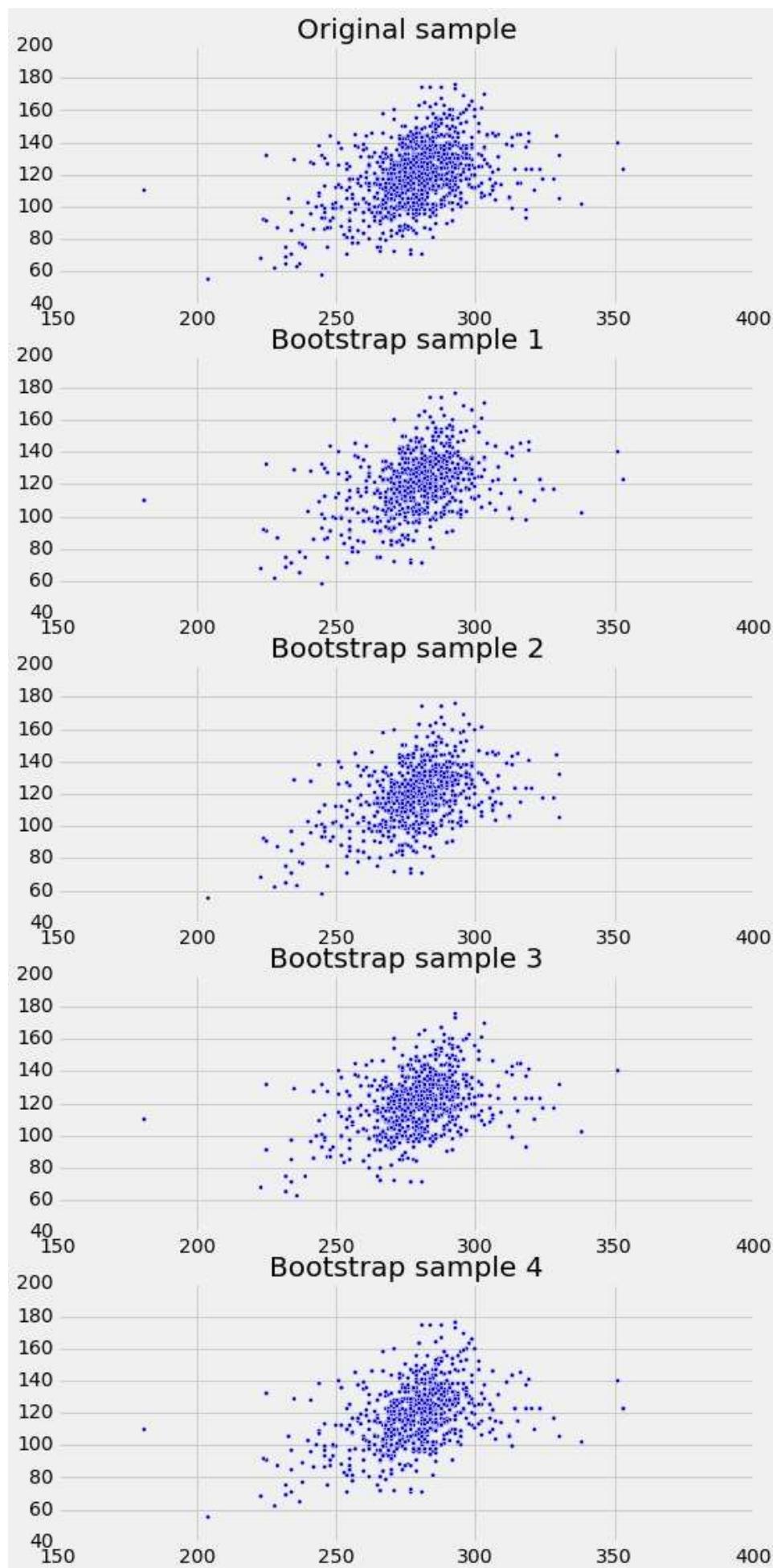
One way to do this would be by generating new random samples of points and making a prediction based on each new sample. To generate new samples, we can **bootstrap the scatter plot**.

Specifically, we can simulate new samples by random sampling with replacement from the original scatter plot, as many times as there are points in the scatter.

Here is the original scatter diagram from the sample, and four replications of the bootstrap resampling procedure. Notice how the resampled scatter plots are in general a little more sparse than the original. That is because some of the original point do not get selected in the samples.

```
plots.figure(figsize=(8, 18))
plots.subplot(5, 1, 1)
plots.scatter(baby[1], baby[0], s=10)
plots.xlim([150, 400])
plots.title('Original sample')

for i in np.arange(1, 5, 1):
    plots.subplot(5,1,i+1)
    rep = baby.sample(with_replacement=True)
    plots.scatter(rep[1], rep[0], s=10)
    plots.xlim([150, 400])
    plots.title('Bootstrap sample '+str(i))
```



The next step is to fit the regression line to the scatter plot in each replication, and make a prediction based on each line. The figure below shows 10 such lines, and the corresponding predicted birth weight at 300 gestational days.

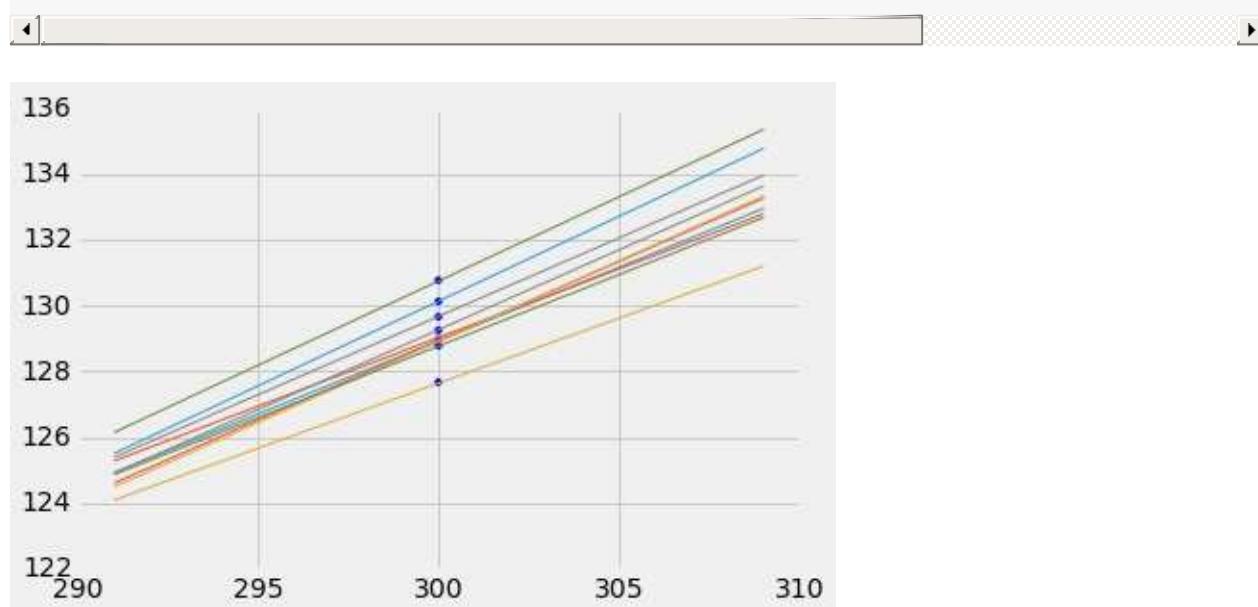
```
x = 300

lines = Table(['slope','intercept'])
for i in range(10):
    rep = baby.sample(with_replacement=True)
    a = slope(rep, 'Gestational Days', 'Birth Weight')
    b = intercept(rep, 'Gestational Days', 'Birth Weight')
    lines.append([a, b])

lines['prediction at x='+str(x)] = lines.column('slope')*x + lines.

xlims = np.array([291, 309])
left = xlims[0]*lines[0] + lines[1]
right = xlims[1]*lines[0] + lines[1]
fit_x = x*lines['slope'] + lines['intercept']

for i in range(10):
    plots.plot(xlims, np.array([left[i], right[i]]), lw=1)
    plots.scatter(x, fit_x[i], s=20)
```



The predictions vary from one line to the next. The table below shows the slope and intercept of each of the 10 lines, along with the prediction.

lines

slope	intercept	prediction at x=300
0.446731	-5.08136	128.938
0.417925	3.65604	129.034
0.395636	8.95328	127.644
0.512511	-22.9981	130.755
0.477034	-13.4303	129.68
0.515773	-24.5881	130.144
0.48213	-15.7025	128.936
0.492106	-18.7225	128.909
0.434989	-1.72409	128.773
0.486076	-16.5506	129.272

## Bootstrap Prediction Interval

If we increase the number of repetitions of the resampling process, we can generate an empirical histogram of the predictions. This will allow us to create a prediction interval, using methods like those we used earlier to create bootstrap confidence intervals for numerical parameters.

Let us define a function called `bootstrap_prediction` to do this. The function takes five arguments:

- the name of the table
- the column labels of the predictor and response variables, in that order
- the value of  $x$  at which to make the prediction
- the desired number of bootstrap repetitions

In each repetition, the function bootstraps the original scatter plot and calculates the equation of the regression line through the bootstrapped plot. It then uses this line to find the predicted value of  $y$  based on the specified value of  $x$ . Specifically, it calls the function `fitted_value` that we defined earlier in this section, to calculate the slope and intercept of the regression line and then calculate the fitted value at the specified  $x$ .

Finally, it collects all of the predicted values into a column of a table, draws the empirical histogram of these values, and prints the interval consisting of the "middle 95%" of the predicted values. It also prints the predicted value based on the regression line through the original scatter plot.

```
# Bootstrap prediction of variable y at new_x
# Data contained in table; prediction by regression of y based on x
# repetitions = number of bootstrap replications of the original sc

def bootstrap_prediction(table, x, y, new_x, repetitions):

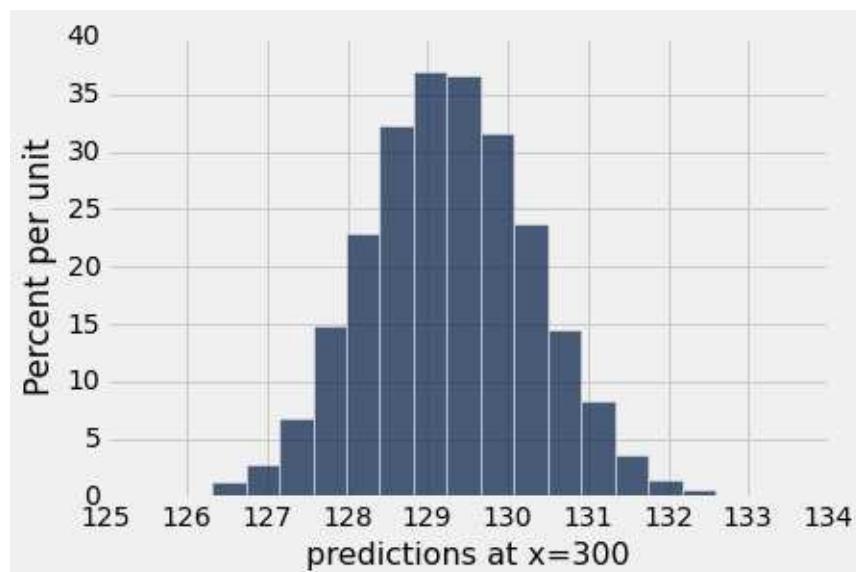
    # For each repetition:
    # Bootstrap the scatter;
    # get the regression prediction at new_x;
    # augment the predictions list
    predictions = []
    for i in range(repetitions):
        bootstrap_sample = table.sample(with_replacement=True)
        predicted_value = fitted_value(bootstrap_sample, x, y, new_x)
        predictions.append(predicted_value)

    # Prediction based on original sample
    original = fitted_value(table, x, y, new_x)

    # Display results
    pred = Table().with_column('Prediction', predictions)
    pred.hist(bins=20)
    plots.xlabel('predictions at x=' + str(new_x))
    print('Height of regression line at x=' + str(new_x) + ':', original)
    print('Approximate 95%-confidence interval:')
    print((pred.percentile(2.5).rows[0][0], pred.percentile(97.5).r
```

```
bootstrap_prediction(baby, 'Gestational Days', 'Birth Weight', 300,
```

```
Height of regression line at x=300: 129.21292417
Approximate 95%-confidence interval:
(127.25683835787581, 131.33042912205815)
```



The figure above shows a bootstrap empirical histogram of the predicted birth weight of a baby at 300 gestational days, based on 5,000 repetitions of the bootstrap process. The empirical distribution is roughly normal.

An approximate 95% prediction interval of scores has been constructed by taking the "middle 95%" of the predictions, that is, the interval from the 2.5th percentile to the 97.5th percentile of the predictions. The interval ranges from about 127 to about 131. The prediction based on the original sample was about 129, which is close to the center of the interval.

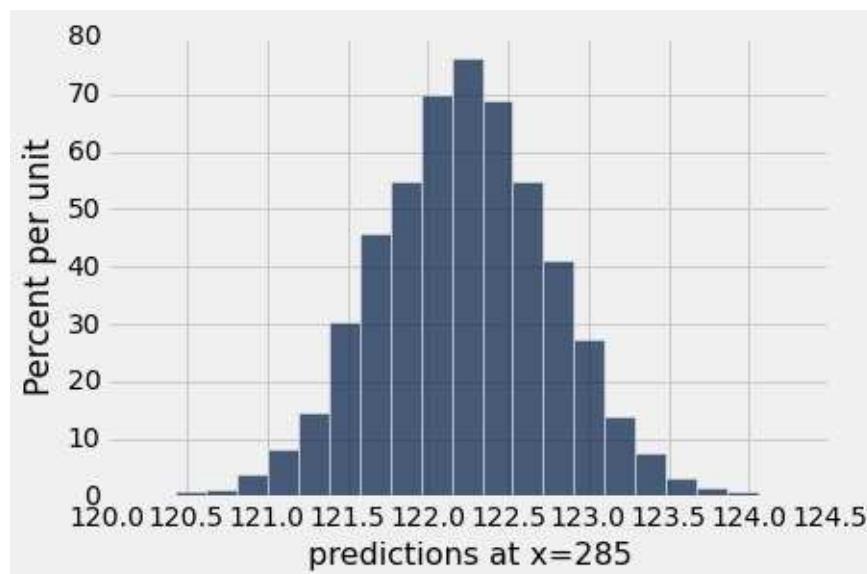
The figure below shows the histogram of 5,000 bootstrap predictions at 285 gestational days. The prediction based on the original sample is about 122 ounces, and the interval ranges from about 121 ounces to about 123 ounces.

```
bootstrap_prediction(baby, 'Gestational Days', 'Birth Weight', 285,
```

Height of regression line at x=285: 122.214571016

Approximate 95%-confidence interval:

(121.17047827189757, 123.29506281889765)



Notice that this interval is narrower than the prediction interval at 300 gestational days. Let us investigate the reason for this.

The mean number of gestational days is about 279 days:

```
np.mean(baby['Gestational Days'])
```

```
279.10136286201021
```

So 285 is nearer to the center of the distribution than 300 is. Typically, the regression lines based on the bootstrap samples are closer to each other near the center of the distribution of the predictor variable. Therefore all of the predicted values are closer together as well. This explains the narrower width of the prediction interval.

You can see this in the figure below, which shows predictions at  $x = 285$  and  $x = 300$  for each of ten bootstrap replications. Typically, the lines are farther apart at  $x = 300$  than at  $x = 285$ , and therefore the predictions at  $x = 300$  are more variable.

```

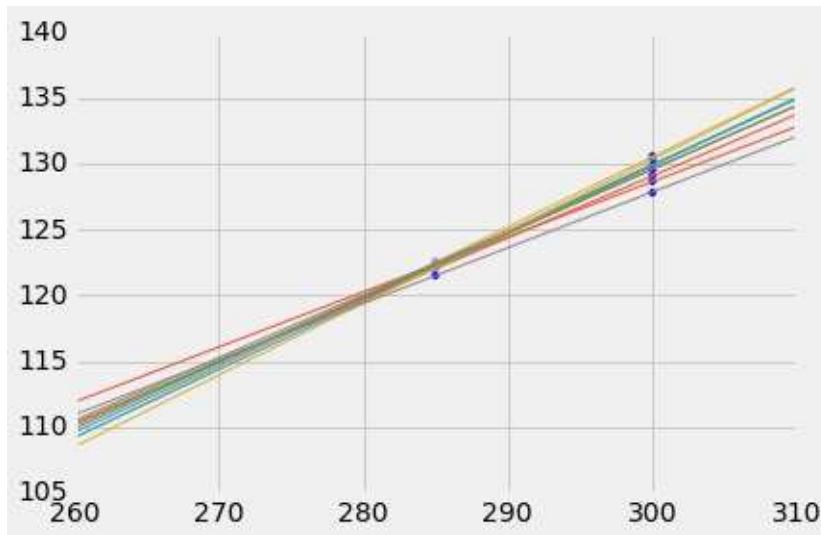
x1 = 300
x2 = 285

lines = Table(['slope','intercept'])
for i in range(10):
    rep = baby.sample(with_replacement=True)
    a = slope(rep, 'Gestational Days', 'Birth Weight')
    b = intercept(rep, 'Gestational Days', 'Birth Weight')
    lines.append([a, b])

xlims = np.array([260, 310])
left = xlims[0]*lines[0] + lines[1]
right = xlims[1]*lines[0] + lines[1]
fit_x1 = x1*lines['slope'] + lines['intercept']
fit_x2 = x2*lines['slope'] + lines['intercept']

plots.xlim(xlims)
for i in range(10):
    plots.plot(xlims, np.array([left[i], right[i]]), lw=1)
    plots.scatter(x1, fit_x1[i], s=20)
    plots.scatter(x2, fit_x2[i], s=20)

```



**Is the observed slope real?¶**

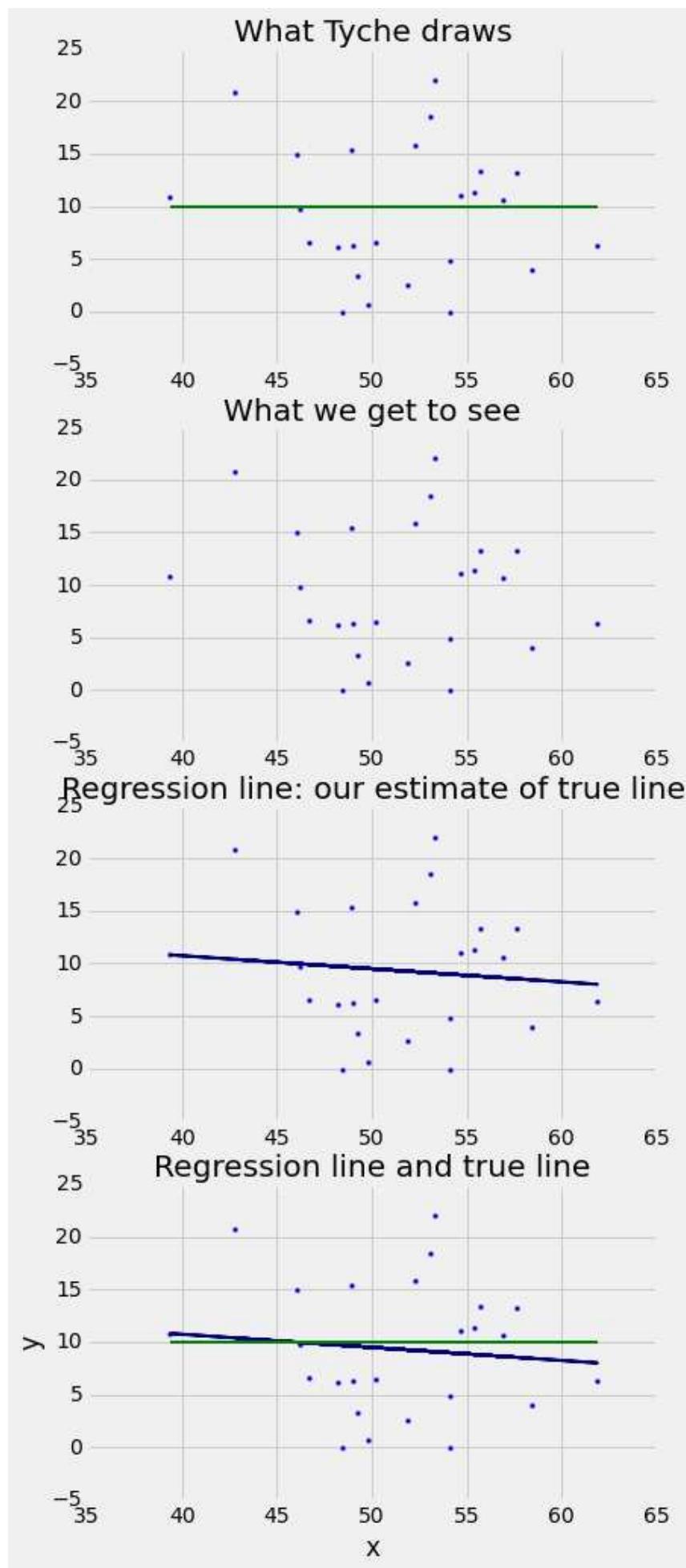
We have developed a method of predicting the birth weight of a new baby in the hospital system, based on the baby's number of gestational days. Our prediction makes use of the positive association that we observed between birth weight and gestational days.

But what if that observation is spurious? In other words, what if Tyche's true line was flat, and the positive association that we observed was just due to randomness in the points that she generated?

Here is a simulation that illustrates why this question arises. We will once again call the function `draw_and_compare`, this time requiring Tyche's true line to have slope 0. Our goal is to see whether our regression line shows a slope that is not 0.

Remember that the arguments to the function `draw_and_compare` are the slope and the intercept of Tyche's true line, and the number of points that she generates.

```
draw_and_compare(0, 10, 25)
```



Run the simulation a few times, keeping the slope of Tyche's line 0 each time. You will notice that while the slope of the true line is 0, the slope of the regression typically is not 0. The regression line sometimes slopes upwards, and sometimes downwards, each time giving us a false impression that the two variables are correlated.

## Testing whether the slope of the true line is 0

These simulations show that before we put too much faith in our regression predictions, it is worth checking whether the true line does indeed have a slope that is not 0.

We are in a good position to do this, because we know how to bootstrap the scatter diagram and draw a regression line through each bootstrapped plot. Each of those lines has a slope, so we can simply collect all the slopes and draw their empirical histogram. We can then construct an approximate 95% confidence interval for the slope of the true line, using the bootstrap percentile method that is now so familiar.

If the confidence interval for the true slope does not contain 0, then we can be about 95% confident that the true slope is not 0. If the interval does contain 0, then we cannot conclude that there is a genuine linear association between the variable that we are trying to predict and the variable that we have chosen to use as a predictor. We might therefore want to look for a different predictor variable on which to base our predictions.

The function `bootstrap_slope` executes our plan. Its arguments are the name of the table and the labels of the predictor and response variables, and the desired number of bootstrap replications. In each replication, the function bootstraps the original scatter plot and calculates the slope of the resulting regression line. It then draws the histogram of all the generated slopes and prints the interval consisting of the "middle 95%" of the slopes.

Notice that the code for `bootstrap_slope` is almost identical to that for `bootstrap_prediction`, except that now all we need from each replication is the slope, not a predicted value.

```

def bootstrap_slope(table, x, y, repetitions):

    # For each repetition:
    # Bootstrap the scatter, get the slope of the regression line,
    # augment the list of generated slopes
    slopes = []
    for i in range(repetitions):
        bootstrap_scatter = table.sample(with_replacement=True)
        slopes.append(slope(bootstrap_scatter, x, y))

    # Slope of the regression line from the original sample
    observed_slope = slope(table, x, y)

    # Display results
    slopes = Table().with_column('slopes', slopes)
    slopes.hist(bins=20)
    print('Slope of regression line:', observed_slope)
    print('Approximate 95%-confidence interval for the true slope:')
    print((slopes.percentile(2.5).rows[0][0], slopes.percentile(97.

```



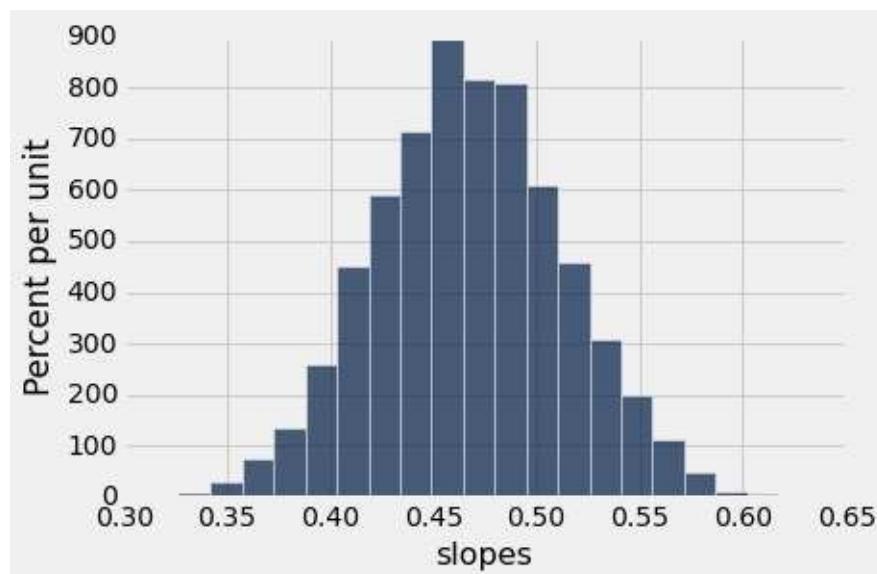
Let us call `bootstrap_slope` to see if that the true linear relation between birth weight and gestational days has slope 0; in other words, if Tyche's true line is flat.

```
bootstrap_slope(baby, 'Gestational Days', 'Birth Weight', 5000)
```

```

Slope of regression line: 0.466556876949
Approximate 95%-confidence interval for the true slope:
(0.37981209488027268, 0.55753912103512426)

```



The approximate 95% confidence interval for the true slope runs from about 0.38 ounces per day to about 0.56 ounces per day. This interval does not contain 0. Therefore we can conclude, with about 95% confidence, that the slope of the true line is not 0.

This conclusion supports our choice of using gestational days to predict birth weight.

Suppose now we try to estimate the birth weight of the baby based on the mother's age. Based on the sample, the slope of the regression line for estimating birth weight based on maternal age is positive, about 0.08 ounces per year:

```
slope(baby, 'Maternal Age', 'Birth Weight')
```

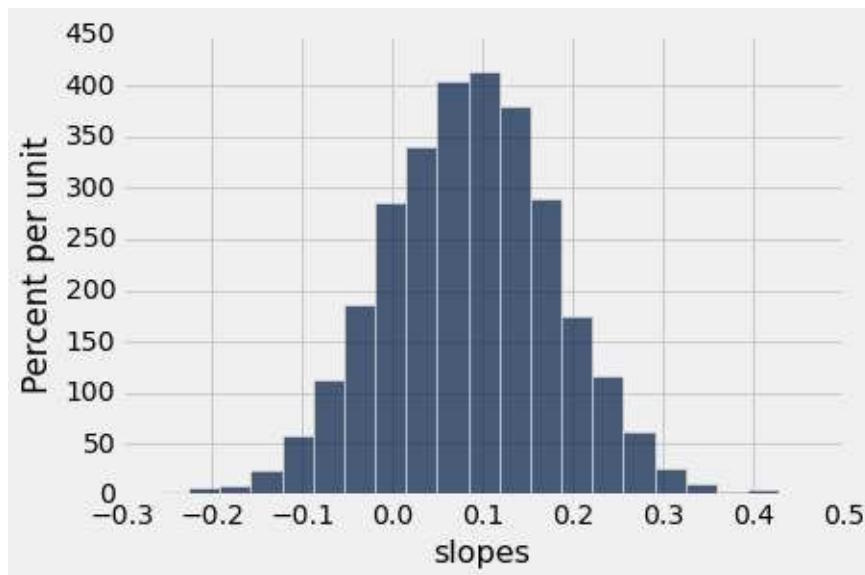
```
0.085007669415825132
```

But an approximate 95% bootstrap confidence interval for the true slope has a negative left end point and a positive right end point – in other words, the interval contains 0. Therefore we cannot reject the hypothesis that the slope of the true linear relation between maternal age and baby's birth weight is 0. Based on this observation, it would be unwise to predict birth weight based on the regression model with maternal age as the predictor.

```
bootstrap_slope(baby, 'Maternal Age', 'Birth Weight', 5000)
```

Slope of regression line: 0.0850076694158

Approximate 95%-confidence interval for the true slope:  
 $(-0.10015674488514391, 0.26974979944992733)$



## Using confidence intervals to test hypotheses

Notice how we have used confidence intervals for the true slope to test the hypothesis that the true slope is 0. Constructing a confidence interval for a parameter is one way to test the null hypothesis that the parameter has a specified value. If that value is not in the confidence interval, we can reject the null hypothesis that the parameter has that value. If the value is in the confidence interval, we do not have enough evidence to reject the null hypothesis.

If the sample size is large and if the empirical distribution of the estimate of the parameter is roughly normal, then this method of testing via confidence intervals is justifiable. Using a 95% confidence interval corresponds to using a 5% cutoff for the P-value. The justifications of these statements are rather technical, and we will leave them to a more advanced course. For now, it is enough to note that confidence intervals can be used to test hypotheses in this way.

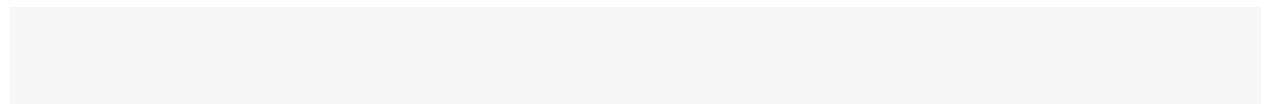
Indeed, the method is more constructive than other methods of testing hypotheses, because not only does it result in a decision about whether or not to reject the null hypothesis, it also provides an estimate of the parameter. For example, in the example of using gestational days to estimate birth weights, we didn't just conclude that the true slope was not 0 – we also estimated that the true slope is between 0.38 ounces per day to 0.56 ounces per day.

## Words of caution

All of the predictions and tests that we have performed in this section assume that the regression model holds. Specifically, the methods assume that the scatter plot resembles points generated by starting with points that are on a straight line and then pushing them off the line by adding random noise.

If the scatter plot does not look like that, then perhaps the model does not hold for the data. If the model does not hold, then calculations that assume the model to be true are not valid.

Therefore, we must first decide whether the regression model holds for our data, before we start making predictions based on the model or testing hypotheses about parameters of the model. A simple way is to do what we did in this section, which is to draw the scatter diagram of the two variables and see whether it looks roughly linear and evenly spread out around a line. In the next section, we will study some more tools for assessing the model.



# Regression Diagnostics

## Interact

The methods that we have developed for inference in regression are valid if the regression model holds for our data. If the data do not satisfy the assumptions of the model, then it can be hard to justify calculations that assume that the model is good.

In this section we will develop some simple descriptive methods that can help us decide whether our data satisfy the regression model.

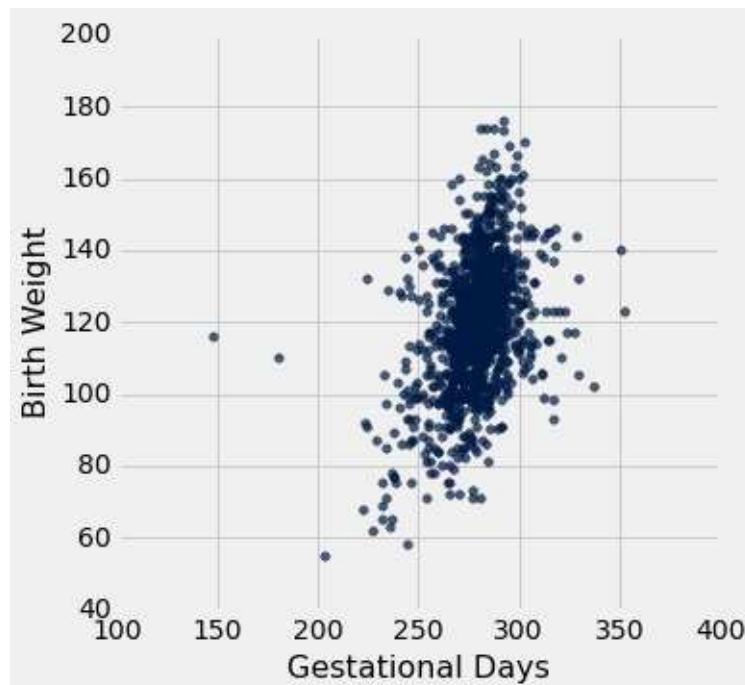
We will start with a review of the details of the model, which specifies that the points in the scatter plot are generated at random as follows.

- Tyche creates the scatter plot by starting with points that lie perfectly on a straight line and then pushing them off the line vertically, either above or below, as follows:
  - For each  $x$ , Tyche finds the corresponding point on the true line, and then adds an error.
  - The errors are drawn at random with replacement from a population of errors that has a normal distribution with mean 0 and an unknown standard deviation.
  - Tyche creates a point whose horizontal coordinate is  $x$  and whose vertical coordinate is "the height of the true line at  $x$ , plus the error". If the error is positive, the point is above the line. If the error is negative, the point is below the line.
- We get to see the resulting points but not the true line on which the points started out nor the random errors that Tyche added.

## First Diagnostic: The Scatter Plot

It is always helpful to start with a visualization. Continuing the example of estimating birth weight based on gestational days, let us look at a scatter plot of the two variables.

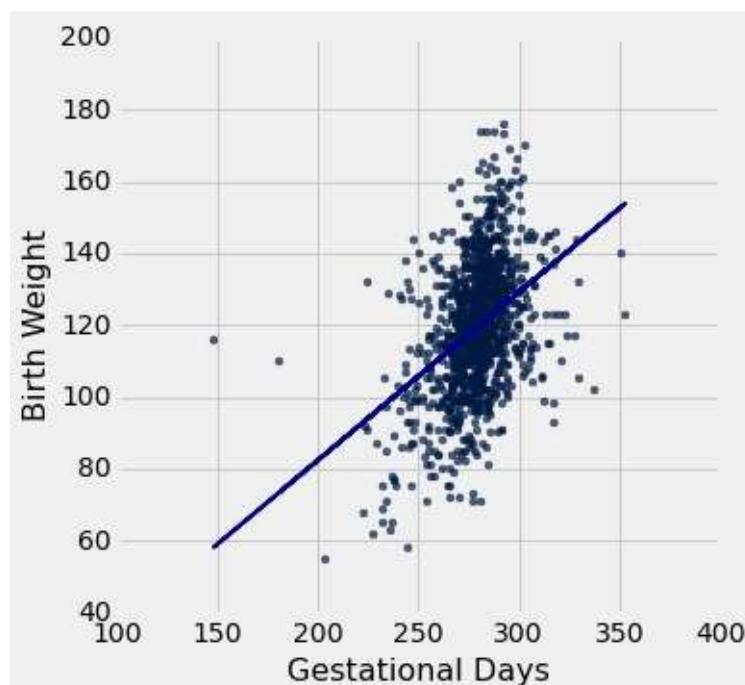
```
baby.scatter('Gestational Days', 'Birth Weight')
```



The points do appear to be roughly clustered around a straight line. There is no strikingly noticeable curve in the scatter.

A plot of the regression line through the scatter plot shows that about half the points are above the line and half are below the line almost symmetrically, supporting the model's assumption that Tyche is adding random errors that are normally distributed with mean 0.

```
scatter_fit(baby, 'Gestational Days', 'Birth Weight')
```



## Second Diagnostic: A Residual Plot

We cannot see the true line that is the basis of the regression model, but the regression line that we calculate acts as a substitute for it. So also the vertical distances of the points from the regression line act as substitutes for the random errors that Tyche uses to push the points vertically off the true line.

As we have seen earlier, the vertical distances of the points from the regression line are called *residuals*. There is one residual for each point in the scatter plot. The residual is the difference between the observed value of  $y$  and the fitted value of  $y$ :

For the point  $(x, y)$ ,

$$\begin{aligned}\text{residual} &= y - \text{fitted value of } y = y \\ &\quad - \text{height of regression line at } y\end{aligned}$$

The table `baby_regression` contains the data, the fitted values and the residuals:

```
baby2 = baby.select(['Gestational Days', 'Birth Weight'])
fitted = fit(baby, 'Gestational Days', 'Birth Weight')
residuals = baby.column('Birth Weight') - fitted
baby_regression = baby2.with_columns([
    'Fitted Value', fitted,
    'Residual', residuals
])
baby_regression
```

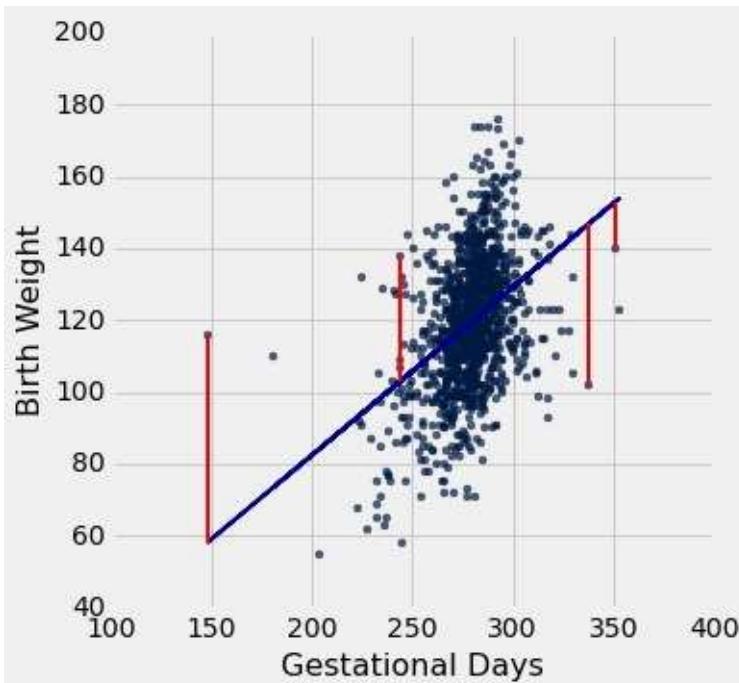
Gestational Days	Birth Weight	Fitted Value	Residual
284	120	121.748	-1.74801
282	113	120.815	-7.8149
279	128	119.415	8.58477
282	108	120.815	-12.8149
286	136	122.681	13.3189
244	138	103.086	34.9143
245	132	103.552	28.4477
289	120	124.081	-4.0808
299	143	128.746	14.2536
351	140	153.007	-13.0073

... (1164 rows omitted)

Once again, it helps to visualize. The figure below shows four of the residuals; for each of the four points, the length of the red segment is the residual for that point. There is one such residual for each of the other 1170 points as well.

```
points = Table().with_columns([
    'Gestational Days', [148, 244, 338, 351],
    'Birth Weight', [116, 138, 102, 140]
])

scatter_fit(baby, 'Gestational Days', 'Birth Weight')
for i in range(4):
    f = fitted_value(baby, 'Gestational Days', 'Birth Weight', poi
plots.plot([points.column(0)[i]]*2, [f, points.column(1)[i]], c
```



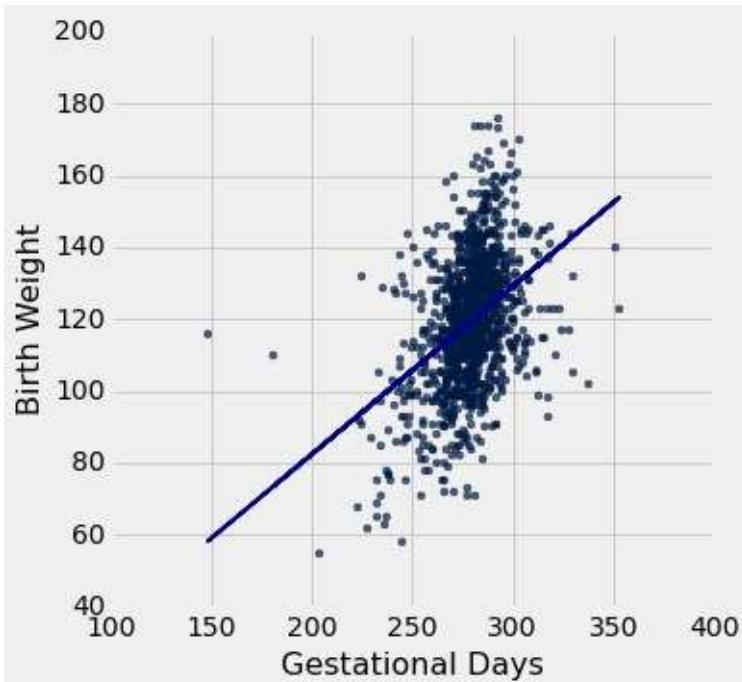
According to the regression model, Tyche generates errors by drawing at random with replacement from a population of errors that is normally distributed with mean 0. To see whether this looks plausible for our data, we will examine the residuals, which are our substitutes for Tyche's errors.

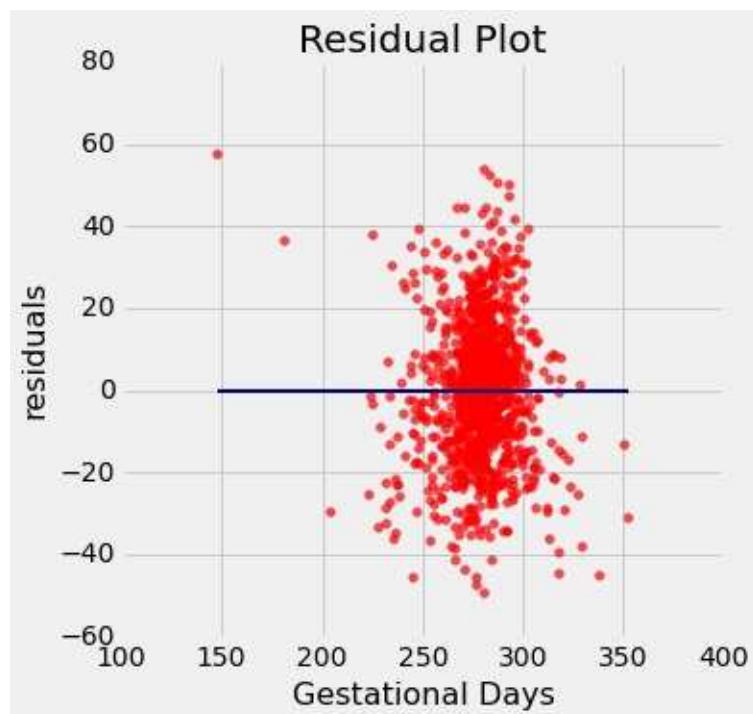
A *residual plot* can be drawn by plotting the residuals against  $x$ . The function `residual_plot` does just that. The function `regression_diagnostic_plots` draws the original scatter plot as well as the residual plot for ease of comparison.

```
def residual_plot(table, x, y):
    fitted = fit(table, x, y)
    residuals = table[y] - fitted
    t = Table().with_columns([
        x, table[x],
        'residuals', residuals
    ])
    t.scatter(x, 'residuals', color='r')
    xlims = [min(table[x]), max(table[x])]
    plots.plot(xlims, [0,0], color='darkblue', lw=2)
    plots.title('Residual Plot')
```

```
def regression_diagnostic_plots(table, x, y):
    scatter_fit(table, x, y)
    residual_plot(table, x, y)
```

```
regression_diagnostic_plots(baby, 'Gestational Days', 'Birth Weight')
```

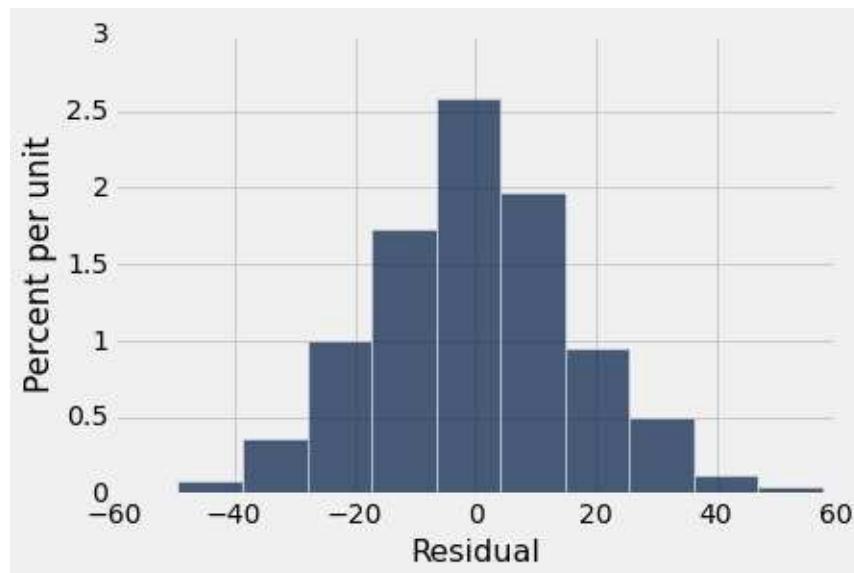




The height of each red point in the plot above is a residual. Notice how the residuals are distributed fairly symmetrically above and below the horizontal line at 0. Notice also that the vertical spread of the plot is fairly even across the most common values of  $x$ , in about the 200-300 range of gestational days. In other words, apart from a few outlying points, the plot isn't narrower in some places and wider in others.

All of this is consistent with the model's assumptions that the errors are drawn at random with replacement from a normally distributed population with mean 0. Indeed, a histogram of the residuals is centered at 0 and looks roughly bell shaped.

```
baby_regression.select('Residual').hist()
```



These observations show us that the data are fairly consistent with the assumptions of the regression model. Therefore it makes sense to do inference based on the model, as we did in the previous section.

## Using the Residual Plot to Detect Nonlinearity

Residual plots can be used to detect whether the regression model does not hold. The most serious departure from the model is non-linearity.

If two variables have a non-linear relation, the non-linearity is sometimes visible in the scatter plot. Often, however, it is easier to spot non-linearity in a residual plot than in the original scatter plot. This is usually because of the scales of the two plots: the residual plot allows us to zoom in on the errors and hence makes it easier to spot patterns.



As an example, we will study a [dataset](#) on the age and length of dugongs, which are marine mammals related to manatees and sea cows. The data are in a table called `dugong`. Age is measured in years and length in meters. The ages are estimated; most dugongs don't keep track of their birthdays.

```
dugong = Table.read_table('http://www.statsci.org/data/oz/dugongs.t  
dugong
```

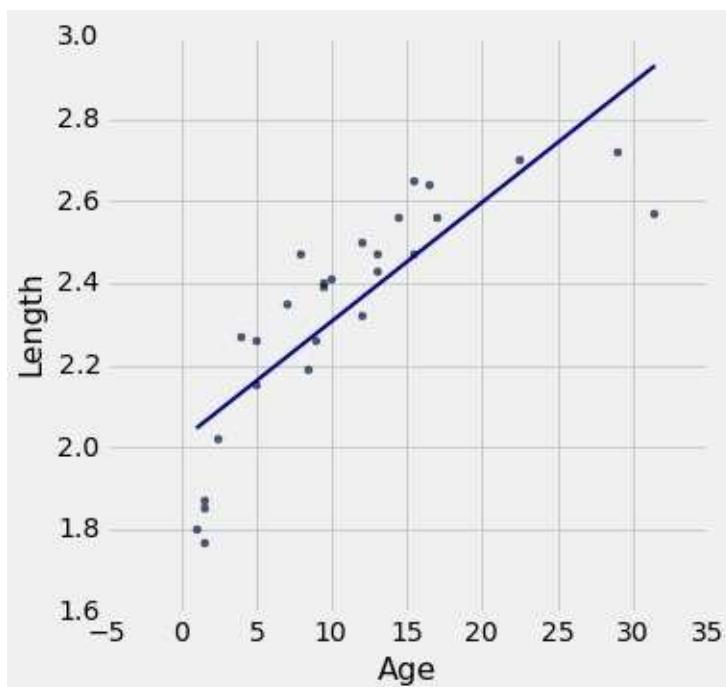
Age	Length
1	1.8
1.5	1.85
1.5	1.87
1.5	1.77
2.5	2.02
4	2.27
5	2.15
5	2.26
7	2.35
8	2.47
... (17 rows omitted)	

Here is a regression of length (the response) on age (the predictor). The correlation between the two variables is about 0.83, which is quite high.

```
correlation(dugong, 'Age', 'Length')
```

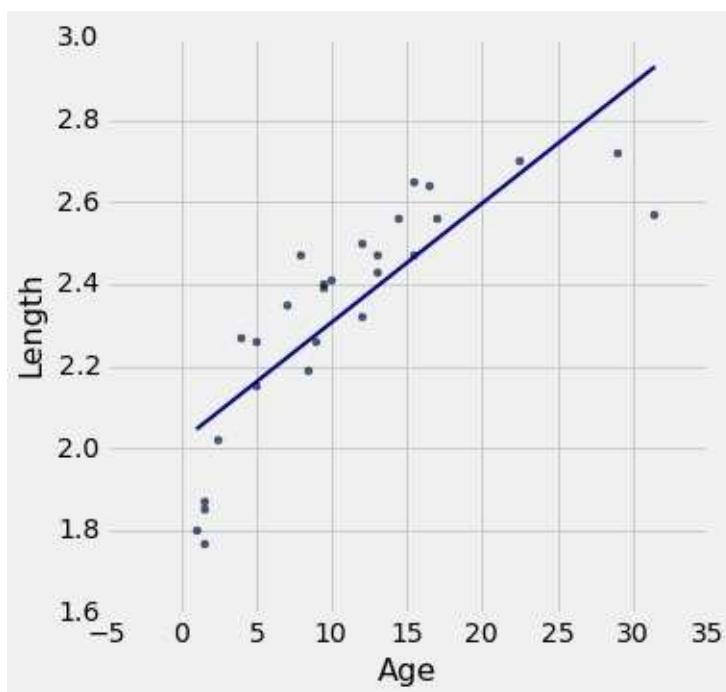
```
0.82964745549057139
```

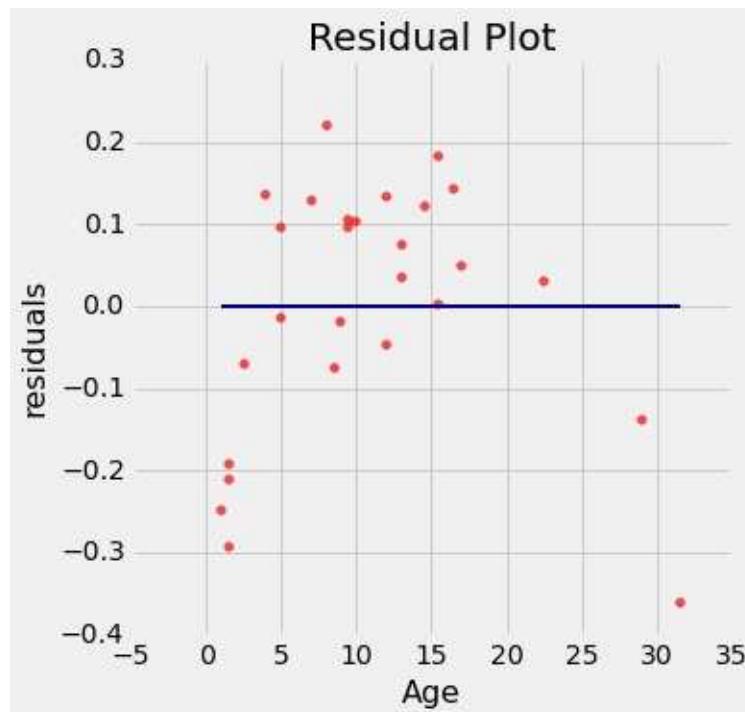
```
scatter_fit(dugong, 'Age', 'Length')
```



High correlation notwithstanding, the plot shows a curved pattern that is much more visible in the residual plot.

```
regression_diagnostic_plots(dugong, 'Age', 'Length')
```





At the low end of ages, the residuals are almost all negative; then they are almost all positive; then negative again at the high end of ages. Such a discernible pattern would have been unlikely had Tyche been picking errors at random with replacement and using them to push points off a straight line. Therefore the regression model does not hold for these data.

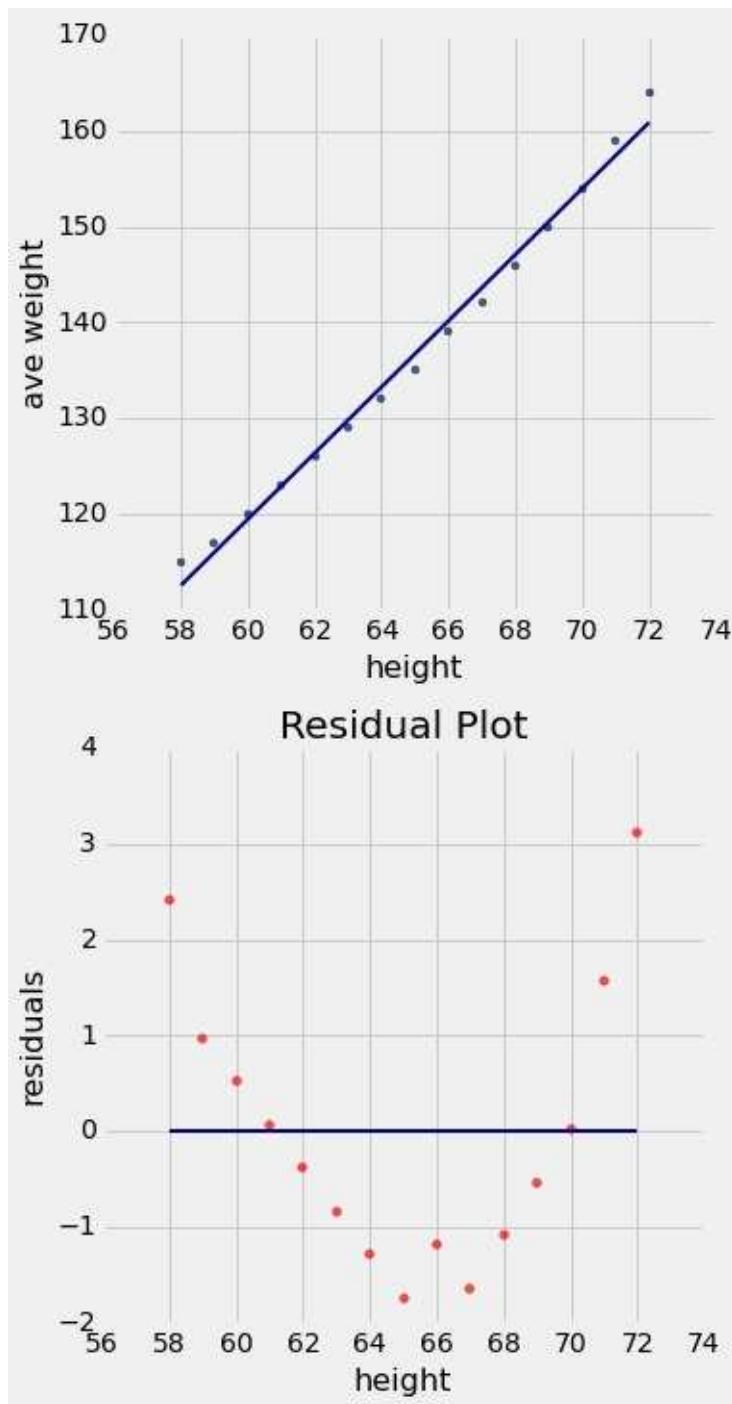
As another example, the dataset `us_women` contains the average weights of U.S. women of different heights in the 1970s. Weights are measured in pounds and heights in inches. The correlation is spectacularly high – more than 0.99.

```
us_women = Table.read_table('us_women.csv')
correlation(us_women, 'height', 'ave weight')
```

```
0.99549476778421608
```

The points seem to hug the regression line pretty closely, but the residual plot flashes a warning sign: it is U-shaped, indicating that the relation between the two variables is not linear.

```
regression_diagnostic_plots(us_women, 'height', 'ave weight')
```



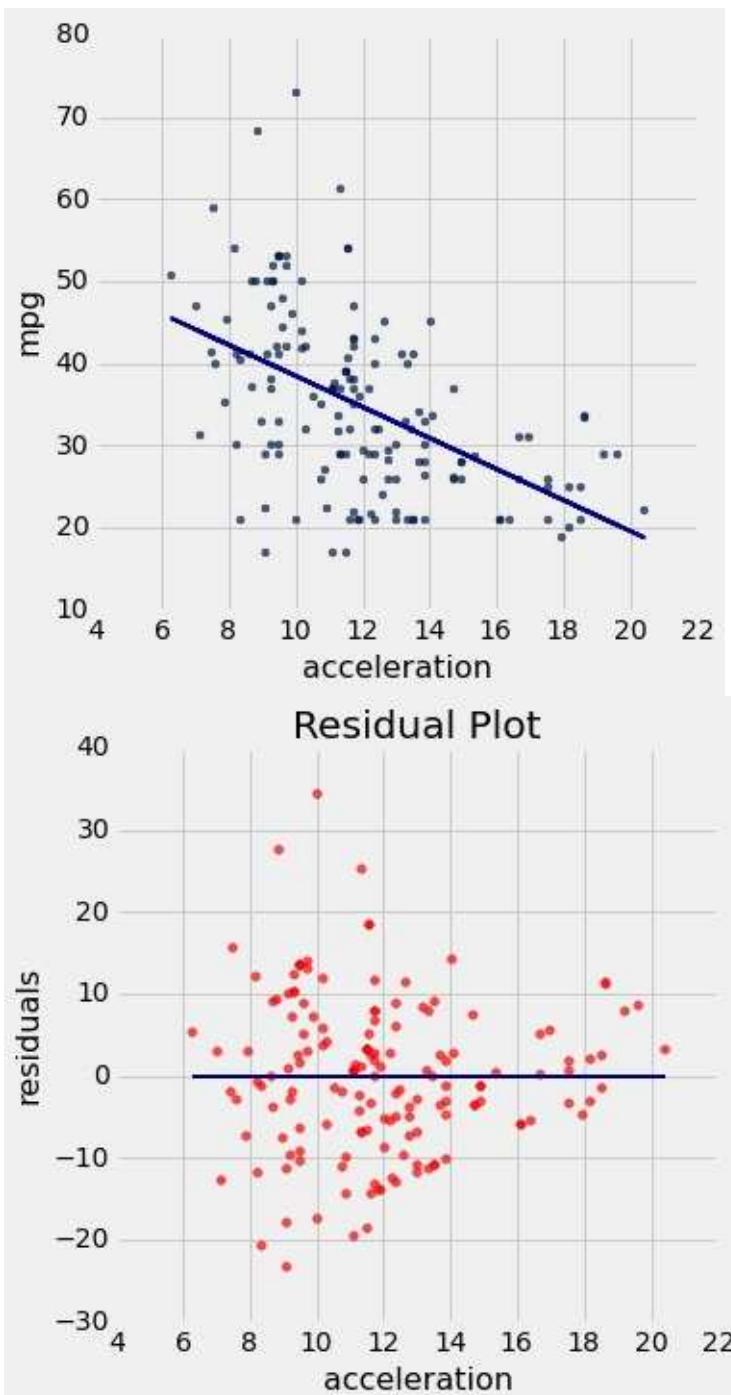
These examples demonstrate that even when correlation is high, fitting a straight line might not be the right thing to do. Residual plots help us decide whether or not to use linear regression.

## Using the Residual Plot to Detect Heteroscedasticity

*Heteroscedasticity* is a word that will surely be of interest to those who are preparing for Spelling Bees. For data scientists, its interest lies in its meaning, which is "uneven spread".

Recall the table `hybrid` that contains data on hybrid cars in the U.S. Here is a regression of fuel efficiency on the rate of acceleration. The association is negative: cars that accelerate quickly tend to be less efficient.

```
regression_diagnostic_plots(hybrid, 'acceleration', 'mpg')
```



Notice how the residual plot flares out towards the low end of the accelerations. In other words, the variability in the size of the errors is greater for low values of acceleration than for high values. Such a pattern would be unlikely under the regression model, because the model assumes that Tyche picks the errors at random with replacement *from the same normally distributed population*.

Uneven vertical variation in a residual plot can indicate that the regression model does not hold. Uneven variation is often more easily noticed in a residual plot than in the original scatter plot.

## The Accuracy of the Regression Estimate

We will end our discussion of regression by pointing out some interesting mathematical facts that lead to a measure of the accuracy of regression. We will leave the proofs to another course, but we will demonstrate the facts by examples. Note that all of the facts hold for all scatter plots, whether or not the regression model is good.

**Fact 1.** No matter what the shape of the scatter diagram, the residuals have an average of 0.

In all the residual plots above, you have seen the horizontal line at 0 going through the center of the plot. That is a visualization of this fact.

As a numerical example, here is the average of the residuals in the regression of birth weight on gestational days of babies:

```
np.mean(baby_regression.column('Residual'))
```

```
-6.3912532279613784e-15
```

That doesn't look like 0, but in fact it is a very small number that is 0 apart from rounding error.

Here is the average of the residuals in the regression of the length of dugongs on their age:

```
dugong_residuals = dugong.column('Length') - fit(dugong, 'Age', 'Length')
np.mean(dugong_residuals)
```

```
-2.8783559897689243e-16
```

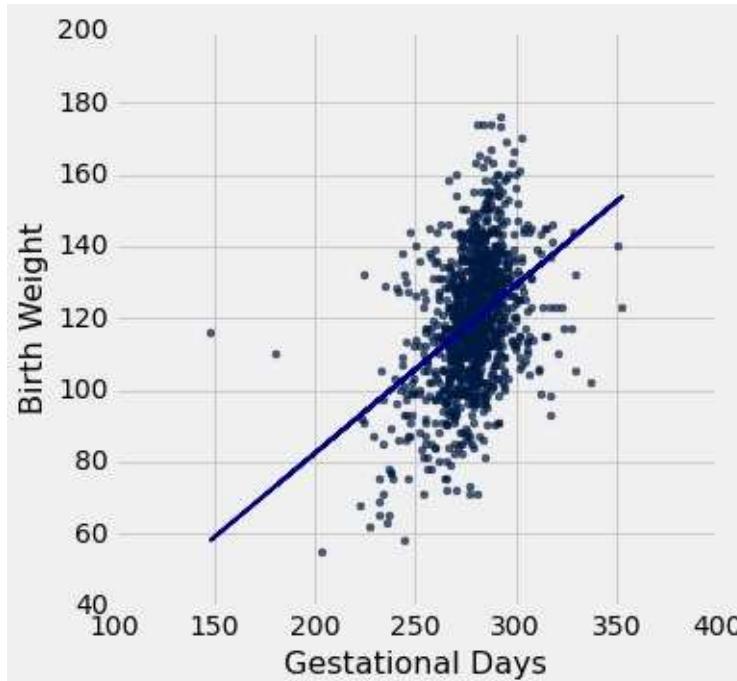
Once again the mean of the residuals is 0 apart from rounding error.

This is analogous to the fact that if you take any list of numbers and calculate the list of deviations from average, the average of the deviations is 0.

**Fact 2.** No matter what the shape of the scatter diagram, the SD of the fitted values is  $|r|$  times the SD of the observed values of  $y$ .

To understand this result, it is worth noting that the fitted values are all on the regression line whereas the observed values of  $y$  are the heights of all the points in the scatter plot and are more variable.

```
scatter_fit(baby, 'Gestational Days', 'Birth Weight')
```



The fitted values of birth weight are, therefore, less variable than the observed values of birth weight. But how much less variable?

The scatter plot shows a correlation of about 0.4:

```
correlation(baby, 'Gestational Days', 'Birth Weight')
```

```
0.40754279338885108
```

Here is ratio of the SD of the fitted values and the SD of the observed values of birth weight:

```
fitted_birth_weight = baby_regression.column('Fitted Value')
observed_birth_weight = baby_regression.column('Birth Weight')
np.std(fitted_birth_weight)/np.std(observed_birth_weight)
```

```
0.40754279338885108
```

The ratio is  $r$ . Thus the SD of the fitted values is  $r$  times the SD of the observed values of  $y$ .

The example of fuel efficiency and acceleration is one where the correlation is negative:

```
correlation(hybrid, 'acceleration', 'mpg')
```

-0.5060703843771186

```
fitted_mpg = fit(hybrid, 'acceleration', 'mpg')
observed_mpg = hybrid.column('mpg')
np.std(fitted_mpg)/np.std(observed_mpg)
```

0.5060703843771186

The ratio of the two SDs is  $|r|$ . Thus the SD of the fitted values is  $|r|$  times the SD of the observed values of fuel efficiency.

The relation says that the closer the scatter diagram is to a straight line, the closer the fitted values will be to the observed values of  $y$ , and therefore the closer the variance of the fitted values will be to the variance of the observed values.

Therefore, the closer the scatter diagram is to a straight line, the closer  $r^2$  will be to 1, and the closer  $r$  will be to 1 or -1.

**Fact 3.** No matter what the shape of the scatter diagram, the SD of the residuals is  $\sqrt{1 - r^2}$  times the SD of the observed values of  $y$ . In other words, the root mean squared error of regression is  $\sqrt{1 - r^2}$  times the SD of  $y$ .

This fact gives a measure of the accuracy of the regression estimate.

Again, we will demonstrate this by example. In the case of gestational days and birth weights,  $r$  is about 0.4 and  $\sqrt{1 - r^2}$  is about 0.91:

```
r = correlation(baby, 'Gestational Days', 'Birth Weight')
np.sqrt(1 - r**2)
```

0.91318611003278638

```
residuals = baby_regression.column('Residual')
observed_birth_weight = baby_regression.column('Birth Weight')
np.std(residuals)/np.std(observed_birth_weight)
```

0.9131861100327866

Thus the SD of the residuals is  $\sqrt{1 - r^2}$  times the SD of the observed birth weights.

It will come as no surprise that the same relation is true for the regression of fuel efficiency on acceleration:

```
r = correlation(hybrid, 'acceleration', 'mpg')
np.sqrt(1 - r**2)
```

0.862492183185677

```
residuals = hybrid.column('mpg') - fitted_mpg
observed_mpg = hybrid.column('mpg')
np.std(residuals)/np.std(observed_mpg)
```

0.862492183185677

Let us examine the extreme cases of the general fact that the SD of the residuals is  $\sqrt{1 - r^2}$  times the SD of the observed values of  $y$ . Remember the residuals always average out to 0.

- If  $r = 1$  or  $r = -1$ , then  $\sqrt{1 - r^2} = 0$ . Therefore the residuals have an average of 0 and an SD of 0 as well; therefore, the residuals are all equal to 0. This is consistent with the observation that if  $r = \pm 1$ , the scatter plot is a perfect straight line and there is no error in the regression estimate.
- If  $r = 0$ , the  $\sqrt{1 - r^2} = 1$  and the SD of the regression estimate is equal to the SD of  $y$ . This is consistent with the observation that if  $r = 0$  then the regression line is a flat line at average of  $y$ , and the root mean square error of regression is the root mean squared deviation from the average of  $y$ , which is the SD of  $y$ .

For every other value of  $r$ , the rough size of the error of the regression estimate is somewhere between 0 and the SD of  $y$ .

# **Chapter 5**

## **Probability**

# Conditional Probability

## Interact

Inspired by Peter Norvig's [A Concrete Introduction to Probability](#)

Probability is the study of the observations that are generated by sampling from a known distribution. The statistical methods we have studied so far allow us to reason about the world from data. Probability allows us to reason in the opposite directions: what observations result from known facts about the world.

In practice, we rarely know precisely how random processes in the world work. Even the simplest random process such as [flipping a coin](#) is not as simple as one might assume. Nonetheless, our ability to reason about what data will result from a known random process is useful in many aspects of data science. Fundamentally, the rules of probability allow us to reason about the consequences of assumptions we make.

## Probability Distributions

Probability uses the following vocabulary to describe the relationship between known distributions and their observed outcomes.

- **Experiment:** An occurrence with an uncertain outcome that we can observe. For example, the result of rolling two dice in sequence.
- **Outcome:** The result of an experiment; one particular state of the world. For example: the first die comes up 4 and the second comes up 2. This outcome could be summarized as `(4, 2)`.
- **Sample Space:** The set of all possible outcomes for the experiment. For example, `{(1, 1), (1, 2), (1, 3), ..., (1, 6), (2, 1), (2, 2), ..., (6, 4), (6, 5), (6, 6)}` are all possible outcomes of rolling two dice in sequence. There are  $6 * 6 = 36$  different outcomes.
- **Event:** A subset of possible outcomes that together have some property we are interested in. For example, the event "the two dice sum to 5" is the set of outcomes `{(1, 4), (2, 3), (3, 2), (4, 1)}`.
- **Probability:** The proportion of experiments for which the event occurs. For example, the probability that the two dice sum to 5 is `4/36` or `1/9`.
- **Distribution:** The probability of all events.

The important part of this terminology is that the *sample space* is the set of all *outcomes*, and an *event* is a subset of the sample space. For an outcome that is an element of an event, we say that it is an outcome for which that event occurs.

## Multinomial Distributions

There are many kinds of distributions, but we will focus on the case where the sample space is a fixed, finite set of mutually exclusive outcomes, called a *multinomial* distribution.

For example, either the two dice come up (2, 1) or they come up (4, 5) in a single roll (but both cannot occur), and there are exactly 36 outcome possibilities that each occur with equal chance.

```
two_dice = Table(['First', 'Second', 'Chance'])
for first in np.arange(1, 7):
    for second in np.arange(1, 7):
        two_dice.append([first, second, 1/36])
two_dice.set_format('Chance', PercentFormatter(1))
```

First	Second	Chance
1	1	2.8%
1	2	2.8%
1	3	2.8%
1	4	2.8%
1	5	2.8%
1	6	2.8%
2	1	2.8%
2	2	2.8%
2	3	2.8%
2	4	2.8%
... (26 rows omitted)		

The outcomes are not always equally likely. In the example of two dice rolled in sequence, there are 36 different outcomes that each have a  $1/36$  chance of occurring. If two dice are rolled together and only the sum of the result is observed, instead there are 11 outcomes that have various different chances.

```
two_dice_sums = Table(['Sum', 'Chance']).with_rows([
    [2, 1/36], [3, 2/36], [4, 3/36], [5, 4/36], [6, 5/36], [
    [12, 1/36], [11, 2/36], [10, 3/36], [9, 4/36], [8, 5/36],
]).sort(0)
two_dice_sums.set_format('Chance', PercentFormatter(1))
```

1

Sum	Chance
2	2.8%
3	5.6%
4	8.3%
5	11.1%
6	13.9%
7	16.7%
8	13.9%
9	11.1%
10	8.3%
11	5.6%

... (1 rows omitted)

## Outcomes and Events

These two different dice distributions are closely related to each other. Let's see how.

In a multinomial distribution, each outcome alone is an event with its own chance (or probability). The chances of all outcomes sum to 1. The probability of any event is the sum of the probabilities of the outcomes in that event. Therefore, by writing down the chance of each outcome, we implicitly describe the probability of every event for the whole distribution.

Let's consider the event that the sum of the dice in two sequential rolls is 5. We can represent that event as the matching rows in the `two_dice` table.

```
dice_sums = two_dice.column('First') + two_dice.column('Second')
sum_of_5 = two_dice.where(dice_sums == 5)
sum_of_5
```

First	Second	Chance
1	4	2.8%
2	3	2.8%
3	2	2.8%
4	1	2.8%

The probability of the event is the sum of the resulting chances.

```
sum(sum_of_5.column('Chance'))
```

```
0.1111111111111111
```

If we consistently store the probabilities of individual outcomes in a column called `Chance`, then we can define the probability of any event using the `P` function.

```
def P(event):
    return sum(event.column('Chance'))

P(sum_of_5)
```

```
0.1111111111111111
```

One distribution's events can be another distribution's outcomes, as is the case with the `two_dice` and `two_dice_sum` distributions. There are 11 different possible sums in the `two_dice` table, and these are 11 different mutually exclusive events.

```
with_sums = two_dice.with_column('Sum', dice_sums)
with_sums
```

First	Second	Chance	Sum
1	1	2.8%	2
1	2	2.8%	3
1	3	2.8%	4
1	4	2.8%	5
1	5	2.8%	6
1	6	2.8%	7
2	1	2.8%	3
2	2	2.8%	4
2	3	2.8%	5
2	4	2.8%	6
3	1	2.8%	4
3	2	2.8%	5
3	3	2.8%	6
3	4	2.8%	7
4	1	2.8%	5
4	2	2.8%	6
4	3	2.8%	7
4	4	2.8%	8
5	1	2.8%	6
5	2	2.8%	7
5	3	2.8%	8
5	4	2.8%	9
6	1	2.8%	7
6	2	2.8%	8
6	3	2.8%	9
6	4	2.8%	10
7	1	2.8%	8
7	2	2.8%	9
7	3	2.8%	10
7	4	2.8%	11
8	1	2.8%	9
8	2	2.8%	10
8	3	2.8%	11
9	1	2.8%	10
9	2	2.8%	11
10	1	2.8%	11
11	1	2.8%	12

... (26 rows omitted)

Grouping these outcomes by their sum shows how many different outcomes appear in each event.

```
with_sums.groupby('Sum')
```

Sum	count
2	1
3	2
4	3
5	4
6	5
7	6
8	5
9	4
10	3
11	2

... (1 rows omitted)

Finally, the chance of each sum event is the total chance of all outcomes that match the sum event. In this way, we can generate the `two_dice_sums` distribution via addition.

```
grouped = with_sums.select(['Sum', 'Chance']).group('Sum', sum)
grouped.relabeled(1, 'Chance').set_format('Chance', PercentFormatter)
```

[1] ↗ ↘

Sum	Chance
2	2.8%
3	5.6%
4	8.3%
5	11.1%
6	13.9%
7	16.7%
8	13.9%
9	11.1%
10	8.3%
11	5.6%

... (1 rows omitted)

Both distributions will assign the same probability to certain events. For example, the probability of the event that the sum of the dice is above 8 can be computed from either distribution.

```
P(with_sums.where('Sum', 8))
```

```
0.1388888888888889
```

```
P(two_dice_sums.where('Sum', 8))
```

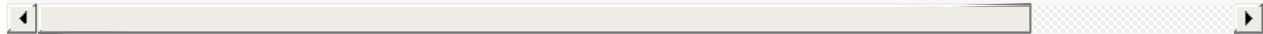
```
0.1388888888888889
```

## U.S. Birth Times

Here's an example based on data collected in the world.

More babies in the United States are born in the morning than the night. In 2013, the Center for Disease Control measured the following proportions of babies for each hour of the day. The `Time` column is a description of each hour, and the `Hour` column is its numeric equivalent.

```
birth = Table.read_table('birth_time.csv').select(['Time', 'Hour',  
birth.set_format('Chance', PercentFormatter(1)).show()
```



Time	Hour	Chance
6 a.m.	6	2.9%
7 a.m.	7	4.5%
8 a.m.	8	6.3%
9 a.m.	9	5.0%
10 a.m.	10	5.0%
11 a.m.	11	5.0%
Noon	12	6.0%
1 p.m.	13	5.7%
2 p.m.	14	5.1%
3 p.m.	15	4.8%
4 p.m.	16	4.9%
5 p.m.	17	5.0%
6 p.m.	18	4.5%
7 p.m.	19	4.0%
8 p.m.	20	4.0%
9 p.m.	21	3.7%
10 p.m.	22	3.5%
11 p.m.	23	3.3%
Midnight	0	2.9%
1 a.m.	1	2.9%
2 a.m.	2	2.8%
3 a.m.	3	2.7%
4 a.m.	4	2.7%
5 a.m.	5	2.8%

If we assume that this distribution describes the world correctly, what is the chance that a baby will be born between 8 a.m. and 6 p.m.? It turns out that more than half of all babies are born during "business hours", even though this time interval only covers 5/12 of the day.

```
business_hours = birth.where('Hour', are.between(8, 18))
business_hours
```

Time	Hour	Chance
8 a.m.	8	6.3%
9 a.m.	9	5.0%
10 a.m.	10	5.0%
11 a.m.	11	5.0%
Noon	12	6.0%
1 p.m.	13	5.7%
2 p.m.	14	5.1%
3 p.m.	15	4.8%
4 p.m.	16	4.9%
5 p.m.	17	5.0%

```
P(business_hours)
```

```
0.5280000000000002
```

On the other hand, births late at night are uncommon. The chance of having a baby between midnight and 6 a.m. is much less than 25%, even though this time interval covers 1/4 of the day.

```
P(birth.where('Hour', are.between(0, 6)))
```

```
0.1679999999999998
```

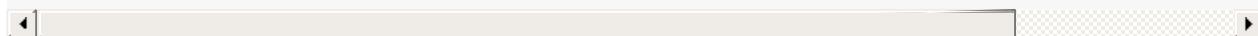
## Conditional Distributions

A common way to transform one distribution into another is to *condition* on an event.

Conditioning means assuming that the event occurs. The conditional distribution *given* an event takes the chances of all outcomes for which the event occurs and scales up their chances so that they sum to 1.

To scale up the chances, we divide the chance of each outcome in the event by the probability of the event. For example, if we condition on the event that the sum of two dice is above 8, we arrive at the following conditional distribution.

```
above_8 = two_dice_sums.where('Sum', are.above(8))
given_8 = above_8.with_column('Chance', above_8.column('Chance') / given_8)
```



Sum	Chance
9	40.0%
10	30.0%
11	20.0%
12	10.0%

A conditional distribution describes the chances under the original distribution, updated to reflect a new piece of information. In this case, we see the chances under the original `two_dice_sums` distribution, *given that* the sum is above 8.

The conditioning operation can be expressed by the `given` function, which takes an event table and returns the corresponding conditional distribution.

```
def given(event):
    return event.with_column('Chance', event.column('Chance') / P(e
given(two_dice_sums.where('Sum', are.above(8)))
```



Sum	Chance
9	40.0%
10	30.0%
11	20.0%
12	10.0%

Given that a baby is born in the US during business hours (between 8 a.m. and 6 p.m.), what are the chances that it is born before noon? To answer this question, we first form the conditional distribution given business hours, then compute the probability of the event that the baby is born before noon.

```
given(business_hours)
```

Time	Hour	Chance
8 a.m.	8	11.9%
9 a.m.	9	9.5%
10 a.m.	10	9.5%
11 a.m.	11	9.5%
Noon	12	11.4%
1 p.m.	13	10.8%
2 p.m.	14	9.7%
3 p.m.	15	9.1%
4 p.m.	16	9.3%
5 p.m.	17	9.5%

```
P(given(business_hours).where('Hour', are.below(12)))
```

```
0.40340909090909094
```

This result is not the same as the chance that a baby is born between 8 a.m. and noon in general.

```
morning = birth.where('Hour', are.between(8, 12))
P(morning)
```

```
0.21300000000000002
```

Nor is it the chance that a baby is born before noon in general.

```
P(birth.where('Hour', are.below(12)))
```

```
0.4550000000000007
```

Instead, it is the probability that both the event and the conditioning event are true (i.e., that the birth is in the morning), divided by the probability of the conditioning event (i.e., that the birth is during business hours).

```
P(morning) / P(business_hours)
```

```
0.40340909090909094
```

The `morning` event is a *joint* event that can be described as both during business hours and before noon. A joint event is just an event, but one that is described by the intersection of two other events.

```
business_hours.where('Hour', are.below(12))
```

Time	Hour	Chance
8 a.m.	8	6.3%
9 a.m.	9	5.0%
10 a.m.	10	5.0%
11 a.m.	11	5.0%

In general, the probability of an event B (e.g., before noon) conditioned on an event A (e.g., during business hours) is the probability of the joint event of A and B divided by the probability of the conditioning event A.

```
P(business_hours.where('Hour', are.below(12))) / P(business_hours)
```

```
0.40340909090909094
```

The standard notation for this relationship uses a vertical bar for conditioning and a comma for a joint event.

$$P(B|A) = \frac{P(A, B)}{P(A)}$$

## Joint Distributions

A joint event is one in which two different events both occur. Similarly, a joint outcome is an outcome that is composed of two different outcomes. For example, the outcomes of the `two_dice` distribution are each joint outcomes of the first and second dice rolls.

```
two_dice
```

First	Second	Chance
1	1	2.8%
1	2	2.8%
1	3	2.8%
1	4	2.8%
1	5	2.8%
1	6	2.8%
2	1	2.8%
2	2	2.8%
2	3	2.8%
2	4	2.8%
2	5	2.8%
2	6	2.8%
3	1	2.8%
3	2	2.8%
3	3	2.8%
3	4	2.8%
3	5	2.8%
3	6	2.8%
4	1	2.8%
4	2	2.8%
4	3	2.8%
4	4	2.8%
4	5	2.8%
4	6	2.8%
5	1	2.8%
5	2	2.8%
5	3	2.8%
5	4	2.8%
5	5	2.8%
5	6	2.8%
6	1	2.8%
6	2	2.8%
6	3	2.8%
6	4	2.8%
6	5	2.8%
6	6	2.8%

... (26 rows omitted)

Another common way to transform distributions is to form a joint distribution from conditional distributions. The definition of a conditional probability provides a formula for computing a joint probability, simply by multiplying both sides of the equation above by  $P(B)$ .

$$P(B|A) \times P(A) = P(A, B)$$

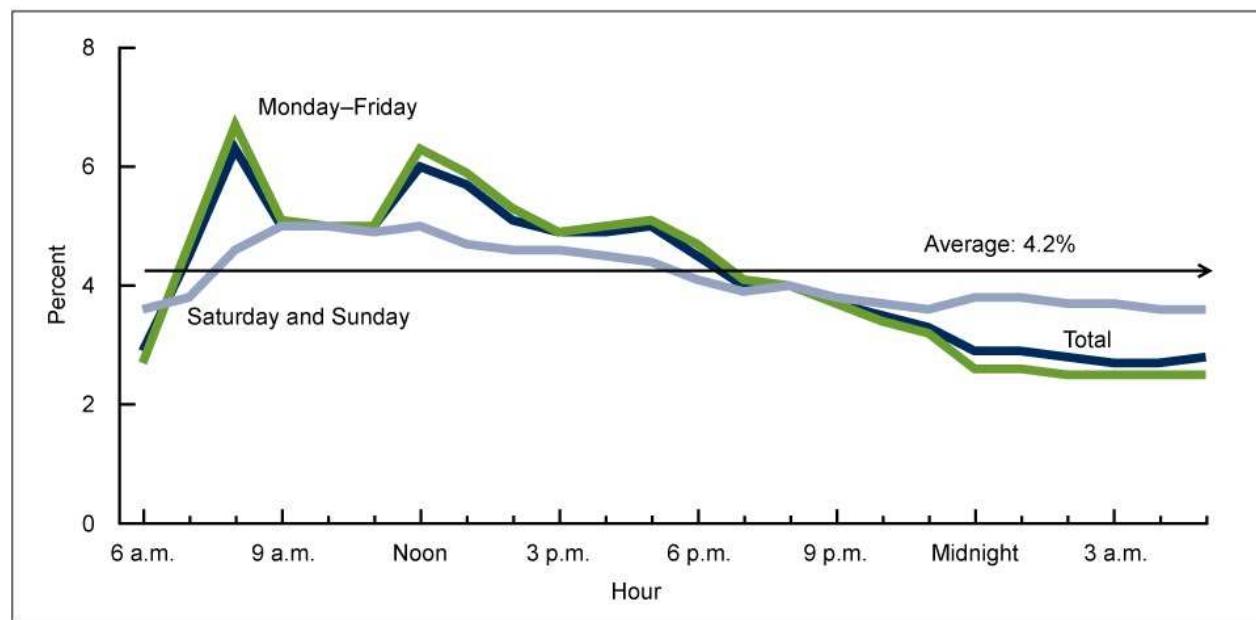
The chance of each outcome of a joint distribution can be computed from the probability of the first outcome and the conditional probability of the second outcome given the first outcome. In the `two_dice` table above, the chance of any first outcome is  $1/6$ , and the chance of any second outcome given any first outcome is  $1/6$ , so the chance of any joint outcome is  $1/36$  or 2.8%.

## U.S. Birth Days

The CDC also published that in 2013, 78.25% of babies born on weekdays (Monday through Friday) and 21.75% of babies were born on weekends (Saturday and Sunday). Notably, this distribution is not 5/7 (71.4%) to 2/7 (28.6%), but instead favors weekdays.

Moreover, the CDC published the conditional distributions of 2013 birth times, given a weekday birth (green) and given a weekend birth (gray). The black line is the distribution of all births.

Figure 1. Percent distribution of births, by hour and day of the week of delivery: 41 states and the District of Columbia, 2013



NOTES: The differences in the percent distributions are statistically significant. Access data table for Figure 1 at: [http://www.cdc.gov/nchs/data/databriefs/db\\_200\\_table.pdf#1](http://www.cdc.gov/nchs/data/databriefs/db_200_table.pdf#1).

SOURCE: CDC/NCHS, National Vital Statistics System.

All of the proportions used to generate this chart appear in the `birth_day` table.

```
birth_day = Table.read_table('birth_time.csv').drop('Chance')
birth_day.set_format([2, 3], PercentFormatter(1))
```

Time	Hour	Weekday	Weekend
6 a.m.	6	2.7%	3.6%
7 a.m.	7	4.7%	3.8%
8 a.m.	8	6.7%	4.6%
9 a.m.	9	5.1%	5.0%
10 a.m.	10	5.0%	5.0%
11 a.m.	11	5.0%	4.9%
Noon	12	6.3%	5.0%
1 p.m.	13	5.9%	4.7%
2 p.m.	14	5.3%	4.6%
3 p.m.	15	4.9%	4.6%
... (14 rows omitted)			

The `weekday` and `weekend` columns are the chances of two different conditional distributions.

```
weekday = birth_day.select(['Hour', 'Weekday']).relabeled(1, 'Chance')
weekend = birth_day.select(['Hour', 'Weekend']).relabeled(1, 'Chance')
```

The joint distribution is created by multiplying each conditional chance column by the corresponding probability of the conditioning event. In this case, we multiply weekday chances by the chance that the baby was born on a weekday (78.25%); likewise, we multiply weekend chances by the chance that the baby was born on a weekend (21.75%).

```
birth_joint = Table(['Day', 'Hour', 'Chance'])
for row in weekday.rows:
    birth_joint.append(['Weekday', row.item('Hour'), row.item('Chance') * .7825])
for row in weekend.rows:
    birth_joint.append(['Weekend', row.item('Hour'), row.item('Chance') * .2175])
birth_joint.set_format('Chance', PercentFormatter(1))
```

Day	Hour	Chance
Weekday	6	2.1%
Weekday	7	3.7%
Weekday	8	5.2%
Weekday	9	4.0%
Weekday	10	3.9%
Weekday	11	3.9%
Weekday	12	4.9%
Weekday	13	4.6%
Weekday	14	4.1%
Weekday	15	3.8%
...		
(38 rows omitted)		

This joint distribution has 48 joint outcomes, and the total chance sums to 1.

```
P(birth_joint)
```

```
1.0
```

Now, we can compute the probability of any event that involves days and hours. For example, the chance of a weekday morning birth is quite high.

```
P(birth_joint.where('Day', 'Weekday').where('Hour', are.between(8,
```

```
0.17058499999999999
```

We have seen that less than 22% of babies are born on weekends. However, if we know that a baby was born between 5 a.m. and 6 a.m., then the chances that it is a weekend baby are quite a bit higher than 22%.

```
early_morning = birth_joint.where('Hour', 5)
early_morning
```

Day	Hour	Chance
Weekday	5	2.0%
Weekend	5	0.8%

```
P(given(early_morning).where('Day', 'Weekend'))
```

```
0.28584466551063248
```

## Bayes' Rule

In the final example above, we began with a distribution over weekday and weekend births,  $P(A)$ , along with conditional distributions over hours,  $P(B|A)$ . We were able to compute the probability of a weekend birth conditioned on an hour, an outcome in the distribution  $P(A|B)$ . Bayes' rule is the formula that expresses the relationship between all of these quantities.

$$P(A|B) = \frac{P(A) \times P(B|A)}{P(B)}$$

The numerator on the right-hand side is just the joint probability of A and B. Bayes' rule writes this probability in its expanded form because so often we are given these two components and must form the joint distribution through multiplication.

The denominator on the right-hand side is an event that can be computed from the joint probability. For example, the `early_morning` event in the example above is an event just about hours, but it includes all joint outcomes where the correct hour occurs.

Each probability in this equation has a name. The names are derived from the most typical application of Bayes' rule, which is to update one's beliefs based on new evidence.

- $P(A)$  is called the *prior*; it is the probability of event A before any evidence is observed.
- $P(B|A)$  is called the *likelihood*; it is the conditional probability of the evidence event B given the event A.
- $P(B)$  is called the *evidence*; it is the probability of the evidence event B for any outcome.
- $P(A|B)$  is called the *posterior*; it is the probability of event A after evidence event B is observed.

Depending on the joint distribution of A and B, observing some B can make A more or less likely.

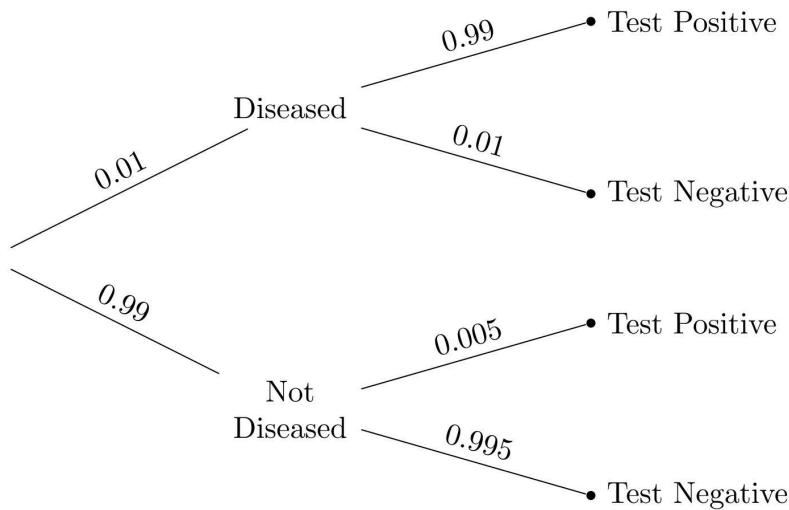
## Diagnostic Example

In a population, there is a rare disease. Researchers have developed a medical test for the disease. Mostly, the test correctly identifies whether or not the tested person has the disease. But sometimes, the test is wrong. Here are the relevant proportions.

- 1% of the population has the disease
- If a person has the disease, the test returns the correct result with chance 99%.
- If a person does not have the disease, the test returns the correct result with chance 99.5%.

**One person is picked at random from the population.** Given that the person tests positive, what is the chance that the person has the disease?

We begin by partitioning the population into four categories in the tree diagram below.



ANSWER

By Bayes' Rule, the chance that the person has the disease given that he or she has tested positive is the chance of the top "Test Positive" branch relative to the total chance of the two "Test Positive" branches. The answer is

$$\frac{0.01 \times 0.99}{0.01 \times 0.99 + 0.99 \times 0.005} = 0.667$$

```

# The person is picked at random from the population.

# By Bayes' Rule:
# Chance that the person has the disease, given that test was +
(0.01*0.99)/(0.01*0.99 + 0.99*0.005)
  
```

0.6666666666666666

This is 2/3, and it seems rather small. The test has very high accuracy, 99% or higher. Yet is our answer saying that if a patient gets tested and the test result is positive, there is only a 2/3 chance that he or she has the disease?

To understand our answer, it is important to recall the chance model: our calculation is valid for **a person picked at random from the population**. Among all the people in the population, the people who test positive split into two groups: those who have the disease and test positive, and those who don't have the disease and test positive. The latter group is called the group of *false positives*. The proportion of true positives is twice as high as that of

the false positives –  $0.01 \times 0.99$  compared to  $0.99 \times 0.005$  – and hence the chance of a true positive given a positive test result is  $2/3$ . The chance is affected both by the accuracy of the test and also by the probability that the sampled person has the disease.

The same result can be computed using a table. Below, we begin with the joint distribution over true health and test results.

```
rare = Table(['Health', 'Test', 'Chance']).with_rows([
    ['Diseased', 'Positive', 0.01 * 0.99],
    ['Diseased', 'Negative', 0.01 * 0.01],
    ['Not Diseased', 'Positive', 0.99 * 0.005],
    ['Not Diseased', 'Negative', 0.99 * 0.995]
])
rare
```

Health	Test	Chance
Diseased	Positive	0.0099
Diseased	Negative	0.0001
Not Diseased	Positive	0.00495
Not Diseased	Negative	0.98505

The chance that a person selected at random is diseased, given that they tested positive, is computed from the following expression.

```
positive = rare.where('Test', 'Positive')
P(given(positive).where('Health', 'Diseased'))
```

0.6666666666666663

But a patient who goes to get tested for a disease is not well modeled as a random member of the population. People get tested because they think they might have the disease, or because their doctor thinks so. In such a case, saying that their chance of having the disease is 1% is not justified; they are not picked at random from the population.

So, while our answer is correct for a random member of the population, it does not answer the question for a person who has walked into a doctor's office to be tested.

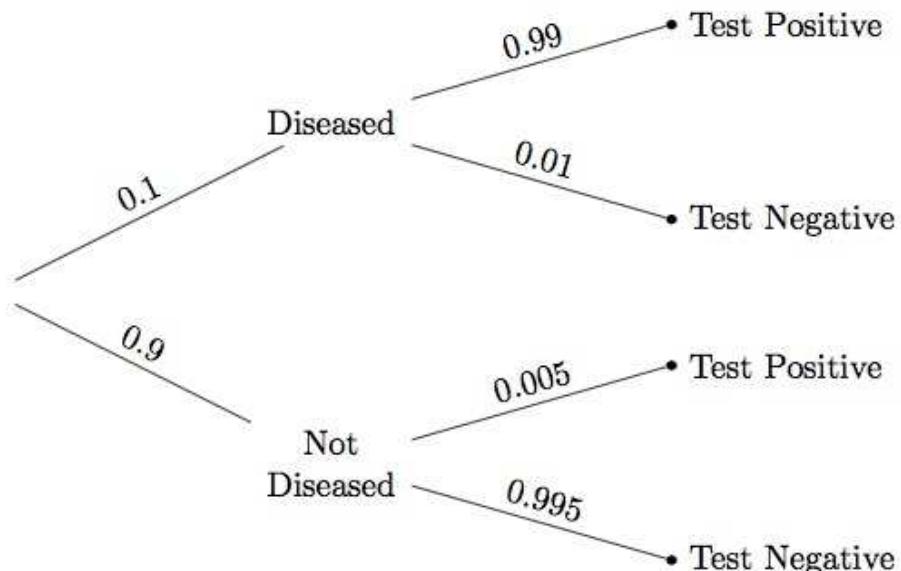
To answer the question for such a person, we must first ask ourselves what is the probability that the person has the disease. It is natural to think that it is larger than 1%, as the person has some reason to believe that he or she might have the disease. But how much larger?

This cannot be decided based on relative frequencies. The probability that a particular individual has the disease has to be based on a subjective opinion, and is therefore called a *subjective probability*. Some researchers insist that all probabilities must be relative frequencies, but subjective probabilities abound. The chance that a candidate wins the next election, the chance that a big earthquake will hit the Bay Area in the next decade, the chance that a particular country wins the next soccer World Cup: none of these are based on relative frequencies or long run frequencies. Each one contains a subjective element.

It is fine to work with subjective probabilities as long as you keep in mind that there will be a subjective element in your answer. Be aware also that different people can have different subjective probabilities of the same event. For example, the patient's subjective probability that he or she has the disease could be quite different from the doctor's subjective probability of the same event. Here we will work from the patient's point of view.

Suppose the patient assigned a number to his/her degree of uncertainty about whether he/she had the disease, *before* seeing the test result. This number is the patient's *subjective prior probability* of having the disease.

If that probability were 10%, then the probabilities on the left side of the tree diagram would change accordingly, with the 0.1 and 0.9 now interpreted as subjective probabilities:



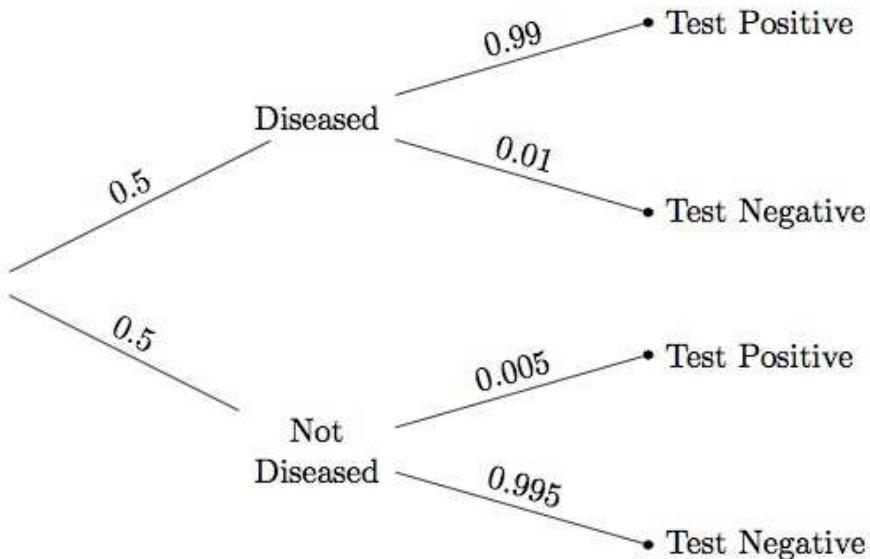
The change has a noticeable effect on the answer, as you can see by running the cell below.

```
# Subjective prior probability of 10% that the person has the disease
# By Bayes' Rule:
# Chance that the person has the disease, given that test was +
(0.1*0.99)/(0.1*0.99 + 0.9*0.005)
```

0.9565217391304347

If the patient's prior probability of having the disease is 10%, then after a positive test result the patient must update that probability to over 95%. This updated probability is called a *posterior* probability. It is calculated *after* learning the test result.

If the patient's prior probability of having the disease is 50%, then the result changes yet again.



```
# Subjective prior probability of 50% that the person has the disease
# By Bayes' Rule:
# Chance that the person has the disease, given that test was +
(0.5*0.99)/(0.5*0.99 + 0.5*0.005)
```

0.9949748743718593

Starting out with a 50-50 subjective chance of having the disease, the patient must update that chance to about 99.5% after getting a positive test result.

**Computational Note.** In the calculation above, the factor of 0.5 is common to all the terms and cancels out. Hence arithmetically it is the same as a calculation where the prior probabilities are apparently missing:

$0.99/(0.99 + 0.005)$

0.9949748743718593

But in fact, they are not missing. They have just canceled out. When the prior probabilities are not all equal, then they are all visible in the calculation as we have seen earlier.

In machine learning applications such as spam detection, Bayes' Rule is used to update probabilities of messages being spam, based on new messages being labeled Spam or Not Spam. You will need more advanced mathematics to carry out all the calculations. But the fundamental method is the same as what you have seen in this section.