

Table of Contents

Introduction	1.1
1. Data Science	1.2
1.1 Introduction	1.2.1
1.1.1 Computational Tools	1.2.1.1
1.1.2 Statistical Techniques	1.2.1.2
1.2 Why Data Science?	1.2.2
1.3 Plotting the Classics	1.2.3
1.3.1 Literary Characters	1.2.3.1
1.3.2 Another Kind of Character	1.2.3.2
2. Causality and Experiments	1.3
2.1 John Snow and the Broad Street Pump	1.3.1
2.2 Snow's "Grand Experiment"	1.3.2
2.3 Establishing Causality	1.3.3
2.4 Randomization	1.3.4
2.5 Endnote	1.3.5
3. Programming in Python	1.4
3.1 Expressions	1.4.1
3.2 Names	1.4.2
3.2.1 Example: Growth Rates	1.4.2.1
3.3 Call Expressions	1.4.3
3.4 Introduction to Tables	1.4.4
4. Data Types	1.5
4.1 Numbers	1.5.1
4.2 Strings	1.5.2
4.2.1 String Methods	1.5.2.1
4.3 Comparisons	1.5.3
5. Sequences	1.6
5.1 Arrays	1.6.1
5.2 Ranges	1.6.2
5.3 More on Arrays	1.6.3

6. Tables	1.7
6.1 Sorting Rows	1.7.1
6.2 Selecting Rows	1.7.2
6.3 Example: Population Trends	1.7.3
6.4 Example: Trends in Gender	1.7.4
7. Visualization	1.8
7.1 Categorical Distributions	1.8.1
7.2 Numerical Distributions	1.8.2
7.3 Overlaid Graphs	1.8.3
8. Functions and Tables	1.9
8.1 Applying Functions to Columns	1.9.1
8.2 Classifying by One Variable	1.9.2
8.3 Cross-Classifying	1.9.3
8.4 Joining Tables by Columns	1.9.4
8.5 Bike Sharing in the Bay Area	1.9.5
9. Randomness	1.10
9.1 Conditional Statements	1.10.1
9.2 Iteration	1.10.2
9.3 Simulation	1.10.3
9.4 The Monty Hall Problem	1.10.4
9.5 Finding Probabilities	1.10.5
10. Sampling and Empirical Distributions	1.11
10.1 Empirical Distributions	1.11.1
10.2 Sampling from a Population	1.11.2
10.3 Empirical Distribution of a Statistic	1.11.3
11. Testing Hypotheses	1.12
11.1 Assessing Models	1.12.1
11.2 Multiple Categories	1.12.2
11.3 Decisions and Uncertainty	1.12.3
12. Comparing Two Samples	1.13
12.1 A/B Testing	1.13.1
12.2 Deflategate	1.13.2
12.3 Causality	1.13.3
13. Estimation	1.14

13.1 Percentiles	1.14.1
13.2 The Bootstrap	1.14.2
13.3 Confidence Intervals	1.14.3
13.4 Using Confidence Intervals	1.14.4
14. Why the Mean Matters	1.15
14.1 Properties of the Mean	1.15.1
14.2 Variability	1.15.2
14.3 The SD and the Normal Curve	1.15.3
14.4 The Central Limit Theorem	1.15.4
14.5 The Variability of the Sample Mean	1.15.5
14.6 Choosing a Sample Size	1.15.6
15. Prediction	1.16
15.1 Correlation	1.16.1
15.2 The Regression Line	1.16.2
15.3 The Method of Least Squares	1.16.3
15.4 Least Squares Regression	1.16.4
15.5 Visual Diagnostics	1.16.5
15.6 Numerical Diagnostics	1.16.6
16. Inference for Regression	1.17
16.1 A Regression Model	1.17.1
16.2 Inference for the True Slope	1.17.2
16.3 Prediction Intervals	1.17.3
17. Classification	1.18
17.1 Nearest Neighbors	1.18.1
17.2 Training and Testing	1.18.2
17.3 Rows of Tables	1.18.3
17.4 Implementing the Classifier	1.18.4
17.5 The Accuracy of the Classifier	1.18.5
17.6 Multiple Regression	1.18.6
18. Updating Predictions	1.19
18.1 A "More Likely Than Not" Binary Classifier	1.19.1
18.2 Making Decisions	1.19.2

Computational and Inferential Thinking

The Foundations of Data Science

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

Contributions by [David Wagner](#) and [Henry Milner](#)

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

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

Old versions of this textbook: [Fall 2017](#)

The contents of this book are licensed for free consumption under the following license:

[Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International \(CC BY-NC-ND 4.0\)](#)

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 and descriptive statistics, 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 is 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. A 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, datahub.berkeley.edu. If you have a @berkeley.edu email address, you already have full access to the programming environment hosted on that site. If not, please [complete this form](#) to request access.

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 robust 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, machine learning, optimization, 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 effective inferential techniques.

The world has too many unanswered questions and difficult challenges to leave this critical reasoning to only a few specialists. All educated members of society 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.

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.

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. Later sections of the text will describe most of the features of the Python programming language used below.

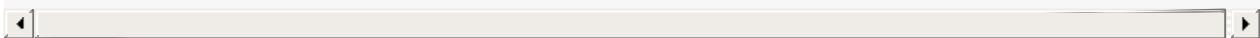
First, we read the text of both books into lists of chapters, called `huck_finn_chapters` and `little_women_chapters`. In Python, a name cannot contain any spaces, and so we will often use an underscore `_` to stand in for a space. The `=` in the lines below give a name on the left to the result of some 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. The `#` symbol starts a comment, which is ignored by the computer but helpful for people reading the code.

```
# Read two books, fast!

huck_finn_url =
'https://www.inferentialthinking.com/chapters/01/3/huck_finn.txt'

huck_finn_text = read_url(huck_finn_url)
huck_finn_chapters = huck_finn_text.split('CHAPTER ')[44:]

little_women_url =
'https://www.inferentialthinking.com/chapters/01/3/little_women.
txt'
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, it can 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 them into a table to see how each chapter begins.

```
# Display the chapters of Huckleberry Finn in a table.
```

```
Table().with_column('Chapters', huck_finn_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.

Literary Characters

Interact

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

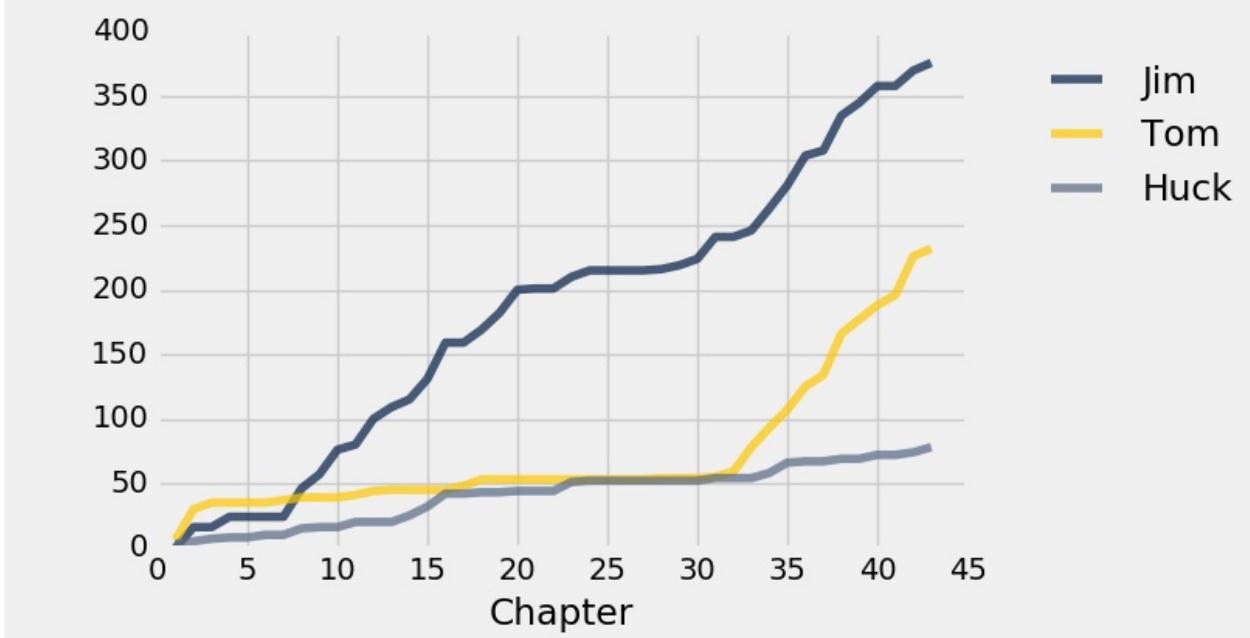
```
# Count how many times the names Jim, Tom, and Huck appear in
# each chapter.

counts = Table().with_columns([
    'Jim', np.char.count(huck_finn_chapters, 'Jim'),
    'Tom', np.char.count(huck_finn_chapters, 'Tom'),
    'Huck', np.char.count(huck_finn_chapters, 'Huck')
])

# Plot the cumulative counts:
# how many times in Chapter 1, how many times in Chapters 1 and
# 2, and so on.

cum_counts = counts.cumsum().with_column('Chapter', np.arange(1,
44, 1))
cum_counts.plot(column_for_xticks=3)
plots.title('Cumulative Number of Times Each Name Appears',
y=1.08);
```

Cumulative Number of Times Each Name Appears



In the plot above, the horizontal axis shows chapter numbers and the vertical axis shows how many times each character has been mentioned up to and including that chapter.

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 he arrives and joins Huck and Jim, 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.

```
# The chapters of Little Women, in a table
Table().with_column('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)

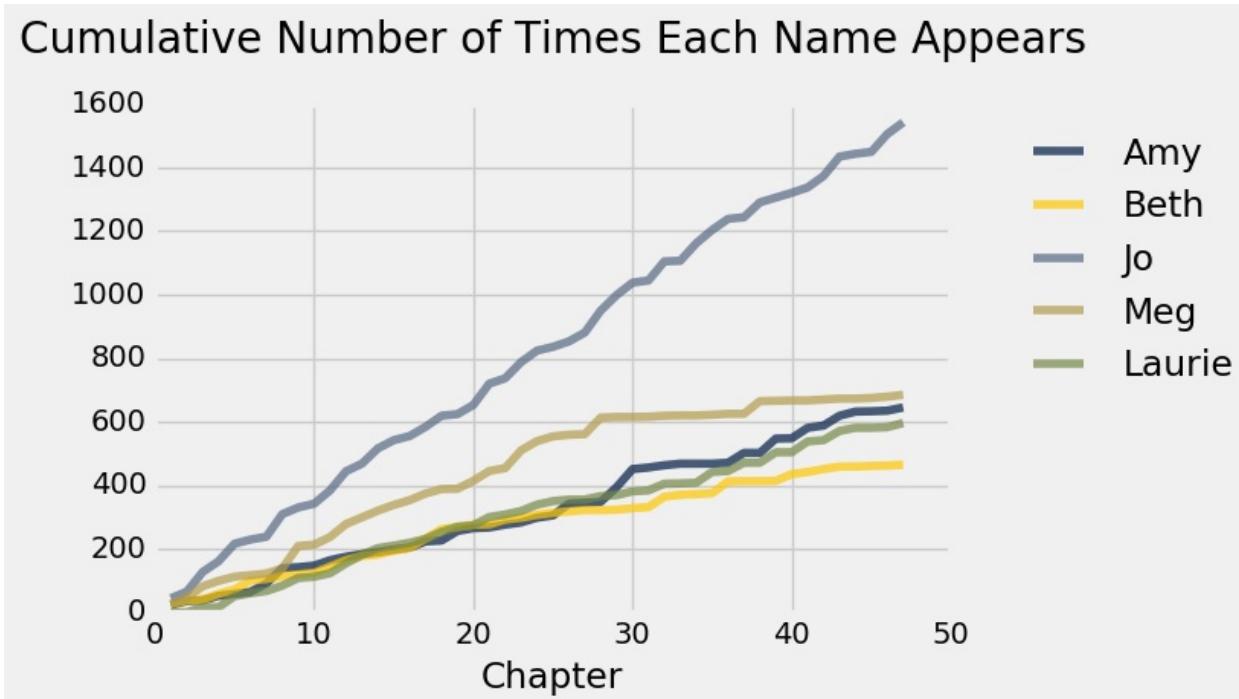
We can track the mentions of main characters to learn about the plot of this book as well. The protagonist Jo interacts with her sisters Meg, Beth, and Amy regularly, up until Chapter 27 when she moves to New York alone.

```
# Counts of names in the chapters of Little Women

counts = Table().with_columns([
    'Amy', np.char.count(little_women_chapters, 'Amy'),
    'Beth', np.char.count(little_women_chapters, 'Beth'),
    'Jo', np.char.count(little_women_chapters, 'Jo'),
    'Meg', np.char.count(little_women_chapters, 'Meg'),
    'Laurie', np.char.count(little_women_chapters,
    'Laurie'),])

# Plot the cumulative counts.

cum_counts = counts.cumsum().with_column('Chapter', np.arange(1,
48, 1))
cum_counts.plot(column_for_xticks=5)
plots.title('Cumulative Number of Times Each Name Appears',
y=1.08);
```



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.

Another Kind of Character

Interact

In some situations, 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 of visualizing information derived from multiple sources, let us first use the computer to get some information that would be tedious to acquire by hand. In the context of novels, the word "character" has a second meaning: a printed symbol such as a letter or number or punctuation symbol. Here, we ask the computer to count the number of characters and the number of periods in each chapter of both *Huckleberry Finn* and *Little Women*.

```
# In each chapter, count the number of all characters;
# call this the "length" of the chapter.
# Also count the number of periods.

chars_periods_huck_finn = Table().with_columns([
    'Huck Finn Chapter Length', [len(s) for s in
huck_finn_chapters],
    'Number of Periods', np.char.count(huck_finn_chapters,
'.'))
])

chars_periods_little_women = Table().with_columns([
    'Little Women Chapter Length', [len(s) for s in
little_women_chapters],
    'Number of Periods',
np.char.count(little_women_chapters, '.'))
])
```

Here are the data for *Huckleberry Finn*. Each row of the table 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.

1.3.2 Another Kind of Character

```
chars_periods_huck_finn
```

Huck Finn 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*.

```
chars_periods_little_women
```

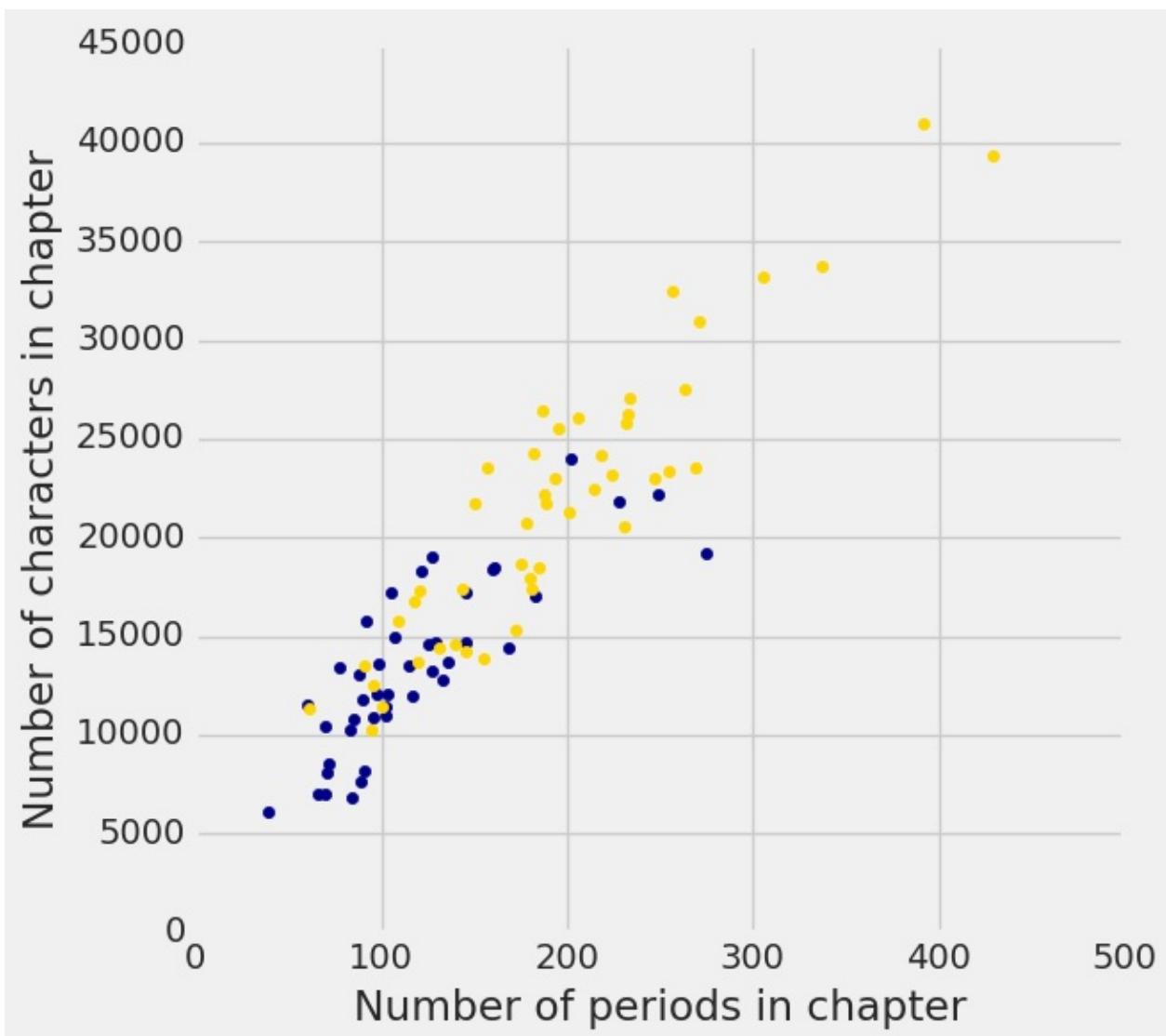
Little Women Chapter Length	Number of Periods
21759	189
22148	188
20558	231
25526	195
23395	255
14622	140
14431	131
22476	214
33767	337
18508	185

... (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 books. 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 gold dots to *Little Women*. The horizontal axis represents the number of periods and the vertical axis represents the number of characters.

```
plots.figure(figsize=(6, 6))
plots.scatter(chars_periods_huck_finn.column(1),
              chars_periods_huck_finn.column(0),
              color='darkblue')
plots.scatter(chars_periods_little_women.column(1),
              chars_periods_little_women.column(0),
              color='gold')
plots.xlabel('Number of periods in chapter')
plots.ylabel('Number of characters in chapter');
```



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.

Now look at all the chapters that contain about 100 periods. The plot shows that those chapters contain about 10,000 characters to about 15,000 characters, roughly. That's about 100 to 150 characters per period.

Indeed, it appears from looking at the plot that on average both books tend to have somewhere between 100 and 150 characters between periods, as a very rough estimate. Perhaps these two great 19th century novels were signaling something so very familiar to 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

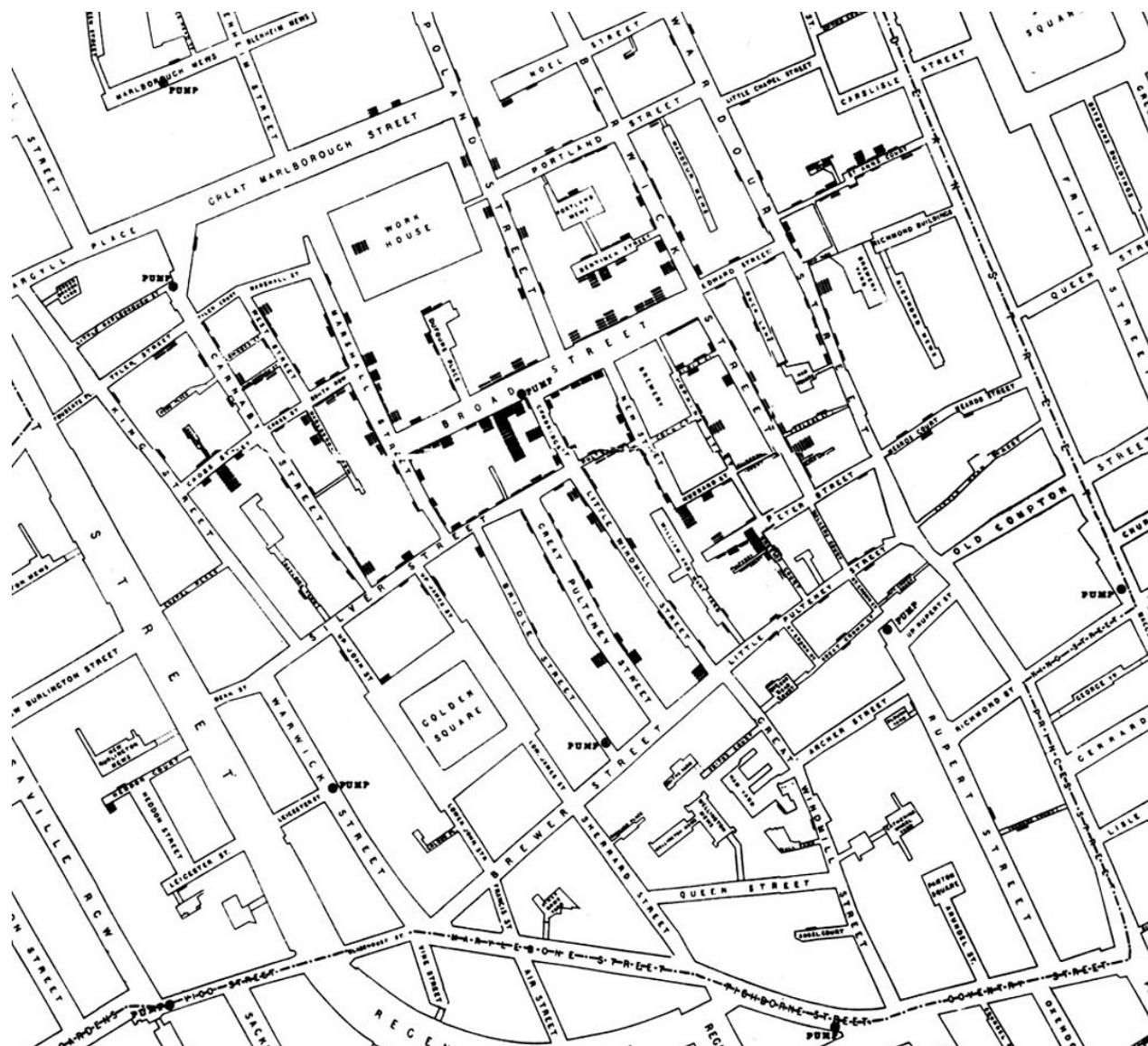
One of the earliest examples of astute observation eventually leading to the establishment of causality dates back more than 150 years. To get your mind into the right timeframe, try to imagine London in the 1850's. It was the world's wealthiest city but many of its people were desperately poor. Charles 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*.

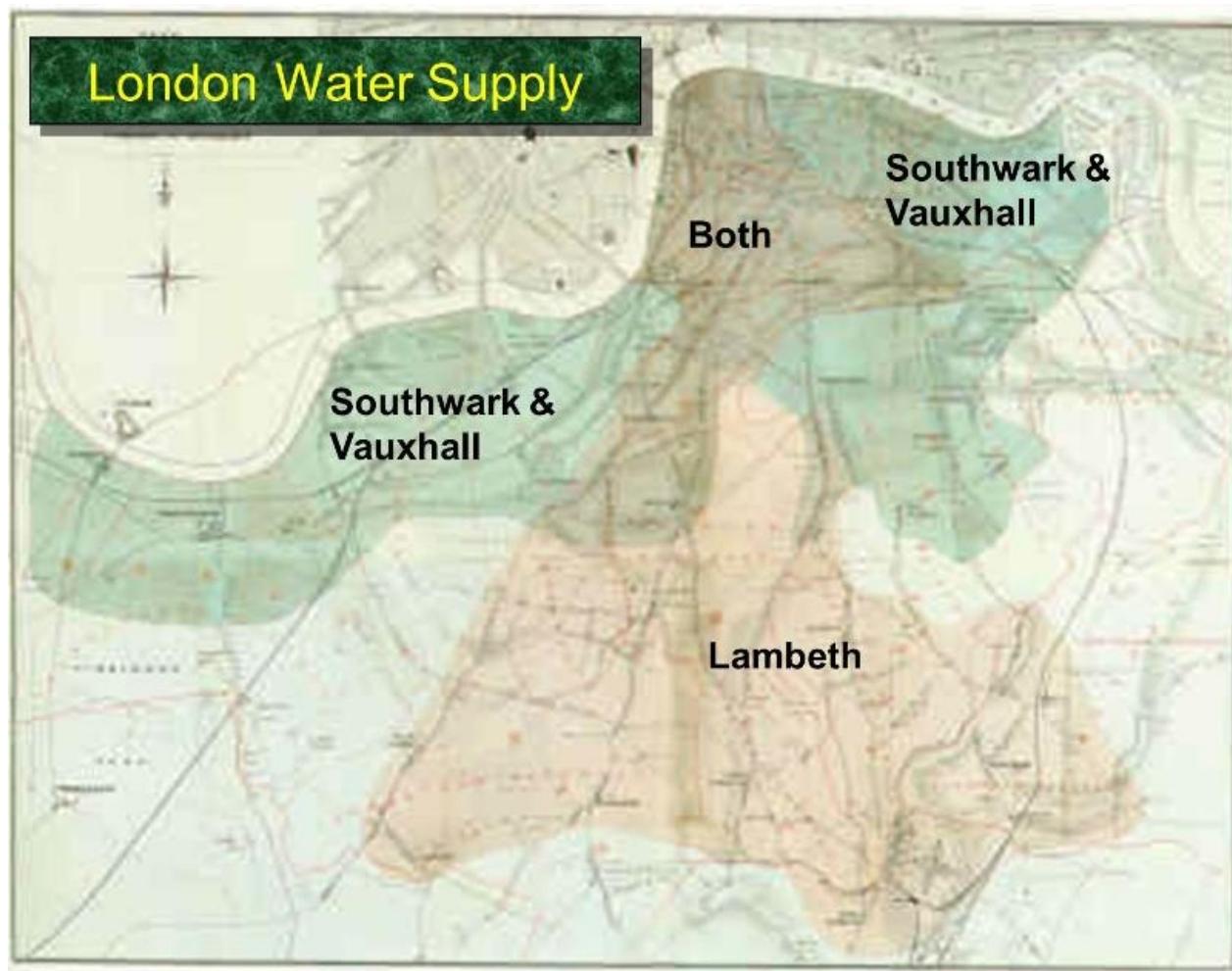
Scientists use comparison to identify an association between a treatment and an outcome. They compare the outcomes of a group of individuals who got the treatment (the *treatment group*) to the outcomes of a group who did not (the *control group*). For example, researchers today might compare the average 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.

The map below shows the areas served by the two companies. Snow honed in on the region where the two service areas overlap.



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 ..."

2.2 Snow's "Grand Experiment"

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.

Programming in Python

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.

Expressions

Interact

Programming languages are much simpler than human languages. Nonetheless, there are some rules of grammar to learn in any language, and that is where we will begin. In this text, we will use the [Python](#) programming language. Learning the grammar rules is essential, and the same rules used in the most basic programs are also central to more sophisticated programs.

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 grammar 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 an expression structure doesn't match any of the rules of the language.

```
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 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 - 9 + 10
```

```
17.555555555555557
```

```
1 + 2 * (3 * 4 * 5 / 6) ** 3 + 7 + 8 - 9 + 10
```

```
2017.0
```

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* statement. In an assignment, a name is followed by `=`, which is followed by any 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 = 1/4  
half = 2 * quarter  
half
```

0.5

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  
half
```

0.5

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. 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 also 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
```

3.2.1 Example: Growth Rates

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%).

3.2.1 Example: Growth Rates

```
1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 * 1.035 *  
1.035 * 1.035 - 1
```

```
0.410598760621121
```

This same computation can be expressed using names and exponents.

```
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.

In summary, a growth rate `g` is used to describe the relative size of an `initial` amount and a `changed` amount after some amount of time `t`. To compute **`changed`**, apply the growth rate `g` repeatedly, `t` times using exponentiation.

```
initial * (1 + g) ** t
```

To compute `g`, raise the total growth to the power of `1/t` and subtract one.

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

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 (<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
```

The list of [Python's built-in functions](#) is quite long and includes many functions that are never needed in data science applications. The list of [mathematical functions in the `math` module](#) is similarly long. This text will introduce the most important functions in context, rather than expecting the reader to memorize or understand these lists.

Introduction to Tables

Interact

We can now apply Python to analyze data. We will work with data stored in Table structures.

Tables are a fundamental way of representing data sets. A table can be viewed in two ways:

- a sequence of named columns that each describe a single attribute of all entries in a data set, or
- a sequence of rows that each contain all information about a single individual in a data set.

We will study tables in great detail in the next several chapters. For now, we will just introduce a few methods without going into technical details.

The table `cones` has been imported for us; later we will see how, but here we will just work with it. First, let's take a look at it.

cones		
Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25
bubblegum	pink	4.75

The table has six rows. Each row corresponds to one ice cream cone. The ice cream cones are the *individuals*.

Each cone has three attributes: flavor, color, and price. Each column contains the data on one of these attributes, and so all the entries of any single column are of the same kind. Each column has a label. We will refer to columns by their labels.

A table method is just like a function, but it must operate on a table. So the call looks like

```
name_of_table.method(arguments)
```

For example, if you want to see just the first two rows of a table, you can use the table method `show`.

```
cones.show(2)
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75

... (4 rows omitted)

You can replace 2 by any number of rows. If you ask for more than six, you will only get six, because `cones` only has six rows.

Choosing Sets of Columns

The method `select` creates a new table consisting of only the specified columns.

```
cones.select('Flavor')
```

Flavor
strawberry
chocolate
chocolate
strawberry
chocolate
bubblegum

This leaves the original table unchanged.

```
cones
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25
bubblegum	pink	4.75

You can select more than one column, by separating the column labels by commas.

```
cones.select('Flavor', 'Price')
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
chocolate	5.25
bubblegum	4.75

You can also *drop* columns you don't want. The table above can be created by dropping the `color` column.

```
cones.drop('Color')
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
chocolate	5.25
bubblegum	4.75

You can name this new table and look at it again by just typing its name.

```
no_colors = cones.drop('Color')

no_colors
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	5.25
strawberry	5.25
chocolate	5.25
bubblegum	4.75

Like `select`, the `drop` method creates a smaller table and leaves the original table unchanged. In order to explore your data, you can create any number of smaller tables by using choosing or dropping columns. It will do no harm to your original data table.

Sorting Rows

The `sort` method creates a new table by arranging the rows of the original table in ascending order of the values in the specified column. Here the `cones` table has been sorted in ascending order of the price of the cones.

```
cones.sort('Price')
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
bubblegum	pink	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25

To sort in descending order, you can use an *optional* argument to `sort`. As the name implies, optional arguments don't have to be used, but they can be used if you want to change the default behavior of a method.

By default, `sort` sorts in increasing order of the values in the specified column. To sort in decreasing order, use the optional argument `descending=True`.

```
cones.sort('Price', descending=True)
```

Flavor	Color	Price
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25
bubblegum	pink	4.75
chocolate	light brown	4.75
strawberry	pink	3.55

Like `select` and `drop`, the `sort` method leaves the original table unchanged.

Selecting Rows that Satisfy a Condition

The `where` method creates a new table consisting only of the rows that satisfy a given condition. In this section we will work with a very simple condition, which is that the value in a specified column must be equal to a value that we also specify. Thus the `where` method has two arguments.

The code in the cell below creates a table consisting only of the rows corresponding to chocolate cones.

```
cones.where('Flavor', 'chocolate')
```

Flavor	Color	Price
chocolate	light brown	4.75
chocolate	dark brown	5.25
chocolate	dark brown	5.25

The arguments, separated by a comma, are the label of the column and the value we are looking for in that column. The `where` method can also be used when the condition that the rows must satisfy is more complicated. In those situations the call will be a little more complicated as well.

It is important to provide the value exactly. For example, if we specify `Chocolate` instead of `chocolate`, then `where` correctly finds no rows where the flavor is `Chocolate`.

```
cones.where('Flavor', 'Chocolate')
```

Flavor	Color	Price
--------	-------	-------

Like all the other table methods in this section, `where` leaves the original table unchanged.

Example: Salaries in the NBA

"The NBA is the highest paying professional sports league in the world," reported CNN in March 2016. The table `nba` contains the salaries of all National Basketball Association players in 2015-2016.

Each row represents one player. The columns are:

Column Label	Description
PLAYER	Player's name
POSITION	Player's position on team
TEAM	Team name
SALARY	Player's salary in 2015-2016, in millions of dollars

The code for the positions is PG (Point Guard), SG (Shooting Guard), PF (Power Forward), SF (Small Forward), and C (Center). But what follows doesn't involve details about how basketball is played.

The first row shows that Paul Millsap, Power Forward for the Atlanta Hawks, had a salary of almost **\$18.7** million in 2015-2016.

```
nba
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

... (407 rows omitted)

Fans of Stephen Curry can find his row by using `where`.

```
nba.where('PLAYER', 'Stephen Curry')
```

PLAYER	POSITION	TEAM	SALARY
Stephen Curry	PG	Golden State Warriors	11.3708

We can also create a new table called `warriors` consisting of just the data for the Golden State Warriors.

```
warriors = nba.where('TEAM', 'Golden State Warriors')
warriors
```

PLAYER	POSITION	TEAM	SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5

... (4 rows omitted)

By default, the first 10 lines of a table are displayed. You can use `show` to display more or fewer. To display the entire table, use `show` with no argument in the parentheses.

```
warriors.show()
```

PLAYER	POSITION	TEAM	SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5
Festus Ezeli	C	Golden State Warriors	2.00875
Brandon Rush	SF	Golden State Warriors	1.27096
Kevon Looney	SF	Golden State Warriors	1.13196
Anderson Varejao	PF	Golden State Warriors	0.289755

The `nba` table is sorted in alphabetical order of the team names. To see how the players were paid in 2015-2016, it is useful to sort the data by salary. Remember that by default, the sorting is in increasing order.

```
nba.sort('SALARY')
```

PLAYER	POSITION	TEAM	SALARY
Thanasis Antetokounmpo	SF	New York Knicks	0.030888
Jordan McRae	SG	Phoenix Suns	0.049709
Cory Jefferson	PF	Phoenix Suns	0.049709
Elliot Williams	SG	Memphis Grizzlies	0.055722
Orlando Johnson	SG	Phoenix Suns	0.055722
Phil Pressey	PG	Phoenix Suns	0.055722
Keith Appling	PG	Orlando Magic	0.061776
Sean Kilpatrick	SG	Denver Nuggets	0.099418
Erick Green	PG	Utah Jazz	0.099418
Jeff Ayres	PF	Los Angeles Clippers	0.111444

... (407 rows omitted)

These figures are somewhat difficult to compare as some of these players changed teams during the season and received salaries from more than one team; only the salary from the last team appears in the table.

The CNN report is about the other end of the salary scale – the players who are among the highest paid in the world. To identify these players we can sort in descending order of salary and look at the top few rows.

```
nba.sort('SALARY', descending=True)
```

PLAYER	POSITION	TEAM	SALARY
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.8949
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.3594
Chris Bosh	PF	Miami Heat	22.1927
Chris Paul	PG	Los Angeles Clippers	21.4687
Kevin Durant	SF	Oklahoma City Thunder	20.1586
Derrick Rose	PG	Chicago Bulls	20.0931
Dwyane Wade	SG	Miami Heat	20

... (407 rows omitted)

Kobe Bryant, since retired, was the highest earning NBA player in 2015-2016.

Data Types

Interact

Every value has a type, and the built-in `type` function returns the type of the result of any expression.

One type we have encountered already is a built-in function. Python indicates that the type is a `builtin_function_or_method`; the distinction between a *function* and a *method* is not important at this stage.

```
type(abs)
```

```
builtin_function_or_method
```

This chapter will explore many useful types of data.

Numbers

Interact

Computers are designed to perform numerical calculations, but there are some important details about working with numbers that every programmer working with quantitative data should know. Python (and most other programming languages) distinguishes between two different types of numbers:

- Integers are called `int` values in the Python language. They can only represent whole numbers (negative, zero, or positive) that don't have a fractional component
- Real numbers are called `float` values (or *floating point values*) in the Python language. They can represent whole or fractional numbers but have some limitations.

The type of a number is evident from the way it is displayed: `int` values have no decimal point and `float` values always have a decimal point.

```
# Some int values  
2
```

```
2
```

```
1 + 3
```

```
4
```

```
-12345678900000000000
```

```
-12345678900000000000
```

```
# Some float values  
1.2
```

```
1.2
```

```
3.0
```

```
3.0
```

When a `float` value is combined with an `int` value using some arithmetic operator, then the result is always a `float` value. In most cases, two integers combine to form another integer, but any number (`int` or `float`) divided by another will be a `float` value. Very large or very small `float` values are displayed using scientific notation.

```
1.5 + 2
```

```
3.5
```

```
3 / 1
```

```
3.0
```

```
-12345678900000000000.0
```

```
-1.23456789e+19
```

The `type` function can be used to find the type of any number.

```
type(3)
```

```
int
```

```
type(3 / 1)
```

```
float
```

The `type` of an expression is the type of its final value. So, the `type` function will never indicate that the type of an expression is a name, because names are always evaluated to their assigned values.

```
x = 3  
type(x) # The type of x is an int, not a name
```

```
int
```

```
type(x + 2.5)
```

```
float
```

More About Float Values

Float values are very flexible, but they do have limits.

1. A `float` can represent extremely large and extremely small numbers. There are limits, but you will rarely encounter them.
2. A `float` only represents 15 or 16 significant digits for any number; the remaining precision is lost. This limited precision is enough for the vast majority of applications.
3. After combining `float` values with arithmetic, the last few digits may be incorrect. Small rounding errors are often confusing when first encountered.

The first limit can be observed in two ways. If the result of a computation is a very large number, then it is represented as infinite. If the result is a very small number, then it is represented as zero.

```
2e306 * 10
```

```
2e+307
```

```
2e306 * 100
```

```
inf
```

```
2e-322 / 10
```

```
2e-323
```

```
2e-322 / 100
```

```
0.0
```

The second limit can be observed by an expression that involves numbers with more than 15 significant digits. These extra digits are discarded before any arithmetic is carried out.

```
0.666666666666666 - 0.666666666666666123456789
```

```
0.0
```

The third limit can be observed when taking the difference between two expressions that should be equivalent. For example, the expression `2 ** 0.5` computes the square root of 2, but squaring this value does not exactly recover 2.

```
2 ** 0.5
```

```
1.4142135623730951
```

```
(2 ** 0.5) * (2 ** 0.5)
```

```
2.0000000000000004
```

```
(2 ** 0.5) * (2 ** 0.5) - 2
```

```
4.440892098500626e-16
```

The final result above is `0.00000000000004440892098500626`, a number that is very close to zero. The correct answer to this arithmetic expression is 0, but a small error in the final significant digit appears very different in scientific notation. This behavior appears in almost all programming languages because it is the result of the standard way that arithmetic is carried out on computers.

Although `float` values are not always exact, they are certainly reliable and work the same way across all different kinds of computers and programming languages.

Strings

Interact

Much of the world's data is text, and a piece of text represented in a computer is called a *string*. A string can represent a word, a sentence, or even the contents of every book in a library. Since text can include numbers (like this: 5) or truth values (True), a string can also describe those things.

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" + " " + "science"
```

```
'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

Interact

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'
```

Note that the line `t = s.replace('t', 'ing')` doesn't change the string `s`, which is still "train". The method call `s.replace('t', 'ing')` just has a value, which is the string "ingrain".

```
s
```

```
'train'
```

This is the first time we've seen methods, but methods are not unique to strings. As we will see shortly, other types of objects can have them.

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><</code>	<code>2 < 3</code>	<code>2 < 2</code>
Greater than	<code>></code>	<code>3 > 2</code>	<code>3 > 3</code>
Less than or equal	<code><=</code>	<code>2 <= 2</code>	<code>3 <= 2</code>
Greater or equal	<code>>=</code>	<code>3 >= 3</code>	<code>2 >= 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. By grouping values together, we can write code that performs a computation on many pieces of data at once.

Calling the function `make_array` on several values places them into an *array*, which is a kind of sequential collection. Below, we collect four different temperatures into an array called `highs`. 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.

```
baseline_high = 14.48
highs = make_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])
```

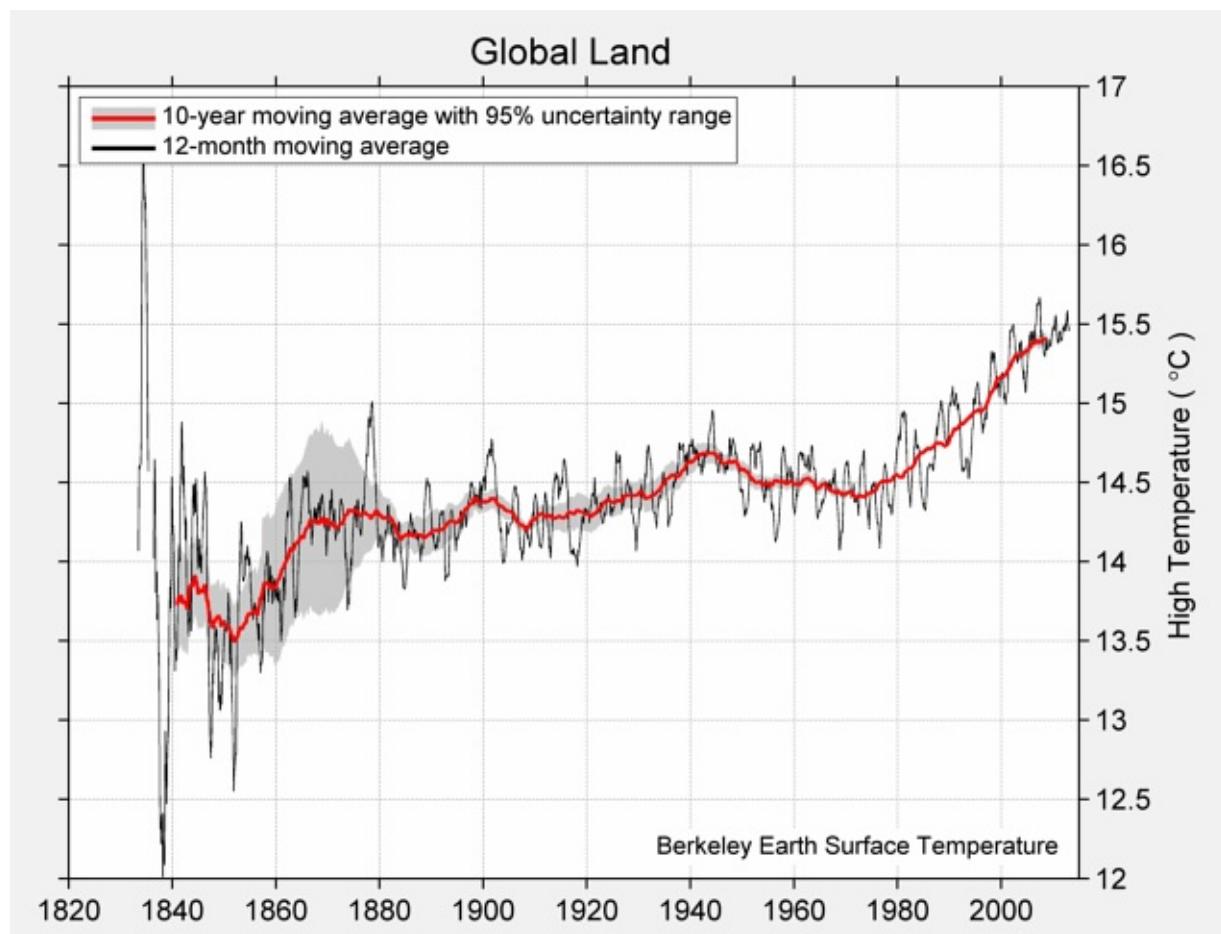
Collections allow us to pass multiple values into a function using a single name. For instance, the `sum` function computes the sum of all values in a collection, and the `len` function computes its length. (That's the number of values we put in it.) Using them together, we can compute the average of a collection.

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

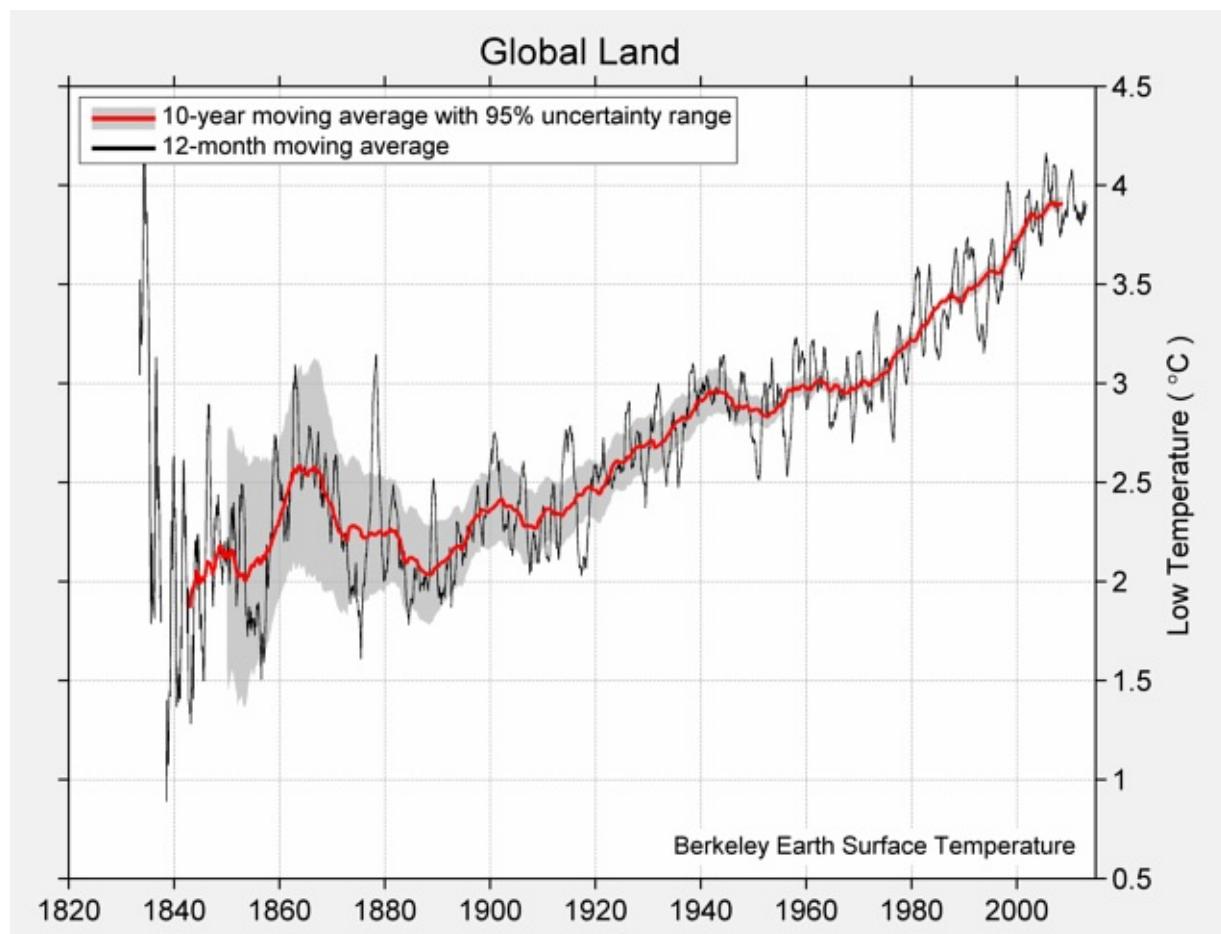
```
14.434000000000001
```

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

Mean of Daily High Temperature



Mean of Daily Low Temperature [¶](#)



Arrays

Interact

While there are many kinds of collections in Python, we will work primarily with arrays in this class. We've already seen that the `make_array` function can be used to create arrays of numbers.

Arrays can also contain strings or other types of values, but a single array can only contain a single kind of data. (It usually doesn't make sense to group together unlike data anyway.)

For example:

```
english_parts_of_speech = make_array("noun", "pronoun", "verb",
"adverb", "adjective", "conjunction", "preposition",
"interjection")
english_parts_of_speech
```

```
array(['noun', 'pronoun', 'verb', 'adverb', 'adjective',
'conjunction',
'preposition', 'interjection'],
dtype='<U12')
```

Returning to the temperature data, we create arrays of average daily [high temperatures](#) for the decades surrounding 1850, 1900, 1950, and 2000.

```
baseline_high = 14.48
highs = make_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])
```

Arrays 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 Fahrenheit by writing the familiar

conversion formula.

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

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

$$\text{highs} \\ (9/5) * \begin{matrix} 13.6 \\ 14.387 \\ 14.585 \\ 15.164 \end{matrix} + 32 = \begin{matrix} (9/5) * 13.6 + 32 \\ (9/5) * 14.387 + 32 \\ (9/5) * 14.585 + 32 \\ (9/5) * 15.164 + 32 \end{matrix} = \begin{matrix} 56.48 \\ 57.8966 \\ 58.253 \\ 59.2952 \end{matrix}$$

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 examples below is part of a call expression; it's calling a function with no arguments to perform a computation on the array called `highs`.

```
highs.size
```

```
4
```

```
highs.sum()
```

```
57.736000000000004
```

```
highs.mean()
```

```
14.434000000000001
```

Functions on Arrays

The `numpy` package, abbreviated `np` in programs, provides Python programmers with convenient and powerful functions for creating and manipulating arrays.

```
import numpy as np
```

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.

However, you **don't need to memorize these**. Use this as a reference.

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 useful in a surprisingly large number of situations, so it's worthwhile to learn about them.

Ranges are defined using the `np.arange` function, which takes either one, two, or three arguments: a start, and end, and a 'step'.

If you pass one argument to `np.arange`, this becomes the `end` value, with `start=0`, `step=1` assumed. Two arguments give the `start` and `end` with `step=1` assumed. Three arguments give the `start`, `end` and `step` explicitly.

A range always includes its `start` value, but does not include its `end` value. It counts up by `step`, and it stops before it gets to the `end`.

```
np.arange(end): An array starting with 0 of increasing consecutive integers, stopping before end.
```

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

Notice how the array starts at 0 and goes only up to 4, not to the end value of 5.

```
np.arange(start, end): An array of consecutive increasing integers from start, stopping before end.
```

```
np.arange(3, 9)
```

```
array([3, 4, 5, 6, 7, 8])
```

```
np.arange(start, end, step): A range with a difference of step between each pair of consecutive values, starting from start and stopping before end.
```

```
np.arange(3, 30, 5)
```

```
array([ 3,  8, 13, 18, 23, 28])
```

This array starts at 3, then takes a step of 5 to get to 8, then another step of 5 to get to 13, and so on.

When you specify a step, the start, end, and step can all be either positive or negative and may be whole numbers or fractions.

```
np.arange(1.5, -2, -0.5)
```

```
array([ 1.5,  1. ,  0.5,  0. , -0.5, -1. , -1.5])
```

Example: Leibniz's formula for π

The great German mathematician and philosopher [Gottfried Wilhelm Leibniz](#) (1646 - 1716) discovered a wonderful formula for π as an infinite sum of simple fractions. The formula is

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

Though some math is needed to establish this, we can use arrays to convince ourselves that the formula works. Let's calculate the first 5000 terms of Leibniz's infinite sum and see if it is close to π .

$$4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots - \frac{1}{9999} \right)$$

We will calculate this finite sum by adding all the positive terms first and then subtracting the sum of all the negative terms [1]:

$$\begin{aligned} & 4 \\ & \cdot \left(\left(1 + \frac{1}{5} + \frac{1}{9} + \dots + \frac{1}{9997} \right) \right. \\ & \left. - \left(\frac{1}{3} + \frac{1}{7} + \frac{1}{11} + \dots + \frac{1}{9999} \right) \right) \end{aligned}$$

The positive terms in the sum have 1, 5, 9, and so on in the denominators. The array `by_four_to_20` contains these numbers up to 17:

```
by_four_to_20 = np.arange(1, 20, 4)
by_four_to_20
```

```
array([ 1,  5,  9, 13, 17])
```

To get an accurate approximation to π , we'll use the much longer array `positive_term_denominators`.

```
positive_term_denominators = np.arange(1, 10000, 4)
positive_term_denominators
```

```
array([ 1,      5,      9, ..., 9989, 9993, 9997])
```

The positive terms we actually want to add together are just 1 over these denominators:

```
positive_terms = 1 / positive_term_denominators
```

The negative terms have 3, 7, 11, and so on on in their denominators. This array is just 2 added to `positive_term_denominators`.

```
negative_terms = 1 / (positive_term_denominators + 2)
```

The overall sum is

```
4 * ( sum(positive_terms) - sum(negative_terms) )
```

```
3.1413926535917955
```

This is very close to $\pi = 3.14159\dots$. Leibniz's formula is looking good!

Footnotes 

[1] Surprisingly, when we add *infinitely* many fractions, the order can matter! But our approximation to π uses only a large finite number of fractions, so it's okay to add the terms in any convenient order.

[Interact](#)

More on Arrays

It's often necessary to compute something that involves data from more than one array. If two arrays are of the same size, Python makes it easy to do calculations involving both arrays.

For our first example, we return once more to the temperature data. This time, we create arrays of average daily `high` and `low` temperatures for the decades surrounding 1850, 1900, 1950, and 2000.

```
baseline_high = 14.48
highs = make_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 = make_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])
```

Suppose we'd like to compute the average daily *range* of temperatures for each decade. That is, we want to subtract the average daily high in the 1850s from the average daily low in the 1850s, and the same for each other decade.

We could write this laboriously using `.item`:

```
make_array(
    highs.item(0) - lows.item(0),
    highs.item(1) - lows.item(1),
    highs.item(2) - lows.item(2),
    highs.item(3) - lows.item(3)
)
```

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

As when we converted an array of temperatures from Celsius to Fahrenheit, Python provides a much cleaner way to write this:

```
highs - lows
```

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

highs lows

13.6
14.387
14.585
15.164

-

2.128
2.371
2.874
3.728

=

13.6 - 2.128
14.387 - 2.371
14.585 - 2.874
15.164 - 3.728

=

11.472
12.016
11.711
11.436

What we've seen in these examples are special cases of a general feature of arrays.

Elementwise arithmetic on pairs of numerical arrays

If an arithmetic operator acts on two arrays of the same size, then the operation is performed on each corresponding pair of elements in the two arrays. The final result is an array.

For example, if `array1` and `array2` have the same number of elements, then the value of `array1 * array2` is an array. Its first element is the first element of `array1` times the first element of `array2`, its second element is the second element of `array1` times the second element of `array2`, and so on.

Example: Wallis' Formula for π

The number π is important in many different areas of math. Centuries before computers were invented, mathematicians worked on finding simple ways to approximate the numerical value of π . We have already seen Leibniz's formula for π . About half a century before Leibniz, the English mathematician [John Wallis](#) (1616-1703) also expressed π in terms of simple fractions, as an infinite product.

$$\pi = 2 \cdot \left(\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \dots \right)$$

This is a product of "even/odd" fractions. Let's use arrays to multiply a million of them, and see if the product is close to π .

Remember that multiplication can done in any order [\[1\]](#), so we can readjust our calculation to:

$$\pi \approx 2 \cdot \left(\frac{2}{1} \cdot \frac{4}{3} \cdot \frac{6}{5} \dots \frac{1,000,000}{999999} \right) \cdot \left(\frac{2}{3} \cdot \frac{4}{5} \cdot \frac{6}{7} \dots \frac{1,000,000}{1,000,001} \right)$$

We're now ready to do the calculation. We start by creating an array of even numbers 2, 4, 6, and so on upto 1,000,000. Then we create two lists of odd numbers: 1, 3, 5, 7, ... upto 999,999, and 3, 5, 7, ... upto 1,000,001.

```
even = np.arange(2, 1000001, 2)
one_below_even = even - 1
one_above_even = even + 1
```

Remember that `np.prod` multiplies all the elements of an array together. Now we can calculate Wallis' product, to a good approximation.

```
2 * np.prod(even/one_below_even) * np.prod(even/one_above_even)
```

```
3.1415910827951143
```

That's π correct to five decimal places. Wallis clearly came up with a great formula.

Footnotes

[1] As we saw in the example about Leibniz's formula, when we add *infinitely* many fractions, the order can matter. The same is true with multiplying fractions, as we are doing here. But our approximation to π uses only a large finite number of fractions, so it's okay to multiply the terms in any convenient order.

Tables

Interact

Tables are a fundamental object type for representing data sets. A table can be viewed in two ways:

- a sequence of named columns that each describe a single aspect of all entries in a data set, or
- 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. An empty table is usefully because it can be extended to contain new rows and columns.

```
Table()
```

The `with_columns` method on a table constructs a new table with additional labeled columns. Each column of a table is an array. To add one new column to a table, call `with_columns` with a label and an array. (The `with_column` method can be used with the same effect.)

Below, we begin each example with an empty table that has no columns.

```
Table().with_columns('Number of petals', make_array(8, 34, 5))
```

Number of petals
8
34
5

To add two (or more) new columns, provide the label and array for each column. All columns must have the same length, or an error will occur.

```
Table().with_columns(
    'Number of petals', make_array(8, 34, 5),
    'Name', make_array('lotus', 'sunflower', 'rose')
)
```

Number of petals	Name
8	lotus
34	sunflower
5	rose

We can give this table a name, and then extend the table with another column.

```
flowers = Table().with_columns(
    'Number of petals', make_array(8, 34, 5),
    'Name', make_array('lotus', 'sunflower', 'rose')
)

flowers.with_columns(
    'Color', make_array('pink', 'yellow', 'red')
)
```

Number of petals	Name	Color
8	lotus	pink
34	sunflower	yellow
5	rose	red

The `with_columns` method creates a new table each time it is called, so the original table is not affected. For example, the table `flowers` still has only the two columns that it had when it was created.

```
flowers
```

Number of petals	Name
8	lotus
34	sunflower
5	rose

Creating tables in this way involves a lot of typing. If the data have already been entered somewhere, it is usually possible to use Python to read it into a table, instead of typing it all in cell by cell.

Often, tables are created from files that contain comma-separated values. Such files are called CSV files.

Below, we use the Table method `read_table` to read a CSV file that contains some of the data used by Minard in his graphic about Napoleon's Russian campaign. The data are placed in a table named `minard`.

```
minard = Table.read_table('minard.csv')
minard
```

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moidexno	Retreat	12000

We will use this small table to demonstrate some useful Table methods. We will then use those same methods, and develop other methods, on much larger tables of data.

The Size of the Table

The method `num_columns` gives the number of columns in the table, and `num_rows` the number of rows.

```
minard.num_columns
```

5

```
minard.num_rows
```

```
8
```

Column Labels

The method `labels` can be used to list the labels of all the columns. With `minard` we don't gain much by this, but it can be very useful for tables that are so large that not all columns are visible on the screen.

```
minard.labels
```

```
('Longitude', 'Latitude', 'City', 'Direction', 'Survivors')
```

We can change column labels using the `relabelled` method. This creates a new table and leaves `minard` unchanged.

```
minard.relabelled('City', 'City Name')
```

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moidexno	Retreat	12000

However, this method does not change the original table.

```
minard
```

Longitude	Latitude	City	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moidexno	Retreat	12000

A common pattern is to assign the original name `minard` to the new table, so that all future uses of `minard` will refer to the relabeled table.

```
minard = minard.relabel('City', 'City Name')
minard
```

Longitude	Latitude	City Name	Direction	Survivors
32	54.8	Smolensk	Advance	145000
33.2	54.9	Dorogobouge	Advance	140000
34.4	55.5	Chjat	Advance	127100
37.6	55.8	Moscou	Advance	100000
34.3	55.2	Wixma	Retreat	55000
32	54.6	Smolensk	Retreat	24000
30.4	54.4	Orscha	Retreat	20000
26.8	54.3	Moidexno	Retreat	12000

Accessing the Data in a Column

We can use a column's label to access the array of data in the column.

```
minard.column('Survivors')
```

```
array([145000, 140000, 127100, 100000, 55000, 24000, 20000,
12000])
```

The 5 columns are indexed 0, 1, 2, 3, and 4. The column `Survivors` can also be accessed by using its column index.

```
minard.column(4)
```

```
array([145000, 140000, 127100, 100000, 55000, 24000, 20000,  
12000])
```

The 8 items in the array are indexed 0, 1, 2, and so on, up to 7. The items in the column can be accessed using `item`, as with any array.

```
minard.column(4).item(0)
```

```
145000
```

```
minard.column(4).item(5)
```

```
24000
```

Working with the Data in a Column¶

Because columns are arrays, we can use array operations on them to discover new information. For example, we can create a new column that contains the percent of all survivors at each city after Smolensk.

```
initial = minard.column('Survivors').item(0)  
minard = minard.with_columns(  
    'Percent Surviving', minard.column('Survivors')/initial  
)  
minard
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moidexno	Retreat	12000	8.28%

To make the proportions in the new columns appear as percents, we can use the method

`set_format` with the option `PercentFormatter`. The `set_format` method takes `Formatter` objects, which exist for dates (`DateFormatter`), currencies (`CurrencyFormatter`), numbers, and percentages.

```
minard.set_format('Percent Surviving', PercentFormatter)
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moidexno	Retreat	12000	8.28%

Choosing Sets of Columns

The method `select` creates a new table that contains only the specified columns.

```
minard.select('Longitude', 'Latitude')
```

6. Tables

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

The same selection can be made using column indices instead of labels.

```
minard.select(0, 1)
```

Longitude	Latitude
32	54.8
33.2	54.9
34.4	55.5
37.6	55.8
34.3	55.2
32	54.6
30.4	54.4
26.8	54.3

The result of using `select` is a new table, even when you select just one column.

```
minard.select('Survivors')
```

6. Tables

Survivors
145000
140000
127100
100000
55000
24000
20000
12000

Notice that the result is a table, unlike the result of `column`, which is an array.

```
minard.column('Survivors')
```

```
array([145000, 140000, 127100, 100000, 55000, 24000, 20000,  
12000])
```

Another way to create a new table consisting of a set of columns is to `drop` the columns you don't want.

```
minard.drop('Longitude', 'Latitude', 'Direction')
```

City Name	Survivors	Percent Surviving
Smolensk	145000	100.00%
Dorogobouge	140000	96.55%
Chjat	127100	87.66%
Moscou	100000	68.97%
Wixma	55000	37.93%
Smolensk	24000	16.55%
Orscha	20000	13.79%
Moidexno	12000	8.28%

6. Tables

Neither `select` nor `drop` change the original table. Instead, they create new smaller tables that share the same data. The fact that the original table is preserved is useful! You can generate multiple different tables that only consider certain columns without worrying that one analysis will affect the other.

```
minard
```

Longitude	Latitude	City Name	Direction	Survivors	Percent Surviving
32	54.8	Smolensk	Advance	145000	100.00%
33.2	54.9	Dorogobouge	Advance	140000	96.55%
34.4	55.5	Chjat	Advance	127100	87.66%
37.6	55.8	Moscou	Advance	100000	68.97%
34.3	55.2	Wixma	Retreat	55000	37.93%
32	54.6	Smolensk	Retreat	24000	16.55%
30.4	54.4	Orscha	Retreat	20000	13.79%
26.8	54.3	Moidexno	Retreat	12000	8.28%

All of the methods that we have used above can be applied to any table.

Sorting Rows

Interact

"The NBA is the highest paying professional sports league in the world," [reported CNN](#) in March 2016. The table `nba_salaries` contains the salaries of all National Basketball Association players in 2015-2016.

Each row represents one player. The columns are:

Column Label	Description
PLAYER	Player's name
POSITION	Player's position on team
TEAM	Team name
'15 - '16 SALARY	Player's salary in 2015-2016, in millions of dollars

The code for the positions is PG (Point Guard), SG (Shooting Guard), PF (Power Forward), SF (Small Forward), and C (Center). But what follows doesn't involve details about how basketball is played.

The first row shows that Paul Millsap, Power Forward for the Atlanta Hawks, had a salary of almost **\$18.7** million in 2015-2016.

```
# This table can be found online:  
https://www.statcrunch.com/app/index.php?dataid=1843341  
nba_salaries = Table.read_table('nba_salaries.csv')  
nba_salaries
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

... (407 rows omitted)

The table contains 417 rows, one for each player. Only 10 of the rows are displayed. The `show` method allows us to specify the number of rows, with the default (no specification) being all the rows of the table.

```
nba_salaries.show(3)
```

PLAYER	POSITION	TEAM	'15-'16 SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625

... (414 rows omitted)

Glance through about 20 rows or so, and you will see that the rows are in alphabetical order by team name. It's also possible to list the same rows in alphabetical order by player name using the `sort` method. The argument to `sort` is a column label or index.

```
nba_salaries.sort('PLAYER').show(5)
```

6.1 Sorting Rows

PLAYER	POSITION	TEAM	'15-'16 SALARY
Aaron Brooks	PG	Chicago Bulls	2.25
Aaron Gordon	PF	Orlando Magic	4.17168
Aaron Harrison	SG	Charlotte Hornets	0.525093
Adreian Payne	PF	Minnesota Timberwolves	1.93884
Al Horford	C	Atlanta Hawks	12

... (412 rows omitted)

To examine the players' salaries, it would be much more helpful if the data were ordered by salary.

To do this, we will first simplify the label of the column of salaries (just for convenience), and then sort by the new label `SALARY`.

This arranges all the rows of the table in *increasing* order of salary, with the lowest salary appearing first. The output is a new table with the same columns as the original but with the rows rearranged.

```
nba = nba_salaries.relabel("'15-'16 SALARY", "SALARY")
nba.sort('SALARY')
```

PLAYER	POSITION	TEAM	SALARY
Thanasis Antetokounmpo	SF	New York Knicks	0.030888
Jordan McRae	SG	Phoenix Suns	0.049709
Cory Jefferson	PF	Phoenix Suns	0.049709
Elliot Williams	SG	Memphis Grizzlies	0.055722
Orlando Johnson	SG	Phoenix Suns	0.055722
Phil Pressey	PG	Phoenix Suns	0.055722
Keith Appling	PG	Orlando Magic	0.061776
Sean Kilpatrick	SG	Denver Nuggets	0.099418
Erick Green	PG	Utah Jazz	0.099418
Jeff Ayres	PF	Los Angeles Clippers	0.111444

... (407 rows omitted)

These figures are somewhat difficult to compare as some of these players changed teams during the season and received salaries from more than one team; only the salary from the last team appears in the table. Point Guard Phil Pressey, for example, moved from Philadelphia to Phoenix during the year, and might be moving yet again to the Golden State Warriors.

The CNN report is about the other end of the salary scale – the players who are among the highest paid in the world.

To order the rows of the table in *decreasing* order of salary, we must use `sort` with the option `descending=True`.

```
nba.sort('SALARY', descending=True)
```

PLAYER	POSITION	TEAM	SALARY
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.8949
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.3594
Chris Bosh	PF	Miami Heat	22.1927
Chris Paul	PG	Los Angeles Clippers	21.4687
Kevin Durant	SF	Oklahoma City Thunder	20.1586
Derrick Rose	PG	Chicago Bulls	20.0931
Dwyane Wade	SG	Miami Heat	20

... (407 rows omitted)

Kobe Bryant, in his final season with the Lakers, was the highest paid at a salary of **\$25** million. Notice that the MVP Stephen Curry doesn't appear among the top 10. He is quite a bit further down the list, as we will see later.

Named Arguments¶

The `descending=True` portion of this call expression is called a *named argument*. When a function or method is called, each argument has both a position and a name. Both are evident from the help text of a function or method.

```
help(nba.sort)
```

Help on method sort in module datascience.tables:

sort(column_or_label, descending=False, distinct=False) method
of datascience.tables.Table instance

Return a Table of rows sorted according to the values in a column.

Args:

``column_or_label``: the column whose values are used for sorting.

``descending``: if True, sorting will be in descending, rather than

ascending order.

``distinct``: if True, repeated values in ``column_or_label`` will be omitted.

Returns:

An instance of ``Table`` containing rows sorted based on the values in ``column_or_label``.

```
>>> marbles = Table().with_columns(
...     "Color", make_array("Red", "Green", "Blue", "Red",
"Green", "Green"),
...     "Shape", make_array("Round", "Rectangular",
"Rectangular", "Round", "Rectangular", "Round"),
...     "Amount", make_array(4, 6, 12, 7, 9, 2),
...     "Price", make_array(1.30, 1.30, 2.00, 1.75, 1.40,
1.00))
>>> marbles
Color | Shape      | Amount | Price
Red   | Round      | 4      | 1.3
Green | Rectangular | 6      | 1.3
Blue  | Rectangular | 12     | 2
```

```

Red    | Round      | 7     | 1.75
Green  | Rectangular | 9     | 1.4
Green  | Round      | 2     | 1
>>> marbles.sort("Amount")
Color | Shape      | Amount | Price
Green | Round      | 2     | 1
Red   | Round      | 4     | 1.3
Green | Rectangular | 6     | 1.3
Red   | Round      | 7     | 1.75
Green | Rectangular | 9     | 1.4
Blue  | Rectangular | 12    | 2
>>> marbles.sort("Amount", descending = True)
Color | Shape      | Amount | Price
Blue  | Rectangular | 12    | 2
Green | Rectangular | 9     | 1.4
Red   | Round      | 7     | 1.75
Green | Rectangular | 6     | 1.3
Red   | Round      | 4     | 1.3
Green | Round      | 2     | 1
>>> marbles.sort(3) # the Price column
Color | Shape      | Amount | Price
Green | Round      | 2     | 1
Red   | Round      | 4     | 1.3
Green | Rectangular | 6     | 1.3
Green | Rectangular | 9     | 1.4
Red   | Round      | 7     | 1.75
Blue  | Rectangular | 12    | 2
>>> marbles.sort(3, distinct = True)
Color | Shape      | Amount | Price
Green | Round      | 2     | 1
Red   | Round      | 4     | 1.3
Green | Rectangular | 9     | 1.4
Red   | Round      | 7     | 1.75
Blue  | Rectangular | 12    | 2

```

At the very top of this `help` text, the *signature* of the `sort` method appears:

```
sort(column_or_label, descending=False, distinct=False)
```

This describes the positions, names, and default values of the three arguments to `sort`. When calling this method, you can use either positional arguments or named arguments, so the following three calls do exactly the same thing.

```
sort('SALARY', True)
sort('SALARY', descending=True)
sort(column_or_label='SALARY', descending=True)
```

When an argument is simply `True` or `False`, it's a useful convention to include the argument name so that it's more obvious what the argument value means.

Selecting Rows

Interact

Often, we would like to extract just those rows that correspond to entries with a particular feature. For example, we might want only the rows corresponding to the Warriors, or to players who earned more than \$10 million. Or we might just want the top five earners.

Specified Rows

The Table method `take` does just that – it takes a specified set of rows. Its argument is a row index or array of indices, and it creates a new table consisting of only those rows.

For example, if we wanted just the first row of `nba`, we could use `take` as follows.

```
nba
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

... (407 rows omitted)

```
nba.take(0)
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717

This is a new table with just the single row that we specified.

We could also get the fourth, fifth, and sixth rows by specifying a range of indices as the argument.

```
nba.take(np.arange(3, 6))
```

PLAYER	POSITION	TEAM	SALARY
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4

If we want a table of the top 5 highest paid players, we can first sort the list by salary and then `take` the first five rows:

```
nba.sort('SALARY', descending=True).take(np.arange(5))
```

PLAYER	POSITION	TEAM	SALARY
Kobe Bryant	SF	Los Angeles Lakers	25
Joe Johnson	SF	Brooklyn Nets	24.8949
LeBron James	SF	Cleveland Cavaliers	22.9705
Carmelo Anthony	SF	New York Knicks	22.875
Dwight Howard	C	Houston Rockets	22.3594

Rows Corresponding to a Specified Feature

More often, we will want to access data in a set of rows that have a certain feature, but whose indices we don't know ahead of time. For example, we might want data on all the players who made more than **\$10** million, but we don't want to spend time counting rows in the sorted table.

The method `where` does the job for us. Its output is a table with the same columns as the original but only the rows *where* the feature occurs.

The first argument of `where` is the label of the column that contains the information about whether or not a row has the feature we want. If the feature is "made more than **\$10** million", the column is `SALARY`.

The second argument of `where` is a way of specifying the feature. A couple of examples will make the general method of specification easier to understand.

In the first example, we extract the data for all those who earned more than \$10 million.

```
nba.where('SALARY', are.above(10))
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Joe Johnson	SF	Brooklyn Nets	24.8949
Thaddeus Young	PF	Brooklyn Nets	11.236
Al Jefferson	C	Charlotte Hornets	13.5
Nicolas Batum	SG	Charlotte Hornets	13.1253
Kemba Walker	PG	Charlotte Hornets	12
Derrick Rose	PG	Chicago Bulls	20.0931
Jimmy Butler	SG	Chicago Bulls	16.4075
Joakim Noah	C	Chicago Bulls	13.4

... (59 rows omitted)

The use of the argument `are.above(10)` ensured that each selected row had a value of `SALARY` that was greater than 10.

There are 69 rows in the new table, corresponding to the 69 players who made more than 10 million dollars. Arranging these rows in order makes the data easier to analyze. DeMar DeRozan of the Toronto Raptors was the "poorest" of this group, at a salary of just over 10 million dollars.

```
nba.where('SALARY', are.above(10)).sort('SALARY')
```

PLAYER	POSITION	TEAM	SALARY
DeMar DeRozan	SG	Toronto Raptors	10.05
Gerald Wallace	SF	Philadelphia 76ers	10.1059
Luol Deng	SF	Miami Heat	10.1516
Monta Ellis	SG	Indiana Pacers	10.3
Wilson Chandler	SF	Denver Nuggets	10.4494
Brendan Haywood	C	Cleveland Cavaliers	10.5225
Jrue Holiday	PG	New Orleans Pelicans	10.5955
Tyreke Evans	SG	New Orleans Pelicans	10.7346
Marcin Gortat	C	Washington Wizards	11.2174
Thaddeus Young	PF	Brooklyn Nets	11.236

... (59 rows omitted)

How much did Stephen Curry make? For the answer, we have to access the row where the value of `PLAYER` is equal to `Stephen Curry`. That is placed a table consisting of just one line:

```
nba.where('PLAYER', are.equal_to('Stephen Curry'))
```

PLAYER	POSITION	TEAM	SALARY
Stephen Curry	PG	Golden State Warriors	11.3708

Curry made just under **\$11.4** million dollars. That's a lot of money, but it's less than half the salary of LeBron James. You'll find that salary in the "Top 5" table earlier in this section, or you could find it replacing `'Stephen Curry'` by `'LeBron James'` in the line of code above.

In the code, `are` is used again, but this time with the *predicate* `equal_to` instead of `above`. Thus for example you can get a table of all the Warriors:

```
nba.where('TEAM', are.equal_to('Golden State Warriors')).show()
```

PLAYER	POSITION	TEAM	SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5
Festus Ezeli	C	Golden State Warriors	2.00875
Brandon Rush	SF	Golden State Warriors	1.27096
Kevon Looney	SF	Golden State Warriors	1.13196
Anderson Varejao	PF	Golden State Warriors	0.289755

This portion of the table is already sorted by salary, because the original table listed players sorted by salary within the same team. The `.show()` at the end of the line ensures that all rows are shown, not just the first 10.

It is so common to ask for the rows for which some column is equal to some value that the `are.equal_to` call is optional. Instead, the `where` method can be called with only a column name and a value to achieve the same effect.

```
nba.where('TEAM', 'Denver Nuggets') # equivalent to
nba.where('TEAM', are.equal_to('Denver Nuggets'))
```

PLAYER	POSITION	TEAM	SALARY
Danilo Gallinari	SF	Denver Nuggets	14
Kenneth Faried	PF	Denver Nuggets	11.236
Wilson Chandler	SF	Denver Nuggets	10.4494
JJ Hickson	C	Denver Nuggets	5.6135
Jameer Nelson	PG	Denver Nuggets	4.345
Will Barton	SF	Denver Nuggets	3.53333
Emmanuel Mudiay	PG	Denver Nuggets	3.10224
Darrell Arthur	PF	Denver Nuggets	2.814
Jusuf Nurkic	C	Denver Nuggets	1.842
Joffrey Lauvergne	C	Denver Nuggets	1.70972

... (4 rows omitted)

Multiple Features

You can access rows that have multiple specified features, by using `where` repeatedly. For example, here is a way to extract all the Point Guards whose salaries were over \$15 million.

```
nba.where('POSITION', 'PG').where('SALARY', are.above(15))
```

PLAYER	POSITION	TEAM	SALARY
Derrick Rose	PG	Chicago Bulls	20.0931
Kyrie Irving	PG	Cleveland Cavaliers	16.4075
Chris Paul	PG	Los Angeles Clippers	21.4687
Russell Westbrook	PG	Oklahoma City Thunder	16.7442
John Wall	PG	Washington Wizards	15.852

General Form

By now you will have realized that the general way to create a new table by selecting rows with a given feature is to use `where` and `are` with the appropriate condition:

```
original_table_name.where(column_label_string, are.condition)
```

```
nba.where('SALARY', are.between(10, 10.3))
```

PLAYER	POSITION	TEAM	SALARY
Luol Deng	SF	Miami Heat	10.1516
Gerald Wallace	SF	Philadelphia 76ers	10.1059
Danny Green	SG	San Antonio Spurs	10
DeMar DeRozan	SG	Toronto Raptors	10.05

Notice that the table above includes Danny Green who made **\$10** million, but *not* Monta Ellis who made **\$10.3** million. As elsewhere in Python, the range `between` includes the left end but not the right.

If we specify a condition that isn't satisfied by any row, we get a table with column labels but no rows.

```
nba.where('PLAYER', are.equal_to('Barack Obama'))
```

PLAYER	POSITION	TEAM	SALARY

Some More Conditions

Here are some predicates of `are` that you might find useful. Note that `x` and `y` are numbers, `STRING` is a string, and `z` is either a number or a string; you have to specify these depending on the feature you want.

Predicate	Description
<code>are.equal_to(z)</code>	Equal to <code>z</code>
<code>are.above(x)</code>	Greater than <code>x</code>
<code>are.above_or_equal_to(x)</code>	Greater than or equal to <code>x</code>
<code>are.below(x)</code>	Less than <code>x</code>
<code>are.below_or_equal_to(x)</code>	Less than or equal to <code>x</code>
<code>are.between(x, y)</code>	Greater than or equal to <code>x</code> , and less than <code>y</code>
<code>are.strictly_between(x, y)</code>	Greater than <code>x</code> and less than <code>y</code>
<code>are.between_or_equal_to(x, y)</code>	Greater than or equal to <code>x</code> , and less than or equal to <code>y</code>
<code>are.containing(s)</code>	Contains the string <code>s</code>

You can also specify the negation of any of these conditions, by using `.not_` before the condition:

Predicate	Description
are.not_equal_to(z)	Not equal to z
are.not_above(x)	Not above x

... and so on. The usual rules of logic apply – for example, "not above x" is the same as "below or equal to x".

We end the section with a series of examples.

The use of `are.containing` can help save some typing. For example, you can just specify `Warriors` instead of `Golden State Warriors`:

```
nba.where('TEAM', are.containing('Warriors')).show()
```

PLAYER	POSITION	TEAM	SALARY
Klay Thompson	SG	Golden State Warriors	15.501
Draymond Green	PF	Golden State Warriors	14.2609
Andrew Bogut	C	Golden State Warriors	13.8
Andre Iguodala	SF	Golden State Warriors	11.7105
Stephen Curry	PG	Golden State Warriors	11.3708
Jason Thompson	PF	Golden State Warriors	7.00847
Shaun Livingston	PG	Golden State Warriors	5.54373
Harrison Barnes	SF	Golden State Warriors	3.8734
Marreese Speights	C	Golden State Warriors	3.815
Leandro Barbosa	SG	Golden State Warriors	2.5
Festus Ezeli	C	Golden State Warriors	2.00875
Brandon Rush	SF	Golden State Warriors	1.27096
Kevon Looney	SF	Golden State Warriors	1.13196
Anderson Varejao	PF	Golden State Warriors	0.289755

You can extract data for all the guards, both Point Guards and Shooting Guards:

```
nba.where('POSITION', are.containing('G'))
```

PLAYER	POSITION	TEAM	SALARY
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452
Jason Richardson	SG	Atlanta Hawks	0.947276
Lamar Patterson	SG	Atlanta Hawks	0.525093
Terran Petteway	SG	Atlanta Hawks	0.525093
Avery Bradley	PG	Boston Celtics	7.73034
Isaiah Thomas	PG	Boston Celtics	6.91287
Marcus Smart	PG	Boston Celtics	3.43104

... (171 rows omitted)

You can get all the players who were not Cleveland Cavaliers and had a salary of no less than **\$20** million:

```
other_than_Cavs = nba.where('TEAM', are.not_equal_to('Cleveland
Cavaliers'))
other_than_Cavs.where('SALARY', are.not_below(20))
```

PLAYER	POSITION	TEAM	SALARY
Joe Johnson	SF	Brooklyn Nets	24.8949
Derrick Rose	PG	Chicago Bulls	20.0931
Dwight Howard	C	Houston Rockets	22.3594
Chris Paul	PG	Los Angeles Clippers	21.4687
Kobe Bryant	SF	Los Angeles Lakers	25
Chris Bosh	PF	Miami Heat	22.1927
Dwyane Wade	SG	Miami Heat	20
Carmelo Anthony	SF	New York Knicks	22.875
Kevin Durant	SF	Oklahoma City Thunder	20.1586

The same table can be created in many ways. Here is another, and no doubt you can think of more.

```
other_than_Cavs.where('SALARY', are.above_or_equal_to(20))
```

PLAYER	POSITION	TEAM	SALARY
Joe Johnson	SF	Brooklyn Nets	24.8949
Derrick Rose	PG	Chicago Bulls	20.0931
Dwight Howard	C	Houston Rockets	22.3594
Chris Paul	PG	Los Angeles Clippers	21.4687
Kobe Bryant	SF	Los Angeles Lakers	25
Chris Bosh	PF	Miami Heat	22.1927
Dwyane Wade	SG	Miami Heat	20
Carmelo Anthony	SF	New York Knicks	22.875
Kevin Durant	SF	Oklahoma City Thunder	20.1586

As you can see, the use of `where` with `are` gives you great flexibility in accessing rows with features that interest you. Don't hesitate to experiment!

Trends in the US Population

Interact

We are now ready to work with large tables of data. The file below contains "Annual Estimates of the Resident Population by Single Year of Age and Sex for the United States." Notice that `read_table` can read data directly from a URL.

```
# As of Jan 2017, this census file is online here:
data = 'http://www2.census.gov/programs-
surveys/popest/datasets/2010-2015/national/asrh/nc-est2015-
agesex-res.csv'

# A local copy can be accessed here in case census.gov moves the
file:
# data = 'nc-est2015-agesex-res.csv'

full_census_table = Table.read_table(data)
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	4077551
0	5	4056858	4056872	4064653
0	6	4066381	4066412	4073013
0	7	4030579	4030594	4043046
0	8	4046486	4046497	4025604
0	9	4148353	4148369	4125415

... (296 rows omitted)

Only the first 10 rows of the table are displayed. Later we will see how to display the entire table; however, this is typically not useful with large tables.

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 in completed years, 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. Let us `select` the relevant columns.

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

SEX	AGE	POPESTIMATE2010	POPESTIMATE2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349

... (296 rows omitted)

We can also simplify the labels of the selected columns.

```
us_pop = partial_census_table.relabeled('POPESTIMATE2010',
    '2010').relabeled('POPESTIMATE2014', '2014')
us_pop
```

SEX	AGE	2010	2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349

... (296 rows omitted)

We now have a table that is easy to work with. Each column of the table is an array of the same length, and so columns can be combined using arithmetic. Here is the change in population between 2010 and 2014.

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

```
array([-1555, -8112, -131198, ..., 6443, 12950,
4693244])
```

Let us augment `us_pop` with a column that contains these changes, both in absolute terms and as percents relative to the value in 2010.

```
change = us_pop.column('2014') - us_pop.column('2010')
census = us_pop.with_columns(
    'Change', change,
    'Percent Change', change/us_pop.column('2010'))
)
census.set_format('Percent Change', PercentFormatter)
```

6.3 Example: Population Trends

SEX	AGE	2010	2014	Change	Percent Change
0	0	3951330	3949775	-1555	-0.04%
0	1	3957888	3949776	-8112	-0.20%
0	2	4090862	3959664	-131198	-3.21%
0	3	4111920	4007079	-104841	-2.55%
0	4	4077551	4005716	-71835	-1.76%
0	5	4064653	4006900	-57753	-1.42%
0	6	4073013	4135930	62917	1.54%
0	7	4043046	4155326	112280	2.78%
0	8	4025604	4120903	95299	2.37%
0	9	4125415	4108349	-17066	-0.41%

... (296 rows omitted)

Sorting the data. Let us sort the table in decreasing order of the absolute change in population.

```
census.sort('Change', descending=True)
```

SEX	AGE	2010	2014	Change	Percent Change
0	999	309346863	318907401	9560538	3.09%
1	999	152088043	156955337	4867294	3.20%
2	999	157258820	161952064	4693244	2.98%
0	67	2693707	3485241	791534	29.38%
0	64	2706055	3487559	781504	28.88%
0	66	2621335	3347060	725725	27.69%
0	65	2678525	3382824	704299	26.29%
0	71	1953607	2519705	566098	28.98%
0	34	3822189	4364748	542559	14.19%
0	23	4217228	4702156	484928	11.50%

... (296 rows omitted)

Not surprisingly, the top row of the sorted table is the line that corresponds to the entire population: both sexes and all age groups. From 2010 to 2014, the population of the United States increased by about 9.5 million people, a change of just over 3%.

The next two rows correspond to all the men and all the women respectively. The male population grew more than the female population, both in absolute and percentage terms. Both percent changes were around 3%.

Now take a look at the next few rows. The percent change jumps from about 3% for the overall population to almost 30% for the people in their late sixties and early seventies. This stunning change contributes to what is known as the greying of America.

By far the greatest absolute change was among those in the 64-67 agegroup in 2014. What could explain this large increase? We can explore this question by examining the years in which the relevant groups were born.

- Those who were in the 64-67 age group in 2010 were born in the years 1943 to 1946. The attack on Pearl Harbor was in late 1941, and by 1942 U.S. forces were heavily engaged in a massive war that ended in 1945.
- Those who were 64 to 67 years old in 2014 were born in the years 1947 to 1950, at the height of the post-WWII baby boom in the United States.

The post-war jump in births is the major reason for the large changes that we have observed.

Trends in Gender Ratios

Interact

We are now equipped with enough coding skills to examine features and trends in subgroups of the U.S. population. In this example, we will look at the distribution of males and females across age groups. We will continue using the `us_pop` table from the previous section.

`us_pop`

SEX	AGE	2010	2014
0	0	3951330	3949775
0	1	3957888	3949776
0	2	4090862	3959664
0	3	4111920	4007079
0	4	4077551	4005716
0	5	4064653	4006900
0	6	4073013	4135930
0	7	4043046	4155326
0	8	4025604	4120903
0	9	4125415	4108349
... (296 rows omitted)			

As we know from having examined this dataset earlier, a [description of the table](#) appears online. Here is a reminder of what the table contains.

Each row represents an age group. The `SEX` column contains numeric codes: `0` stands for the total, `1` for male, and `2` for female. The `AGE` column contains ages in completed years, but the special value `999` represents the entire population regardless of age. The rest of the columns contain estimates of the US population.

Understanding AGE = 100 $\ddot{\text{P}}$

As a preliminary, let's interpret data in the final age category in the table, where `AGE` is 100. The code below extracts the rows for the combined group of men and women (`SEX` code 0) for the highest ages.

```
us_pop.where('SEX', are.equal_to(0)).where('AGE',
are.between(97, 101))
```

SEX	AGE	2010	2014
0	97	68893	83089
0	98	47037	59726
0	99	32178	41468
0	100	54410	71626

Not surprisingly, the numbers of people are smaller at higher ages – for example, there are fewer 99-year-olds than 98-year-olds.

It does come as a surprise, though, that the numbers for `AGE` 100 are quite a bit larger than those for age 99. A closer examination of the documentation shows that it's because the Census Bureau used 100 as the code for everyone aged 100 or more.

The row with `AGE` 100 doesn't just represent 100-year-olds – it also includes those who are older than 100. That is why the numbers in that row are larger than in the row for the 99-year-olds.

Overall Proportions of Males and Females

We will now begin looking at gender ratios in 2014. First, let's look at all the age groups together. Remember that this means looking at the rows where the "age" is coded 999. The table `all_ages` contains this information. There are three rows: one for the total of both genders, one for males (`SEX` code 1), and one for females (`SEX` code 2).

```
us_pop_2014 = us_pop.drop('2010')
all_ages = us_pop_2014.where('AGE', are.equal_to(999))
all_ages
```

SEX	AGE	2014
0	999	318907401
1	999	156955337
2	999	161952064

Row 0 of `all_ages` contains the total U.S. population in each of the two years. The United States had just under 319 million in 2014.

Row 1 contains the counts for males and Row 2 for females. Compare these two rows to see that in 2014, there were more females than males in the United States.

The population counts in Row 1 and Row 2 add up to the total population in Row 0.

For comparability with other quantities, we will need to convert these counts to percents out of the total population. Let's access the total for 2014 and name it. Then, we'll show a population table with a proportion column. Consistent with our earlier observation that there were more females than males, about 50.8% of the population in 2014 was female and about 49.2% male in each of the two years.

```
pop_2014 = all_ages.column('2014').item(0)
all_ages.with_column(
    'Proportion', all_ages.column('2014')/pop_2014
).set_format('Proportion', PercentFormatter)
```

SEX	AGE	2014	Proportion
0	999	318907401	100.00%
1	999	156955337	49.22%
2	999	161952064	50.78%

Proportions of Boys and Girls among Infants

When we look at infants, however, the opposite is true. Let's define infants to be babies who have not yet completed one year, represented in the row corresponding to `AGE` 0. Here are their numbers in the population. You can see that male infants outnumbered female infants.

```
infants = us_pop_2014.where('AGE', are.equal_to(0))
infants
```

SEX	AGE	2014
0	0	3949775
1	0	2020326
2	0	1929449

As before, we can convert these counts to percents out of the total numbers of infants. The resulting table shows that in 2014, just over 51% of infants in the U.S. were male.

```
infants_2014 = infants.column('2014').item(0)
infants.with_column(
    'Proportion', infants.column('2014')/infants_2014
).set_format('Proportion', PercentFormatter)
```

SEX	AGE	2014	Proportion
0	0	3949775	100.00%
1	0	2020326	51.15%
2	0	1929449	48.85%

In fact, it has long been observed that the proportion of boys among newborns is slightly more than 1/2. The reason for this is not thoroughly understood, and [scientists are still working on it](#).

Female:Male Gender Ratio at Each Age

We have seen that while there are more baby boys than baby girls, there are more females than males overall. So it's clear that the split between genders must vary across age groups.

To study this variation, we will separate out the data for the females and the males, and eliminate the row where all the ages are aggregated and `AGE` is coded as 999.

The tables `females` and `males` contain the data for each the two genders.

```
females_all_rows = us_pop_2014.where('SEX', are.equal_to(2))
females = females_all_rows.where('AGE', are.not_equal_to(999))
females
```

6.4 Example: Trends in Gender

SEX	AGE	2014
2	0	1929449
2	1	1931375
2	2	1935991
2	3	1957483
2	4	1961199
2	5	1962561
2	6	2024870
2	7	2032494
2	8	2015285
2	9	2010659

... (91 rows omitted)

```
males_all_rows = us_pop_2014.where('SEX', are.equal_to(1))
males = males_all_rows.where('AGE', are.not_equal_to(999))
males
```

SEX	AGE	2014
1	0	2020326
1	1	2018401
1	2	2023673
1	3	2049596
1	4	2044517
1	5	2044339
1	6	2111060
1	7	2122832
1	8	2105618
1	9	2097690

... (91 rows omitted)

The plan now is to compare the number of women and the number of men at each age, for each of the two years. Array and Table methods give us straightforward ways to do this. Both of these tables have one row for each age.

```
males.column('AGE')
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
11, 12,
       13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25,
       26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38,
       39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51,
       52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64,
       65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77,
       78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90,
       91, 92, 93, 94, 95, 96, 97, 98, 99, 100])
```

```
females.column('AGE')
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10,
11, 12,
       13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25,
       26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38,
       39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51,
       52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64,
       65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,
76, 77,
       78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90,
       91, 92, 93, 94, 95, 96, 97, 98, 99, 100])
```

For any given age, we can get the Female:Male gender ratio by dividing the number of females by the number of males. To do this in one step, we can use `column` to extract the array of female counts and the corresponding array of male counts, and then simply divide one array by the other. Elementwise division will create an array of gender ratios for all the years.

```
ratios = Table().with_columns(
    'AGE', females.column('AGE'),
    '2014 F:M RATIO',
    females.column('2014')/males.column('2014')
)
ratios
```

AGE	2014 F:M RATIO
0	0.955019
1	0.956884
2	0.956672
3	0.955058
4	0.959248
5	0.959998
6	0.959172
7	0.957445
8	0.957099
9	0.958511
... (91 rows omitted)	

You can see from the display that the ratios are all around 0.96 for children aged nine or younger. When the Female:Male ratio is less than 1, there are fewer females than males. Thus what we are seeing is that there were fewer girls than boys in each of the age groups 0, 1, 2, and so on through 9. Moreover, in each of these age groups, there were about 96 girls for every 100 boys.

So how can the overall proportion of females in the population be higher than the males?

Something extraordinary happens when we examine the other end of the age range. Here are the Female:Male ratios for people aged more than 75.

```
ratios.where('AGE', are.above(75)).show()
```

6.4 Example: Trends in Gender

AGE	2014 F:M RATIO
76	1.23487
77	1.25797
78	1.28244
79	1.31627
80	1.34138
81	1.37967
82	1.41932
83	1.46552
84	1.52048
85	1.5756
86	1.65096
87	1.72172
88	1.81223
89	1.91837
90	2.01263
91	2.09488
92	2.2299
93	2.33359
94	2.52285
95	2.67253
96	2.87998
97	3.09104
98	3.41826
99	3.63278
100	4.25966

Not only are all of these ratios greater than 1, signifying more women than men in all of these age groups, many of them are considerably greater than 1.

- At ages 89 and 90 the ratios are close to 2, meaning that there were about twice as many women as men at those ages in 2014.
- At ages 98 and 99, there were about 3.5 to 4 times as many women as men.

6.4 Example: Trends in Gender

If you are wondering how many people there were at these advanced ages, you can use Python to find out:

```
males.where('AGE', are.between(98, 100))
```

SEX	AGE	2014
1	98	13518
1	99	8951

```
females.where('AGE', are.between(98, 100))
```

SEX	AGE	2014
2	98	46208
2	99	32517

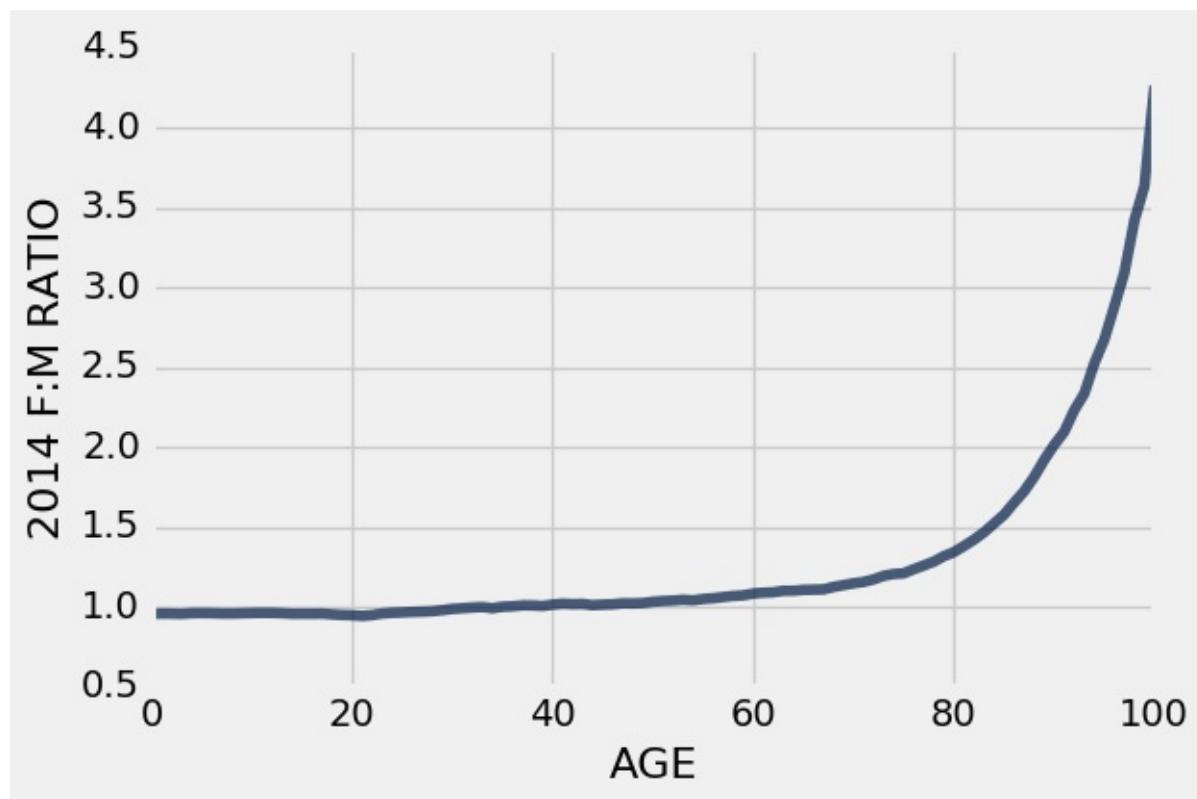
The graph below shows the gender ratios plotted against age. The blue curve shows the 2014 ratio by age.

The ratios are almost 1 (signifying close to equal numbers of males and females) for ages 0 through 60, but they start shooting up dramatically (more females than males) starting at about age 65.

That females outnumber males in the U.S. is partly due to the marked gender imbalance in favor of women among senior citizens.

```
ratios.plot('AGE')
```

6.4 Example: Trends in Gender



[Interact](#)

Visualization

Tables are a powerful way of organizing and visualizing data. However, large tables of numbers can be difficult to interpret, no matter how organized they are. Sometimes it is much easier to interpret graphs than numbers.

In this chapter we will develop some of the fundamental graphical methods of data analysis. Our source of data is the [Internet Movie Database](#), an online database that contains information about movies, television shows, video games, and so on. The site [Box Office Mojo](#) provides many summaries of IMDB data, some of which we have adapted. We have also used data summaries from [The Numbers](#), a site with a tagline that says it is "where data and the movie business meet."

Scatter Plots and Line Graphs

The table `actors` contains data on Hollywood actors, both male and female. The columns are:

Column	Contents
Actor	Name of actor
Total Gross	Total gross domestic box office receipt, in millions of dollars, of all of the actor's movies
Number of Movies	The number of movies the actor has been in
Average per Movie	Total gross divided by number of movies
#1 Movie	The highest grossing movie the actor has been in
Gross	Gross domestic box office receipt, in millions of dollars, of the actor's #1 Movie

In the calculation of the gross receipt, the data tabulators did not include movies where an actor had a cameo role or a speaking role that did not involve much screen time.

The table has 50 rows, corresponding to the 50 top grossing actors. The table is already sorted by `Total Gross`, so it is easy to see that Harrison Ford is the highest grossing actor. In total, his movies have brought in more money at domestic box office than the movies of any other actor.

```
actors = Table.read_table('actors.csv')
actors
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Harrison Ford	4871.7	41	118.8	Star Wars: The Force Awakens	936.7
Samuel L. Jackson	4772.8	69	69.2	The Avengers	623.4
Morgan Freeman	4468.3	61	73.3	The Dark Knight	534.9
Tom Hanks	4340.8	44	98.7	Toy Story 3	415
Robert Downey, Jr.	3947.3	53	74.5	The Avengers	623.4
Eddie Murphy	3810.4	38	100.3	Shrek 2	441.2
Tom Cruise	3587.2	36	99.6	War of the Worlds	234.3
Johnny Depp	3368.6	45	74.9	Dead Man's Chest	423.3
Michael Caine	3351.5	58	57.8	The Dark Knight	534.9
Scarlett Johansson	3341.2	37	90.3	The Avengers	623.4

... (40 rows omitted)

Terminology. A *variable* is a formal name for what we have been calling a "feature", such as 'number of movies.' The term *variable* emphasizes that the feature can have different values for different individuals – the numbers of movies that actors have been in varies across all the actors.

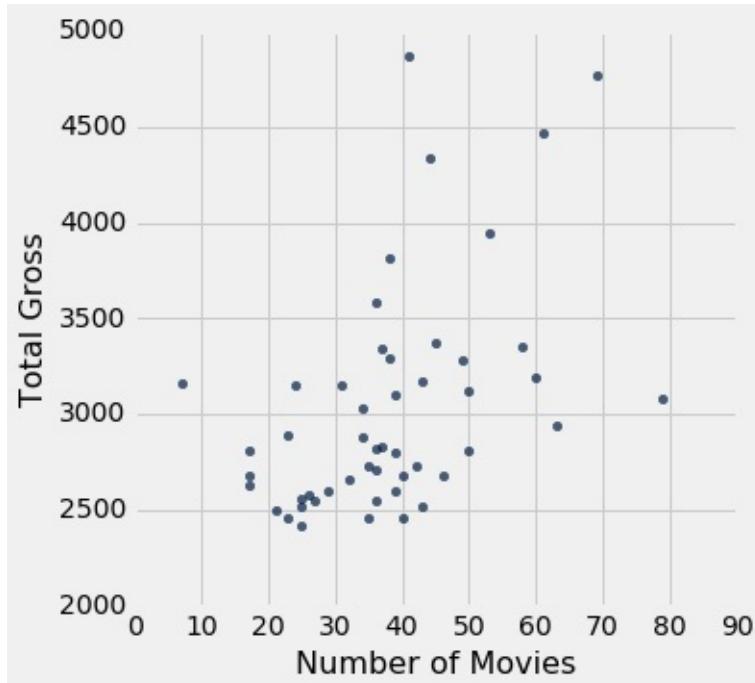
Variables that have numerical values, such as 'number of movies' or 'average gross receipts per movie' are called *quantitative* or *numerical* variables.

Scatter Plots

A *scatter plot* displays the relation between two numerical variables. You saw an example of a scatter plot in an early section where we looked at the number of periods and number of characters in two classic novels.

The Table method `scatter` draws a scatter plot consisting of one point for each row of the table. Its first argument is the label of the column to be plotted on the horizontal axis, and its second argument is the label of the column on the vertical.

```
actors.scatter('Number of Movies', 'Total Gross')
```



The plot contains 50 points, one point for each actor in the table. You can see that it slopes upwards, in general. The more movies an actor has been in, the more the total gross of all of those movies – in general.

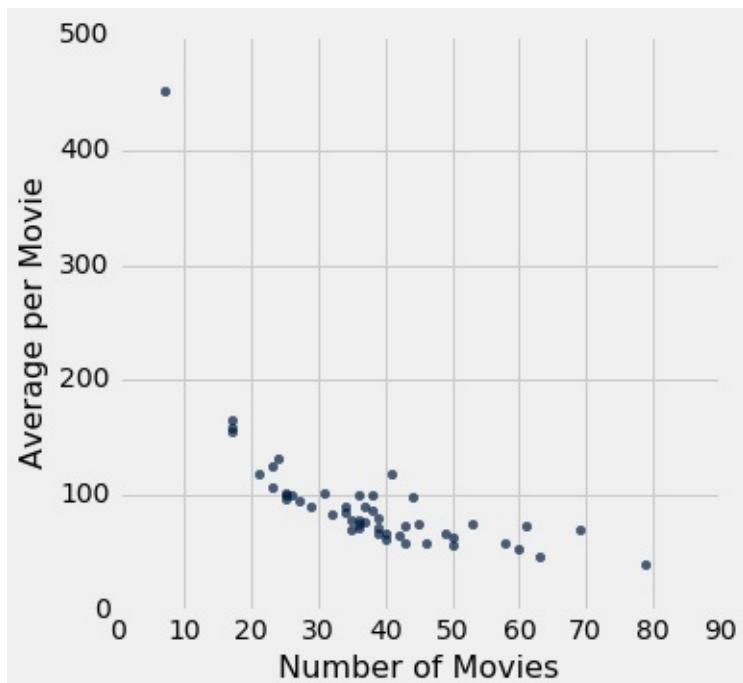
Formally, we say that the plot shows an *association* between the variables, and that the association is *positive*: high values of one variable tend to be associated with high values of the other, and low values of one with low values of the other, in general.

Of course there is some variability. Some actors have high numbers of movies but middling total gross receipts. Others have middling numbers of movies but high receipts. That the association is positive is simply a statement about the broad general trend.

Later in the course we will study how to quantify association. For the moment, we will just think about it qualitatively.

Now that we have explored how the number of movies is related to the *total* gross receipt, let's turn our attention to how it is related to the *average* gross receipt per movie.

```
actors.scatter('Number of Movies', 'Average per Movie')
```



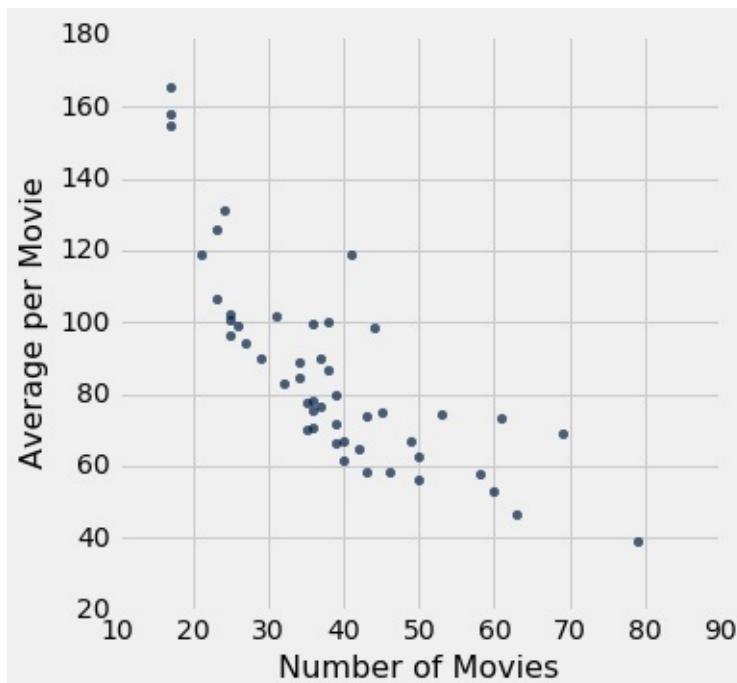
This is a markedly different picture and shows a *negative* association. In general, the more movies an actor has been in, the *less* the average receipt per movie.

Also, one of the points is quite high and off to the left of the plot. It corresponds to one actor who has a low number of movies and high average per movie. This point is an *outlier*. It lies outside the general range of the data. Indeed, it is quite far from all the other points in the plot.

We will examine the negative association further by looking at points at the right and left ends of the plot.

For the right end, let's zoom in on the main body of the plot by just looking at the portion that doesn't have the outlier.

```
no_outlier = actors.where('Number of Movies', are.above(10))
no_outlier.scatter('Number of Movies', 'Average per Movie')
```



The negative association is still clearly visible. Let's identify the actors corresponding to the points that lie on the right hand side of the plot where the number of movies is large:

```
actors.where('Number of Movies', are.above(60))
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Samuel L. Jackson	4772.8	69	69.2	The Avengers	623.4
Morgan Freeman	4468.3	61	73.3	The Dark Knight	534.9
Robert DeNiro	3081.3	79	39	Meet the Fockers	279.3
Liam Neeson	2942.7	63	46.7	The Phantom Menace	474.5

The great actor Robert DeNiro has the highest number of movies and the lowest average receipt per movie. Other fine actors are at points that are not very far away, but DeNiro's is at the extreme end.

To understand the negative association, note that the more movies an actor is in, the more variable those movies might be, in terms of style, genre, and box office draw. For example, an actor might be in some high-grossing action movies or comedies (such as *Meet the Fockers*), and also in a large number of smaller films that may be excellent but don't draw large crowds. Thus the actor's value of average receipts per movie might be relatively low.

To approach this argument from a different direction, let us now take a look at the outlier.

```
actors.where('Number of Movies', are.below(10))
```

Actor	Total Gross	Number of Movies	Average per Movie	#1 Movie	Gross
Anthony Daniels	3162.9	7	451.8	Star Wars: The Force Awakens	936.7

As an actor, Anthony Daniels might not have the stature of Robert DeNiro. But his 7 movies had an astonishingly high average receipt of nearly **452** million dollars per movie.

What were these movies? You might know about the droid C-3PO in Star Wars:



That's [Anthony Daniels](#) inside the metallic suit. He plays C-3PO.

Mr. Daniels' entire filmography (apart from cameos) consists of movies in the high-grossing Star Wars franchise. That explains both his high average receipt and his low number of movies.

Variables such as genre and production budget have an effect on the association between the number of movies and the average receipt per movie. This example is a reminder that studying the association between two variables often involves understanding other related variables as well.

Line Graphs

Line graphs are among the most common visualizations and are often used to study chronological trends and patterns.

The table `movies_by_year` contains data on movies produced by U.S. studios in each of the years 1980 through 2015. The columns are:

Column	Content
Year	Year
Total Gross	Total domestic box office gross, in millions of dollars, of all movies released
Number of Movies	Number of movies released
#1 Movie	Highest grossing movie

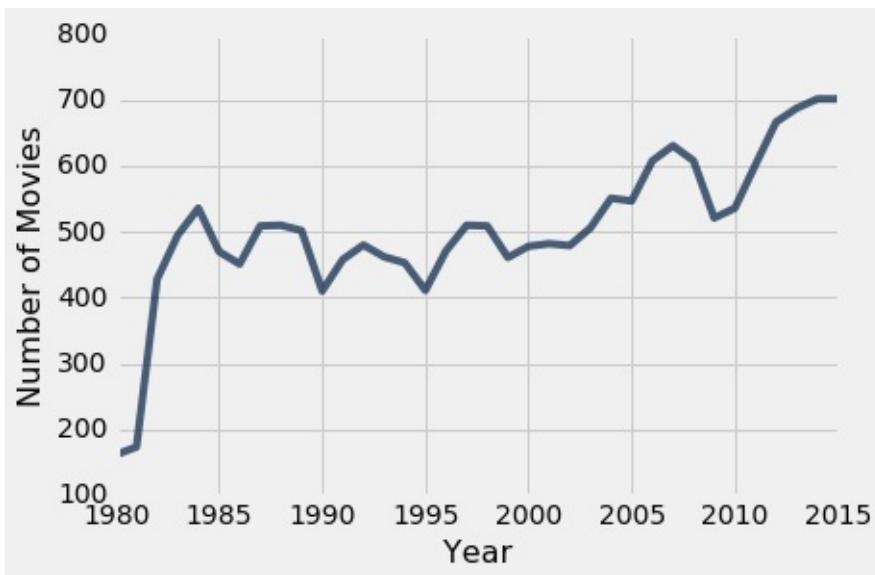
```
movies_by_year = Table.read_table('movies_by_year.csv')
movies_by_year
```

Year	Total Gross	Number of Movies	#1 Movie
2015	11128.5	702	Star Wars: The Force Awakens
2014	10360.8	702	American Sniper
2013	10923.6	688	Catching Fire
2012	10837.4	667	The Avengers
2011	10174.3	602	Harry Potter / Deathly Hallows (P2)
2010	10565.6	536	Toy Story 3
2009	10595.5	521	Avatar
2008	9630.7	608	The Dark Knight
2007	9663.8	631	Spider-Man 3
2006	9209.5	608	Dead Man's Chest

... (26 rows omitted)

The Table method `plot` produces a line graph. Its two arguments are the same as those for `scatter`: first the column on the horizontal axis, then the column on the vertical. Here is a line graph of the number of movies released each year over the years 1980 through 2015.

```
movies_by_year.plot('Year', 'Number of Movies')
```

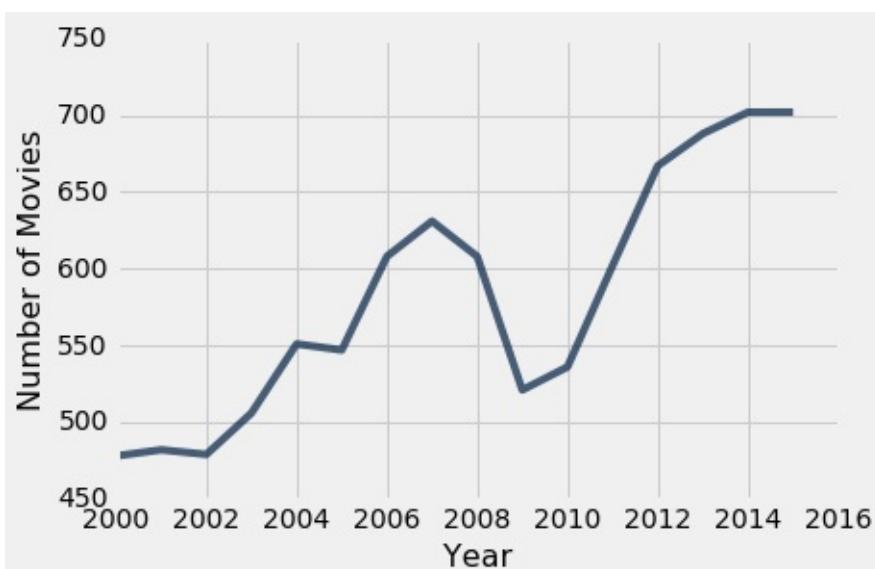


The graph rises sharply and then has a gentle upwards trend though the numbers vary noticeably from year to year. The sharp rise in the early 1980's is due in part to studios returning to the forefront of movie production after some years of filmmaker driven movies in the 1970's.

Our focus will be on more recent years. In keeping with the theme of movies, the table of rows corresponding to the years 2000 through 2015 have been assigned to the name `century_21`.

```
century_21 = movies_by_year.where('Year', are.above(1999))
```

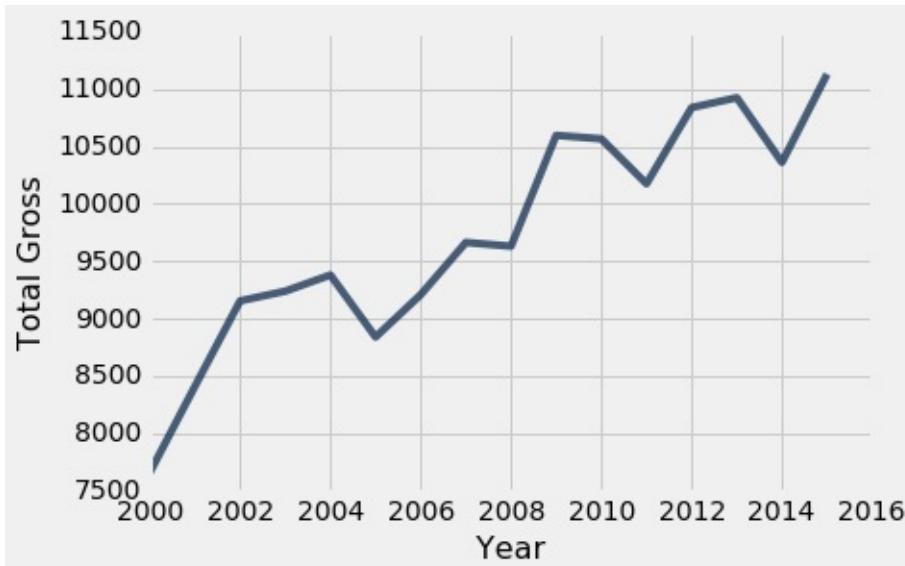
```
century_21.plot('Year', 'Number of Movies')
```



The global financial crisis of 2008 has a visible effect – in 2009 there is a sharp drop in the number of movies released.

The dollar figures, however, didn't suffer much.

```
century_21.plot('Year', 'Total Gross')
```



The total domestic gross receipt was higher in 2009 than in 2008, even though there was a financial crisis and a much smaller number of movies were released.

One reason for this apparent contradiction is that people tend to go to the movies when there is a recession. "[In Downturn, Americans Flock to the Movies](#)," said the New York Times in February 2009. The article quotes Martin Kaplan of the University of Southern California saying, "People want to forget their troubles, and they want to be with other people." When holidays and expensive treats are unaffordable, movies provide welcome entertainment and relief.

In 2009, another reason for high box office receipts was the movie Avatar and its 3D release. Not only was Avatar the #1 movie of 2009, it is also by some calculations the second highest grossing movie of all time, as we will see later.

```
century_21.where('Year', are.equal_to(2009))
```

Year	Total Gross	Number of Movies	#1 Movie
2009	10595.5	521	Avatar

[Interact](#)

Visualizing Categorical Distributions

Data come in many forms that are not numerical. Data can be pieces of music, or places on a map. They can also be categories into which you can place individuals. Here are some examples of *categorical* variables.

- The individuals are cartons of ice-cream, and the variable is the flavor in the carton.
- The individuals are professional basketball players, and the variable is the player's team.
- The individuals are years, and the variable is the genre of the highest grossing movie of the year.
- The individuals are survey respondents, and the variable is the response they choose from among "Not at all satisfied," "Somewhat satisfied," and "Very satisfied."

The table `icecream` contains data on 30 cartons of ice-cream.

```
icecream = Table().with_columns(
    'Flavor', make_array('Chocolate', 'Strawberry', 'Vanilla'),
    'Number of Cartons', make_array(16, 5, 9)
)
icecream
```

Flavor	Number of Cartons
Chocolate	16
Strawberry	5
Vanilla	9

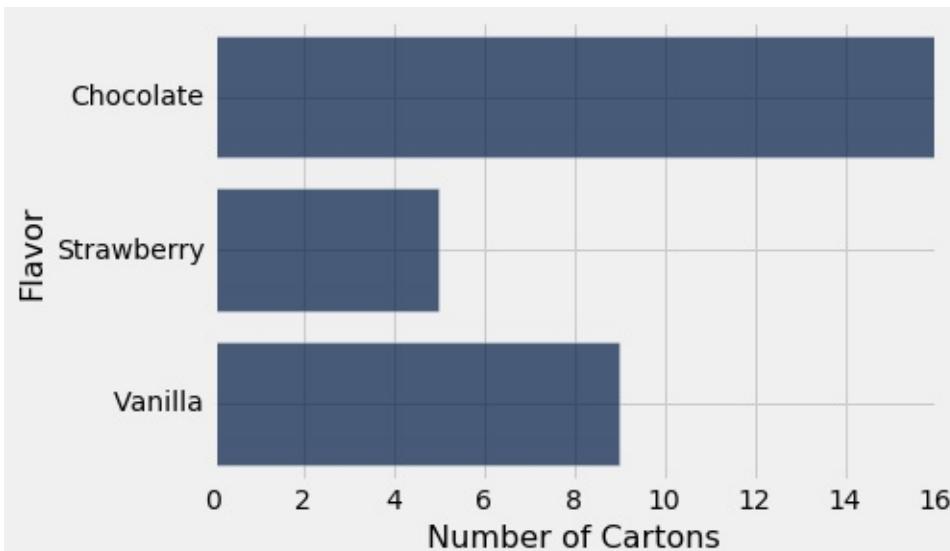
The *values* of the categorical variable "flavor" are chocolate, strawberry, and vanilla. The table shows the number of cartons of each flavor. We call this a *distribution table*. A *distribution* shows all the values of a variable, along with the frequency of each one.

Bar Chart

The bar chart is a familiar way of visualizing categorical distributions. It displays a bar for each category. The bars are equally spaced and equally wide. The length of each bar is proportional to the frequency of the corresponding category.

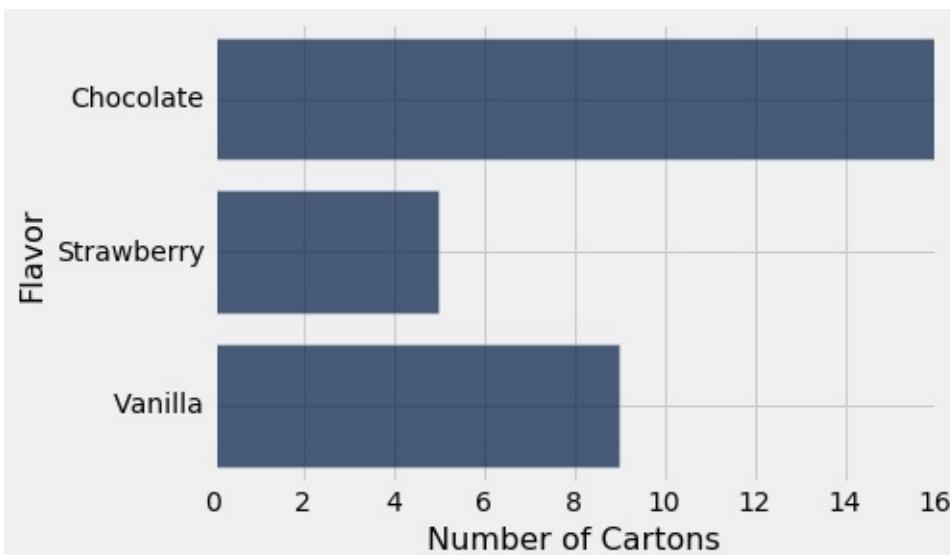
We will draw bar charts with horizontal bars because it's easier to label the bars that way. The `Table` method is therefore called `barm`. It takes two arguments: the first is the column label of the categories, and the second is the column label of the frequencies.

```
icecream.barm('Flavor', 'Number of Cartons')
```



If the table consists just of a column of categories and a column of frequencies, as in `icecream`, the method call is even simpler. You can just specify the column containing the categories, and `barm` will use the values in the other column as frequencies.

```
icecream.barm('Flavor')
```



Features of Categorical Distributions

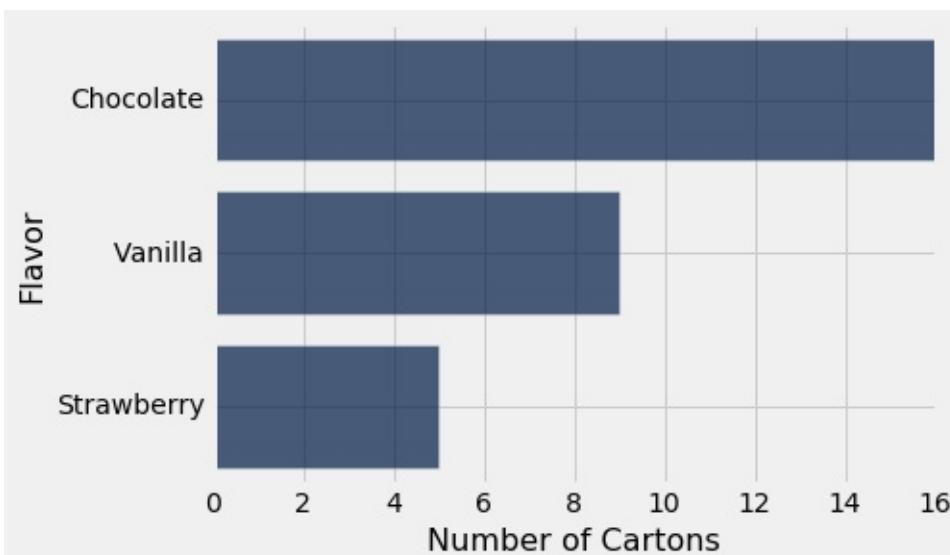
Apart from purely visual differences, there is an important fundamental distinction between bar charts and the two graphs that we saw in the previous sections. Those were the scatter plot and the line plot, both of which display two numerical variables – the variables on both axes are numerical. In contrast, the bar chart has categories on one axis and numerical frequencies on the other.

This has consequences for the chart. First, the width of each bar and the space between consecutive bars is entirely up to the person who is producing the graph, or to the program being used to produce it. Python made those choices for us. If you were to draw the bar graph by hand, you could make completely different choices and still have a perfectly correct bar graph, provided you drew all the bars with the same width and kept all the spaces the same.

Most importantly, the bars can be drawn in any order. The categories "chocolate," "vanilla," and "strawberry" have no universal rank order, unlike for example the numbers 5, 7, and 10.

This means that we can draw a bar chart that is easier to interpret, by rearranging the bars in decreasing order. To do this, we first rearrange the rows of `icecream` in decreasing order of `Number of Cartons`, and then draw the bar chart.

```
icecream.sort('Number of Cartons',
               descending=True).barh('Flavor')
```



This bar chart contains exactly the same information as the previous ones, but it is a little easier to read. While this is not a huge gain in reading a chart with just three bars, it can be quite significant when the number of categories is large.

Grouping Categorical Data

To construct the table `icecream`, someone had to look at all 30 cartons of ice-cream and count the number of each flavor. But if our data does not already include frequencies, we have to compute the frequencies before we can draw a bar chart. Here is an example where this is necessary.

The table `top` consists of U.S.A.'s top grossing movies 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 that produced the movie. The third contains the domestic box office gross in dollars, and the fourth contains the gross amount that would have been earned from ticket sales at 2016 prices. The fifth contains the release year of the movie.

There are 200 movies on the list. Here are the top ten according to unadjusted gross receipts.

```
top = Table.read_table('top_movies.csv')
top
```

Title	Studio	Gross	Gross (Adjusted)	Year
Star Wars: The Force Awakens	Buena Vista (Disney)	906723418	906723400	2015
Avatar	Fox	760507625	846120800	2009
Titanic	Paramount	658672302	1178627900	1997
Jurassic World	Universal	652270625	687728000	2015
Marvel's The Avengers	Buena Vista (Disney)	623357910	668866600	2012
The Dark Knight	Warner Bros.	534858444	647761600	2008
Star Wars: Episode I - The Phantom Menace	Fox	474544677	785715000	1999
Star Wars	Fox	460998007	1549640500	1977
Avengers: Age of Ultron	Buena Vista (Disney)	459005868	465684200	2015
The Dark Knight Rises	Warner Bros.	448139099	500961700	2012

... (190 rows omitted)

The Disney subsidiary Buena Vista shows up frequently in the top ten, as do Fox and Warner Brothers. Which studios will appear most frequently if we look among all 200 rows?

To figure this out, first notice that all we need is a table with the movies and the studios; the other information is unnecessary.

```
movies_and_studios = top.select('Title', 'Studio')
```

The Table method `group` allows us to count how frequently each studio appears in the table, by calling each studio a category and assigning each row to one category. The `group` method takes as its argument the label of the column that contains the categories, and returns a table of counts of rows in each category. The column of counts is always called `count`, but you can change that if you like by using `relabeled`.

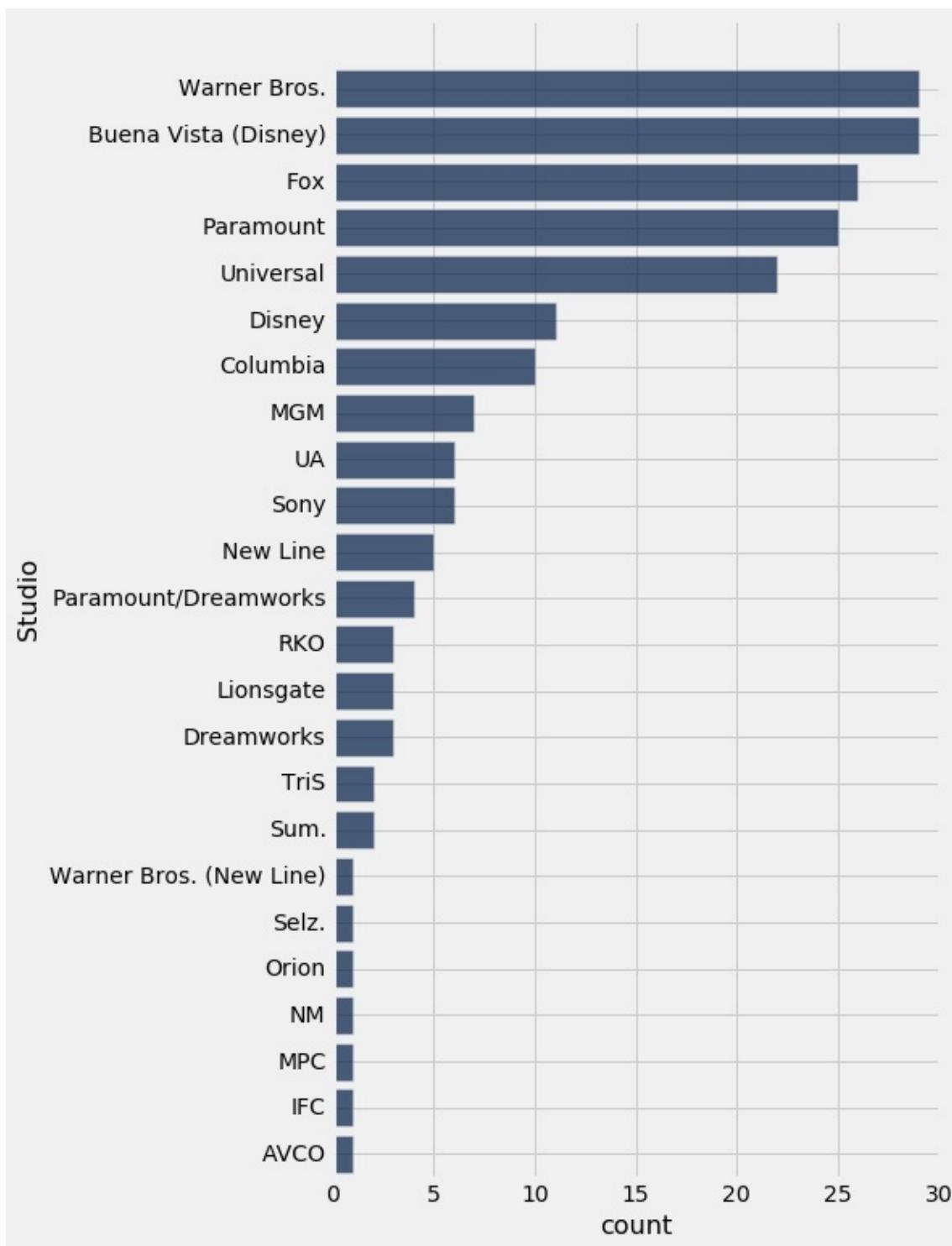
```
movies_and_studios.group('Studio')
```

Studio	count
AVCO	1
Buena Vista (Disney)	29
Columbia	10
Disney	11
Dreamworks	3
Fox	26
IFC	1
Lionsgate	3
MGM	7
MPC	1
... (14 rows omitted)	

Thus `group` creates a distribution table that shows how the movies are distributed among the categories (studios).

We can now use this table, along with the graphing skills that we acquired above, to draw a bar chart that shows which studios are most frequent among the 200 highest grossing movies.

```
studio_distribution = movies_and_studios.group('Studio')
studio_distribution.sort('count',
                           descending=True).barh('Studio')
```

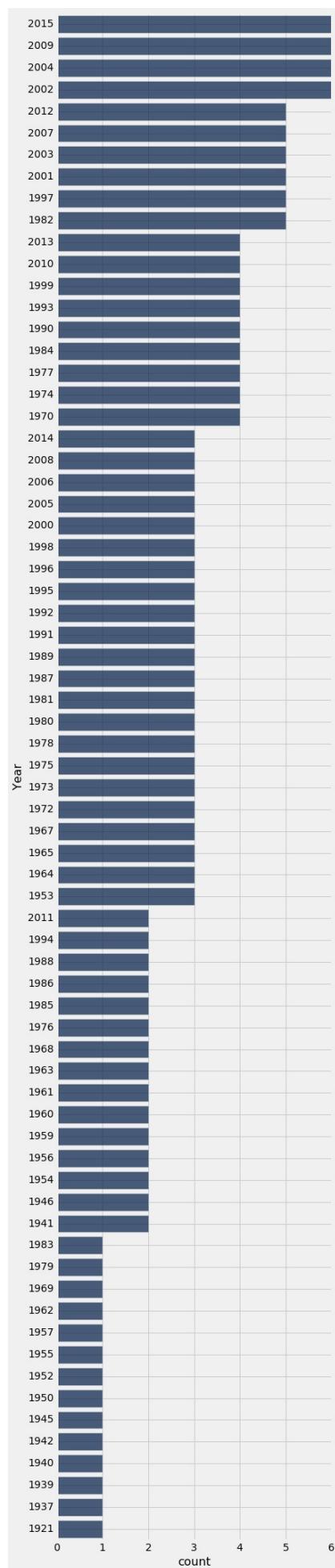


Warner Brothers and Buena Vista are the most common studios among the top 200 movies. Warner Brothers produces the Harry Potter movies and Buena Vista produces Star Wars.

Because total gross receipts are being measured in unadjusted dollars, it is not very surprising that the top movies are more frequently from recent years than from bygone decades. In absolute terms, movie tickets cost more now than they used to, and thus gross receipts are higher. This is borne out by a bar chart that show the distribution of the 200 movies by year of release.

```
movies_and_years = top.select('Title', 'Year')
movies_and_years.group('Year').sort('count',
descending=True).barh('Year')
```

7.1 Categorical Distributions



All of the longest bars correspond to years after 2000. This is consistent with our observation that recent years should be among the most frequent.

Towards numerical variables

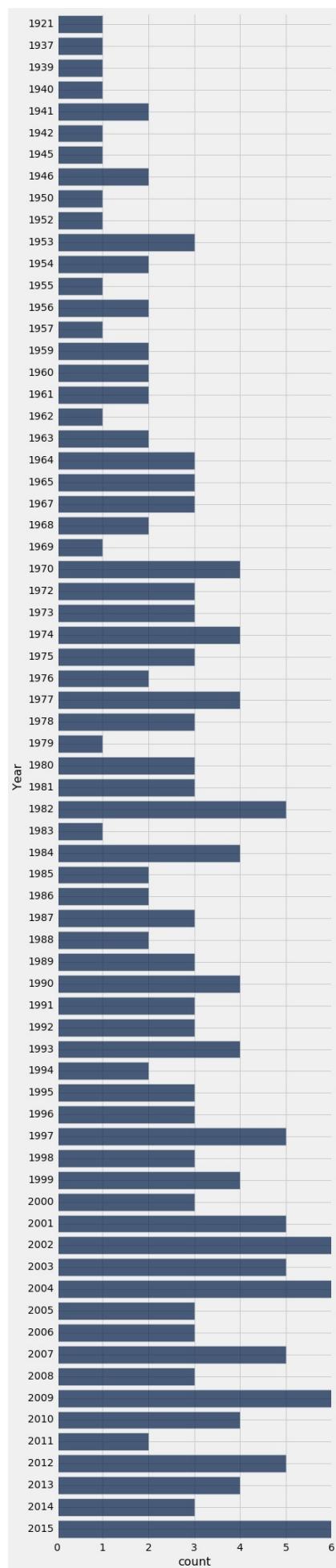
There is something unsettling about this chart. Though it does answer the question of which release years appear most frequently among the 200 top grossing movies, it doesn't list all the years in chronological order. It is treating `Year` as a categorical variable.

But years are fixed chronological units that do have an order. They also have fixed numerical spacings relative to each other. Let's see what happens when we try to take that into account.

By default, `barh` sorts the categories (years) from lowest to highest. So we will run the code without sorting by count.

```
movies_and_years.groupby('Year').barh('Year')
```

7.1 Categorical Distributions



Now the years are in increasing order. But there is still something disquieting about this bar chart. The bars at 1921 and 1937 are just as far apart from each other as the bars at 1937 and 1939. The bar chart doesn't show that none of the 200 movies were released in the years 1922 through 1936, nor in 1938. Such inconsistencies and omissions make the distribution in the early years hard to understand based on this visualization.

Bar charts are intended as visualizations of categorical variables. When the variable is numerical, the numerical relations between its values have to be taken into account when we create visualizations. That is the topic of the next section.

[Interact](#)

Visualizing Numerical Distributions

Many of the variables that data scientists study are *quantitative* or *numerical*. Their values are numbers on which you can perform arithmetic. Examples that we have seen include the number of periods in chapters of a book, the amount of money made by movies, and the age of people in the United States.

The values of a categorical variable can be given numerical codes, but that doesn't make the variable quantitative. In the example in which we studied Census data broken down by age group, the categorial variable `SEX` had the numerical codes `1` for 'Male,' `2` for 'Female,' and `0` for the aggregate of both groups `1` and `2`. While 0, 1, and 2 are numbers, in this context it doesn't make sense to subtract 1 from 2, or take the average of 0, 1, and 2, or perform other arithmetic on the three values. `SEX` is a categorical variable even though the values have been given a numerical code.

For our main example, we will return to a dataset that we studied when we were visualizing categorical data. It is the table `top`, which consists of data from U.S.A.'s top grossing movies of all time. For convenience, here is the description of the table again.

The first column contains the title of the movie. The second column contains the name of the studio that produced the movie. The third contains the domestic box office gross in dollars, and the fourth contains the gross amount that would have been earned from ticket sales at 2016 prices. The fifth contains the release year of the movie.

There are 200 movies on the list. Here are the top ten according to the unadjusted gross receipts in the column `Gross`.

```
top = Table.read_table('top_movies.csv')
# Make the numbers in the Gross and Gross (Adjusted) columns
# look nicer:
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)

Visualizing the Distribution of the Adjusted Receipts

In this section we will draw graphs of the distribution of the numerical variable in the column `Gross (Adjusted)`. For simplicity, let's create a smaller table that has the information that we need. And since three-digit numbers are easier to work with than nine-digit numbers, let's measure the `Adjusted Gross` receipts in millions of dollars. Note how `round` is used to retain only two decimal places.

```
millions = top.select(0).with_column('Adjusted Gross',
                                      np.round(top.column(3)/1e6,
                                               2))
millions
```

Title	Adjusted 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

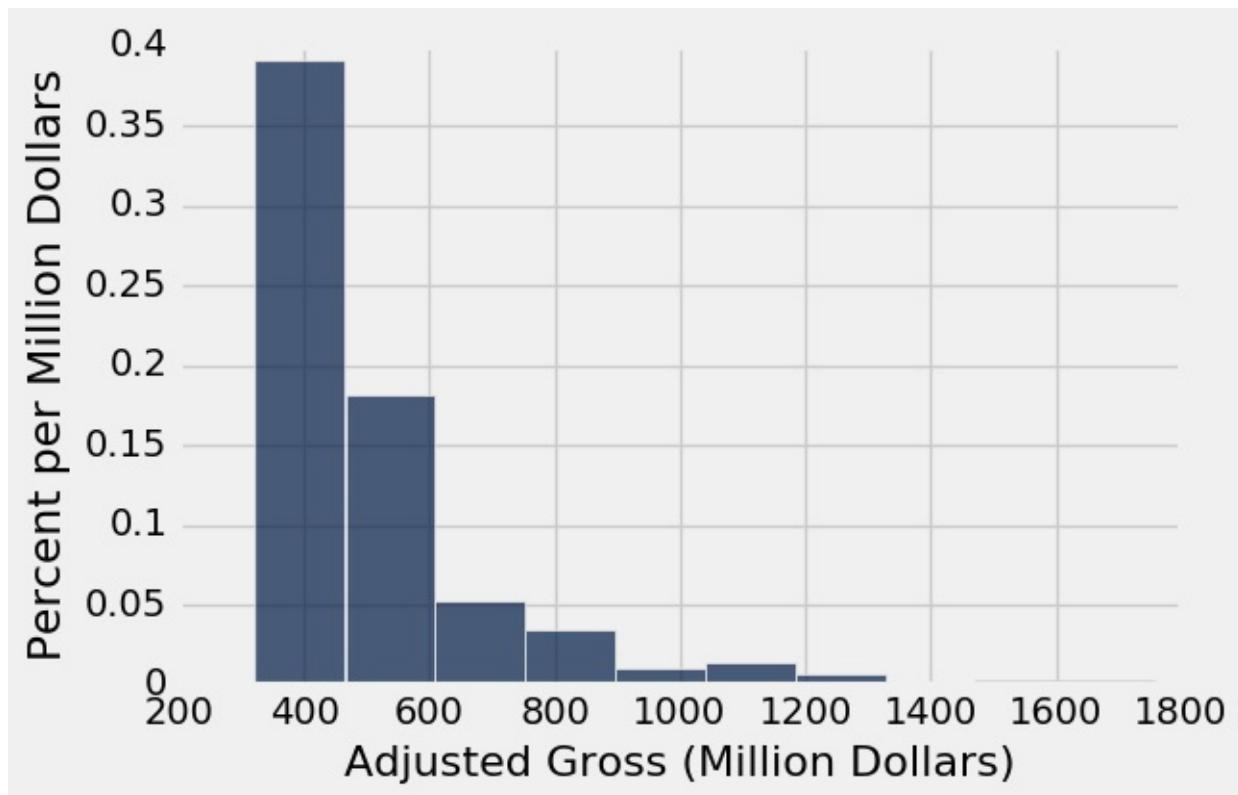
... (190 rows omitted)

A Histogram

A *histogram* of a numerical dataset looks very much like a bar chart, though it has some important differences that we will examine in this section. First, let's just draw a histogram of the adjusted receipts.

The `hist` method generates a histogram of the values in a column. The optional `unit` argument is used in the labels on the two axes. The histogram shows the distribution of the adjusted gross amounts, in millions of 2016 dollars.

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



The Horizontal Axis

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, `hist` does have to account for situations where there might have been values at the edges. So `hist` has an *endpoint convention*: bins include the data at their left endpoint, but not the data at their right endpoint.

We will use the notation $[a, b)$ for the bin that starts at a and ends at b but doesn't include b .

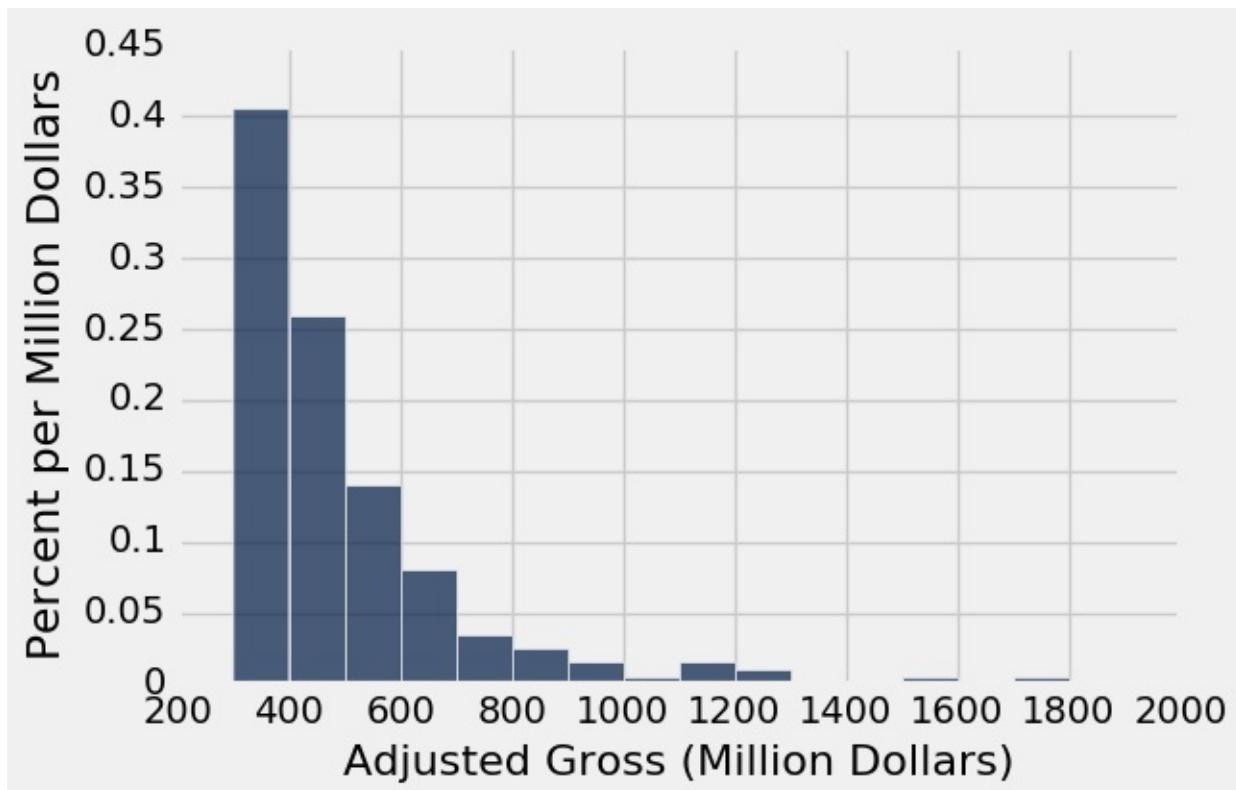
Sometimes, adjustments have to be made in the first or last bin, to ensure that the smallest and largest values of the variable are included. You saw an example of such an adjustment in the Census data studied earlier, where an age of "100" years actually meant "100 years old or older."

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 none of the movies 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 ends of the bins are situated. For example, it is not easy to pinpoint exactly where the value 500 lies on the horizontal axis. So it is hard to judge exactly where one bar ends and the next begins.

The optional argument `bins` can be used with `hist` to specify the endpoints of the bins. It must consist of a sequence of numbers that starts with the left end of the first bin and ends with the right end of the last bin. We will start by setting the numbers in `bins` to be 300, 400, 500, and so on, ending with 2000.

```
millions.hist('Adjusted Gross', bins=np.arange(300,2001,100),
unit="Million Dollars")
```



The horizontal axis of this figure is easier to read. The labels 200, 400, 600, and so on are centered at the corresponding values. The tallest bar is for movies that grossed between 300 million and 400 million dollars.

A very small number of movies grossed 800 million dollars or more. 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 in large populations also often have this kind of shape.

The Counts in the Bins

The counts of values in the bins 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. The first column lists the left endpoints of the bins (but see the note about the final value, below). The second column contains the counts of all values

in the `Adjusted Gross` column that are in the corresponding bin. That is, it counts all the `Adjusted Gross` values that are greater than or equal to the value in `bin`, but less than the next value in `bin`.

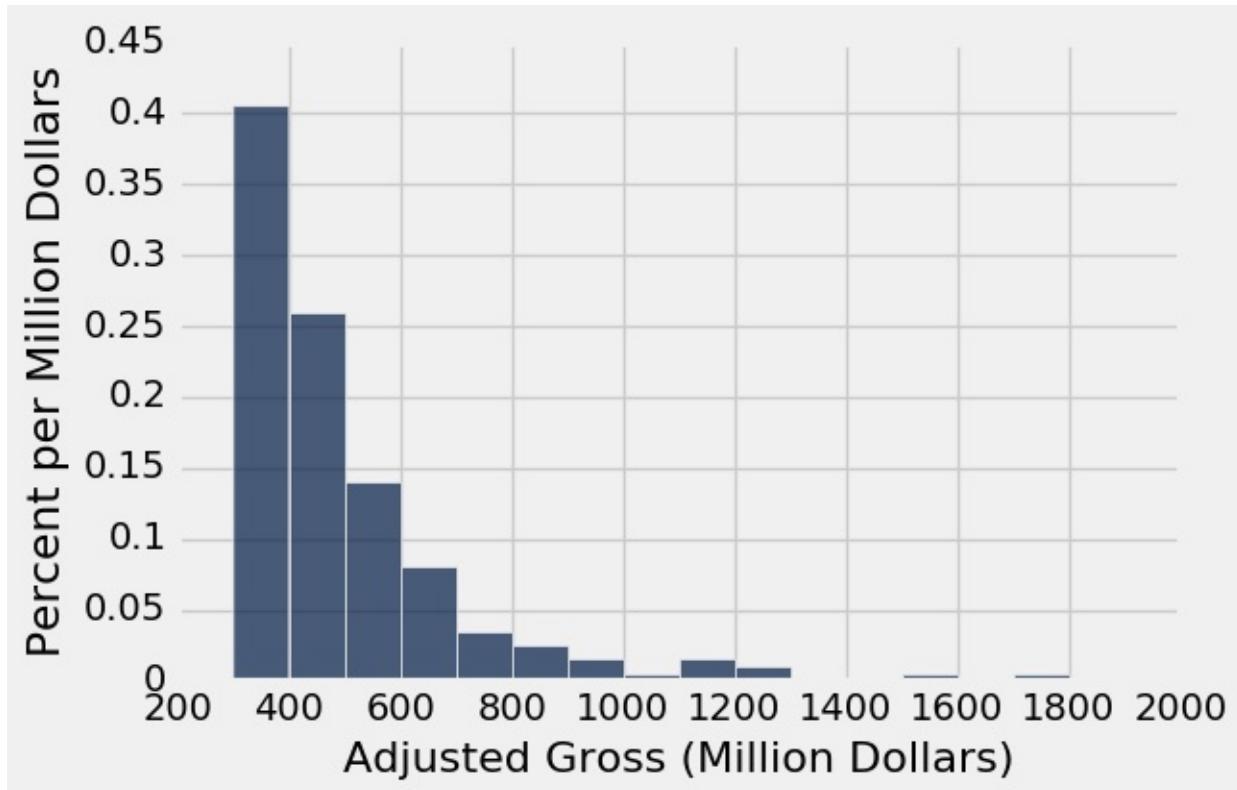
```
bin_counts = millions.bin('Adjusted Gross',
bins=np.arange(300, 2001, 100))
bin_counts.show()
```

bin	Adjusted Gross count
300	81
400	52
500	28
600	16
700	7
800	5
900	3
1000	1
1100	3
1200	2
1300	0
1400	0
1500	1
1600	0
1700	1
1800	0
1900	0
2000	0

Notice the `bin` value 2000 in the last row. That's not the left end-point of any bar – it's the right end point of the last bar. By the endpoint convention, the data there are not included. So the corresponding `count` is recorded as 0, and would have been recorded as 0 even if there had been movies that made more than \$2,000\$ million dollars. When either `bin` or `hist` is called with a `bins` argument, the graph only considers values that are in the specified bins.

Once values have been binned, the resulting counts can be used to generate a histogram using the `bin_column` named argument to specify which column contains the bin lower bounds.

```
bin_counts.hist('Adjusted Gross count', bin_column='bin',
unit='Million Dollars')
```



The Vertical Axis: Density Scale

The horizontal axis of a histogram is straightforward to read, once we have taken care of details like the ends of the bins. The features of the vertical axis require a little more attention. We will go over them one by one.

Let's start by examining how to calculate the numbers on the vertical axis. If the calculation seems a little strange, have patience – the rest of the section will explain the reasoning.

Calculation. The height of each bar is the percent of elements that fall into the corresponding bin, relative to the width of the bin.

```

counts = bin_counts.relabeled('Adjusted Gross count', 'Count')
percents = counts.with_column(
    'Percent', (counts.column('Count')/200)*100
)
heights = percents.with_column(
    'Height', percents.column('Percent')/100
)
heights

```

bin	Count	Percent	Height
300	81	40.5	0.405
400	52	26	0.26
500	28	14	0.14
600	16	8	0.08
700	7	3.5	0.035
800	5	2.5	0.025
900	3	1.5	0.015
1000	1	0.5	0.005
1100	3	1.5	0.015
1200	2	1	0.01

... (8 rows omitted)

Go over the numbers on the vertical axis of the histogram above to check that the column `Heights` looks correct.

The calculations will become clear if we just examine the first row of the table.

Remember that there are 200 movies in the dataset. The [300, 400) bin contains 81 movies. That's 40.5% of all the movies:

$$\text{Percent} = \frac{81}{200} \cdot 100 = 40.5$$

The width of the [300, 400) bin is $400 - 300 = 100$. So

$$\text{Height} = \frac{40.5}{100} = 0.405$$

The code for calculating the heights used the facts that there are 200 movies in all and that the width of each bin is 100.

Units. The height of the bar is 40.5% divided by 100 million dollars, and so the height is 0.405% per million dollars.

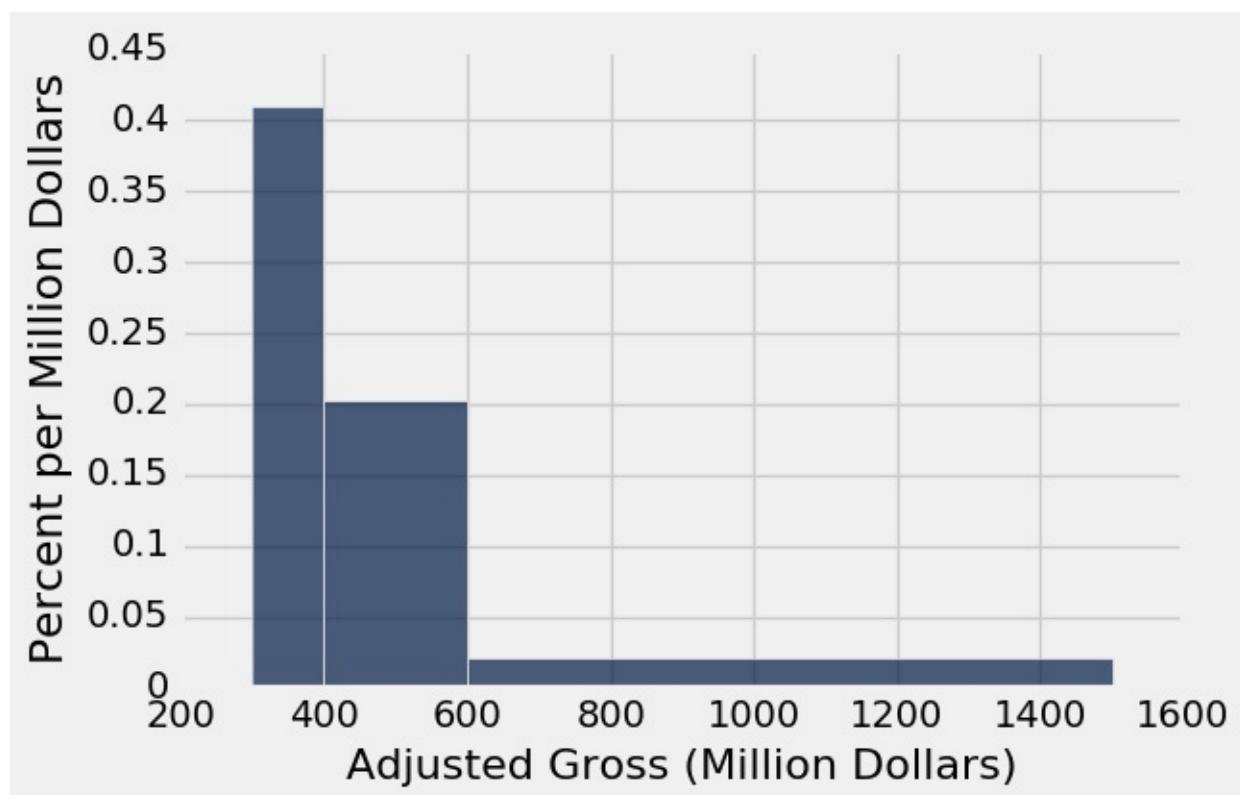
This method of drawing histograms creates a vertical axis that is said to be *on the density scale*. The height of bar is **not** the percent of entries in the bin; it is the percent of entries in the bin relative to the amount of space in the bin. That is why the height measures crowdedness or *density*.

Let's see why this matters.

Unequal Bins

An advantage of the histogram over a bar chart is that a histogram can contain bins of unequal width. Below, the values in the `Millions` column are binned into three uneven categories.

```
uneven = make_array(300, 400, 600, 1500)
millions.hist('Adjusted Gross', bins=uneven, unit="Million
Dollars")
```



Here are the counts in the three bins.

```
millions.bin('Adjusted Gross', bins=uneven)
```

bin	Adjusted Gross count
300	81
400	80
600	37
1500	0

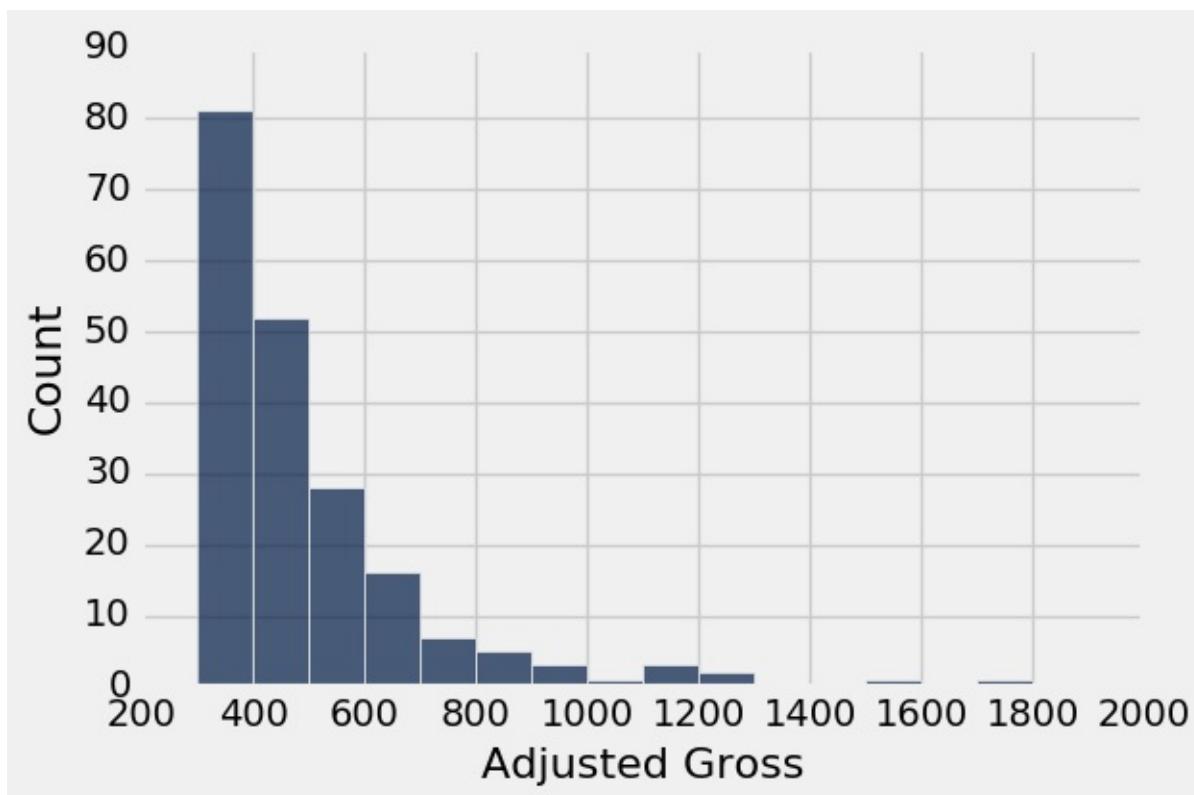
Although the ranges [300, 400) and [400, 600) have nearly identical counts, the bar over the former is twice as tall as the latter because it is only half as wide. The density of values in the [300, 400) is twice as much as the density in [400, 600).

Histograms help us visualize where on the number line the data are most concentrated, especially when the bins are uneven.

The Problem with Simply Plotting Counts

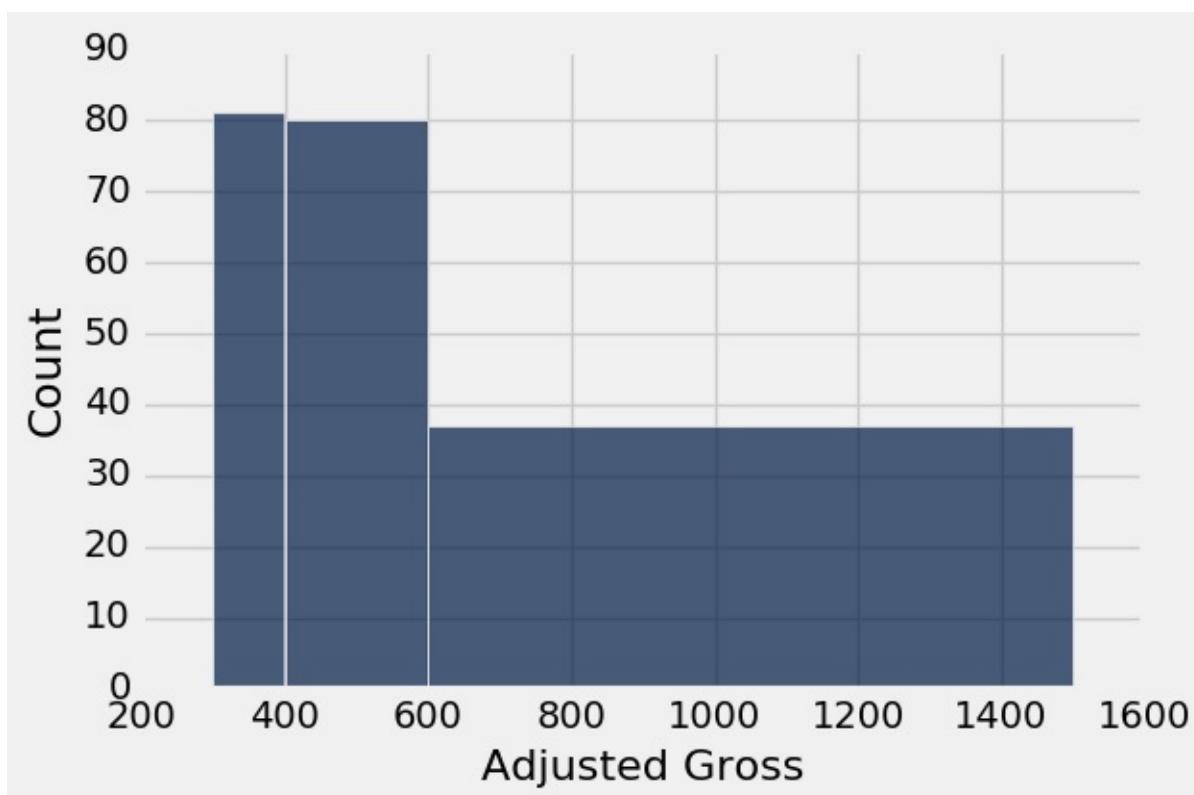
It is possible to display counts directly in a chart, using the `normed=False` option of the `hist` method. The resulting chart has the same shape as a histogram when the bins all have equal widths, though the numbers on the vertical axis are different.

```
millions.hist('Adjusted 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.

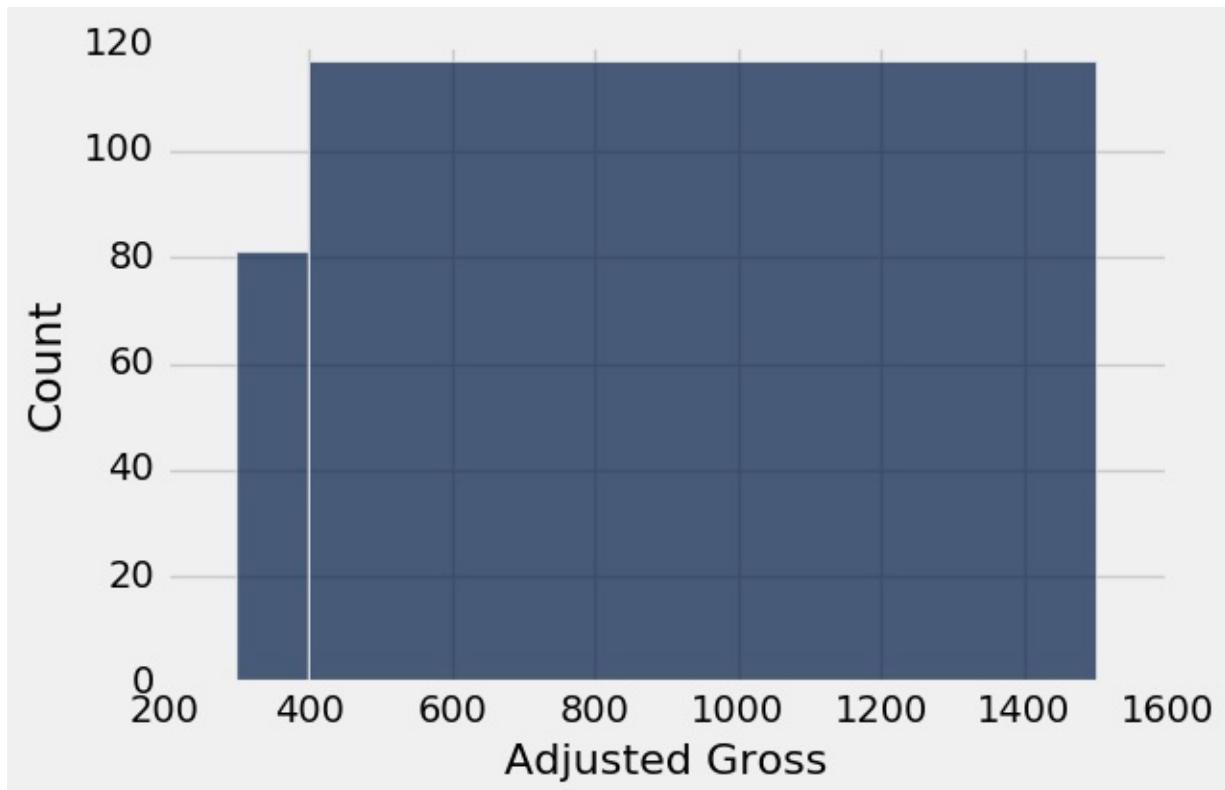
```
millions.hist('Adjusted Gross', bins=uneven, normed=False)
```



Even though the method used is called `hist`, **the figure above is NOT A HISTOGRAM**. It misleadingly exaggerates the proportion of movies grossing at least 600 million dollars. The height of each bar is simply plotted at the number of movies in the bin, *without accounting for the difference in the widths of the bins*.

The picture becomes even more absurd if the last two bins are combined.

```
very_uneven = make_array(300, 400, 1500)
millions.hist('Adjusted Gross', bins=very_uneven, normed=False)
```



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

The Histogram: General Principles and Calculation

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

That is why a histogram has two defining properties:

1. The bins are drawn to scale and are contiguous (though some might be empty), because the values on the horizontal axis are numerical.
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:

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

The calculation of the heights just uses the fact that the bar is a rectangle:

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

and so

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

The units of height are "percent per unit on the horizontal axis."

When drawn using this method, the histogram is said to be drawn on the density scale. On this scale:

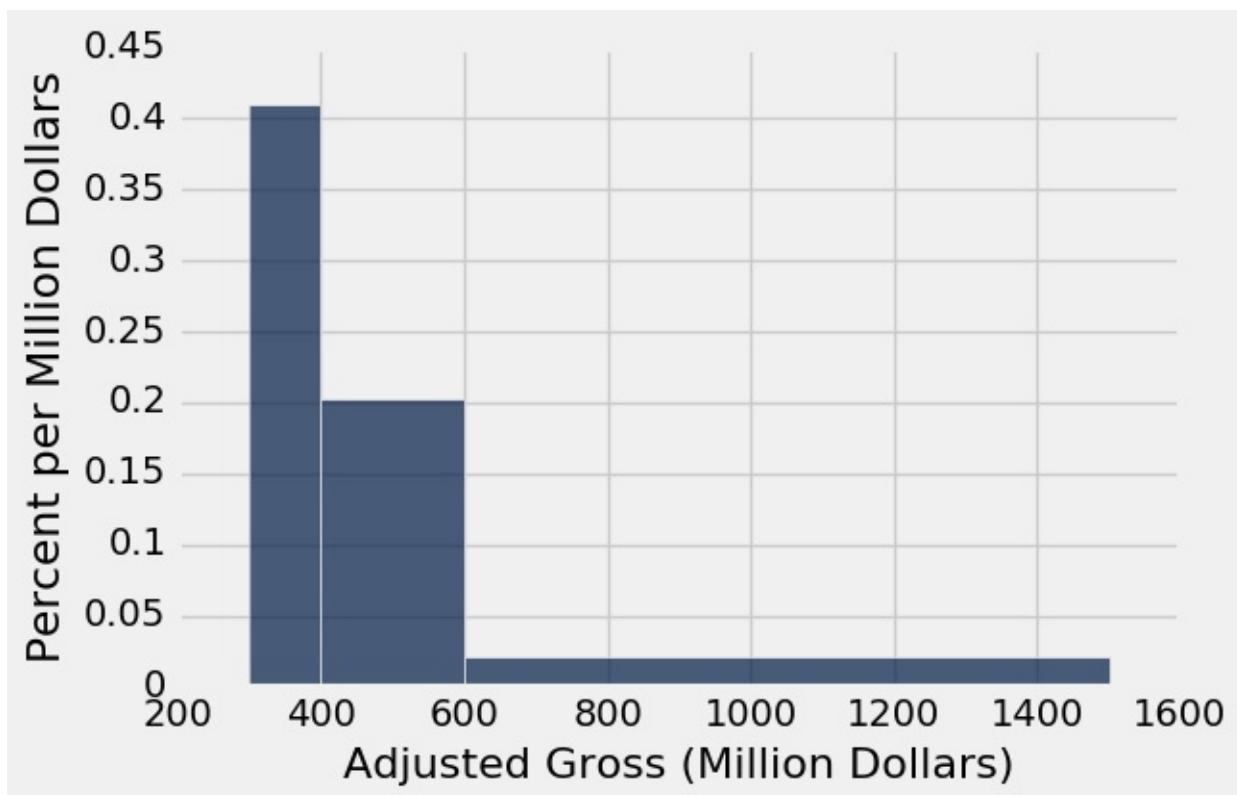
- The area of each bar is equal to the percent of data values that are in the corresponding bin.
- The total area of all the bars in the histogram is 100%. Speaking in terms of proportions, we say that the areas of all the bars in a histogram "sum to 1".

Flat Tops and the Level of Detail

Even though the density scale correctly represents percents using area, some detail is lost by grouping values into bins.

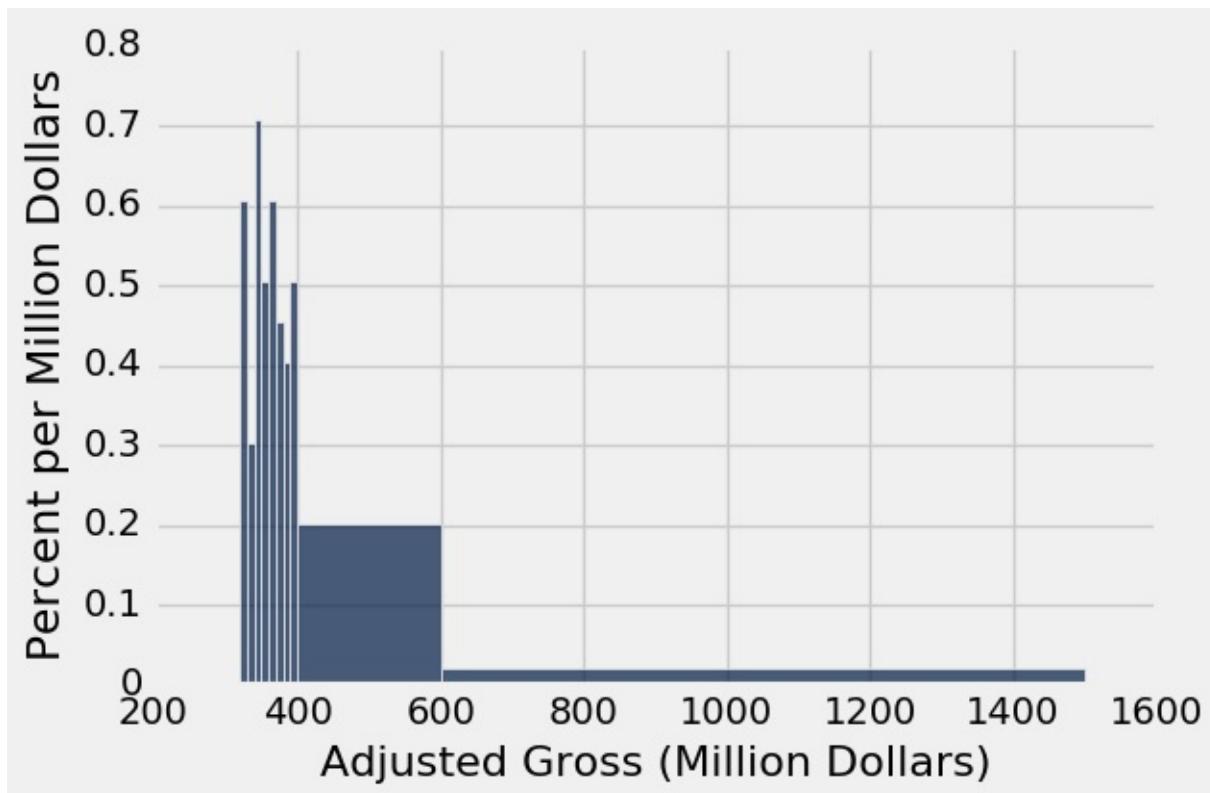
Take another look at the [300, 400) bin in the figure below. The flat top of the bar, at the level 0.405% per million dollars, hides the fact that the movies are somewhat unevenly distributed across that bin.

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



To see this, let us split the [300, 400) bin into 10 narrower bins, each of width 10 million dollars.

```
some_tiny_bins = make_array(300, 310, 320, 330, 340, 350, 360,
370, 380, 390, 400, 600, 1500)
millions.hist('Adjusted Gross', bins=some_tiny_bins,
unit='Million Dollars')
```



Some of the skinny bars are taller than 0.405 and others are shorter; the first two have heights of 0 because there are no data between 300 and 320. By putting a flat top at the level 0.405 across the whole bin, we are deciding to ignore the finer detail and are using the flat level as a rough approximation. Often, though not always, this is sufficient for understanding the general shape of the distribution.

The height as a rough approximation. This observation gives us a different way of thinking about the height. Look again at the [300, 400) bin in the earlier histograms. As we have seen, the bin is 100 million dollars wide and contains 40.5% of the data. Therefore the height of the corresponding bar is 0.405% per million dollars.

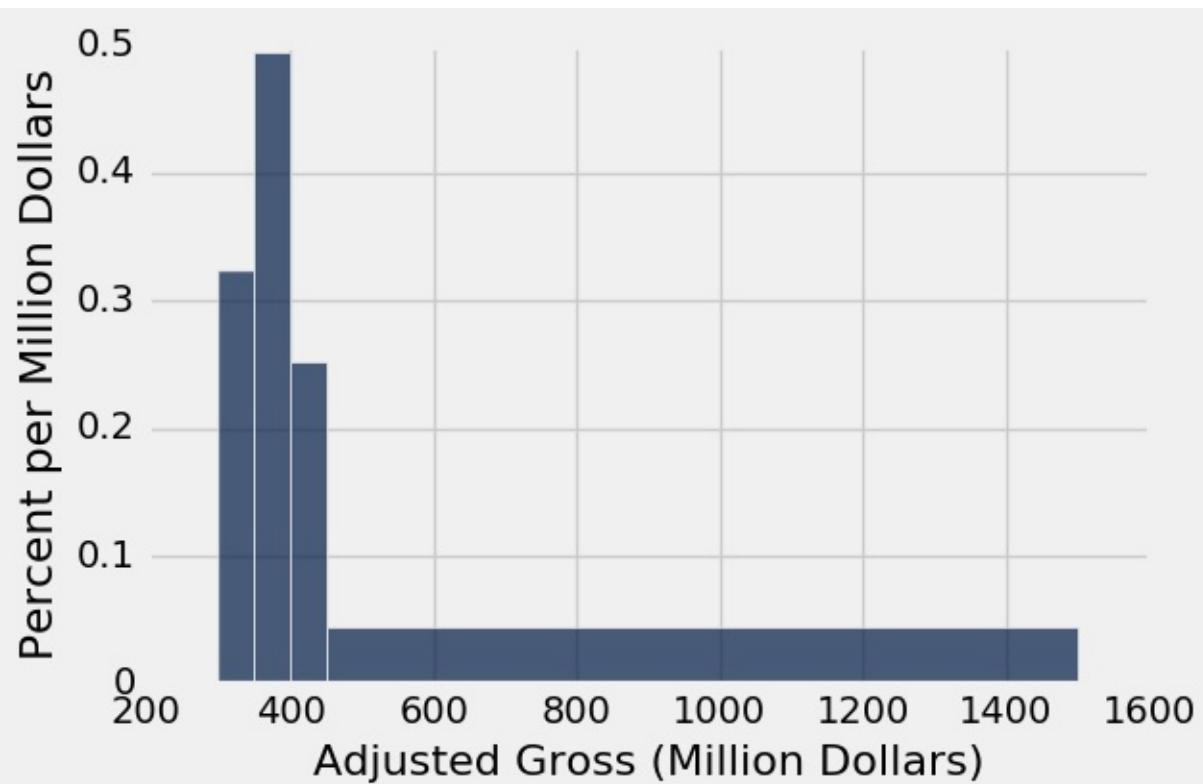
Now think of the bin as consisting of 100 narrow bins that are each 1 million dollars wide. The bar's height of "0.405% per million dollars" means that as a rough approximation, 0.405% of the movies are in each of those 100 skinny bins of width 1 million dollars.

Notice that because we have the entire dataset that is being used to draw the histograms, we can draw the histograms to 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" created by the flat tops.

Histograms Q&A

Let's draw the histogram again, this time with four bins, and check our understanding of the concepts.

```
uneven_again = make_array(300, 350, 400, 450, 1500)
millions.hist('Adjusted Gross', bins=uneven_again, unit='Million
Dollars')
```



```
millions.bin('Adjusted Gross', bins=uneven_again)
```

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

Look again at the histogram, and 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 of space in the bin, not the number of movies in the bin. The [450, 1500) bin does have more movies than the [400, 450) bin, but it is also a whole lot wider. So it is less crowded. The density of movies in it is much lower.

Differences Between Bar Charts and Histograms

- Bar charts display one quantity per category. They are often used to display the distributions of categorical variables. Histograms display the distributions of quantitative variables.
- All the bars in a bar chart have the same width, and there is an equal amount of space between consecutive bars. The bars of a histogram can have different widths, and they are contiguous.
- The lengths (or heights, if the bars are drawn vertically) of the bars in a bar chart are proportional to the value for each category. The heights of bars in a histogram measure densities; the *areas* of bars in a histogram are proportional to the numbers of entries in the bins.

[Interact](#)

Overlaid Graphs

In this chapter, we have learned how to visualize data by drawing graphs. A common use of such visualizations is to compare two datasets. In this section, we will see how to *overlay* plots, that is, draw them in a single graphic on a common pair of axes.

For the overlay to make sense, the graphs that are being overlaid must represent the same variables and be measured in the same units.

To draw overlaid graphs, the methods `scatter`, `plot`, and `barh` can all be called in the same way. For `scatter` and `plot`, one column must serve as the common horizontal axis for all the overlaid graphs. For `barh`, one column must serve as the common axis which is the set of categories. The general call looks like:

```
name_of_table.method(column_label_of_common_axis, array_of_labels_of_variables_to_plot)
```

More commonly, we will first select only the columns needed for our graph, and then call the method by just specifying the variable on the common axis:

```
name_of_table.method(column_label_of_common_axis)
```

Scatter Plots

[Francis Galton](#) (1822-1911) was an English polymath who was a pioneer in the analysis of relations between numerical variables. He was particularly interested in the controversial area of eugenics – indeed, he coined that term – which involves understanding how physical traits are passed down from one generation to the next.

Galton meticulously collected copious amounts of data, some of which we will analyze in this course. Here is a subset of Galton's data on heights of parents and their children.

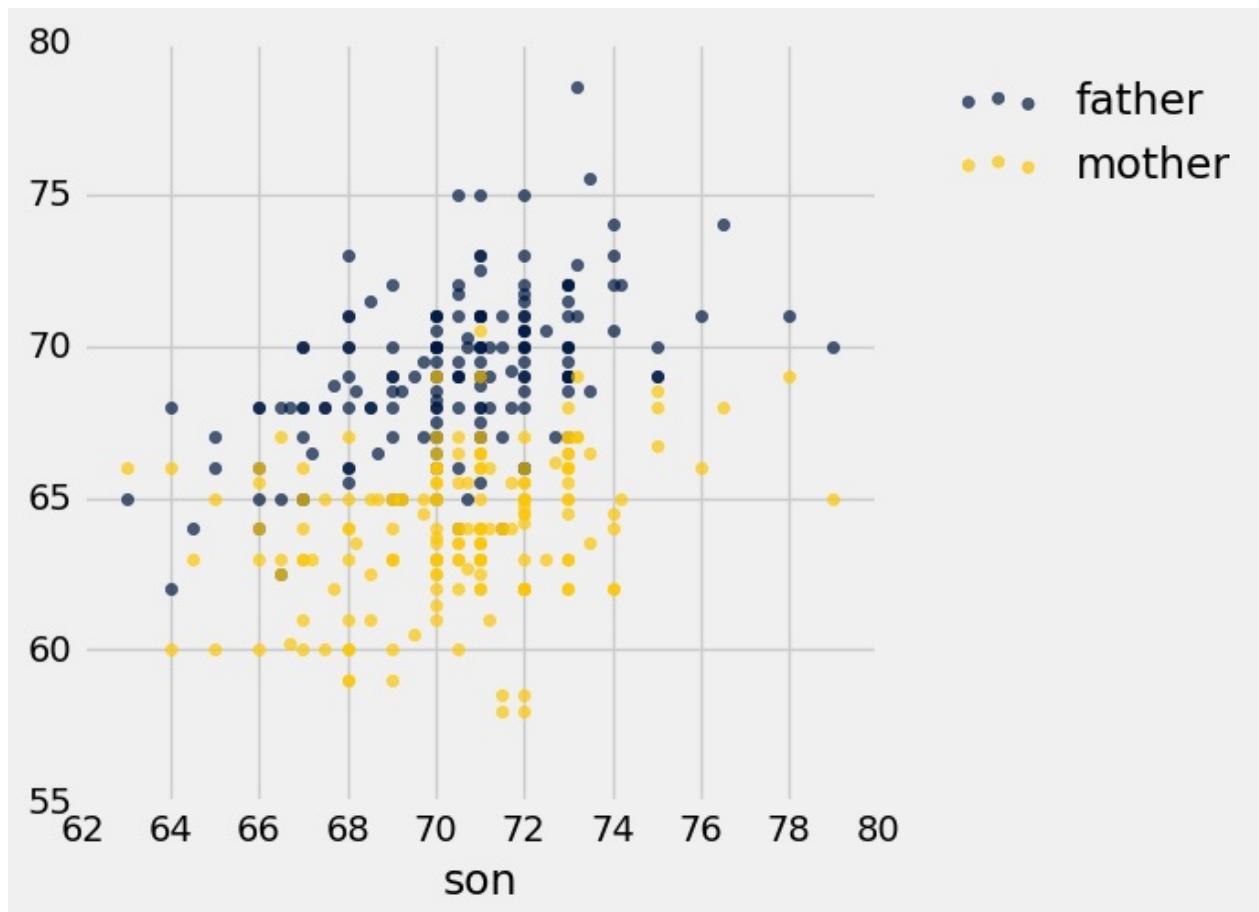
Specifically, the population consists of 179 men who were the first-born in their families. The data are their own heights and the heights of their parents. All heights were measured in inches.

```
heights = Table.read_table('galton_subset.csv')
heights
```

father	mother	son
78.5	67	73.2
75.5	66.5	73.5
75	64	71
75	64	70.5
75	58.5	72
74	68	76.5
74	62	74
73	67	71
73	67	68
73	66.5	71
... (169 rows omitted)		

The `scatter` method allows us to visualize how the sons' heights are related to the heights of both their parents. In the graph, the sons' heights will form the common horizontal axis.

```
heights.scatter('son')
```



Notice how we only specified the variable (sons' heights) on the common horizontal axis. Python drew two scatter plots: one each for the relation between this variable and the other two.

Both the gold and the blue scatter plots slope upwards and show a positive association between the sons' heights and the heights of both their parents. The blue (fathers) plot is in general higher than the gold, because the fathers were in general taller than the mothers.

Line Plots

Our next example involves data on children of more recent times. We will return to the Census data table `us_pop`, created below again for reference. From this, we will extract the counts of all children in each of the age categories 0 through 18 years.

```
# Read the full Census table
census_url = 'http://www2.census.gov/programs-
surveys/popest/datasets/2010-2015/national/asrh/nc-est2015-
agesex-res.csv'
full_census_table = Table.read_table(census_url)

# Select columns from the full table and relabel some of them
partial_census_table = full_census_table.select(['SEX', 'AGE',
'POPESTIMATE2010', 'POPESTIMATE2014'])
us_pop = partial_census_table.relabelled('POPESTIMATE2010',
'2010').relabelled('POPESTIMATE2014', '2014')

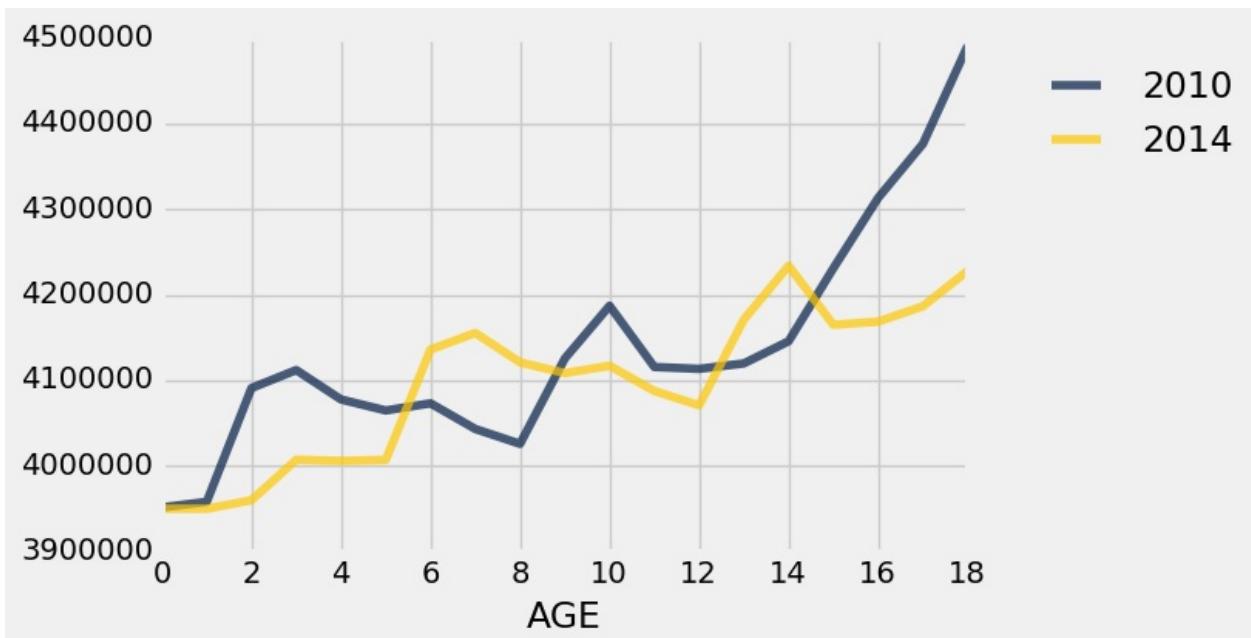
# Access the rows corresponding to all children, ages 0-18
children = us_pop.where('SEX', are.equal_to(0)).where('AGE',
are.below(19)).drop('SEX')
children.show()
```

AGE	2010	2014
0	3951330	3949775
1	3957888	3949776
2	4090862	3959664
3	4111920	4007079
4	4077551	4005716
5	4064653	4006900
6	4073013	4135930
7	4043046	4155326
8	4025604	4120903
9	4125415	4108349
10	4187062	4116942
11	4115511	4087402
12	4113279	4070682
13	4119666	4171030
14	4145614	4233839
15	4231002	4164796
16	4313252	4168559
17	4376367	4186513
18	4491005	4227920

We can now draw two overlaid line plots, showing the numbers of children in the different age groups for each of the years 2010 and 2014. The method call is analogous to the

`scatter` call in the previous example.

```
children.plot('AGE')
```



On this scale, it's important to remember that we only have data at ages 0, 1, 2, and so on; the graphs "join the dots" in between.

The graphs cross each other in a few places: for example, there were more 4-year-olds in 2010 than in 2014, and there were more 14-year-olds in 2014 than in 2010.

Of course, the 14-year-olds in 2014 mostly consist of the 10-year-olds in 2010. To see this, look at the gold graph at AGE 14 and the blue graph at AGE 10. Indeed, you will notice that the entire gold graph (2014) looks like the blue graph (2010) slid over to the right by 4 years. The slide is accompanied by a slight rise due to the net effect of children who entered the country between 2010 and 2014 outnumbering those who left. Fortunately at these ages there is not much loss of life.

Bar Charts

For our final example of this section, we look at distributions of ethnicities of adults and children in California as well as in the entire United States.

The Kaiser Family Foundation has complied Census data on the distribution of race and ethnicity in the U.S. The Foundation's website provides compilations of data for [the entire U.S. population](#) in 2014, as well as for [U.S. children](#) who were younger than 18 years old that year.

Here is a table adapted from their data for the United States and California. The columns represent everyone in the U.S.A., everyone in California, children in the U.S.A., and children in California. The body of the table contains proportions in the different categories. Each column shows the distribution of ethnicities in the group of people corresponding to that column. So in each column, the entries add up to 1.

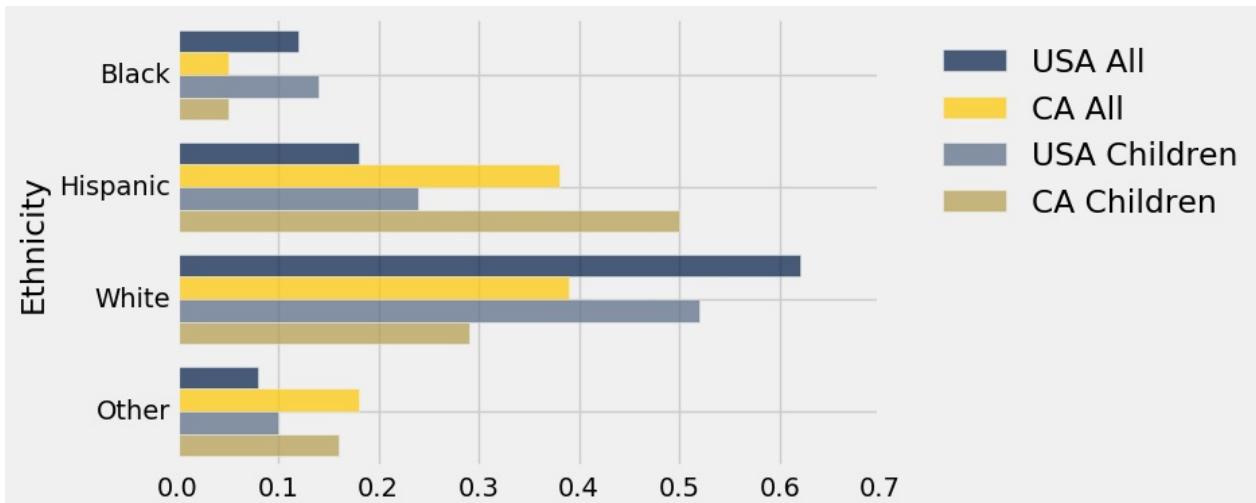
```
usa_ca = Table.read_table('usa_ca_2014.csv')
usa_ca
```

Ethnicity	USA All	CA All	USA Children	CA Children
Black	0.12	0.05	0.14	0.05
Hispanic	0.18	0.38	0.24	0.5
White	0.62	0.39	0.52	0.29
Other	0.08	0.18	0.1	0.16

It is natural to want to compare these distributions. It makes sense to compare the columns directly, because all the entries are proportions and are therefore on the same scale.

The method `barh` allows us to visualize the comparisons by drawing multiple bar charts on the same axes. The call is analogous to those for `scatter` and `plot`: we have to specify the common axis of categories.

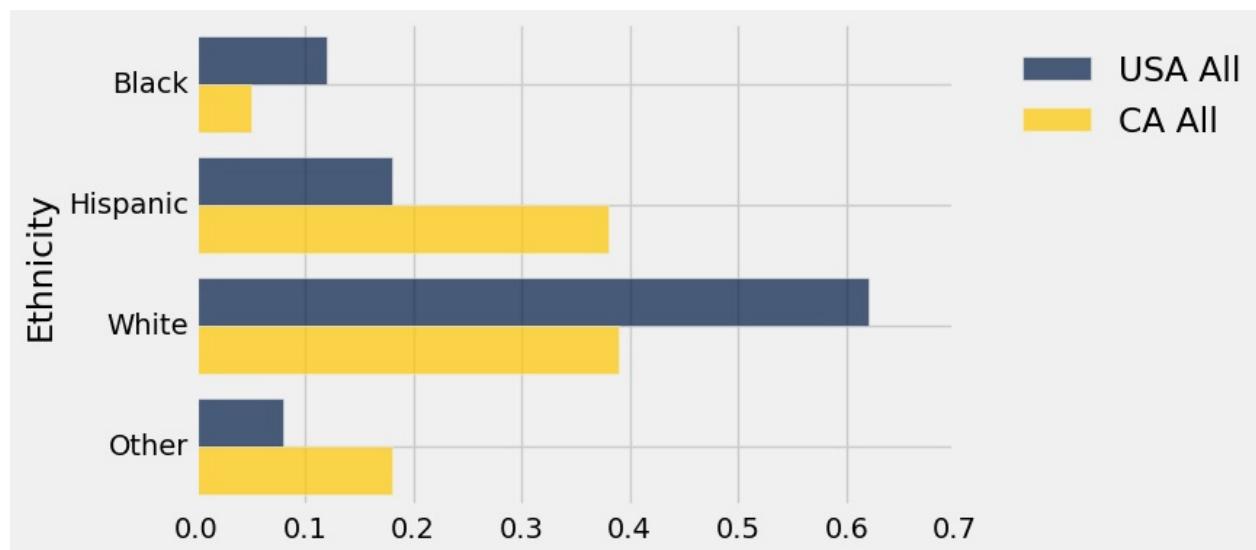
```
usa_ca.barh('Ethnicity')
```



While drawing the overlaid bar charts is straightforward, there is a bit too much information on this graph for us to be able to sort out similarities and differences between populations. It seems clear that the distributions of ethnicities for everyone in the U.S. and for children in the U.S. are more similar to each other than any other pair, but it's much easier to compare the populations one pair at a time.

Let's start by comparing the entire populations of the U.S.A. and California.

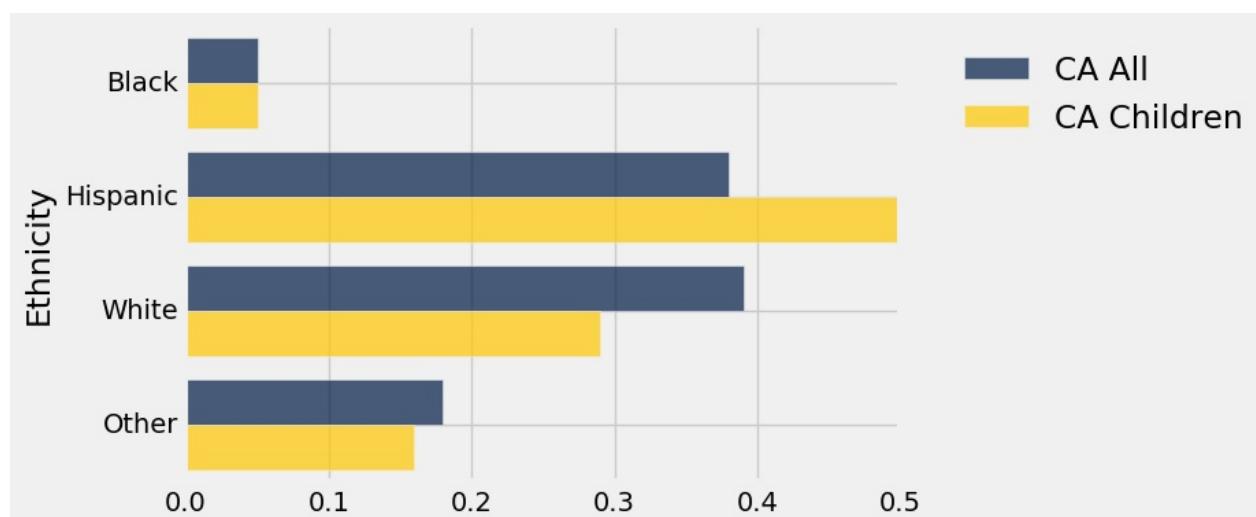
```
usa_ca.select('Ethnicity', 'USA All', 'CA All').barh('Ethnicity')
```



The two distributions are quite different. California has higher proportions in the Hispanic and other categories, and correspondingly lower proportions of Black and White. The differences are largely due to California's geographical location and patterns of immigration, both historically and in more recent decades. For example, the other category in California includes a significant proportion of Asians and Pacific Islanders.

As you can see from the graph, almost 40% of the Californian population in 2014 was Hispanic. A comparison with the population of children in the state indicates that the Hispanic proportion is likely to be greater in future years. Among Californian children, 50% are in the Hispanic category.

```
usa_ca.select('Ethnicity', 'CA All', 'CA Children').barh('Ethnicity')
```



More complex datasets naturally give rise to varied and interesting visualizations, including overlaid graphs of different kinds. To analyze such data, it helps to have some more skills in data manipulation, so that we can get the data into a form that allows us to use methods like those in this section. In the next chapter we will develop some of these skills.

[Interact](#)

Functions and Tables

We are building up a useful inventory of techniques for identifying patterns and themes in a data set by using functions already available in Python. We will now explore a core feature of the Python programming language: function definition.

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 want to perform the same manipulation on every value in a column of a table.

Defining a Function

The definition of the `double` function below simply doubles a number.

```
# Our first function definition

def double(x):
    """ Double x """
    return 2*x
```

We start any function definition by writing `def`. Here is a breakdown of the other parts (the *syntax*) of this small function:

Signature

- Calls to the function will look like this (with the same name and number of arguments). Example: `double(3)`.
- When you call `double`, the argument can be any expression. (The name `x` doesn't affect calls.)
- In the body of the function, `x` is the name of the argument, as if the body included the code `x = <the first argument>`.

```
# Our first function definition

def double(x):
    """ Double x """
    return 2*x
```

The diagram shows a flow from the 'Signature' section to the code example. A green line points from the word 'Signature' to the first line of the code ('# Our first function definition'). A blue line points from the word 'Body' to the 'return 2*x' line. An orange line points from the word 'Docstring' to the triple-quoted string ('''' Double x ''''').

Documentation (“docstring”)

- Text that describes what the function does.
- Can be any string, traditionally triple-quoted so it can span several lines.
- Traditionally, the first line describes what the function does, briefly.
- Subsequent lines can give more detail and examples.
- Running `double?` will show this text, just like `max?` will show the documentation for the built-in function `max`.

Body

- All the code in here runs each time you call the function.
- The special statement `return` tells Python what the value of each call to this function is: it's the value of the expression after `return`.
- For example, the value of `double(3)` is 6. (Remember, when the argument is 3, it's like the body starts with `x = 3`.)
- Often, the body will have multiple lines of code that build up to computing the returned value. You can write any Python code here that you could write anywhere else.

Indentation

- Each line of code in the body is indented (that is, it's preceded by spaces).
- Traditionally, we use 2 or 4 spaces. They only need to be consistent.
- This tells Python that those lines are part of the body.
- The function's body ends at any unindented line.

When we run the cell above, no particular number is doubled, and the code inside the body of `double` is not yet evaluated. In this respect, our function is analogous to a *recipe*. Each time we follow the instructions in a recipe, we need to start with ingredients. Each time we want to use our function to double a number, we need to specify a number.

We can call `double` in exactly the same way we have called other functions. Each time we do that, the code in the body is executed, with the value of the argument given the name `x`.

```
double(17)
```

34

```
double(-0.6/4)
```

-0.3

The two expressions above are both *call expressions*. In the second one, the value of the expression `-0.6/4` is computed and then passed as the argument named `x` to the `double` function. Each call expression results in the body of `double` being executed, but with a different value of `x`.

The body of `double` has only a single line:

```
return 2*x
```

Executing this `return statement` completes execution of the `double` function's body and computes the value of the call expression.

The argument to `double` can be any expression, as long as its value is a number. For example, it can be a name. The `double` function does not know or care how its argument is computed or stored; its only job is to execute its own body using the values of the arguments passed to it.

```
any_name = 42
double(any_name)
```

84

The argument can also be any value that can be doubled. For example, a whole array of numbers can be passed as an argument to `double`, and the result will be another array.

```
double(make_array(3, 4, 5))
```

```
array([ 6, 8, 10])
```

However, names that are defined inside a function, including arguments like `double`'s `x`, have only a fleeting existence. They are defined only while the function is being called, and they are only accessible inside the body of the function. We can't refer to `x` outside the body of `double`. The technical terminology is that `x` has *local scope*.

Therefore the name `x` isn't recognized outside the body of the function, even though we have called `double` in the cells above.

```
x
```

```
-----  
-----  
NameError Traceback (most recent  
call last)  
<ipython-input-18-401b30e3b8b5> in <module>()  
----> 1 x  
  
NameError: name 'x' is not defined
```

Docstrings. Though `double` is relatively easy to understand, many functions perform complicated tasks and are difficult to use without explanation. (You may have discovered this yourself!) Therefore, a well-composed function has a name that evokes its behavior, as well as documentation. In Python, this is called 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, as long as it is the first thing in a function's body. Docstrings are typically defined using triple quotation marks at the start and end, which allows a 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.

Here is a definition of a function called `percent` that takes two arguments. The definition includes a docstring.

```
# A function with more than one argument  
  
def percent(x, total):  
    """Convert x to a percentage of total.  
  
    More precisely, this function divides x by total,  
    multiplies the result by 100, and rounds the result  
    to two decimal places.  
  
    >>> percent(4, 16)  
    25.0  
    >>> percent(1, 6)  
    16.67  
    """  
    return round((x/total)*100, 2)
```

```
percent(33, 200)
```

```
16.5
```

Contrast the function `percent` defined above with the function `percents` defined below. The latter takes an array as its argument, and converts all the numbers in the array to percents out of the total of the values in the array. The percents are all rounded to two decimal places, this time replacing `round` by `np.round` because the argument is an array and not a number.

```
def percents(counts):
    """Convert the values in array_x to percents out of the
    total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, 2)
```

The function `percents` returns an array of percents that add up to 100 apart from rounding.

```
some_array = make_array(7, 10, 4)
percents(some_array)
```

```
array([ 33.33,  47.62,  19.05])
```

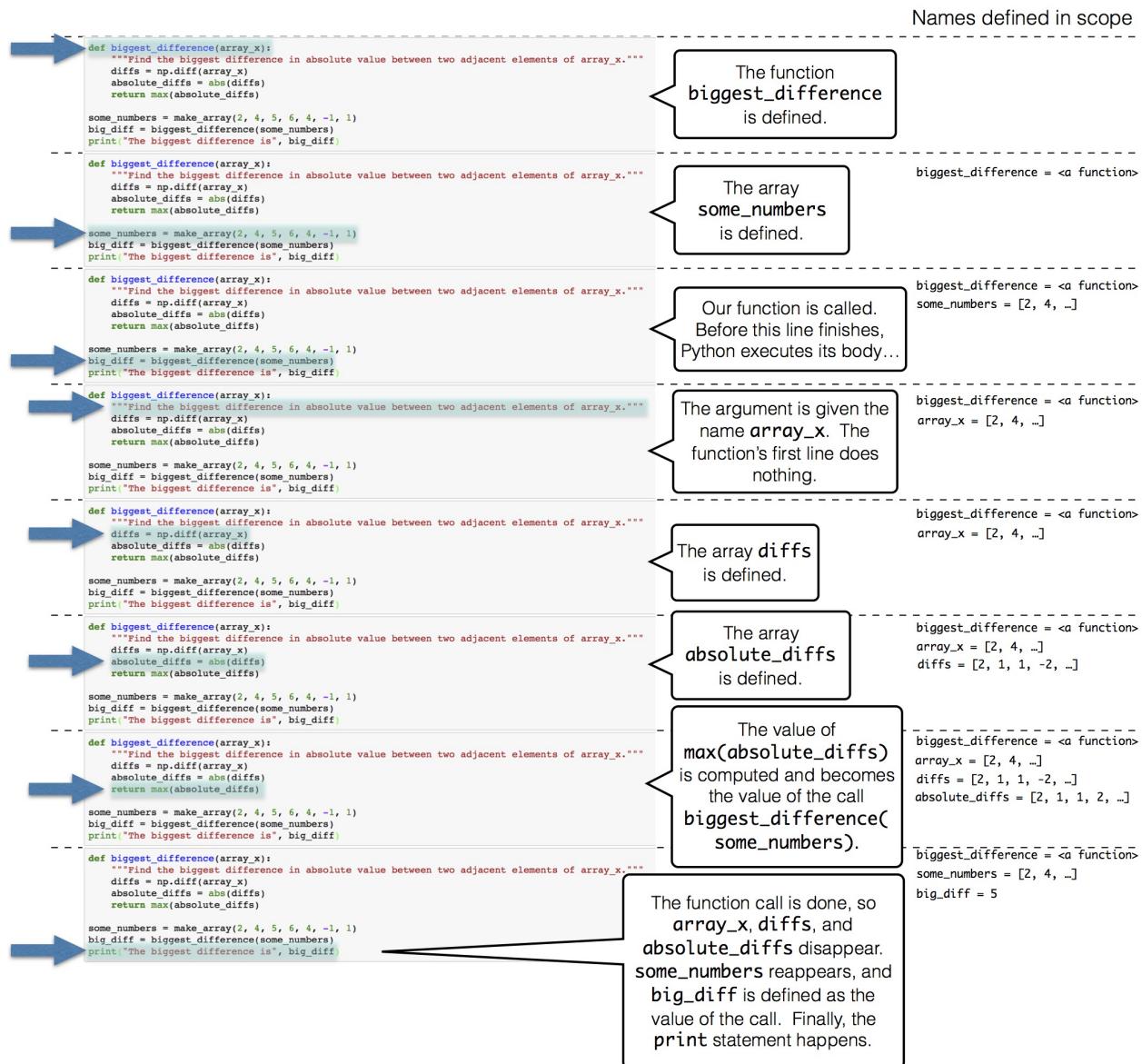
It is helpful to understand the steps Python takes to execute a function. To facilitate this, we have put a function definition and a call to that function in the same cell below.

```
def biggest_difference(array_x):
    """Find the biggest difference in absolute value between two
    adjacent elements of array_x."""
    diffs = np.diff(array_x)
    absolute_diffs = abs(diffs)
    return max(absolute_diffs)

some_numbers = make_array(2, 4, 5, 6, 4, -1, 1)
big_diff = biggest_difference(some_numbers)
print("The biggest difference is", big_diff)
```

The biggest difference is 5

Here is what happens when we run that cell:



Multiple Arguments

There can be multiple ways to generalize an expression or block of code, and so a function can take multiple arguments that each determine different aspects of the result. For example, the `percents` function we defined previously rounded to two decimal places every time. The following two-argument definition allows different calls to round to different amounts.

```

def percents(counts, decimal_places):
    """Convert the values in array_x to percents out of the
total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, decimal_places)

parts = make_array(2, 1, 4)
print("Rounded to 1 decimal place: ", percents(parts, 1))
print("Rounded to 2 decimal places:", percents(parts, 2))
print("Rounded to 3 decimal places:", percents(parts, 3))

```

```

Rounded to 1 decimal place: [ 28.6 14.3 57.1]
Rounded to 2 decimal places: [ 28.57 14.29 57.14]
Rounded to 3 decimal places: [ 28.571 14.286 57.143]

```

The flexibility of this new definition comes at a small price: each time the function is called, the number of decimal places must be specified. Default argument values allow a function to be called with a variable number of arguments; any argument that isn't specified in the call expression is given its default value, which is stated in the first line of the `def` statement. For example, in this final definition of `percents`, the optional argument `decimal_places` is given a default value of 2.

```

def percents(counts, decimal_places=2):
    """Convert the values in array_x to percents out of the
total of array_x."""
    total = counts.sum()
    return np.round((counts/total)*100, decimal_places)

parts = make_array(2, 1, 4)
print("Rounded to 1 decimal place:", percents(parts, 1))
print("Rounded to the default number of decimal places:",
percents(parts))

```

```

Rounded to 1 decimal place: [ 28.6 14.3 57.1]
Rounded to the default number of decimal places: [ 28.57 14.29
57.14]

```

Note: Methods

Functions are called by placing argument expressions in parentheses after the function name. Any function that is defined in isolation is called in this way. You have also seen examples of methods, which are like functions but are called using dot notation, such as `some_table.sort(some_label)`. The functions that you define will always be called using the function name first, passing in all of the arguments.

[Interact](#)

Applying a Function to a Column¶

We have seen many examples of creating new columns of tables by applying functions to existing columns or to other arrays. All of those functions took arrays as their arguments. But frequently we will want to convert the entries in a column by a function that doesn't take an array as its argument. For example, it might take just one number as its argument, as in the function `cut_off_at_100` defined below.

```
def cut_off_at_100(x):
    """The smaller of x and 100"""
    return min(x, 100)
```

```
cut_off_at_100(17)
```

```
17
```

```
cut_off_at_100(117)
```

```
100
```

```
cut_off_at_100(100)
```

```
100
```

The function `cut_off_at_100` simply returns its argument if the argument is less than or equal to 100. But if the argument is greater than 100, it returns 100.

In our earlier examples using Census data, we saw that the variable `AGE` had a value 100 that meant "100 years old or older". Cutting off ages at 100 in this manner is exactly what `cut_off_at_100` does.

To use this function on many ages at once, we will have to be able to *refer* to the function itself, without actually calling it. Analogously, we might show a cake recipe to a chef and ask her to use it to bake 6 cakes. In that scenario, we are not using the recipe to bake any cakes

ourselves; our role is merely to refer the chef to the recipe. Similarly, we can ask a table to call `cut_off_at_100` on 6 different numbers in a column.

First, we create the table `ages` with a column for people and one for their ages. For example, person `c` is 52 years old.

```
ages = Table().with_columns(
    'Person', make_array('A', 'B', 'C', 'D', 'E', 'F'),
    'Age', make_array(17, 117, 52, 100, 6, 101)
)
ages
```

Person	Age
A	17
B	117
C	52
D	100
E	6
F	101

apply ¶

To cut off each of the ages at 100, we will use the `apply` method. The `apply` method 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 or parentheses). The name of the column of input values is a string that must still appear within quotation marks.

```
ages.apply(cut_off_at_100, 'Age')
```

```
array([ 17, 100, 52, 100, 6, 100])
```

What we have done here is `apply` the function `cut_off_at_100` to each value in the `Age` column of the table `ages`. The output is the array of corresponding return values of the function. For example, 17 stayed 17, 117 became 100, 52 stayed 52, and so on.

This array, which has the same length as the original `Age` column of the `ages` table, can be used as the values in a new column called `cut off Age` alongside the existing `Person` and `Age` columns.

```
ages.with_column(
    'Cut Off Age', ages.apply(cut_off_at_100, 'Age')
)
```

Person	Age	Cut Off Age
A	17	17
B	117	100
C	52	52
D	100	100
E	6	6
F	101	100

Functions as Values

We've seen that Python has many kinds of values. For example, `6` is a number value, `"cake"` is a text value, `Table()` is an empty table, and `ages` is a name for a table value (since we defined it above).

In Python, every function, including `cut_off_at_100`, is also a value. It helps to think about recipes again. A recipe for cake is a real thing, distinct from cakes or ingredients, and you can give it a name like "Ani's cake recipe." When we defined `cut_off_at_100` with a `def` statement, we actually did two separate things: we created a function that cuts off numbers at 100, and we gave it the name `cut_off_at_100`.

We can refer to any function by writing its name, without the parentheses or arguments necessary to actually call it. We did this when we called `apply` above. When we write a function's name by itself as the last line in a cell, Python produces a text representation of the function, just like it would print out a number or a string value.

```
cut_off_at_100
```

```
<function __main__.cut_off_at_100>
```

Notice that we did not write `"cut_off_at_100"` with quotes (which is just a piece of text), or `cut_off_at_100()` (which is a function call, and an invalid one at that). We simply wrote `cut_off_at_100` to refer to the function.

Just like we can define new names for other values, we can define new names for functions. For example, suppose we want to refer to our function as `cut_off` instead of `cut_off_at_100`. We can just write this:

```
cut_off = cut_off_at_100
```

Now `cut_off` is a name for a function. It's the same function as `cut_off_at_100`, so the printed value is exactly the same.

```
cut_off
```

```
<function __main__.cut_off_at_100>
```

Let us see another application of `apply`.

Example: Prediction

Data Science is often used to make predictions about the future. If we are trying to predict an outcome for a particular individual – for example, how she will respond to a treatment, or whether he will buy a product – it is natural to base the prediction on the outcomes of other similar individuals.

Charles Darwin's cousin [Sir Francis Galton](#) was a pioneer in using this idea to make predictions based on numerical data. He studied how physical characteristics are passed down from one generation to the next.

The data below are Galton's carefully collected measurements on the heights of parents and their adult children. Each row corresponds to one adult child. The variables are a numerical code for the family, the heights (in inches) of the father and mother, a "midparent height" which is a weighted average [1] of the height of the two parents, the number of children in the family, as well as the child's birth rank (1 = oldest), gender, and height.

```
# Galton's data on heights of parents and their adult children
galton = Table.read_table('galton.csv')
galton
```

family	father	mother	midparentHeight	children	childNum	gender
1	78.5	67	75.43	4	1	male
1	78.5	67	75.43	4	2	female
1	78.5	67	75.43	4	3	female
1	78.5	67	75.43	4	4	female
2	75.5	66.5	73.66	4	1	male
2	75.5	66.5	73.66	4	2	male
2	75.5	66.5	73.66	4	3	female
2	75.5	66.5	73.66	4	4	female
3	75	64	72.06	2	1	male
3	75	64	72.06	2	2	female

... (924 rows omitted)

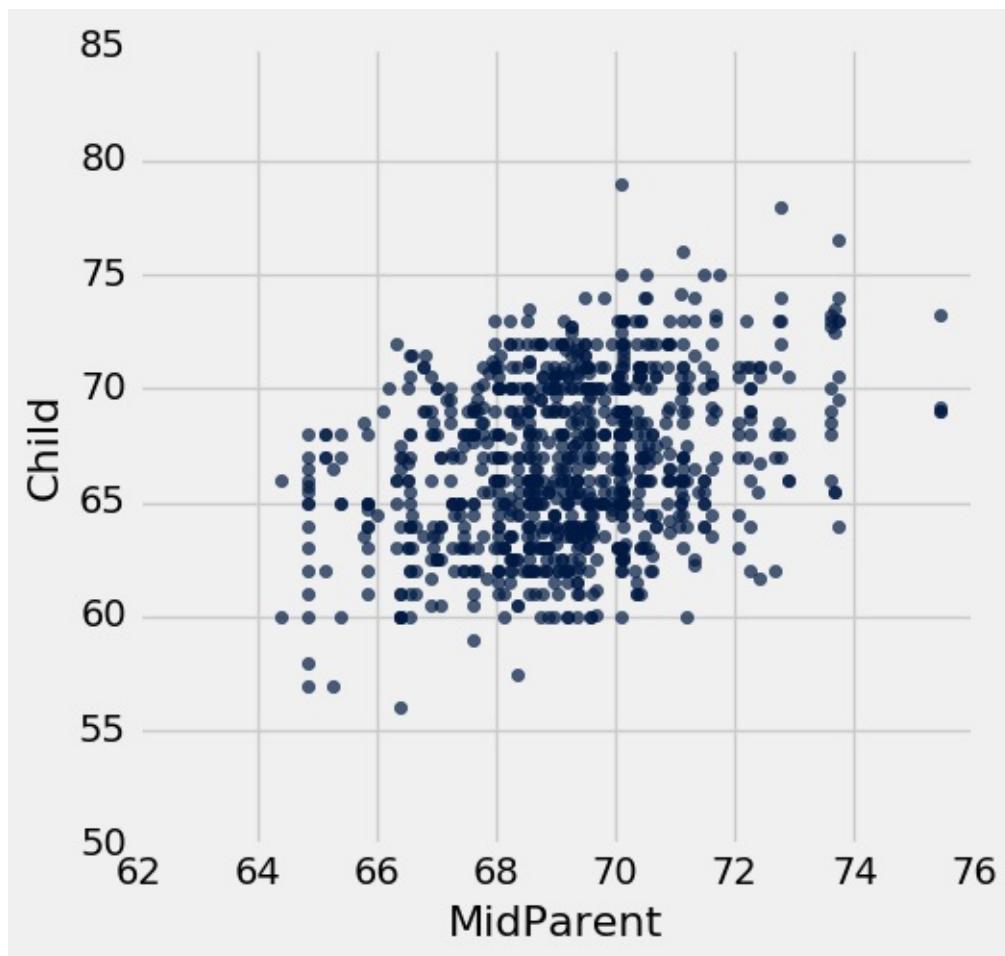
A primary reason for collecting the data was to be able to predict the adult height of a child born to parents similar to those in the dataset. Let us try to do this, using midparent height as the variable on which to base our prediction. Thus midparent height is our *predictor* variable.

The table `heights` consists of just the midparent heights and child's heights. The scatter plot of the two variables shows a positive association, as we would expect for these variables.

```
heights = galton.select(3, 7).relabeled(0,
                                         'MidParent').relabeled(1, 'Child')
heights
```

MidParent	Child
75.43	73.2
75.43	69.2
75.43	69
75.43	69
73.66	73.5
73.66	72.5
73.66	65.5
73.66	65.5
72.06	71
72.06	68
... (924 rows omitted)	

```
heights.scatter(0)
```



Now suppose Galton encountered a new couple, similar to those in his dataset, and wondered how tall their child would be. What would be a good way for him to go about predicting the child's height, given that the midparent height was, say, 68 inches?

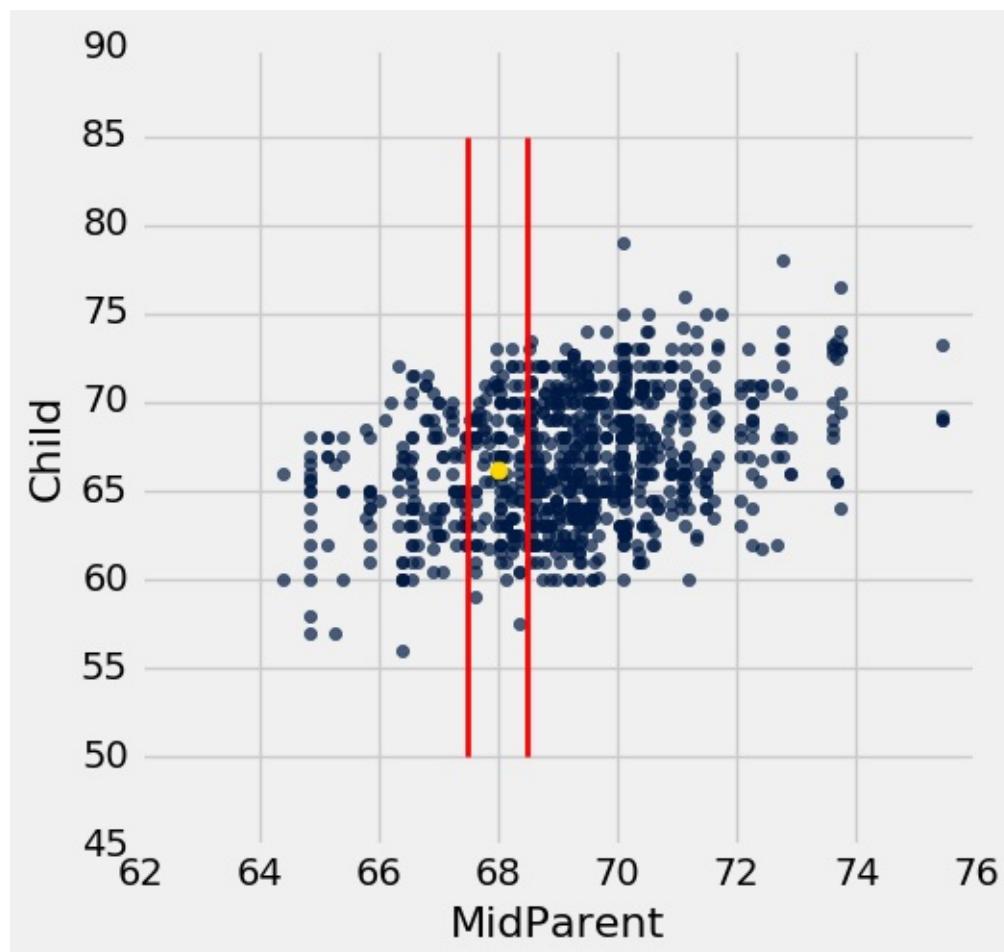
One reasonable approach would be to base the prediction on all the points that correspond to a midparent height of around 68 inches. The prediction equals the average child's height calculated from those points alone.

Let's pretend we are Galton and execute this plan. For now we will just make a reasonable definition of what "around 68 inches" means, and work with that. Later in the course we will examine the consequences of such choices.

We will take "close" to mean "within half an inch". The figure below shows all the points corresponding to a midparent height between 67.5 inches and 68.5 inches. These are all the points in the strip between the red lines. Each of these points corresponds to one child; our prediction of the height of the new couple's child is the average height of all the children in the strip. That's represented by the gold dot.

Ignore the code, and just focus on understanding the mental process of arriving at that gold dot.

```
heights.scatter('MidParent')
_ = plots.plot([67.5, 67.5], [50, 85], color='red', lw=2)
_ = plots.plot([68.5, 68.5], [50, 85], color='red', lw=2)
_ = plots.scatter(68, 66.24, color='gold', s=40)
```



In order to calculate exactly where the gold dot should be, we first need to identify all the points in the strip. These correspond to the rows where `MidParent` is between 67.5 inches and 68.5 inches.

```
close_to_68 = heights.where('MidParent', are.between(67.5,  
68.5))  
close_to_68
```

MidParent	Child
68.44	62
67.94	71.2
67.94	67
68.33	62.5
68.23	73
68.23	72
68.23	69
67.98	73
67.98	71
67.98	71

... (121 rows omitted)

The predicted height of a child who has a midparent height of 68 inches is the average height of the children in these rows. That's 66.24 inches.

```
close_to_68.column('Child').mean()
```

```
66.24045801526718
```

We now have a way to predict the height of a child given any value of the midparent height near those in our dataset. We can define a function `predict_child` that does this. The body of the function consists of the code in the two cells above, apart from choices of names.

```

def predict_child(mpht):
    """Predict the height of a child whose parents have a
midparent height of mpht.

    The prediction is the average height of the children whose
    midparent height is
        in the range mpht plus or minus 0.5.
    """

    close_points = heights.where('MidParent', are.between(mpht-
0.5, mpht + 0.5))
    return close_points.column('Child').mean()

```

1**1**

Given a midparent height of 68 inches, the function `predict_child` returns the same prediction (66.24 inches) as we got earlier. The advantage of defining the function is that we can easily change the value of the predictor and get a new prediction.

```
predict_child(68)
```

```
66.24045801526718
```

```
predict_child(74)
```

```
70.415789473684214
```

How good are these predictions? We can get a sense of this by comparing the predictions with the data that we already have. To do this, we first apply the function `predict_child` to the column of `Midparent` heights, and collect the results in a new column called `Prediction`.

```

# Apply predict_child to all the midparent heights

heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
)

```

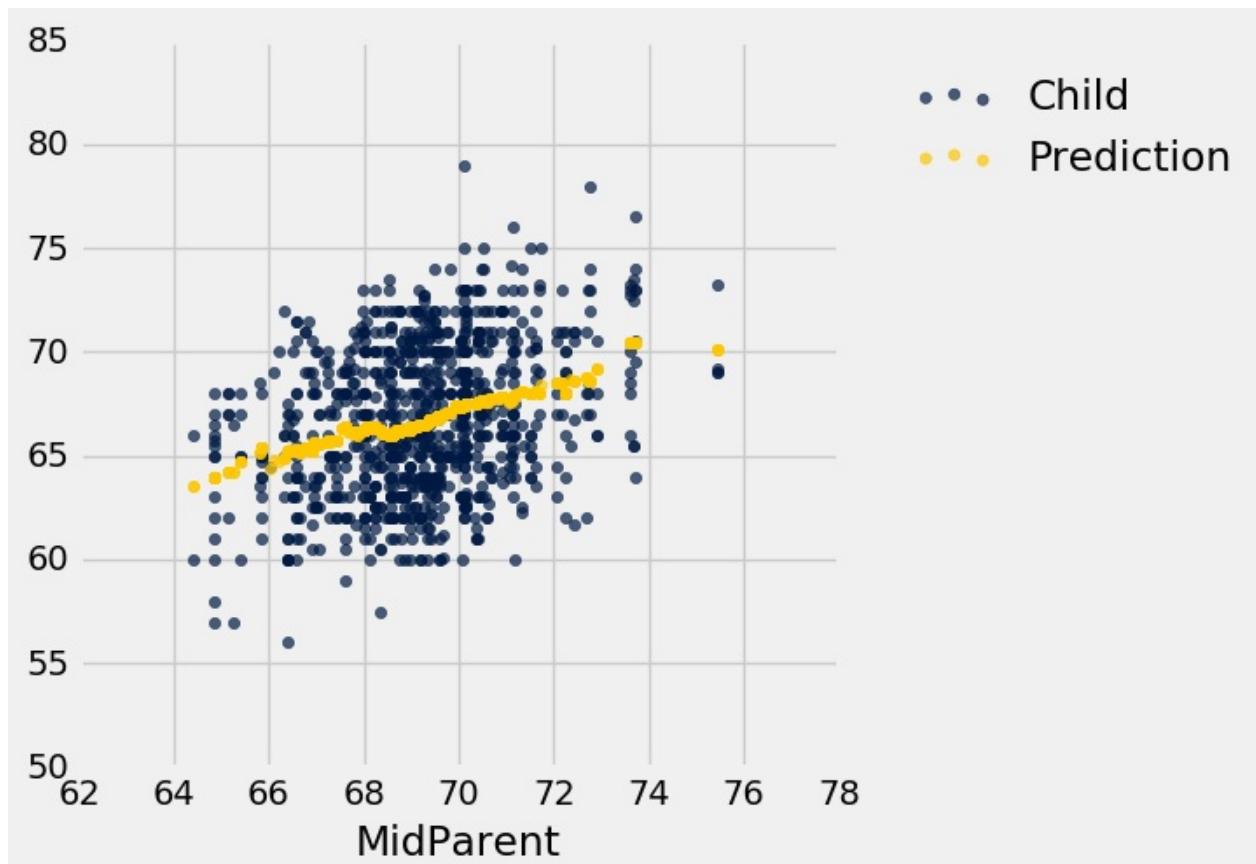
```
heights_with_predictions
```

MidParent	Child	Prediction
75.43	73.2	70.1
75.43	69.2	70.1
75.43	69	70.1
75.43	69	70.1
73.66	73.5	70.4158
73.66	72.5	70.4158
73.66	65.5	70.4158
73.66	65.5	70.4158
72.06	71	68.5025
72.06	68	68.5025

... (924 rows omitted)

To see where the predictions lie relative to the observed data, we can draw overlaid scatter plots with `MidParent` as the common horizontal axis.

```
heights_with_predictions.scatter('MidParent')
```



The graph of gold dots is called a *graph of averages*, because each gold dot is the center of a vertical strip like the one we drew earlier. Each one provides a prediction of a child's height given the midparent height. For example, the scatter shows that for a midparent height of 72 inches, the predicted height of the child would be somewhere between 68 inches and 69 inches, and indeed `predict_child(72)` returns 68.5.

Galton's calculations and visualizations were very similar to ours, except that he didn't have Python. He drew the graph of averages through the scatter diagram and noticed that it roughly followed a straight line. This straight line is now called the *regression line* and is one of the most common methods of making predictions. Galton's friend, the mathematician Karl Pearson, used these analyses to formalize the notion of *correlation*.

This example, like the one about John Snow's analysis of cholera deaths, shows how some of the fundamental concepts of modern data science have roots going back more than a century. Galton's methods such as the one we have used here are precursors to *nearest neighbor* prediction methods that now have powerful applications in diverse settings. The modern field of *machine learning* includes the automation of such methods to make predictions based on vast and rapidly evolving datasets.

Footnotes

[1] Galton multiplied the heights of all the women by 1.08 before taking the average height of the men and the women. For a discussion of this, see [Chance](#), a magazine published by the American Statistical Association.

[Interact](#)

Classifying by One Variable

Data scientists often need to classify individuals into groups according to shared features, and then identify some characteristics of the groups. For example, in the example using Galton's data on heights, we saw that it was useful to classify families according to the parents' midparent heights, and then find the average height of the children in each group.

This section is about classifying individuals into categories that are not numerical. We begin by recalling the basic use of `group`.

Counting the Number in Each Category

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.

Here is a small table of data on ice cream cones. The `group` method can be used to list the distinct flavors and provide the counts of each flavor.

```
cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'chocolate', 'chocolate',
    'strawberry', 'chocolate'),
    'Price', make_array(3.55, 4.75, 6.55, 5.25, 5.25)
)
cones
```

Flavor	Price
strawberry	3.55
chocolate	4.75
chocolate	6.55
strawberry	5.25
chocolate	5.25

```
cones.group('Flavor')
```

Flavor	count
chocolate	3
strawberry	2

There are two distinct categories, chocolate and strawberry. The call to `group` creates a table of counts in each category. The column is called `count` by default, and contains the number of rows in each category.

Notice that this can all be worked out from just the `Flavor` column. The `Price` column has not been used.

But what if we wanted the total price of the cones of each different flavor? That's where the second argument of `group` comes in.

Finding a Characteristic of Each Category

The optional second argument of `group` 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 prices 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.

To find the total price of each flavor, we call `group` again, with `Flavor` as its first argument as before. But this time there is a second argument: the function name `sum`.

```
cones.group('Flavor', sum)
```

Flavor	Price sum
chocolate	16.55
strawberry	8.8

To create this new table, `group` has calculated the sum of the `Price` entries in all the rows corresponding to each distinct flavor. The prices in the three `chocolate` rows add up to **\$16.55** (you can assume that price is being measured in dollars). The prices in the two `strawberry` rows have a total of **\$8.80**.

The label of the newly created "sum" column is `Price sum`, which is created by taking the label of the column being summed, and appending the word `sum`.

Because `group` finds the `sum` of all columns other than the one with the categories, there is no need to specify that it has to `sum` the prices.

To see in more detail what `group` is doing, notice that you could have figured out the total prices yourself, not only by mental arithmetic but also using code. For example, to find the total price of all the chocolate cones, you could start by creating a new table consisting of only the chocolate cones, and then accessing the column of prices:

```
cones.where('Flavor', are.equal_to('chocolate')).column('Price')
```

```
array([ 4.75,  6.55,  5.25])
```

```
sum(cones.where('Flavor',
are.equal_to('chocolate')).column('Price'))
```

```
16.550000000000001
```

This is what `group` is doing for each distinct value in `Flavor`.

```

# For each distinct value in `Flavor`, access all the rows
# and create an array of `Price` 

cones_choc = cones.where('Flavor',
are.equal_to('chocolate')).column('Price')
cones_strawb = cones.where('Flavor',
are.equal_to('strawberry')).column('Price')

# Display the arrays in a table

grouped_cones = Table().with_columns(
    'Flavor', make_array('chocolate', 'strawberry'),
    'Array of All the Prices', make_array(cones_choc,
cones_strawb)
)

# Append a column with the sum of the `Price` values in each
array

price_totals = grouped_cones.with_column(
    'Sum of the Array', make_array(sum(cones_choc),
sum(cones_strawb))
)
price_totals

```

Flavor	Array of All the Prices	Sum of the Array
chocolate	[4.75 6.55 5.25]	16.55
strawberry	[3.55 5.25]	8.8

You can replace `sum` by any other functions that work on arrays. For example, you could use `max` to find the largest price in each category:

```
cones.group('Flavor', max)
```

Flavor	Price max
chocolate	6.55
strawberry	5.25

Once again, `group` creates arrays of the prices in each `Flavor` category. But now it finds the `max` of each array:

```
price_maxes = grouped_cones.with_column(
    'Max of the Array', make_array(max(cones_choc),
    max(cones_strawb)))
)
price_maxes
```

Flavor	Array of All the Prices	Max of the Array
chocolate	[4.75 6.55 5.25]	6.55
strawberry	[3.55 5.25]	5.25

Indeed, the original call to `group` with just one argument has the same effect as using `len` as the function and then cleaning up the table.

```
lengths = grouped_cones.with_column(
    'Length of the Array', make_array(len(cones_choc),
    len(cones_strawb)))
)
lengths
```

Flavor	Array of All the Prices	Length of the Array
chocolate	[4.75 6.55 5.25]	3
strawberry	[3.55 5.25]	2

Example: NBA Salaries

The table `nba` contains data on the 2015-2016 players in the National Basketball Association. We have examined these data earlier. Recall that salaries are measured in millions of dollars.

```
nba1 = Table.read_table('nba_salaries.csv')
nba = nba1.relabeled("'15-'16 SALARY", 'SALARY')
nba
```

PLAYER	POSITION	TEAM	SALARY
Paul Millsap	PF	Atlanta Hawks	18.6717
Al Horford	C	Atlanta Hawks	12
Tiago Splitter	C	Atlanta Hawks	9.75625
Jeff Teague	PG	Atlanta Hawks	8
Kyle Korver	SG	Atlanta Hawks	5.74648
Thabo Sefolosha	SF	Atlanta Hawks	4
Mike Scott	PF	Atlanta Hawks	3.33333
Kent Bazemore	SF	Atlanta Hawks	2
Dennis Schroder	PG	Atlanta Hawks	1.7634
Tim Hardaway Jr.	SG	Atlanta Hawks	1.30452

... (407 rows omitted)

- How much money did each team pay for its players' salaries?

The only columns involved are `TEAM` and `SALARY`. We have to `group` the rows by `TEAM` and then `sum` the salaries of the groups.

```
teams_and_money = nba.select('TEAM', 'SALARY')
teams_and_money.groupby('TEAM').sum()
```

TEAM	SALARY sum
Atlanta Hawks	69.5731
Boston Celtics	50.2855
Brooklyn Nets	57.307
Charlotte Hornets	84.1024
Chicago Bulls	78.8209
Cleveland Cavaliers	102.312
Dallas Mavericks	65.7626
Denver Nuggets	62.4294
Detroit Pistons	42.2118
Golden State Warriors	94.0851

... (20 rows omitted)

2. How many NBA players were there in each of the five positions?

We have to classify by `POSITION`, and count. This can be done with just one argument to `group`:

```
nba.group('POSITION')
```

POSITION	count
C	69
PF	85
PG	85
SF	82
SG	96

3. What was the average salary of the players at each of the five positions?

This time, we have to group by `POSITION` and take the mean of the salaries. For clarity, we will work with a table of just the positions and the salaries.

```
positions_and_money = nba.select('POSITION', 'SALARY')
positions_and_money.group('POSITION', np.mean)
```

POSITION	SALARY mean
C	6.08291
PF	4.95134
PG	5.16549
SF	5.53267
SG	3.9882

Center was the most highly paid position, at an average of over 6 million dollars.

If we had not selected the two columns as our first step, `group` would not attempt to "average" the categorical columns in `nba`. (It is impossible to average two strings like "Atlanta Hawks" and "Boston Celtics".) It performs arithmetic only on numerical columns and leaves the rest blank.

```
nba.group('POSITION', np.mean)
```

POSITION	PLAYER mean	TEAM mean	SALARY mean
C			6.08291
PF			4.95134
PG			5.16549
SF			5.53267
SG			3.9882

[Interact](#)

Cross-Classifying by More than One Variable

When individuals have multiple features, there are many different ways to classify them. For example, if we have a population of college students for each of whom we have recorded a major and the number of years in college, then the students could be classified by major, or by year, or by a combination of major and year.

The `group` method also allows us to classify individuals according to multiple variables. This is called *cross-classifying*.

Two Variables: Counting the Number in Each Paired Category

The table `more_cones` records the flavor, color, and price of six ice cream cones.

```
more_cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'chocolate', 'chocolate',
'strawberry', 'chocolate', 'bubblegum'),
    'Color', make_array('pink', 'light brown', 'dark brown',
'pink', 'dark brown', 'pink'),
    'Price', make_array(3.55, 4.75, 5.25, 5.25, 5.25, 4.75)
)

more_cones
```

Flavor	Color	Price
strawberry	pink	3.55
chocolate	light brown	4.75
chocolate	dark brown	5.25
strawberry	pink	5.25
chocolate	dark brown	5.25
bubblegum	pink	4.75

We know how to use `group` to count the number of cones of each flavor:

```
more_cones.group('Flavor')
```

Flavor	count
bubblegum	1
chocolate	3
strawberry	2

But now each cone has a color as well. To classify the cones by both flavor and color, we will pass a list of labels as an argument to `group`. The resulting table has one row for every *unique combination* of values that appear together in the grouped columns. As before, a single argument (a list, in this case, but an array would work too) gives row counts.

Although there are six cones, there are only four unique combinations of flavor and color. Two of the cones were dark brown chocolate, and two pink strawberry.

```
more_cones.group(['Flavor', 'Color'])
```

Flavor	Color	count
bubblegum	pink	1
chocolate	dark brown	2
chocolate	light brown	1
strawberry	pink	2

Two Variables: Finding a Characteristic of Each Paired Category

A second argument aggregates all other columns that are not in the list of grouped columns.

```
more_cones.group(['Flavor', 'Color'], sum)
```

Flavor	Color	Price sum
bubblegum	pink	4.75
chocolate	dark brown	10.5
chocolate	light brown	4.75
strawberry	pink	8.8

Three or More Variables. You can use `group` to classify rows by three or more categorical variables. Just include them all in the list that is the first argument. But cross-classifying by multiple variables can become complex, as the number of distinct combinations of

categories can be quite large.

Pivot Tables: Rearranging the Output of `group`

Many uses of cross-classification involve just two categorical variables, like `Flavor` and `Color` in the example above. In these cases it is possible to display the results of the classification in a different kind of table, called a *pivot table*. Pivot tables, also known as *contingency tables*, make it easier to work with data that have been classified according to two variables.

Recall the use of `group` to count the number of cones in each paired category of flavor and color:

```
more_cones.groupby(['Flavor', 'Color'])
```

Flavor	Color	count
bubblegum	pink	1
chocolate	dark brown	2
chocolate	light brown	1
strawberry	pink	2

The same data can be displayed differently using the `Table` method `pivot`. Ignore the code for a moment, and just examine the table of outcomes.

```
more_cones.pivot('Flavor', 'Color')
```

Color	bubblegum	chocolate	strawberry
dark brown	0	2	0
light brown	0	1	0
pink	1	0	2

Notice how this table displays all nine possible pairs of flavor and color, including pairs like "dark brown bubblegum" that don't exist in our data. Notice also that the count in each pair appears in the body of the table: to find the number of light brown chocolate cones, run your eye along the row `light brown` until it meets the column `chocolate`.

The `group` method takes a list of two labels because it is flexible: it could take one or three or more. On the other hand, `pivot` always takes two column labels, one to determine the columns and one to determine the rows.

pivot

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

Like `group`, `pivot` can be used with additional arguments to find characteristics of each paired category. An optional third argument called `values` indicates a column of values that will replace the counts in each cell of the grid. All of these values will not be displayed, however; the fourth argument `collect` indicates how to collect them all into one aggregated value to be displayed in the cell.

An example will help clarify this. Here is `pivot` being used to find the total price of the cones in each cell.

```
more_cones.pivot('Flavor', 'Color', values='Price', collect=sum)
```

Color	bubblegum	chocolate	strawberry
dark brown	0	10.5	0
light brown	0	4.75	0
pink	4.75	0	8.8

And here is `group` doing the same thing.

```
more_cones.group(['Flavor', 'Color'], sum)
```

Flavor	Color	Price sum
bubblegum	pink	4.75
chocolate	dark brown	10.5
chocolate	light brown	4.75
strawberry	pink	8.8

Though the numbers in both tables are the same, table produced by `pivot` is easier to read and lends itself more easily to analysis. The advantage of `pivot` is that it places grouped values into adjacent columns, so that they can be combined and compared.

Example: Education and Income of Californian Adults

The State of California's Open Data Portal is a rich source of information about the lives of Californians. It is our source of a [dataset](#) on educational attainment and personal income among Californians over the years 2008 to 2014. The data are derived from the U.S. Census Current Population Survey.

For each year, the table records the `Population Count` of Californians in many different combinations of age, gender, educational attainment, and personal income. We will study only the data for the year 2014.

```
full_table = Table.read_table('educ_inc.csv')
ca_2014 = full_table.where('Year', are.equal_to('1/1/14
0:00')).where('Age', are.not_equal_to('00 to 17'))
ca_2014
```

Year	Age	Gender	Educational Attainment	Personal Income	Population Count
1/1/14 0:00	18 to 64	Female	No high school diploma	H: 75,000 and over	2058
1/1/14 0:00	65 to 80+	Male	No high school diploma	H: 75,000 and over	2153
1/1/14 0:00	65 to 80+	Female	No high school diploma	G: 50,000 to 74,999	4666
1/1/14 0:00	65 to 80+	Female	High school or equivalent	H: 75,000 and over	7122
1/1/14 0:00	65 to 80+	Female	No high school diploma	F: 35,000 to 49,999	7261
1/1/14 0:00	65 to 80+	Male	No high school diploma	G: 50,000 to 74,999	8569
1/1/14 0:00	18 to 64	Female	No high school diploma	G: 50,000 to 74,999	14635
1/1/14 0:00	65 to 80+	Male	No high school diploma	F: 35,000 to 49,999	15212
1/1/14 0:00	65 to 80+	Male	College, less than 4- yr degree	B: 5,000 to 9,999	15423
1/1/14 0:00	65 to 80+	Female	Bachelor's degree or higher	A: 0 to 4,999	15459

... (117 rows omitted)

Each row of the table corresponds to a combination of age, gender, educational level, and income. There are 127 such combinations in all!

As a first step it is a good idea to start with just one or two variables. We will focus on just one pair: educational attainment and personal income.

```
educ_inc = ca_2014.select('Educational Attainment', 'Personal Income', 'Population Count')
educ_inc
```

Educational Attainment	Personal Income	Population Count
No high school diploma	H: 75,000 and over	2058
No high school diploma	H: 75,000 and over	2153
No high school diploma	G: 50,000 to 74,999	4666
High school or equivalent	H: 75,000 and over	7122
No high school diploma	F: 35,000 to 49,999	7261
No high school diploma	G: 50,000 to 74,999	8569
No high school diploma	G: 50,000 to 74,999	14635
No high school diploma	F: 35,000 to 49,999	15212
College, less than 4-yr degree	B: 5,000 to 9,999	15423
Bachelor's degree or higher	A: 0 to 4,999	15459

... (117 rows omitted)

Let's start by looking at educational level alone. The categories of this variable have been subdivided by the different levels of income. So we will group the table by `Educational Attainment` and `sum` the `Population Count` in each category.

```
education = educ_inc.select('Educational Attainment',
                            'Population Count')
educ_totals = education.group('Educational Attainment', sum)
educ_totals
```

Educational Attainment	Population Count sum
Bachelor's degree or higher	8525698
College, less than 4-yr degree	7775497
High school or equivalent	6294141
No high school diploma	4258277

There are only four categories of educational attainment. The counts are so large that it is more helpful to look at percents. For this, we will use the function `percents` that we defined in an earlier section. It converts an array of numbers to an array of percents out of the total in the input array.

```
def percents(array_x):
    return np.round( (array_x/sum(array_x))*100, 2)
```

We now have the distribution of educational attainment among adult Californians. More than 30% have a Bachelor's degree or higher, while almost 16% lack a high school diploma.

```
educ_distribution = educ_totals.with_column(
    'Population Percent', percents(educ_totals.column(1))
)
educ_distribution
```

Educational Attainment	Population Count sum	Population Percent
Bachelor's degree or higher	8525698	31.75
College, less than 4-yr degree	7775497	28.96
High school or equivalent	6294141	23.44
No high school diploma	4258277	15.86

By using `pivot`, we can get a contingency table (a table of counts) of adult Californians cross-classified by `Educational Attainment` and `Personal Income`.

```
totals = educ_inc.pivot('Educational Attainment', 'Personal
Income', values='Population Count', collect=sum)
totals
```

Personal Income	Bachelor's degree or higher	College, less than 4-yr degree	High school or equivalent	No high school diploma
A: 0 to 4,999	575491	985011	1161873	1204529
B: 5,000 to 9,999	326020	810641	626499	597039
C: 10,000 to 14,999	452449	798596	692661	664607
D: 15,000 to 24,999	773684	1345257	1252377	875498
E: 25,000 to 34,999	693884	1091642	929218	464564
F: 35,000 to 49,999	1122791	1112421	782804	260579
G: 50,000 to 74,999	1594681	883826	525517	132516
H: 75,000 and over	2986698	748103	323192	58945

Here you see the power of `pivot` over other cross-classification methods. Each column of counts is a distribution of personal income at a specific level of educational attainment. Converting the counts to percents allows us to compare the four distributions.

```
distributions = totals.select(0).with_columns(
    "Bachelor's degree or higher", percents(totals.column(1)),
    'College, less than 4-yr degree',
    percents(totals.column(2)),
    'High school or equivalent', percents(totals.column(3)),
    'No high school diploma', percents(totals.column(4))
)

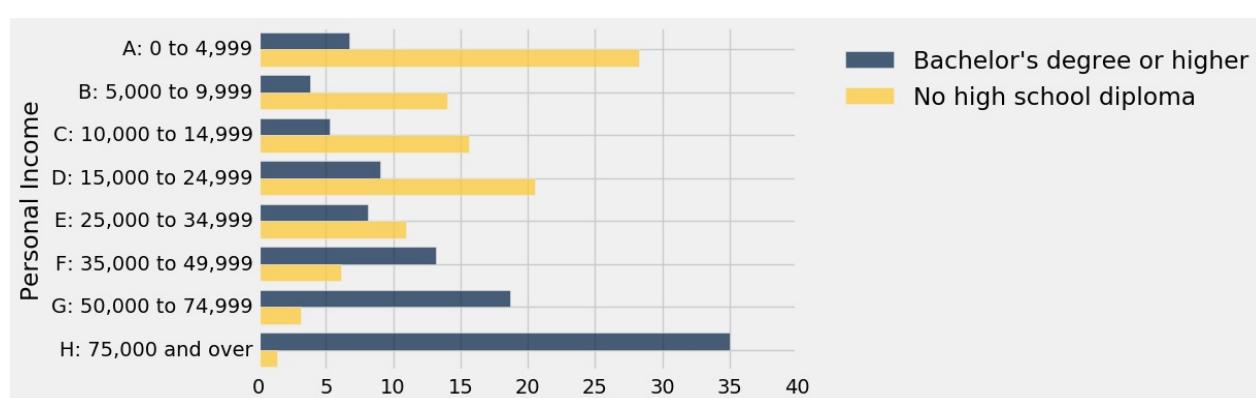
distributions
```

Personal Income	Bachelor's degree or higher	College, less than 4-yr degree	High school or equivalent	No high school diploma
A: 0 to 4,999	6.75	12.67	18.46	28.29
B: 5,000 to 9,999	3.82	10.43	9.95	14.02
C: 10,000 to 14,999	5.31	10.27	11	15.61
D: 15,000 to 24,999	9.07	17.3	19.9	20.56
E: 25,000 to 34,999	8.14	14.04	14.76	10.91
F: 35,000 to 49,999	13.17	14.31	12.44	6.12
G: 50,000 to 74,999	18.7	11.37	8.35	3.11
H: 75,000 and over	35.03	9.62	5.13	1.38

At a glance, you can see that over 35% of those with Bachelor's degrees or higher had incomes of **\$75,000** and over, whereas fewer than 10% of the people in the other education categories had that level of income.

The bar chart below compares the personal income distributions of adult Californians who have no high diploma with those who have completed a Bachelor's degree or higher. The difference in the distributions is striking. There is a clear positive association between educational attainment and personal income.

```
distributions.select(0, 1, 4).barh(0)
```



[Interact](#)

Joining Tables by Columns

Often, data about the same individuals is maintained in more than one table. For example, one university office might have data about each student's time to completion of degree, while another has data about the student's tuition and financial aid.

To understand the students' experience, it may be helpful to put the two datasets together. If the data are in two tables, each with one row per student, then we would want to put the columns together, making sure to match the rows so that each student's information remains on a single row.

Let us do this in the context of a simple example, and then use the method with a larger dataset.

The table `cones` is one we have encountered earlier. Now suppose each flavor of ice cream comes with a rating that is in a separate table.

```
cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'vanilla', 'chocolate',
    'strawberry', 'chocolate'),
    'Price', make_array(3.55, 4.75, 6.55, 5.25, 5.75)
)
cones
```

Flavor	Price
strawberry	3.55
vanilla	4.75
chocolate	6.55
strawberry	5.25
chocolate	5.75

```
ratings = Table().with_columns(
    'Kind', make_array('strawberry', 'chocolate', 'vanilla'),
    'Stars', make_array(2.5, 3.5, 4)
)
ratings
```

Kind	Stars
strawberry	2.5
chocolate	3.5
vanilla	4

Each of the tables has a column that contains ice cream flavors: `cones` has the column `Flavor`, and `ratings` has the column `Kind`. The entries in these columns can be used to link the two tables.

The method `join` creates a new table in which each cone in the `cones` table is augmented with the Stars information in the `ratings` table. For each cone in `cones`, `join` finds a row in `ratings` whose `Kind` matches the cone's `Flavor`. We have to tell `join` to use those columns for matching.

```
rated = cones.join('Flavor', ratings, 'Kind')
rated
```

Flavor	Price	Stars
chocolate	6.55	3.5
chocolate	5.75	3.5
strawberry	3.55	2.5
strawberry	5.25	2.5
vanilla	4.75	4

Each cone now has not only its price but also the rating of its flavor.

In general, a call to `join` that augments a table (say `table1`) with information from another table (say `table2`) looks like this:

```
table1.join(table1_column_for_joining, table2, table2_column_for_joining)
```

The new table `rated` allows us to work out the price per star, which you can think of as an informal measure of value. Low values are good – they mean that you are paying less for each rating star.

```
rated.with_column('/$Star', rated.column('Price') /
rated.column('Stars')).sort(3)
```

Flavor	Price	Stars	\$/Star
vanilla	4.75	4	1.1875
strawberry	3.55	2.5	1.42
chocolate	5.75	3.5	1.64286
chocolate	6.55	3.5	1.87143
strawberry	5.25	2.5	2.1

Though strawberry has the lowest rating among the three flavors, the less expensive strawberry cone does well on this measure because it doesn't cost a lot per star.

Side note. Does the order we list the two tables matter? Let's try it. As you see it, this changes the order that the columns appear in, and can potentially changes the order of the rows, but it doesn't make any fundamental difference.

```
ratings.join('Kind', cones, 'Flavor')
```

Kind	Stars	Price
chocolate	3.5	6.55
chocolate	3.5	5.75
strawberry	2.5	3.55
strawberry	2.5	5.25
vanilla	4	4.75

Also note that the join will only contain information about items that appear in both tables. Let's see an example. Suppose there is a table of reviews of some ice cream cones, and we have found the average review for each flavor.

```
reviews = Table().with_columns(
    'Flavor', make_array('vanilla', 'chocolate', 'vanilla',
    'chocolate'),
    'Stars', make_array(5, 3, 5, 4)
)
reviews
```

Flavor	Stars
vanilla	5
chocolate	3
vanilla	5
chocolate	4

```
average_review = reviews.groupby('Flavor', np.average)
average_review
```

Flavor	Stars average
chocolate	3.5
vanilla	5

We can join `cones` and `average_review` by providing the labels of the columns by which to join.

```
cones.join('Flavor', average_review, 'Flavor')
```

Flavor	Price	Stars average
chocolate	6.55	3.5
chocolate	5.75	3.5
vanilla	4.75	5

Notice how the strawberry cones have disappeared. None of the reviews are for strawberry cones, so there is nothing to which the `strawberry` rows can be joined. This might be a problem, or it might not be - that depends on the analysis we are trying to perform with the joined table.

[Interact](#)

Bike Sharing in the Bay Area

We end this chapter by using all the methods we have learned to examine a new and large dataset. We will also introduce `map_table`, a powerful visualization tool.

The [Bay Area Bike Share](#) service published a [dataset](#) describing every bicycle rental from September 2014 to August 2015 in their system. There were 354,152 rentals in all. The columns are:

- An ID for the rental
- Duration of the rental, in seconds
- Start date
- Name of the Start Station and code for Start Terminal
- Name of the End Station and code for End Terminal
- A serial number for the bike
- Subscriber type and zip code

```
trips = Table.read_table('trip.csv')
trips
```

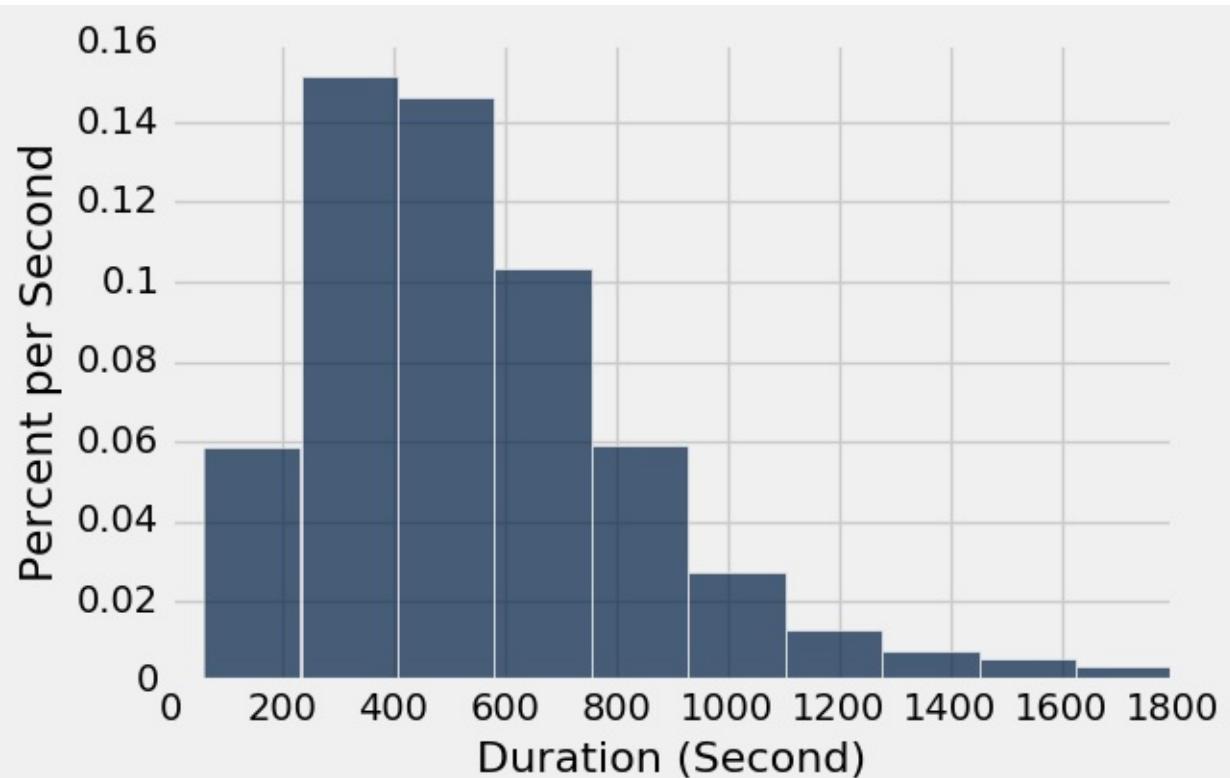
Trip ID	Duration	Start Date	Start Station	Start Terminal	End Date	End St
913460	765	8/31/2015 23:26	Harry Bridges Plaza (Ferry Building)	50	8/31/2015 23:39	San Francis Caltrain (Townse 4th)
913459	1036	8/31/2015 23:11	San Antonio Shopping Center	31	8/31/2015 23:28	Mounta View Ci Hall
913455	307	8/31/2015 23:13	Post at Kearny	47	8/31/2015 23:18	2nd at S Park
913454	409	8/31/2015 23:10	San Jose City Hall	10	8/31/2015 23:17	San Sa at 1st
913453	789	8/31/2015 23:09	Embarcadero at Folsom	51	8/31/2015 23:22	Embarc at Sans
913452	293	8/31/2015 23:07	Yerba Buena Center of the Arts (3rd @ Howard)	68	8/31/2015 23:12	San Francis Caltrain (Townse 4th)
913451	896	8/31/2015 23:07	Embarcadero at Folsom	51	8/31/2015 23:22	Embarc at Sans
913450	255	8/31/2015 22:16	Embarcadero at Sansome	60	8/31/2015 22:20	Steuart Market
913449	126	8/31/2015 22:12	Beale at Market	56	8/31/2015 22:15	Tempor Transba Termina (Howard Beale)
913448	932	8/31/2015 21:57	Post at Kearny	47	8/31/2015 22:12	South V Ness at Market

... (354142 rows omitted)

We'll focus only on the *free trips*, which are trips that last less than 1800 seconds (half an hour). There is a charge for longer trips.

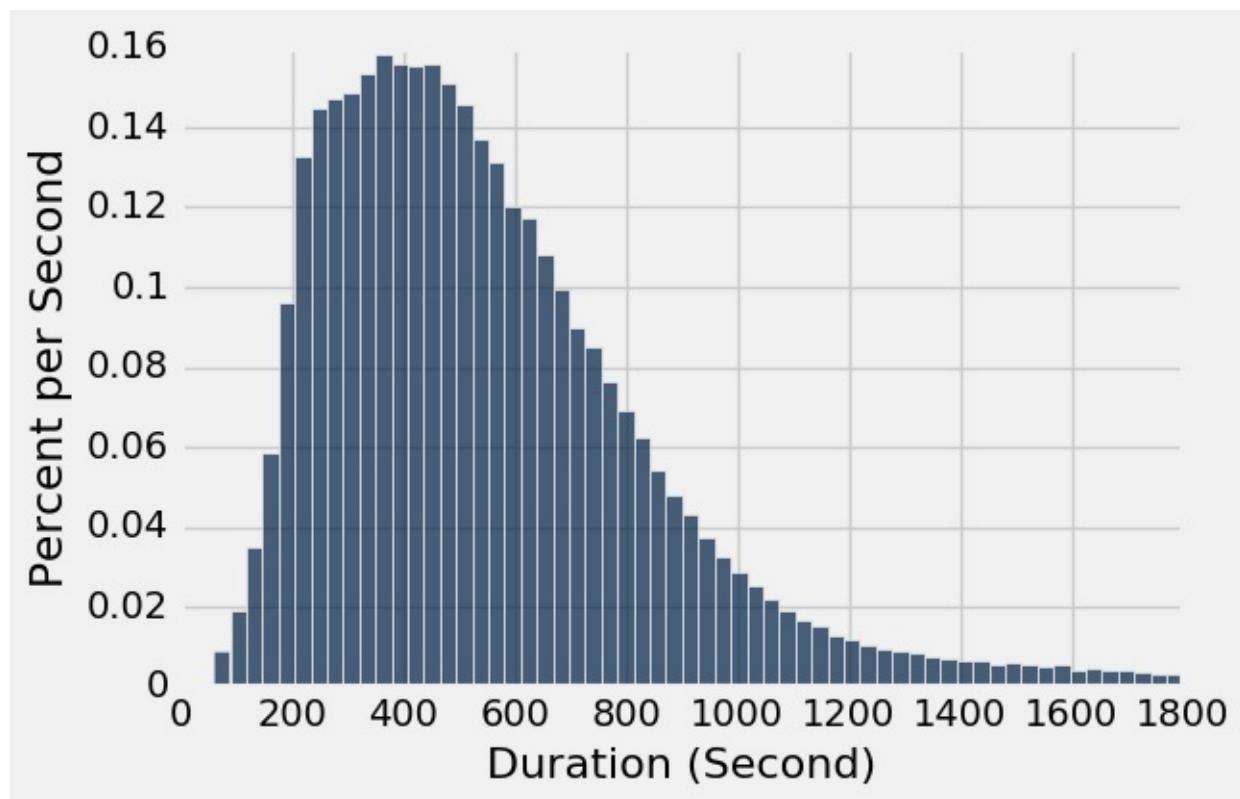
The histogram below shows that most of the trips took around 10 minutes (600 seconds) or so. Very few took near 30 minutes (1800 seconds), possibly because people try to return the bikes before the cutoff time so as not to have to pay.

```
commute = trips.where('Duration', are.below(1800))
commute.hist('Duration', unit='Second')
```



We can get more detail by specifying a larger number of bins. But the overall shape doesn't change much.

```
commute.hist('Duration', bins=60, unit='Second')
```



Exploring the Data with `group` and `pivot` ¶

We can use `group` to identify the most highly used Start Station:

```
starts = commute.groupby('Start Station').sort('count',
descending=True)
starts
```

Start Station	count
San Francisco Caltrain (Townsend at 4th)	25858
San Francisco Caltrain 2 (330 Townsend)	21523
Harry Bridges Plaza (Ferry Building)	15543
Temporary Transbay Terminal (Howard at Beale)	14298
2nd at Townsend	13674
Townsend at 7th	13579
Steuart at Market	13215
Embarcadero at Sansome	12842
Market at 10th	11523
Market at Sansome	11023

... (60 rows omitted)

The largest number of trips started at the Caltrain Station on Townsend and 4th in San Francisco. People take the train into the city, and then use a shared bike to get to their next destination.

The `group` method can also be used to classify the rentals by both Start Station and End Station.

```
commute.groupby(['Start Station', 'End Station'])
```

Start Station	End Station	count
2nd at Folsom	2nd at Folsom	54
2nd at Folsom	2nd at South Park	295
2nd at Folsom	2nd at Townsend	437
2nd at Folsom	5th at Howard	113
2nd at Folsom	Beale at Market	127
2nd at Folsom	Broadway St at Battery St	67
2nd at Folsom	Civic Center BART (7th at Market)	47
2nd at Folsom	Clay at Battery	240
2nd at Folsom	Commercial at Montgomery	128
2nd at Folsom	Davis at Jackson	28

... (1619 rows omitted)

Fifty-four trips both started and ended at the station on 2nd at Folsom. A much large number (437) were between 2nd at Folsom and 2nd at Townsend.

The `pivot` method does the same classification but displays its results in a contingency table that shows all possible combinations of Start and End Stations, even though some of them didn't correspond to any trips. Remember that the first argument of a `pivot` statement specifies the column labels of the pivot table; the second argument labels the rows.

There is a train station as well as a Bay Area Rapid Transit (BART) station near Beale at Market, explaining the high number of trips that start and end there.

```
commute.pivot('Start Station', 'End Station')
```

End Station	2nd at Folsom	2nd at South Park	2nd at Townsend	5th at Howard	Adobe on Almaden	Arena Green / SAP Center	Beale at Market
2nd at Folsom	54	190	554	107	0	0	40
2nd at South Park	295	164	71	180	0	0	208
2nd at Townsend	437	151	185	92	0	0	608
5th at Howard	113	177	148	83	0	0	59
Adobe on Almaden	0	0	0	0	11	4	0
Arena Green / SAP Center	0	0	0	0	7	64	0
Beale at Market	127	79	183	59	0	0	59
Broadway St at Battery St	67	89	279	119	0	0	102
California Ave Caltrain Station	0	0	0	0	0	0	0
Castro Street and El Camino Real	0	0	0	0	0	0	0

... (60 rows omitted)

We can also use `pivot` to find the shortest time of the rides between Start and End Stations. Here `pivot` has been given `Duration` as the optional `values` argument, and `min` as the function which to perform on the values in each cell.

```
commute.pivot('Start Station', 'End Station', 'Duration', min)
```

End Station	2nd at Folsom	2nd at South Park	2nd at Townsend	5th at Howard	Adobe on Almaden	Arena Green / SAP Center	Beale at Market
2nd at Folsom	61	97	164	268	0	0	271
2nd at South Park	61	60	77	86	0	0	78
2nd at Townsend	137	67	60	423	0	0	311
5th at Howard	215	300	384	68	0	0	351
Adobe on Almaden	0	0	0	0	84	275	0
Arena Green / SAP Center	0	0	0	0	305	62	0
Beale at Market	219	343	417	387	0	0	60
Broadway St at Battery St	351	424	499	555	0	0	194
California Ave Caltrain Station	0	0	0	0	0	0	0
Castro Street and El Camino Real	0	0	0	0	0	0	0

... (60 rows omitted)

Someone had a very quick trip (271 seconds, or about 4.5 minutes) from 2nd at Folsom to Beale at Market, about five blocks away. There are no bike trips between the 2nd Avenue stations and Adobe on Almaden, because the latter is in a different city.

Drawing Maps

The table `stations` contains geographical information about each bike station, including latitude, longitude, and a "landmark" which is the name of the city where the station is located.

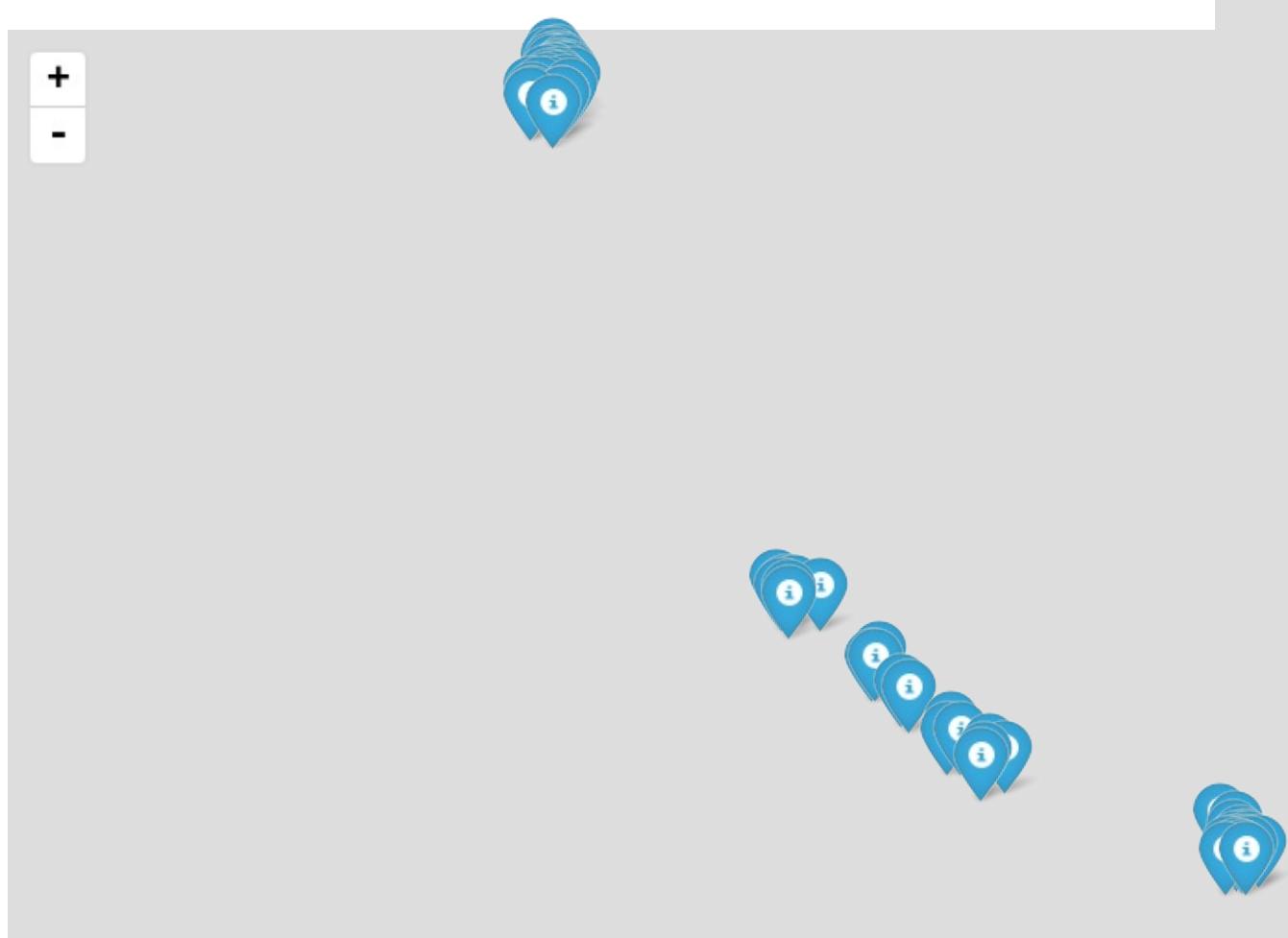
```
stations = Table.read_table('station.csv')
stations
```

station_id	name	lat	long	dockcount	landmark	installdate
2	San Jose Diridon Caltrain Station	37.3297	-121.902	27	San Jose	8/6/2011
3	San Jose Civic Center	37.3307	-121.889	15	San Jose	8/5/2011
4	Santa Clara at Almaden	37.334	-121.895	11	San Jose	8/6/2011
5	Adobe on Almaden	37.3314	-121.893	19	San Jose	8/5/2011
6	San Pedro Square	37.3367	-121.894	15	San Jose	8/7/2011
7	Paseo de San Antonio	37.3338	-121.887	15	San Jose	8/7/2011
8	San Salvador at 1st	37.3302	-121.886	15	San Jose	8/5/2011
9	Japantown	37.3487	-121.895	15	San Jose	8/5/2011
10	San Jose City Hall	37.3374	-121.887	15	San Jose	8/6/2011
11	MLK Library	37.3359	-121.886	19	San Jose	8/6/2011
... (60 rows omitted)						



We can draw a map of where the stations are located, using `Marker.map_table`. The function operates on a table, whose columns are (in order) latitude, longitude, and an optional identifier for each point.

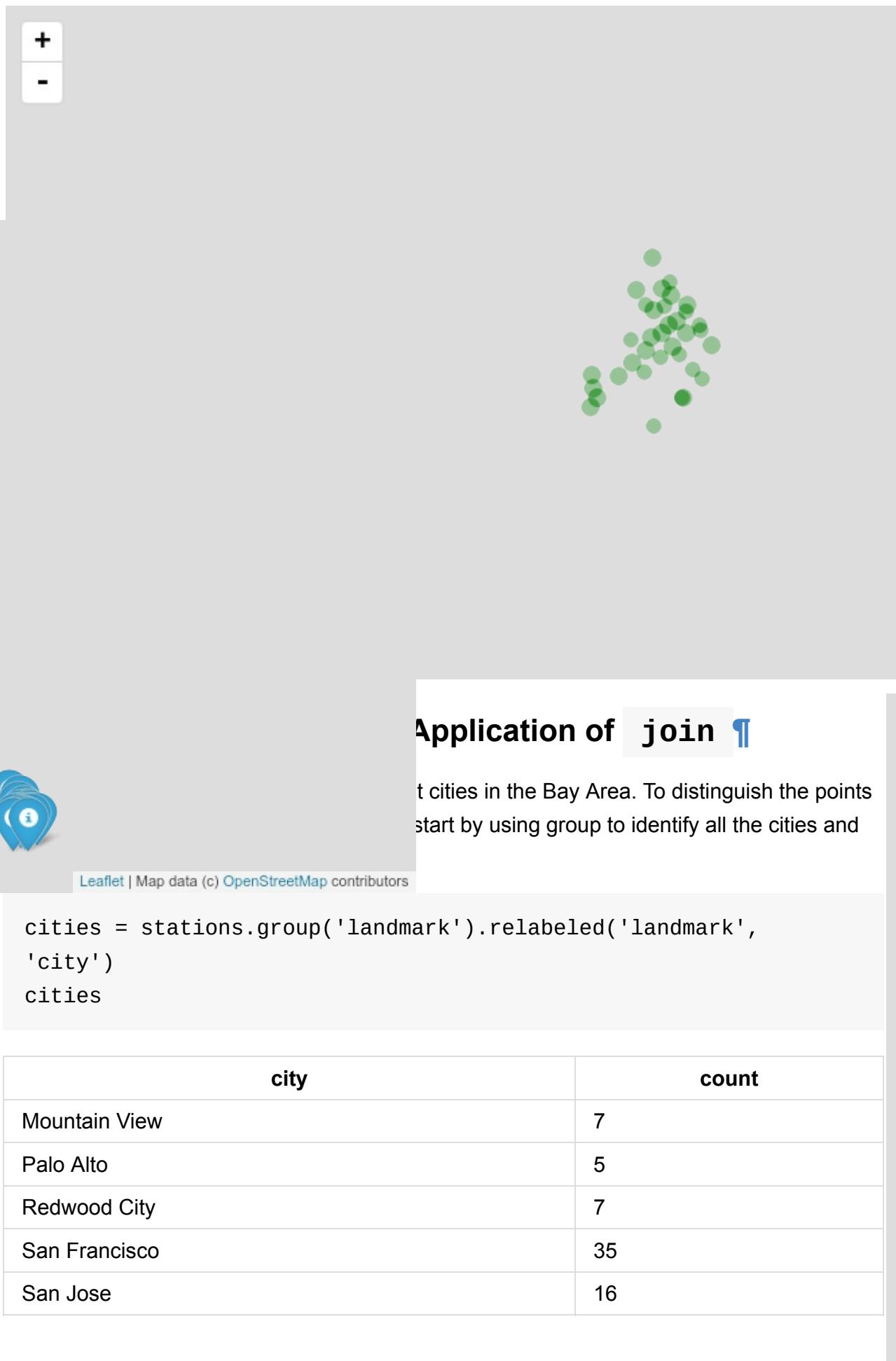
```
Marker.map_table(stations.select('lat', 'long', 'name'))
```



The map is created using [OpenStreetMap](#), which is an open online mapping system that you can use just as you would use Google Maps or any other online map. Zoom in to San Francisco to see how the stations are distributed. Click on a marker to see which station it is.

You can also represent points on a map by colored circles. Here is such a map of the San Francisco bike stations.

```
sf = stations.where('landmark', are.equal_to('San Francisco'))
sf_map_data = sf.select('lat', 'long', 'name')
Circle.map_table(sf_map_data, color='green', radius=200)
```



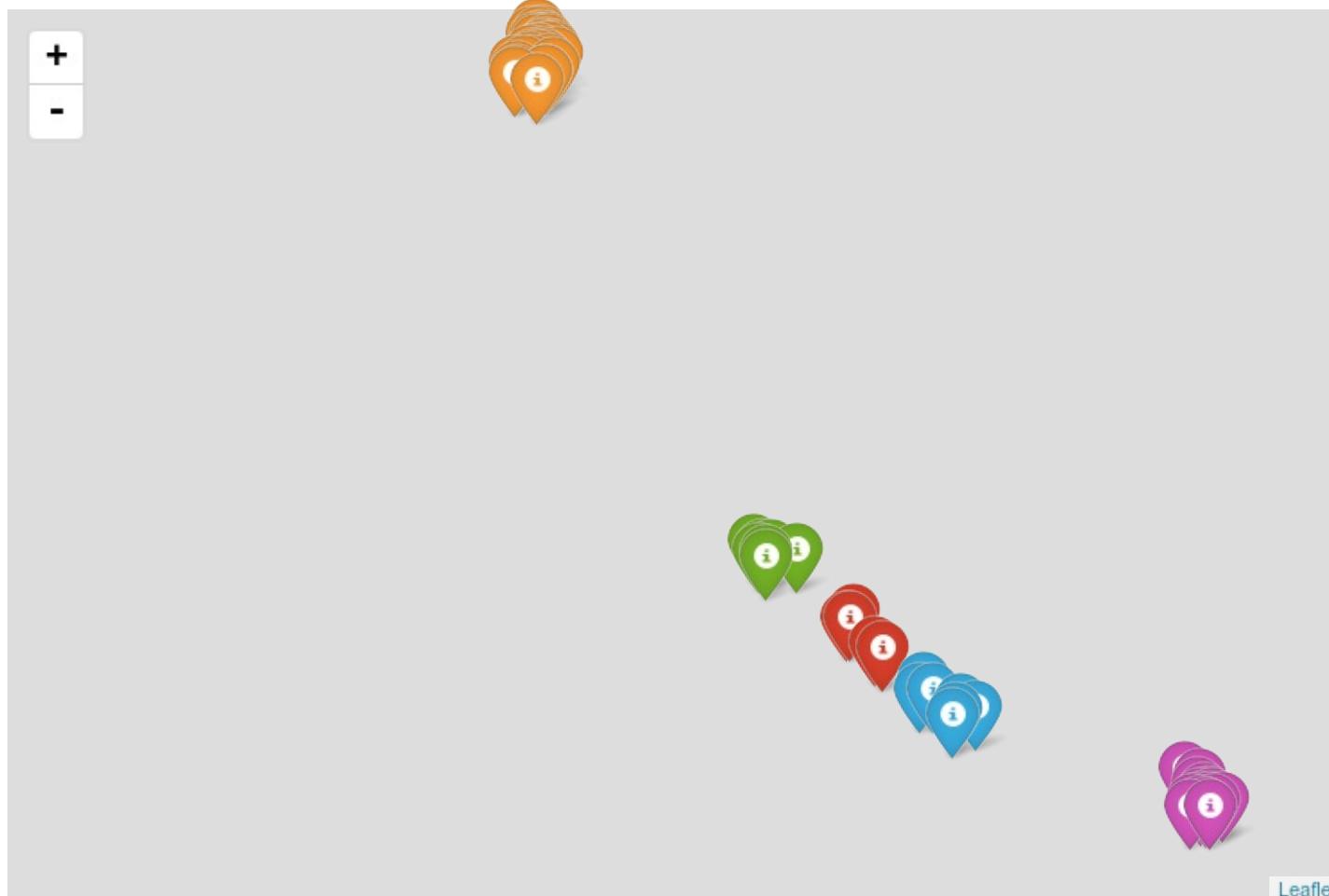
```
color', make_array('blue', 'red',
```

count	color
7	blue
5	red
7	green
35	orange
16	purple

landmark , and then select the columns we

```
rk', colors, 'city')  
'long', 'name', 'color')
```

Leaflet | Map data (c) OpenStreetMap contributors



Now the markers have five different colors for the five different cities.

To see where most of the bike rentals originate, let's identify the start stations:

```
starts = commute.groupby('Start Station').sort('count',
descending=True)
starts
```

Start Station	count
San Francisco Caltrain (Townsend at 4th)	25858
San Francisco Caltrain 2 (330 Townsend)	21523
Harry Bridges Plaza (Ferry Building)	15543
Temporary Transbay Terminal (Howard at Beale)	14298
2nd at Townsend	13674
Townsend at 7th	13579
Steuart at Market	13215
Embarcadero at Sansome	12842
Market at 10th	11523
	11023



ed to map these stations, by first joining `starts`

```
('name', starts, 'Start Station')
```



name	station_id	lat	long	dockcount	landmark	installa
2nd at Folsom	62	37.7853	-122.396	19	San Francisco	8/22/20
2nd at South Park	64	37.7823	-122.393	15	San Francisco	8/22/20
2nd at Townsend	61	37.7805	-122.39	27	San Francisco	8/22/20
5th at Howard	57	37.7818	-122.405	15	San Francisco	8/21/20
Adobe on Almaden	5	37.3314	-121.893	19	San Jose	8/5/201
Arena Green / SAP Center	14	37.3327	-121.9	19	San Jose	8/5/201
Beale at Market	56	37.7923	-122.397	19	San Francisco	8/20/20
Broadway St at Battery St	82	37.7985	-122.401	15	San Francisco	1/22/20
California Ave Caltrain Station	36	37.4291	-122.143	15	Palo Alto	8/14/20
Castro Street and El Camino Real	32	37.386	-122.084	11	Mountain View	12/31/20

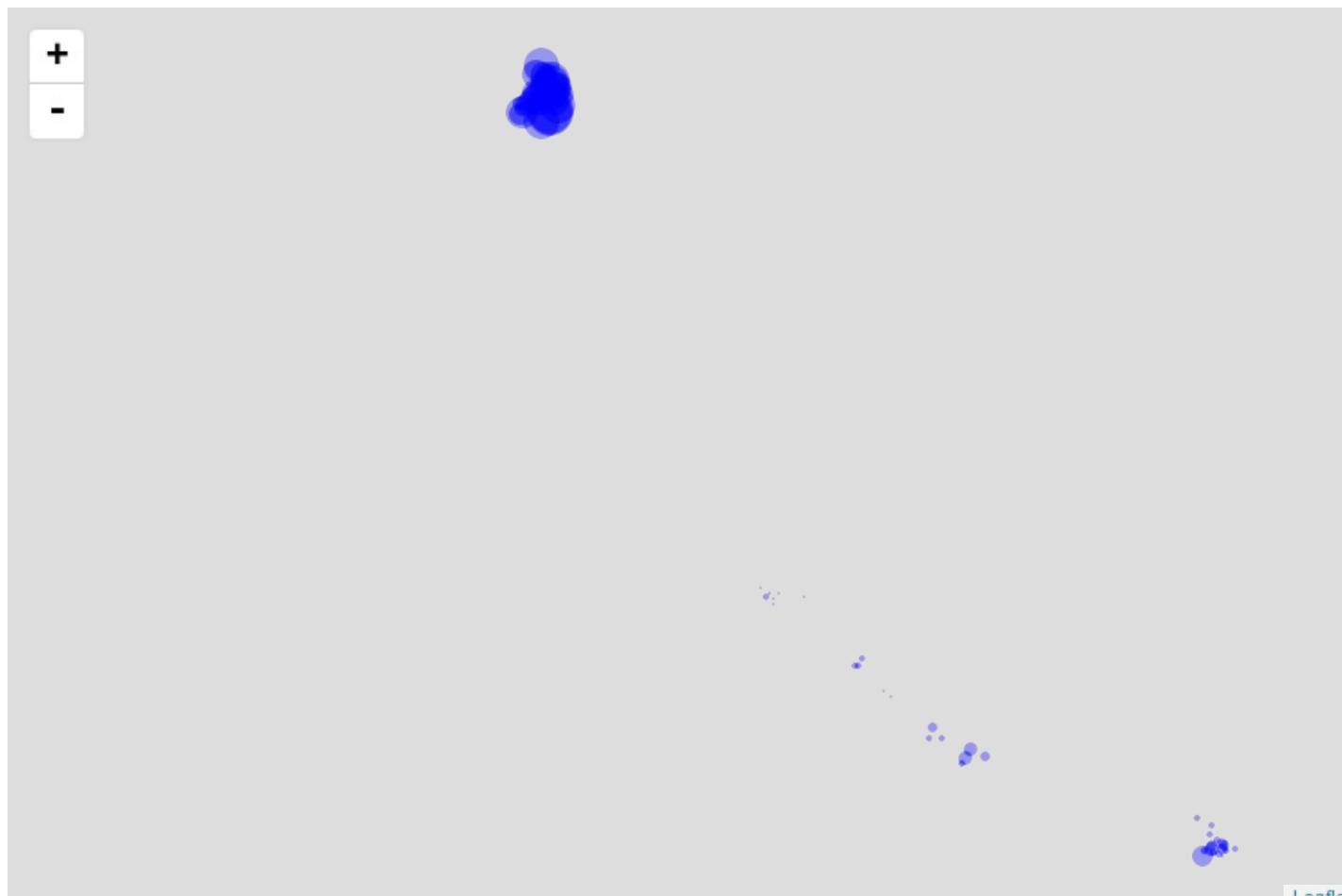
... (58 rows omitted)

Now we extract just the data needed for drawing our map, adding a color and an area to each station. The area is 1000 times the count of the number of rentals starting at each station, where the constant 1000 was chosen so that the circles would appear at an appropriate scale on the map.

```
starts_map_data = station_starts.select('lat', 'long',
    'name').with_columns(
    'color', 'blue',
    'area', station_starts.column('count') * 1000
)
starts_map_data.show(3)
Circle.map_table(starts_map_data)
```

lat	long	name	color	area
37.7853	-122.396	2nd at Folsom	blue	7841000
37.7823	-122.393	2nd at South Park	blue	9274000
37.7805	-122.39	2nd at Townsend	blue	13674000

... (65 rows omitted)



That huge blob in San Francisco shows that the eastern section of the city is the unrivaled capital of bike rentals in the Bay Area.



[Interact](#)

Randomness

In the previous chapters we developed skills needed to make insightful descriptions of data. Data scientists also have to be able to understand randomness. For example, they have to be able to assign individuals to treatment and control groups at random, and then try to say whether any observed differences in the outcomes of the two groups are simply due to the random assignment or genuinely due to the treatment.

In this chapter, we begin our analysis of randomness. To start off, we will use Python to make choices at random. In `numpy` there is a sub-module called `random` that contains many functions that involve random selection. One of these functions is called `choice`. It picks one item at random from an array, and it is equally likely to pick any of the items. The function call is `np.random.choice(array_name)`, where `array_name` is the name of the array from which to make the choice.

Thus the following code evaluates to `treatment` with chance 50%, and `control` with chance 50%.

```
two_groups = make_array('treatment', 'control')
np.random.choice(two_groups)
```

```
'treatment'
```

The big difference between the code above and all the other code we have run thus far is that the code above doesn't always return the same value. It can return either `treatment` or `control`, and we don't know ahead of time which one it will pick. We can repeat the process by providing a second argument, the number of times to repeat the process.

```
np.random.choice(two_groups, 10)
```

```
array(['treatment', 'control', 'treatment', 'control',
       'control',
       'treatment', 'treatment', 'control', 'control',
       'control'],
      dtype='<U9')
```

A fundamental question about random events is whether or not they occur. For example:

- Did an individual get assigned to the treatment group, or not?
- Is a gambler going to win money, or not?
- Has a poll made an accurate prediction, or not?

Once the event has occurred, you can answer "yes" or "no" to all these questions. In programming, it is conventional to do this by labeling statements as True or False. For example, if an individual did get assigned to the treatment group, then the statement, "The individual was assigned to the treatment group" would be `True`. If not, it would be `False`.

Booleans and Comparison

In Python, Boolean values, named for the logician [George Boole](#), represent truth and take only two possible values: `True` and `False`. Whether problems involve randomness or not, 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><</code>	<code>2 < 3</code>	<code>2 < 2</code>
Greater than	<code>></code>	<code>3 > 2</code>	<code>3 > 3</code>
Less than or equal	<code><=</code>	<code>2 <= 2</code>	<code>3 <= 2</code>
Greater or equal	<code>>=</code>	<code>3 >= 3</code>	<code>2 >= 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>

Notice the two equal signs `==` in the comparison to determine equality. This is necessary because Python already uses `=` to mean assignment to a name, as we have seen. It can't use the same symbol for a different purpose. Thus if you want to check whether 5 is equal to the `10/2`, then you have to be careful: `5 = 10/2` returns an error message because Python assumes you are trying to assign the value of the expression `10/2` to a name that is the numeral 5. Instead, you must use `5 == 10/2`, which evaluates to `True`.

```
5 = 10/2
```

```
File "<ipython-input-4-5c7d3e808777>", line 1
 5 = 10/2
      ^
SyntaxError: can't assign to literal
```

```
5 == 10/2
```

```
True
```

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
```

Comparing Strings

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'
```

Let's return to random selection. Recall the array `two_groups` which consists of just two elements, `treatment` and `control`. To see whether a randomly assigned individual went to the treatment group, you can use a comparison:

```
np.random.choice(two_groups) == 'treatment'
```

```
False
```

As before, the random choice will not always be the same, so the result of the comparison won't always be the same either. It will depend on whether `treatment` or `control` was chosen. With any cell that involves random selection, it is a good idea to run the cell several times to get a sense of the variability in the result.

Comparing an Array and a Value

Recall that we can perform arithmetic operations on many numbers in an array at once. For example, `make_array(0, 5, 2)*2` is equivalent to `make_array(0, 10, 4)`. In similar fashion, if we compare an array and one value, each element of the array is compared to that value, and the comparison evaluates to an array of Booleans.

```
tosses = make_array('Tails', 'Heads', 'Tails', 'Heads', 'Heads')
tosses == 'Heads'
```

```
array([False,  True, False,  True,  True], dtype=bool)
```

The `numpy` method `count_nonzero` evaluates to the number of non-zero (that is, `True`) elements of the array.

```
np.count_nonzero(tosses == 'Heads')
```

```
3
```


[Interact](#)

Conditional Statements

In many situations, actions and results depends on a specific set of conditions being satisfied. For example, individuals in randomized controlled trials receive the treatment if they have been assigned to the treatment group. A gambler makes money if she wins her bet.

In this section we will learn how to describe such situations using code. 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.

Let us start defining a function that returns the sign of a number.

```
def sign(x):  
  
    if x > 0:  
        return 'Positive'
```

```
sign(3)
```

```
'Positive'
```

This function returns the correct sign if the input is a positive number. But if the input is not a positive number, then the *if expression* evaluates to a false value, and so the `return` statement is skipped and the function call has no value.

```
sign(-3)
```

So let us refine our function to return `Negative` if the input is a negative number. We can do this by adding an `elif` clause, where `elif` is Python's shorthand for the phrase "else, if".

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'
```

Now `sign` returns the correct answer when the input is -3:

```
sign(-3)
```

```
'Negative'
```

What if the input is 0? To deal with this case, we can add another `elif` clause:

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'

    elif x == 0:
        return 'Neither positive nor negative'
```

```
sign(0)
```

```
'Neither positive nor negative'
```

Equivalently, we can replace the final `elif` clause by an `else` clause, whose body will be executed only if all the previous comparisons are false; that is, if the input value is equal to 0.

```
def sign(x):

    if x > 0:
        return 'Positive'

    elif x < 0:
        return 'Negative'

    else:
        return 'Neither positive nor negative'
```

```
sign(0)
```

```
'Neither positive nor negative'
```

The General Form

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).

Example: "The Other One"

We will now use conditional statements to define a function that seems rather artificial and contrary, but will come in handy later in the chapter. It takes an array with two elements (for example, `red` and `blue`), and another element to compare. If that element is `red`, the function returns `blue`. If the element is (for example) `blue`, the function returns `red`. That is why we'll call the function `other_one`.

```
def other_one(x, a_b):  
  
    """Compare x with the two elements of a_b;  
    if it is equal to one of them, return the other one;  
    if it is not equal to either of them, return an error  
message.  
    """  
  
    if x == a_b.item(0):  
        return a_b.item(1)  
  
    elif x == a_b.item(1):  
        return a_b.item(0)  
  
    else:  
        return 'The input is not valid.'
```

```
colors = make_array('red', 'blue')  
other_one('red', colors)
```

```
'blue'
```

```
other_one('blue', colors)
```

```
'red'
```

```
other_one('potato', colors)
```

```
'The input is not valid.'
```

[Interact](#)

Iteration

It is often the case in programming – especially when dealing with randomness – that we want to repeat a process multiple times. For example, we might want to assign each person in a study to the treatment group or to control, based on tossing a coin. We can do this without actually tossing a coin for each person; we can just use `np.random.choice` instead.

Here is a reminder of how `np.random.choice` works. Run the cell a few times to see how the output changes.

```
np.random.choice(['Heads', 'Tails'])
```

```
'Heads'
```

To come up with Heads or Tails for each individual in our study, we could copy-paste the code multiple times, but that's tedious and prone to typos, and if we wanted to do it a thousand times or a million times, forget it.

A more automated solution is to use a `for` statement to loop over the contents of a sequence. This is called *iteration*. A `for` statement begins with the word `for`, followed by a name we want to give each 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(3):
    print(i)
```

```
0
1
2
```

It is instructive to imagine code that exactly replicates a `for` statement without the `for` statement. This is called *unrolling* the loop.

A `for` statement simple replicates the code inside it, but before each iteration, it assigns a new value from the given sequence to the name we chose. For example, here is an unrolled version of the loop above:

```
i = np.arange(3).item(0)
print(i)
i = np.arange(3).item(1)
print(i)
i = np.arange(3).item(2)
print(i)
```

```
0
1
2
```

Notice that the name `i` is arbitrary, just like any name we assign with `=`.

Here we use a `for` statement in a more realistic way: we print 5 random choices from `coin`, thus *simulating* the results five tosses of a coin. We use the word *simulating* to remind ourselves that we are not physically tossing coins but using Python to mimic the process.

```
coin = make_array('Heads', 'Tails')

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

```
Heads
Heads
Tails
Heads
Heads
```

In this case, we simply perform exactly the same (random) action several times, so the code inside our `for` statement does not actually refer to `i`.

Augmenting Arrays

While the `for` statement above does simulate the results of five tosses of a coin, the results are simply printed and aren't in a form that we can use for computation. Thus a typical use of a `for` statement is to create an array of results, by augmenting it each time.

The `append` method in `numpy` helps us do this. The call `np.append(array_name, value)` evaluates to a new array that is `array_name` augmented by `value`. When you use `append`, keep in mind that all the entries of an array must have the same type.

```
pets = make_array('Cat', 'Dog')
np.append(pets, 'Another Pet')
```

```
array(['Cat', 'Dog', 'Another Pet'],
      dtype='<U11')
```

This keeps the array `pets` unchanged:

```
pets
```

```
array(['Cat', 'Dog'],
      dtype='<U3')
```

But often while using `for` loops it will be convenient to mutate an array – that is, change it – when augmenting it. This is done by assigning the augmented array to the same name as the original.

```
pets = np.append(pets, 'Another Pet')
pets
```

```
array(['Cat', 'Dog', 'Another Pet'],
      dtype='<U11')
```

Example: Counting the Number of Heads

We can now simulate five tosses of a coin and place the results into an array. We will start by creating an empty array and then appending the outcome of each toss. Notice that the body of the `for` loop contains two statements. Both statements are executed for each value in the given sequence `np.arange(5)`.

```
coin = make_array('Heads', 'Tails')

outcomes = make_array()

for i in np.arange(5):
    outcome_of_toss = np.random.choice(coin)
    outcomes = np.append(outcomes, outcome_of_toss)

outcomes
```

```
array(['Heads', 'Tails', 'Heads', 'Heads', 'Tails'],
      dtype='<U32')
```

Let us rewrite the cell with the `for` statement unrolled:

```
coin = make_array('Heads', 'Tails')

outcomes = make_array()

i = np.arange(5).item(0)
outcome_of_toss = np.random.choice(coin)
outcomes = np.append(outcomes, outcome_of_toss)

i = np.arange(5).item(1)
outcome_of_toss = np.random.choice(coin)
outcomes = np.append(outcomes, outcome_of_toss)

i = np.arange(5).item(2)
outcome_of_toss = np.random.choice(coin)
outcomes = np.append(outcomes, outcome_of_toss)

i = np.arange(5).item(3)
outcome_of_toss = np.random.choice(coin)
outcomes = np.append(outcomes, outcome_of_toss)

i = np.arange(5).item(4)
outcome_of_toss = np.random.choice(coin)
outcomes = np.append(outcomes, outcome_of_toss)

outcomes
```

```
array(['Heads', 'Heads', 'Tails', 'Tails', 'Heads'],
      dtype='<U32')
```

By capturing the results in an array we have given ourselves the ability to use array methods to do computations. For example, we can use `np.count_nonzero` to count the number of heads in the five tosses.

```
np.count_nonzero(outcomes == 'Heads')
```

Keep in mind that we have used the `for` loop to simulate a random experiment, and therefore if you run the cell again, the array `outcomes` is likely to be different. In upcoming sections of the course we will study how different the outcomes could be.

Iteration is a powerful technique. For example, by running exactly the same code for 1000 tosses instead of 5, we can count the number of heads in 1000 tosses.

```
outcomes = make_array()

for i in np.arange(1000):
    outcome_of_toss = np.random.choice(coin)
    outcomes = np.append(outcomes, outcome_of_toss)

np.count_nonzero(outcomes == 'Heads')
```

521

[Interact](#)

Simulation

Simulation is the process of using a computer to mimic a physical experiment. In this class, those experiments will almost invariably involve chance.

We have seen how to simulate the results of tosses of a coin. The steps in that simulation were examples of the steps that will constitute every simulation we do in this course. In this section we will set out those steps and follow them in examples.

Step 1: What to Simulate

Specify the quantity you want to simulate. For example, you might decide that you want to simulate the outcomes of tosses of a coin.

Step 2: Simulating One Value

Figure out how to simulate *one* value of the quantity you specified in Step 1. In our example, you have to figure out how to simulate the outcome of *one* toss of a coin. If your quantity is more complicated, you might need several lines of code to come up with one simulated value.

Step 3: Number of Repetitions

Decide how many times you want to simulate the quantity. You will have to repeat Step 2 that many times. In one of our earlier examples we had decided to simulate the outcomes of 1000 tosses of a coin, and so we needed 1000 repetitions of generating the outcome of a single toss.

Step 4: Coding the Simulation

Put it all together in code.

- Create an empty array in which to collect all the simulated values. We will call this the collection array.
- Create a "repetitions sequence," that is, a sequence whose length is the number of repetitions you specified in Step 3. For `n` repetitions we will almost always use the sequence `np.arange(n)`.
- Create a `for` loop. For each element of the repetitions sequence:
 - Simulate *one* value based on the code you developed in Step 2.
 - Augment the collection array with this simulated value.

That's it! Once you have carried out the steps above, your simulation is done. The collection array contains all the simulated values.

At this point you can use the collection array as you would any other array. You can visualize the distribution of the simulated values, count how many simulated values fall into a particular category, and so on.

Number of Heads in 100 Tosses

It is natural to expect that in 100 tosses of a coin, there will be 50 heads, give or take a few.

But how many is "a few"? What's the chance of getting exactly 50 heads? Questions like these matter in data science not only because they are about interesting aspects of randomness, but also because they can be used in analyzing experiments where assignments to treatment and control groups are decided by the toss of a coin.

In this example we will simulate the number of heads in 100 tosses of a coin. The histogram of our results will give us some insight into how many heads are likely.

Let's get started on the simulation, following the steps above.

Step 1: What to Simulate

The quantity we are going to simulate is the number of heads in 100 tosses.

Step 2: Simulating One Value

We have to figure out how to make one set of 100 tosses and count the number of heads. Let's start by creating a coin.

```
coin = make_array('Heads', 'Tails')
```

In our earlier example we used `np.random.choice` and a `for` loop to generate multiple tosses. But sets of coin tosses are needed so often in data science that `np.random.choice` simulates them for us if we include a second argument that is the number of times to toss.

Here are the results of 10 tosses.

```
ten_tosses = np.random.choice(coin, 10)
ten_tosses
```

```
array(['Tails', 'Tails', 'Tails', 'Tails', 'Tails', 'Tails',
       'Heads',
       'Tails', 'Tails', 'Heads'],
      dtype='<U5')
```

We can count the number of heads by using `np.count_nonzero` as before:

```
np.count_nonzero(ten_tosses == 'Heads')
```

```
2
```

Our goal is to simulate the number of heads in 100 tosses, not 10. To do that we can just repeat the same code, replacing 10 by 100.

```
outcomes = np.random.choice(coin, 100)
num_heads = np.count_nonzero(outcomes == 'Heads')
num_heads
```

```
49
```

Step 3: Number of Repetitions

How many repetitions we want is up to us. The more we use, the more reliable our simulations will be, but the longer it will take to run the code. Python is pretty fast at tossing coins. Let's go for 10,000 repetitions. That means we are going to do the following 10,000 times:

- Toss a coin 100 times and count the number of heads.

That's a lot of tossing! It's good that we have Python to do it for us.

Step 4: Coding the Simulation

We are ready to write the code to execute the entire simulation.

```
# An empty array to collect the simulated values
heads = make_array()

# Repetitions sequence
num_repetitions = 10000
repetitions_sequence = np.arange(num_repetitions)

# for loop
for i in repetitions_sequence:

    # simulate one value
    outcomes = np.random.choice(coin, 100)
    num_heads = np.count_nonzero(outcomes == 'Heads')

    # augment the collection array with the simulated value
    heads = np.append(heads, num_heads)

# That's it! The simulation is done.
```

Check that the array `heads` contains 10,000 entries, one for each repetition of the experiment.

```
len(heads)
```

```
10000
```

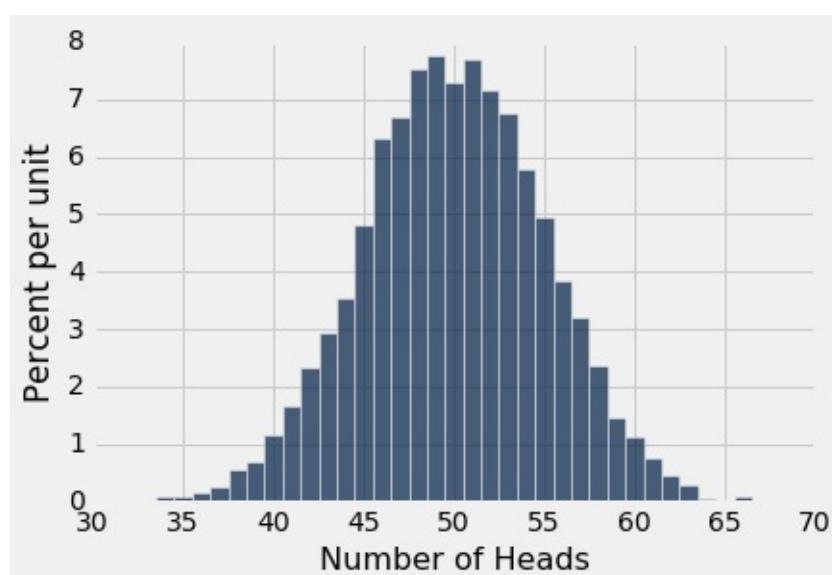
To get a sense of the variability in the number of heads in 100 tosses, we can collect the results in a table and draw a histogram.

```
simulation_results = Table().with_column(
    'Repetition', np.arange(1, num_repetitions + 1),
    'Number of Heads', heads
)
```

```
simulation_results
```

Repetition	Number of Heads
1	51
2	57
3	38
4	51
5	49
6	55
7	50
8	57
9	48
10	44
... (9990 rows omitted)	

```
simulation_results.hist('Number of Heads', bins =
np.arange(30.5, 69.6, 1))
```



Each bins has width 1 and is centered at each value of the number of heads.

Not surprisingly, the histogram looks roughly symmetric around 50 heads. The height of the bar at 50 is about 8% per unit. Since each bin is 1 unit wide, this is the same as saying that about 8% of the repetitions produced exactly 50 heads. That's not a huge percent, but it's the largest compared to the percent at every other number of heads.

The histogram also shows that in almost all of the repetitions, the number of heads in 100 tosses was somewhere between 35 and 65. Indeed, the bulk of the repetitions produced numbers of heads in the range 45 to 55.

While in theory it is *possible* that the number of heads can be anywhere between 0 and 100, the simulation shows that the range of *probable* values is much smaller.

This is an instance of a more general phenomenon about the variability in coin tossing, as we will see later in the course.

A More Compact Version of the Code

We wrote the code for the simulation to show each of the steps in detail. Here are the same steps written in a more compact form. You can see that the code starts out the same way as before, but then some steps are combined.

```
heads = make_array()

num_repetitions = 10000

for i in np.arange(num_repetitions):
    outcomes = np.random.choice(coin, 100)
    heads = np.append(heads, np.count_nonzero(outcomes ==
'Heads'))
```

```
heads
```

```
array([ 42.,  52.,  41., ...,  57.,  56.,  61.])
```

Moves in Monopoly

Each move in the game Monopoly is determined by the total number of spots of two rolls of a die. If you play Monopoly, what should you expect to get when you roll the die two times?

We can explore this by simulating the sum of two rolls of a die. We will run the simulation 10,000 times as we did in the previous example. Notice that in this paragraph we have completed Steps 1 and 3 of our simulation process.

Step 2 is the one in which we simulate one pair of rolls and add up the number of spots.

```
die = np.arange(1, 7)
sum(np.random.choice(die, 2))
```

```
4
```

That simulates one value of the sum of two rolls. We are now all set to run the simulation according to the steps that are now familiar.

```
moves = make_array()

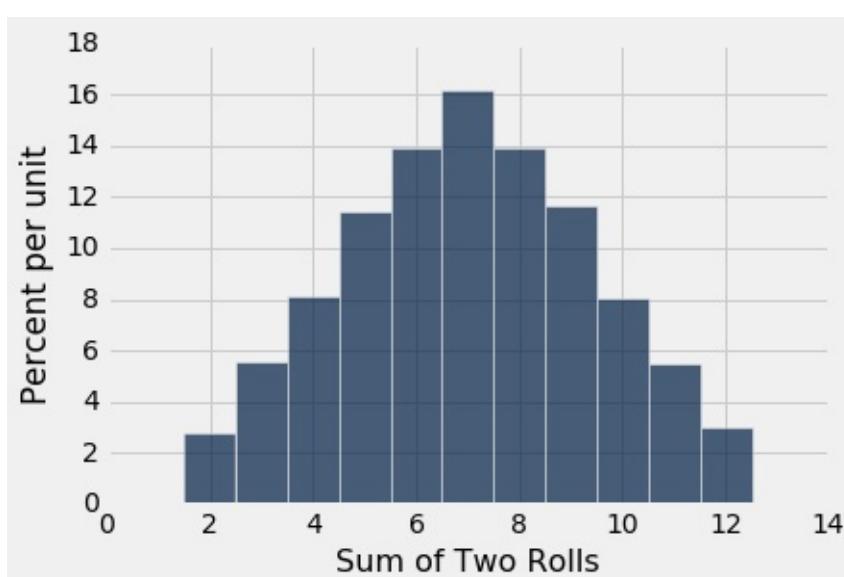
num_repetitions = 10000

for i in np.arange(num_repetitions):
    one_move = sum(np.random.choice(die, 2))
    moves = np.append(moves, one_move)
```

Here is a histogram of the results.

```
results = Table().with_column(
    'Repetition', np.arange(1, num_repetitions + 1),
    'Sum of Two Rolls', moves
)

results.hist('Sum of Two Rolls', bins = np.arange(1.5, 12.6, 1))
```



Seven is the most common value, with the frequencies falling off symmetrically on either side.

[Interact](#)

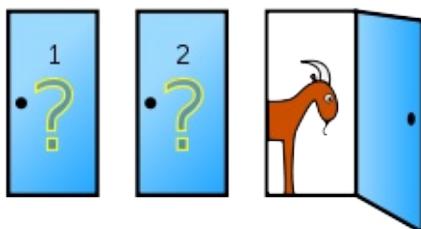
The Monty Hall Problem

This [problem](#) has flummoxed many people over the years, [mathematicians included](#). Let's see if we can work it out by simulation.

The setting is derived from a television game show called "Let's Make a Deal". Monty Hall hosted this show in the 1960's, and it has since led to a number of spin-offs. An exciting part of the show was that while the contestants had the chance to win great prizes, they might instead end up with "zonks" that were less desirable. This is the basis for what is now known as *the Monty Hall problem*.

The setting is a game show in which the contestant is faced with three closed doors. Behind one of the doors is a fancy car, and behind each of the other two there is a goat. The contestant doesn't know where the car is, and has to attempt to find it under the following rules.

- The contestant makes an initial choice, but that door isn't opened.
- At least one of the other two doors must have a goat behind it. Monty opens one of these doors to reveal a goat, displayed in all its glory in [Wikipedia](#):



- There are two doors left, one of which was the contestant's original choice. One of the doors has the car behind it, and the other one has a goat. The contestant now gets to choose which of the two doors to open.

The contestant has a decision to make. Which door should she choose to open, if she wants the car? Should she stick with her initial choice, or switch to the other door? That is the Monty Hall problem.

The Solution

In any problem involving chances, the assumptions about randomness are important. It's reasonable to assume that there is a $1/3$ chance that the contestant's initial choice is the door that has the car behind it.

The solution to the problem is quite straightforward under this assumption, though the straightforward solution doesn't convince everyone. Here it is anyway.

- The chance that the car is behind the originally chosen door is 1/3.
- The car is behind either the originally chosen door or the door that remains. It can't be anywhere else.
- Therefore, the chance that the car is behind the door that remains is 2/3.
- Therefore, the contestant should switch.

That's it. End of story.

Not convinced? Then let's simulate the game and see how the results turn out.

Simulation

The simulation will be more complex than those we have done so far. Let's break it down.

Step 1: What to Simulate

For each play we will simulate what's behind all three doors:

- the one the contestant first picks
- the one that Monty throws out
- the remaining door So we will be keeping track of three quantities, not just one.

Step 2: Simulating One Play

The bulk of our work consists of simulating one play of the game. This involves several pieces.

The Doors

We start by setting up two useful arrays – `doors` and `goats` – that will allow us to distinguish the three doors and the two goats.

```
doors = make_array('Car', 'Goat 1', 'Goat 2')
goats = make_array('Goat 1', 'Goat 2')
```

Identifying Goats

We are going to have to recognize whether a door has a goat behind it or not. We can just label each door with what's behind it; the contestant can't see what we are doing!

The function `is_goat` takes a door label and returns a Boolean signifying whether or not it is a goat.

```
def is_goat(door_name):  
  
    if door_name == "Goat 1":  
        return True  
    elif door_name == "Goat 2":  
        return True  
    else:  
        return False
```

Let's check that this function can indeed tell goats from cars.

```
is_goat('Goat 1')
```

```
True
```

```
is_goat('Goat 2')
```

```
True
```

```
is_goat('Car')
```

```
False
```

One Play

If the contestant's original choice is a door with a goat, Monty must throw out the other goat, and what will remain is the car. If the original choice is the door with a car, Monty must throw out one of the two goats, and what will remain is the other goat.

It is clear, therefore, that the function `other_one` defined in an earlier section will be useful. It takes a string and a two-element array; if the string is equal to one of the elements, it returns the other one.

```
def other_one(x, a_b):
    if x == a_b.item(0):
        return a_b.item(1)
    elif x == a_b.item(1):
        return a_b.item(0)
    else:
        return 'Input Not Valid'
```

If the contestant's original choice is a goat, then the outcome of the game could be one of the following two:

```
original = 'Goat 1'
make_array(original, other_one(original, goats), 'Car')
```

```
array(['Goat 1', 'Goat 2', 'Car'],
      dtype='<U6')
```

```
original = 'Goat 2'
make_array(original, other_one(original, goats), 'Car')
```

```
array(['Goat 2', 'Goat 1', 'Car'],
      dtype='<U6')
```

If the original choice happens to be the car, then let's assume Monty throws out one of the two goats at random, and the other goat is behind the remaining door.

```
original = 'Car'
throw_out = np.random.choice(goats)
make_array(original, throw_out, other_one(throw_out, goats))
```

```
array(['Car', 'Goat 1', 'Goat 2'],
      dtype='<U6')
```

A Function to Simulate One Play

Now we define a function `monty_hall` that simulates the game and returns an array of three strings in this order:

- what is behind the contestant's original choice of door
- what Monty throws out
- what is behind the remaining door

We can now put all this code together into a single function `monty_hall` to simulate the result of one game. The function takes no arguments.

The contestant's original choice will be a door chosen at random from among the three doors.

To check whether the original choice is a goat or not, we first write a little function named `is_goat`.

```
def monty_hall():

    """ Play the Monty Hall game once
    and return an array of three strings:

    original choice, what Monty throws out, what remains
    """

    original = np.random.choice(doors)

    if is_goat(original):
        return make_array(original, other_one(original, goats),
'Car')

    else:
        throw_out = np.random.choice(goats)
        return make_array(original, throw_out,
other_one(throw_out, goats))
```

Let's play the game a few times! Here is one outcome. You should run the cell several times to see how the outcome changes.

```
monty_hall()
```

```
array(['Car', 'Goat 1', 'Goat 2'],
      dtype='<U6')
```

Step 3: Number of Repetitions¶

To gauge the frequency with which the different outcomes occur, we have to play the games many times and collect the results. Let's run 10,000 repetitions.

Step 4: Coding the Simulation¶

It's time to run the simulation. We will start by defining three empty arrays, one each for the original choice, what Monty throws out, and what remains.

```
original = make_array()      # original choice
throw_out = make_array()     # what Monty throws out
remains = make_array()       # what remains

num_repetitions = 10000

for i in np.arange(num_repetitions):
    result = monty_hall()    # the result of one game

    # Collect the results in the appropriate arrays
    original = np.append(original, result.item(0))
    throw_out = np.append(throw_out, result.item(1))
    remains = np.append(remains, result.item(2))
```

The simulation is done. As always, the majority of the work is in generating the simulated outcome of one repetition.

Visualization¶

We can now put all the results into one table for ease of visualization.

```
results = Table().with_columns(
    'Original Door Choice', original,
    'Monty Throws Out', throw_out,
    'Remaining Door', remains
)
results
```

Original Door Choice	Monty Throws Out	Remaining Door
Goat 2	Goat 1	Car
Goat 1	Goat 2	Car
Car	Goat 1	Goat 2
Goat 2	Goat 1	Car
Car	Goat 2	Goat 1
Goat 2	Goat 1	Car
Goat 1	Goat 2	Car
Goat 1	Goat 2	Car
Goat 1	Goat 2	Car
Car	Goat 1	Goat 2

... (9990 rows omitted)

To see whether the contestant should stick with her original choice or switch, let's see how frequently the car is behind each of her two options.

```
results.group('Original Door Choice')
```

Original Door Choice	count
Car	3324
Goat 1	3340
Goat 2	3336

```
results.group('Remaining Door')
```

Remaining Door	count
Car	6676
Goat 1	1656
Goat 2	1668

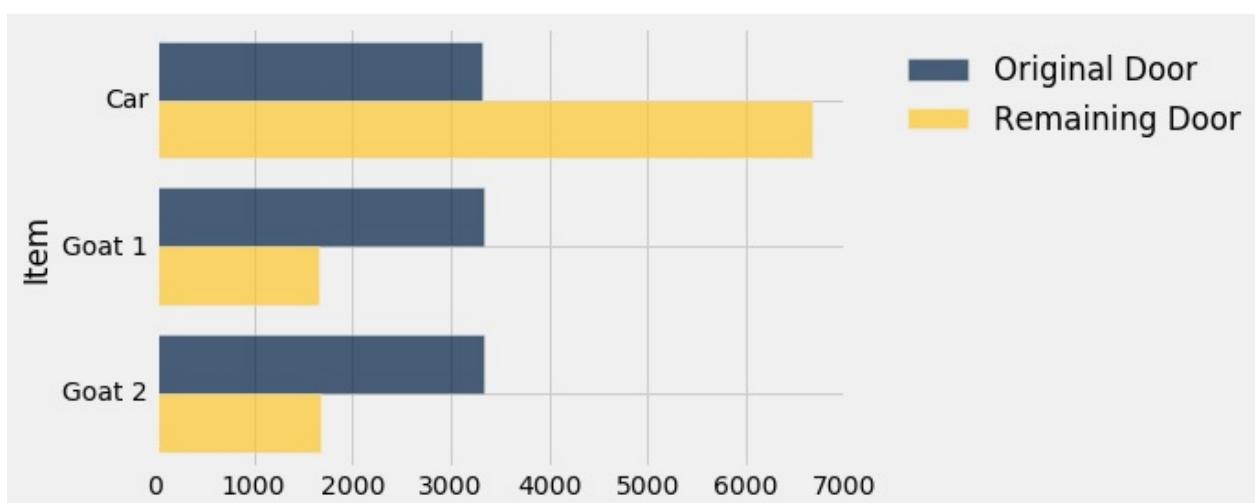
As our solution said, the car is behind the remaining door two-thirds of the time, to a pretty good approximation. The contestant is twice as likely to get the car if she switches than if she sticks with her original choice.

To see this graphically, we can join the two tables above and draw overlaid bar charts.

```
results_o = results.groupby('Original Door Choice')
results_r = results.groupby('Remaining Door')
joined = results_o.join('Original Door Choice', results_r,
'Remaining Door')
combined = joined.relabel(0, 'Item').relabel(1, 'Original
Door').relabel(2, 'Remaining Door')
combined
```

Item	Original Door	Remaining Door
Car	3324	6676
Goat 1	3340	1656
Goat 2	3336	1668

```
combined.bah(0)
```



Notice how the three blue bars are almost equal – the original choice is equally likely to be any of the three available items. But the gold bar corresponding to `car` is twice as long as the blue.

The simulation confirms that the contestant is twice as likely to win if she switches.

[Interact](#)

Finding Probabilities

Over the centuries, there has been considerable philosophical debate about what probabilities are. Some people think that probabilities are relative frequencies; others think they are long run relative frequencies; still others think that probabilities are a subjective measure of their own personal degree of uncertainty.

In this course, most probabilities will be relative frequencies, though many will have subjective interpretations. Regardless, the ways in which probabilities are calculated and combined are consistent across the different interpretations.

By convention, probabilities are numbers between 0 and 1, or, equivalently, 0% and 100%. Impossible events have probability 0. Events that are certain have probability 1.

Math is the main tool for finding probabilities exactly, though computers are useful for this purpose too. Simulation can provide excellent approximations, with high probability. In this section, we will informally develop a few simple rules that govern the calculation of probabilities. In subsequent sections we will return to simulations to approximate probabilities of complex events.

We will use the standard notation $P(\text{event})$ to denote the probability that "event" happens, and we will use the words "chance" and "probability" interchangeably.

When an Event Doesn't Happen

If the chance that event happens is 40%, then the chance that it doesn't happen is 60%. This natural calculation can be described in general as follows:

$$\begin{aligned} P(\text{an event doesn't happen}) &= 1 \\ &- P(\text{the event happens}) \end{aligned}$$

When All Outcomes are Equally Likely

If you are rolling an ordinary die, a natural assumption is that all six faces are equally likely. Then probabilities of how one roll comes out can be easily calculated as a ratio. For example, the chance that the die shows an even number is

$$\frac{\text{number of even faces}}{\text{number of all faces}} = \frac{\#\{2, 4, 6\}}{\#\{1, 2, 3, 4, 5, 6\}} = \frac{3}{6}$$

Similarly,

$$\begin{aligned} P(\text{die shows a multiple of 3}) &= \frac{\#\{3, 6\}}{\#\{1, 2, 3, 4, 5, 6\}} \\ &= \frac{2}{6} \end{aligned}$$

In general,

$$P(\text{an event happens}) = \frac{\#\{\text{outcomes that make the event happen}\}}{\#\{\text{all outcomes}\}}$$

provided all the outcomes are equally likely.

Not all random phenomena are as simple as one roll of a die. The two main rules of probability, developed below, allow mathematicians to find probabilities even in complex situations.

When Two Events Must Both Happen

Suppose you have a box that contains three tickets: one red, one blue, and one green. Suppose you draw two tickets at random without replacement; that is, you shuffle the three tickets, draw one, shuffle the remaining two, and draw another from those two. What is the chance you get the green ticket first, followed by the red one?

There are six possible pairs of colors: RB, BR, RG, GR, BG, GB (we've abbreviated the names of each color to just its first letter). All of these are equally likely by the sampling scheme, and only one of them (GR) makes the event happen. So

$$\begin{aligned} P(\text{green first, then red}) \\ = \frac{\#\{\text{GR}\}}{\#\{\text{RB, BR, RG, GR, BG, GB}\}} = \frac{1}{6} \end{aligned}$$

But there is another way of arriving at the answer, by thinking about the event in two stages. First, the green ticket has to be drawn. That has chance $1/3$, which means that the green ticket is drawn first in about $1/3$ of all repetitions of the experiment. But that doesn't complete the event. *Among the $1/3$ of repetitions when green is drawn first*, the red ticket has to be drawn next. That happens in about $1/2$ of those repetitions, and so:

$$P(\text{green first, then red}) = \frac{1}{2} \text{ of } \frac{1}{3} = \frac{1}{6}$$

This calculation is usually written "in chronological order," as follows.

$$P(\text{green first, then red}) = \frac{1}{3} \times \frac{1}{2} = \frac{1}{6}$$

The factor of $1/2$ is called "the conditional chance that the red ticket appears second, given that the green ticket appeared first."

In general, we have the **multiplication rule**:

$$\begin{aligned} P(\text{two events both happen}) &= P(\text{one event happens}) \\ &\quad \times P \\ &\quad (\text{the other event happens, given that the first one happened}) \\ &\quad) \end{aligned}$$

Thus, when there are two conditions – one event must happen, as well as another – the chance is a *fraction of a fraction*, which is smaller than either of the two component fractions. The more conditions that have to be satisfied, the less likely they are to all be satisfied.

When an Event Can Happen in Two Different Ways

Suppose instead we want the chance that one of the two tickets is green and the other red. This event doesn't specify the order in which the colors must appear. So they can appear in either order.

A good way to tackle problems like this is to *partition* the event so that it can happen in exactly one of several different ways. The natural partition of "one green and one red" is: GR, RG.

Each of GR and RG has chance $1/6$ by the calculation above. So you can calculate the chance of "one green and one red" by adding them up.

$$\begin{aligned} P(\text{one green and one red}) &= P(\text{GR}) + P(\text{RG}) = \frac{1}{6} \\ &\quad + \frac{1}{6} = \frac{2}{6} \end{aligned}$$

In general, we have the **addition rule**:

$$\begin{aligned} P(\text{an event happens}) &= P(\text{first way it can happen}) \\ &\quad + P(\text{second way it can happen}) \end{aligned}$$

provided the event happens in exactly one of the two ways.

Thus, when an event can happen in one of two different ways, the chance that it happens is a sum of chances, and hence bigger than the chance of either of the individual ways.

The multiplication rule has a natural extension to more than two events, as we will see below. So also the addition rule has a natural extension to events that can happen in one of several different ways.

We end the section with examples that use combinations of all these rules.

At Least One Success

Data scientists often work with random samples from populations. A question that sometimes arises is about the likelihood that a particular individual in the population is selected to be in the sample. To work out the chance, that individual is called a "success," and the problem is to find the chance that the sample contains a success.

To see how such chances might be calculated, we start with a simpler setting: tossing a coin two times.

If you toss a coin twice, there are four equally likely outcomes: HH, HT, TH, and TT. We have abbreviated "Heads" to H and "Tails" to T. The chance of getting at least one head in two tosses is therefore 3/4.

Another way of coming up with this answer is to work out what happens if you *don't* get at least one head: both the tosses have to land tails. So

$$\begin{aligned} P(\text{at least one head in two tosses}) &= 1 \\ - P(\text{both tails}) &= 1 - \frac{1}{4} = \frac{3}{4} \end{aligned}$$

Notice also that

$$P(\text{both tails}) = \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2} = \left(\frac{1}{2}\right)^2$$

by the multiplication rule.

These two observations allow us to find the chance of at least one head in any given number of tosses. For example,

$$\begin{aligned} P(\text{at least one head in 17 tosses}) &= 1 \\ - P(\text{all 17 are tails}) &= 1 - \left(\frac{1}{2}\right)^{17} \end{aligned}$$

And now we are in a position to find the chance that the face with six spots comes up at least once in rolls of a die.

For example,

$$\begin{aligned} P(\text{a single roll is not } 6) &= P(1) + P(2) + P(3) + P(4) \\ &\quad + P(5) = \frac{5}{6} \end{aligned}$$

Therefore,

$$\begin{aligned} P(\text{at least one } 6 \text{ in two rolls}) &= 1 \\ - P(\text{both rolls are not } 6) &= 1 - \left(\frac{5}{6}\right)^2 \end{aligned}$$

and

$$P(\text{at least one } 6 \text{ in 17 rolls}) = 1 - \left(\frac{5}{6}\right)^{17}$$

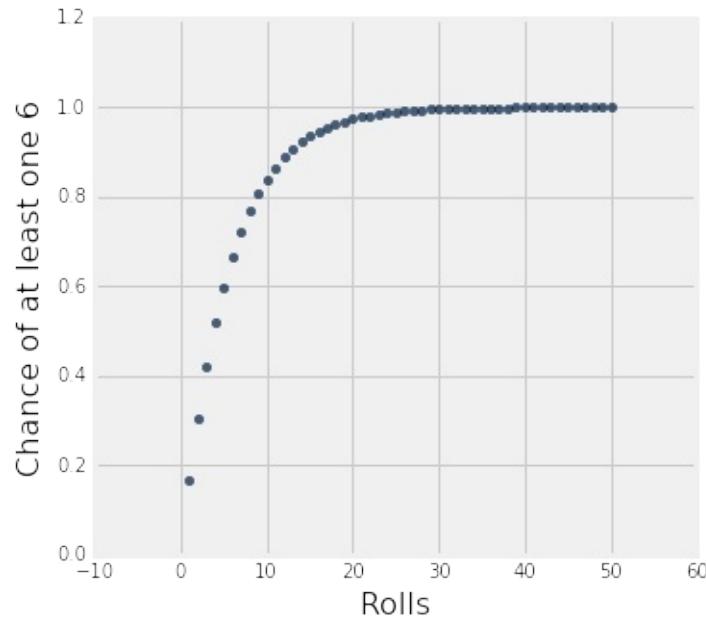
The table below shows these probabilities as the number of rolls increases from 1 to 50.

```
rolls = np.arange(1, 51, 1)
results = Table().with_columns(
    'Rolls', rolls,
    'Chance of at least one 6', 1 - (5/6)**rolls
)
results
```

Rolls	Chance of at least one 6
1	0.166667
2	0.305556
3	0.421296
4	0.517747
5	0.598122
6	0.665102
7	0.720918
8	0.767432
9	0.806193
10	0.838494
... (40 rows omitted)	

The chance that a 6 appears at least once rises rapidly as the number of rolls increases.

```
results.scatter('Rolls')
```



In 50 rolls, you are almost certain to get at least one 6.

```
results.where('Rolls', are.equal_to(50))
```

Rolls	Chance of at least one 6
50	0.99989

Calculations like these can be used to find the chance that a particular individual is selected in a random sample. The exact calculation will depend on the sampling scheme. But what we have observed above can usually be generalized: increasing the size of the random sample increases the chance that an individual is selected.

[Interact](#)

Sampling and Empirical Distributions

An important part of data science consists of making conclusions based on the data in random samples. In order to correctly interpret their results, data scientists have to first understand exactly what random samples are.

In this chapter we will take a more careful look at sampling, with special attention to the properties of large random samples.

Let's start by drawing some samples. Our examples are based on the `top_movies.csv` data set.

```
top1 = Table.read_table('top_movies.csv')
top2 = top1.with_column('Row Index', np.arange(top1.num_rows))
top = top2.move_to_start('Row Index')

top.set_format(make_array(3, 4), NumberFormatter)
```

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

... (190 rows omitted)

Sampling Rows of a Table

Each row of a data table represents an individual; in `top`, each individual is a movie. Sampling individuals can thus be achieved by sampling the rows of a table.

The contents of a row are the values of different variables measured on the same individual. So the contents of the sampled rows form samples of values of each of the variables.

Deterministic Samples

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

You have done this many times, for example by using `take`:

```
top.take(make_array(3, 18, 100))
```

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

You have also used `where`:

```
top.where('Title', are.containing('Harry Potter'))
```

Row Index	Title	Studio	Gross	Gross (Adjusted)	Year
22	Harry Potter and the Deathly Hallows Part 2	Warner Bros.	381,011,219	417,512,200	2011
43	Harry Potter and the Sorcerer's Stone	Warner Bros.	317,575,550	486,442,900	2001
54	Harry Potter and the Half-Blood Prince	Warner Bros.	301,959,197	352,098,800	2009
59	Harry Potter and the Order of the Phoenix	Warner Bros.	292,004,738	369,250,200	2007
62	Harry Potter and the Goblet of Fire	Warner Bros.	290,013,036	393,024,800	2005
69	Harry Potter and the Chamber of Secrets	Warner Bros.	261,988,482	390,768,100	2002
76	Harry Potter and the Prisoner of Azkaban	Warner Bros.	249,541,069	349,598,600	2004

While these are samples, they are not random samples. They don't involve chance.

Probability Samples

For describing random samples, some terminology will be helpful.

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.

A Random Sampling Scheme

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.

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.choice(np.arange(10))  
top.take(np.arange(start, top.num_rows, 10))
```

RowIndex	Title	Studio	Gross	Gross (Adjusted)	Year
4	Marvel's The Avengers	Buena Vista (Disney)	623,357,910	668,866,600	2012
14	The Lion King	Buena Vista (Disney)	422,783,777	775,573,900	1994
24	Star Wars: Episode III - Revenge of the Sith	Fox	380,270,577	516,123,900	2005
34	The Hunger Games: Mockingjay - Part 1	Lionsgate	337,135,885	354,324,000	2014
44	Indiana Jones and the Kingdom of the Crystal Skull	Paramount	317,101,119	384,231,200	2008
54	Harry Potter and the Half-Blood Prince	Warner Bros.	301,959,197	352,098,800	2009
64	Home Alone	Fox	285,761,243	589,287,500	1990
74	Night at the Museum	Fox	250,863,268	322,261,900	2006
84	Beverly Hills Cop	Paramount	234,760,478	584,205,200	1984
94	Saving Private Ryan	Dreamworks	216,540,909	397,999,500	1998

... (10 rows omitted)

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

This systematic sample is a probability sample. In this scheme, all rows have chance $1/10$ of being chosen. For example, Row 23 is chosen if and only if Row 3 is chosen, and the chance of that is $1/10$.

But not all subsets have the same chance of being chosen. 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 $1/10$. Other subsets, like the subset containing the first 11 rows of the table, are selected with chance 0.

Random Samples Drawn With or Without Replacement

In this course, we will mostly deal with the two most straightforward methods of sampling.

The first is random sampling with replacement, which (as we have seen earlier) is the default behavior of `np.random.choice` when it samples from an array.

The other, called a "simple random sample", is a sample drawn at random *without replacement*. Sampled individuals are not replaced in the population before the next individual is drawn. This is the kind of sampling that happens when you deal a hand from a deck of cards, for example.

In this chapter, we will use simulation to study the behavior of large samples drawn at random with or without replacement.

Drawing a random sample requires care and precision. It is not haphazard, even though that is a colloquial meaning of the word "random". If you stand at a street corner and take as your sample the first ten people who pass by, you might think you're sampling at random because you didn't choose who walked by. But it's not a random sample – it's a *sample of convenience*. You didn't know ahead of time the probability of each person entering the sample; perhaps you hadn't even specified exactly who was in the population.

[Interact](#)

Empirical Distributions

In data science, the word "empirical" means "observed". Empirical distributions are distributions of observed data, such as data in random samples.

In this section we will generate data and see what the empirical distribution looks like.

Our setting is a simple experiment: rolling a die multiple times and keeping track of which face appears. The table `die` 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.

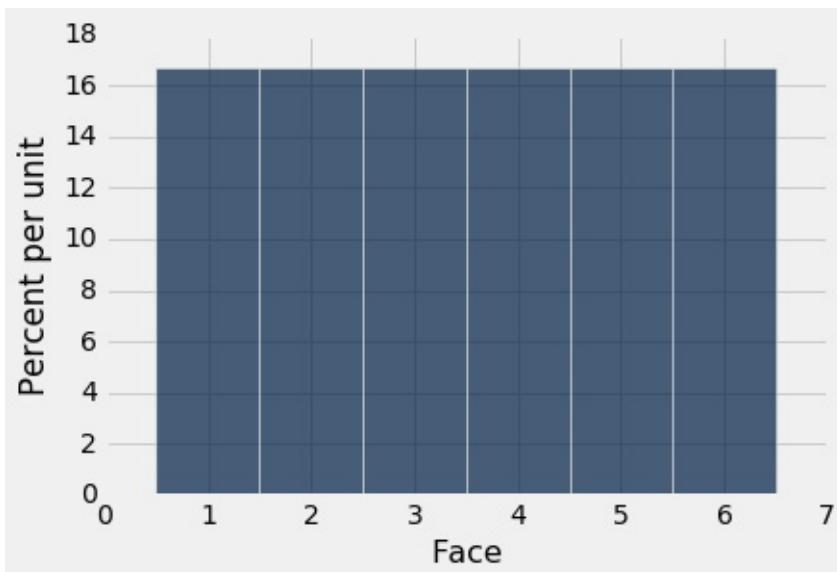
```
die = Table().with_column('Face', np.arange(1, 7, 1))
die
```

Face
1
2
3
4
5
6

A Probability Distribution

The histogram below helps us visualize the fact that every face appears with probability 1/6. We say that the histogram shows the *distribution* of probabilities over all the possible faces. Since all the bars represent the same percent chance, the distribution is called *uniform on the integers 1 through 6*.

```
die_bins = np.arange(0.5, 6.6, 1)
die.hist(bins = die_bins)
```



Variables whose successive values are separated by the same fixed amount, such as the values on rolls of a die (successive values separated by 1), fall into a class of variables that are called *discrete*. The histogram above is called a *discrete histogram*. Its bins are specified by the array `die_bins` and ensure that each bar is centered over the corresponding integer value.

It is important to remember that the die can't show 1.3 spots, or 5.2 spots – it always shows an integer number of spots. But our visualization spreads the probability of each value over the area of a bar. While this might seem a bit arbitrary at this stage of the course, it will become important later when we overlay smooth curves over discrete histograms.

Before going further, let's make sure that the numbers on the axes make sense. The probability of each face is $1/6$, which is 16.67% when rounded to two decimal places. The width of each bin is 1 unit. So the height of each bar is 16.67% per unit. This agrees with the horizontal and vertical scales of the graph.

Empirical Distributions

The distribution above consists of the theoretical probability of each face. It is not based on data. It can be studied and understood without any dice being rolled.

Empirical distributions, on the other hand, are distributions of observed data. They can be visualized by *empirical histograms*.

Let us get some data by simulating rolls of a die. This can be done by sampling at random with replacement from the integers 1 through 6. We have used `np.random.choice` for such simulations before. But now we will introduce a Table method for doing this. This will make it possible for us to use our familiar Table methods for visualization.

The Table method is called `sample`. It draws at random with replacement from the rows of a table. Its argument is the sample size, and it returns a table consisting of the rows that were selected. An optional argument `with_replacement=False` specifies that the sample should be drawn without replacement, but that does not apply to rolling a die.

Here are the results of 10 rolls of a die.

```
die.sample(10)
```

Face
1
5
4
3
1
6
6
3
4
2

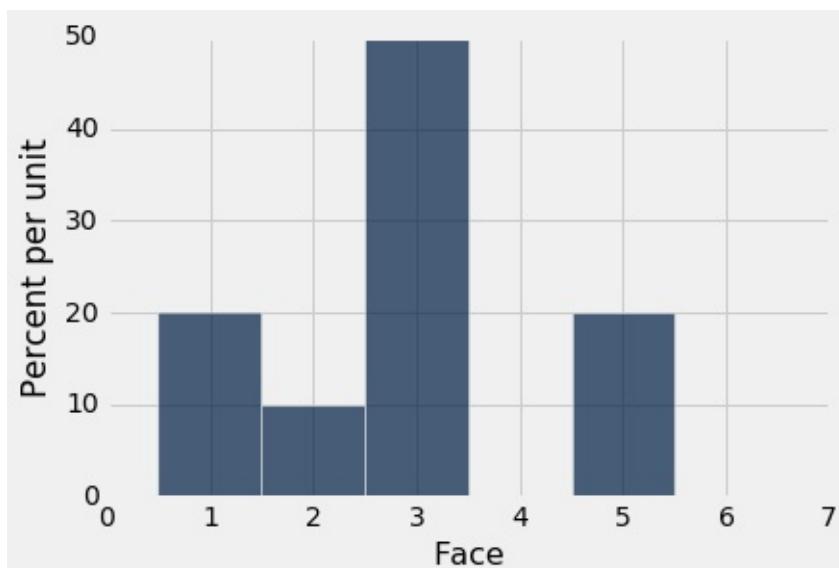
We can use the same method to simulate as many rolls as we like, and then draw empirical histograms of the results. Because we are going to do this repeatedly, we define a function `empirical_hist_die` that takes as its argument the sample size; the function rolls the die as many times as its argument and then draws a histogram.

```
def empirical_hist_die(n):
    die.sample(n).hist(bins = die_bins)
```

Empirical Histograms

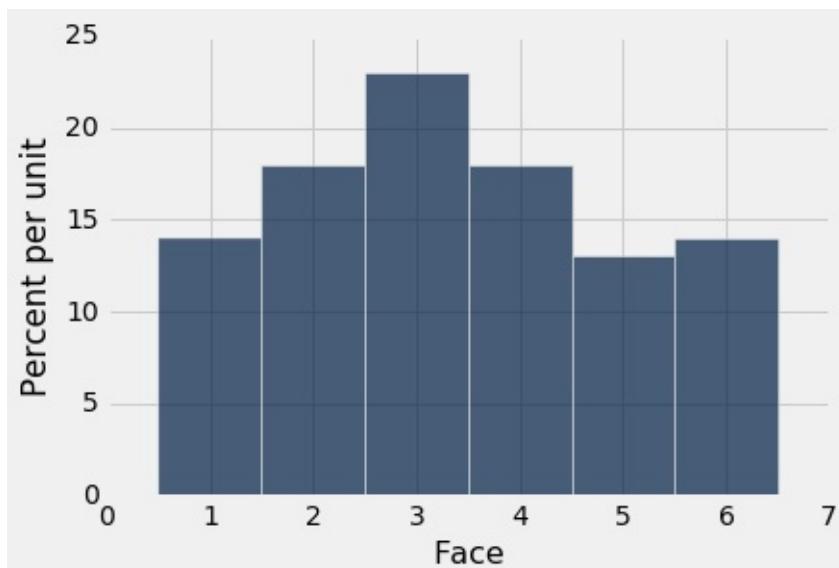
Here is an empirical histogram of 10 rolls. It doesn't look very much like the probability histogram above. Run the cell a few times to see how it varies.

```
empirical_hist_die(10)
```

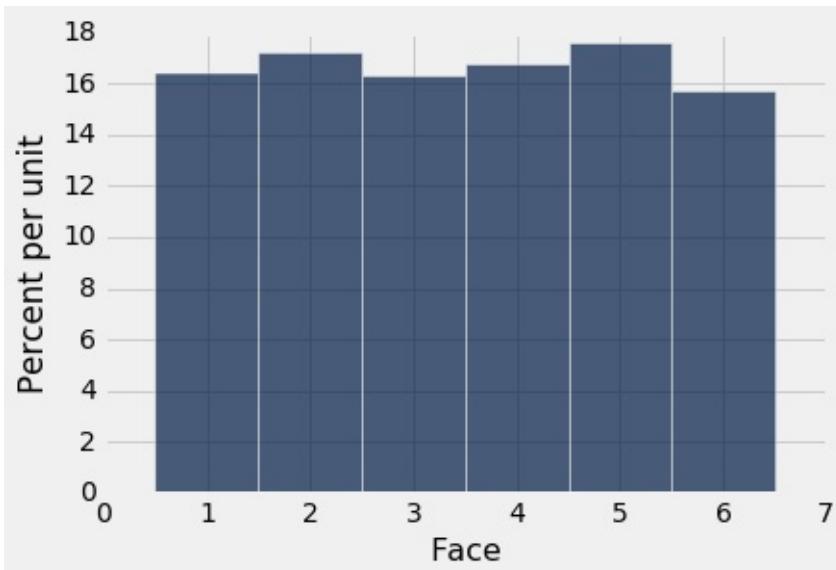


When the sample size increases, the empirical histogram begins to look more like the histogram of theoretical probabilities.

```
empirical_hist_die(100)
```



```
empirical_hist_die(1000)
```



As we increase the number of rolls in the simulation, the area of each bar gets closer 16.67%, which is the area of each bar in the probability histogram.

What we have observed in 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.

[Interact](#)

Sampling from a Population

The law of averages also holds when the random sample is drawn from individuals in a large population.

As an example, we will study a population of flight delay times. The table `united` contains data for United Airlines domestic flights departing from San Francisco in the summer of 2015. The data are made publicly available by the [Bureau of Transportation Statistics](#) in the United States Department of Transportation.

There are 13,825 rows, each corresponding to a flight. The columns are the date of the flight, the flight number, the destination airport code, and the departure delay time in minutes. Some delay times are negative; those flights left early.

```
united = Table.read_table('united_summer2015.csv')
united
```

Date	Flight Number	Destination	Delay
6/1/15	73	HNL	257
6/1/15	217	EWR	28
6/1/15	237	STL	-3
6/1/15	250	SAN	0
6/1/15	267	PHL	64
6/1/15	273	SEA	-6
6/1/15	278	SEA	-8
6/1/15	292	EWR	12
6/1/15	300	HNL	20
6/1/15	317	IND	-10
... (13815 rows omitted)			

One flight departed 16 minutes early, and one was 580 minutes late. The other delay times were almost all between -10 minutes and 200 minutes, as the histogram below shows.

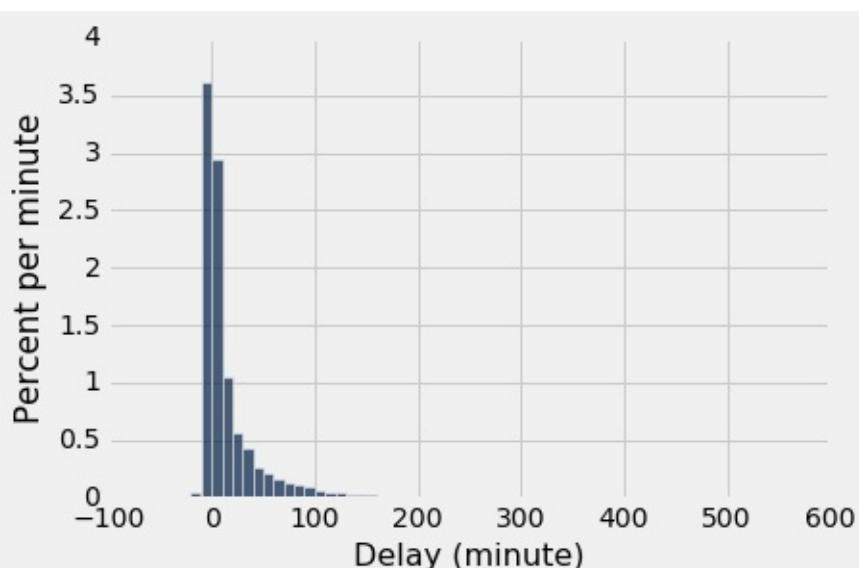
```
united.column('Delay').min()
```

```
- 16
```

```
united.column('Delay').max()
```

```
580
```

```
delay_bins = np.append(np.arange(-20, 301, 10), 600)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
```

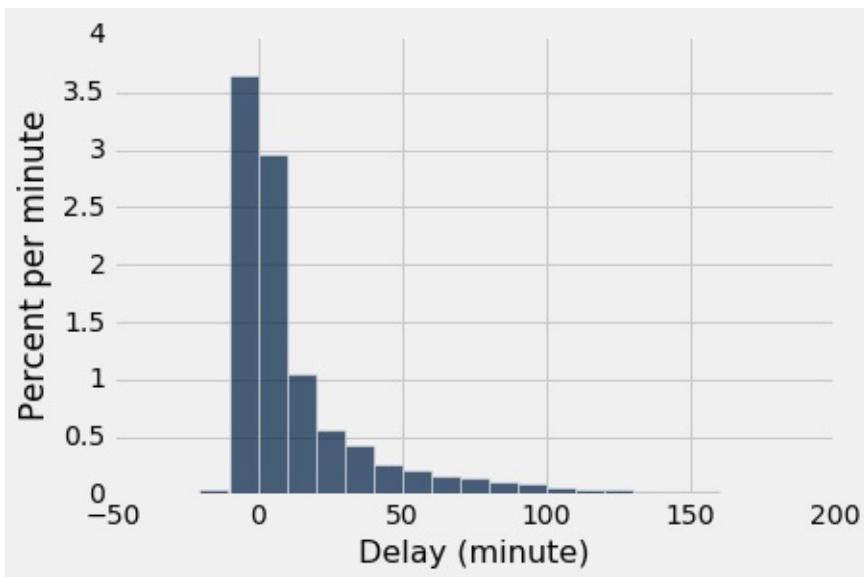


For the purposes of this section, it is enough to zoom in on the bulk of the data and ignore the 0.8% of flights that had delays of more than 200 minutes. This restriction is just for visual convenience; the table still retains all the data.

```
united.where('Delay', are.above(200)).num_rows/united.num_rows
```

```
0.008390596745027125
```

```
delay_bins = np.arange(-20, 201, 10)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
```



The height of the [0, 10) bar is just under 3% per minute, which means that just under 30% of the flights had delays between 0 and 10 minutes. That is confirmed by counting rows:

```
united.where('Delay', are.between(0, 10)).num_rows/united.num_rows
```

```
0.2935985533453888
```

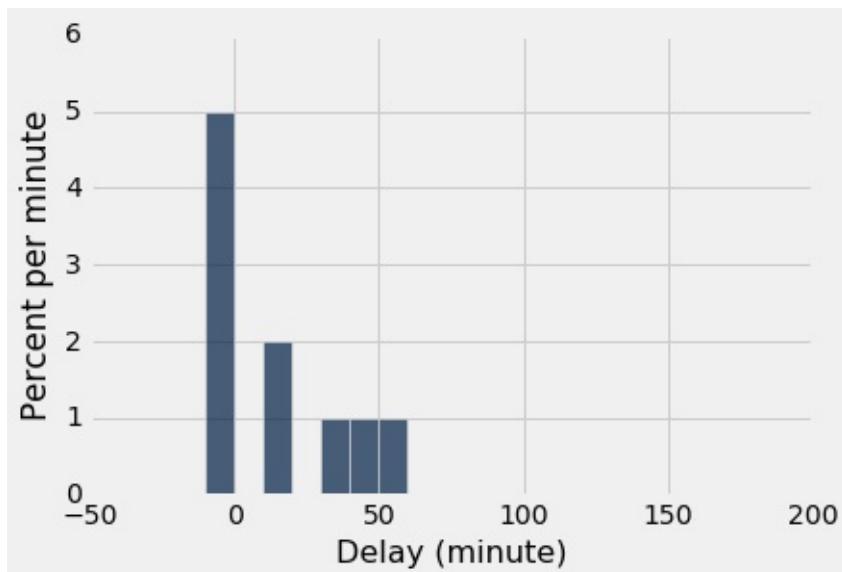
Empirical Distribution of the Sample

Let us now think of the 13,825 flights as a population, and draw random samples from it with replacement. It is helpful to package our analysis code into a function. The function `empirical_hist_delay` takes the sample size as its argument and draws an empirical histogram of the results.

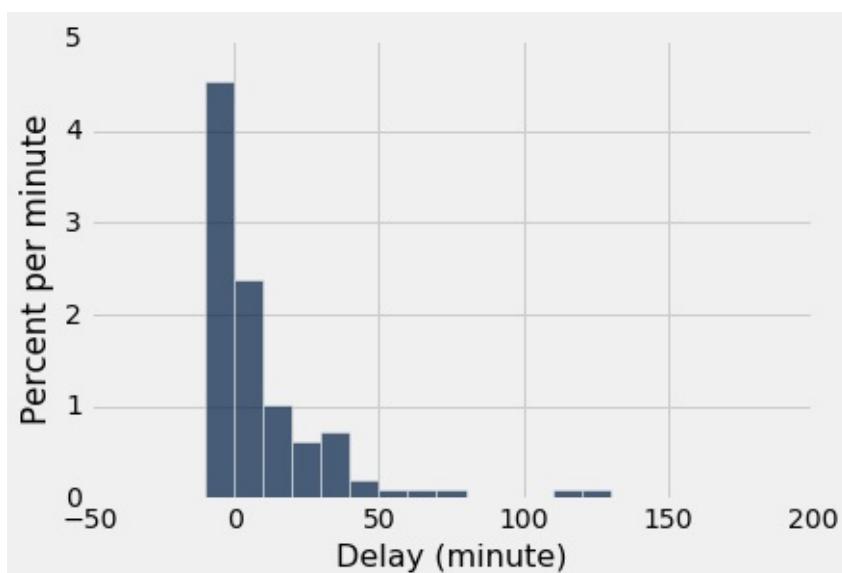
```
def empirical_hist_delay(n):
    united.sample(n).select('Delay').hist(bins = delay_bins,
    unit = 'minute')
```

As we saw with the dice, as the sample size increases, the empirical histogram of the sample more closely resembles the histogram of the population. Compare these histograms to the population histogram above.

```
empirical_hist_delay(10)
```

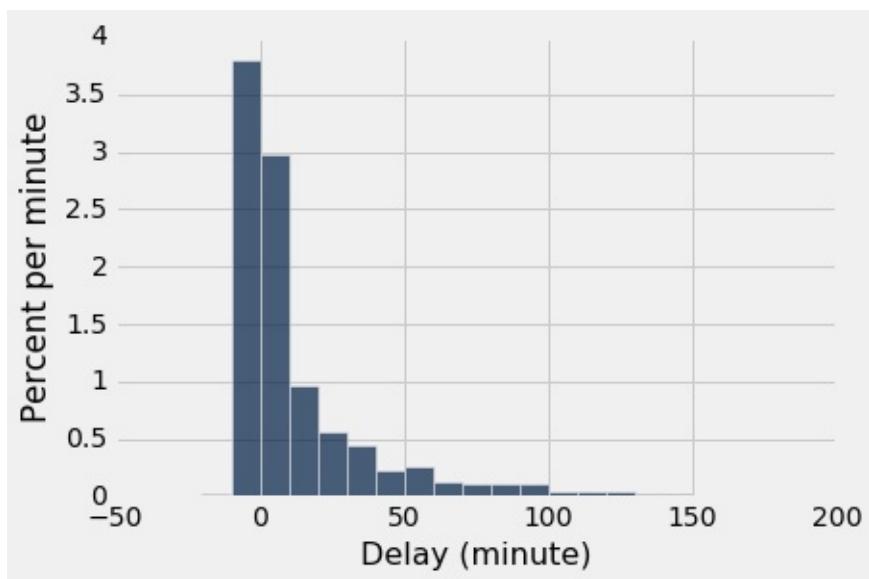


```
empirical_hist_delay(100)
```



The most consistently visible discrepancies are among the values that are rare in the population. In our example, those values are in the the right hand tail of the distribution. But as the sample size increases, even those values begin to appear in the sample in roughly the correct proportions.

```
empirical_hist_delay(1000)
```



Convergence of the Empirical Histogram of the Sample

What we have observed in this section can be summarized as follows:

For a large random sample, the empirical histogram of the sample resembles the histogram of the population, with high probability.

This justifies the use of large random samples in statistical inference. The idea is that since a large random sample is likely to resemble the population from which it is drawn, quantities computed from the values in the sample are likely to be close to the corresponding quantities in the population.

[Interact](#)

Empirical Distribution of a Statistic

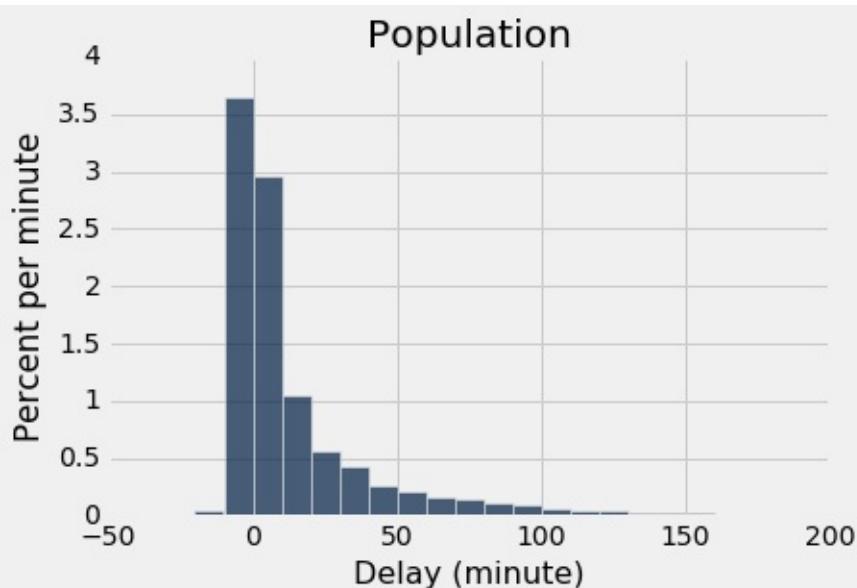
The Law of Averages implies that with high probability, the empirical distribution of a large random sample will resemble the distribution of the population from which the sample was drawn.

The resemblance is visible in two histograms: the empirical histogram of a large random sample is likely to resemble the histogram of the population.

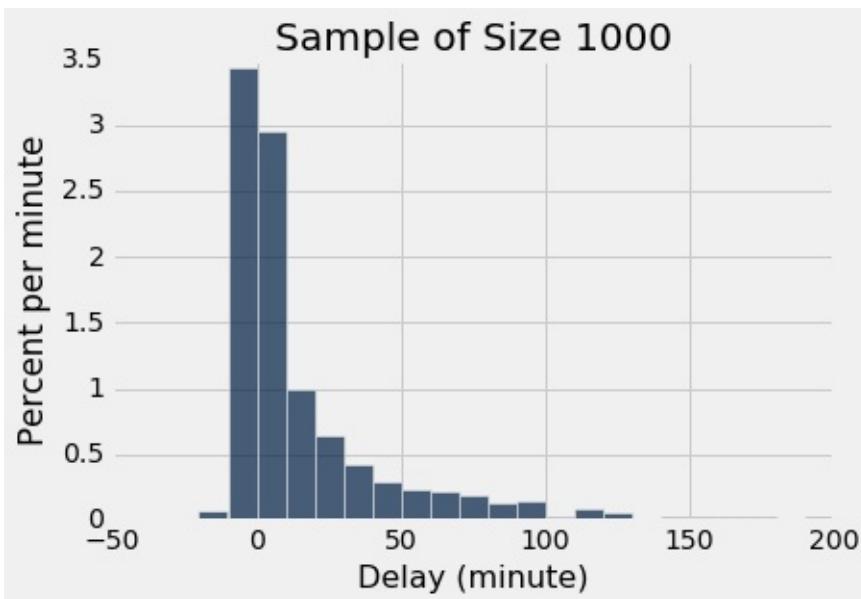
As a reminder, here is the histogram of the delays of all the flights in `united`, and an empirical histogram of the delays of a random sample of 1,000 of these flights.

```
united = Table.read_table('united_summer2015.csv')
```

```
delay_bins = np.arange(-20, 201, 10)
united.select('Delay').hist(bins = delay_bins, unit = 'minute')
plots.title('Population');
```



```
sample_1000 = united.sample(1000)
sample_1000.select('Delay').hist(bins = delay_bins, unit =
'minute')
plots.title('Sample of Size 1000');
```



The two histograms clearly resemble each other, though they are not identical.

Parameter

Frequently, we are interested in numerical quantities associated with a population.

- In a population of voters, what percent will vote for Candidate A?
- In a population of Facebook users, what is the largest number of Facebook friends that the users have?
- In a population of United flights, what is the median departure delay?

Numerical quantities associated with a population are called *parameters*. For the population of flights in `united`, we know the value of the parameter "median delay":

```
np.median(united.column('Delay'))
```

```
2.0
```

The `NumPy` function `median` returns the median (half-way point) of an array. Among all the flights in `united`, the median delay was 2 minutes. That is, about 50% of flights in the population had delays of 2 or fewer minutes:

```
united.where('Delay',
are.below_or_equal_to(2)).num_rows/united.num_rows
```

```
0.5018444846292948
```

Half of all flights left no more than 2 minutes after their scheduled departure time. That's a very short delay!

Note. The percent isn't exactly 50 because of "ties," that is, flights that had delays of exactly 2 minutes. There were 480 such flights. Ties are quite common in data sets, and we will not worry about them in this course.

```
united.where('Delay', are.equal_to(2)).num_rows
```

```
480
```

Statistic¶

In many situations, we will be interested in figuring out the value of an unknown parameter. For this, we will rely on data from a large random sample from the population.

A *statistic* (note the singular!) is any number computed using the data in a sample. The sample median, therefore, is a statistic.

Remember that `sample_1000` contains a random sample of 1000 flights from `united`. The observed value of the sample median is:

```
np.median(sample_1000.column('Delay'))
```

```
3.0
```

Our sample – one set of 1,000 flights – gave us one observed value of the statistic. This raises an important problem of inference:

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.

```
np.median(united.sample(1000).column('Delay'))
```

```
2.0
```

Run the cell above a few times to see how the answer varies. Often it is equal to 2, the same value as the population parameter. But sometimes it is different.

Just how different could the statistic have been? One way to answer this is to simulate the statistic many times and note the values. A histogram of those values will tell us about the distribution of the statistic.

Let's recall the main steps in a simulation.

Simulating a Statistic

We will simulate the sample median using the steps we set up in an earlier chapter when we started studying simulation. You can replace the sample size of 1000 by any other sample size, and the sample median by any other statistic.

Step 1: Decide which statistic to simulate. We have already decided that: we are going to simulate the median of a random sample of size 1000 drawn from the population of flight delays.

Step 2: Write the code to generate one value of the statistic. Draw a random sample of size 1000 and compute the median of the sample. We did this in the code cell above. Here it is again for reference.

```
np.median(united.sample(1000).column('Delay'))
```

```
3.0
```

Step 3: Decide how many simulated values to generate. Let's do 5,000 repetitions.

Step 4: Write the code to generate an array of simulated values. As in all simulations, we start by creating an empty array in which we will collect our results. We will then set up a `for` loop for generating all the simulated values. The body of the loop will consist of generating one simulated value of the sample median, and appending it to our collection array.

The simulation takes a noticeable amount of time to run. That is because it is performing 5000 repetitions of the process of drawing a sample of size 1000 and computing its median. That's a lot of sampling and repeating!

```
medians = make_array()

for i in np.arange(5000):
    new_median = np.median(united.sample(1000).column('Delay'))
    medians = np.append(medians, new_median)
```

The simulation is done. All 5,000 simulated sample medians have been collected in the array `medians`. Now it's time to visualize the results.

Visualization

Here are the simulated values displayed in the table `simulated_medians`.

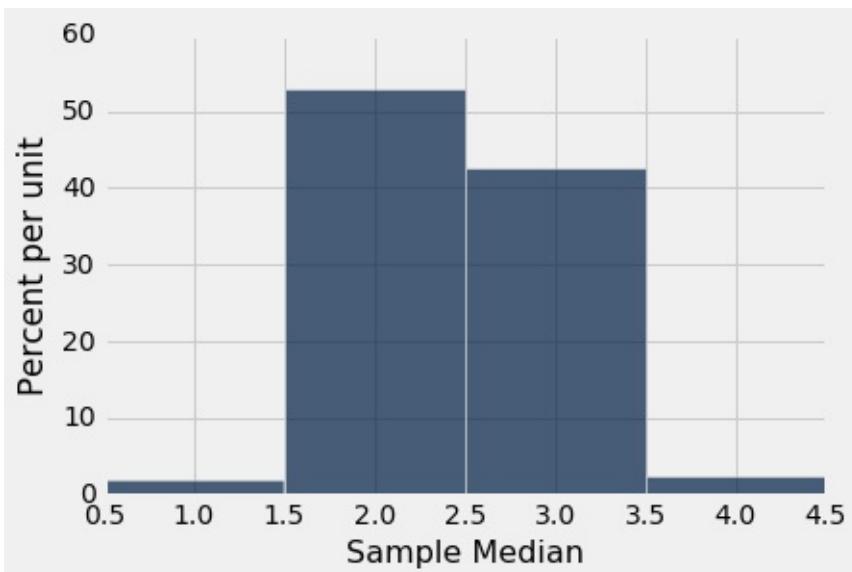
```
simulated_medians = Table().with_column('Sample Median',
                                         medians)
simulated_medians
```

Sample Median
2
3
3
3
3
3
2
3
3
3

... (4990 rows omitted)

We can also visualize the simulated data using a histogram. The histogram is called an *empirical histogram of the statistic*. It displays the *empirical distribution* of the statistic. Remember that *empirical* means *observed*.

```
simulated_medians.hist(bins=np.arange(0.5, 5, 1))
```



You can see that the sample median is very likely to be close to 2, which was the value of the population median. Since samples of 1000 flight delays are likely to resemble the population of delays, it is not surprising that the median delays of those samples should be close to the median delay in the population.

This is an example of how a statistic can provide a good estimate of a parameter.

The Power of Simulation

If we could generate all possible random samples of size 1000, we would know all possible values of the statistic (the sample median), as well as the probabilities of all those values. We could visualize all the values and probabilities in the probability histogram of the statistic.

But in many situations (including this one), the number of all possible samples is large enough to exceed the capacity of the computer, and purely mathematical calculations of the probabilities can be intractably difficult.

This is where empirical histograms come in.

We know that by the Law of Averages, the empirical histogram of the statistic is likely to resemble the probability histogram of the statistic, if the sample size is large and if you repeat the sampling process numerous times.

This means that simulating random processes repeatedly is a way of approximating probability distributions *without figuring out the probabilities mathematically or generating all possible random samples*. 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 in other ways.

[Interact](#)

Testing Hypotheses

Data scientists are often faced with yes-no questions about the world. You have seen some examples of such questions in this course:

- Is chocolate good for you?
- Did water from the Broad Street pump cause cholera?
- Have the demographics in California changed over the past decade?

Whether we answer questions like these depends on the data we have. Census data about California can settle questions about demographics with hardly any uncertainty about the answer. We know that Broad Street pump water was contaminated by waste from cholera victims, so we can make a pretty good guess about whether it caused cholera.

Whether chocolate or any other treatment is good for you will almost certainly have to be decided by medical experts, but an initial step consists of using data science to analyze data from studies and randomized experiments.

In this chapter, we will try to answer such yes-no questions, basing our conclusions on random samples and empirical distributions.

Interact

Assessing Models¶

In data science, a "model" is a set of assumptions about data. Often, models include assumptions about chance processes used to generate data.

Sometimes, data scientists have to decide whether or not their models are good. In this section we will discuss two examples of making such decisions. In later sections we will use the methods developed here as the building blocks of a general framework for testing hypotheses.

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 Negroes."

Jury panels are supposed to be selected at random from the eligible population. Because 26% of the eligible population was black, 8 black men on a panel of 100 might seem low.

A Model¶

But one view of the data – a model, in other words – is that the panel was selected at random and ended up with a small number of black men just due to chance. This model is consistent with what the Supreme Court wrote in its ruling.

The model specifies the details of a chance process. It says the data are like a random sample from a population in which 26% of the people are black. We are in a good position to assess this model, because:

- We can simulate data based on the model. That is, we can simulate drawing at random from a population of whom 26% are black.
- Our simulation will show what a panel *would* look like *if* it were selected at random.
- We can then compare the results of the simulation with the composition of Robert Swain's panel.

- If the results of our simulation are not consistent with the composition of Swain's panel, that will be evidence against the model of random selection.

Let's go through the process in detail.

The Statistic

First, we have to choose a statistic to simulate. The statistic has to be able to help us decide between the model and alternative views about the data. The model says the panel was drawn at random. The alternative viewpoint, suggested by Robert Swain's appeal, is that the panel was not drawn at random because it contained too few black men. A natural statistic, then, is the number of black men in our simulated sample of 100 men representing the panel. Small values of the statistic will favor the alternative viewpoint.

Predicting the Statistic Under the Model

If the model were true, how big would the statistic typically be? To answer that, we have to start by working out the details of the simulation.

Generating One Value of the Statistic

First let's figure out how to simulate one value of the statistic. For this, we have to sample 100 times at random from the population of eligible jurors and count the number of black men we get.

One way is to set up a table representing the eligible population and use `sample` as we did in the previous chapter. But there is also a quicker way, using a `datascience` function tailored for sampling at random from categorical distributions. We will use it several times in this chapter.

The `sample_proportions` function in the `datascience` library takes two arguments:

- the sample size
- the distribution of the categories in the population, as a list or array of proportions that add up to 1

It returns an array containing the distribution of the categories in a random sample of the given size taken from the population. That's an array consisting of the sample proportions in all the different categories.

To see how to use this, remember that according to our model, the panel is selected at random from a population of men among whom 26% were black and 74% were not. Thus the distribution of the two categories can be represented as the list `[0.26, 0.74]`, which we

have assigned to the name `eligible_population`. Now let's sample at random 100 times from this distribution, and see what proportions of the two categories we get in our sample.

```
eligible_population = [0.26, 0.74]
sample_proportions(100, eligible_population)
```

```
array([ 0.3,  0.7])
```

That was easy! The proportion of black men in the random sample is `item(0)` of the output array.

Because there are 100 men in the sample, the number of men in each category is 100 times the proportion. So we can just as easily simulate counts instead of proportions, and access the count of black men only.

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

```
# count of black men in a simulated panel
(100 * sample_proportions(100, eligible_population)).item(0)
```

```
25.0
```

Running the Simulation

To get a sense of the variability without running the cell over and over, let's generate 10,000 simulated values of the count. The code follows the same steps that we have used in every simulation.

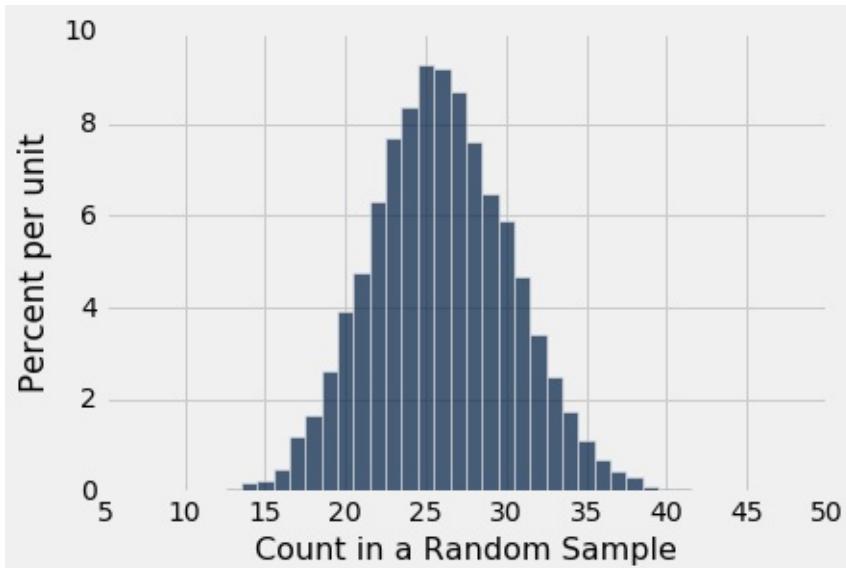
```
counts = make_array()

repetitions = 10000
for i in np.arange(repetitions):
    simulated_count = (100 * sample_proportions(100,
eligible_population)).item(0)
    counts = np.append(counts, simulated_count)
```

The Prediction

To interpret the results of our simulation, we start as usual by visualizing the results by an empirical histogram.

```
Table().with_column(
    'Count in a Random Sample', counts
).hist(bins = np.arange(5.5, 46.6, 1))
```



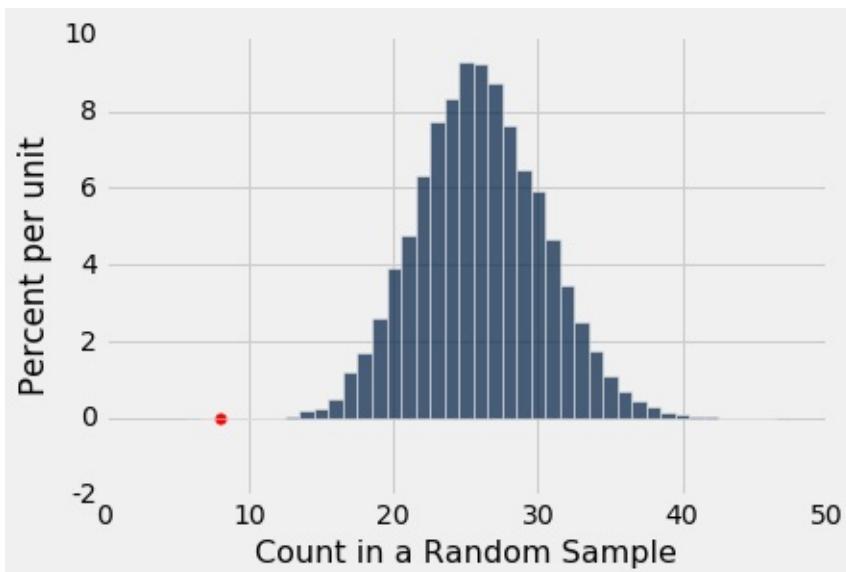
The histogram tells us what the model of random selection predicts about our statistic, the count of black men in the sample.

To generate each simulated count, we drew at 100 times at random from a population in which 26% were black. So, as you would expect, most of the simulated counts are around 26. They are not exactly 26 – there is some variation. The counts range between about 10 and 45.

Comparing the Prediction and the Data

Though the simulated counts are quite varied, very few of them came out to be eight or less. The value eight is far out in the left hand tail of the histogram. It's the red dot on the horizontal axis of the histogram.

```
Table().with_column(
    'Count in a Random Sample', counts
).hist(bins = np.arange(5.5, 46.6, 1))
plots.scatter(8, 0, color='red', s=30);
```



Thus the simulation shows that if we select a panel of 100 jurors at random from the eligible population, we are very unlikely to get counts of black men as low as the eight that were in Swain's jury panel. This is evidence that the model of random selection of the jurors in the panel is not consistent with the data from the panel.

When the data and a model are inconsistent, the model is hard to justify. After all, the data are real. The model is just a set of assumptions. When assumptions are at odds with reality, we have to question those assumptions.

While it is *possible* that a panel like Robert Swain's could have been generated by chance, our simulation demonstrates that it is very unlikely. Thus our assessment is that the model of random draws is not supported by the evidence. Swain's jury panel does not look like the result of random sampling from the population of eligible jurors.

This method of assessing models is very general. Here is an example in which we use it to assess a model in a completely different setting.

Mendel's Pea Flowers

[Gregor Mendel](#) (1822-1884) was an Austrian monk who is widely recognized as the founder of the modern field of genetics. Mendel performed careful and large-scale experiments on plants to come up with fundamental laws of genetics.

Many of his experiments were on varieties of pea plants. He formulated sets of assumptions about each variety; these were his models. He then tested the validity of his models by growing the plants and gathering data.

Let's analyze the data from one such experiment to see if Mendel's model was good.

In a particular variety, each plant has either purple flowers or white. The color in each plant is unaffected by the colors in other plants. Mendel hypothesized that the plants should bear purple or white flowers at random, in the ratio 3:1.

Mendel's Model

For every plant, there is a 75% chance that it will have purple flowers, and a 25% chance that the flowers will be white, regardless of the colors in all the other plants.

Approach to Assessment

To go about assessing Mendel's model, we can simulate plants under the assumptions of the model and see what it predicts. Then we will be able to compare the predictions with the data that Mendel recorded.

The Statistic

Our goal is to see whether or not Mendel's model is good. We need to simulate a statistic that will help us make this decision.

If the model is good, the percent of purple-flowering plants in the sample should be close to 75%. If the model is not good, the percent purple-flowering will be away from 75%. It may be higher, or lower; the direction doesn't matter.

The key for us is the *distance* between 75% and the percent of purple-flowering plants in the sample. Big distances are evidence that the model isn't good.

Our statistic, therefore, is the **distance between the sample percent and 75%**:

$$|\text{sample percent of purple-flowering plants} - 75|$$

Predicting the Statistic Under the Model

To see how big the distance would be if Mendel's model were true, we can use `sample_proportions` to simulate the distance under the assumptions of the model.

First, we have to figure out how many times to sample. To do this, remember that we are going to compare our simulation with Mendel's plants. So we should simulate the same number of plants that he had.

Mendel grew a lot of plants. There were 929 plants of the variety corresponding to this model. So we have to sample 929 times.

Generating One Value of the Statistic

The steps in the calculation:

- Sample 929 times at random from the distribution specified by the model and find the sample proportion in the purple-flowering category.
- Multiply the proportion by 100 to get a percent.
- Subtract 75 and take the absolute value of the difference.

That's the statistic: the distance between the sample percent and 75.

```
model_proportions = [0.75, 0.25]
abs(100 * sample_proportions(929, model_proportions).item(0) - 75)
```

```
2.610333692142092
```

That's one simulated value of the distance between the sample percent of purple-flowering plants and 75% as predicted by Mendel's model.

Running the Simulation

To get a sense of how variable the distance could be, we have to simulate it many more times. We will generate 10,000 values of the distance.

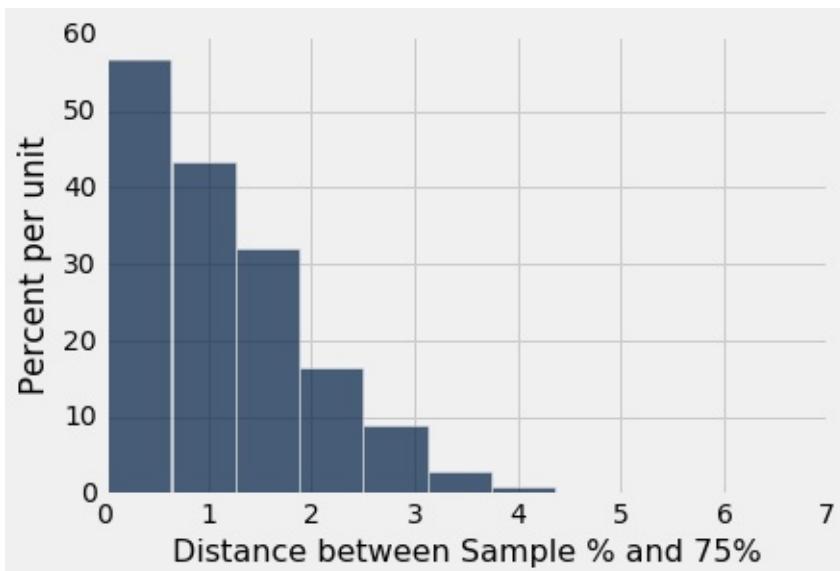
```
distances = make_array()

repetitions = 10000
for i in np.arange(repetitions):
    one_distance = abs(100 * sample_proportions(929,
model_proportions).item(0) - 75)
    distances = np.append(distances, one_distance)
```

The Prediction

The empirical histogram of the simulated values shows the distribution of the distance as predicted by the model.

```
Table().with_column(
    'Distance between Sample % and 75%', distances
).hist()
```



Look on the horizontal axis to see the typical values of the distance, as predicted by the model. They are rather small. For example, a high proportion of the distances are in the range 0 to 1, meaning that for a high proportion of the samples, the percent of purple-flowering plants is within 1% of 75%, that is, the sample percent is in the range 74% to 76%.

Comparing the Prediction and the Data

To assess the model, we have to compare this prediction with the data. Mendel recorded the number of purple and white flowering plants. Among the 929 plants that he grew, 705 were purple flowering. That's just about 75.89%.

705 / 929

0.7588805166846071

So the observed value of our statistic – the distance between Mendel's sample percent and 75 – is about 0.89:

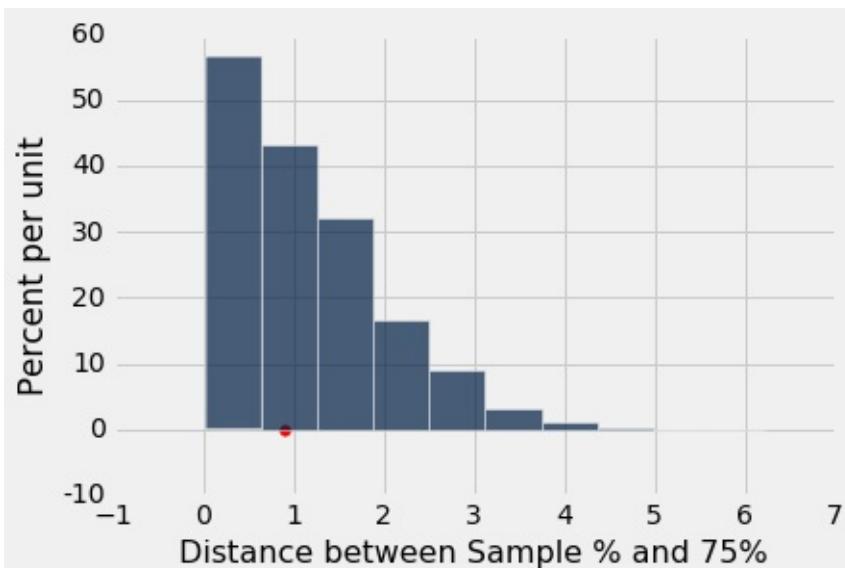
```
observed_statistic = abs (100 * (705 / 929) - 75)
observed_statistic
```

```
0.8880516684607045
```

Just by eye, locate roughly where 0.89 is on the horizontal axis of the histogram. You will see that it is clearly in the heart of the distribution predicted by Mendel's model.

The cell below redraws the histogram with the observed value plotted on the horizontal axis.

```
Table().with_column(  
    'Distance between Sample % and 75%', distances  
).hist()  
plots.scatter(observed_statistic, 0, color='red', s=30);
```



The observed statistic is like a typical distance predicted by the model. By this measure, the data are consistent with the histogram that we generated under the assumptions of Mendel's model. This is evidence in favor of the model.

[Interact](#)

Multiple Categories

We have developed a way of assessing models about chance processes that generate data in two categories. The method extends to models involving data in multiple categories. The process of assessment is the same as before, the only difference being that we have to come up with a new statistic to simulate.

Let's do this in an example that addresses the same kind of question that was raised in the case of Robert Swain's jury panel. This time, the data are more recent.

Jury Selection in Alameda County

In 2010, the American Civil Liberties Union (ACLU) of Northern California presented a [report](#) on jury selection in Alameda County, California. The report concluded that certain ethnic groups are underrepresented among jury panelists in Alameda County, and suggested some reforms of the process by which eligible jurors are assigned to panels. In this section, we will perform our own analysis of the data and examine some questions that arise as a result.

Some details about jury panels and juries will be helpful in interpreting the results of our analysis.

Jury Panels

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.

Composition of Panels in Alameda County

The focus of the study by the ACLU of Northern California was the ethnic composition of jury panels in Alameda County. The ACLU compiled data on the ethnic 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 1,453. 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 ethnicity, the first value is the proportion of all eligible juror candidates of that ethnicity. The second value is the proportion of people of that ethnicity among those who appeared for the process of selection into the jury.

```

jury = Table().with_columns(
    'Ethnicity', make_array('Asian', 'Black', 'Latino', 'White',
    'Other'),
    'Eligible', make_array(0.15, 0.18, 0.12, 0.54, 0.01),
    'Panels', make_array(0.26, 0.08, 0.08, 0.54, 0.04)
)

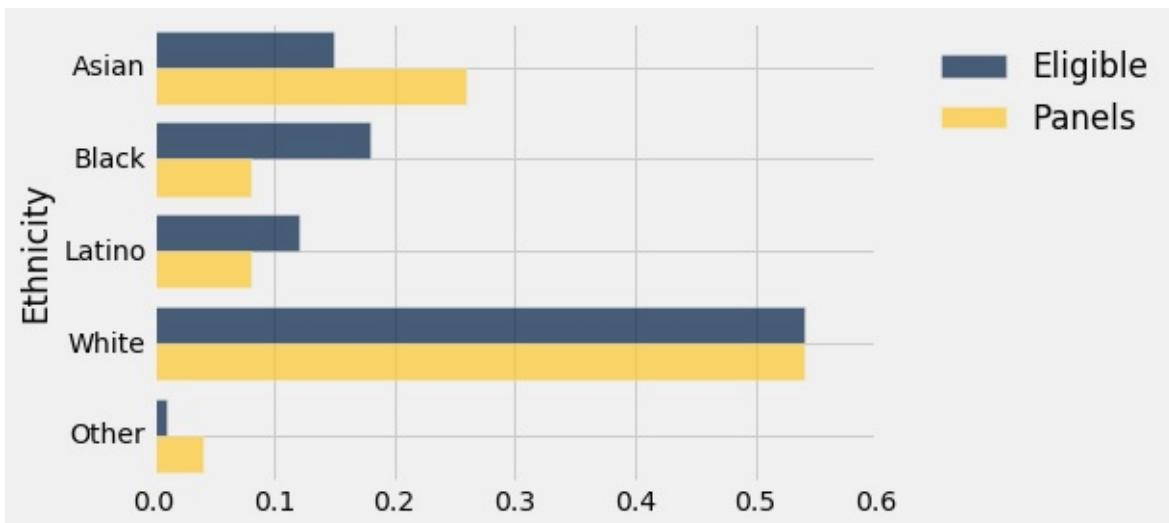
jury

```

Ethnicity	Eligible	Panels
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

Some ethnicities are overrepresented and some are underrepresented on the jury panels in the study. A bar chart is helpful for visualizing the differences.

```
jury.barh('Ethnicity')
```



Comparison with Panels Selected at Random

What if we select a random sample of 1,453 people from the population of eligible jurors?

Will the distribution of their ethnicities look like the distribution of the panels above?

We can answer these questions by using `sample_proportions` and augmenting the `jury` table with a column of the proportions in our sample.

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.

In the cell below, we sample at random 1453 times from the distribution of eligible jurors, and display the distribution of the random sample along with the distributions of the eligible jurors and the panel in the data.

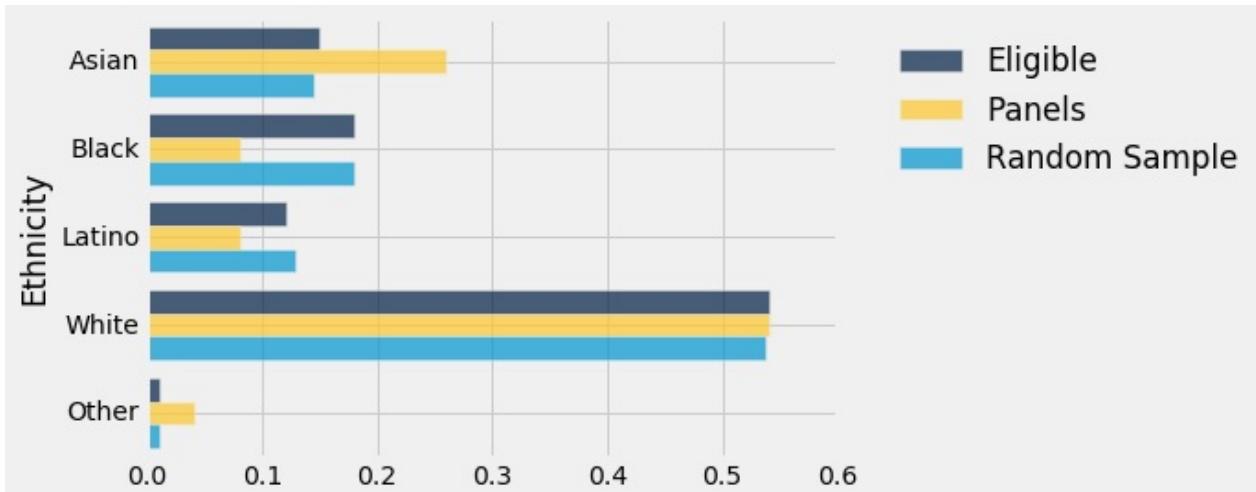
```
eligible_population = jury.column('Eligible')
sample_distribution = sample_proportions(1453,
eligible_population)
panels_and_sample = jury.with_column('Random Sample',
sample_distribution)
panels_and_sample
```

Ethnicity	Eligible	Panels	Random Sample
Asian	0.15	0.26	0.144529
Black	0.18	0.08	0.179628
Latino	0.12	0.08	0.128699
White	0.54	0.54	0.53682
Other	0.01	0.04	0.0103235

The distribution of the random sample is quite close to the distribution of the eligible population, unlike the distribution of the panels.

As always, it helps to visualize.

```
panels_and_sample.bah('Ethnicity')
```



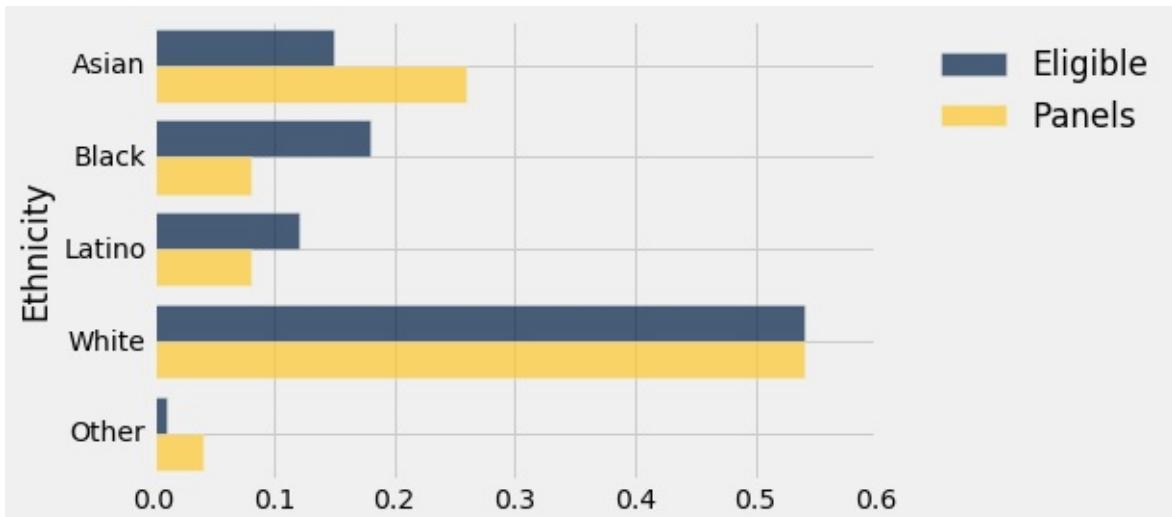
The bar chart shows that the distribution of the random sample resembles the eligible population but the distribution of the panels does not.

To assess whether this observation is particular to one random sample or more general, we can simulate multiple panels under the model of random selection and see what the simulations predict. But we won't be able to look at thousands of bar charts like the one above. We need a statistic that will help us assess whether or not the model of random selection is supported by the data.

A New Statistic: The Distance between Two Distributions

We know how to measure how different two numbers are – if the numbers are x and y , the distance between them is $|x - y|$. Now we have to quantify the distance between two distributions. For example, we have to measure the distance between the blue and gold distributions below.

```
jury.barch('Ethnicity')
```



For this we will compute a quantity called the *total variation distance* between two distributions. The calculation is as an extension of the calculation of the distance between two numbers.

To compute the total variation distance, we first take the difference between the two proportions in each category.

```
# Augment the table with a column of differences between
# proportions

jury_with_diffs = jury.with_column(
    'Difference', jury.column('Panels') -
    jury.column('Eligible')
)
jury_with_diffs
```

Ethnicity	Eligible	Panels	Difference
Asian	0.15	0.26	0.11
Black	0.18	0.08	-0.1
Latino	0.12	0.08	-0.04
White	0.54	0.54	0
Other	0.01	0.04	0.03

Take a look at the column `Difference` 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.

This is numerical evidence of the fact that in the bar chart, the gold bars exceed the blue bars by exactly as much as the blue bars exceed the gold. The proportions in each of the two columns `Panels` 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.

```
jury_with_diffs = jury_with_diffs.with_column(
    'Absolute Difference',
    np.abs(jury_with_diffs.column('Difference'))
)

jury_with_diffs
```

Ethnicity	Eligible	Panels	Difference	Absolute Difference
Asian	0.15	0.26	0.11	0.11
Black	0.18	0.08	-0.1	0.1
Latino	0.12	0.08	-0.04	0.04
White	0.54	0.54	0	0
Other	0.01	0.04	0.03	0.03

```
jury_with_diffs.column('Absolute Difference').sum()/2
```

```
0.14000000000000001
```

This quantity 0.14 is the *total variation distance* (TVD) between the distribution of ethnicities in the eligible juror population and the distribution in the panels.

We could 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.

Simulating One Value of the Statistic

We will use the total variation distance between distributions as the statistic to simulate. It will help us decide whether the model of random selection is good, because large values of the distance will be evidence against the model.

Keep in mind that **the observed value of our statistic is 0.14**, calculated above.

Since we are going to be computing total variation distance repeatedly, we will write a function to compute it.

The function `total_variation_distance` returns the TVD between distributions in two arrays.

```
def total_variation_distance(distribution_1, distribution_2):
    return sum(np.abs(distribution_1 - distribution_2)) / 2
```

This function will help us calculate our statistic in each repetition of the simulation. But first, let's check that it gives the right answer when we use it to compute the distance between the blue (eligible) and gold (panels) distributions above.

```
total_variation_distance(jury.column('Panels'),
jury.column('Eligible'))
```

```
0.14000000000000001
```

This agrees with the value that we computed directly without using the function.

In the cell below we use the function to compute the TVD between the distributions of the eligible jurors and one random sample. This is the code for simulating one value of our statistic. Recall that `eligible_population` is the array containing the distribution of the eligible jurors.

```
sample_distribution = sample_proportions(1453,
eligible_population)
total_variation_distance(sample_distribution,
eligible_population)
```

```
0.030956641431521031
```

Notice that the distance is quite a bit smaller than 0.14, the distance between the distribution of the panels and the eligible jurors.

We are now ready to run a simulation to assess the model of random selection.

Predicting the Statistic Under the Model of Random Selection

The total variation distance between the distributions of the random sample and 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 see how much the statistic varies across different random samples.

The code below simulates the statistic based on a large number of replications of the random sampling process, following our usual sequence of steps for simulation. The body of the `for` loop repeats the code we used to simulate one value of the statistics, and then appends the simulated value to the collection array `tvds`.

```
# Simulate total variation distance between
# distribution of sample selected at random
# and distribution of eligible population

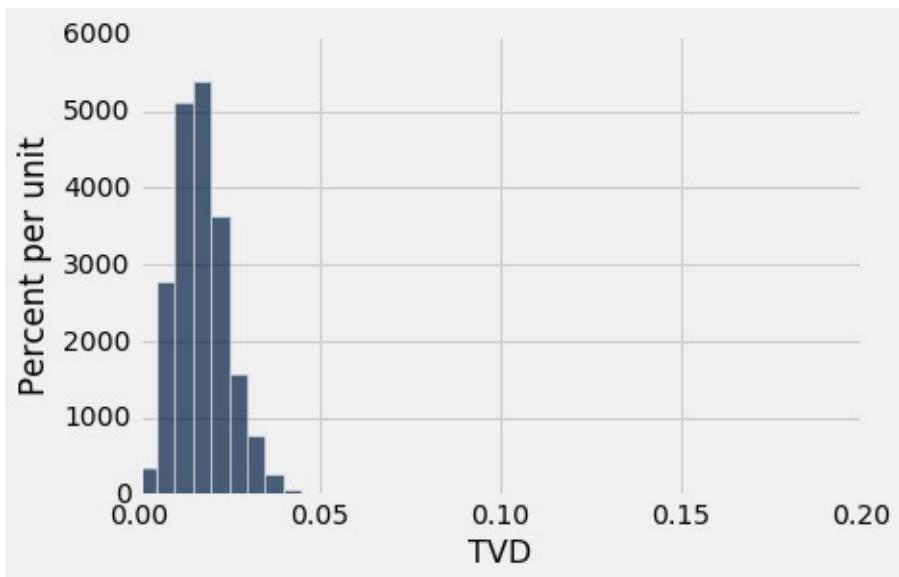
eligible_population = jury.column('Eligible')
panel_size = 1453

tvds = make_array()

repetitions = 5000
for i in np.arange(repetitions):
    sample_distribution = sample_proportions(panel_size,
eligible_population)
    new_tvd = total_variation_distance(sample_distribution,
eligible_population)
    tvds = np.append(tvds, new_tvd)
```

The empirical histogram of the simulated distances 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 about 0.05.

```
Table().with_column('TVD', tvds).hist(bins=np.arange(0, 0.2,
0.005))
```



Assessing the Model of Random Selection

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 at all like a typical distance between a random sample and the eligible population.

The data in the panels is not consistent with the predicted values of the statistic based on the model of random selection. So our analysis supports the ACLU's calculation that the panels were not representative of the distribution provided for the eligible jurors.

Some Possible Explanations for the Differences

As with most such analyses, however, our analysis does not say *why* the distributions are different or what the difference might imply.

The ACLU report discusses several possible 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. The ACLU listed several other reasons as well. 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 from the distribution in the `Eligible` column.

Questions about the Data

We have developed a powerful technique that helps decide whether one distribution looks like a random sample from another. But data science is about more than techniques. In particular, data science always involves a thoughtful examination of how the data were gathered.

Eligible Jurors. First, it is important to remember that not everyone is eligible to serve on a jury. On its [website](#), the Superior Court of Alameda County says, "You may be called to serve if you are 18 years old, a U.S. citizen and a resident of the county or district where summoned. You must be able to understand English, and be physically and mentally capable of serving. In addition, you must not have served as any kind of juror in the past 12 months, nor have been convicted of a felony."

The Census doesn't maintain records of the populations in all these categories. Thus the ACLU had to obtain the demographics of eligible jurors in some other way. Here is their own description of the process they followed and some flaws that it might contain.

"For the purpose of determining the demographics of Alameda County's jury eligible population, we used a declaration that was prepared for the Alameda County trial of People v. Stuart Alexander in 2002. In the declaration, Professor Weeks, a demographer at San Diego State University, estimated the jury eligible population for Alameda County, using the 2000 Census data. To arrive at this estimate, Professor Weeks took into account the number of people who are not eligible for jury services because they do not speak English, are not citizens, are under 18, or have a felony conviction. Thus, his estimate should be an accurate assessment of who is actually eligible for jury service in Alameda County, much more so than simply reviewing the Census report of the race and ethnicity of all people living in Alameda County. It should be noted, however, that the Census data on which Professor Weeks relied is now ten years old and the demographics of the county may have changed by two or three percent in some categories."

Thus the distribution of ethnicities of eligible jurors used in the analysis is itself an estimate and might be somewhat out of date.

Panels. In addition, panels aren't selected from the entire eligible population. The Superior Court of Alameda County says, "The objective of the court is to provide an accurate cross-section of the county's population. The names of jurors are selected at random from everyone who is a registered voter and/or has a driver's license or identification card issued by the Department of Motor Vehicles."

All of this raises complex questions about how to accurately estimate the ethnic composition of eligible jurors in Alameda County.

It is not clear exactly how the 1453 panelists were classified into the different ethnic categories (the ACLU report says that "attorneys ... cooperated in collecting jury pool data"). There are serious social, cultural, and political factors that affect who gets classified or self-

classifies into each ethnic category. We also don't know whether the definitions of those categories in the panels are the same as those used by Professor Weeks who in turn used Census categories in his estimation process. Thus there are also questions about the correspondence between the two distributions being compared.

Thus, while we have a clear conclusion about the data in our table – the panels do not look like a random sample from the distribution provided for eligible jurors – questions about the nature of the data prevent us from concluding anything broader.

[Interact](#)

Decisions and Uncertainty

We have seen several examples of assessing models that involve chance, by comparing observed data to the predictions made by the models. In all of our examples, there has been no doubt about whether the data were consistent with the model's predictions. The data were either very far away from the predictions, or very close to them.

But outcomes are not always so clear cut. How far is "far"? Exactly what does "close" mean? While these questions don't have universal answers, there are guidelines and conventions that you can follow. In this section we will describe some of them.

But first let us develop a general framework of decision making, into which all our examples will fit.

What we have developed while assessing models are some of the fundamental concepts of statistical tests of hypotheses. 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. You will see that they are consistent with the sequence of steps we have used for assessing models.

Step 1: The Hypotheses

All statistical tests attempt to choose between two views of the world. Specifically, the choice is between two views about how the data were generated. These two views are called *hypotheses*.

The null hypothesis. This is a clearly defined model about chances. It says that the data were generated at random under clearly specified assumptions about the randomness. 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.

From a practical perspective, **the null hypothesis is a hypothesis under which you can simulate data.**

In the example about Mendel's model for the colors of pea plants, the null hypothesis is that the assumptions of his model are good: each plant has a 75% chance of having purple flowers, independent of all other plants.

Under this hypothesis, we were able to simulate random samples, by using `sample_proportions(929, [0.75, 0.25])`. We used a sample size of 929 because that's the number of plants Mendel grew.

The alternative hypothesis. This says that some reason other than chance made the data differ from the predictions of the model in the null hypothesis.

In the example about Mendel's plants, the alternative hypothesis is simply that his model isn't good.

Step 2: The Test Statistic

In order to decide between the two hypothesis, we must choose a statistic that we can use to make the decision. This is called the **test statistic**.

In the example of Mendel's plants, our statistic was the absolute difference between the sample percent and 75% which was predicted by his model.

$$|\text{sample percent of purple-flowering plants} - 75|$$

To see how to make the choice in general, look at the alternative hypothesis. What values of the statistic will make you think that the alternative hypothesis is a better choice than the null?

- If the answer is "big values," you might have a good choice of statistic.
- So also if the answer is "small values."
- But if the answer is "both big values and small values," we recommend that you look again at your statistic and see if taking an absolute value can change the answer to just "big values".

In the case of the pea plants, a sample percent of around 75% will be consistent with the model, but percents much bigger or much less than 75 will make you think that the model isn't good. This indicates that the statistic should be the *distance* between the sample percent and 75, that is, the absolute value of the difference between them. Big values of the distance will make you lean towards the alternative.

The **observed value of the test statistic** is the value of the statistic you get from the data in the study, not a simulated value. Among Mendel's 929 plants, 705 had purple flowers. The observed value of the test statistic was therefore

```
abs ( 100 * (705 / 929) - 75 )
```

```
0.8880516684607045
```

Step 3: The Distribution of the Test Statistic, Under the Null Hypothesis

The main computational aspect of a test of hypotheses is figuring out *what the values of the test statistic might be if the null hypothesis were true*.

The test statistic is simulated based on the assumptions of the model in the null hypothesis. That model involves chance, so the statistic comes out differently when you simulate it multiple times.

By simulating the statistic repeatedly, we get a good sense of its possible values and which ones are more likely than others. In other words, we get a good approximation to the probability distribution of the statistic, as predicted by the model in the null hypothesis.

As with all distributions, it is very useful to visualize this distribution by a histogram. We have done so in all our examples.

Step 4. The Conclusion of the Test

The choice between the null and alternative hypotheses depends on the comparison between what you computed in Steps 2 and 3: the observed value of the 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. This was the case with the assessment of Mendel's model.

But if the two are not consistent with each other, as is the case in our example about Alameda County jury panels, then the data do not support the null hypothesis. That is why we concluded that the jury panels were not selected at random. Something other than chance affected their composition.

If the data do not support the null hypothesis, we say that the test *rejects* the null hypothesis.

The Meaning of "Consistent"

In the example about Alameda County juries, it was apparent that our observed test statistic was far from what was predicted by the null hypothesis. In the example about pea flowers, it is just as clear that the observed statistic is consistent with the distribution that the null predicts. So in both of the examples, it is clear which hypothesis to choose.

But sometimes the decision is not so clear. 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 under the null. That will allow your reader to make his or her own judgment about whether the two are consistent.

Here is an example where the decision requires judgment.

The GSI's Defense

A Berkeley Statistics class of about 350 students was divided into 12 discussion sections led by Graduate Student Instructors (GSIs). After the midterm, students in Section 3 noticed that their scores were on average lower than the rest of the class.

In such situations, students tend to grumble about the section's GSI. Surely, they feel, there must have been something wrong with the GSI's teaching. Or else why would their section have done worse than others?

The GSI, typically more experienced about statistical variation, often has a different perspective: if you simply draw a section of students at random from the whole class, their average score could resemble the score that the students are unhappy about, just by chance.

The GSI's position is a clearly stated chance model. We can simulate data under this model. Let's test it out.

Null Hypothesis. The average score of the students in Section 3 is like the average score of the same number of students picked at random from the class.

Alternative Hypothesis. No, it's too low.

A natural statistic here is the average of the scores. Low values of the average will make us lean towards the alternative.

Let's take a look at the data.

The table `scores` contains the section number and midterm score for each student in the class. The midterm scores were integers in the range 0 through 25; 0 means that the student didn't take the test.

```
scores = Table.read_table('scores_by_section.csv')
scores
```

Section	Midterm
1	22
2	12
2	23
2	14
1	20
3	25
4	19
1	24
5	8
6	14

... (349 rows omitted)

To find the average score in each section, we will use `group`.

```
section_averages = scores.group('Section', np.average)
section_averages.show()
```

Section	Midterm average
1	15.5938
2	15.125
3	13.6667
4	14.7667
5	17.4545
6	15.0312
7	16.625
8	16.3103
9	14.5667
10	15.2353
11	15.8077
12	15.7333

The average score of Section 3 is 13.667, which does look low compared to the other section averages. But is it lower than the average of a section of the same size selected at random from the class?

To answer this, we can select a section at random from the class and find its average. To select a section at random we need to know how big Section 3 is, which we can by once again using `group`.

```
scores.groupby('Section')
```

Section	count
1	32
2	32
3	27
4	30
5	33
6	32
7	24
8	29
9	30
10	34

... (2 rows omitted)

Section 3 had 27 students.

Now we can figure out how to create one simulated value of our test statistic, the random sample average.

First we have to select 27 scores at random without replacement. Since the data are already in a table, we will use the Table method `sample`.

Remember that by default, `sample` draws with replacement. The optional argument `with_replacement = False` produces a random sample drawn without replacement.

```
scores_only = scores.drop('Section')
sampled_scores = scores_only.sample(27, with_replacement=False)
sampled_scores
```

Midterm
13
21
23
15
24
18
17
13
8
0
...
(17 rows omitted)

The average of these 27 randomly selected scores is

```
np.average(sampled_scores.column('Midterm'))
```

```
15.592592592592593
```

That's the average of 27 randomly selected scores. The cell below collects the code necessary for generating this random average.

Now we can simulate the random sample average by repeating the calculation multiple times.

```
averages = make_array()

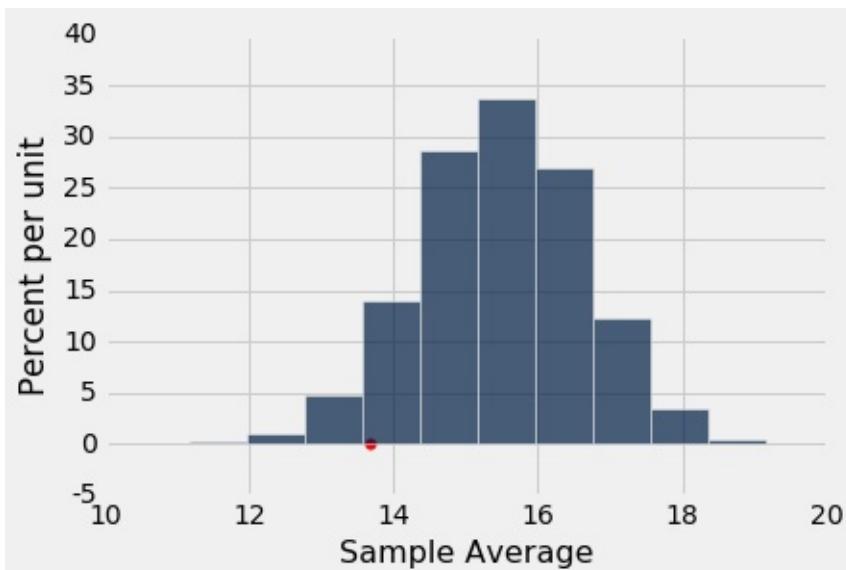
repetitions = 10000
for i in np.arange(repetitions):
    sampled_scores = scores_only.sample(27,
with_replacement=False)
    averages = np.append(averages,
np.average(sampled_scores.column('Midterm')))
```

Here is the histogram of the simulated averages. It shows the distribution of what the Section 3 average might have been, if Section 3 had been selected at random from the class.

The observed Section 3 average score of 13.667 is shown as a red dot on the horizontal axis. You can ignore the last line of code; it just draws the dot.

```
sample_averages = Table().with_column('Sample Average',
averages)
sample_averages.hist()

observed_statistic = 13.667
plots.scatter(observed_statistic, 0, color='red', s=30);
```



As we said earlier, small values of the test statistic will make us lean towards the alternative hypothesis, that the average score in the section is too low for it to look like a random sample from the class.

Is the observed statistic of 13.667 "too low" in relation to this distribution? In other words, is the red far enough out into the left hand tail of the histogram for you to think that it is "too far"?

It's up to you to decide! Use your judgment. Go ahead – it's OK to do so.

Conventional Cut-offs and the P-value

If you don't want to make your own judgment, there are conventions that you can follow. These conventions tell us how far out into the tails is considered "too far".

The conventions are based on the area in the tail, starting at the observed statistic (the red dot) and looking in the direction that makes us lean toward the alternative (the left side, in this example). If the area of the tail is small, the observed statistic is far away from the values most commonly predicted by the null hypothesis.

Remember that in a histogram, area represents percent. To find the area in the tail, we have to find the percent of sample averages that were less than or equal to the average score of Section 3, where the red dot is. The array `averages` contains the averages for all 10,000 repetitions of the random sampling, and `section_3_average` is 13.667, the average score of Section 3.

```
np.count_nonzero(averages <= section_3_average)/repetitions
```

```
0.0569
```

Just about 5.7% of the simulated random sample averages were 3.667 or below. If we had drawn the students of Section 3 at random from the whole class, the chance that their average would be 13.667 or lower is about 5.7%.

This chance has an impressive name. It is called the *observed significance level* of the test. That's a mouthful, and so it is commonly called the *P-value* of the test.

Definition of P-value¶

The P-value is the chance, based on the model in the null hypothesis, that the test statistic is equal to the value that was observed in the data or is even further in the direction of the alternative.

If a P-value is small, that means the tail beyond the observed statistic is small and so the observed statistic is far away from what the null predicts. This implies that the data support the alternative hypothesis better than they support the null.

How small is "small"? According to the conventions:

- If the P-value is less than 5%, it is considered small and the result is called "statistically significant."
- If the P-value is even smaller – less than 1% – the result is called "highly statistically significant."

By this convention, our P-value of 5.7% is not considered small. So we have to conclude that the GSI's defense holds good – the average score of Section 3 is like those generated by random chance. Formally, the result of the test is not statistically significant.

When you make a conclusion in this way, we recommend that you don't just say whether or not the result is statistically significant. Along with your conclusion, provide the observed statistic and the P-value as well, so that readers can use their own judgment.

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.

[Interact](#)

Comparing Two Samples

We have seen several examples of assessing whether a single sample looks like random draws from a specified chance model.

- Did the Alameda County jury panels look like a random sample from the population of eligible jurors?
- Did the pea plants that Mendel grew have colors that were consistent with the chances he specified in his model?

In all of these cases there was just one random sample, and we were trying to decide how it had been generated. But often, data scientists have to compare two random samples with each other. For example, they might have to compare the outcomes of patients who have been assigned at random to a treatment group and a control group. Or they might have randomized internet users to receive two different versions of a website, after which they would want to compare the actions of the two random groups.

In this chapter, we develop a way of using Python to compare two random samples and answer questions about the similarities and differences between them. You will see that the methods we develop have diverse applications. Our examples are from medicine and public health as well as football!

[Interact](#)

A/B Testing

In modern data analytics, deciding whether two numerical samples come from the same underlying distribution is called *A/B testing*. The name refers to the labels of the two samples, A and B.

We will develop the method in the context of an example. The data come from a sample of newborns in a large hospital system. We will treat it as if it were a simple random sample though the sampling was done in multiple stages. [Stat Labs](#) by Deborah Nolan and Terry Speed has details about a larger dataset from which this set is drawn.

Smokers and Nonsmokers

The table `baby` contains the following variables for 1,174 mother-baby pairs: 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, pregnancy weight in pounds, and whether or not the mother smoked during pregnancy.

```
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)

One of the aims of the study was to see whether maternal smoking was associated with birth weight. Let's see what we can say about the two variables.

We'll start by selecting just `Birth Weight` and `Maternal Smoker`. There are 715 non-smokers among the women in the sample, and 459 smokers.

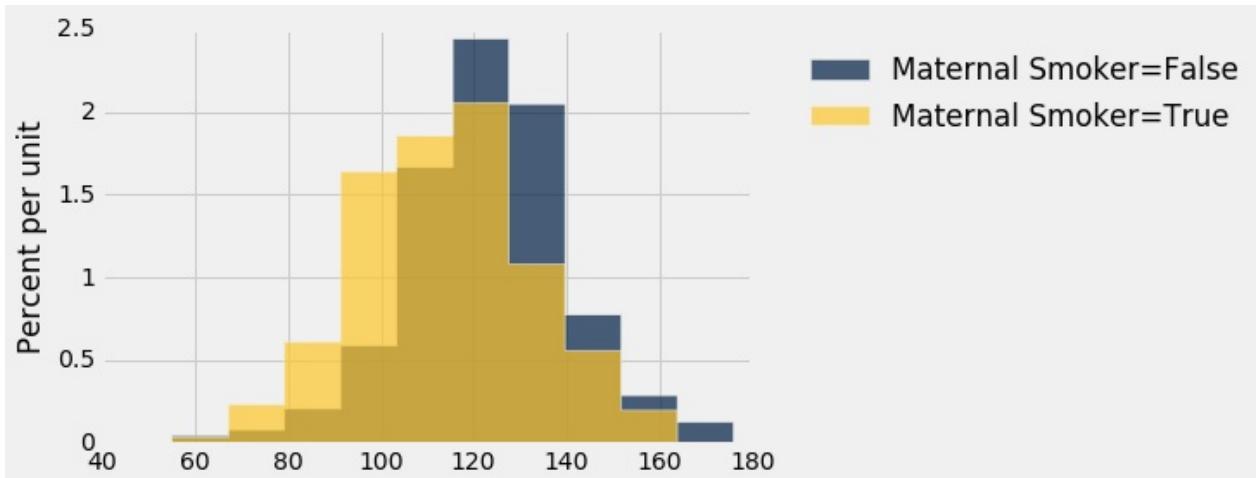
```
smoking_and_birthweight = baby.select('Maternal Smoker', 'Birth Weight')
```

```
smoking_and_birthweight.group('Maternal Smoker')
```

Maternal Smoker	count
False	715
True	459

Let's look at the distribution of the birth weights of the babies of the non-smoking mothers compared to those of the smoking mothers. To generate two overlaid histograms, we will use `hist` with the optional `group` argument which is a column label or index. The rows of the table are first grouped by this column and then a histogram is drawn for each one.

```
smoking_and_birthweight.hist('Birth Weight', group = 'Maternal Smoker')
```



The distribution of the weights of the babies born to mothers who smoked appears to be shifted slightly to the left of the distribution corresponding to non-smoking mothers. The weights of the babies of the mothers who smoked seem lower, on average than the weights of the babies of the non-smokers.

This raises the question of whether the difference reflects just chance variation or a difference in the distributions in the larger population. Could it be that there is no difference between the two distributions in the population, but we are seeing a difference in the samples just because of the mothers who happened to be selected?

The Hypotheses

We can try to answer this question by a test of hypotheses. The chance model that we will test says that there is no underlying difference; the distributions in the samples are different just due to chance. Formally, this is the null hypothesis.

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: In the population, the babies of the mothers who smoke have a lower birth weight, on average, than the babies of the non-smokers.

Test Statistic

The alternative hypothesis compares the average birth weights of the two groups and says that the average for the mothers who smoke is smaller. Therefore it is reasonable for us to use the difference between the two group means as our statistic.

We will do the subtraction in the order "average weight of the smoking group — average weight of the non-smoking group". Small values (that is, large negative values) of this statistic will favor the alternative hypothesis.

The observed value of the test statistic is about **-9.27** ounces.

```
means_table = smoking_and_birthweight.groupby('Maternal Smoker',
np.average)
means_table
```

Maternal Smoker	Birth Weight average
False	123.085
True	113.819

```
means = means_table.column(1)
observed_difference = means.item(1) - means.item(0)
observed_difference
```

-9.266142572024918

Predicting the Statistic Under the Null Hypothesis

To see how the statistic should vary under the null hypothesis, we have to figure out how to simulate the statistic under that hypothesis. A clever method based on *random permutations* does just that.

If there were no difference between the two distributions in the underlying population, then whether a birth weight has the label `True` or `False` with respect to maternal smoking should make no difference to the average. The idea, then, is to shuffle all the birth weights randomly among the mothers. This is called *random permutation*.

Take the difference of the two new group means: the mean of the shuffled weights assigned to the smokers and the mean of the shuffled weights assigned to the non-smokers. This is a simulated value of the test statistic under the null hypothesis.

Let's see how to do this. It's always a good idea to start with the data.

`smoking_and_birthweight`

Maternal Smoker	Birth Weight
False	120
False	113
True	128
True	108
False	136
False	138
False	132
True	120
True	143
False	140

... (1164 rows omitted)

There are 1,174 rows in the table. To shuffle all the birthweights, we will draw a random sample of 1,174 rows without replacement. Then the sample will include all the rows of the table, in random order.

We can use the Table method `sample` with the optional `with_replacement=False` argument. We don't have to specify a sample size, because by default, `sample` draws as many times as there are rows in the table.

```
shuffled_weights =
smoking_and_birthweight.sample(with_replacement =
False).column(1)
original_and_shuffled =
smoking_and_birthweight.with_column('Shuffled Birth Weight',
shuffled_weights)
```

`original_and_shuffled`

Maternal Smoker	Birth Weight	Shuffled Birth Weight
False	120	145
False	113	124
True	128	100
True	108	116
False	136	124
False	138	117
False	132	118
False	120	110
True	143	125
False	140	85

... (1164 rows omitted)

Each mother now has a random birth weight assigned to her. If the null hypothesis is true, all these random arrangements should be equally likely.

Let's see how different the average weights are in the two randomly selected groups.

```
all_group_means = original_and_shuffled.groupby('Maternal Smoker',
np.average)
all_group_means
```

Maternal Smoker	Birth Weight average	Shuffled Birth Weight average
False	123.085	118.829
True	113.819	120.449

The averages of the two randomly selected groups are quite a bit closer than the averages of the two original groups.

```
shuffled_means = original_and_shuffled.groupby('Maternal Smoker',
np.average).column(2)
difference = shuffled_means.item(1) - shuffled_means.item(0)
difference
```

1.6194311135487567

But could a different shuffle have resulted in a larger difference between the group averages? To get a sense of the variability, we must simulate the difference many times.

Let's collect all the code that we need for simulating one value of the difference between averages, under the null hypothesis. Notice that because we are using the same label each time for the column of shuffled weights, the existing column just gets overwritten by the newly generated one. This works well for us because we don't need to save all the shuffled values. We just need to save the value of the statistic.

```
# Generate one value of the test statistic under the null hypothesis

# Shuffle all the weights and assign the shuffled weights to the two groups of mothers
shuffled_weights =
smoking_and_birthweight.sample(with_replacement =
False).column(1)
original_and_shuffled =
smoking_and_birthweight.with_column('Shuffled Birth Weight',
shuffled_weights)

# Find the difference between the means of two randomly assigned groups
shuffled_means = original_and_shuffled.group('Maternal Smoker',
np.average).column(2)
difference = shuffled_means.item(1) - shuffled_means.item(0)
difference
```

```
0.7072322013498393
```

Permutation Test

Tests based on random permutations of the data are called *permutation tests*. We are performing one in this example. In the cell below, we will simulate our test statistic – the difference between the averages of the two groups – many times and collect the differences in an array. The code in the body of the for loop is just copied over from the cell above.

```

differences = make_array()

repetitions = 5000
for i in np.arange(repetitions):

    shuffled_weights =
smoking_and_birthweight.sample(with_replacement =
False).column(1)
    original_and_shuffled =
smoking_and_birthweight.with_column('Shuffled Birth Weight',
shuffled_weights)

    shuffled_means = original_and_shuffled.group('Maternal
Smoker', np.average).column(2)
    simulated_difference = shuffled_means.item(1) -
shuffled_means.item(0)

    differences = np.append(differences, simulated_difference)

```

The array `differences` contains 5,000 simulated values of our test statistic – the difference between the mean weight in the smoking group and the mean weight in the non-smoking group.

Conclusion of the Test

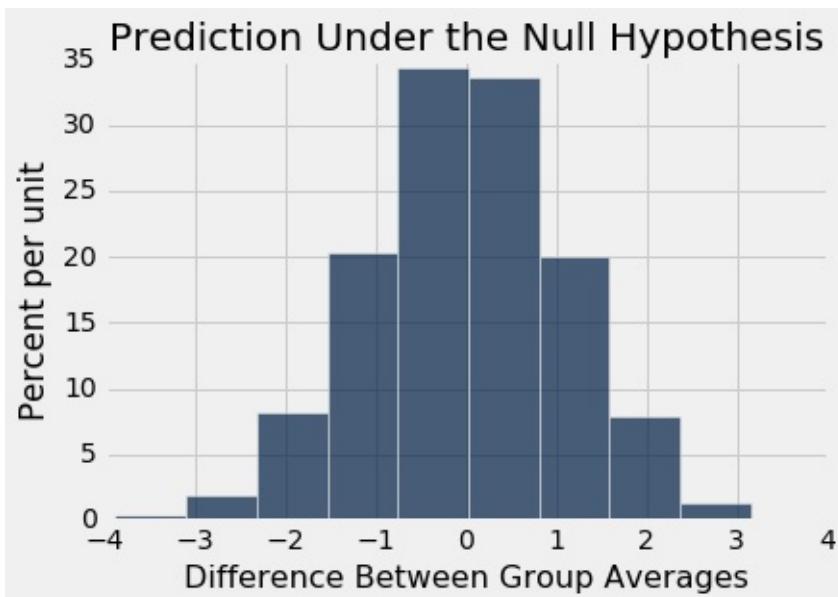
The histogram below shows the distribution of these 5,000 values. It is the empirical distribution of the test statistic simulated under the null hypothesis. It is a prediction made by the null hypothesis, about the statistic.

```

Table().with_column('Difference Between Group Averages',
differences).hist()
print('Observed Difference:', observed_difference)
plots.title('Prediction Under the Null Hypothesis');

```

Observed Difference: -9.266142572024918



Notice how the distribution is centered around 0. This makes sense, because under the null hypothesis the two groups should have roughly the same average. Therefore the difference between the group averages should be around 0.

The observed difference in the original sample is about **-9.27** ounces, which doesn't even appear on the horizontal scale of the histogram. The observed value of the statistic and the predicted behavior of the statistic under the null hypothesis are inconsistent.

The conclusion of the test is that the data support the alternative more than they support the null. The average birth weight of babies born to mothers who smoke is less than the average birth weight of babies born to non-smokers.

If you want to compute an empirical P-value, remember that low values of the statistic favor the alternative hypothesis.

```
empirical_P = np.count_nonzero(differences <=
    observed_difference) / repetitions
empirical_P
```

```
0.0
```

The empirical P-value is 0, meaning that none of the 5,000 observed samples resulted in a difference of -9.27 or lower. This is an approximation; the exact chance of getting a difference in that range is not 0 but it is vanishingly small.

A Function to Simulate the Differences Under the Null Hypothesis

We will want to perform permutation tests for the difference between averages in other contexts as well. Let us define a function that generates the array of simulated differences, based on the code that we wrote above. That will save us time later.

The function `difference_of_permuted_sample_means` takes four arguments:

- the name of the data table
- the label of the column containing the variable whose average is of interest
- the label of the column of group labels
- the number of repetitions

It returns and array of simulated differences in group means, each computed by first randomly permuting the data and assigning random values to each group. The length of the array is equal to the number of repetitions.

```
def permuted_sample_average_difference(table, label,
group_label, repetitions):

    tbl = table.select(group_label, label)

    differences = make_array()
    for i in np.arange(repetitions):
        shuffled =tbl.sample(with_replacement =
False).column(1)
        original_and_shuffled =tbl.with_column('Shuffled Data',
shuffled)

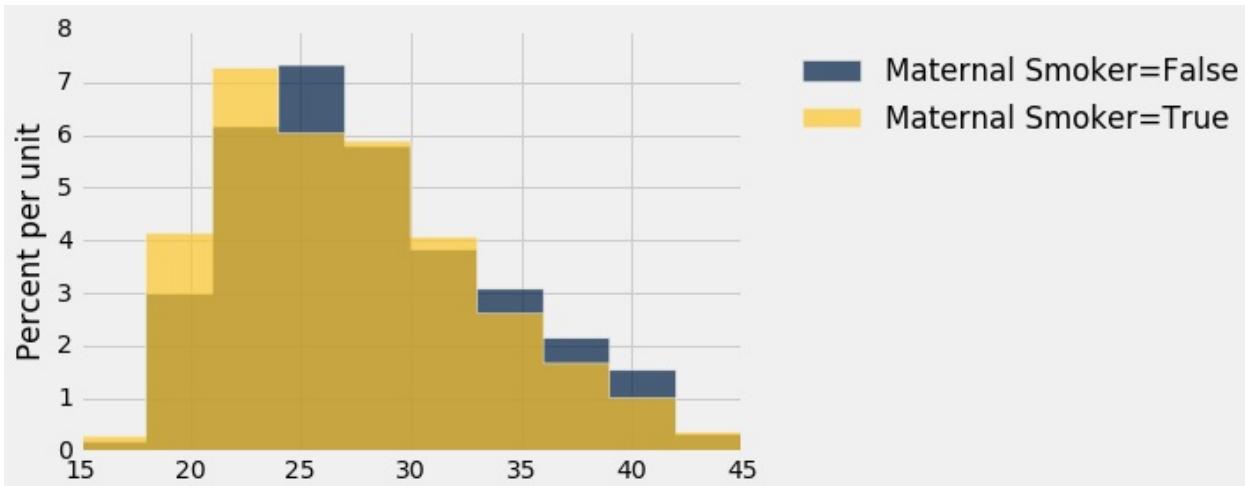
        shuffled_means =
original_and_shuffled.group(group_label, np.average).column(2)
        simulated_difference = shuffled_means.item(1) -
shuffled_means.item(0)

        differences = np.append(differences,
simulated_difference)

    return differences
```

As an example of the use of this function, we will test whether there was any difference in the ages of the smoking and non-smoking mothers. The histograms of the two distributions in the sample are a little different. The smokers seem a little younger on average.

```
smoking_and_age = baby.select('Maternal Smoker', 'Maternal Age')
smoking_and_age.hist('Maternal Age', group = 'Maternal Smoker')
```



```
smoking_and_age.group('Maternal Smoker', np.average)
```

Maternal Smoker	Maternal Age average
False	27.5441
True	26.7364

The observed difference between the average ages is about **-0.8** years.

```
observed_means = smoking_and_age.group('Maternal Smoker',
np.average).column(1)
observed_difference = observed_means.item(1) -
observed_means.item(0)
observed_difference
```

```
-0.8076725017901509
```

If the underlying distributions of ages in the two groups are the same, then the empirical distribution of the difference based on permuted samples will predict how the statistic will vary due to chance.

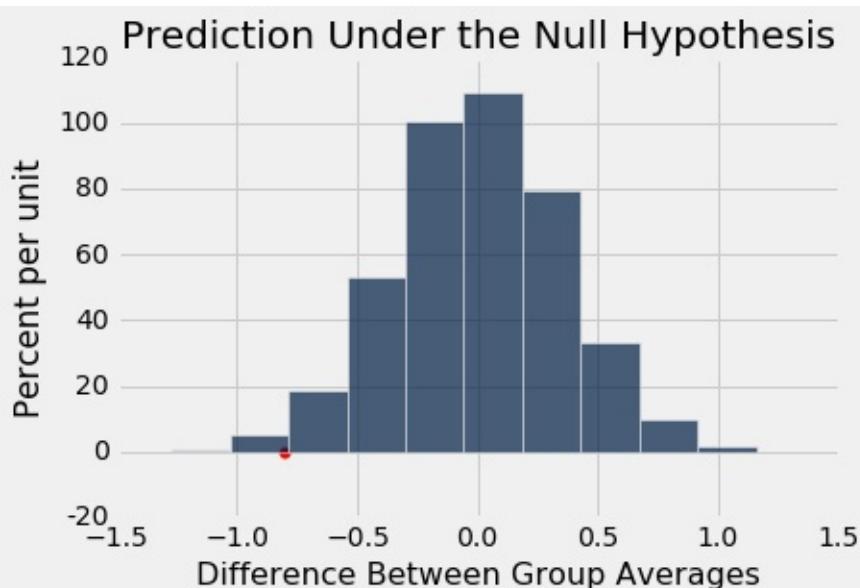
We can generate such differences using the function we just defined.

```
differences = permuted_sample_average_difference(baby, 'Maternal Age', 'Maternal Smoker', 5000)
```

The observed difference is in the tail of the empirical distribution of the differences simulated under the null hypothesis.

```
Table().with_column('Difference Between Group Averages',
differences).hist()
plots.scatter(observed_difference, 0, color='red', s=30)
plots.title('Prediction Under the Null Hypothesis')
print('Observed Difference:', observed_difference)
```

Observed Difference: -0.8076725017901509



The empirical P-value of the test is the proportion of simulated differences that were equal to or less than the observed difference. This is because low values of the difference favor the alternative hypothesis that the smokers were younger on average.

```
empirical_P = np.count_nonzero(differences <=
observed_difference) / 5000
empirical_P
```

0.012

The empirical P-value is just over 1%, which is less than 5% and therefore the result is statistically significant. The test supports the hypothesis that the smokers were younger on average.

[Interact](#)

Deflategate

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. Each row corresponds to one football. 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('deflategate.csv')
football.show()
```

Team	Blakeman	Prioleau
Patriots	11.5	11.8
Patriots	10.85	11.2
Patriots	11.15	11.5
Patriots	10.7	11
Patriots	11.1	11.45
Patriots	11.6	11.95
Patriots	11.85	12.3
Patriots	11.1	11.55
Patriots	10.95	11.35
Patriots	10.5	10.9
Patriots	10.9	11.35
Colts	12.7	12.35
Colts	12.75	12.3
Colts	12.5	12.95
Colts	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(1)+football.column(2))/2
).drop(1, 2)
football.show()
```

Team	Combined
Patriots	11.65
Patriots	11.025
Patriots	11.325
Patriots	10.85
Patriots	11.275
Patriots	11.775
Patriots	12.075
Patriots	11.325
Patriots	11.15
Patriots	10.7
Patriots	11.125
Colts	12.525
Colts	12.525
Colts	12.725
Colts	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.

We can calculate the drop in pressure for each football, by first setting up an array of the starting values. For this we will need an array consisting of 11 values each of which is 12.5, and another consisting of four values each of which is all 13. We will use the NumPy function `np.ones`, which takes a count as its argument and returns an array of that many elements, each of which is 1.

```
np.ones(11)
```

```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

```
patriots_start = 12.5 * np.ones(11)
colts_start = 13 * np.ones(4)
start = np.append(patriots_start, colts_start)
start
```

```
array([ 12.5,  12.5,  12.5,  12.5,  12.5,  12.5,  12.5,  12.5,
       12.5,  12.5,  13. ,  13. ,  13. ,  13. ])
```

The drop in pressure for each football is the difference between the starting pressure and the combined pressure measurement.

```
drop = start - football.column('Combined')
football = football.with_column('Pressure Drop', drop)
football.show()
```

Team	Combined	Pressure Drop
Patriots	11.65	0.85
Patriots	11.025	1.475
Patriots	11.325	1.175
Patriots	10.85	1.65
Patriots	11.275	1.225
Patriots	11.775	0.725
Patriots	12.075	0.425
Patriots	11.325	1.175
Patriots	11.15	1.35
Patriots	10.7	1.8
Patriots	11.125	1.375
Colts	12.525	0.475
Colts	12.525	0.475
Colts	12.725	0.275
Colts	12.35	0.65

It looks as though the Patriots' drops were larger than the Colts'. Let's look at the average drop in each of the two groups. We no longer need the combined scores.

```
football = football.drop('Combined')
football.groupby('Team', np.average)
```

Team	Pressure Drop average
Colts	0.46875
Patriots	1.20227

The average drop for the Patriots was about 1.2 psi compared to about 0.47 psi for the Colts.

The question now is why the Patriots' footballs had a larger drop in pressure, on average, than the Colts footballs. Could it be due to chance?

The Hypotheses

How does chance come in here? Nothing was being selected at random. But we can make a chance model by hypothesizing that the 11 Patriots' drops look like a random sample of 11 out of all the 15 drops, with the Colts' drops being the remaining four. That's a completely specified chance model under which we can simulate data. So it's the **null hypothesis**.

For the alternative, we can take the position that the Patriots' drops are too large, on average, to resemble a random sample drawn from all the drops.

Test Statistic

A natural statistic is the difference between the two average drops, which we will compute as "average drop for Patriots - average drop for Colts". Large values of this statistic will favor the alternative hypothesis.

```
observed_means = football.groupby('Team', np.average).column(1)

observed_difference = observed_means.item(1) -
observed_means.item(0)
observed_difference
```

0.733522727272728

This positive difference reflects the fact that the average drop in pressure of the Patriots' balls was greater than that of the Colts.

Predicting the Statistic Under the Null Hypothesis

If the null hypothesis were true, then the Patriots' drops would be comparable to 11 drops drawn at random without replacement from all 15 drops, and the Colts' drops would be the remaining four. We can simulate this by randomly permuting all 15 drops and assigning each team the appropriate number of permuted values.

```
shuffled_drops =
football.sample(with_replacement=False).column(1)
original_and_shuffled = football.with_column('Shuffled Drop',
shuffled_drops)
original_and_shuffled.show()
```

Team	Pressure Drop	Shuffled Drop
Patriots	0.85	1.175
Patriots	1.475	1.375
Patriots	1.175	0.85
Patriots	1.65	0.65
Patriots	1.225	1.65
Patriots	0.725	1.225
Patriots	0.425	0.725
Patriots	1.175	0.475
Patriots	1.35	1.475
Patriots	1.8	0.275
Patriots	1.375	1.175
Colts	0.475	1.8
Colts	0.475	0.475
Colts	0.275	0.425
Colts	0.65	1.35

How do all the group averages compare?

```
original_and_shuffled.groupby('Team', np.average)
```

Team	Pressure Drop average	Shuffled Drop average
Colts	0.46875	1.0125
Patriots	1.20227	1.00455

The two teams' average drop values are closer when the balls are randomly assigned to the two teams than they were for the balls actually used in the game.

Permutation Test¶

It's time for a step that is now familiar. We will do repeated simulations of the test statistic under the null hypothesis, by repeatedly permuting the footballs and assigning random sets to the two teams.

In the last section we defined a function called `permuted_sample_average_difference` to do this. Here is the definition again. The code is based on the steps we took to compare the averages of the shuffled data.

```

def permuted_sample_average_difference(table, label,
group_label, repetitions):

    tbl = table.select(group_label, label)

    differences = make_array()
    for i in np.arange(repetitions):
        shuffled =tbl.sample(with_replacement =
False).column(1)
        original_and_shuffled =tbl.with_column('Shuffled Data',
shuffled)

        shuffled_means =
original_and_shuffled.group(group_label, np.average).column(2)
        simulated_difference = shuffled_means.item(1) -
shuffled_means.item(0)

        differences = np.append(differences,
simulated_difference)

    return differences

```

```

differences = permuted_sample_average_difference(football,
'Pressure Drop', 'Team', 10000)

```

The array `differences` contains 10,000 values of the test statistic simulated under the null hypothesis.

Conclusion of the Test¶

To calculate the empirical P-value, it's important to recall the alternative hypothesis, which is that the Patriots' drops are too large to be the result of chance variation alone.

The "direction of the alternative" is towards large drops for the Patriots, with correspondingly large values for our test statistic "Patriots' average - Colts' average". So the P-value is the chance (computed under the null hypothesis) of getting a test statistic equal to our observed value of 0.73352272727272805 or *larger*.

```
empirical_P = np.count_nonzero(differences >=
observed_difference) / 10000
empirical_P
```

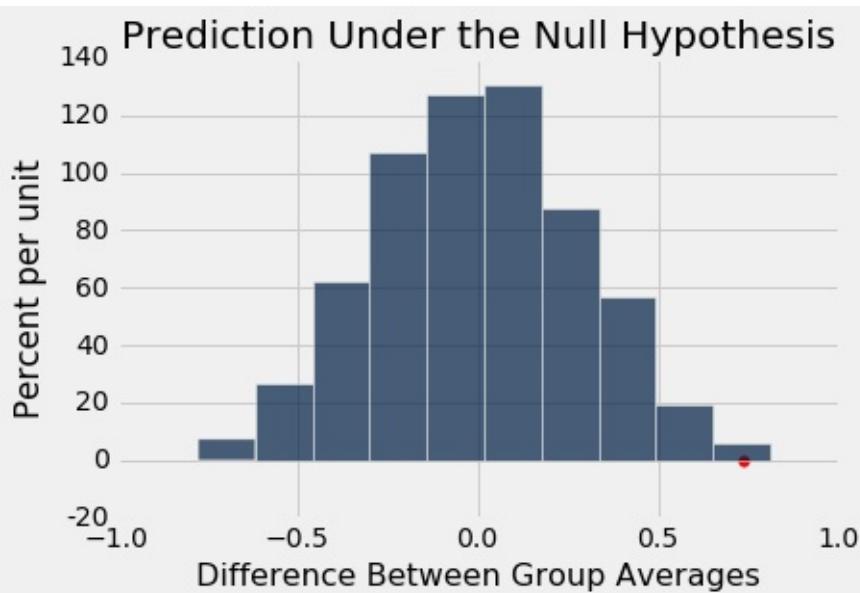
0.0028

That's a pretty small P-value. To visualize this, here is the empirical distribution of the test statistic under the null hypothesis, with the observed statistic marked on the horizontal axis.

```
Table().with_column('Difference Between Group Averages',
differences).hist()
plots.scatter(observed_difference, 0, color='red', s=30)
plots.title('Prediction Under the Null Hypothesis')
print('Observed Difference:', observed_difference)
print('Empirical P-value:', empirical_P)
```

Observed Difference: 0.733522727272728

Empirical P-value: 0.0028



As in previous examples of this test, the bulk of the distribution is centered around 0. Under the null hypothesis, the Patriots' drops are a random sample of all 15 drops, and therefore so are the Colts'. Therefore the two sets of drops should be about equal on average, and therefore their difference should be around 0.

But the observed value of the test statistic is quite far away from the heart of the distribution. By any reasonable cutoff for what is "small", the empirical P-value is small. So we end up rejecting the null hypothesis of randomness, and conclude that the Patriots drops were too large to reflect chance variation alone.

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 close to the center of the interval "0.45 to 1.02 psi" and therefore consistent with the official analysis.

Remember that our test of hypotheses does not establish the reason *why* the difference is not due to chance. Establishing causality is usually more complex than running a test of hypotheses.

But the all-important question in the football world was about causation: the question was whether the excess drop of pressure in the Patriots' footballs was deliberate. If you are curious about the answer given by the investigators, here is the [full report](#).

[Interact](#)

Causality

Our methods for comparing two samples have a powerful use in the analysis of randomized controlled experiments. Since the treatment and control groups are assigned randomly in such experiments, differences in their outcomes can be compared to what would happen just due to chance if the treatment had no effect at all. If the observed differences are more marked than what we would predict as purely due to chance, we will have evidence of *causation*. Because of the unbiased assignment of individuals to the treatment and control groups, differences in the outcomes of the two groups can be ascribed to the treatment.

The key to the analysis of randomized controlled experiments is understanding exactly how chance enters the picture. This helps us set up clear null and alternative hypotheses. Once that's done, we can simply use the methods of the previous sections to complete the analysis.

Let's see how to do this in an example.

Treating Chronic Back Pain: A Randomized Controlled Trial

Low-back pain in adults can be very persistent and hard to treat. Common methods run the gamut from corticosteroids to acupuncture. A [randomized controlled trial \(RCT\)](#) examined the effect of using Botulinum Toxin A as a treatment. Botulinum toxin is a neurotoxic protein that causes the disease botulism; [Wikipedia](#) says that botulinum "is the most acutely lethal toxin known." There are seven types of botulinum toxin. Botulinum Toxin A is one of the types that can cause disease in humans, but it is also used in medicine to treat various diseases involving the muscles. The RCT analyzed by Foster, Clapp, and Jabbari in 2001 examined it as a treatment for low back pain.

Thirty one patients with low-back pain were randomized into treatment and control groups, with 15 in the treatment group and 16 in control. The control group was given normal saline, and the trials were run double-blind so that neither doctors nor patients knew which group they were in.

Eight weeks after the start of the study, nine of the 15 in the treatment group and two of the 16 in the control group had pain relief (according to a precise definition used by the researchers). These data are in the table [bta](#) and appear to show that the treatment has a clear benefit.

```
bta = Table.read_table('bta.csv')
bta.show()
```

Group	Result
Control	1
Control	1
Control	0
Treatment	1
Treatment	0
Treatment	0

Treatment	0

Remember that counting is the same as adding zeros and ones. The sum of 1's in the control group is the number of control group patients who had pain relief. So the *average* of the number of 1's is the *proportion* of control group patients who had pain relief.

```
bta.group('Group', np.mean)
```

Group	Result mean
Control	0.125
Treatment	0.6

In the treatment group, 60% of the patients had pain relief, compared to only 12.5% in the control group. None of the patients suffered any side effects.

So the indications are that botulinum toxin A did better than the saline. But the conclusion isn't yet a slam-dunk. The patients were assigned at random into the two groups, so perhaps the difference could just be due to chance?

To understand what this means, we have to consider the possibility that among the 31 people in the study, some were simply better able to recover than others, even without any help from the treatment. What if an unusually large proportion of them got assigned to the treatment group, just by chance? Then even if the treatment did nothing more than the saline in the control group, the results of the treatment group might look better than those of the control group.

To account for this possibility, let's start by carefully setting up the chance model.

Potential Outcomes

Before the patients are randomized into the two groups, our minds instinctively imagine two possible outcomes for each patient: the outcome that the patient would have if assigned to the treatment group, and the outcome that the same patient would have if assigned to the control group. These are called the two *potential outcomes* of the patient.

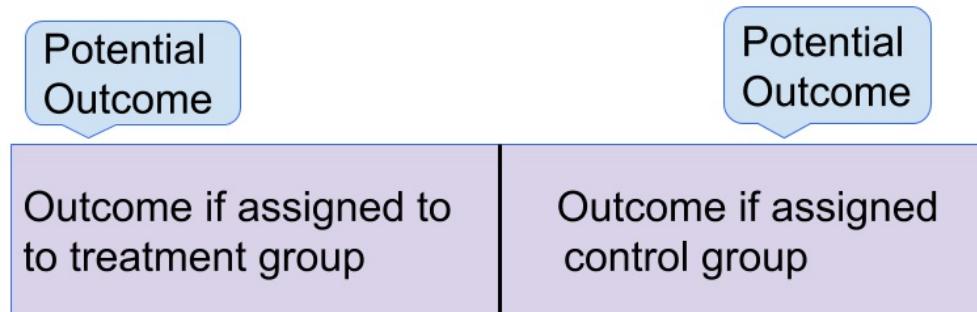
Thus there are 31 potential treatment outcomes and 31 potential control outcomes. The question is about the distributions of these two sets of 31 outcomes each. Are they the same, or are they different?

We can't answer this just yet, because we don't get to see all 31 values in each group. We just get to see a randomly selected 16 of the potential control outcomes, and the treatment outcomes of *the remaining* 15 patients.

Here is a good way to visualize the setting. Each patient has a two-sided ticket:

Before the Randomization

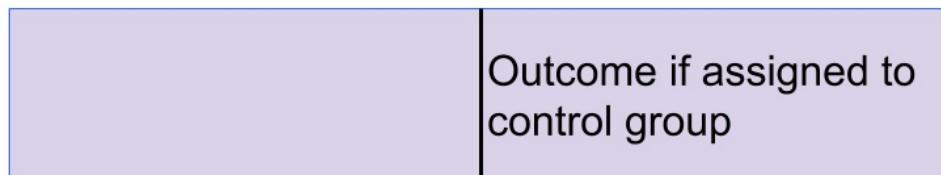
- In the population there is one imaginary ticket for each of the 31 participants in the experiment.
- Each participant's ticket looks like this:



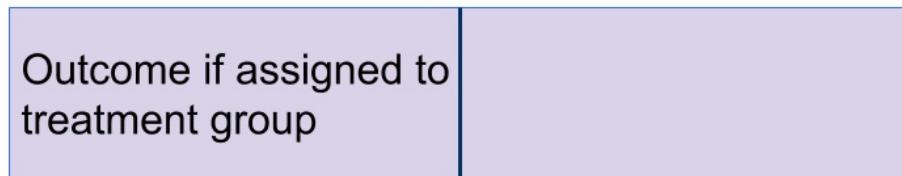
After the randomization, we get to see the right half of a randomly selected set of tickets, and the left half of the remaining group.

The Data

16 randomly picked tickets show:



The remaining 15 tickets show:



The table `observed_outcomes` collects the information about every patient's potential outcomes, leaving the unobserved half of each "ticket" blank. (It's just another way of thinking about the `bta` table, carrying the same information.)

```
observed_outcomes = Table.read_table("observed_outcomes.csv")
observed_outcomes.show()
```

Group	Outcome if assigned treatment	Outcome if assigned control
Control	Unknown	1
Control	Unknown	1
Control	Unknown	0
Treatment	1	Unknown

Treatment	0	Unknown

The Hypotheses

The question is whether the treatment does anything. In terms of the table `observed_outcomes`, the question is whether the distribution of the 31 "treatment" values in Column 1 (including the unknown ones) is different from the distribution of the 31 "control" values in Column 2 (again including the unknown ones).

Null Hypothesis: The distribution of all 31 potential "treatment" outcomes is the same as that of all 31 potential "control" outcomes. Botulinum toxin A does nothing different from saline; the difference in the two samples is just due to chance.

Alternative Hypothesis: The distribution of 31 potential "treatment" outcomes is different from that of the 31 control outcomes. The treatment does something different from the control.

There are 31 observed outcomes jointly in the two groups. If the null hypothesis were true, it wouldn't matter which of those 31 outcomes were labeled "treatment" and which "control." Any random subset of 16 out of the 31 values could be called "control" and the remaining 15 "treatment".

We can simulate this. We can randomly permute the 31 values, split them into two groups of 16 and 15, and see how different the distributions in the two groups are. Since the data are zeros and ones, we can just see how different the two proportions are.

That's exactly what we did for A/B testing in the previous section. Sample A is now the control group and Sample B the treatment group. We will carry out the test below showing the details of all the steps. You should confirm that they are the same as the steps carried out for A/B testing.

The Test Statistic

If the two group proportions are very different from each other, we will lean towards the alternative hypothesis that the two underlying distributions are different. So our test statistic will be the distance between the two group proportions, that is, the absolute value of the

difference between them.

Large values of the test statistic will favor the alternative hypothesis over the null.

Since the two group proportions were 0.6 and 0.125, the observed value of the test statistic is $|0.6 - 0.125| = 0.475$.

```
bta.group('Group', np.average)
```

Group	Result average
Control	0.125
Treatment	0.6

```
observed_proportions = bta.group('Group', np.average).column(1)
observed_distance = abs(observed_proportions.item(0) -
observed_proportions.item(1))
observed_distance
```

```
0.475
```

Predicting the Statistic Under the Null Hypothesis

We can simulate results under the null hypothesis, to see how our test statistic should come out if the null hypothesis is true.

Generating One Value of the Statistic

The simulation follows exactly the same process we used in the previous section. We start by randomly permuting the `results` column and assigning "control" and "treatment" labels to the permuted results.

```
shuffled_results = bta.sample(with_replacement=False).column(1)
```

```
bta_with_shuffled_results = bta.with_column('Shuffled Results',
shuffled_results)
bta_with_shuffled_results.show()
```

Group	Result	Shuffled Results
Control	1	0
Control	1	0
Control	0	0
Control	0	0
Control	0	1
Control	0	0
Control	0	0
Control	0	1
Control	0	0
Control	0	0
Control	0	1
Control	0	0
Control	0	0
Control	0	0
Treatment	1	1
Treatment	1	0
Treatment	1	1
Treatment	1	1
Treatment	1	0
Treatment	1	1
Treatment	0	0
Treatment	0	1
Treatment	0	0
Treatment	0	0
Treatment	0	1
Treatment	0	1

We then get the group means of the shuffled results:

```
bta_with_shuffled_results.groupby('Group', np.average)
```

Group	Result average	Shuffled Results average
Control	0.125	0.25
Treatment	0.6	0.466667

The group proportions in the "shuffled" column look quite different from those in the study's results.

We can use the simulated proportions to calculate the simulated value of the test statistic. By doing this repeatedly, we will get a sense of how the statistic varies under the null hypothesis.

```
proportions = bta_with_shuffled_results.groupby('Group',
np.average).column(2)
simulated_distance = abs(proportions.item(0) -
proportions.item(1))
simulated_distance
```

```
0.2166666666666667
```

Permutation Test

You can see that we are doing exactly what we did in our previous examples of the permutation test. Here is the function we defined earlier to generate the simulated differences under the null hypothesis. It simply collects the code above and puts it in the body of a `for` loop.

```

def permuted_sample_average_difference(table, label,
group_label, repetitions):

    tbl = table.select(group_label, label)

    differences = make_array()
    for i in np.arange(repetitions):
        shuffled =tbl.sample(with_replacement =
False).column(1)
        original_and_shuffled =tbl.with_column('Shuffled Data',
shuffled)

        shuffled_means =
original_and_shuffled.group(group_label, np.average).column(2)
        simulated_difference = shuffled_means.item(1) -
shuffled_means.item(0)

        differences = np.append(differences,
simulated_difference)

    return differences

```

We will call this function to generate an array of differences between proportions in randomly selected "control" and "treatment" groups.

```

differences = permuted_sample_average_difference(bta, 'Result',
'Group', 20000)

```

Our statistic is the distance between the two proportions, that is, the absolute value of the difference.

```

distances = np.abs(differences)

```

Conclusion of the Test

The array `distances` contains 20,000 values of our test statistic simulated under the null hypothesis.

To find the P-value of the test, remember that large values of the test statistic favor the alternative hypothesis. So the empirical P-value is the proportion of simulated statistics that were equal to or larger than the observed statistic.

```
empirical_P = np.count_nonzero(distances >= observed_distance) / repetitions
empirical_P
```

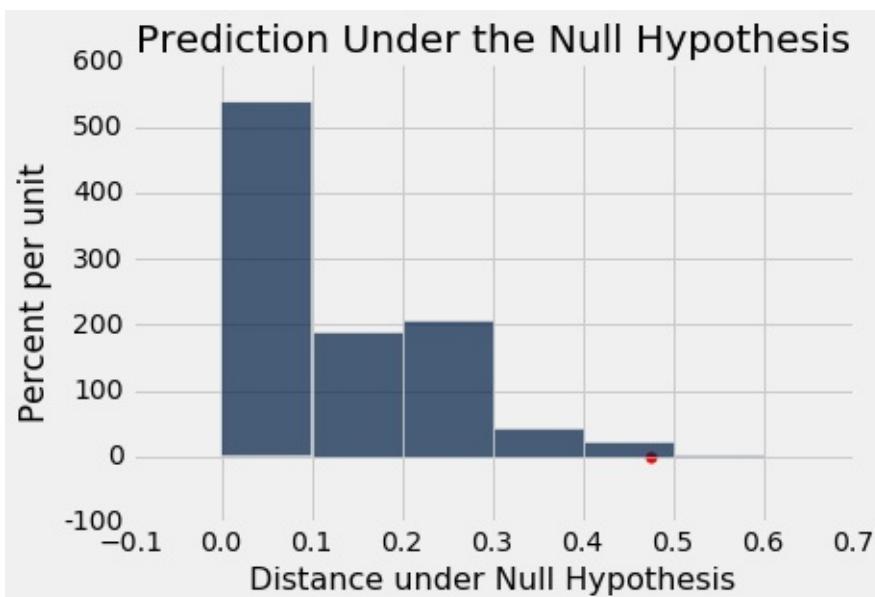
0.01015

This is a small P-value. The observed statistic, shown as the red dot below, is in the tail of the empirical histogram of the test statistic generated under the null hypothesis.

The result is statistically significant. The test favors the alternative hypothesis more than the null. The evidence supports the hypothesis that the treatment is doing something.

```
Table().with_column('Distance under Null Hypothesis',
distances).hist(bins = np.arange(0, 0.7, 0.1))
plots.scatter(observed_distance, 0, color='red', s=30)
plots.title('Prediction Under the Null Hypothesis')
print('Observed Distance', observed_distance)
print('Empirical P-value:', round(empirical_P, 4) *100, '%')
```

Observed Distance 0.475
 Empirical P-value: 1.01 %



The study reports a P-value of 0.009, or 0.9%, which is not far from our empirical value.

Causality

Because the trials were randomized, the test is evidence that the treatment *causes* the difference. The random assignment of patients to the two groups ensures that there is no confounding variable that could affect the conclusion of causality.

If the treatment had not been randomly assigned, our test would still point toward an *association* between the treatment and back pain outcomes among our 31 patients. But beware: without randomization, this association would not imply that the treatment caused a change in back pain outcomes. For example, if the patients themselves had chosen whether to administer the treatment, perhaps the patients experiencing more pain would be more likely to choose the treatment *and* more likely to experience some reduction in pain even without medication. Pre-existing pain would then be a *confounding factor* in the analysis.

A Meta-Analysis

While the RCT does provide evidence that the botulinum toxin A treatment helped patients, a study of 31 patients isn't enough to establish the effectiveness of a medical treatment. This is not just because of the small sample size. Our results in this section are valid for the 31 patients in the study, but we are really interested in the population of *all possible patients*. If the 31 patients were a random sample from *that* larger population, our confidence interval would be valid for that population. But they were not a random sample.

In 2011, a group of researchers performed a [meta-analysis](#) of the studies on the treatment. That is, they identified all the available studies of such treatments for low-back pain and summarized the collated results.

There were several studies but not many could be included in a scientifically sound manner: "We excluded evidence from nineteen studies due to non-randomisation, incomplete or unpublished data." Only three randomized controlled trials remained, one of which is the one we have studied in this section. The meta-analysis gave it the highest assessment among all the studies (LBP stands for low-back pain): "We identified three studies that investigated the merits of BoNT for LBP, but only one had a low risk of bias and evaluated patients with non-specific LBP (N = 31)."

Putting it all together, the meta-analysis concluded, "There is low quality evidence that BoNT injections improved pain, function, or both better than saline injections and very low quality evidence that they were better than acupuncture or steroid injections. ... Further research is very likely to have an important impact on the estimate of effect and our confidence in it.

Future trials should standardize patient populations, treatment protocols and comparison groups, enlist more participants and include long-term outcomes, cost-benefit analysis and clinical relevance of findings."

It takes a lot of careful work to establish that a medical treatment has a beneficial effect. Knowing how to analyze randomized controlled trials is a crucial part of this work. Now that you know how to do that, you are well positioned to help medical and other professions establish cause-and-effect relations.

[Interact](#)

Estimation

In the previous chapter we began to develop ways of inferential thinking. In particular, we learned how to use data to decide between two hypotheses about the world. But often we just want to know how big something is.

For example, in an earlier chapter we investigated how many warplanes the enemy might have. In an election year, we might want to know what percent of voters favor a particular candidate. To assess the current economy, we might be interested in the median annual income of households in the United States.

In this chapter, we will develop a way to *estimate* an unknown *parameter*. Remember that a parameter is a numerical value associated with a population.

To figure out the value of a parameter, we need data. If we have the relevant data for the entire population, we can simply calculate the parameter.

But if the population is very large – for example, if it consists of all the households in the United States – then it might be too expensive and time-consuming to gather data from the entire population. In such situations, data scientists rely on sampling at random from the population.

This leads to a question of inference: How to make justifiable conclusions about the unknown parameter, based on the data in the random sample? We will answer this question by using inferential thinking.

A statistic based on a random sample can be a reasonable estimate of an unknown parameter in the population. For example, you might want to use the median annual income of sampled households as an estimate of the median annual income of all households in the U.S.

But the value of any statistic depends on the sample, and the sample is based on random draws. So every time data scientists come up with an estimate based on a random sample, they are faced with a question:

"How different could this estimate have been, if the sample had come out differently?"

In this chapter you will learn one way of answering this question. The answer will give you the tools to estimate a numerical parameter and quantify the amount of error in your estimate.

We will start with a preliminary about percentiles. The most famous percentile is the median, often used in summaries of income data. Other percentiles will be important in the method of estimation that we are about to develop. So we will start by defining percentiles carefully.

[Interact](#)

Percentiles

Numerical data can be sorted in increasing or decreasing order. Thus the values of a numerical data set have a *rank order*. A percentile is the value at a particular rank.

For example, if your score on a test is on the 95th percentile, a common interpretation is that only 5% of the scores were higher than yours. The median is the 50th percentile; it is commonly assumed that 50% the values in a data set are above the median.

But some care is required in giving percentiles a precise definition that works for all ranks and all lists. To see why, consider an extreme example where all the students in a class score 75 on a test. Then 75 is a natural candidate for the median, but it's not true that 50% of the scores are above 75. Also, 75 is an equally natural candidate for the 95th percentile or the 25th or any other percentile. Ties – that is, equal data values – have to be taken into account when defining percentiles.

You also have to be careful about exactly how far up the list to go when the relevant index isn't clear. For example, what should be the 87th percentile of a collection of 10 values? The 8th value of the sorted collection, or the 9th, or somewhere in between?

In this section, we will give a definition that works consistently for all ranks and all lists.

A Numerical Example

Before giving a general definition of all percentiles, we will define the 80th percentile of a collection of values to be the smallest value in the collection that is at least as large as 80% of all 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 = make_array(12, 17, 6, 9, 7)
```

The 80th percentile is the smallest value that is at least as large as 80% of the elements of `sizes`, that is, four-fifths of the five elements. That's 12:

```
np.sort(sizes)
```

```
array([ 6,  7,  9, 12, 17])
```

The 80th percentile is a value on the list, namely 12. You can see that 80% of the values are less than or equal to it, and that it is the smallest value on the list for which this is true.

Analogously, the 70th percentile is the smallest value in the collection that is at least as large as 70% of the elements of `sizes`. Now 70% of 5 elements is "3.5 elements", so the 70th percentile is the 4th element on the list. That's 12, the same as the 80th percentile for these data.

The percentile function¶

The `percentile` function takes two arguments: a rank between 0 and 100, and a array. It returns the corresponding percentile of the array.

```
percentile(70, sizes)
```

```
12
```

The General Definition¶

Let p be a number between 0 and 100. The p th percentile of a collection is the smallest value in the collection that is at least as large as $p\%$ of all the values.

By this definition, any percentile between 0 and 100 can be computed for any collection of values, and it is always an element of the collection.

In practical terms, suppose there are n elements in the collection. To find the p th percentile:

- Sort the collection in increasing order.
- Find $p\%$ of n : $(p/100) \times n$. Call that k .
- If k is an integer, take the k th element of the sorted collection.
- If k is not an integer, round it up to the next integer, and take that element of the sorted collection.

Example¶

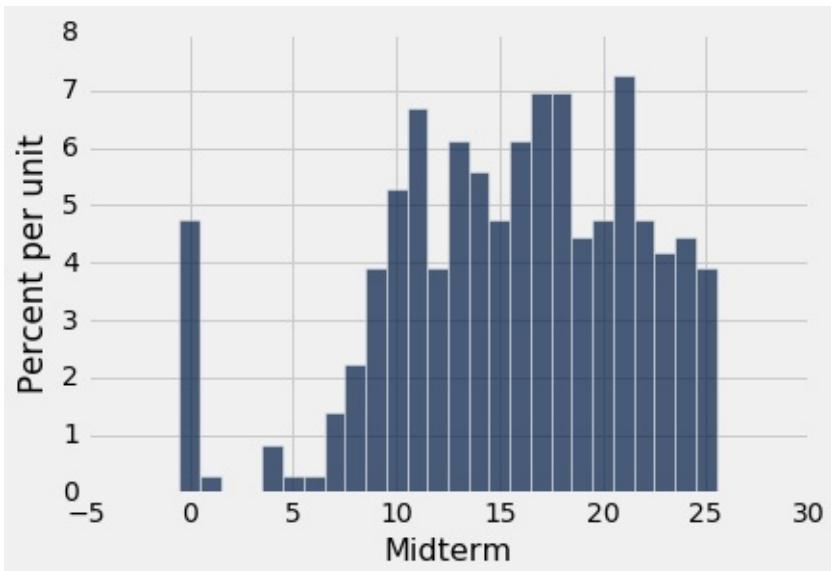
The table `scores_and_sections` contains one row for each student in a class of 359 students. The columns are the student's discussion section and midterm score.

```
scores_and_sections = Table.read_table('scores_by_section.csv')
scores_and_sections
```

Section	Midterm
1	22
2	12
2	23
2	14
1	20
3	25
4	19
1	24
5	8
6	14

... (349 rows omitted)

```
scores_and_sections.select('Midterm').hist(bins=np.arange(-0.5, 25.6, 1))
```



What was the 85th percentile of the scores? To use the `percentile` function, create an array `scores` containing the midterm scores, and find the 85th percentile:

```
scores = scores_and_sections.column(1)
```

```
percentile(85, scores)
```

22

According to the percentile function, the 85th percentile was 22. To check that this is consistent with our new definition, let's apply the definition directly.

First, put the scores in increasing order:

```
sorted_scores = np.sort(scores_and_sections.column(1))
```

There are 359 scores in the array. So next, find 85% of 359, which is 305.15.

```
0.85 * 359
```

```
305.15
```

That's not an integer. By our definition, the median is the 306th element of `sorted_scores`, which, by Python's indexing convention, is item 305 of the array.

```
# The 306th element of the sorted array  
  
sorted_scores.item(305)
```

```
22
```

That's the same as the answer we got by using `percentile`. In future, we will just use `percentile`.

Quartiles

The *first quartile* of a numerical collection is the 25th percentile. The terminology arises from *the first quarter*. The second quartile is the median, and the third quartile is the 75th percentile.

For our `scores` data, those values are:

```
percentile(25, scores)
```

11

```
percentile(50, scores)
```

16

```
percentile(75, scores)
```

20

Distributions of scores are sometimes summarized by the "middle 50%" interval, between the first and third quartiles.

[Interact](#)

The Bootstrap

A data scientist is using the data in a random sample to estimate an unknown parameter. She uses the sample to calculate the value of a statistic that she will use as her estimate.

Once she has calculated the observed value of her statistic, she could just present it as her estimate and go on her merry way. But she's a data scientist. She knows that her random sample is just one of numerous possible random samples, and thus her estimate is just one of numerous plausible estimates.

By how much could those estimates vary? To answer this, it appears as though she needs to draw another sample from the population, and compute a new estimate based on the new sample. But she doesn't have the resources to go back to the population and draw another sample.

It looks as though the data scientist is stuck.

Fortunately, a brilliant idea called *the bootstrap* can help her out. Since it is not feasible to generate new samples from the population, the bootstrap generates new random samples by a method called *resampling*: the new samples are drawn at random *from the original sample*.

In this section, we will see how and why the bootstrap works. In the rest of the chapter, we will use the bootstrap for inference.

Employee Compensation in the City of San Francisco

[SF OpenData](#) is a website where the City and County of San Francisco make some of their data publicly available. One of the data sets contains compensation data for employees of the City. These include medical professionals at City-run hospitals, police officers, fire fighters, transportation workers, elected officials, and all other employees of the City.

Compensation data for the calendar year 2015 are in the table `sf2015`.

```
sf2015 = Table.read_table('san_francisco_2015.csv')
```

```
sf2015
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	2	Public Works, Transportation & Commerce	WTR	PUC Water Department
Calendar	2015	2	Public Works, Transportation & Commerce	DPW	General Services Agency - Public Works
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency
Calendar	2015	1	Public Protection	POL	Police
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	2	Public Works, Transportation & Commerce	MTA	Municipal Transportation Agency
Calendar	2015	6	General Administration & Finance	CAT	City Attorney
Calendar	2015	3	Human Welfare & Neighborhood Development	DSS	Human Services

... (42979 rows omitted)

There is one row for each of 42,979 employees. There are numerous columns containing information about City departmental affiliation and details of the different parts of the employee's compensation package. Here is the row corresponding to the late Edward Lee, the Mayor at that time.

```
sf2015.where('Job', are.equal_to('Mayor'))
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	6	General Administration & Finance	MYR	Mayor

We are going to study the final column, `Total Compensation`. That's the employee's salary plus the City's contribution towards his/her retirement and benefit plans.

Financial packages in a calendar year can sometimes be hard to understand as they depend on the date of hire, whether the employee is changing jobs within the City, and so on. For example, the lowest values in the `Total Compensation` column look a little strange.

```
sf2015.sort('Total Compensation')
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	1	Public Protection	FIR	Fire Department
Calendar	2015	4	Community Health	DPH	Public Health
Calendar	2015	1	Public Protection	JUV	Juvenile Probation
Calendar	2015	6	General Administration & Finance	CPC	City Planning
Calendar	2015	6	General Administration & Finance	CPC	City Planning
Calendar	2015	2	Public Works, Transportation & Commerce	PUC	PUC Public Utilities Commission
Calendar	2015	1	Public Protection	JUV	Juvenile Probation
Calendar	2015	1	Public Protection	ECD	Department of Emergency Management
Calendar	2015	7	General City Responsibilities	UNA	General Fund Unallocated
Calendar	2015	4	Community Health	DPH	Public Health

... (42979 rows omitted)

For clarity of comparison, we will focus our attention on those who had at least the equivalent of a half-time job for the whole year. At a minimum wage of about \$10 per hour, and 20 hours per week for 52 weeks, that's a salary of about \$10,000.

```
sf2015 = sf2015.where('Salaries', are.above(10000))
```

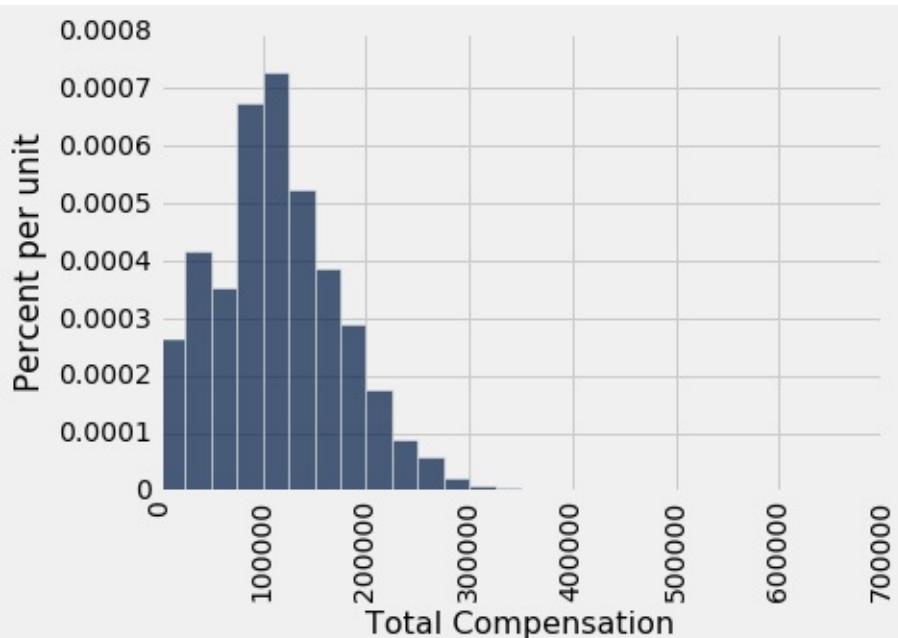
```
sf2015.num_rows
```

```
36569
```

Population and Parameter

Let this table of just over 36,500 rows be our population. Here is a histogram of the total compensations.

```
sf_bins = np.arange(0, 700000, 25000)
sf2015.select('Total Compensation').hist(bins=sf_bins)
```



While most of the values are below \$300,000, a few are quite a bit higher. For example, the total compensation of the Chief Investment Officer was almost \$650,000. That is why the horizontal axis stretches to \$700,000.

```
sf2015.sort('Total Compensation', descending=True).show(2)
```

Year Type	Year	Organization Group Code	Organization Group	Department Code	Department
Calendar	2015	6	General Administration & Finance	RET	Retirement System
Calendar	2015	6	General Administration & Finance	ADM	General Services Agency - City Admin

... (36567 rows omitted)

Now let the parameter be the median of the total compensations.

Since we have the luxury of having all of the data from the population, we can simply calculate the parameter:

```
pop_median = percentile(50, sf2015.column('Total Compensation'))
pop_median
```

```
110305.78999999999
```

The median total compensation of all employees was just over \$110,300.

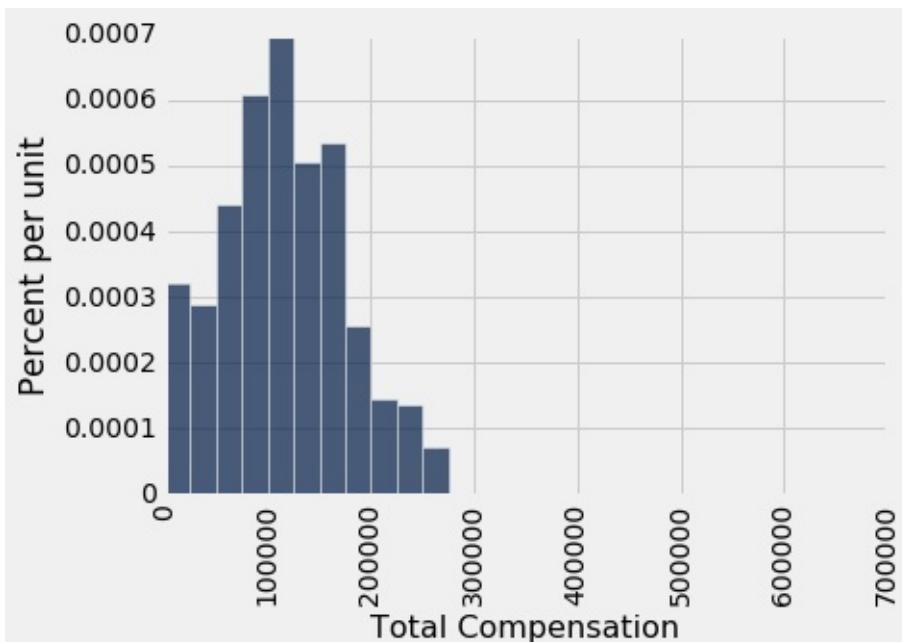
From a practical perspective, there is no reason for us to draw a sample to estimate this parameter since we simply know its value. But in this section we are going to pretend we don't know the value, and see how well we can estimate it based on a random sample.

In later sections, we will come down to earth and work in situations where the parameter is unknown. For now, we are all-knowing.

A Random Sample and an Estimate

Let us draw a sample of 500 employees at random without replacement, and let the median total compensation of the sampled employees serve as our estimate of the parameter.

```
our_sample = sf2015.sample(500, with_replacement=False)
our_sample.select('Total Compensation').hist(bins=sf_bins)
```



```
est_median = percentile(50, our_sample.column('Total Compensation'))
est_median
```

113598.99000000001

The sample size is large. By the law of averages, the distribution of the sample resembles that of the population, and consequently the sample median is not very far from the population median (though of course it is not exactly the same).

So now we have one estimate of the parameter. But had the sample come out differently, the estimate would have had a different value. We would like to be able to quantify the amount by which the estimate could vary across samples. That measure of variability will help us measure how accurately we can estimate the parameter.

To see how different the estimate would be if the sample had come out differently, we could just draw another sample from the population, but that would be cheating. We are trying to mimic real life, in which we won't have all the population data at hand.

Somehow, we have to get another random sample without sampling from the population.

The Bootstrap: Resampling from the Sample

What we do have is a large random sample from the population. As we know, a large random sample is likely to resemble the population from which it is drawn. This observation allows data scientists to *lift themselves up by their own bootstraps*: the sampling procedure

can be replicated by *sampling from the sample*.

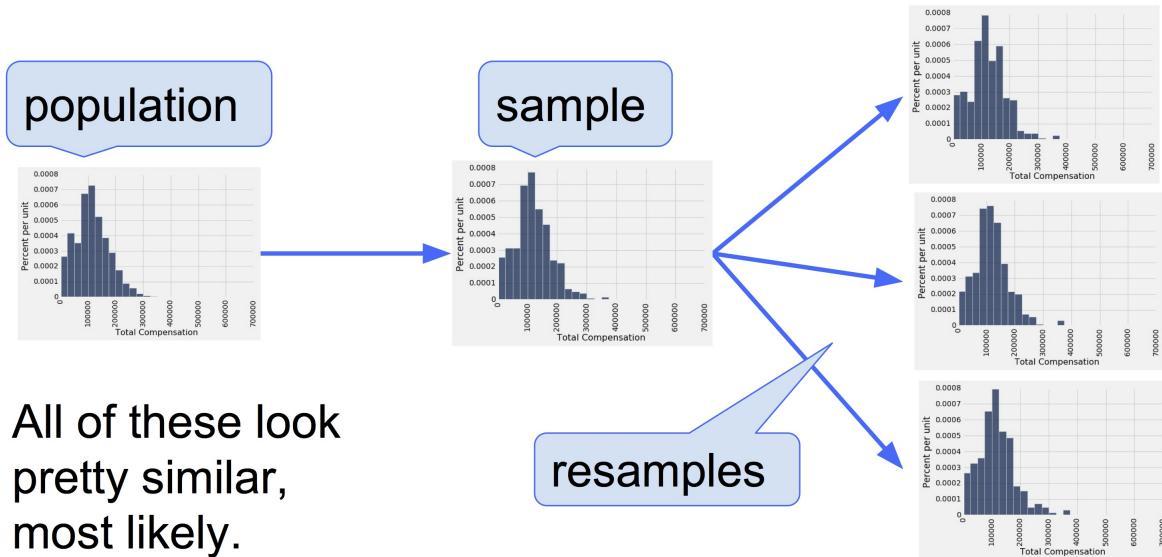
Here are the steps of *the bootstrap method* for generating another random sample that resembles the population:

- Treat the original sample as if it were the population.
- Draw from the sample, at random with replacement, the same number of times as the original sample size.

It is important to resample the same number of times as the original sample size. The reason is that the variability of an estimate depends on the size of the sample. Since our original sample consisted of 500 employees, our sample median was based on 500 values. To see how different the sample could have been, we have to compare it to the median of other samples of size 500.

If we drew 500 times at random *without* replacement from our sample of size 500, we would just get the same sample back. By drawing *with* replacement, we create the possibility for the new samples to be different from the original, because some employees might be drawn more than once and others not at all.

Why is this a good idea? By the law of averages, the distribution of the original sample is likely to resemble the population, and the distributions of all the "resamples" are likely to resemble the original sample. So the distributions of all the resamples are likely to resemble the population as well.



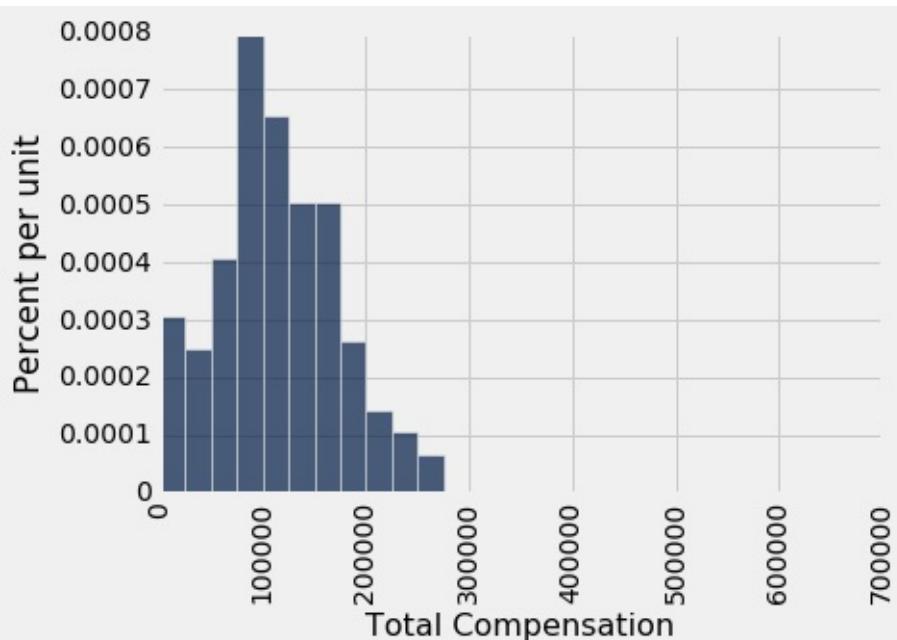
A Resampled Median

Recall that when the `sample` method is used without specifying a sample size, by default the sample size equals the number of rows of the table from which the sample is drawn. That's perfect for the bootstrap! Here is one new sample drawn from the original sample,

and the corresponding sample median.

```
resample_1 = our_sample.sample()
```

```
resample_1.select('Total Compensation').hist(bins=sf_bins)
```



```
resampled_median_1 = percentile(50, resample_1.column('Total Compensation'))
resampled_median_1
```

```
110001.16
```

By resampling, we have another estimate of the population median. By resampling again and again, we will get many such estimates, and hence an empirical distribution of the estimates.

```
resample_2 = our_sample.sample()
resampled_median_2 = percentile(50, resample_2.column('Total Compensation'))
resampled_median_2
```

```
110261.39999999999
```

Bootstrap Empirical Distribution of the Sample Median

Let us define a function `bootstrap_median` that takes our original sample, the label of the column containing the variable, and the number of bootstrap samples we want to take, and returns an array of the corresponding resampled medians.

Each time we resample and find the median, we *replicate* the bootstrap process. So the number of bootstrap samples will be called the number of replications.

```
def bootstrap_median(original_sample, label, replications):
    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """
    just_one_column = original_sample.select(label)
    medians = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_median = percentile(50,
                                      bootstrap_sample.column(0))
        medians = np.append(medians, resampled_median)

    return medians
```

We now replicate the bootstrap process 5,000 times. The array `bstrap_medians` contains the medians of all 5,000 bootstrap samples. Notice that the code takes longer to run than our previous code. It has a lot of resampling to do!

```
bstrap_medians = bootstrap_median(our_sample, 'Total
Compensation', 5000)
```

Here is the histogram of the 5000 medians. The red dot is the population parameter: it is the median of the entire population, which we happen to know but did not use in the bootstrap process.

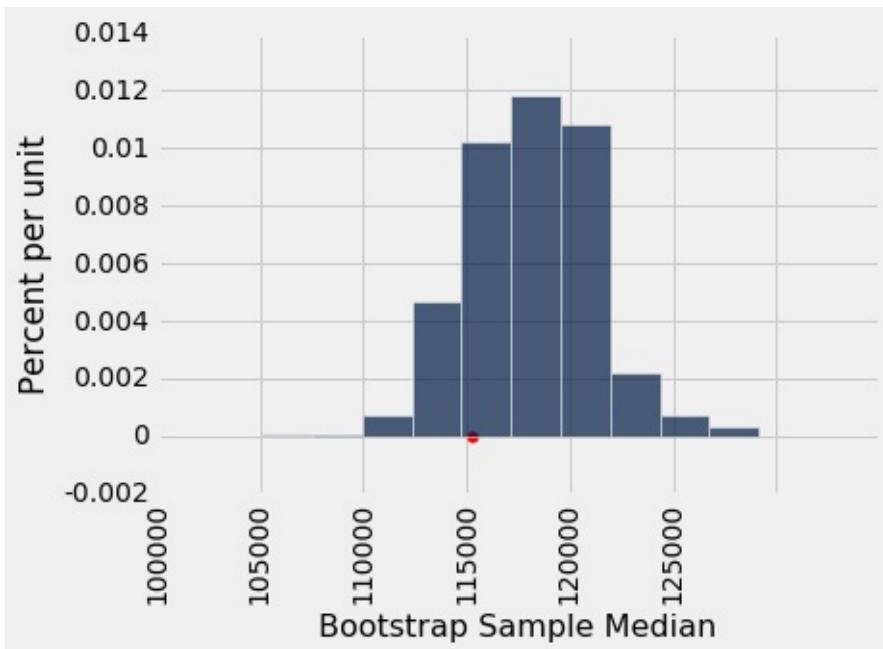
```

resampled_medians = Table().with_column('Bootstrap Sample Median', bstrap_medians)

#median_bins=np.arange(100000, 130000, 2500)
#resampled_medians.hist(bins = median_bins)
resampled_medians.hist()

plots.scatter(pop_median, 0, color='red', s=30);

```



It is important to remember that the red dot is fixed: it is \$110,305.79, the population median. The empirical histogram is the result of random draws, and will be situated randomly relative to the red dot.

Remember also that the point of all these computations is to estimate the population median, which is the red dot. Our estimates are all the randomly generated sampled medians whose histogram you see above. We want those estimates to contain the parameter – if they don't, then they are off.

Do the Estimates Capture the Parameter?

How often does the empirical histogram of the resampled medians sit firmly over the red dot, and not just brush the dot with its tails? To answer this, we must define "sit firmly". Let's take that to mean "the middle 95% of the resampled medians contains the red dot".

Here are the two ends of the "middle 95%" interval of resampled medians:

```
left = percentile(2.5, bstrap_medians)
left
```

```
107652.71000000001
```

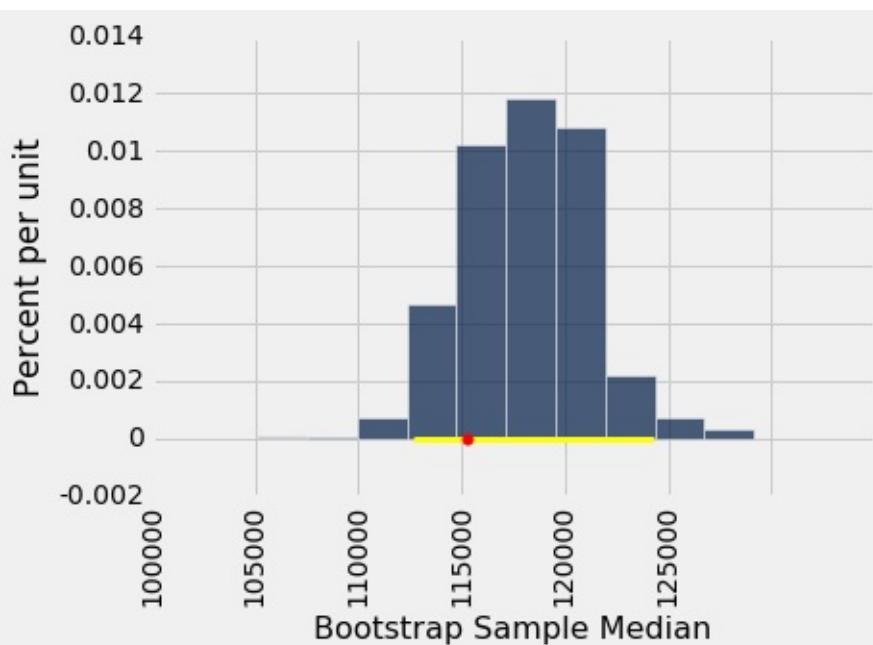
```
right = percentile(97.5, bstrap_medians)
right
```

```
119256.73
```

The population median of \$110,305 is between these two numbers. The interval and the population median are shown on the histogram below.

```
#median_bins=np.arange(100000, 130000, 2500)
#resampled_medians.hist(bins = median_bins)
resampled_medians.hist()

plots.plot(make_array(left, right), make_array(0, 0),
color='yellow', lw=3, zorder=1)
plots.scatter(pop_median, 0, color='red', s=30, zorder=2);
```



The "middle 95%" interval of estimates captured the parameter in our example. But was that a fluke?

To see how frequently the interval contains the parameter, we have to run the entire process over and over again. Specifically, we will repeat the following process 100 times:

- Draw an original sample of size 500 from the population.
- Carry out 5,000 replications of the bootstrap process and generate the "middle 95%" interval of resampled medians.

We will end up with 100 intervals, and count how many of them contain the population median.

Spoiler alert: The statistical theory of the bootstrap says that the number should be around 95. It may be in the low 90s or high 90s, but not much farther off 95 than that.

```
# THE BIG SIMULATION: This one takes several minutes.

# Generate 100 intervals, in the table intervals

left_ends = make_array()
right_ends = make_array()

total_comps = sf2015.select('Total Compensation')

for i in np.arange(100):
    first_sample = total_comps.sample(500,
with_replacement=False)
    medians = bootstrap_median(first_sample, 'Total
Compensation', 5000)
    left_ends = np.append(left_ends, percentile(2.5, medians))
    right_ends = np.append(right_ends, percentile(97.5,
medians))

intervals = Table().with_columns(
    'Left', left_ends,
    'Right', right_ends
)
```

For each of the 100 replications, we get one interval of estimates of the median.

```
intervals
```

Left	Right
100547	115112
98788.4	112129
107981	121218
100965	114796
102596	112056
105386	113909
105225	116918
102844	116712
106584	118054
108451	118421

... (90 rows omitted)

The good intervals are those that contain the parameter we are trying to estimate. Typically the parameter is unknown, but in this section we happen to know what the parameter is.

```
pop_median
```

```
110305.78999999999
```

How many of the 100 intervals contain the population median? That's the number of intervals where the left end is below the population median and the right end is above.

```
intervals.where('Left', are.below(pop_median)).where('Right',
are.above(pop_median)).num_rows
```

```
95
```

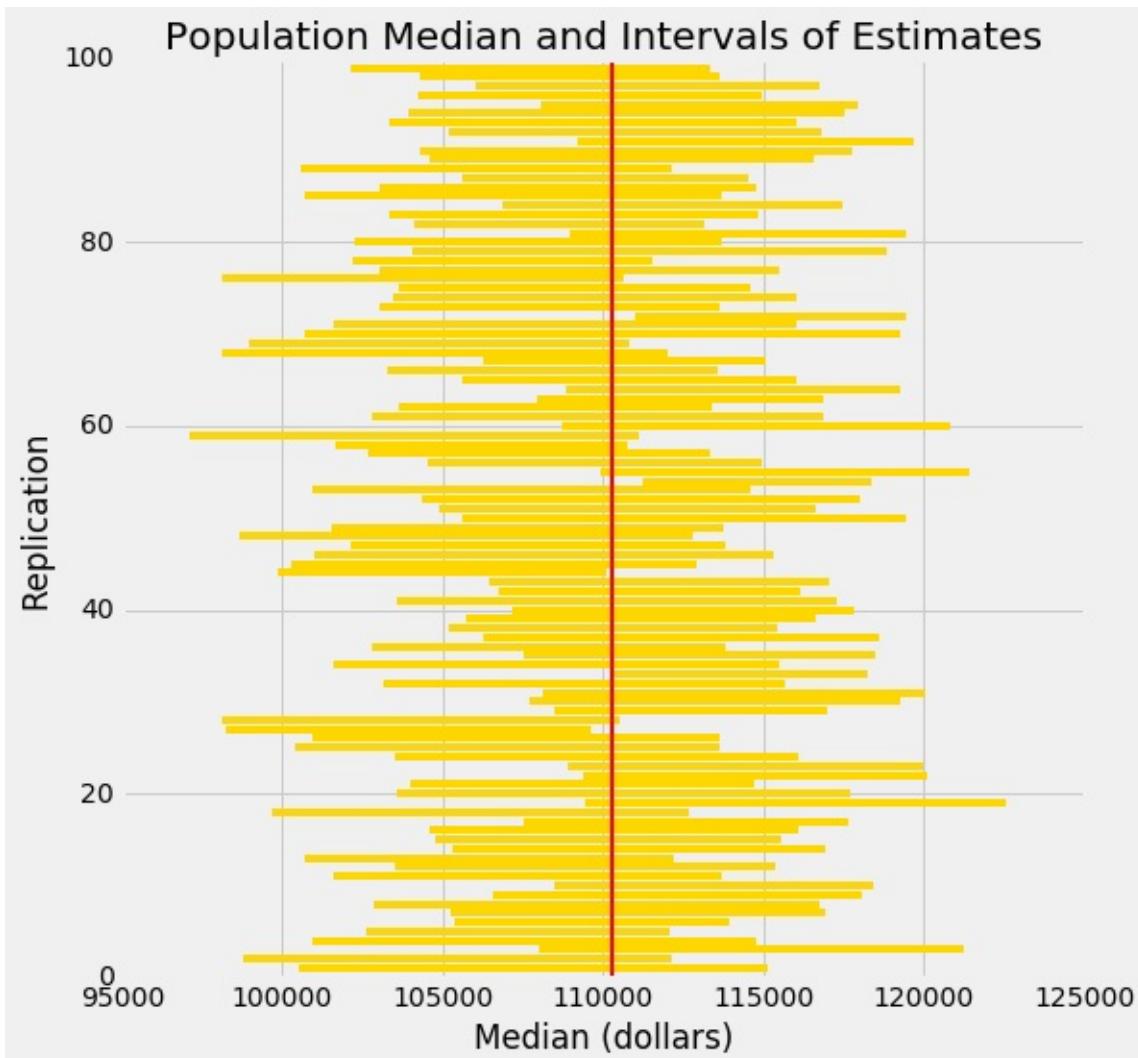
It takes a few minutes to construct all the intervals, but try it again if you have the patience. Most likely, about 95 of the 100 intervals will be good ones: they will contain the parameter.

It's hard to show you all the intervals on the horizontal axis as they have large overlaps – after all, they are all trying to estimate the same parameter. The graphic below shows each interval on the same axes by stacking them vertically. The vertical axis is simply the number of the replication from which the interval was generated.

The red line is where the parameter is. Good intervals cover the parameter; there are about 95 of these, typically.

If an interval doesn't cover the parameter, it's a dud. The duds are the ones where you can see "daylight" around the red line. There are very few of them – about 5, typically – but they do happen.

Any method based on sampling has the possibility of being off. The beauty of methods based on random sampling is that we can quantify how often they are likely to be off.



To summarize what the simulation shows, suppose you are estimating the population median by the following process:

- Draw a large random sample from the population.
- Bootstrap your random sample and get an estimate from the new random sample.
- Repeat the above step thousands of times, and get thousands of estimates.
- Pick off the "middle 95%" interval of all the estimates.

That gives you one interval of estimates. Now if you repeat **the entire process** 100 times, ending up with 100 intervals, then about 95 of those 100 intervals will contain the population parameter.

In other words, this process of estimation captures the parameter about 95% of the time.

You can replace 95% by a different value, as long as it's not 100. Suppose you replace 95% by 80% and keep the sample size fixed at 500. Then your intervals of estimates will be shorter than those we simulated here, because the "middle 80%" is a smaller range than the "middle 95%". Only about 80% of your intervals will contain the parameter.

[Interact](#)

Confidence Intervals

We have developed a method for estimating a parameter by using random sampling and the bootstrap. Our method produces an interval of estimates, to account for chance variability in the random sample. By providing an interval of estimates instead of just one estimate, we give ourselves some wiggle room.

In the previous example we saw that our process of estimation produced a good interval about 95% of the time, a "good" interval being one that contains the parameter. We say that we are *95% confident* that the process results in a good interval. Our interval of estimates is called a *95% confidence interval* for the parameter, and 95% is called the *confidence level* of the interval.

The situation in the previous example was a bit unusual. Because we happened to know value of the parameter, we were able to check whether an interval was good or a dud, and this in turn helped us to see that our process of estimation captured the parameter about 95 out of every 100 times we used it.

But usually, data scientists don't know the value of the parameter. That is the reason they want to estimate it in the first place. In such situations, they provide an interval of estimates for the unknown parameter by using methods like the one we have developed. Because of statistical theory and demonstrations like the one we have seen, data scientists can be confident that their process of generating the interval results in a good interval a known percent of the time.

Confidence Interval for a Population Median: Bootstrap Percentile Method

We will now use the bootstrap method to estimate an unknown population median. The data come from a sample of newborns in a large hospital system; we will treat it as if it were a simple random sample though the sampling was done in multiple stages. [Stat Labs](#) by Deborah Nolan and Terry Speed has details about a larger dataset from which this set is drawn.

The table `baby` contains the following variables for mother-baby pairs: 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, pregnancy weight in pounds, and whether or not the mother smoked during pregnancy.

```
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)

Birth weight is an important factor in the health of a newborn infant – smaller babies tend to need more medical care in their first days than larger newborns. It is therefore helpful to have an estimate of birth weight before the baby is born. One way to do this is to examine the relationship between birth weight and the number of gestational days.

A simple measure of this relationship is the ratio of birth weight to the number of gestational days. The table `ratios` contains the first two columns of `baby`, as well as a column of the ratios. The first entry in that column was calculated as follows:

$$\frac{120 \text{ ounces}}{284 \text{ days}} \approx 0.4225 \text{ ounces per day}$$

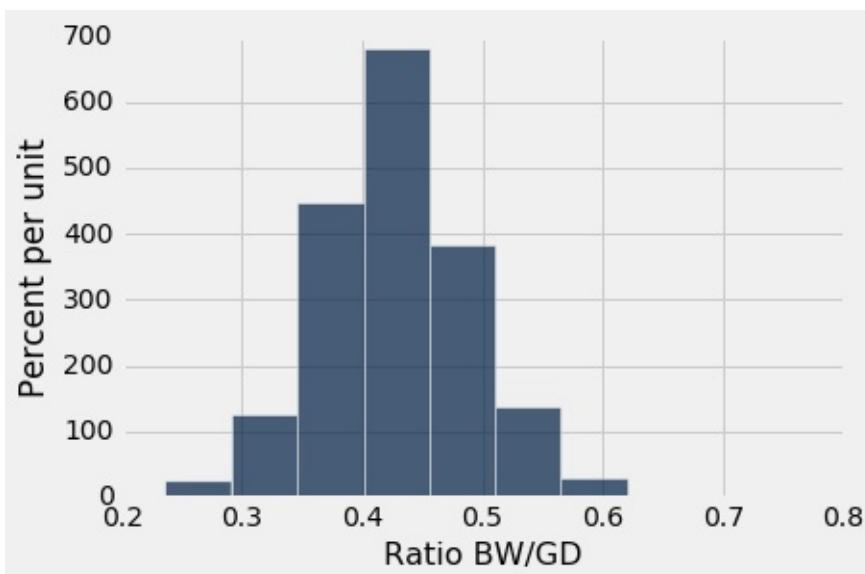
```
ratios = baby.select('Birth Weight', 'Gestational Days').with_column(
    'Ratio BW/GD', baby.column('Birth Weight')/baby.column('Gestational Days'))
```

ratios

Birth Weight	Gestational Days	Ratio BW/GD
120	284	0.422535
113	282	0.400709
128	279	0.458781
108	282	0.382979
136	286	0.475524
138	244	0.565574
132	245	0.538776
120	289	0.415225
143	299	0.478261
140	351	0.39886
... (1164 rows omitted)		

Here is a histogram of the ratios.

```
ratios.select('Ratio BW/GD').hist()
```



At first glance the histogram looks quite symmetric, with the density at its maximum over the interval 4 ounces per day to 4.5 ounces per day. But a closer look reveals that some of the ratios were quite large by comparison. The maximum value of the ratios was just over 0.78 ounces per day, almost double the typical value.

```
ratios.sort('Ratio BW/GD', descending=True).take(0)
```

Birth Weight	Gestational Days	Ratio BW/GD
116	148	0.783784

The median gives a sense of the typical ratio because it is unaffected by the very large or very small ratios. The median ratio in the sample is about 0.429 ounces per day.

```
np.median(ratios.column(2))
```

```
0.42907801418439717
```

But what was the median in the population? We don't know, so we will estimate it.

Our method will be exactly the same as in the previous section. We will bootstrap the sample 5,000 times resulting in 5,000 estimates of the median. Our 95% confidence interval will be the "middle 95%" of all of our estimates.

Recall the function `bootstrap_median` defined in the previous section. We will call this function and construct a 95% confidence interval for the median ratio in the population. Remember that the table `ratios` contains the relevant data from our original sample.

```
def bootstrap_median(original_sample, label, replications):

    """Returns an array of bootstrapped sample medians:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    medians = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_median = percentile(50,
            bootstrap_sample.column(0))
        medians = np.append(medians, resampled_median)

    return medians
```

```
# Generate the medians from 5000 bootstrap samples
bstrap_medians = bootstrap_median(ratios, 'Ratio BW/GD', 5000)

# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_medians)
right = percentile(97.5, bstrap_medians)

make_array(left, right)

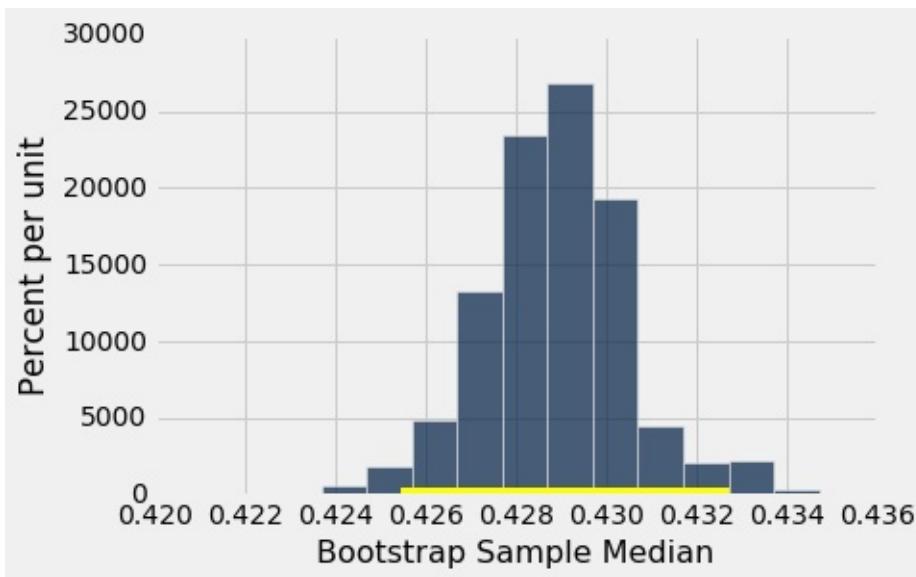
array([ 0.42545455,  0.43272727])
```

The 95% confidence interval goes from about 0.425 ounces per day to about 0.433 ounces per day. We are estimating the median "birth weight to gestational days" ratio in the population is somewhere in the interval 0.425 ounces per day to 0.433 ounces per day.

The estimate of 0.429 based on the original sample happens to be exactly half-way in between the two ends of the interval, though that need not be true in general.

To visualize our results, let us draw the empirical histogram of our bootstrapped medians and place the confidence interval on the horizontal axis.

```
resampled_medians = Table().with_column(
    'Bootstrap Sample Median', bstrap_medians
)
resampled_medians.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0),
           color='yellow', lw=8);
```



This histogram and interval resembles those we drew in the previous section, with one big difference – there is no red dot showing where the parameter is. We don't know where that dot should be, or whether it is even in the interval.

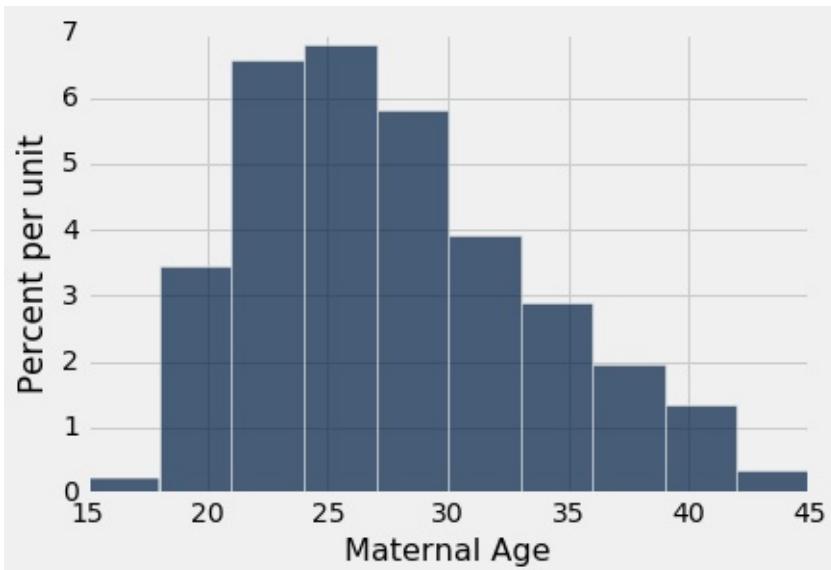
We just have an interval of estimates. It is a 95% confidence interval of estimates, because the process that generates it produces a good interval about 95% of the time. That certainly beats guessing at random!

Keep in mind that this interval is an approximate 95% confidence interval. There are many approximations involved in its computation. The approximation is not bad, but it is not exact.

Confidence Interval for a Population Mean: Bootstrap Percentile Method

What we have done for medians can be done for means as well. Suppose we want to estimate the average age of the mothers in the population. A natural estimate is the average age of the mothers in the sample. Here is the distribution of their ages, and their average age which was about 27.2 years.

```
baby.select('Maternal_Age').hist()
```



```
np.mean(baby.column('Maternal_Age'))
```

```
27.228279386712096
```

What was the average age of the mothers in the population? We don't know the value of this parameter.

Let's estimate the unknown parameter by the bootstrap method. To do this, we will edit the code for `bootstrap_median` to instead define the function `bootstrap_mean`. The code is the same except that the statistics are means instead of medians, and are collected in an array called `means` instead of `medians`.

```

def bootstrap_mean(original_sample, label, replications):

    """Returns an array of bootstrapped sample means:
    original_sample: table containing the original sample
    label: label of column containing the variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    means = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resampled_mean = np.mean(bootstrap_sample.column(0))
        means = np.append(means, resampled_mean)

    return means

```

```

# Generate the means from 5000 bootstrap samples
bstrap_means = bootstrap_mean(baby, 'Maternal Age', 5000)

# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_means)
right = percentile(97.5, bstrap_means)

make_array(left, right)

```

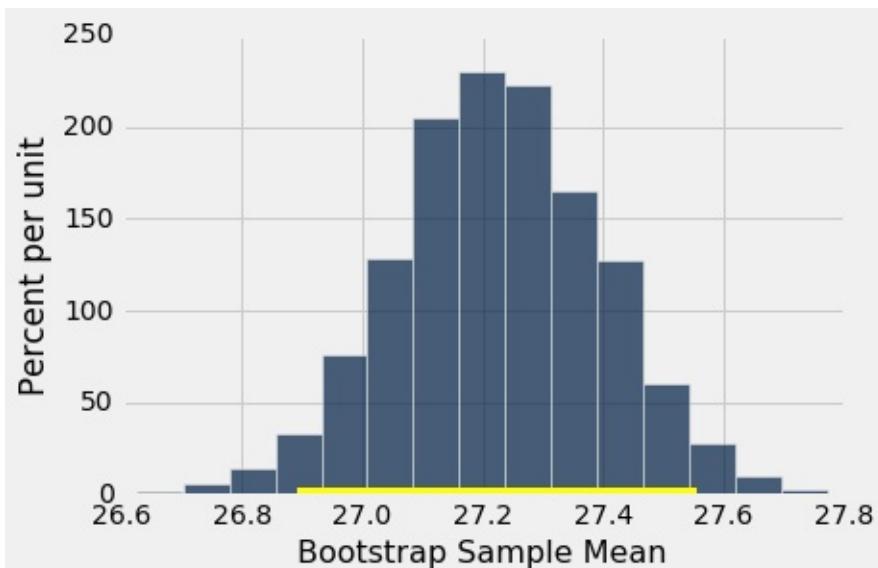
```
array([ 26.89011925,  27.55536627])
```

The 95% confidence interval goes from about 26.9 years to about 27.6 years. That is, we are estimating that the average age of the mothers in the population is somewhere in the interval 26.9 years to 27.6 years.

Notice how close the two ends are to the average of about 27.2 years in the original sample. The sample size is very large – 1,174 mothers – and so the sample averages don't vary much. We will explore this observation further in the next chapter.

The empirical histogram of the 5,000 bootstrapped means is shown below, along with the 95% confidence interval for the population mean.

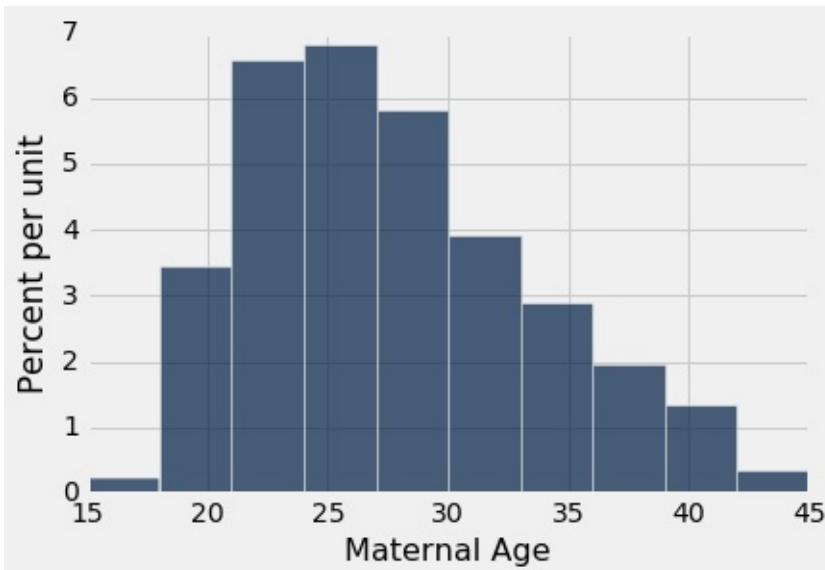
```
resampled_means = Table().with_column(
    'Bootstrap Sample Mean', bstrap_means
)
resampled_means.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0),
color='yellow', lw=8);
```



Once again, the average of the original sample (27.23 years) is close to the center of the interval. That's not very surprising, because each bootstrapped sample is drawn from that same original sample. The averages of the bootstrapped samples are about symmetrically distributed on either side of the average of the sample from which they were drawn.

Notice also that the empirical histogram of the resampled means has roughly a symmetric bell shape, even though the histogram of the sampled ages was not symmetric at all:

```
baby.select('Maternal Age').hist()
```



This is a consequence of the Central Limit Theorem of probability and statistics. In later sections, we will see what the theorem says.

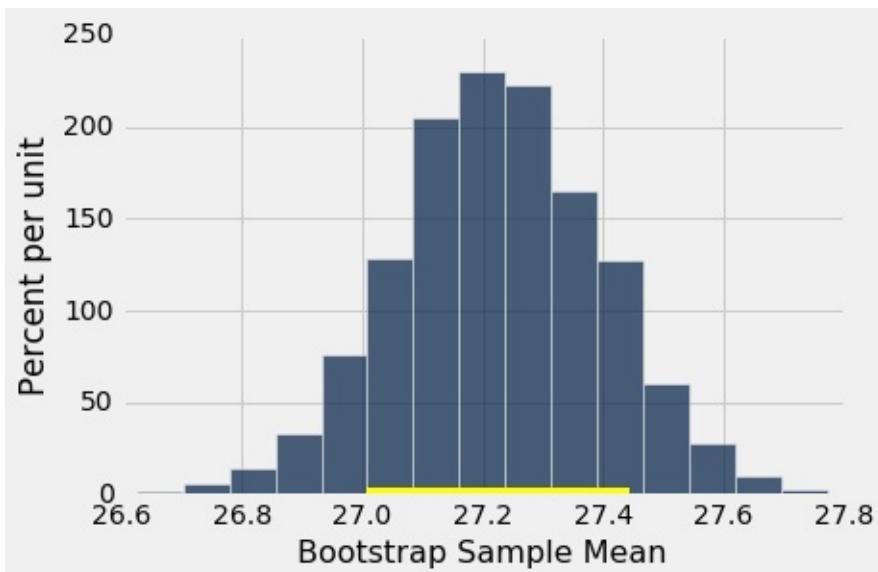
An 80% Confidence Interval

You can use the bootstrapped sample means to construct an interval of any level of confidence. For example, to construct an 80% confidence interval for the mean age in the population, you would take the "middle 80%" of the resampled means. So you would want 10% of the distribution in each of the two tails, and hence the endpoints would be the 10th and 90th percentiles of the resampled means.

```
left_80 = percentile(10, bstrap_means)
right_80 = percentile(90, bstrap_means)
make_array(left_80, right_80)
```

```
array([ 27.0076661 ,  27.44293015])
```

```
resampled_means.hist(bins=15)
plots.plot(make_array(left_80, right_80), make_array(0, 0),
color='yellow', lw=8);
```



This 80% confidence interval is much shorter than the 95% confidence interval. It only goes from about 27.0 years to about 27.4 years. While that's a tight set of estimates, you know that this process only produces a good interval about 80% of the time.

The earlier process produced a wider interval but we had more confidence in the process that generated it.

To get a narrow confidence interval at a high level of confidence, you'll have to start with a larger sample. We'll see why in the next chapter.

Confidence Interval for a Population Proportion: Bootstrap Percentile Method

In the sample, 39% of the mothers smoked during pregnancy.

```
baby.where('Maternal Smoker',
           are.equal_to(True)).num_rows/baby.num_rows
```

```
0.3909710391822828
```

For what follows is useful to observe that this proportion can also be calculated by an array operation:

```
smoking = baby.column('Maternal Smoker')
np.count_nonzero(smoking)/len(smoking)
```

```
0.3909710391822828
```

What percent of mothers in the population smoked during pregnancy? This is an unknown parameter which we can estimate by a bootstrap confidence interval. The steps in the process are analogous to those we took to estimate the population mean and median.

We will start by defining a function `bootstrap_proportion` that returns an array of bootstrapped sample proportions. Once again, we will achieve this by editing our definition of `bootstrap_median`. The only change in computation is in replacing the median of the resample by the proportion of smokers in it. The code assumes that the column of data consists of Boolean values. The other changes are only to the names of arrays, to help us read and understand our code.

```
def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    proportions = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resample_array = bootstrap_sample.column(0)
        resampled_proportion =
            np.count_nonzero(resample_array)/len(resample_array)
            proportions = np.append(proportions,
        resampled_proportion)

    return proportions
```

Let us use `bootstrap_proportion` to construct an approximate 95% confidence interval for the percent of smokers among the mothers in the population. The code is analogous to the corresponding code for the mean and median.

```
# Generate the proportions from 5000 bootstrap samples
bstrap_props = bootstrap_proportion(baby, 'Maternal Smoker',
5000)

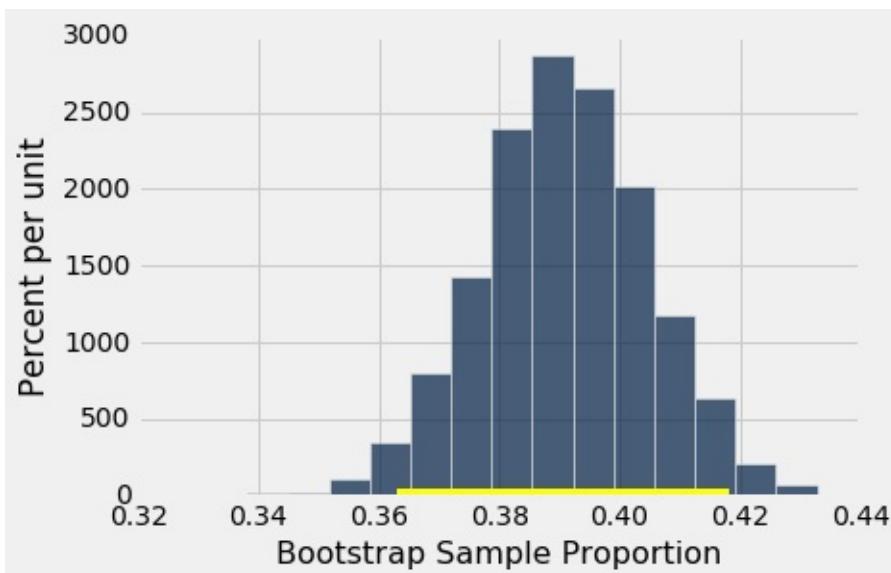
# Get the endpoints of the 95% confidence interval
left = percentile(2.5, bstrap_props)
right = percentile(97.5, bstrap_props)

make_array(left, right)
```

```
array([ 0.36286201,  0.41822828])
```

The confidence interval goes from about 36% to about 42%. The original sample percent of 39% is very close to the center of the interval, as you can see below.

```
resampled_proportions = Table().with_column(
    'Bootstrap Sample Proportion', bstrap_props
)
resampled_proportions.hist(bins=15)
plots.plot(make_array(left, right), make_array(0, 0),
color='yellow', lw=8);
```



Care in Using the Bootstrap

The bootstrap is an elegant and powerful method. Before using it, it is important to keep some points in mind.

- Start with a large random sample. If you don't, the method might not work. Its success is based on large random samples (and hence also resamples from the sample) resembling the population. The Law of Averages says that this is likely to be true provided the random sample is large.
- To approximate the probability distribution of a statistic, it is a good idea to replicate the resampling procedure as many times as possible. A few thousand replications will result in decent approximations to the distribution of sample median, especially if the distribution of the population has one peak and is not very asymmetric. We used 5,000 replications in our examples but would recommend 10,000 in general.
- The bootstrap percentile method works well for estimating the population median or mean based on a large random sample. However, it has limitations, as do all methods of estimation. For example, it is not expected to do well in the following situations.
 - The goal is to estimate the minimum or maximum value in the population, or a very low or very high percentile, or parameters that are greatly influenced by rare elements of the population.
 - The probability distribution of the statistic is not roughly bell shaped.
 - The original sample is very small, say less than 10 or 15.

[Interact](#)

```
def bootstrap_median(original_sample, label, replications):  
  
    """Returns an array of bootstrapped sample medians:  
    original_sample: table containing the original sample  
    label: label of column containing the variable  
    replications: number of bootstrap samples  
    """  
  
    just_one_column = original_sample.select(label)  
    medians = make_array()  
    for i in np.arange(replications):  
        bootstrap_sample = just_one_column.sample()  
        resampled_median = percentile(50,  
                                     bootstrap_sample.column(0))  
        medians = np.append(medians, resampled_median)  
  
    return medians
```

```
def bootstrap_mean(original_sample, label, replications):  
  
    """Returns an array of bootstrapped sample means:  
    original_sample: table containing the original sample  
    label: label of column containing the variable  
    replications: number of bootstrap samples  
    """  
  
    just_one_column = original_sample.select(label)  
    means = make_array()  
    for i in np.arange(replications):  
        bootstrap_sample = just_one_column.sample()  
        resampled_mean = np.mean(bootstrap_sample.column(0))  
        means = np.append(means, resampled_mean)  
  
    return means
```

```

def bootstrap_proportion(original_sample, label, replications):

    """Returns an array of bootstrapped sample proportions:
    original_sample: table containing the original sample
    label: label of column containing the Boolean variable
    replications: number of bootstrap samples
    """

    just_one_column = original_sample.select(label)
    proportions = make_array()
    for i in np.arange(replications):
        bootstrap_sample = just_one_column.sample()
        resample_array = bootstrap_sample.column(0)
        resampled_proportion =
            np.count_nonzero(resample_array)/len(resample_array)
        proportions = np.append(proportions,
                               resampled_proportion)

    return proportions

```

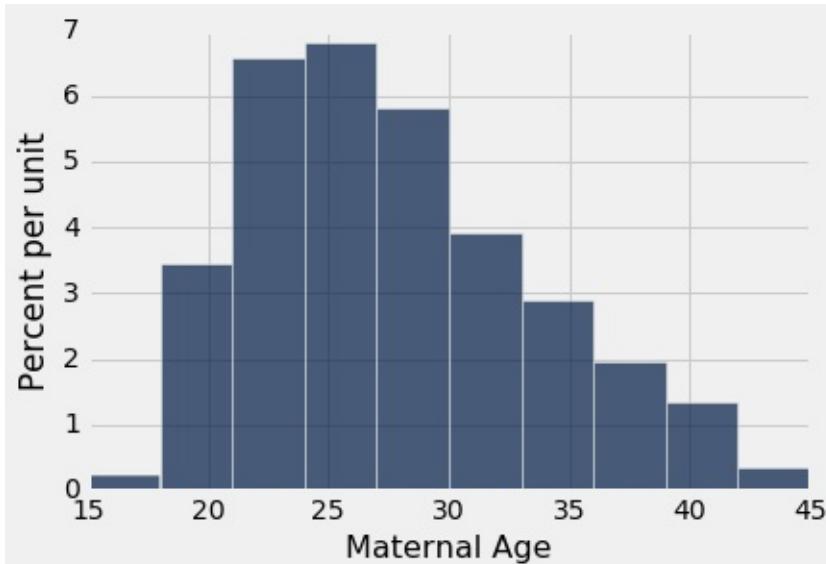
Using Confidence Intervals

A confidence interval has a single purpose – to estimate an unknown parameter based on data in a random sample. In the last section, we said that the interval (36%, 42%) was an approximate 95% confidence interval for the percent of smokers among mothers in the population. That was a formal way of saying that by our estimate, the percent of smokers among the mothers in the population was somewhere between 36% and 42%, and that our process of estimation is correct about 95% of the time.

It is important to resist the impulse to use confidence intervals for other purposes. For example, recall that we calculated the interval (26.9 years, 27.6 years) as an approximate 95% confidence interval for the average age of mothers in the population. A disarmingly common misuse of the interval is to conclude that about 95% of the women were between 26.9 years and 27.6 years old. You don't need to know much about confidence intervals to see that this can't be right – you wouldn't expect 95% of mothers to all be within a few months of each other in age. Indeed, the histogram of the sampled ages shows quite a bit of variation.

```
baby = Table.read_table('baby.csv')
```

```
baby.select('Maternal Age').hist()
```



A small percent of the sampled ages are in the (26.9, 27.6) interval, and you would expect a similar small percent in the population. The interval just estimates one number: the *average* of all the ages in the population.

However, estimating a parameter by confidence intervals does have an important use besides just telling us roughly how big the parameter is.

Using a Confidence Interval to Test Hypotheses

Our approximate 95% confidence interval for the average age in the population goes from 26.9 years to 27.6 years. Suppose someone wants to test the following hypotheses:

Null hypothesis. The average age in the population is 30 years.

Alternative hypothesis. The average age in the population is not 30 years.

Then, if you were using the 5% cutoff for the P-value, you would reject the null hypothesis. This is because 30 is not in the 95% confidence interval for the population average. At the 5% level of significance, 30 is not a plausible value for the population average.

This use of confidence intervals is the result of a *duality* between confidence intervals and tests: if you are testing whether or not the population mean is a particular value x , and you use the 5% cutoff for the P-value, then you will reject the null hypothesis if x is not in your 95% confidence interval for the mean.

This can be established by statistical theory. In practice, it just boils down to checking whether or not the value specified in the null hypothesis lies in the confidence interval.

If you were using the 1% cutoff for the P-value, you would have to check if the value specified in the null hypothesis lies in a 99% confidence interval for the population mean.

To a rough approximation, these statements are also true for population proportions, provided the sample is large.

While we now have a way of using confidence intervals to test a particular kind of hypothesis, you might wonder about the value of testing whether or not the average age in a population is equal to 30. Indeed, the value isn't clear. But there are some situations in which a test of this kind of hypothesis is both natural and useful.

We will study this in the context of data that 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 can be 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
```

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
... (12 rows omitted)				

We will compare the baseline and 15-month scores. As each row corresponds to one patient, we say that the sample of baseline scores and the sample of 15-month scores are *paired* - they are not just two sets of 22 values each, but 22 pairs of values, one for each patient.

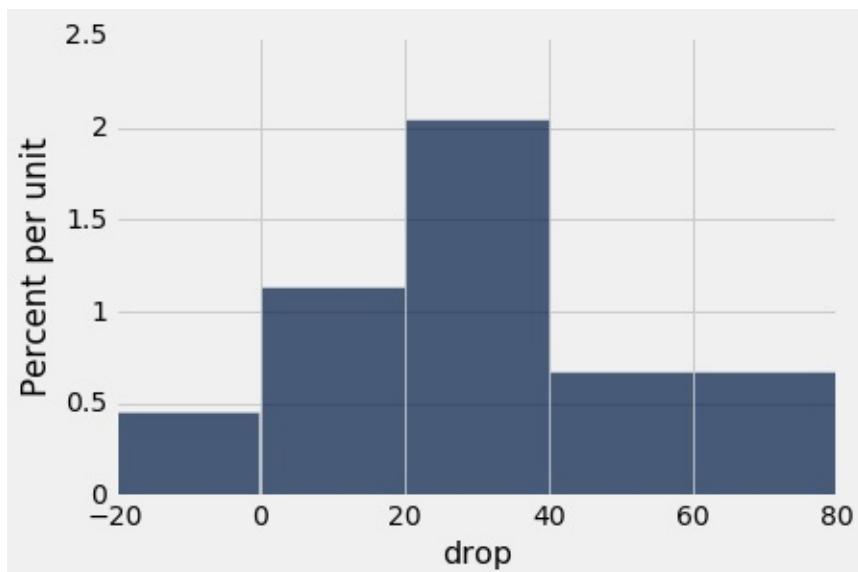
At a glance, you can see that the 15-month scores tend to be lower than the baseline scores – the sampled patients' lungs seem to be doing worse 15 months after the treatment. This is confirmed by the mostly positive values in the column `drop`, the amount by which the score dropped from baseline to 15 months.

```
hodgkins = hodgkins.with_column(
    'drop', hodgkins.column('base') - hodgkins.column('month15')
)
```

```
hodgkins
```

height	rad	chemo	base	month15	drop
164	679	180	160.57	87.77	72.8
168	311	180	98.24	67.62	30.62
173	388	239	129.04	133.33	-4.29
157	370	168	85.41	81.28	4.13
160	468	151	67.94	79.26	-11.32
170	341	96	150.51	80.97	69.54
163	453	134	129.88	69.24	60.64
175	529	264	87.45	56.48	30.97
185	392	240	149.84	106.99	42.85
178	479	216	92.24	73.43	18.81
... (12 rows omitted)					

```
hodgkins.select('drop').hist(bins=np.arange(-20, 81, 20))
```



```
np.mean(hodgkins.column('drop'))
```

```
28.615909090909096
```

But could this be the result of chance variation? It really doesn't seem so, but the data are from a random sample. Could it be that in the entire population of patients, the average drop is just 0?

To answer this, we can set up two hypotheses:

Null hypothesis. In the population, the average drop is 0.

Alternative hypothesis. In the population, the average drop is not 0.

To test this hypothesis with a 1% cutoff for the P-value, let's construct an approximate 99% confidence interval for the average drop in the population.

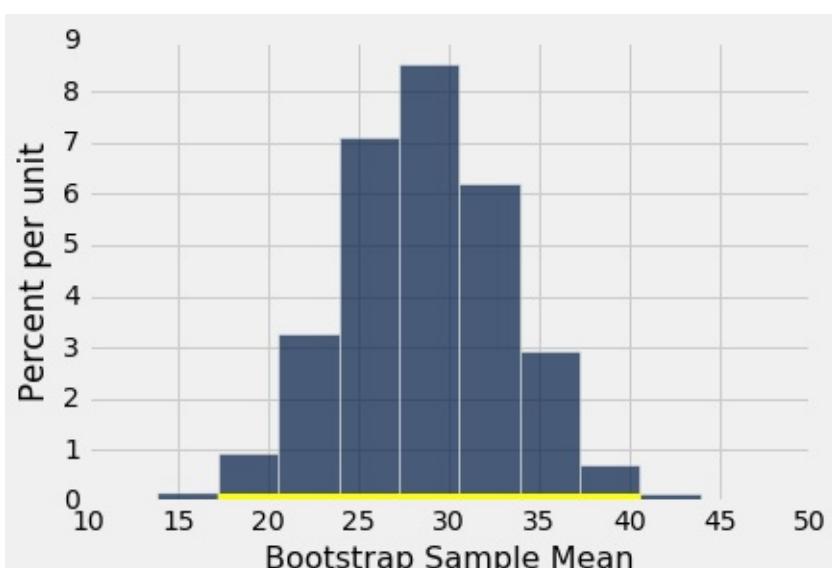
```
bstrap_means = bootstrap_mean(hodgkins, 'drop', 10000)

left = percentile(0.5, bstrap_means)
right = percentile(99.5, bstrap_means)

make_array(left, right)
```

```
array([ 17.25045455,  40.60136364])
```

```
resampled_means = Table().with_column(
    'Bootstrap Sample Mean', bstrap_means
)
resampled_means.hist()
plots.plot(make_array(left, right), make_array(0, 0),
color='yellow', lw=8);
```



The 99% confidence interval for the average drop in the population goes from about 17 to about 40. The interval doesn't contain 0. So we reject the null hypothesis.

But notice that we have done better than simply concluding that the average drop in the population isn't 0. We have estimated how big the average drop is. That's a more useful result than just saying, "It's not 0."

A note on accuracy. Our confidence interval is quite wide, for two main reasons:

- The confidence level is high (99%).
- The sample size is relatively small compared to those in our earlier examples.

In the next chapter, we will examine how the sample size affects accuracy. We will also examine how the empirical distributions of sample means so often come out bell shaped even though the distributions of the underlying data are not bell shaped at all.

Endnote

The terminology of a field usually comes from the leading researchers in that field. [Brad Efron](#), who first proposed the bootstrap technique, used a term that has [American origins](#). Not to be outdone, Chinese statisticians have [proposed their own method](#).

[Interact](#)

Why the Mean Matters

In this course we have studied several different statistics, including total variation distance, the maximum, the median, and also the mean. Under clear assumptions about randomness, we have drawn empirical distributions of all of these statistics. Some, like the maximum and the total variation distance, have distributions that are clearly skewed in one direction or the other. But the empirical distribution of the sample mean has almost always turned out close to bell-shaped, regardless of the population being studied.

If a property of random samples is true *regardless of the population*, it becomes a powerful tool for inference because we rarely know much about the data in the entire population. The distribution of the mean of a large random sample falls into this category of properties. That is why random sample means are extensively used in data science.

In this chapter, we will study means and what we can say about them with only minimal assumptions about the underlying populations. Question that we will address include:

- What exactly does the mean measure?
- How close to the mean are most of the data?
- How is the sample size related to the variability of the sample mean?
- Why do empirical distributions of random sample means come out bell shaped?
- How can we use sample means effectively for inference?

[Interact](#)

Properties of the Mean

In this course, we have used the words "average" and "mean" interchangeably, and will continue to do so. The definition of the mean will be familiar to you from your high school days or even earlier.

Definition. The *average* or *mean* of a collection of numbers is the sum of all the elements of the collection, divided by the number of elements in the collection.

The methods `np.average` and `np.mean` return the mean of an array.

```
not_symmetric = make_array(2, 3, 3, 9)
```

```
np.average(not_symmetric)
```

```
4.25
```

```
np.mean(not_symmetric)
```

```
4.25
```

Basic Properties

The definition and the example above point to some properties of the mean.

- It need not be an element of the collection.
- It need not be an integer even if all the elements of the collection are integers.
- It is somewhere between the smallest and largest values in the collection.
- It need not be halfway between the two extremes; it is not in general true that half the elements in a collection are above the mean.
- If the collection consists of values of a variable measured in specified units, then the mean has the same units too.

We will now study some other properties that are helpful in understanding the mean and its relation to other statistics.

The Mean is a "Smoothener"

You can think of taking the mean as an "equalizing" or "smoothing" operation. For example, imagine the entries in `not_symmetric` above as the dollars in the pockets of four different people. To get the mean, you first put all of the money into one big pot and then divide it evenly among the four people. They had started out with different amounts of money in their pockets (\$2, \$3, \$3, and \$9), but now each person has \$4.25, the mean amount.

Proportions are Means

If a collection consists only of ones and zeroes, then the sum of the collection is the number of ones in it, and the mean of the collection is the proportion of ones.

```
zero_one = make_array(1, 1, 1, 0)
sum(zero_one)
```

3

```
np.mean(zero_one)
```

0.75

You can replace 1 by the Boolean `True` and 0 by `False`:

```
np.mean(make_array(True, True, True, False))
```

0.75

Because proportions are a special case of means, results about random sample means apply to random sample proportions as well.

The Mean and the Histogram

The mean of the collection $\{2, 3, 3, 9\}$ is 4.25, which is not the "halfway point" of the data. So then what does the mean measure?

To see this, notice that the mean can be calculated in different ways.

$$\text{mean} = 4.25$$

$$\begin{aligned}
 &= \frac{2 + 3 + 3 + 9}{4} \\
 &= 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{4} + 3 \cdot \frac{1}{4} + 9 \cdot \frac{1}{4} \\
 &= 2 \cdot \frac{1}{4} + 3 \cdot \frac{2}{4} + 9 \cdot \frac{1}{4} \\
 &= 2 \cdot 0.25 + 3 \cdot 0.5 + 9 \cdot 0.25
 \end{aligned}$$

The last expression is an example of a general fact: when we calculate the mean, each distinct value in the collection is *weighted* by the proportion of times it appears in the collection.

This has an important consequence. The mean of a collection depends only on the distinct values and their proportions, not on the number of elements in the collection. In other words, the mean of a collection depends only on the distribution of values in the collection.

Therefore, if two collections have the same distribution, then they have the same mean.

For example, here is another collection that has the same distribution as `not_symmetric` and hence the same mean.

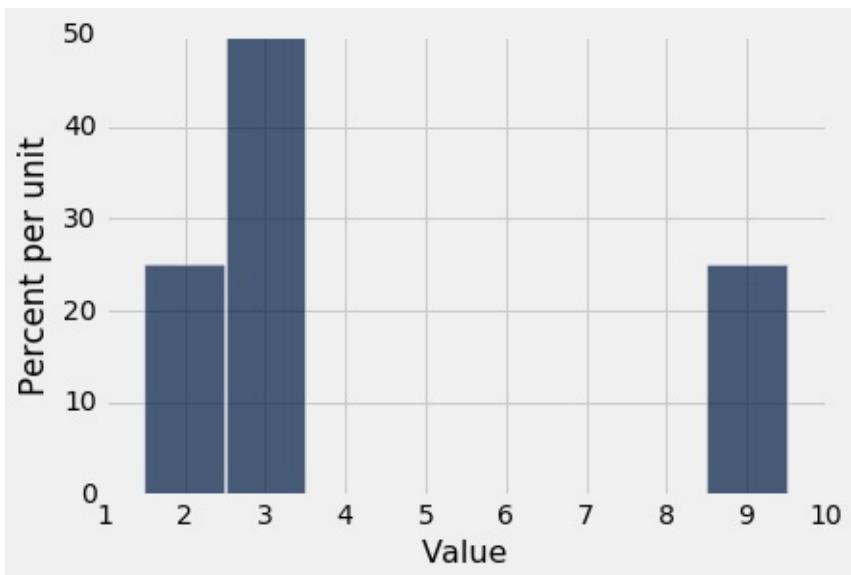
```
not_symmetric
```

```
array([2, 3, 3, 9])
```

```
same_distribution = make_array(2, 2, 3, 3, 3, 3, 9, 9)
np.mean(same_distribution)
```

```
4.25
```

The mean is a physical attribute of the histogram of the distribution. Here is the histogram of the distribution of `not_symmetric` or equivalently the distribution of `same_distribution`.

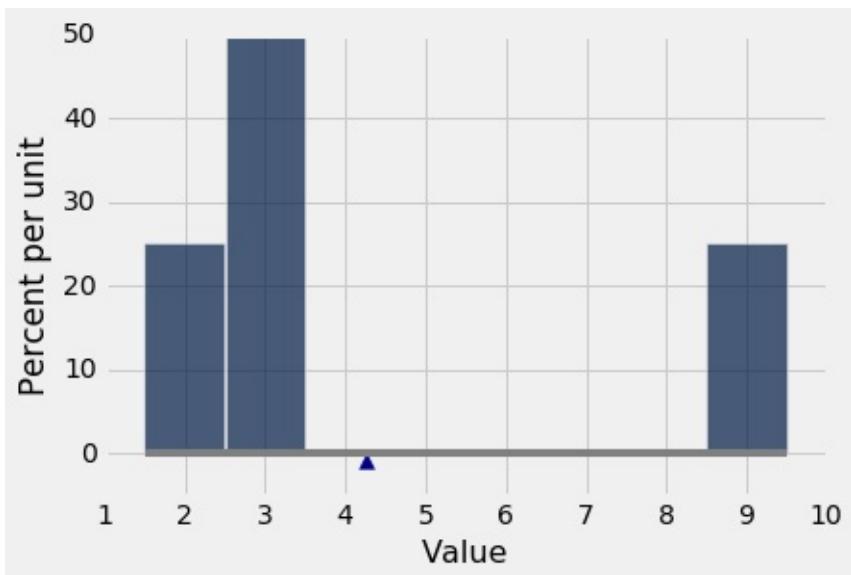


Imagine the histogram as a figure made out of cardboard attached to a wire that runs along the horizontal axis, and imagine the bars as weights attached at the values 2, 3, and 9. Suppose you try to balance this figure on a point on the wire. If the point is near 2, the figure will tip over to the right. If the point is near 9, the figure will tip over to the left. Somewhere in between is the point where the figure will balance; that point is the 4.25, the mean.

The mean is the center of gravity or balance point of the histogram.

To understand why that is, it helps to know some physics. The center of gravity is calculated exactly as we calculated the mean, by using the distinct values weighted by their proportions.

Because the mean is a balance point, it is sometimes displayed as a *fulcrum* or triangle at the base of the histogram.



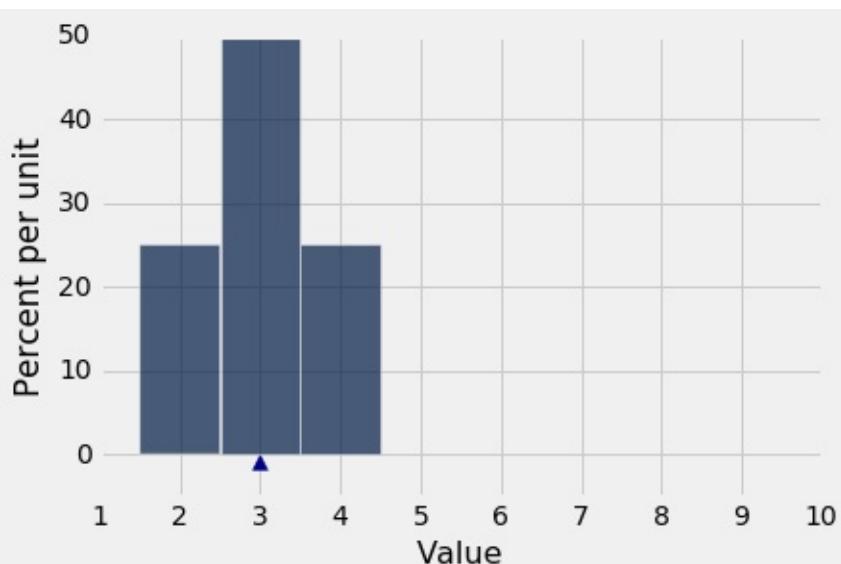
The Mean and the Median

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.

The relationship is easy to see in a simple example. Here is a histogram of the collection {2, 3, 3, 4} which is in the array `symmetric`. The distribution is symmetric about 3. The mean and the median are both equal to 3.

```
symmetric = make_array(2, 3, 3, 4)
```



```
np.mean(symmetric)
```

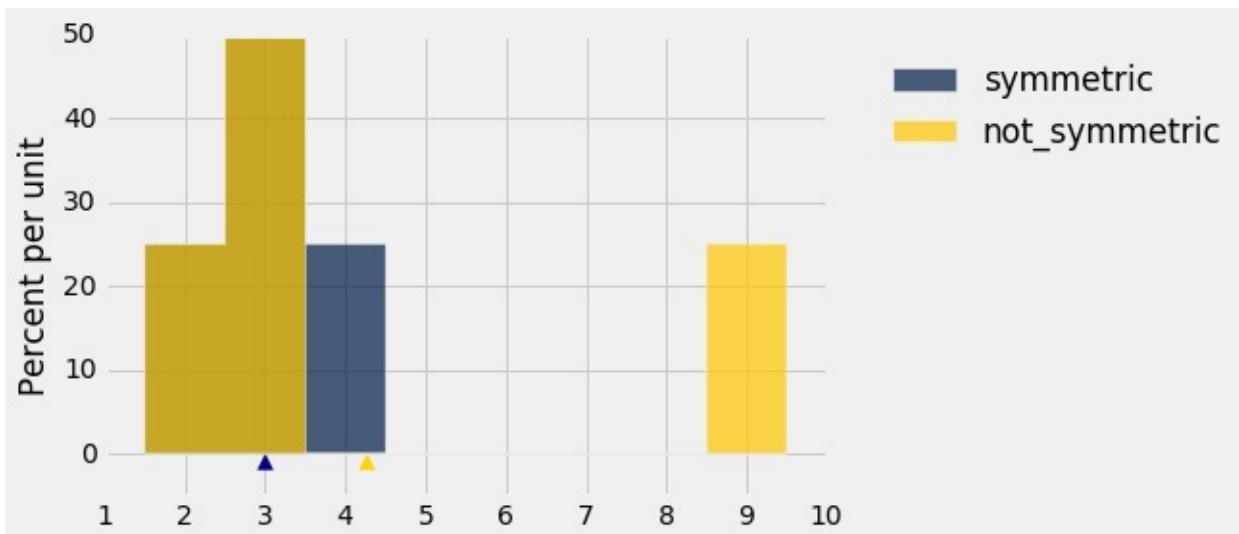
```
3.0
```

```
percentile(50, symmetric)
```

```
3
```

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

What if the distribution is not symmetric? Let's compare `symmetric` and `not_symmetric`.



The blue histogram represents the original `symmetric` distribution. The gold histogram of `not_symmetric` starts out the same as the blue at the left end, but its rightmost bar has slid over to the value 9. The brown part is where the two histograms overlap.

The median and mean of the blue distribution are both equal to 3. The median of the gold distribution is also equal to 3, though the right half is distributed differently from the left.

But the mean of the gold distribution is not 3: the gold histogram would not balance at 3. The balance point has shifted to the right, to 4.25.

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.

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.

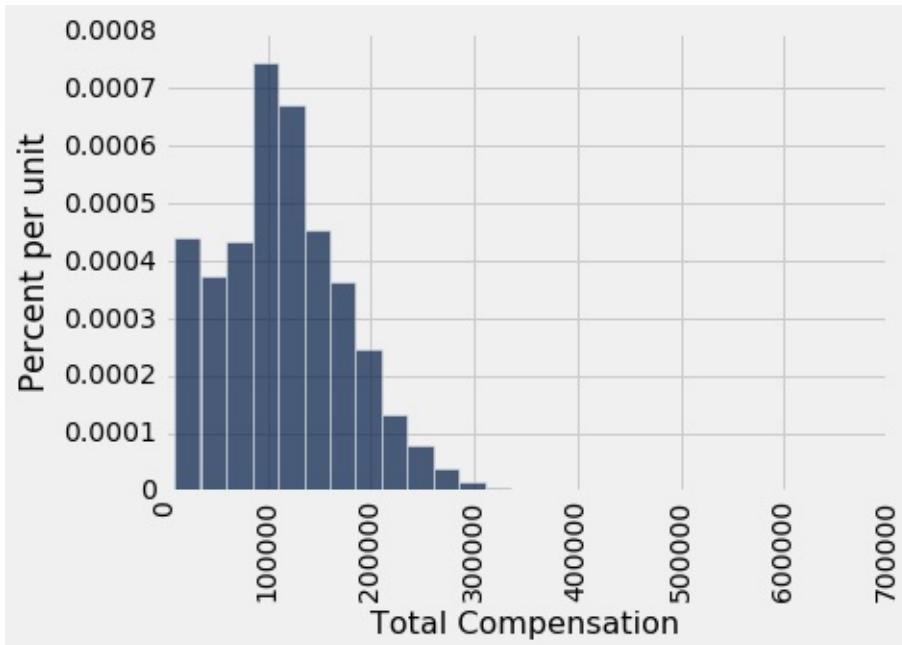
Example

The table `sf2015` contains salary and benefits data for San Francisco City employees in 2015. As before, we will restrict our analysis to those who had the equivalent of at least half-time employment for the year.

```
sf2015 =
Table.read_table('san_francisco_2015.csv').where('Salaries',
are.above(10000))
```

As we saw earlier, the highest compensation was above \$600,000 but the vast majority of employees had compensations below \$300,000.

```
sf2015.select('Total Compensation').hist(bins = np.arange(10000, 700000, 25000))
```



This histogram is skewed to the right; it has a right-hand tail.

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

```
compensation = sf2015.column('Total Compensation')
percentile(50, compensation)
```

```
110305.78999999999
```

```
np.mean(compensation)
```

```
114725.98411824222
```

Distributions of incomes of large populations tend to be right skewed. When the bulk of a population has middle to low incomes, but a very small proportion has very high incomes, the histogram has a long, thin tail to the right.

The mean income is affected by this tail: the farther the tail stretches to the right, the larger the mean becomes. But the median is not affected by values at the extremes of the distribution. That is why economists often summarize income distributions by the median

instead of the mean.

[Interact](#)

Variability

The mean tells us where a histogram balances. But in almost every histogram we have seen, the values spread out on both sides of the mean. How far from the mean can they be? To answer this question, we will develop a measure of variability about the mean.

We will start by describing how to calculate the measure. Then we will see why it is a good measure to calculate.

The Rough Size of Deviations from Average

For simplicity, we will begin our calculations in the context of a simple array `any_numbers` consisting of just four values. As you will see, our method will extend easily to any other array of values.

```
any_numbers = make_array(1, 2, 2, 10)
```

The goal is to measure roughly how far off the numbers are from their average. To do this, we first need the average:

```
# Step 1. The average.  
  
mean = np.mean(any_numbers)  
mean
```

```
3.75
```

Next, let's find out how far each value is from the mean. These are called the *deviations from the average*. A "deviation from average" is just a value minus the average. The table `calculation_steps` displays the results.

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

deviations = any_numbers - mean
calculation_steps = Table().with_columns(
    'Value', any_numbers,
    'Deviation from Average', deviations
)
calculation_steps
```

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's 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
calculation_steps = calculation_steps.with_column(
    'Squared Deviations from Average', squared_deviations
)
calculation_steps
```

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

```
# Step 4. Variance = the mean squared deviation from average

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

13.1875

Variance: The mean squared deviation calculated above is called the *variance* of the values.

While the variance does give us an idea of spread, it is not on the same scale as the original variable as 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:

```
# Step 5.
# 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 five steps described above result in the SD. You can also use the function `np.std` to compute the SD of values in an array:

```
np.std(any_numbers)
```

```
3.6314597615834874
```

Working with the SD¶

To see what we can learn from the SD, let's move to a more interesting dataset than `any_numbers`. The table `nba13` 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, his weight in pounds, and his age in years.

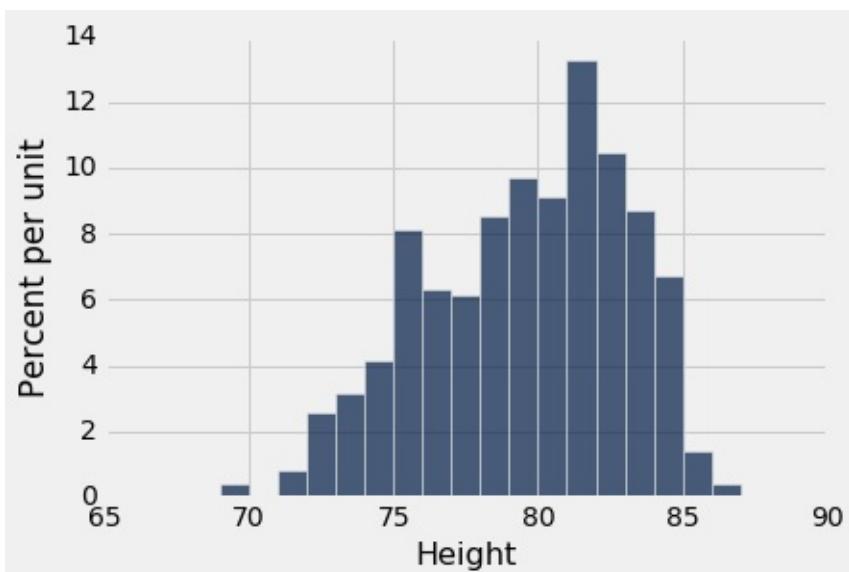
```
nba13 = Table.read_table('nba2013.csv')
nba13
```

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.

```
nba13.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(nba13.column('Height'))
mean_height
```

```
79.065346534653472
```

About how 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(nba13.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.

```
nba13.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 NBA players in 2013. His height was about 2.9 SDs below average.

```
nba13.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 where the histogram is situated on the number line.

First main reason for measuring spread by the SD

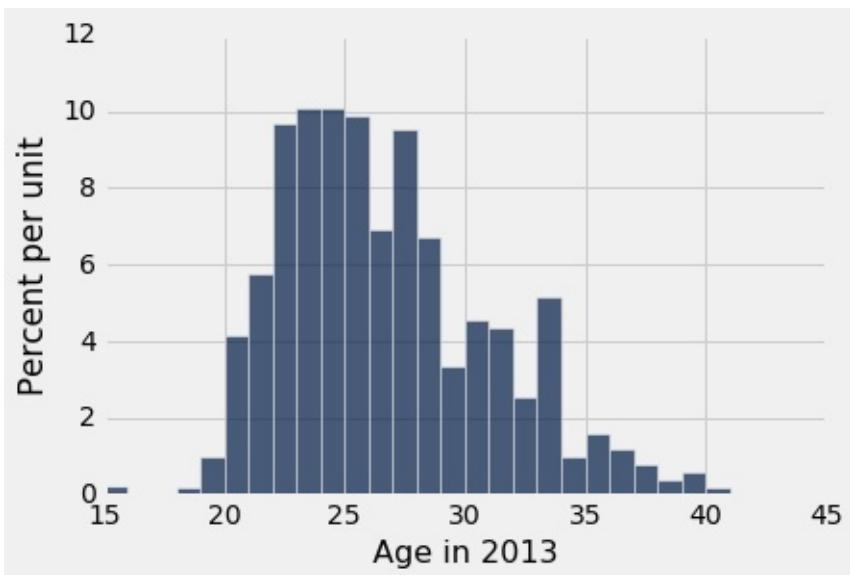
Informal statement. In all numerical data sets, the bulk of the entries are within the range "average \pm a few SDs".

For now, resist the desire to know exactly what fuzzy words like "bulk" and "few" mean. We will make them precise later in this section. Let's just examine the statement in the context of some more 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, along with the mean and SD of the ages.

```
nba13.select('Age in 2013').hist(bins=np.arange(15, 45, 1))
```



```
ages = nba13.column('Age in 2013')
mean_age = np.mean(ages)
sd_age = np.std(ages)
mean_age, sd_age
```

(26.19009900990099, 4.3212004417203067)

The average age was just over 26 years, and the SD was about 4.3 years.

How far off were the ages from the average? Just as we did with the heights, let's look at the two extreme values of the ages.

Juwani Howard was the oldest player, at 40.

```
nba13.sort('Age in 2013', descending=True).show(3)
```

Name	Position	Height	Weight	Age in 2013
Juwani Howard	Forward	81	250	40
Marcus Camby	Center	83	235	39
Derek Fisher	Guard	73	210	39

... (502 rows omitted)

Howard's age was about 3.2 SDs above average.

```
(40 - mean_age)/sd_age
```

```
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.

```
nba13.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 - mean_age)/sd_age
```

```
-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.

Chebychev's Bounds

The Russian mathematician [Pafnuty Chebychev](#) (1821-1894) proved a result that makes our rough statements precise.

For all lists, and all numbers z , the proportion of entries that are in the range "average $\pm z$ SDs" is at least $1 - \frac{1}{z^2}$.

It is important to note that the result gives a bound, not an exact value or an approximation.

What makes the result powerful is that it is true for all lists – all distributions, no matter how irregular.

Specifically, it says that for every list:

- the proportion in the range "average ± 2 SDs" is **at least $1 - 1/4 = 0.75$**
- the proportion in the range "average ± 3 SDs" is **at least $1 - 1/9 \approx 0.89$**

- the proportion in the range "average \pm 4.5 SDs" is at least $1 - 1/4.5^2 \approx 0.95$

As we noted above, Chebychev's result 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, the quantity 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 bounds imply 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(numbers_array):
    "Convert any array of numbers to standard units."
    return (numbers_array -
            np.mean(numbers_array))/np.std(numbers_array)
```

Example

As we saw in an earlier section, the table `united` contains a column `Delay` consisting of the departure delay times, in minutes, of over thousands of United Airlines flights in the summer of 2015. We will create a new column called `Delay (Standard Units)` 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.

```
united = Table.read_table('united_summer2015.csv')
united = united.with_column(
    'Delay (Standard Units)',
    standard_units(united.column('Delay'))
)
united
```

Date	Flight Number	Destination	Delay	Delay (Standard Units)
6/1/15	73	HNL	257	6.08766
6/1/15	217	EWR	28	0.287279
6/1/15	237	STL	-3	-0.497924
6/1/15	250	SAN	0	-0.421937
6/1/15	267	PHL	64	1.19913
6/1/15	273	SEA	-6	-0.573912
6/1/15	278	SEA	-8	-0.62457
6/1/15	292	EWR	12	-0.117987
6/1/15	300	HNL	20	0.0846461
6/1/15	317	IND	-10	-0.675228

... (13815 rows omitted)

The standard units that we can see are consistent with what we expect based on Chebychev's bounds. Most are of quite small size; only one is above 6.

But something rather alarming happens when we sort the delay times from highest to lowest. The standard units that we can see are extremely high!

```
united.sort('Delay', descending=True)
```

Date	Flight Number	Destination	Delay	Delay (Standard Units)
6/21/15	1964	SEA	580	14.269
6/22/15	300	HNL	537	13.1798
6/21/15	1149	IAD	508	12.4453
6/20/15	353	ORD	505	12.3693
8/23/15	1589	ORD	458	11.1788
7/23/15	1960	LAX	438	10.6722
6/23/15	1606	ORD	430	10.4696
6/4/15	1743	LAX	408	9.91236
6/17/15	1122	HNL	405	9.83637
7/27/15	572	ORD	385	9.32979

... (13815 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 10 hours). The highest value of delay is more than 14 in standard units.

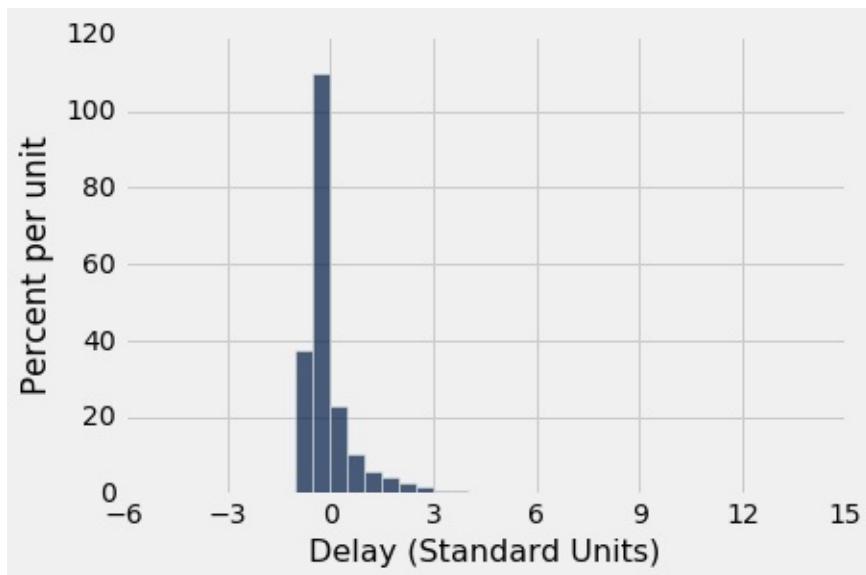
However, the proportion of these extreme values is small, and Chebychev's bounds still hold true. For example, let us calculate the percent of delay times that are in the range "average ± 3 SDs". This is the same as the percent of times for which the standard units are in the range $(-3, 3)$. That is about 98%, as computed below, consistent with Chebychev's bound of "at least 89%".

```
within_3_sd = united.where('Delay (Standard Units)',  
    are.between(-3, 3))  
within_3_sd.num_rows/united.num_rows
```

0.9790235081374322

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 $z = 14.27$ standard units (580 minutes). The area of the histogram outside the range $z = -3$ to $z = 3$ is about 2%, put together in tiny little bits that are mostly invisible in the histogram.

```
united.hist('Delay (Standard Units)', bins=np.arange(-5, 15.5,  
0.5))  
plots.xticks(np.arange(-6, 17, 3));
```



[Interact](#)

The SD and the Normal Curve

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. That is the bell-shaped distribution. This section examines that shape, as it appears frequently in probability histograms and also in some histograms of data.

A Roughly Bell-Shaped Histogram of Data

Let us look at the distribution of heights of mothers in our familiar sample of 1,174 mother-newborn pairs. 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.

```
baby = Table.read_table('baby.csv')
```

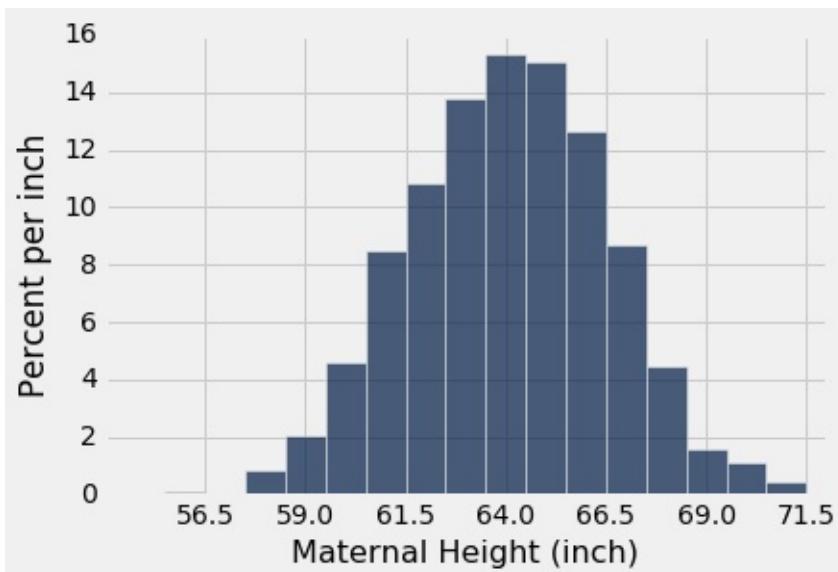
```
heights = baby.column('Maternal Height')
mean_height = np.round(np.mean(heights), 1)
mean_height
```

```
64.0
```

```
sd_height = np.round(np.std(heights), 1)
sd_height
```

```
2.5
```

```
baby.hist('Maternal Height', bins=np.arange(55.5, 72.5, 1),
unit='inch')
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 ± 3 . Because of the shape of the distribution, the "center" has an unambiguous meaning and is clearly visible at 64.

How to Spot the SD on a Bell Shaped Curve

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.

In general, **for bell-shaped distributions, the SD is the distance between the mean and the points of inflection on either side.**

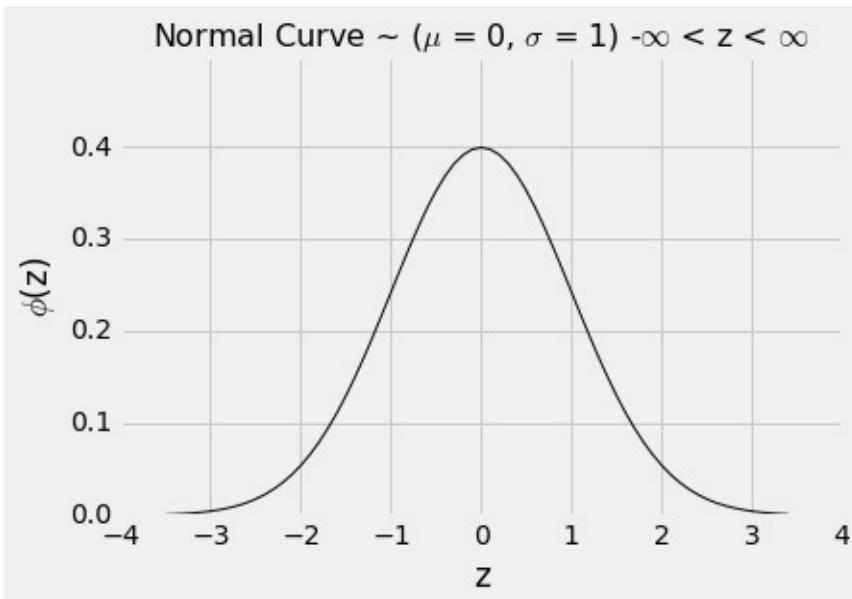
The standard normal curve

All the bell-shaped histograms that we have seen look essentially the same apart from the labels on the 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 outline of a histogram of a variable that has been measured in standard units and has a bell-shaped distribution.

$$\phi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2}, \quad -\infty < z < \infty$$



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.

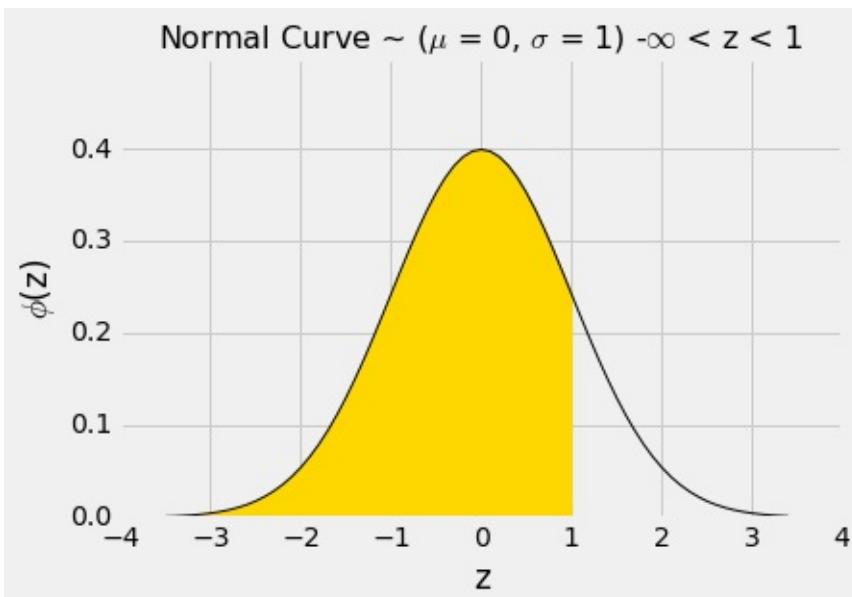
Therefore, areas under the curve have to be approximated. That is why almost all statistics textbooks carry tables of areas under the normal curve. It is also why all statistical systems, including a module of Python, include methods that provide excellent approximations to those areas.

```
from scipy import stats
```

The standard normal "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.



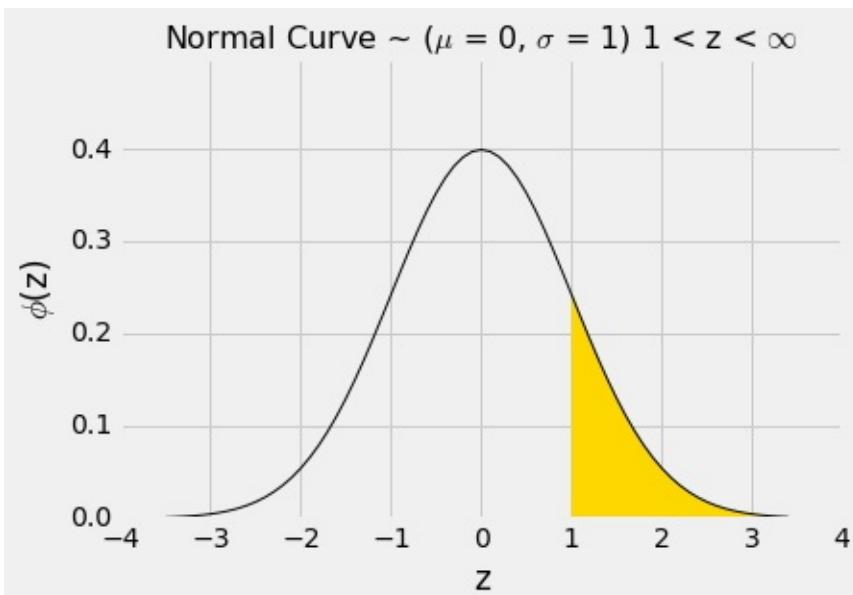
The numerical value of the shaded area can be found by calling `stats.norm.cdf`.

```
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.

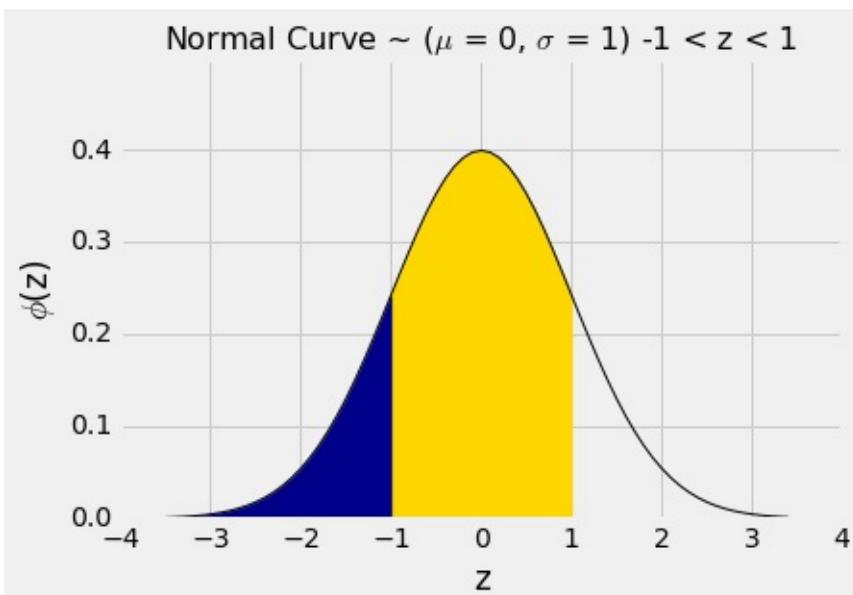
The area to the right of $z = 1$ is about 100% - 84% = 16%.



```
1 - stats.norm.cdf(1)
```

```
0.15865525393145707
```

The area between $z = -1$ and $z = 1$ can be computed in several different ways. It is the gold area under the curve below.



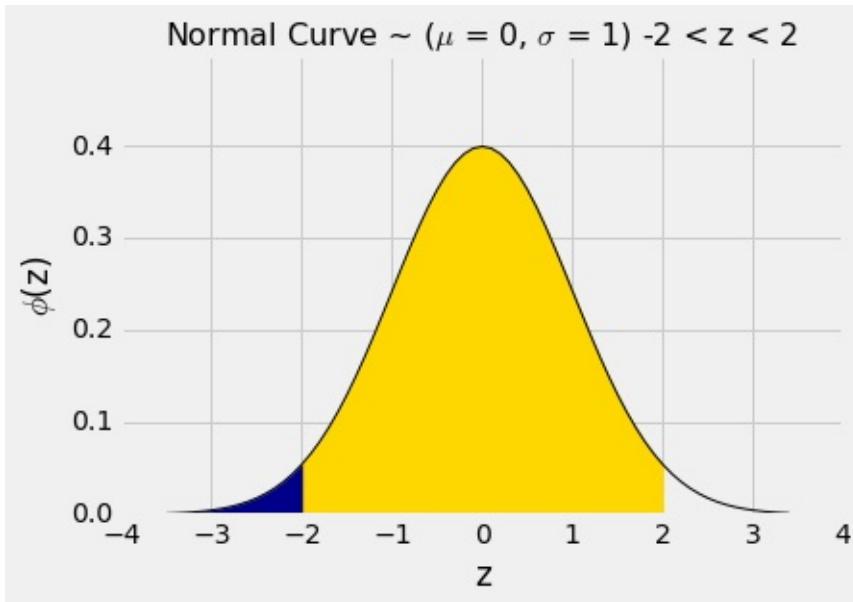
For example, we could calculate the area as "100% - two equal tails", which works out to roughly $100\% - 2 \times 16\% = 68\%$.

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%.



```
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 ± 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 when $z = 1$, Chebychev's bound is correct but not illuminating.

Percent in Range	All Distributions: Bound	Normal Distribution: Approximation
average ± 1 SD	at least 0%	about 68%
average ± 2 SDs	at least 75%	about 95%
average ± 3 SDs	at least 88.888...%	about 99.73%

[Interact](#)

The Central Limit Theorem

Very few of the data histograms that we have seen in this course have been bell shaped.

When we have come across a bell shaped distribution, it has almost invariably been an empirical histogram of a statistic based on a random sample.

The examples below show two very different situations in which an approximate bell shape appears in such histograms.

Net Gain in Roulette

In an earlier section, the bell appeared as the rough shape of the total amount of money we would make if we placed the same bet repeatedly on different spins of a roulette wheel.

```
wheel1
```

Pocket	Color
0	green
00	green
1	red
2	black
3	red
4	black
5	red
6	black
7	red
8	black
...	
(28 rows omitted)	

Recall that the bet on red pays even money, 1 to 1. We defined the function `red_winnings` that returns the net winnings on one \$1 bet on red. Specifically, the function takes a color as its argument and returns 1 if the color is red. For all other colors it returns -1.

```
def red_winnings(color):
    if color == 'red':
        return 1
    else:
        return -1
```

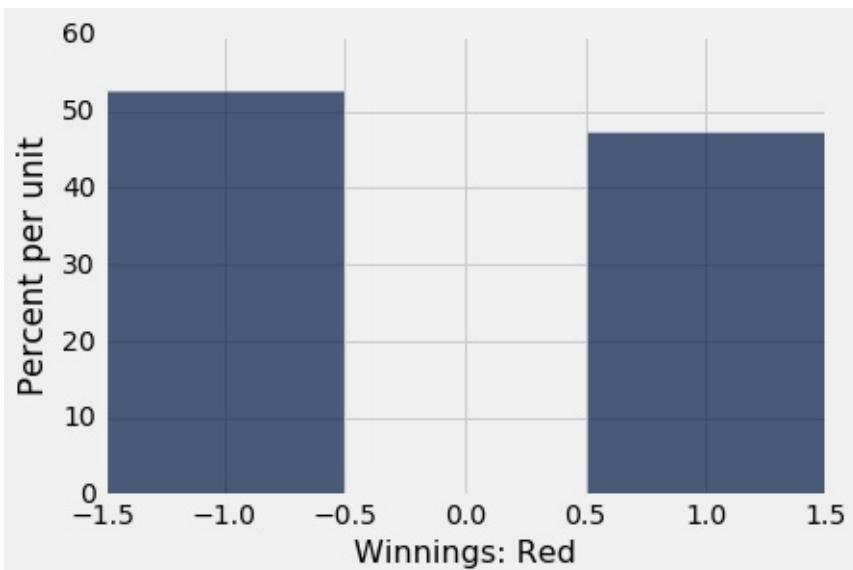
The table `red` shows each pocket's winnings on red.

```
red = wheel.with_column(
    'Winnings: Red', wheel.apply(red_winnings, 'Color')
)
red
```

Pocket	Color	Winnings: Red
0	green	-1
00	green	-1
1	red	1
2	black	-1
3	red	1
4	black	-1
5	red	1
6	black	-1
7	red	1
8	black	-1
... (28 rows omitted)		

Your net gain on one bet is one random draw from the `Winnings: Red` column. There is an 18/38 chance making \$1, and a 20/38 chance of making -\$1. This probability distribution is shown in the histogram below.

```
red.select('Winnings: Red').hist(bins=np.arange(-1.5, 1.6, 1))
```



Now suppose you bet many times on red. Your net winnings will be the sum of many draws made at random with replacement from the distribution above.

It will take a bit of math to list all the possible values of your net winnings along with all of their chances. We won't do that; instead, we will approximate the probability distribution by simulation, as we have done all along in this course.

The code below simulates your net gain if you bet \$1 on red on 400 different spins of the roulette wheel.

```

num_bets = 400
repetitions = 10000

net_gain_red = make_array()

for i in np.arange(repetitions):
    spins = red.sample(num_bets)
    new_net_gain_red = spins.column('Winnings: Red').sum()
    net_gain_red = np.append(net_gain_red, new_net_gain_red)

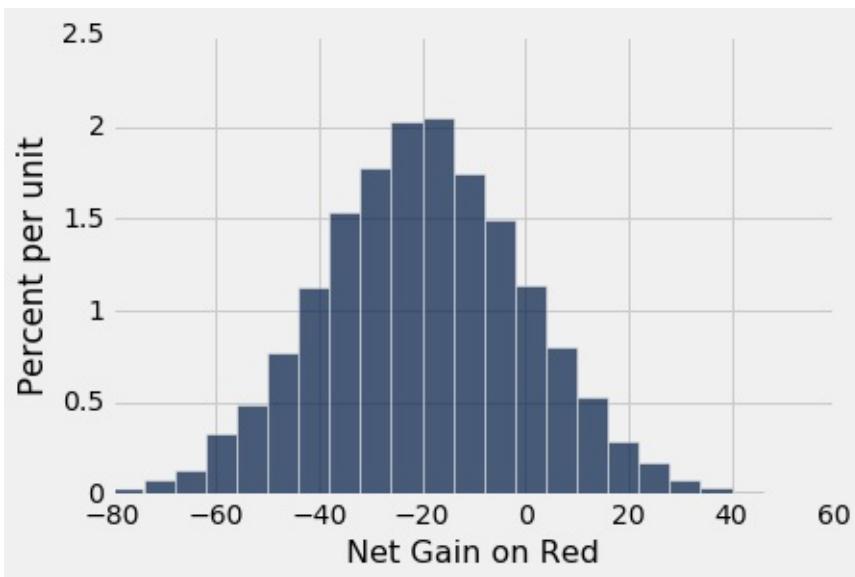
```

```

results = Table().with_column(
    'Net Gain on Red', net_gain_red
)

```

```
results.hist(bins=np.arange(-80, 50, 6))
```



That's a roughly bell shaped histogram, even though the distribution we are drawing from is nowhere near bell shaped.

Center. The distribution is centered near -\$20, roughly. To see why, note that your winnings will be \$1 on about 18/38 of the bets, and -\$1 on the remaining 20/38. So your average winnings per dollar bet will be roughly -5.26 cents:

```
average_per_bet = 1*(18/38) + (-1)*(20/38)
average_per_bet
```

```
-0.05263157894736842
```

So in 400 bets you expect that your net gain will be about -\$21:

```
400 * average_per_bet
```

```
-21.052631578947366
```

For confirmation, we can compute the mean of the 10,000 simulated net gains:

```
np.mean(results.column(0))
```

```
-20.8992
```

Spread. Run your eye along the curve starting at the center and notice that the point of inflection is near 0. On a bell shaped curve, the SD is the distance from the center to a point of inflection. The center is roughly -\$20, which means that the SD of the distribution is around \$20.

In the next section we will see where the \$20 comes from. For now, let's confirm our observation by simply calculating the SD of the 10,000 simulated net gains:

```
np.std(results.column(0))
```

```
20.043159415621083
```

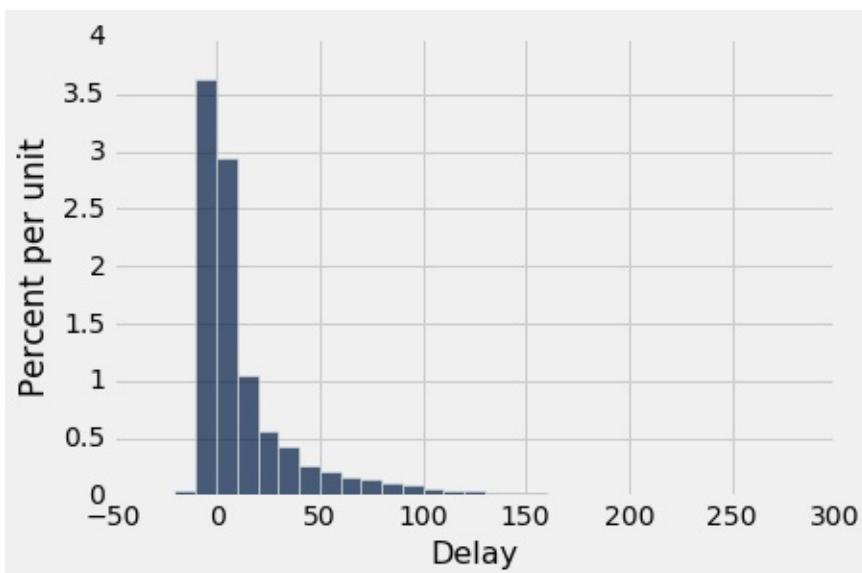
Summary. The net gain in 400 bets is the sum of the 400 amounts won on each individual bet. The probability distribution of that sum is approximately normal, with an average and an SD that we can approximate.

Average Flight Delay

The table `united` contains data on departure delays of 13,825 United Airlines domestic flights out of San Francisco airport in the summer of 2015. As we have seen before, the distribution of delays has a long right-hand tail.

```
united = Table.read_table('united_summer2015.csv')
```

```
united.select('Delay').hist(bins=np.arange(-20, 300, 10))
```



The mean delay was about 16.6 minutes and the SD was about 39.5 minutes. Notice how large the SD is, compared to the mean. Those large deviations on the right have an effect, even though they are a very small proportion of the data.

```
mean_delay = np.mean(united.column('Delay'))  
sd_delay = np.std(united.column('Delay'))  
  
mean_delay, sd_delay
```

```
(16.658155515370705, 39.480199851609314)
```

Now suppose we sampled 400 delays at random with replacement. You could sample without replacement if you like, but the results would be very similar to with-replacement sampling. If you sample a few hundred out of 13,825 without replacement, you hardly change the population each time you pull out a value.

In the sample, what could the average delay be? We expect it to be around 16 or 17, because that's the population average; but it is likely to be somewhat off. Let's see what we get by sampling. We'll work with the table `delay` that only contains the column of delays.

```
delay = united.select('Delay')
```

```
np.mean(delay.sample(400).column('Delay'))
```

```
16.68
```

The sample average varies according to how the sample comes out, so we will simulate the sampling process repeatedly and draw the empirical histogram of the sample average. That will be an approximation to the probability histogram of the sample average.

```

sample_size = 400
repetitions = 10000

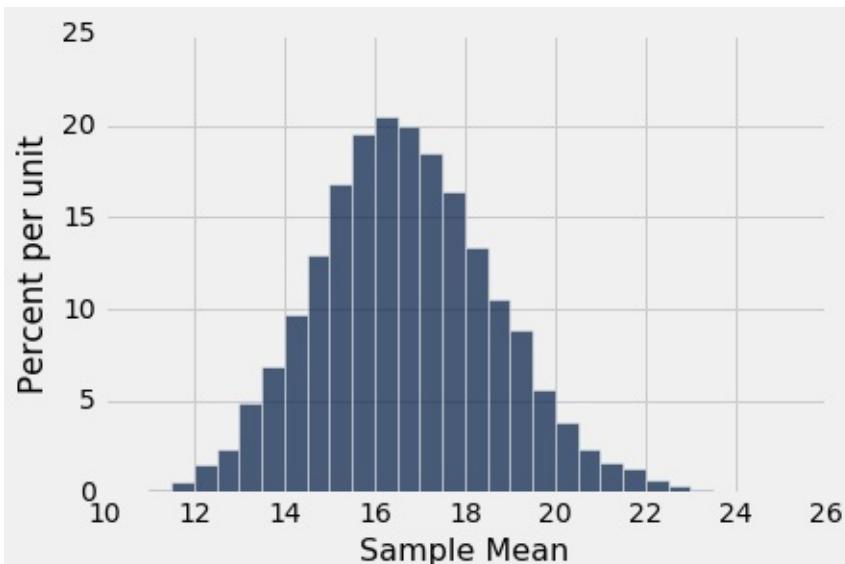
means = make_array()

for i in np.arange(repetitions):
    sample = delay.sample(sample_size)
    new_mean = np.mean(sample.column('Delay'))
    means = np.append(means, new_mean)

results = Table().with_column(
    'Sample Mean', means
)

```

```
results.hist(bins=np.arange(10, 25, 0.5))
```



Once again, we see a rough bell shape, even though we are drawing from a very skewed distribution. The bell is centered somewhere between 16 ad 17, as we expect.

Central Limit Theorem

The reason why the bell shape appears in such settings is a remarkable result of probability theory called the **Central Limit Theorem**.

The Central Limit Theorem says that the probability distribution of the sum or average of a large random sample drawn with replacement will be roughly normal, regardless of the distribution of the population from which the sample is drawn.

As we noted when we were studying Chebychev's bounds, results that can be applied to random samples *regardless of the distribution of the population* are very powerful, because in data science we rarely know the distribution of the population.

The Central Limit Theorem makes it possible to make inferences with very little knowledge about the population, provided we have a large random sample. That is why it is central to the field of statistical inference.

Proportion of Purple Flowers

Recall Mendel's probability model for the colors of the flowers of a species of pea plant. The model says that the flower colors of the plants are like draws made at random with replacement from {Purple, Purple, Purple, White}.

In a large sample of plants, about what proportion will have purple flowers? We would expect the answer to be about 0.75, the proportion purple in the model. And, because proportions are means, the Central Limit Theorem says that the distribution of the sample proportion of purple plants is roughly normal.

We can confirm this by simulation. Let's simulate the proportion of purple-flowered plants in a sample of 200 plants.

```
colors = make_array('Purple', 'Purple', 'Purple', 'White')

model = Table().with_column('Color', colors)

model
```

Color
Purple
Purple
Purple
White

```

props = make_array()

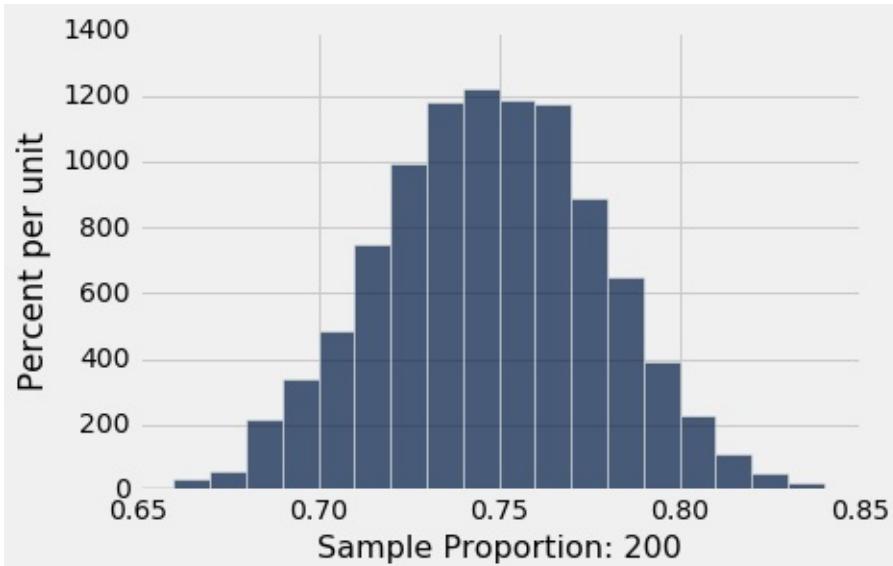
num_plants = 200
repetitions = 10000

for i in np.arange(repetitions):
    sample = model.sample(num_plants)
    new_prop = np.count_nonzero(sample.column('Color') ==
'Purple')/num_plants
    props = np.append(props, new_prop)

results = Table().with_column('Sample Proportion: 200', props)

```

```
results.hist(bins=np.arange(0.65, 0.85, 0.01))
```



There's that normal curve again, as predicted by the Central Limit Theorem, centered at around 0.75 just as you would expect.

How would this distribution change if we increased the sample size? Let's run the code again with a sample size of 800, and collect the results of simulations in the same table in which we collected simulations based on a sample size of 200. We will keep the number of repetitions the same as before so that the two columns have the same length.

```

props2 = make_array()

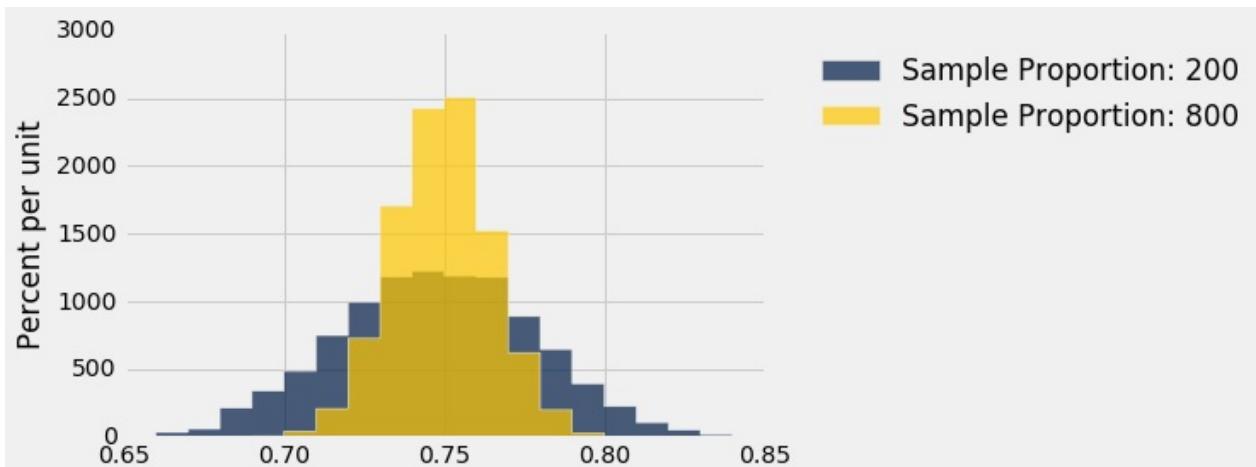
num_plants = 800

for i in np.arange(repetitions):
    sample = model.sample(num_plants)
    new_prop = np.count_nonzero(sample.column('Color') ==
'Purple')/num_plants
    props2 = np.append(props2, new_prop)

results = results.with_column('Sample Proportion: 800', props2)

```

```
results.hist(bins=np.arange(0.65, 0.85, 0.01))
```



Both distributions are approximately normal but one is narrower than the other. The proportions based on a sample size of 800 are more tightly clustered around 0.75 than those from a sample size of 200. Increasing the sample size has decreased the variability in the sample proportion.

This should not be surprising. We have leaned many times on the intuition that a larger sample size generally reduces the variability of a statistic. However, in the case of a sample average, we can *quantify* the relationship between sample size and variability.

Exactly how does the sample size affect the variability of a sample average or proportion? That is the question we will examine in the next section.

[Interact](#)

The Variability of the Sample Mean

By the Central Limit Theorem, the probability distribution of the mean of a large random sample is roughly normal. The bell curve is centered at the population mean. Some of the sample means are higher, and some lower, but the deviations from the population mean are roughly symmetric on either side, as we have seen repeatedly. Formally, probability theory shows that the sample mean is an *unbiased* estimate of the population mean.

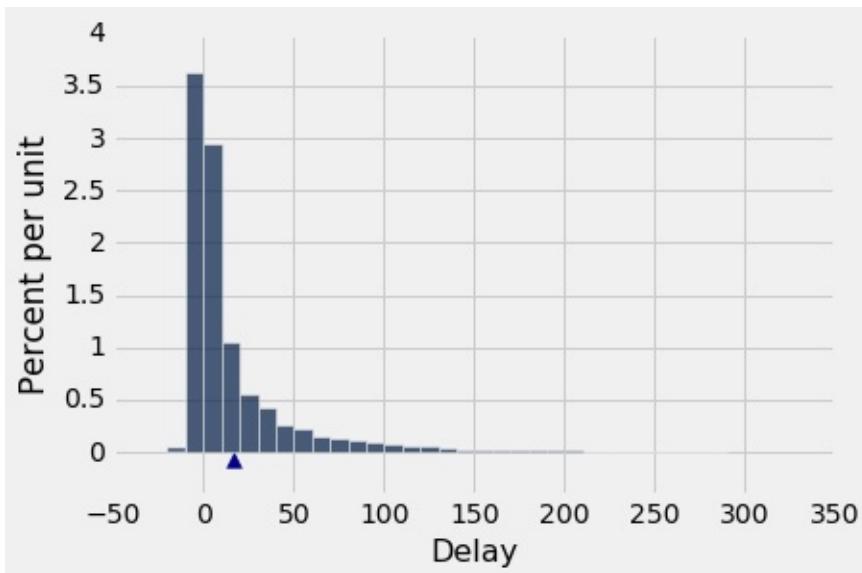
In our simulations, we also noticed that the means of larger samples tend to be more tightly clustered around the population mean than means of smaller samples. In this section, we will quantify the variability of the sample mean and develop a relation between the variability and the sample size.

Let's start with our table of flight delays. The mean delay is about 16.7 minutes, and the distribution of delays is skewed to the right.

```
united = Table.read_table('united_summer2015.csv')
delay = united.select('Delay')
```

```
pop_mean = np.mean(delay.column('Delay'))
pop_mean
```

```
16.658155515370705
```



Now let's take random samples and look at the probability distribution of the sample mean. As usual, we will use simulation to get an empirical approximation to this distribution.

We will define a function `simulate_sample_mean` to do this, because we are going to vary the sample size later. The arguments are the name of the table, the label of the column containing the variable, the sample size, and the number of simulations.

```
"""Empirical distribution of random sample means"""

def simulate_sample_mean(table, label, sample_size,
repetitions):

    means = make_array()

    for i in range(repetitions):
        new_sample = table.sample(sample_size)
        new_sample_mean = np.mean(new_sample.column(label))
        means = np.append(means, new_sample_mean)

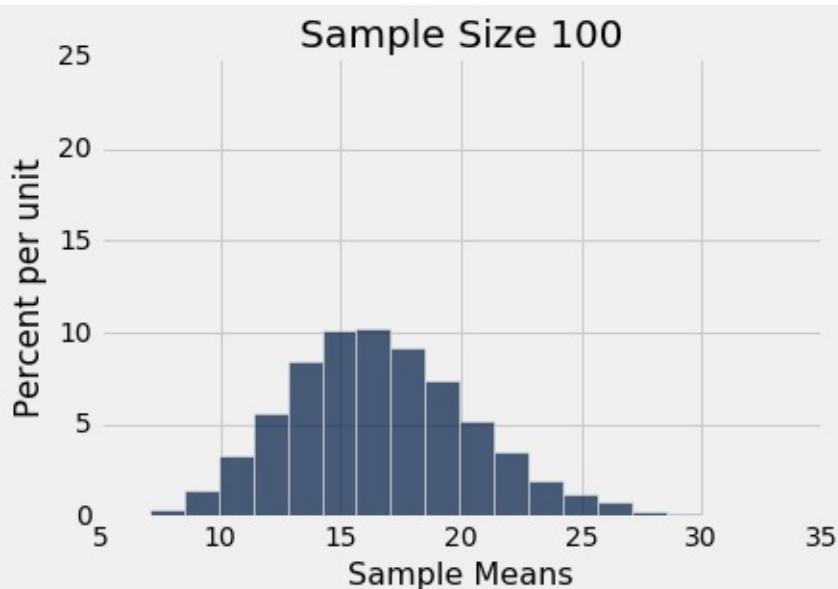
    sample_means = Table().with_column('Sample Means', means)

    # Display empirical histogram and print all relevant
    # quantities
    sample_means.hist(bins=20)
    plots.xlabel('Sample Means')
    plots.title('Sample Size ' + str(sample_size))
    print("Sample size: ", sample_size)
    print("Population mean:", np.mean(table.column(label)))
    print("Average of sample means: ", np.mean(means))
    print("Population SD:", np.std(table.column(label)))
    print("SD of sample means:", np.std(means))
```

Let us simulate the mean of a random sample of 100 delays, then of 400 delays, and finally of 625 delays. We will perform 10,000 repetitions of each of these process. The `xlim` and `ylim` lines set the axes consistently in all the plots for ease of comparison. You can just ignore those two lines of code in each cell.

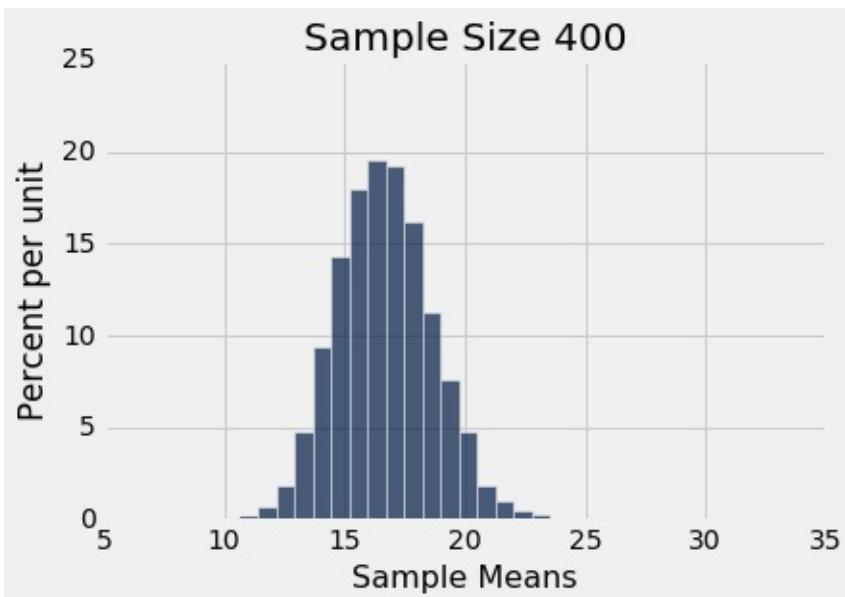
```
simulate_sample_mean(delay, 'Delay', 100, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
```

```
Sample size: 100
Population mean: 16.6581555154
Average of sample means: 16.662059
Population SD: 39.4801998516
SD of sample means: 3.90507237968
```



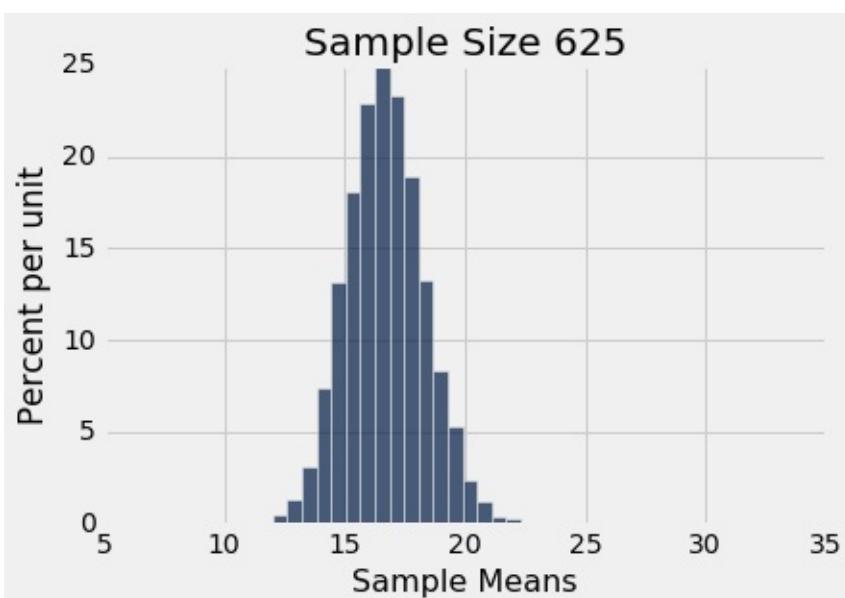
```
simulate_sample_mean(delay, 'Delay', 400, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
```

```
Sample size: 400
Population mean: 16.6581555154
Average of sample means: 16.67117625
Population SD: 39.4801998516
SD of sample means: 1.98326299651
```



```
simulate_sample_mean(delay, 'Delay', 625, 10000)
plots.xlim(5, 35)
plots.ylim(0, 0.25);
```

Sample size: 625
 Population mean: 16.6581555154
 Average of sample means: 16.68523712
 Population SD: 39.4801998516
 SD of sample means: 1.60089096006



You can see the Central Limit Theorem in action – the histograms of the sample means are roughly normal, even though the histogram of the delays themselves is far from normal.

You can also see that each of the three histograms of the sample means is centered very close to the population mean. In each case, the "average of sample means" is very close to 16.66 minutes, the population mean. Both values are provided in the printout above each histogram. As expected, the sample mean is an unbiased estimate of the population mean.

The SD of All the Sample Means

You can also see that the histograms get narrower, and hence taller, as the sample size increases. We have seen that before, but now we will pay closer attention to the measure of spread.

The SD of the population of all delays is about 40 minutes.

```
pop_sd = np.std(delay.column('Delay'))  
pop_sd
```

```
39.480199851609314
```

Take a look at the SDs in the sample mean histograms above. In all three of them, the SD of the population of delays is about 40 minutes, because all the samples were taken from the same population.

Now look at the SD of all 10,000 sample means, when the sample size is 100. That SD is about one-tenth of the population SD. When the sample size is 400, the SD of all the sample means is about one-twentieth of the population SD. When the sample size is 625, the SD of the sample means is about one-twentyfifth of the population SD.

It seems like a good idea to compare the SD of the empirical distribution of the sample means to the quantity "population SD divided by the square root of the sample size."

Here are the numerical values. For each sample size in the first column, 10,000 random samples of that size were drawn, and the 10,000 sample means were calculated. The second column contains the SD of those 10,000 sample means. The third column contains the result of the calculation "population SD divided by the square root of the sample size."

The cell takes a while to run, as it's a large simulation. But you'll soon see that it's worth the wait.

```

repetitions = 10000
sample_sizes = np.arange(25, 626, 25)

sd_means = make_array()

for n in sample_sizes:
    means = make_array()
    for i in np.arange(repetitions):
        means = np.append(means,
                          np.mean(delay.sample(n).column('Delay')))
    sd_means = np.append(sd_means, np.std(means))

sd_comparison = Table().with_columns(
    'Sample Size n', sample_sizes,
    'SD of 10,000 Sample Means', sd_means,
    'pop_sd/sqrt(n)', pop_sd/np.sqrt(sample_sizes)
)

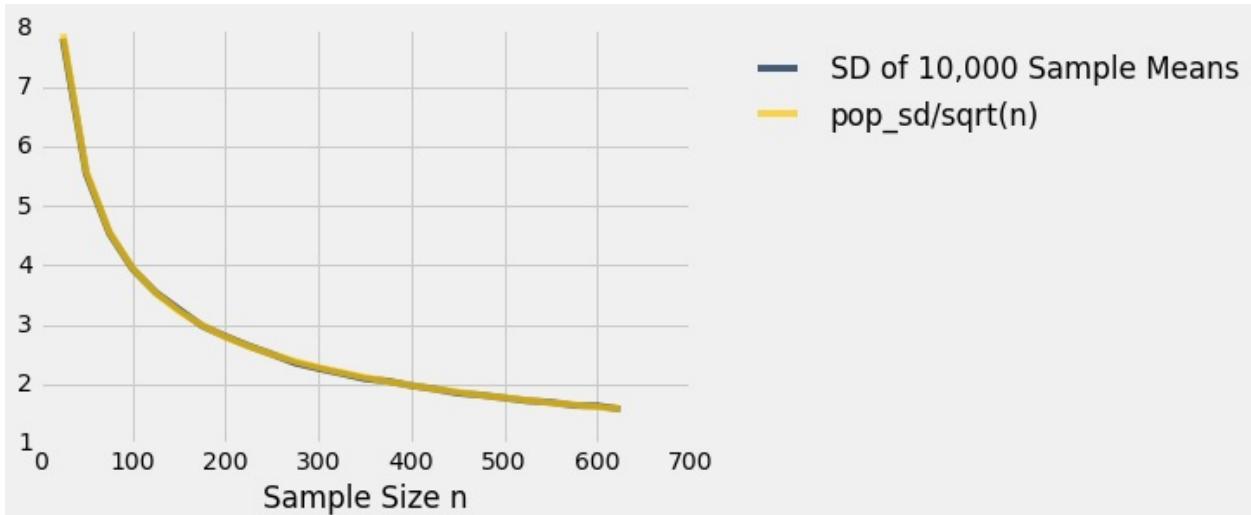
```

sd_comparison

Sample Size n	SD of 10,000 Sample Means	pop_sd/sqrt(n)
25	7.95017	7.89604
50	5.53425	5.58334
75	4.54429	4.55878
100	3.96157	3.94802
125	3.51095	3.53122
150	3.23949	3.22354
175	3.00694	2.98442
200	2.74606	2.79167
225	2.63865	2.63201
250	2.51853	2.49695
... (15 rows omitted)		

The values in the second and third columns are very close. If we plot each of those columns with the sample size on the horizontal axis, the two graphs are essentially indistinguishable.

```
sd_comparison.plot('Sample Size n')
```



There really are two curves there. But they are so close to each other that it looks as though there is just one.

What we are seeing is an instance of a general result. Remember that the graph above is based on 10,000 replications for each sample size. But there are many more than 10,000 samples of each size. The probability distribution of the sample mean is based on the means of *all possible samples* of a fixed size.

Fix a sample size. If the samples are drawn at random with replacement from the population, then

$$\text{SD of all possible sample means} = \frac{\text{Population SD}}{\sqrt{\text{sample size}}}$$

This is the standard deviation of the averages of all the possible samples that could be drawn. ***It measures roughly how far off the sample means are from the population mean.**

The Central Limit Theorem for the Sample Mean

If you draw a large random sample with replacement from a population, then, regardless of the distribution of the population, the probability distribution of the sample mean is roughly normal, centered at the population mean, with an SD equal to the population SD divided by the square root of the sample size.

The Accuracy of the Sample Mean

The SD of all possible sample means measures how variable the sample mean can be. As such, it is taken as a measure of the accuracy of the sample mean as an estimate of the population mean. The smaller the SD, the more accurate the estimate.

The formula shows that:

- The population size doesn't affect the accuracy of the sample mean. The population size doesn't appear anywhere in the formula.
- The population SD is a constant; it's the same for every sample drawn from the population. The sample size can be varied. Because the sample size appears in the denominator, the variability of the sample mean *decreases* as the sample size increases, and hence the accuracy increases.

The Square Root Law

From the table of SD comparisons, you can see that the SD of the means of random samples of 25 flight delays is about 8 minutes. If you multiply the sample size by 4, you'll get samples of size 100. The SD of the means of all of those samples is about 4 minutes. That's smaller than 8 minutes, but it's not 4 times as small; it's only 2 times as small. That's because the sample size in the denominator has a square root over it. The sample size increased by a factor of 4, but the SD went down by a factor of $2 = \sqrt{4}$. In other words, the accuracy went up by a factor of $2 = \sqrt{4}$.

In general, when you multiply the sample size by a factor, the accuracy of the sample mean goes up by the square root of that factor.

So to increase accuracy by a factor of 10, you have to multiply sample size by a factor of 100. Accuracy doesn't come cheap!

[Interact](#)

Choosing a Sample Size

Candidate A is contesting an election. A polling organization wants to estimate the proportion of voters who will vote for her. Let's suppose that they plan to take a simple random sample of voters, though in reality their method of sampling would be more complex. How can they decide how large their sample should be, to get a desired level of accuracy?

We are now in a position to answer this question, after making a few assumptions:

- The population of voters is very large and that therefore we can just as well assume that the random sample will be drawn with replacement.
- The polling organization will make its estimate by constructing an approximate 95% confidence interval for the percent of voters who will vote for Candidate A.
- The desired level of accuracy is that the width of the interval should be no more than 1%. That's pretty accurate! For example, the confidence interval (33.2%, 34%) would be fine but (33.2%, 35%) would not.

We will work with the sample proportion of voters for Candidate A. Recall that a proportion is a mean, when the values in the population are only 0 (the type of individual you are not counting) or 1 (the type of individual you are counting).

Width of Confidence Interval

If we had a random sample, we could go about using the bootstrap to construct a confidence interval for the percent of voters for Candidate A. But we don't have a sample yet – we are trying to find out how big the sample has to be so that our confidence interval is as narrow as we want it to be.

In situations like this, it helps to see what theory predicts.

The Central Limit Theorem says that the probabilities for the sample proportion are roughly normally distributed, centered at the population proportion of 1's, with an SD equal to the SD of the population of 0's and 1's divided by the square root of the sample size.

So the confidence interval will still be the "middle 95%" of a normal distribution, even though we can't pick off the ends as the 2.5th and 97.5th percentiles of bootstrapped proportions.

Is there another way to find how wide the interval would be? Yes, because we know that for normally distributed variables, the interval "center \pm 2 SDs" contains 95% of the data.

The confidence interval will stretch for 2 SDs of the sample proportion, on either side of the center. So the width of the interval will be 4 SDs of the sample proportion.

We are willing to tolerate a width of $1\% = 0.01$. So, using the formula developed in the last section,

$$4 \times \frac{\text{SD of the 0-1 population}}{\sqrt{\text{sample size}}} \leq 0.01$$

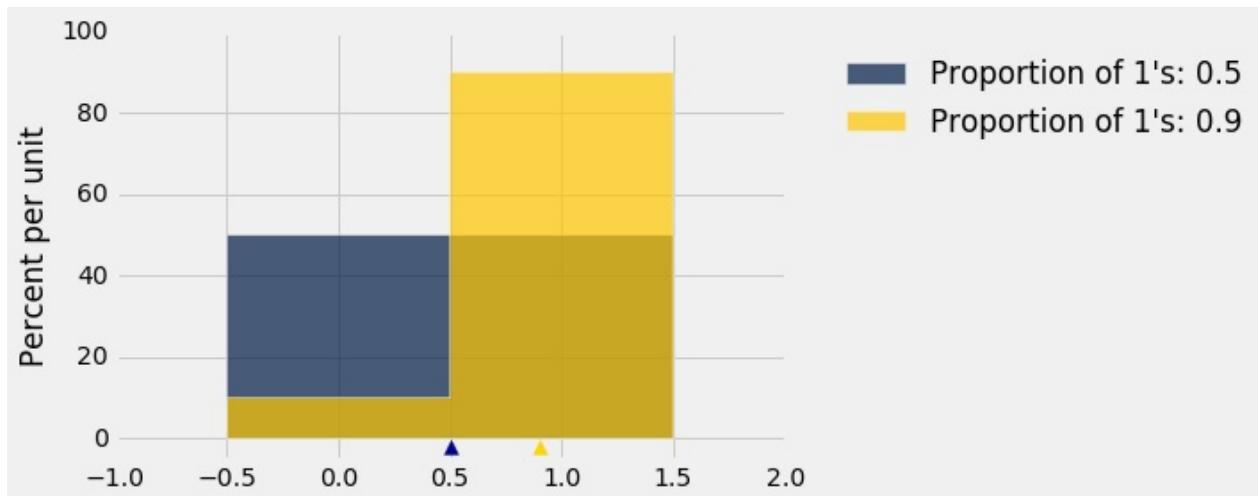
So

$$\sqrt{\text{sample size}} \geq 4 \times \frac{\text{SD of the 0-1 population}}{0.01}$$

The SD of a collection of 0's and 1's

If we knew the SD of the population, we'd be done. We could calculate the square root of the sample size, and then take the square to get the sample size. But we don't know the SD of the population. The population consists of 1 for each voter for Candidate A, and 0 for all other voters, and *we don't know what proportion of each kind there are*. That's what we're trying to estimate.

So are we stuck? No, because we can *bound* the SD of the population. Here are histograms of two such distributions, one for an equal proportion of 1's and 0's, and one with 90% 1's and 10% 0's. Which one has the bigger SD?



Remember that the possible values in the population are only 0 and 1.

The blue histogram (50% 1's and 50% 0's) has more spread than the gold. The mean is 0.5. Half the deviations from mean are equal to 0.5 and the other half equal to -0.5, so the SD is 0.5.

In the gold histogram, all of the area is being squished up around 1, leading to less spread. 90% of the deviations are small: 0.1. The other 10% are -0.9 which is large, but overall the spread is smaller than in the blue histogram.

The same observation would hold if we varied the proportion of 1's or let the proportion of 0's be larger than the proportion of 1's. Let's check this by calculating the SDs of populations of 10 elements that only consist of 0's and 1's, in varying proportions. The function `np.ones` is useful for this. It takes a positive integer as its argument and returns an array consisting of that many 1's.

```
sd = make_array()
for i in np.arange(1, 10, 1):
    # Create an array of i 1's and (10-i) 0's
    population = np.append(np.ones(i), 1-np.ones(10-i))
    sd = np.append(sd, np.std(population))

zero_one_sds = Table().with_columns(
    "Population Proportion of 1's", np.arange(0.1, 1, 0.1),
    "Population SD", sd
)

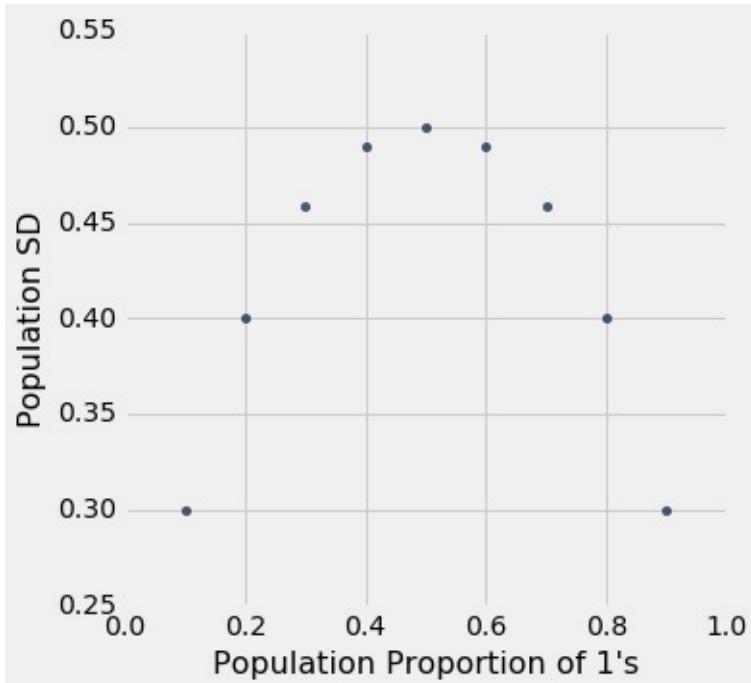
zero_one_sds
```

Population Proportion of 1's	Population SD
0.1	0.3
0.2	0.4
0.3	0.458258
0.4	0.489898
0.5	0.5
0.6	0.489898
0.7	0.458258
0.8	0.4
0.9	0.3

Not surprisingly, the SD of a population with 10% 1's and 90% 0's is the same as that of a population with 90% 1's and 10% 0's. That's because you switch the bars of one histogram to get the other; there is no change in spread.

More importantly for our purposes, the SD increases as the proportion of 1's increases, until the proportion of 1's is 0.5; then it starts to decrease symmetrically.

```
zero_one_sds.scatter("Population Proportion of 1's")
```



Summary: The SD of a population of 1's and 0's is at most 0.5. That's the value of the SD when 50% of the population is coded 1 and the other 50% are coded 0.

The Sample Size

We know that

$$\sqrt{\text{sample size}} \geq 4 \times \frac{\text{SD of the 0-1 population}}{0.01}$$

and that the SD of the 0-1 population is at most 0.5, regardless of the proportion of 1's in the population. So it is safe to take

$$\sqrt{\text{sample size}} \geq 4 \times \frac{0.5}{0.01} = 200$$

So the sample size should be at least $200^2 = 40,000$. That's an enormous sample! But that's what you need if you want to guarantee great accuracy with high confidence no matter what the population looks like.

[Interact](#)

Prediction

An important aspect of data science is to find out what data can tell us about the future. What do data about climate and pollution say about temperatures a few decades from now? Based on a person's internet profile, which websites are likely to interest them? How can a patient's medical history be used to judge how well he or she will respond to a treatment?

To answer such questions, data scientists have developed methods for making *predictions*. In this chapter we will study one of the most commonly used ways of predicting the value of one variable based on the value of another.

The foundations of the method were laid by [Sir Francis Galton](#). As we saw in Section 7.1, Galton studied how physical characteristics are passed down from one generation to the next. Among his best known work is the prediction of the heights of adults based on the heights of their parents. We have studied the dataset that Galton collected for this. The table `heights` contains his data on the midparent height and child's height (all in inches) for a population of 934 adult "children".

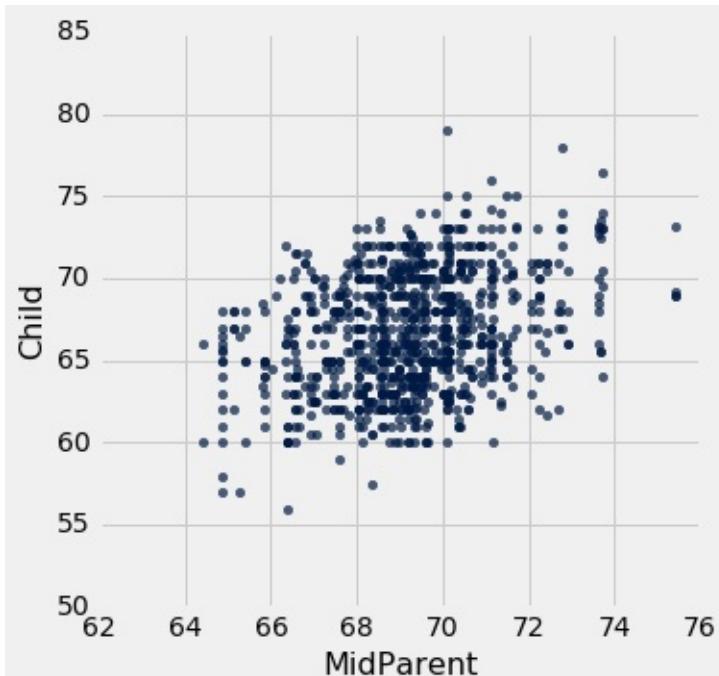
```
# Galton's data on heights of parents and their adult children
galton = Table.read_table('galton.csv')
heights = Table().with_columns(
    'MidParent', galton.column('midparentHeight'),
    'Child', galton.column('childHeight')
)
```

```
heights
```

MidParent	Child
75.43	73.2
75.43	69.2
75.43	69
75.43	69
73.66	73.5
73.66	72.5
73.66	65.5
73.66	65.5
72.06	71
72.06	68

... (924 rows omitted)

```
heights.scatter('MidParent')
```



The primary reason for collecting the data was to be able to predict the adult height of a child born to parents similar to those in the dataset. We made these predictions in Section 7.1, after noticing the positive association between the two variables.

Our approach was to base the prediction on all the points that correspond to a midparent height of around the midparent height of the new person. To do this, we wrote a function called `predict_child` which takes a midparent height as its argument and returns the

average height of all the children who had midparent heights within half an inch of the argument.

```
def predict_child(mpht):
    """Return a prediction of the height of a child
    whose parents have a midparent height of mpht.

    The prediction is the average height of the children
    whose midparent height is in the range mpht plus or minus
    0.5 inches.
    """

    close_points = heights.where('MidParent', are.between(mpht-
0.5, mpht + 0.5))
    return close_points.column('Child').mean()
```

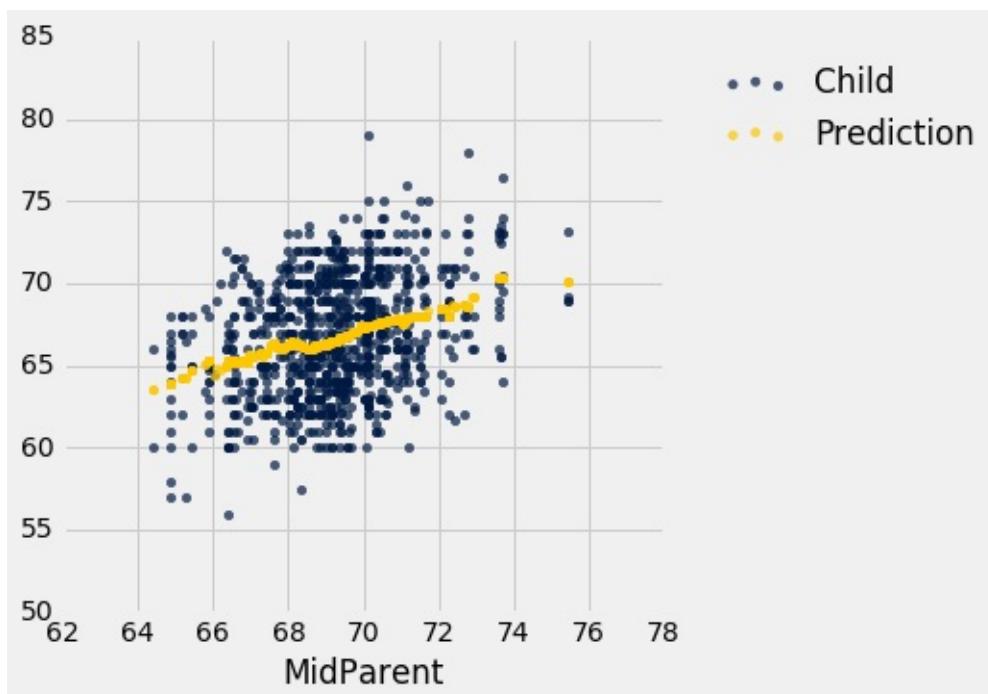
We applied the function to the column of `Midparent` heights, visualized our results.

```
# Apply predict_child to all the midparent heights

heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
)

# Draw the original scatter plot along with the predicted values

heights_with_predictions.scatter('MidParent')
```



The prediction at a given midparent height lies roughly at the center of the vertical strip of points at the given height. This method of prediction is called *regression*. Later in this chapter we will see where this term came from. We will also see whether we can avoid our arbitrary definitions of "closeness" being "within 0.5 inches". But first we will develop a measure that can be used in many settings to decide how good one variable will be as a predictor of another.

[Interact](#)

Correlation

In this section we will develop a measure of how tightly clustered a scatter diagram is about a straight line. Formally, this is called measuring *linear association*.

The table `hybrid` contains data on hybrid passenger cars sold in the United States from 1997 to 2013. The data were adapted from the online data archive of [Prof. Larry Winner](#) of the University of Florida. The columns:

- `vehicle` : model of the car
- `year` : year of manufacture
- `msrp` : manufacturer's suggested retail price in 2013 dollars
- `acceleration` : acceleration rate in km per hour per second
- `mpg` : fuel economy in miles per gallon
- `class` : the model's class.

```
hybrid = Table.read_table('hybrid.csv')
```

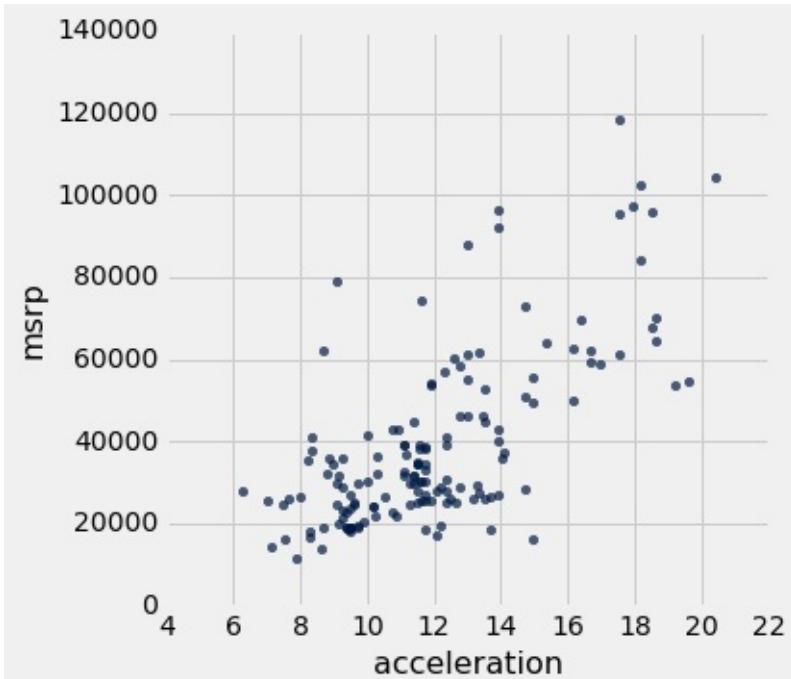
```
hybrid
```

<code>vehicle</code>	<code>year</code>	<code>msrp</code>	<code>acceleration</code>	<code>mpg</code>	<code>class</code>
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 graph below is a scatter plot of `msrp` *versus* `acceleration`. That means `msrp` is plotted on the vertical axis and `acceleration` on the horizontal.

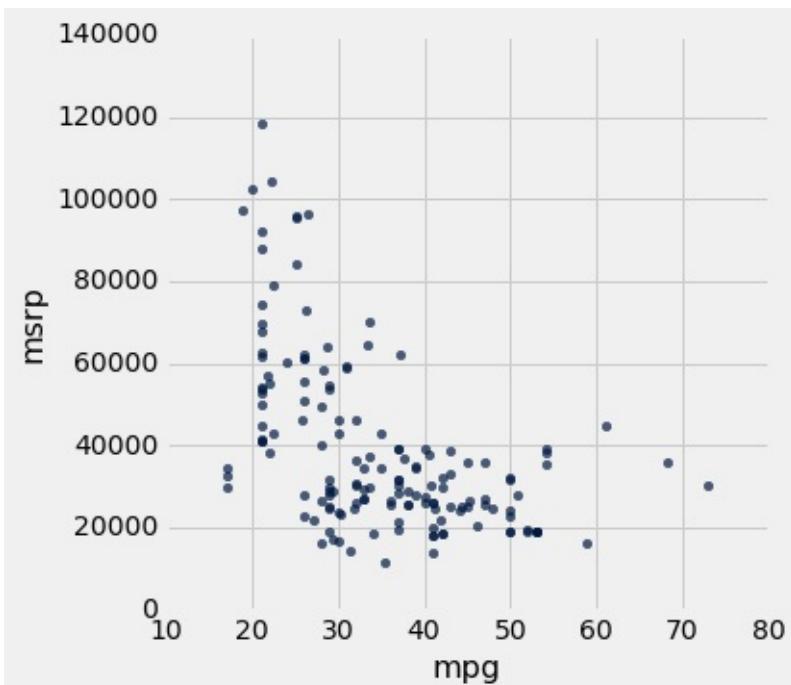
```
hybrid.scatter('acceleration', 'msrp')
```



Notice the positive association. 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.

The scatter diagram of MSRP versus mileage shows a negative association. 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 plot showed, those were also the cars that tended to cost more.

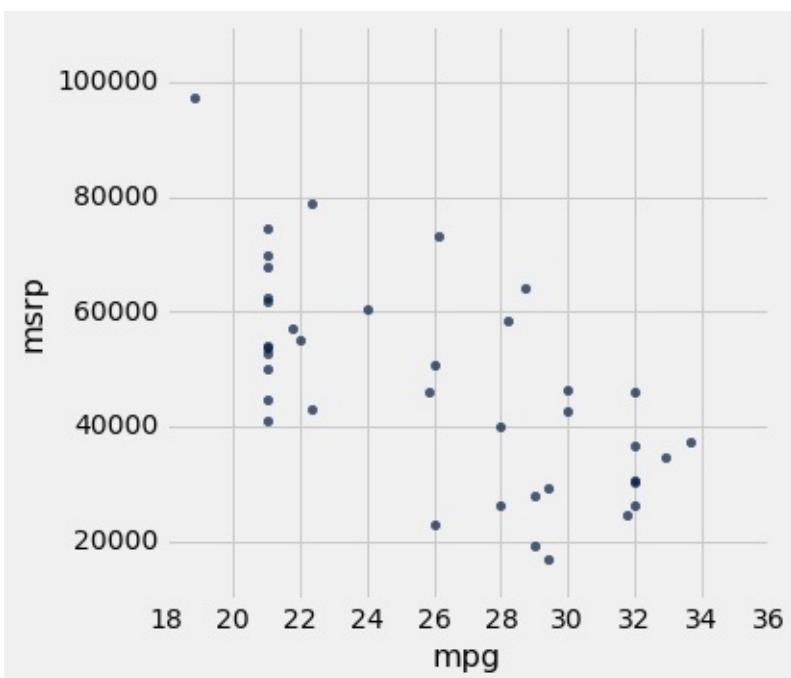
```
hybrid.scatter('mpg', 'msrp')
```



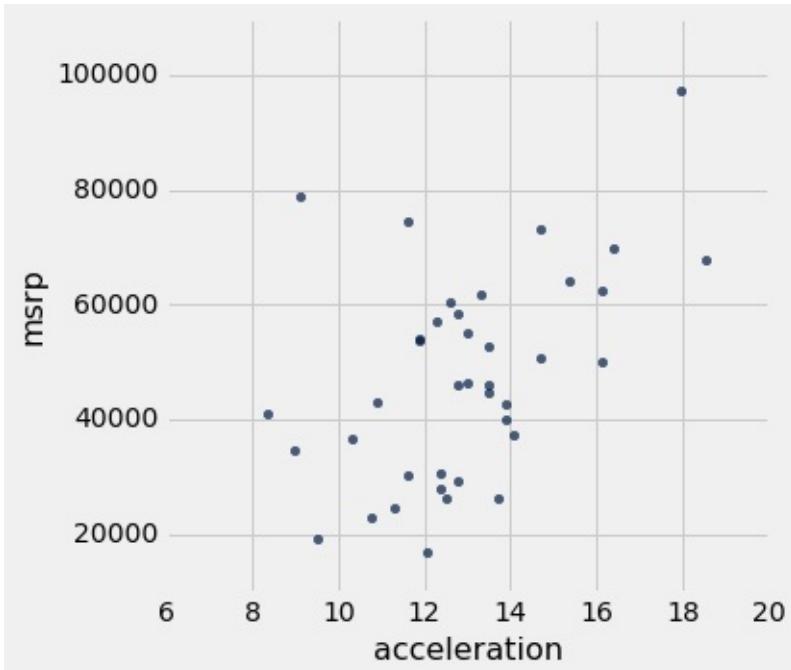
Along with the negative association, the scatter diagram of price versus efficiency shows a non-linear relation between the two variables. The points appear to be clustered around a curve, not around a straight line.

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.

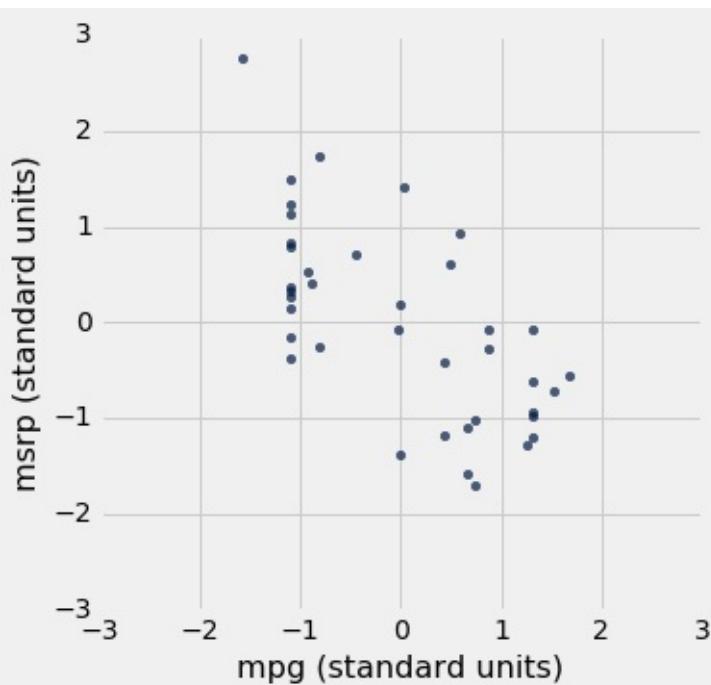
Indeed, we could plot all the variables in standard units and the plots would look the same. This gives us a way to compare the degree of linearity in two scatter diagrams.

Recall that in an earlier section we defined the function `standard_units` to convert an array of numbers to standard units.

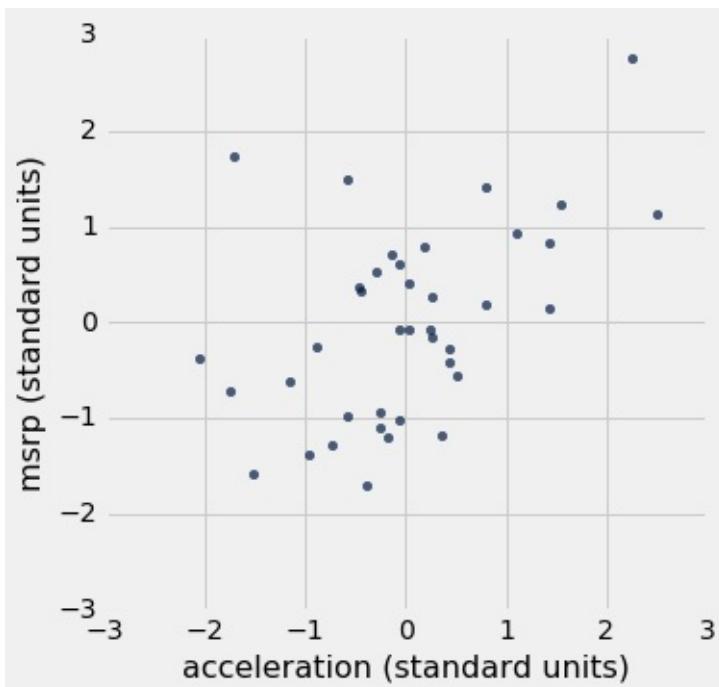
```
def standard_units(any_numbers):
    "Convert any array of numbers to standard units."
    return (any_numbers -
        np.mean(any_numbers))/np.std(any_numbers)
```

We can use this function to re-draw 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);
```



```
Table().with_columns(  
    'acceleration (standard units)',  
    standard_units(suv.column('acceleration')),  
    'msrp (standard units)',  
    standard_units(suv.column('msrp'))  
).scatter(0, 1)  
plots.xlim(-3, 3)  
plots.ylim(-3, 3);
```



The associations that we see in these figures are the same as those we saw before. Also, because the two scatter diagrams are now 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 just observe by simulation.

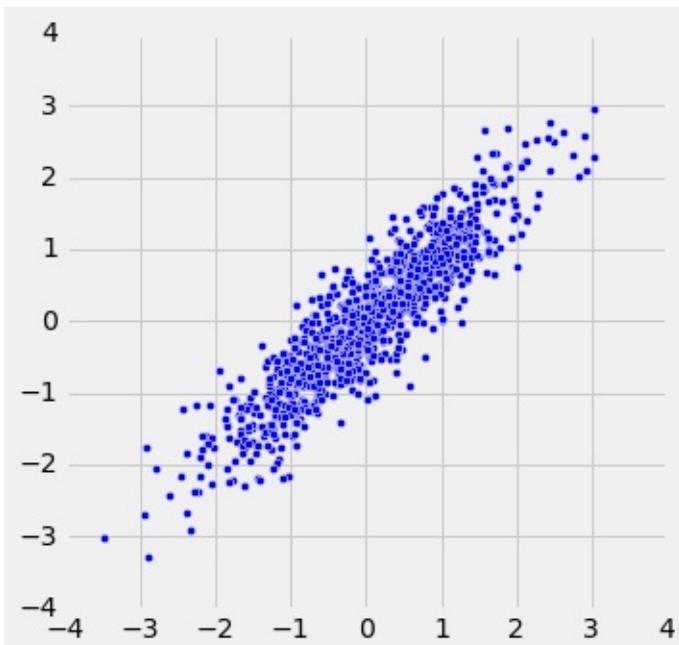
- The correlation coefficient r is a number between -1 and 1 .
- r measures the extent to which the scatter plot clusters around a straight line.
- $r = 1$ if the scatter diagram is a perfect straight line sloping upwards, and $r = -1$ if the scatter diagram is a perfect straight line sloping downwards.

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 .

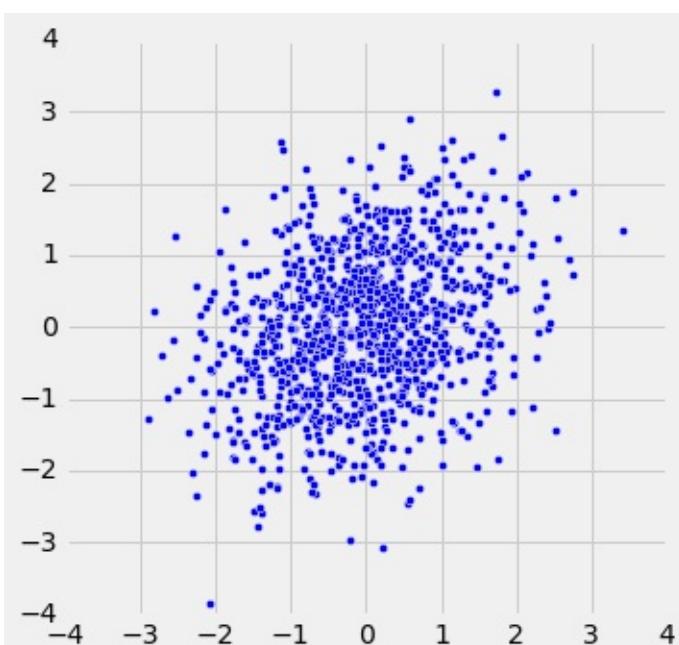
Call `r_scatter` a few times, with different values of r as the argument, and see how the scatter plot changes.

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*.

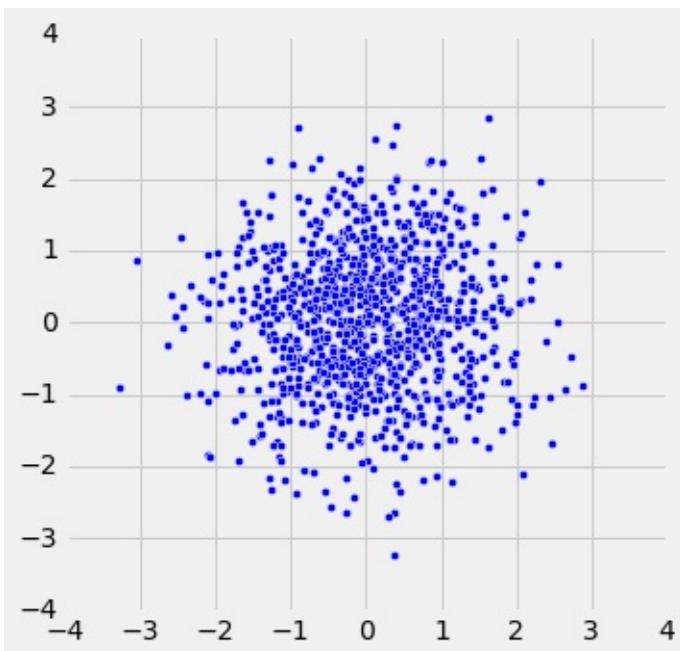
```
r_scatter(0.9)
```



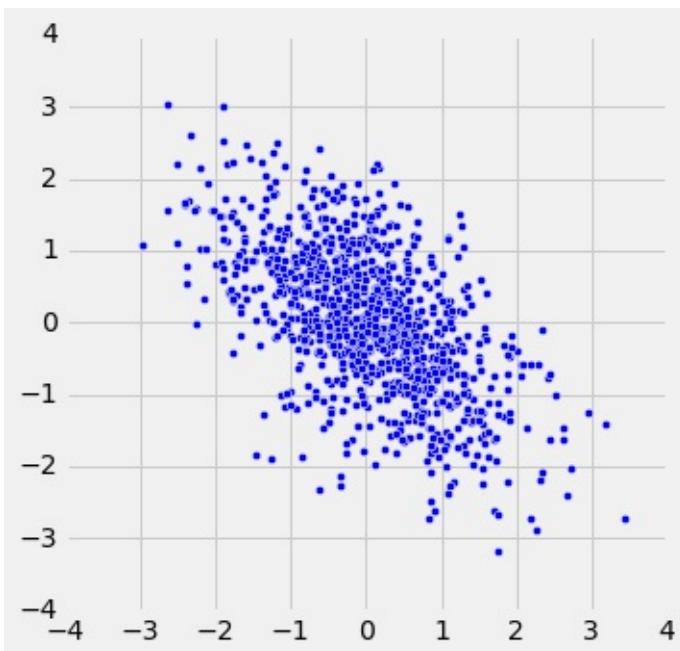
```
r_scatter(0.25)
```



```
r_scatter(0)
```



`r_scatter(-0.55)`



Calculating r

The formula for r is not apparent from our observations so far. It has a mathematical basis that is outside the scope of this class. However, as you will see, 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.

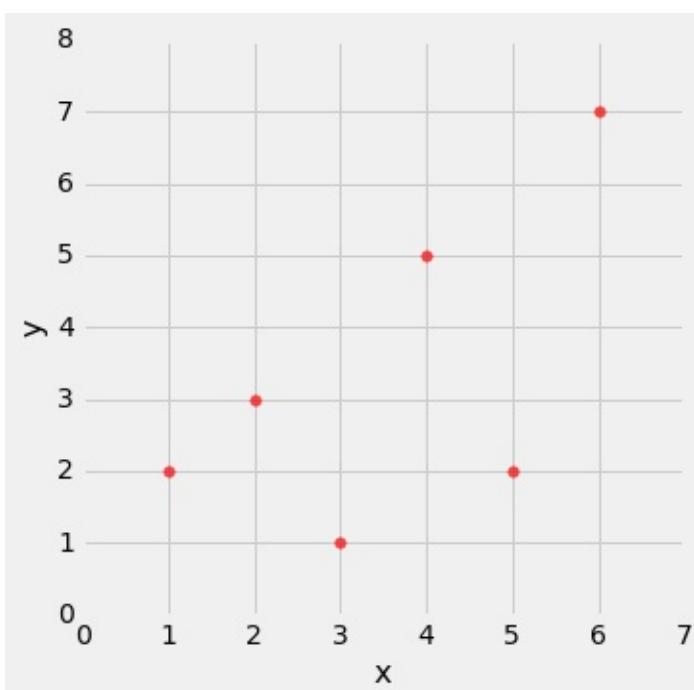
Here are the steps in the calculation. We will apply the steps to a simple table of values of x and y .

```
x = np.arange(1, 7, 1)
y = make_array(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.column(2) * t_su.column(3))
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
```

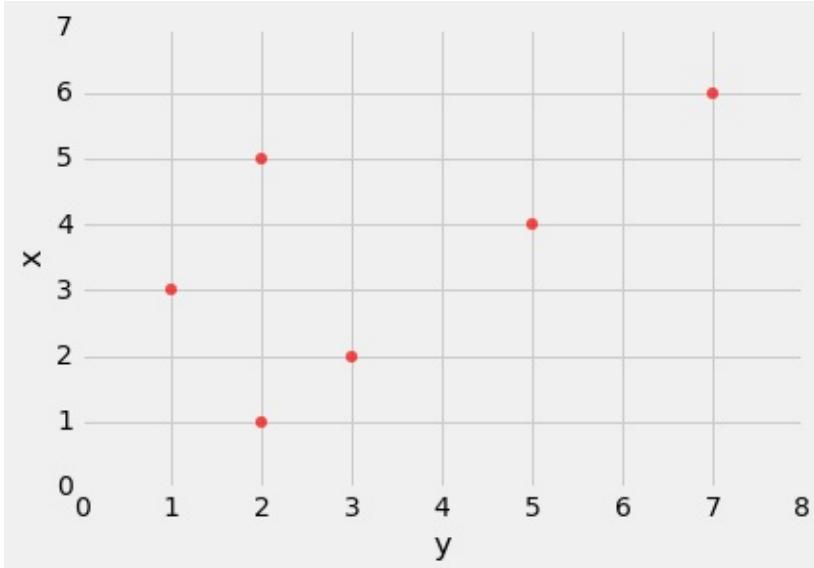
As expected, r is positive but not equal to 1.

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')
```



The correlation function

We are going to be calculating correlations repeatedly, so it will help to define a function that computes it by performing all the steps described above. Let's define a function `correlation` that takes a table and the labels of two columns in the table. The function returns r , the mean of the products of those column values in standard units.

```
def correlation(t, x, y):
    return
    np.mean(standard_units(t.column(x))*standard_units(t.column(y)))
```

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
```

As we noticed, the order in which the variables are specified doesn't matter.

```
correlation(t, 'y', 'x')
```

```
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).

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.

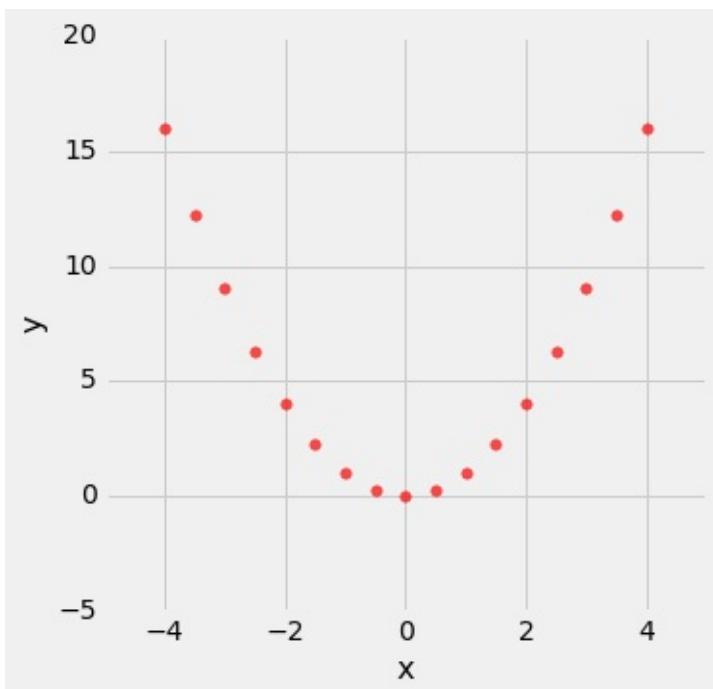
Association is not Causation

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*

Correlation measures only one kind of association – linear. 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.

```
new_x = np.arange(-4, 4.1, 0.5)
nonlinear = Table().with_columns(
    'x', new_x,
    'y', new_x**2
)
nonlinear.scatter('x', 'y', s=30, color='r')
```



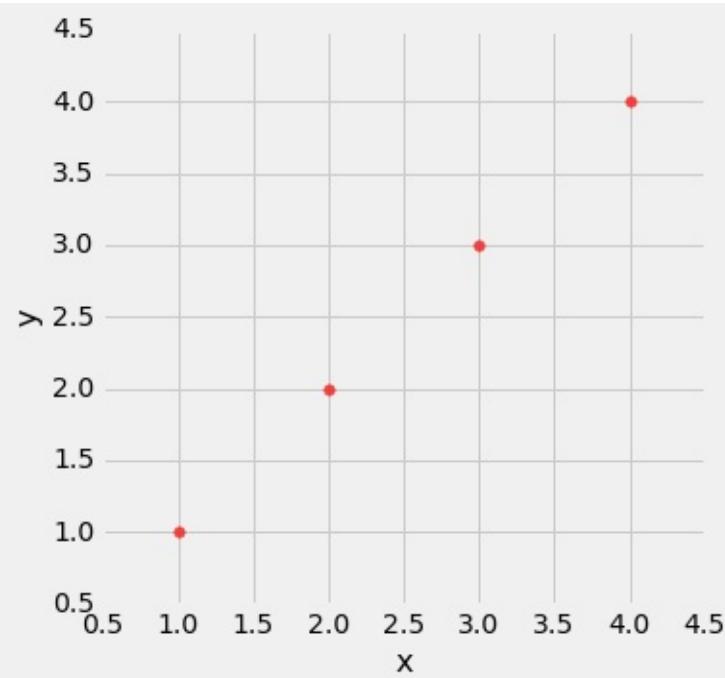
```
correlation(nonlinear, 'x', 'y')
```

```
0.0
```

Correlation is Affected by Outliers

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.

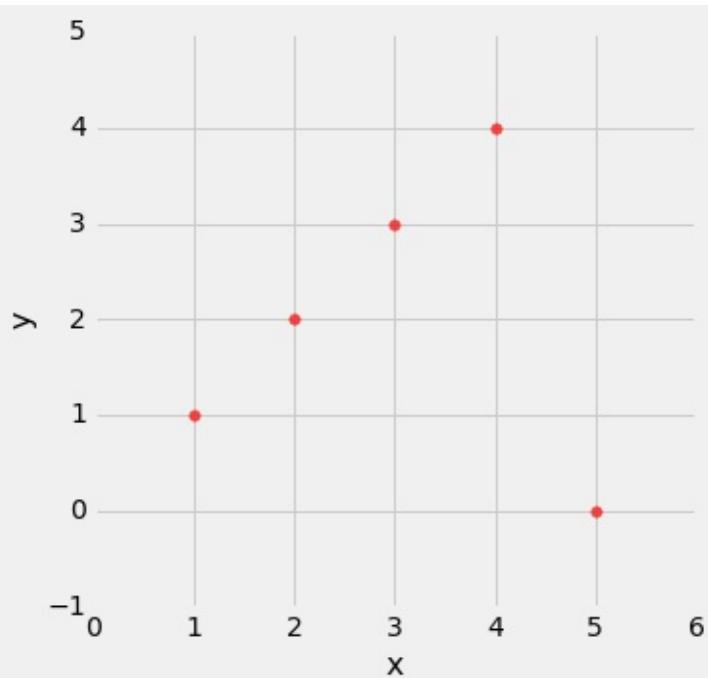
```
line = Table().with_columns(  
    'x', make_array(1, 2, 3, 4),  
    'y', make_array(1, 2, 3, 4)  
)  
line.scatter('x', 'y', s=30, color='r')
```



```
correlation(line, 'x', 'y')
```

```
1.0
```

```
outlier = Table().with_columns(
    'x', make_array(1, 2, 3, 4, 5),
    'y', make_array(1, 2, 3, 4, 0)
)
outlier.scatter('x', 'y', s=30, color='r')
```



```
correlation(outlier, 'x', 'y')
```

0.0

Ecological Correlations Should be Interpreted with Care

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.

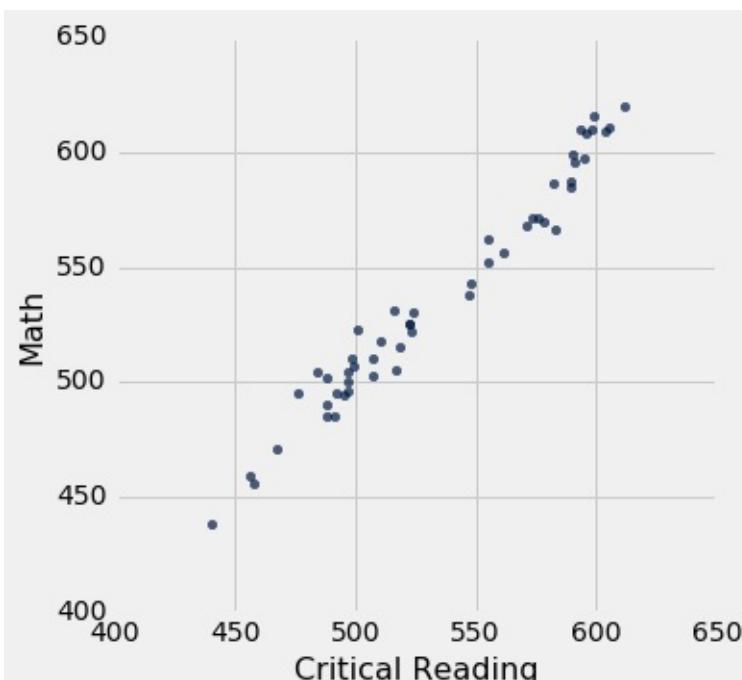
```
sat2014 = Table.read_table('sat2014.csv').sort('State')
sat2014
```

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

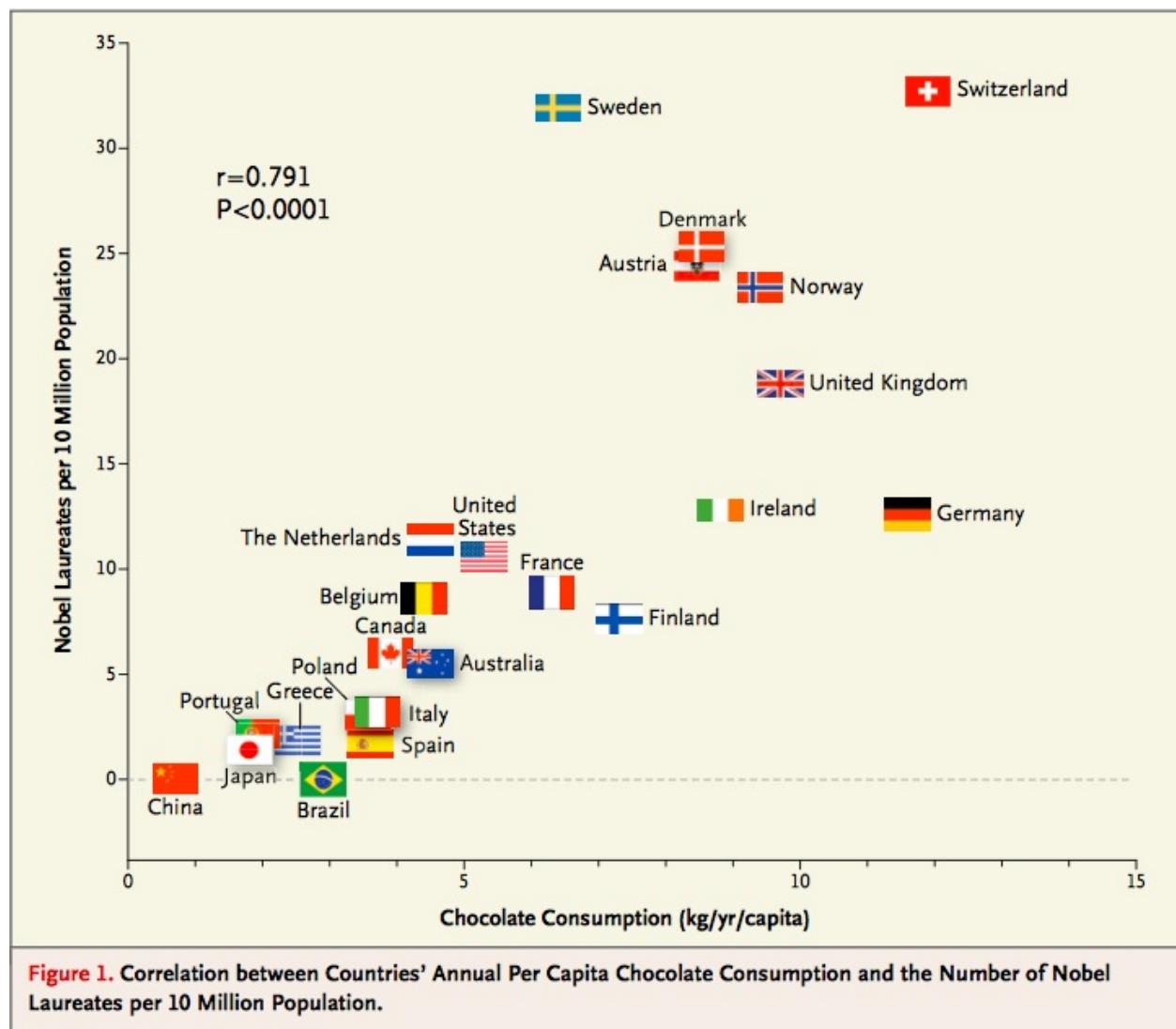
That's an extremely high correlation. But it's important to note that this does not reflect the strength of the relation between the Math and Critical Reading scores of *students*.

The data consist of average scores in each state. But 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 whereas [others](#) were more relaxed. You are welcome to make your own decision! The following graph, provided in the paper, should motivate you to go and take a look.



[Interact](#)

The Regression Line

The correlation coefficient r doesn't just measure how clustered the points in a scatter plot are about a straight line. It also helps identify the straight line about which the points are clustered. In this section we will retrace the path that Galton and Pearson took to discover that line.

Galton's data on the heights of parents and their adult children showed a linear association. The linearity was confirmed when our predictions of the children's heights based on the midparent heights roughly followed a straight line.

```
galton = Table.read_table('galton.csv')

heights = Table().with_columns(
    'MidParent', galton.column('midparentHeight'),
    'Child', galton.column('childHeight')
)
```

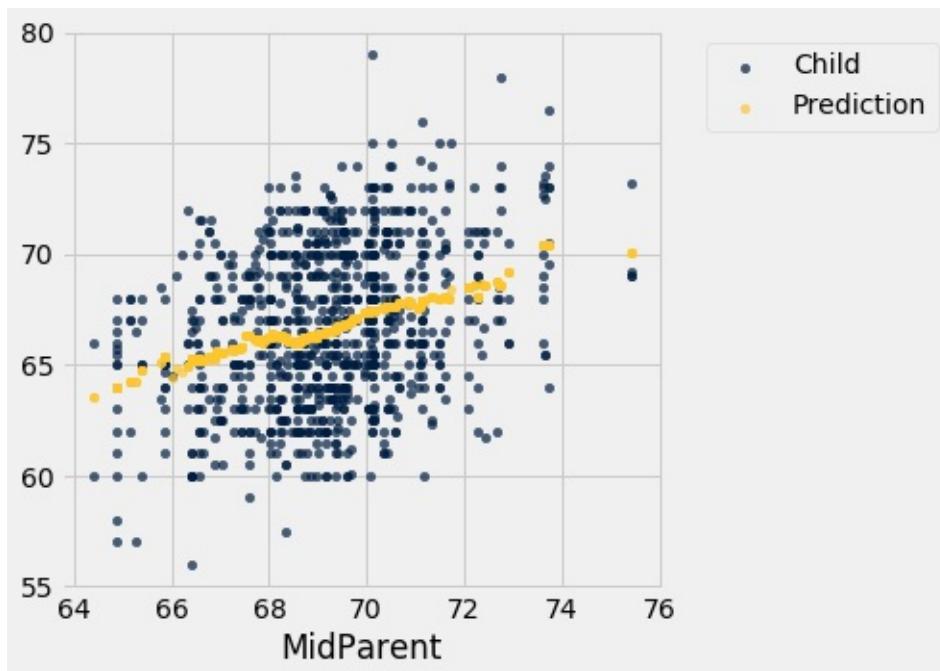
```
def predict_child(mpht):
    """Return a prediction of the height of a child
    whose parents have a midparent height of mpht.

    The prediction is the average height of the children
    whose midparent height is in the range mpht plus or minus
    0.5 inches.
    """

    close_points = heights.where('MidParent', are.between(mpht-
        0.5, mpht + 0.5))
    return close_points.column('Child').mean()
```

```
heights_with_predictions = heights.with_column(
    'Prediction', heights.apply(predict_child, 'MidParent')
)
```

```
heights_with_predictions.scatter('MidParent')
```



Measuring in Standard Units

Let's see if we can find a way to identify this line. First, notice that linear association doesn't depend on the units of measurement – we might as well measure both variables in standard units.

```
def standard_units(xyz):
    "Convert any array of numbers to standard units."
    return (xyz - np.mean(xyz))/np.std(xyz)
```

```
heights_SU = Table().with_columns(
    'MidParent SU', standard_units(heights.column('MidParent')),
    'Child SU', standard_units(heights.column('Child'))
)
heights_SU
```

MidParent SU	Child SU
3.45465	1.80416
3.45465	0.686005
3.45465	0.630097
3.45465	0.630097
2.47209	1.88802
2.47209	1.60848
2.47209	-0.348285
2.47209	-0.348285
1.58389	1.18917
1.58389	0.350559

... (924 rows omitted)

On this scale, we can calculate our predictions exactly as before. But first we have to figure out how to convert our old definition of "close" points to a value on the new scale. We had said that midparent heights were "close" if they were within 0.5 inches of each other. Since standard units measure distances in units of SDs, we have to figure out how many SDs of midparent height correspond to 0.5 inches.

One SD of midparent heights is about 1.8 inches. So 0.5 inches is about 0.28 SDs.

```
sd_midparent = np.std(heights.column(0))
sd_midparent
```

```
1.8014050969207571
```

```
0.5/sd_midparent
```

```
0.27756111096536701
```

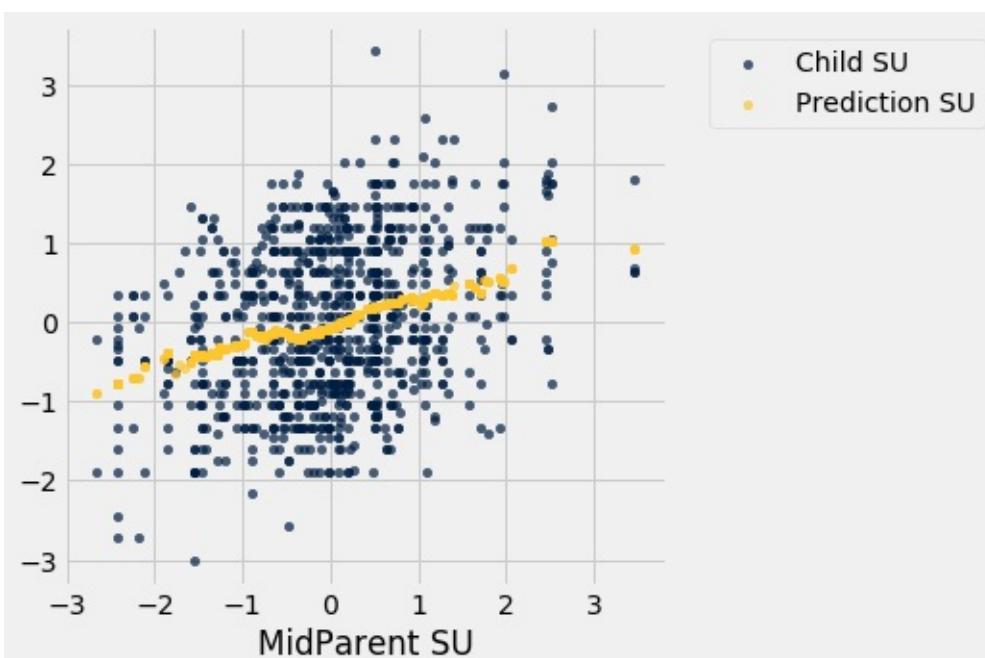
We are now ready to modify our prediction function to make predictions on the standard units scale. All that has changed is that we are using the table of values in standard units, and defining "close" as above.

```
def predict_child_su(mpht_su):
    """Return a prediction of the height (in standard units) of
    a child
    whose parents have a midparent height of mpht_su in standard
    units.

    """
    close = 0.5/sd_midparent
    close_points = heights_SU.where('MidParent SU',
    are.between(mpht_su-close, mpht_su + close))
    return close_points.column('Child SU').mean()
```

```
heights_with_su_predictions = heights_SU.with_column(
    'Prediction SU', heights_SU.apply(predict_child_su,
    'MidParent SU'))
)
```

```
heights_with_su_predictions.scatter('MidParent SU')
```

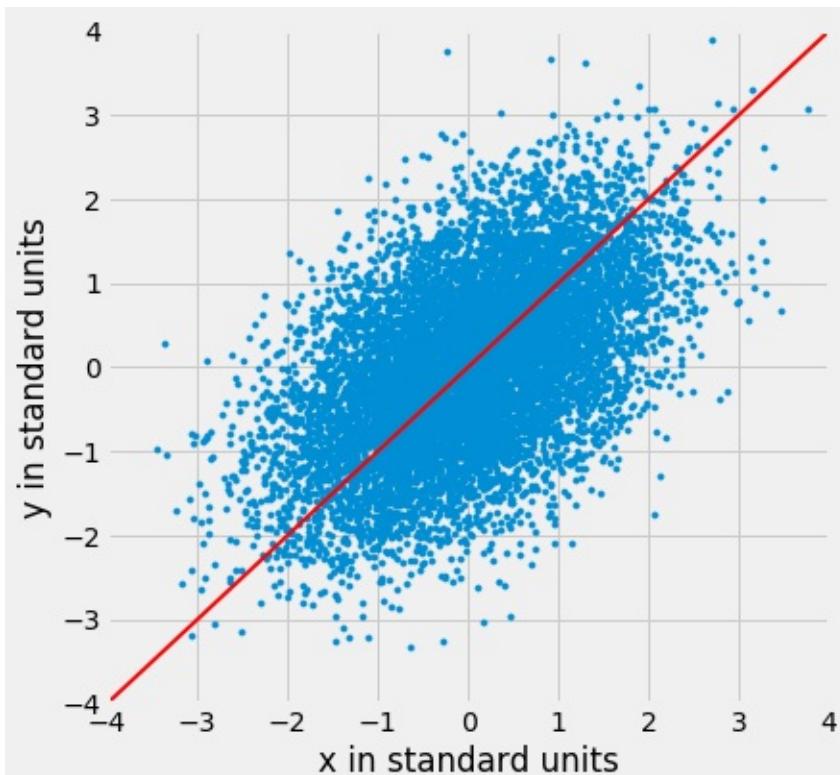


This plot looks exactly like the plot drawn on the original scale. Only the numbers on the axes have changed. This confirms that we can understand the prediction process by just working in standard units.

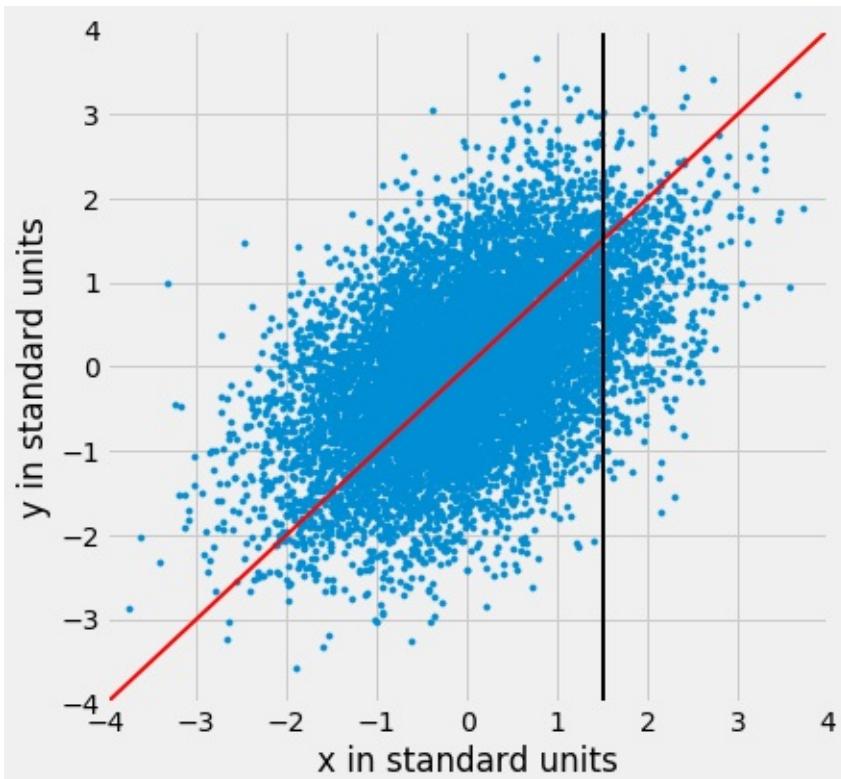
Identifying the Line in Standard Units

Galton's scatter plot has a *football* shape – that is, it is roughly oval like an American football. Not all scatter plots are football shaped, not even those that show linear association. But in this section we will pretend we are Galton and work only with football shaped scatter plots. In the next section, we will generalize our analysis to other shapes of plots.

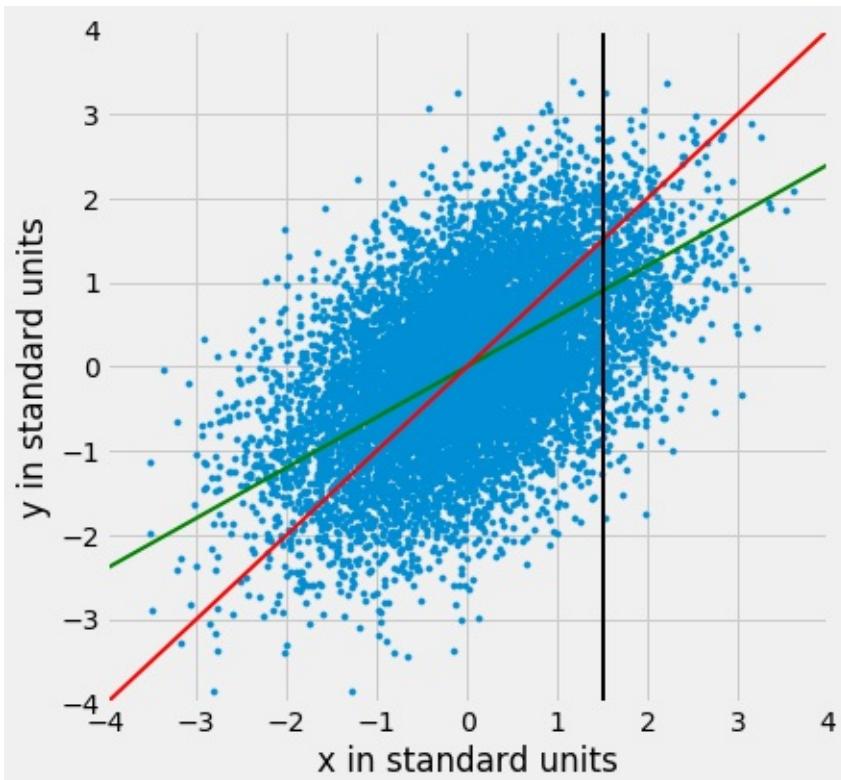
Here is a football shaped scatter plot with both variables measured in standard units. The 45 degree line is shown in red.



But the 45 degree line is not the line that picks off the centers of the vertical strips. You can see that in the figure below, where the vertical line at 1.5 standard units is shown in black. The points on the scatter plot near the black line all have heights roughly in the -2 to 3 range. The red line is too high to pick off the center.



So the 45 degree line is not the "graph of averages." That line is the green one shown below.



Both lines go through the origin $(0, 0)$. The green line goes through the centers of the vertical strips (at least roughly), and is *flatter* than the red 45 degree line.

The slope of the 45 degree line is 1. So the slope of the green "graph of averages" line is a value that is positive but less than 1.

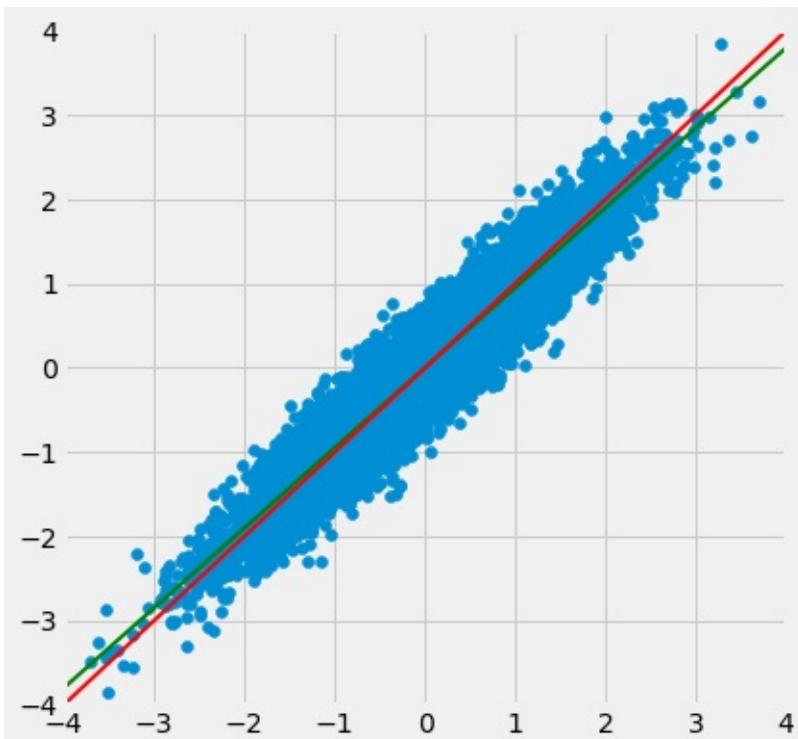
What value could that be? You've guessed it – it's r .

The Regression Line, in Standard Units

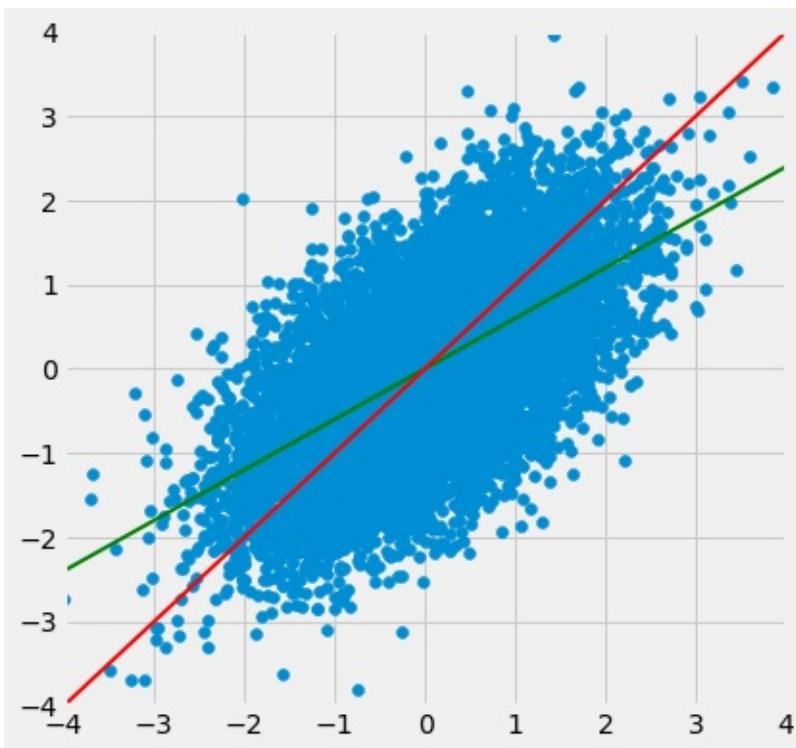
The green "graph of averages" line is called the *regression line*, for reasons we will explain shortly. But first, let's simulate some football shaped scatter plots with different values of r , and see how the line changes. In each case, the red 45 degree line has been drawn for comparison.

The function that performs the simulation is called `regression_line` and takes r as its argument.

```
regression_line(0.95)
```



```
regression_line(0.6)
```



When r is close to 1, the scatter plot, the 45 degree line, and the regression line are all very close to each other. But for more moderate values of r , the regression line is noticeably flatter.

The Regression Effect

In terms of prediction, this means that for parents whose midparent height is at 1.5 standard units, our prediction of the child's height is somewhat *less* than 1.5 standard units. If the midparent height is 2 standard units, we predict that the child's height will be somewhat less than 2 standard units.

In other words, we predict that the child will be somewhat closer to average than the parents were.

This didn't please Sir Francis Galton. He had been hoping that exceptionally tall parents would have children who were just as exceptionally tall. However, the data were clear, and Galton realized that the tall parents have children who are not quite as exceptionally tall, on average. Frustrated, Galton called this phenomenon "regression to mediocrity."

Galton also noticed that exceptionally short parents had children 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 Equation of the Regression Line

In regression, we use the value of one variable (which we will call x) to predict the value of another (which we will call y). When the variables x and y are measured in standard units, the regression line for predicting y based on x has slope r and passes through the origin. Thus the equation of the regression line can be written as:

$$\text{estimate of } y = r$$

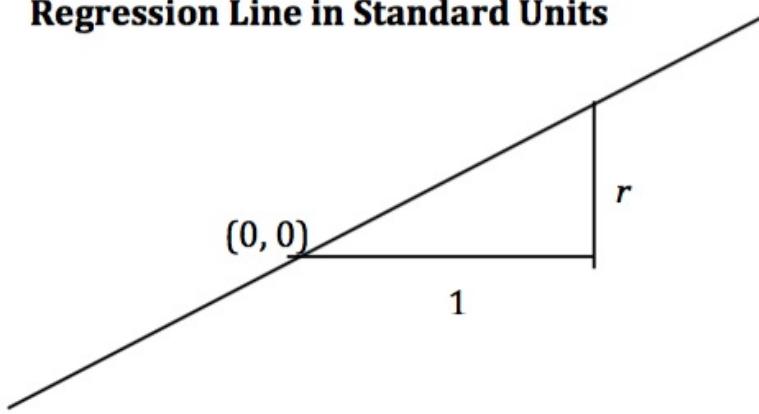
• x when both variables are measured in standard units

In the original units of the data, this becomes

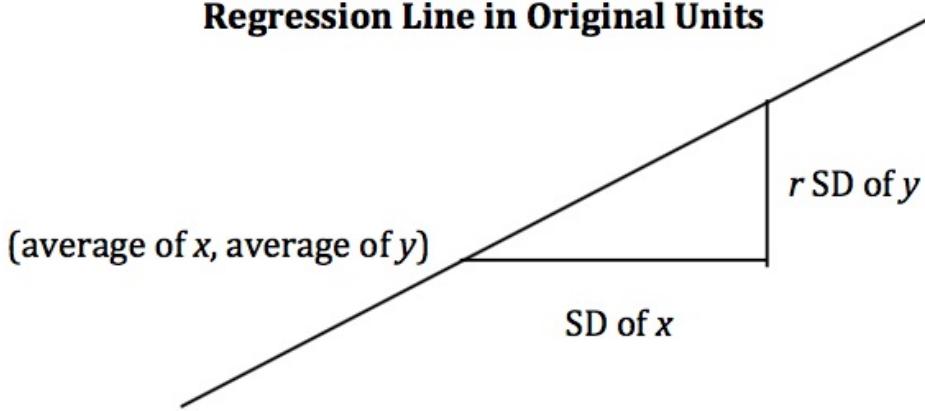
$$\frac{\text{estimate of } y - \text{average of } y}{\text{SD of } y} = r$$

$$\times \frac{\text{the given } x - \text{average of } x}{\text{SD of } x}$$

Regression Line in Standard Units



Regression Line in Original Units



The slope and intercept of the regression line in original units can be derived from the diagram above.

$$\text{slope of the regression line} = r \cdot \frac{\text{SD of } y}{\text{SD of } x}$$

intercept of the regression line = average of y
 – slope · average of x

The three functions below compute the correlation, slope, and intercept. All of them take three arguments: the name of the table, the label of the column containing x , and the label of the column containing y .

```
def correlation(t, label_x, label_y):
    return
    np.mean(standard_units(t.column(label_x))*standard_units(t.column
(label_y)))

def slope(t, label_x, label_y):
    r = correlation(t, label_x, label_y)
    return r*np.std(t.column(label_y))/np.std(t.column(label_x))

def intercept(t, label_x, label_y):
    return np.mean(t.column(label_y)) - slope(t, label_x,
label_y)*np.mean(t.column(label_x))
```

The Regression Line and Galton's Data

The correlation between midparent height and child's height is 0.32:

```
galton_r = correlation(heights, 'MidParent', 'Child')
galton_r
```

0.32094989606395924

We can also find the equation of the regression line for predicting the child's height based on midparent height.

```
galton_slope = slope(heights, 'MidParent', 'Child')
galton_intercept = intercept(heights, 'MidParent', 'Child')
galton_slope, galton_intercept
```

(0.63736089696947895, 22.636240549589751)

The equation of the regression line is

$$\begin{aligned}\text{estimate of child's height} &= 0.64 \cdot \text{midparent height} \\ &+ 22.64\end{aligned}$$

This is also known as the *regression equation*. The principal use of the regression equation is to predict y based on x .

For example, for a midparent height of 70.48 inches, the regression equation predicts the child's height to be 67.56 inches.

```
galton_slope*70.48 + galton_intercept
```

```
67.557436567998622
```

Our original prediction, created by taking the average height of all children who had midparent heights close to 70.48, came out to be pretty close: 67.63 inches compared to the regression line's prediction of 67.55 inches.

```
heights_with_predictions.where('MidParent',
are.equal_to(70.48)).show(3)
```

MidParent	Child	Prediction
70.48	74	67.6342
70.48	70	67.6342
70.48	68	67.6342

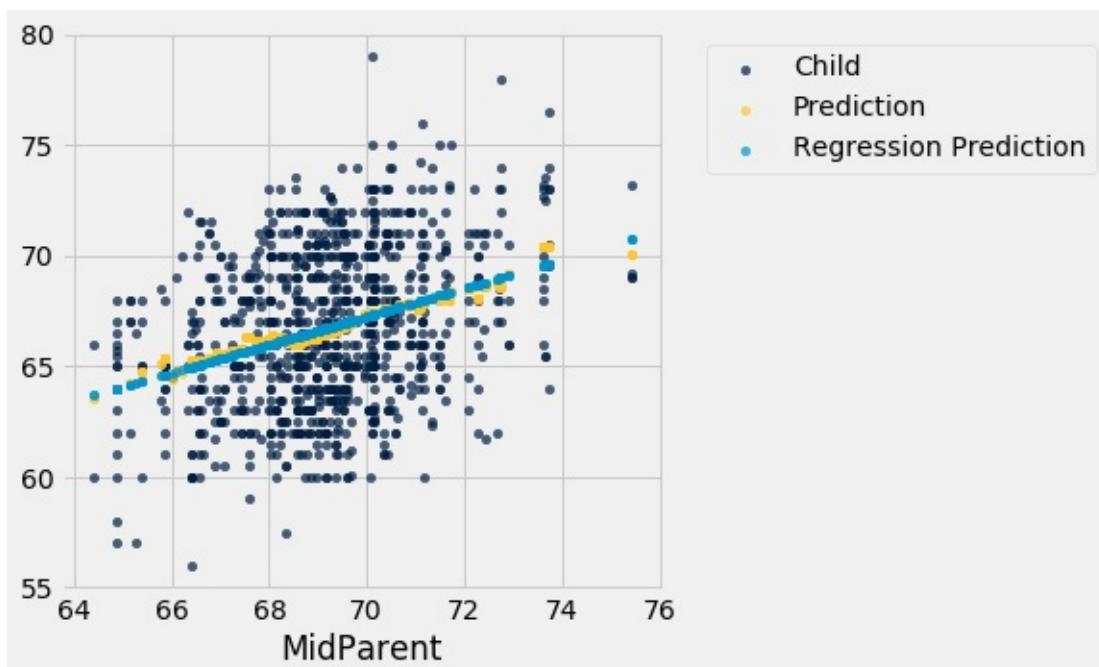
... (5 rows omitted)

Here are all of the rows in Galton's table, along with our original predictions and the new regression predictions of the children's heights.

```
heights_with_predictions = heights_with_predictions.with_column(
    'Regression Prediction',
    galton_slope*heights.column('MidParent') + galton_intercept
)
heights_with_predictions
```

MidParent	Child	Prediction	Regression Prediction
75.43	73.2	70.1	70.7124
75.43	69.2	70.1	70.7124
75.43	69	70.1	70.7124
75.43	69	70.1	70.7124
73.66	73.5	70.4158	69.5842
73.66	72.5	70.4158	69.5842
73.66	65.5	70.4158	69.5842
73.66	65.5	70.4158	69.5842
72.06	71	68.5025	68.5645
72.06	68	68.5025	68.5645
... (924 rows omitted)			

```
heights_with_predictions.scatter('MidParent')
```



The grey dots show the regression predictions, all on the regression line. Notice how the line is very close to the gold graph of averages. For these data, the regression line does a good job of approximating the centers of the vertical strips.

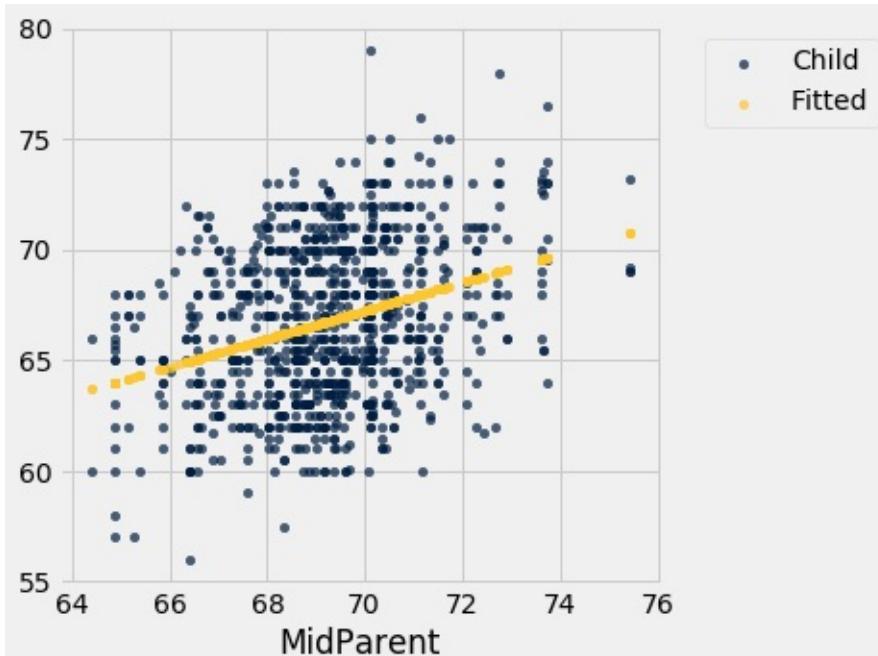
Fitted Values

The predictions all lie on the line and are known as the "fitted values". The function `fit` takes the name of the table and the labels of x and y , and returns an array of fitted values, one fitted value for each point in the scatter plot.

```
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
```

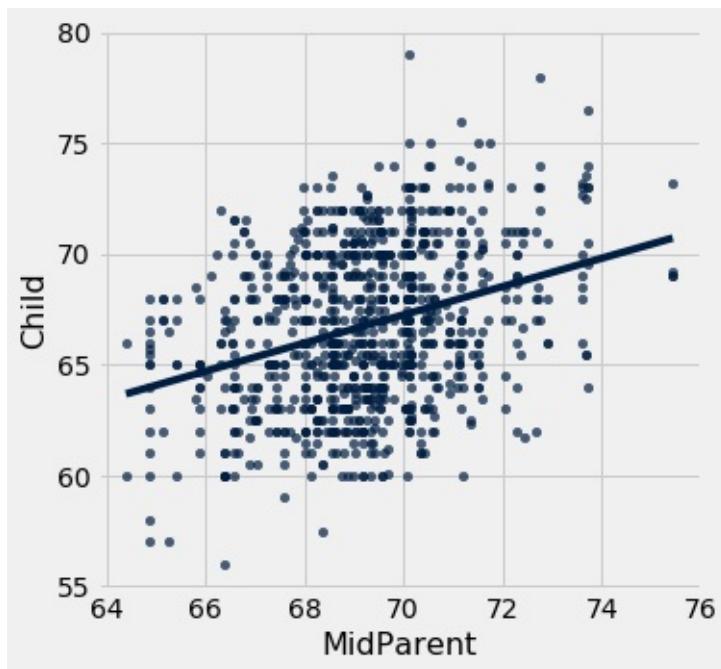
It is easier to see the line in the graph below than in the one above.

```
heights.with_column('Fitted', fit(heights, 'MidParent',
'Child')).scatter('MidParent')
```



Another way to draw the line is to use the option `fit_line=True` with the Table method `scatter`.

```
heights.scatter('MidParent', fit_line=True)
```

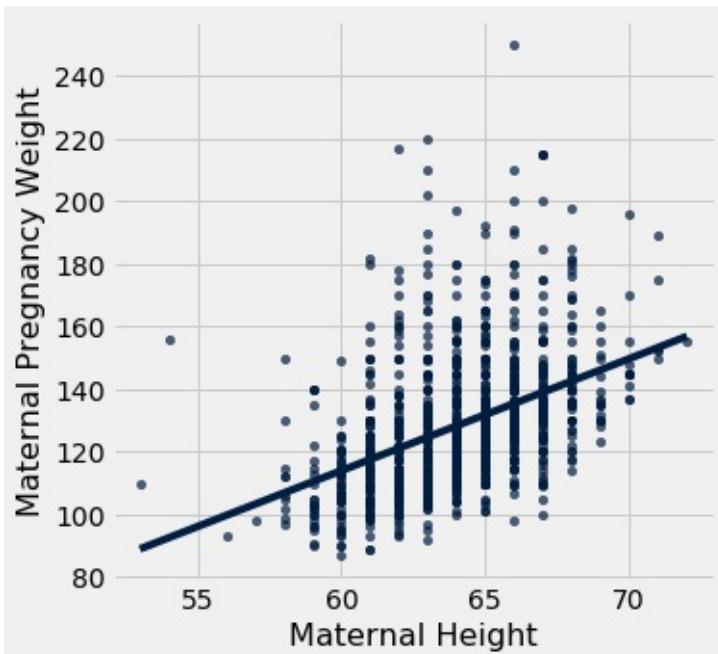


Units of Measurement of the Slope¶

The slope is a ratio, and it's worth taking a moment to study the units in which it is measured. Our example comes from the familiar dataset about mothers who gave birth in a hospital system. The scatter plot of pregnancy weights versus heights looks like a football that has been used in one game too many, but it's close enough to a football that we can justify putting our fitted line through it. In later sections we will see how to make such justifications more formal.

```
baby = Table.read_table('baby.csv')
```

```
baby.scatter('Maternal Height', 'Maternal Pregnancy Weight',
fit_line=True)
```



```
slope(baby, 'Maternal Height', 'Maternal Pregnancy Weight')
```

3.5728462592750558

The slope of the regression line is **3.57 pounds per inch**. This means that for two women who are 1 inch apart in height, our prediction of pregnancy weight will differ by 3.57 pounds. For a woman who is 2 inches taller than another, our prediction of pregnancy weight will be

$$2 \times 3.57 = 7.14$$

pounds more than our prediction for the shorter woman.

Notice that the successive vertical strips in the scatter plot are one inch apart, because the heights have been rounded to the nearest inch. Another way to think about the slope is to take any two consecutive strips (which are necessarily 1 inch apart), corresponding to two groups of women who are separated by 1 inch in height. The slope of 3.57 pounds per inch means that the average pregnancy weight of the taller group is about 3.57 pounds more than that of the shorter group.

Example

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.

$$\text{slope} = \frac{r \cdot \text{SD of } y}{\text{SD of } x} = \frac{0.5 \cdot 2 \text{ inches}}{5 \text{ pounds}}$$

$$= 0.2 \text{ inches per pound}$$

$$\begin{aligned}\text{intercept} &= \text{average of } y - \text{slope} \cdot \text{average of } x \\ &= 14 \text{ inches} - 0.2 \text{ inches per pound} \cdot 50 \text{ pounds} \\ &= 4 \text{ inches}\end{aligned}$$

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.

Endnote¶

Even though we won't establish the mathematical basis for the regression equation, we can see that it gives pretty good predictions when the scatter plot is football shaped. It is a surprising mathematical fact that no matter what the shape of the scatter plot, the same equation gives the "best" among all straight lines. That's the topic of the next section.

[Interact](#)

The Method of Least Squares

We have retraced the steps that Galton and Pearson took to develop the equation of the regression line that runs through a football shaped scatter plot. But not all scatter plots are football shaped, not even linear ones. Does every scatter plot have a "best" line that goes through it? If so, can we still use the formulas for the slope and intercept developed in the previous section, or do we need new ones?

To address these questions, we need a reasonable definition of "best". Recall that the purpose of the line is to *predict* or *estimate* values of y , given values of x . Estimates typically aren't perfect. Each one is off the true value by an *error*. A reasonable criterion for a line to be the "best" is for it to have the smallest possible overall error among all straight lines.

In this section we will make this criterion precise and see if we can identify the best straight line under the criterion.

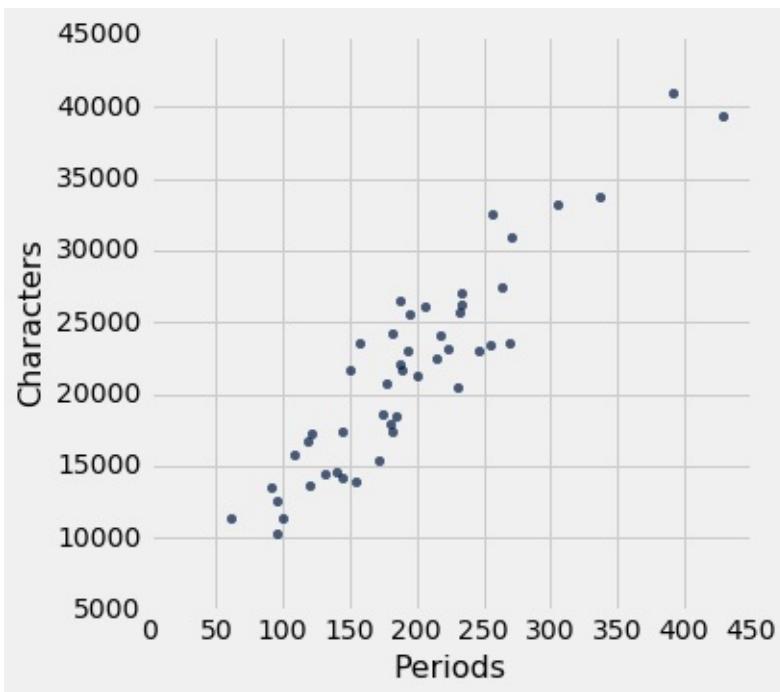
Our first example is a dataset that has one row for every chapter of the novel "Little Women." The goal is to estimate the number of characters (that is, letters, spaces punctuation marks, and so on) based on the number of periods. Recall that we attempted to do this in the very first lecture of this course.

```
little_women = Table.read_table('little_women.csv')
little_women = little_women.move_to_start('Periods')
little_women.show(3)
```

Periods	Characters
189	21759
188	22148
231	20558

... (44 rows omitted)

```
little_women.scatter('Periods', 'Characters')
```



To explore the data, we will need to use the functions `correlation`, `slope`, `intercept`, and `fit` defined in the previous section.

```
correlation(little_women, 'Periods', 'Characters')
```

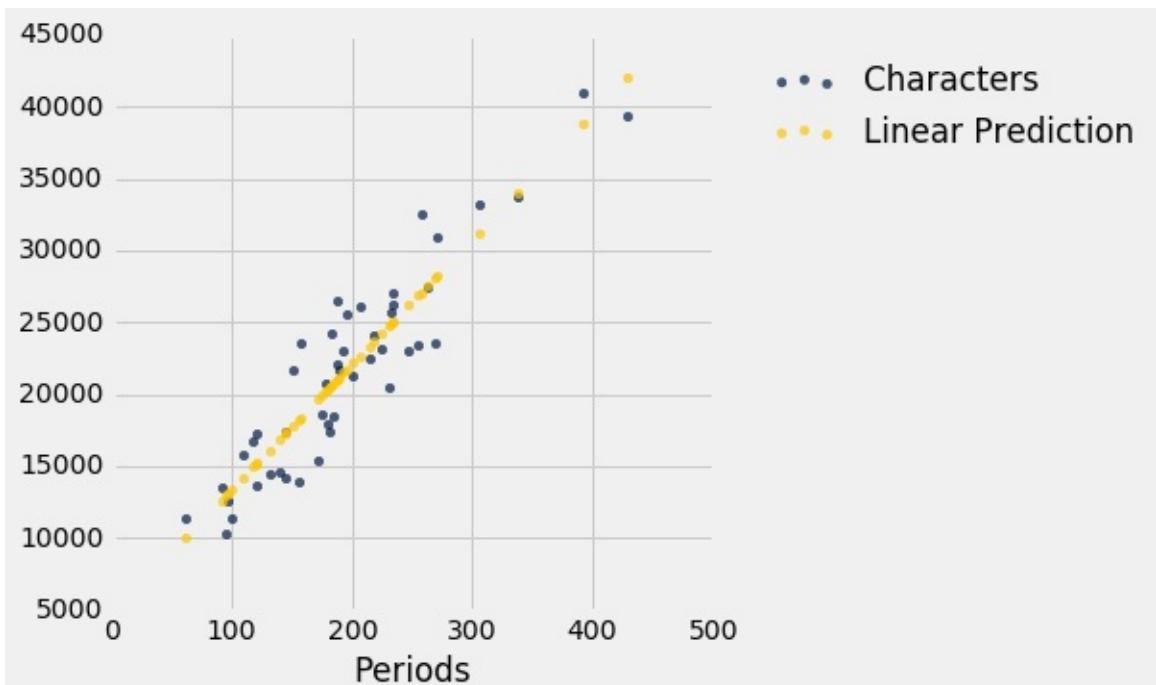
```
0.92295768958548163
```

The scatter plot is remarkably close to linear, and the correlation is more than 0.92.

Error in Estimation

The graph below shows the scatter plot and line that we developed in the previous section. We don't yet know if that's the best among all lines. We first have to say precisely what "best" means.

```
lw_with_predictions = little_women.with_column('Linear
Prediction', fit(little_women, 'Periods', 'Characters'))
lw_with_predictions.scatter('Periods')
```



Corresponding to each point on the scatter plot, there is an error of prediction calculated as the actual value minus the predicted value. It is the vertical distance between the point and the line, with a negative sign if the point is below the line.

```
actual = lw_with_predictions.column('Characters')
predicted = lw_with_predictions.column('Linear Prediction')
errors = actual - predicted
```

```
lw_with_predictions.with_column('Error', errors)
```

Periods	Characters	Linear Prediction	Error
189	21759	21183.6	575.403
188	22148	21096.6	1051.38
231	20558	24836.7	-4278.67
195	25526	21705.5	3820.54
255	23395	26924.1	-3529.13
140	14622	16921.7	-2299.68
131	14431	16138.9	-1707.88
214	22476	23358	-882.043
337	33767	34056.3	-289.317
185	18508	20835.7	-2327.69

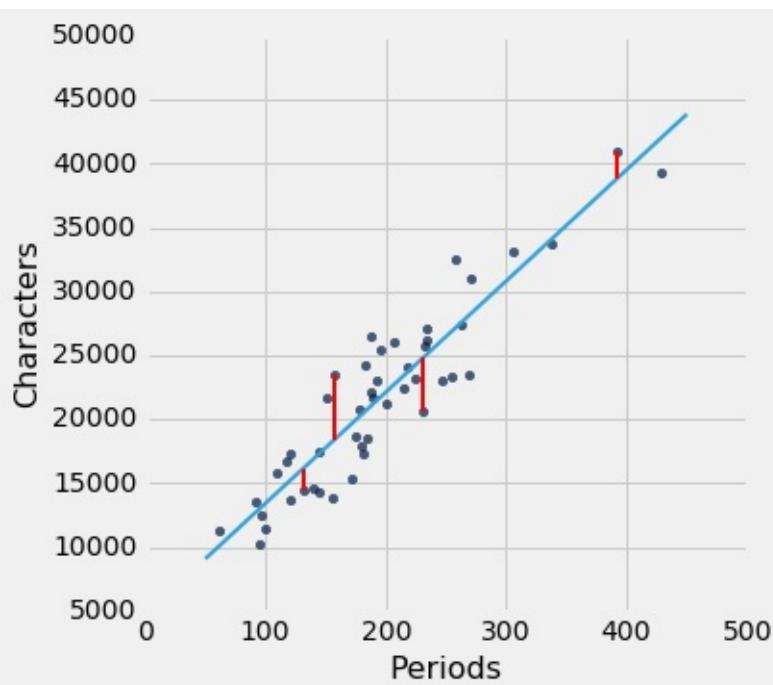
... (37 rows omitted)

We can use `slope` and `intercept` to calculate the slope and intercept of the fitted line. The graph below shows the line (in light blue). The errors corresponding to four of the points are shown in red. There is nothing special about those four points. They were just chosen for clarity of the display. The function `lw_errors` takes a slope and an intercept (in that order) as its arguments and draws the figure.

```
lw_reg_slope = slope(little_women, 'Periods', 'Characters')
lw_reg_intercept = intercept(little_women, 'Periods',
'Characters')
```

```
print('Slope of Regression Line: ', np.round(lw_reg_slope),
'characters per period')
print('Intercept of Regression Line:',
np.round(lw_reg_intercept), 'characters')
lw_errors(lw_reg_slope, lw_reg_intercept)
```

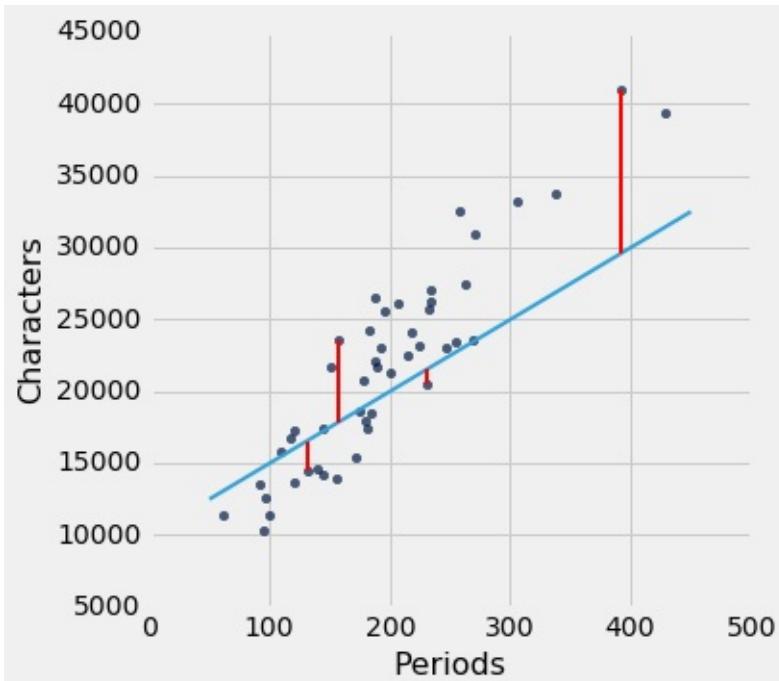
Slope of Regression Line: 87.0 characters per period
 Intercept of Regression Line: 4745.0 characters



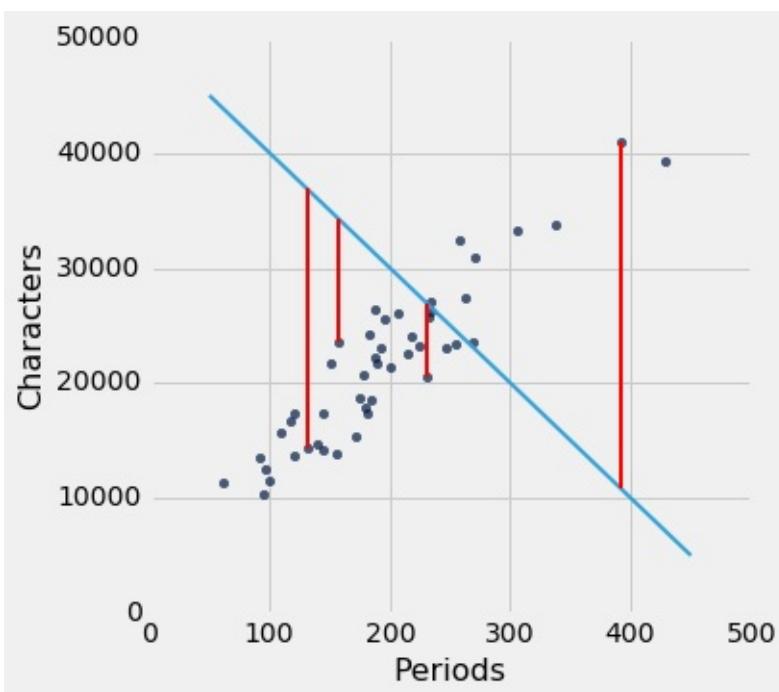
Had we used a different line to create our estimates, the errors would have been different. The graph below shows how big the errors would be if we were to use another line for estimation. The second graph shows large errors obtained by using a line that is downright

silly.

```
lw_errors(50, 10000)
```



```
lw_errors(-100, 50000)
```



Root Mean Squared Error

What we need now is one overall measure of the rough size of the errors. You will recognize the approach to creating this – it's exactly the way we developed the SD.

If you use any arbitrary line to calculate your 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 will take the mean of the squared errors rather than the mean of the errors themselves.

The mean squared error of estimation is a measure of roughly how big the squared errors are, but as we have noted earlier, its units are hard to interpret. Taking the square root yields the root mean square error (rmse), which is in the same units as the variable being predicted and therefore much easier to understand.

Minimizing the Root Mean Squared Error

Our observations so far can be summarized as follows.

- To get estimates of y based on x , you can use any line you want.
- Every line has a root mean squared error of estimation.
- "Better" lines have smaller errors.

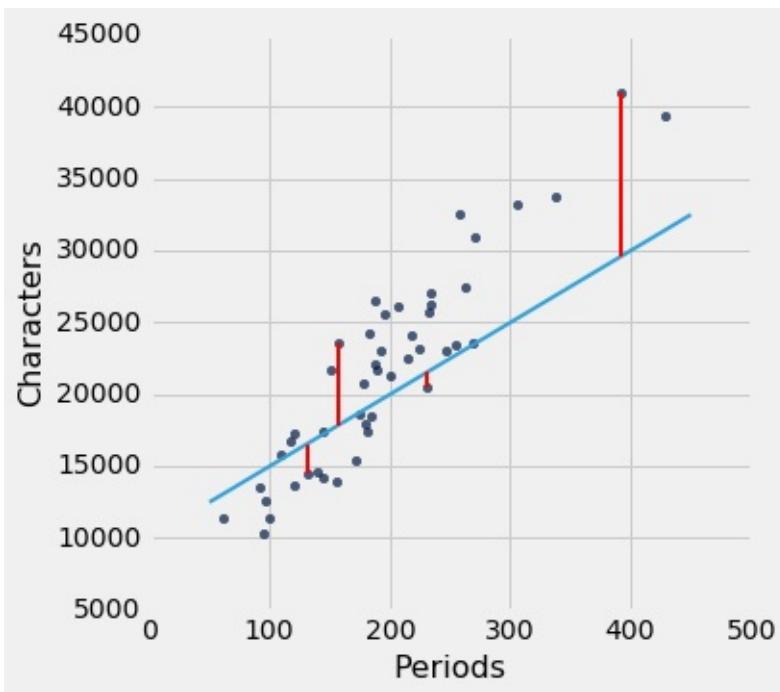
Is there a "best" line? That is, is there a line that minimizes the root mean squared error among all lines?

To answer this question, we will start by defining a function `lw_rmse` to compute the root mean squared error of any line through the Little Women scatter diagram. The function takes the slope and the intercept (in that order) as its arguments.

```
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.mean((y - fitted) ** 2)
    print("Root mean squared error:", mse ** 0.5)
```

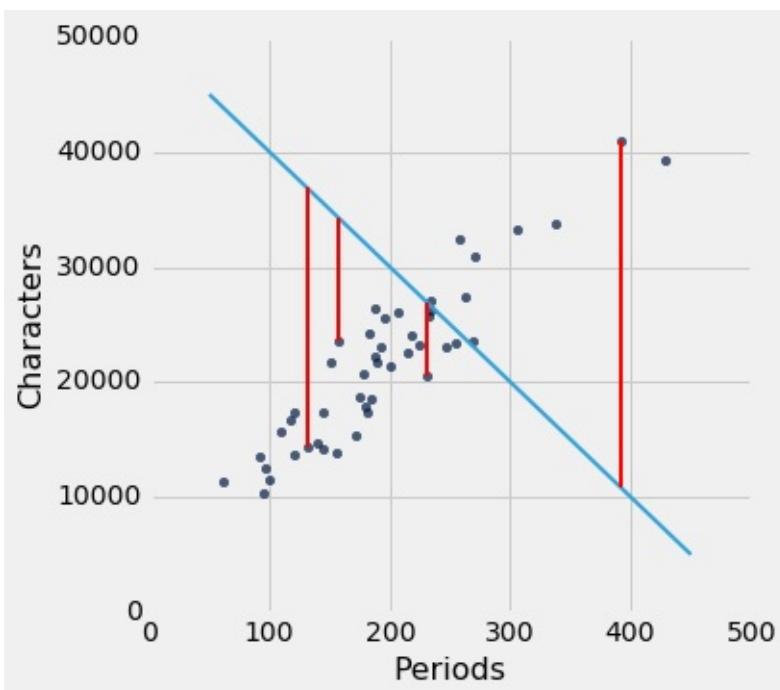
```
lw_rmse(50, 10000)
```

```
Root mean squared error: 4322.16783177
```



```
lw_rmse(-100, 50000)
```

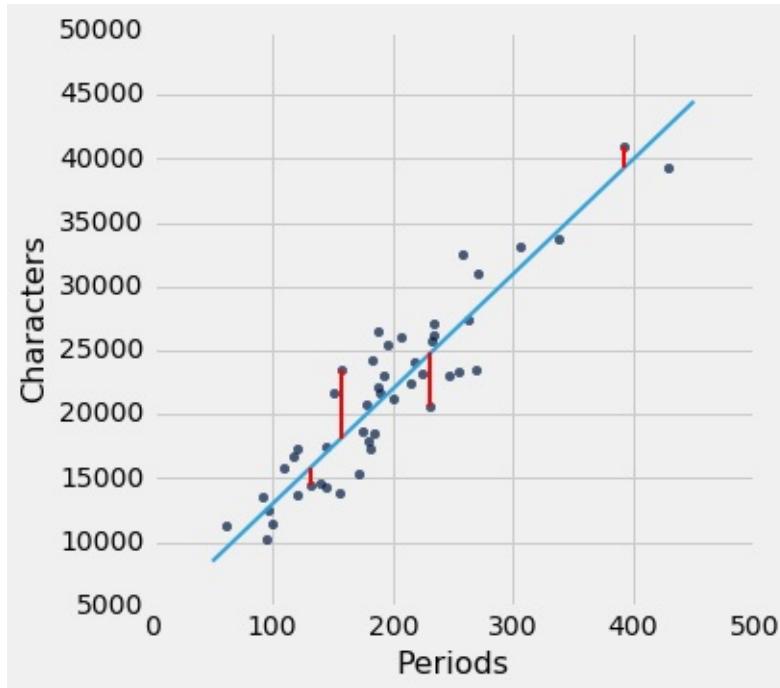
Root mean squared error: 16710.1198374



Bad lines have big values of rmse, as expected. But the rmse is much smaller if we choose a slope and intercept close to those of the regression line.

```
lw_rmse(90, 4000)
```

```
Root mean squared error: 2715.53910638
```

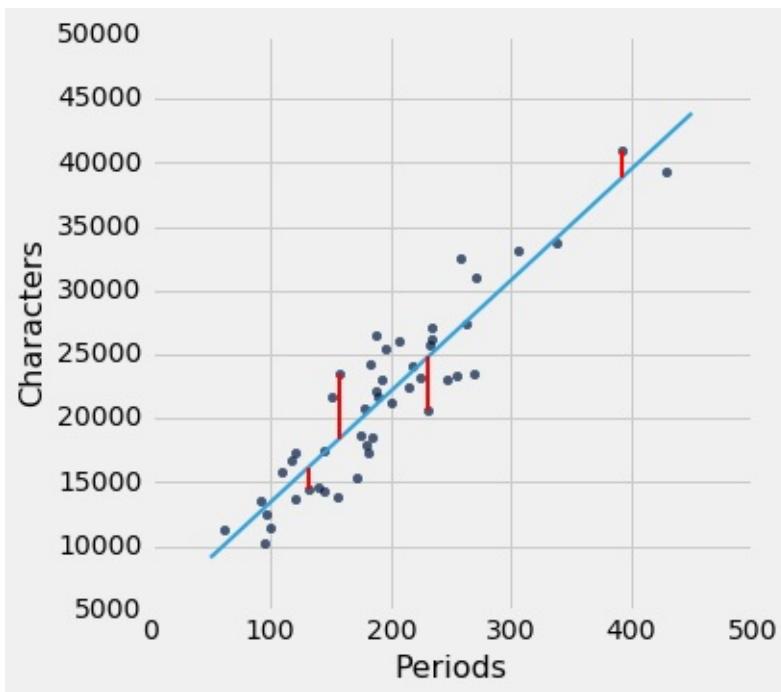


Here is the root mean squared error corresponding to the regression line. By a remarkable fact of mathematics, no other line can beat this one.

- **The regression line is the unique straight line that minimizes the mean squared error of estimation among all straight lines.**

```
lw_rmse(lw_reg_slope, lw_reg_intercept)
```

```
Root mean squared error: 2701.69078531
```



The proof of this statement requires abstract mathematics that is beyond the scope of this course. On the other hand, we do have a powerful tool – Python – that performs large numerical computations with ease. So we can use Python to confirm that the regression line minimizes the mean squared error.

Numerical Optimization

First note that a line that minimizes the root mean squared error is also a line that minimizes the squared error. The square root makes no difference to the minimization. So we will save ourselves a step of computation and just minimize the mean squared error (mse).

We are trying to predict the number of characters (y) based on the number of periods (x) in chapters of Little Women. If we use the line

$$\text{prediction} = ax + b$$

it will have an mse that depends on the slope a and the intercept b . The function `lw_mse` takes the slope and intercept as its arguments and returns the corresponding mse.

```
def lw_mse(any_slope, any_intercept):
    x = little_women.column('Periods')
    y = little_women.column('Characters')
    fitted = any_slope*x + any_intercept
    return np.mean((y - fitted) ** 2)
```

Let's check that `lw_mse` gets the right answer for the root mean squared error of the regression line. Remember that `lw_mse` returns the mean squared error, so we have to take the square root to get the rmse.

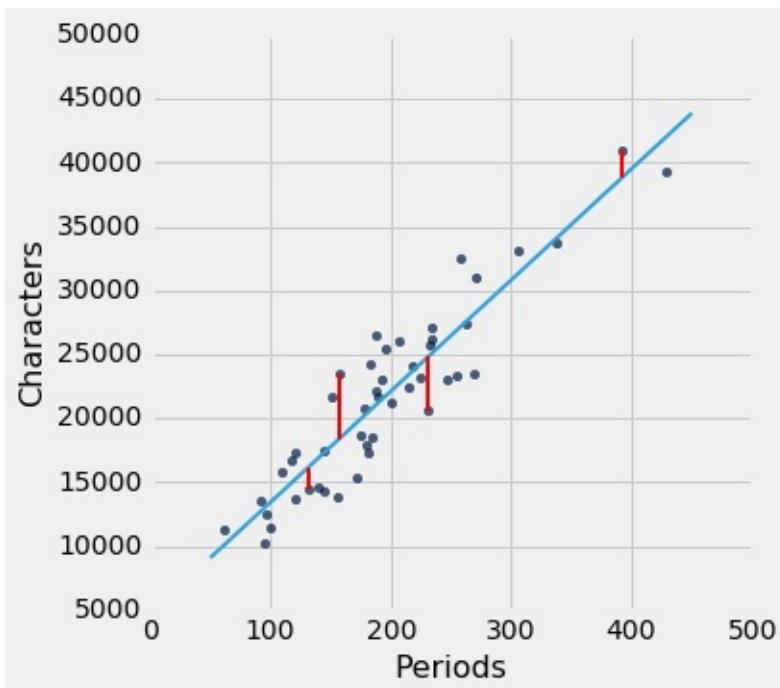
```
lw_mse(lw_reg_slope, lw_reg_intercept)**0.5
```

```
2701.690785311856
```

That's the same as the value we got by using `lw_rmse` earlier:

```
lw_rmse(lw_reg_slope, lw_reg_intercept)
```

```
Root mean squared error: 2701.69078531
```



You can confirm that `lw_mse` returns the correct value for other slopes and intercepts too. For example, here is the rmse of the extremely bad line that we tried earlier.

```
lw_mse(-100, 50000)**0.5
```

```
16710.119837353752
```

And here is the rmse for a line that is close to the regression line.

```
lw_mse(90, 4000)**0.5
```

```
2715.5391063834586
```

If we experiment with different values, we can find a low-error slope and intercept through trial and error, but that would take a while. Fortunately, there is a Python function that does all the trial and error for us.

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.

The argument of `minimize` is a function that itself takes numerical arguments and returns a numerical value. For example, the function `lw_mse` takes a numerical slope and intercept as its arguments and returns the corresponding mse.

The call `minimize(lw_mse)` returns an array consisting of the slope and the intercept that minimize the mse. These minimizing values are excellent approximations arrived at by intelligent trial-and-error, not exact values based on formulas.

```
best = minimize(lw_mse)
best
```

```
array([ 86.97784117, 4744.78484535])
```

These values are the same as the values we calculated earlier by using the `slope` and `intercept` functions. We see small deviations due to the inexact nature of `minimize`, but the values are essentially the same.

```
print("slope from formula:      ", lw_reg_slope)
print("slope from minimize:     ", best.item(0))
print("intercept from formula:   ", lw_reg_intercept)
print("intercept from minimize:  ", best.item(1))
```

slope from formula:	86.9778412583
slope from minimize:	86.97784116615884
intercept from formula:	4744.78479657
intercept from minimize:	4744.784845352655

The Least Squares Line

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.

That is why the regression line is sometimes called the "least squares line."

[Interact](#)

Least Squares Regression

In an earlier section, we developed formulas for the slope and intercept of the regression line through a *football shaped* scatter diagram. It turns out that the slope and intercept of the least squares line have the same formulas as those we developed, *regardless of the shape of the scatter plot*.

We saw this in the example about Little Women, but let's confirm it in an example where the scatter plot clearly isn't football shaped. For the data, we are once again indebted to the rich [data archive of Prof. Larry Winner](#) of the University of Florida. A [2013 study](#) in the International Journal of Exercise Science studied collegiate shot put athletes and examined the relation between strength and shot put distance. The population consists of 28 female collegiate athletes. Strength was measured by the biggest amount (in kilograms) that the athlete lifted in the "1RM power clean" in the pre-season. The distance (in meters) was the athlete's personal best.

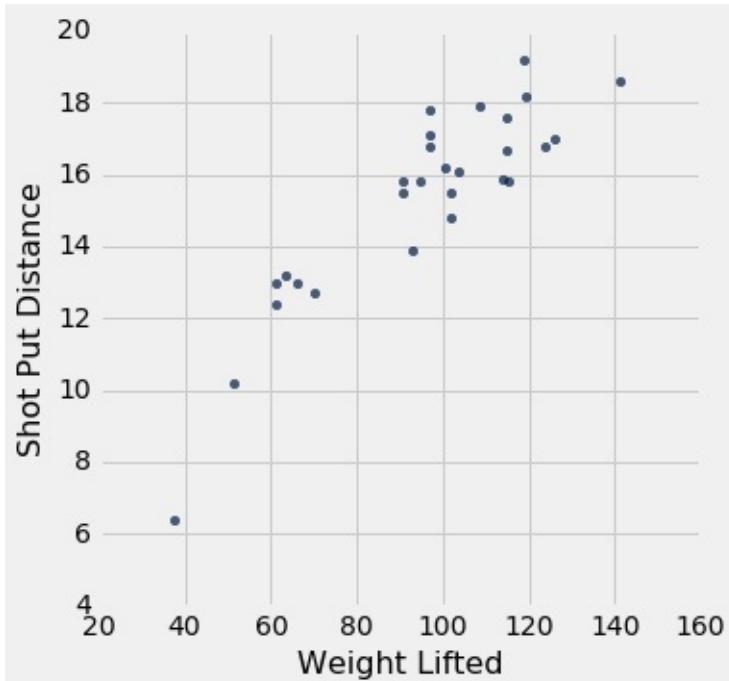
```
shotput = Table.read_table('shotput.csv')
```

```
shotput
```

Weight Lifted	Shot Put Distance
37.5	6.4
51.5	10.2
61.3	12.4
61.3	13
63.6	13.2
66.1	13
70	12.7
92.7	13.9
90.5	15.5
90.5	15.8

... (18 rows omitted)

```
shotput.scatter('Weight Lifted')
```



That's not a football shaped scatter plot. In fact, it seems to have a slight non-linear component. But if we insist on using a straight line to make our predictions, there is still one best straight line among all straight lines.

Our formulas for the slope and intercept of the regression line, derived for football shaped scatter plots, give the following values.

```
slope(shotput, 'Weight Lifted', 'Shot Put Distance')
```

```
0.098343821597819972
```

```
intercept(shotput, 'Weight Lifted', 'Shot Put Distance')
```

```
5.9596290983739522
```

Does it still make sense to use these formulas even though the scatter plot isn't football shaped? We can answer this by finding the slope and intercept of the line that minimizes the mse.

We will define the function `shotput_linear_mse` to take an arbitrary slope and intercept as arguments and return the corresponding mse. Then `minimize` applied to `shotput_linear_mse` will return the best slope and intercept.

```
def shotput_linear_mse(any_slope, any_intercept):
    x = shotput.column('Weight Lifted')
    y = shotput.column('Shot Put Distance')
    fitted = any_slope*x + any_intercept
    return np.mean((y - fitted) ** 2)
```

```
minimize(shotput_linear_mse)
```

```
array([ 0.09834382,  5.95962911])
```

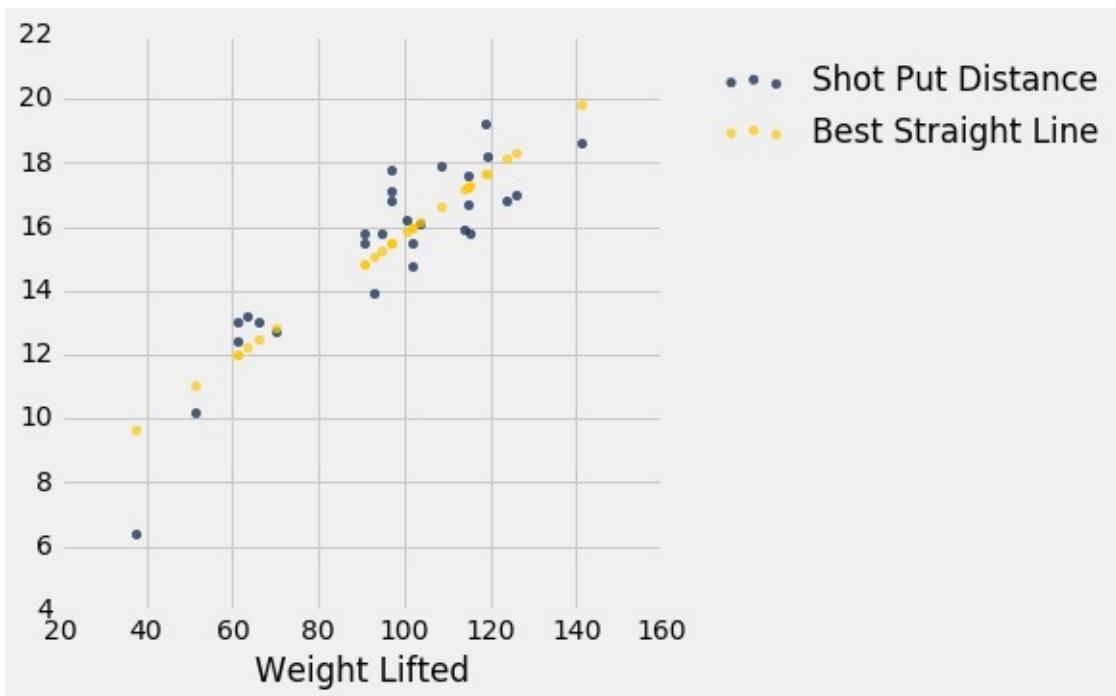
These values are the same as those we got by using our formulas. To summarize:

No matter what the shape of the scatter plot, there is a unique line that minimizes the mean squared error of estimation. It is called the regression line, and its slope and intercept are given by

$$\text{slope of the regression line} = r \cdot \frac{\text{SD of } y}{\text{SD of } x}$$

$$\begin{aligned}\text{intercept of the regression line} &= \text{average of } y \\ &\quad - \text{slope} \cdot \text{average of } x\end{aligned}$$

```
fitted = fit(shotput, 'Weight Lifted', 'Shot Put Distance')
shotput.with_column('Best Straight Line',
fitted).scatter('Weight Lifted')
```



Nonlinear Regression

The graph above reinforces our earlier observation that the scatter plot is a bit curved. So it is better to fit a curve than a straight line. The [study](#) postulated a quadratic relation between the weight lifted and the shot put distance. So let's use quadratic functions as our predictors and see if we can find the best one.

We have to find the best quadratic function among all quadratic functions, instead of the best straight line among all straight lines. The method of least squares allows us to do this.

The mathematics of this minimization is complicated and not easy to see just by examining the scatter plot. But numerical minimization is just as easy as it was with linear predictors! We can get the best quadratic predictor by once again using `minimize`. Let's see how this works.

Recall that a quadratic function has the form

$$f(x) = ax^2 + bx + c$$

for constants a , b , and c .

To find the best quadratic function to predict distance based on weight lifted, using the criterion of least squares, we will first write a function that takes the three constants as its arguments, calculates the fitted values by using the quadratic function above, and then returns the mean squared error.

The function is called `shotput_quadratic_mse`. Notice that the definition is analogous to that of `lw_mse`, except that the fitted values are based on a quadratic function instead of linear.

```
def shotput_quadratic_mse(a, b, c):
    x = shotput.column('Weight Lifted')
    y = shotput.column('Shot Put Distance')
    fitted = a*(x**2) + b*x + c
    return np.mean((y - fitted) ** 2)
```

We can now use `minimize` just as before to find the constants that minimize the mean squared error.

```
best = minimize(shotput_quadratic_mse)
best
```

```
array([-1.04004838e-03,  2.82708045e-01, -1.53182115e+00])
```

Our prediction of the shot put distance for an athlete who lifts x kilograms is about

$$-0.00104x^2 + 0.2827x - 1.5318$$

meters. For example, if the athlete can lift 100 kilograms, the predicted distance is 16.33 meters. On the scatter plot, that's near the center of a vertical strip around 100 kilograms.

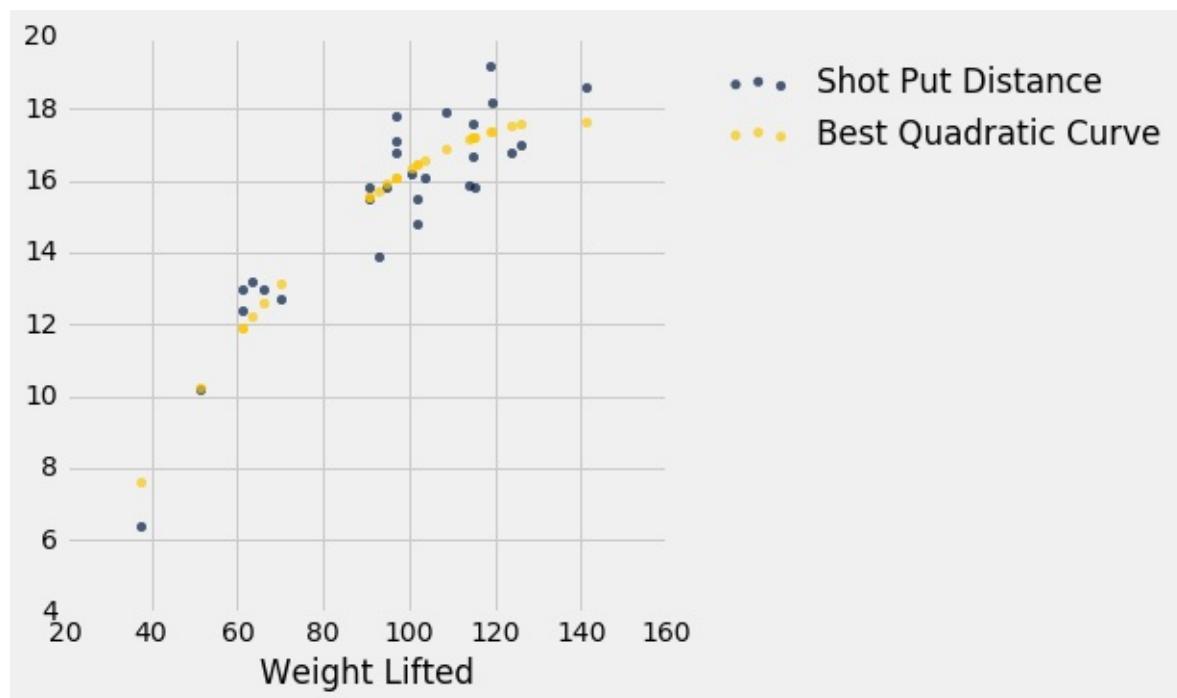
```
(-0.00104)*(100**2) + 0.2827*100 - 1.5318
```

```
16.3382
```

Here are the predictions for all the values of `Weight Lifted`. You can see that they go through the center of the scatter plot, to a rough approximation.

```
x = shotput.column(0)
shotput_fit = best.item(0)*(x**2) + best.item(1)*x +
best.item(2)

shotput.with_column('Best Quadratic Curve',
shotput_fit).scatter(0)
```



[Interact](#)

Visual Diagnostics¶

Suppose a data scientist has decided to use linear regression to estimate values of a response variable based on a predictor. To see how well this method of estimation performs, the data scientist must know how far off the estimates are from the actual values. These differences are called *residuals*.

$$\text{residual} = \text{observed value} - \text{regression estimate}$$

A residual is what's left over – the residue – after estimation.

Residuals are the vertical distances of the points from the regression line. 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 , so for the point (x, y) ,

$$\begin{aligned}\text{residual} &= y - \text{fitted value of } y = y \\ &\quad - \text{height of regression line at } x\end{aligned}$$

The function `residual` calculates the residuals. The calculation assumes all the relevant functions we have already defined: `standard_units`, `correlation`, `slope`, `intercept`, and `fit`.

```
def residual(table, x, y):
    return table.column(y) - fit(table, x, y)
```

Continuing our example of using Galton's data to estimate the heights of adult children (the response) based on the midparent height (the predictor), let us calculate the fitted values and the residuals.

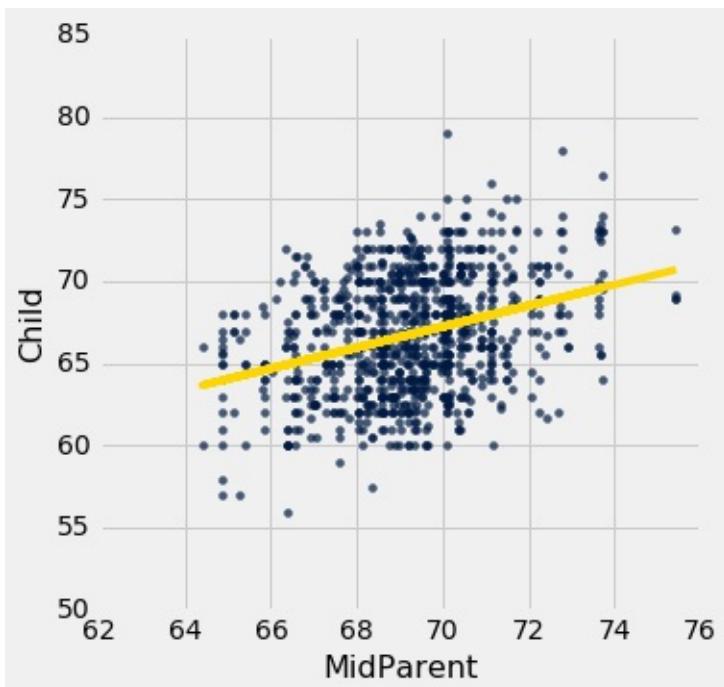
```
heights = heights.with_columns(
    'Fitted Value', fit(heights, 'MidParent', 'Child'),
    'Residual', residual(heights, 'MidParent', 'Child')
)
heights
```

MidParent	Child	Fitted Value	Residual
75.43	73.2	70.7124	2.48763
75.43	69.2	70.7124	-1.51237
75.43	69	70.7124	-1.71237
75.43	69	70.7124	-1.71237
73.66	73.5	69.5842	3.91576
73.66	72.5	69.5842	2.91576
73.66	65.5	69.5842	-4.08424
73.66	65.5	69.5842	-4.08424
72.06	71	68.5645	2.43553
72.06	68	68.5645	-0.564467
... (924 rows omitted)			

When there are so many variables to work with, it is always helpful to start with visualization. The function `scatter_fit` draws the scatter plot of the data, as well as the regression line.

```
def scatter_fit(table, x, y):
    table.scatter(x, y, s=15)
    plots.plot(table.column(x), fit(table, x, y), lw=4,
color='gold')
    plots.xlabel(x)
    plots.ylabel(y)
```

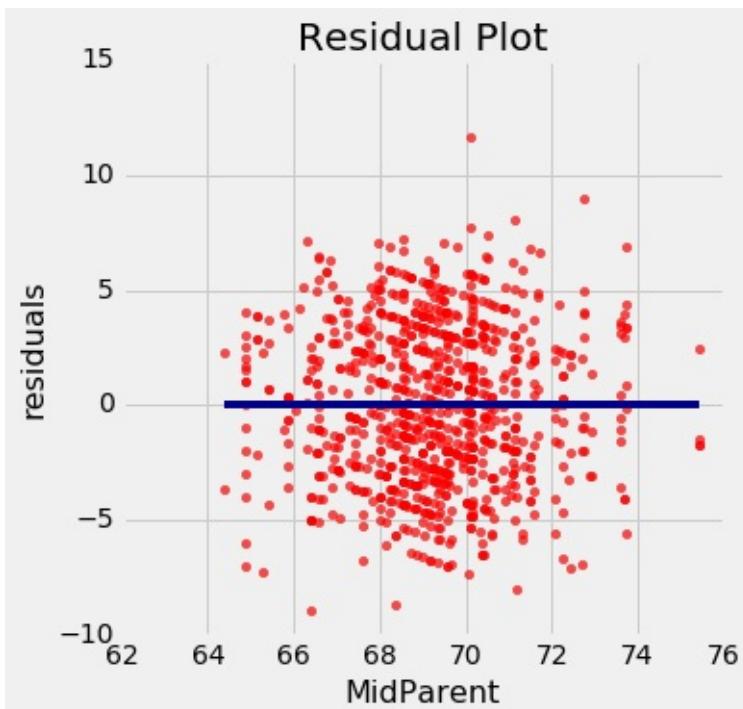
```
scatter_fit(heights, 'MidParent', 'Child')
```



A *residual plot* can be drawn by plotting the residuals against the predictor variable. The function `residual_plot` does just that.

```
def residual_plot(table, x, y):
    x_array = table.column(x)
    t = Table().with_columns(
        x, x_array,
        'residuals', residual(table, x, y)
    )
    t.scatter(x, 'residuals', color='r')
    xlims = make_array(min(x_array), max(x_array))
    plots.plot(xlims, make_array(0, 0), color='darkblue', lw=4)
    plots.title('Residual Plot')
```

```
residual_plot(heights, 'MidParent', 'Child')
```



The midparent heights are on the horizontal axis, as in the original scatter plot. But now the vertical axis shows the residuals. Notice that the plot appears to be centered around the horizontal line at the level 0 (shown in dark blue). Notice also that the plot shows no upward or downward trend. We will observe later that this is true of all regressions.

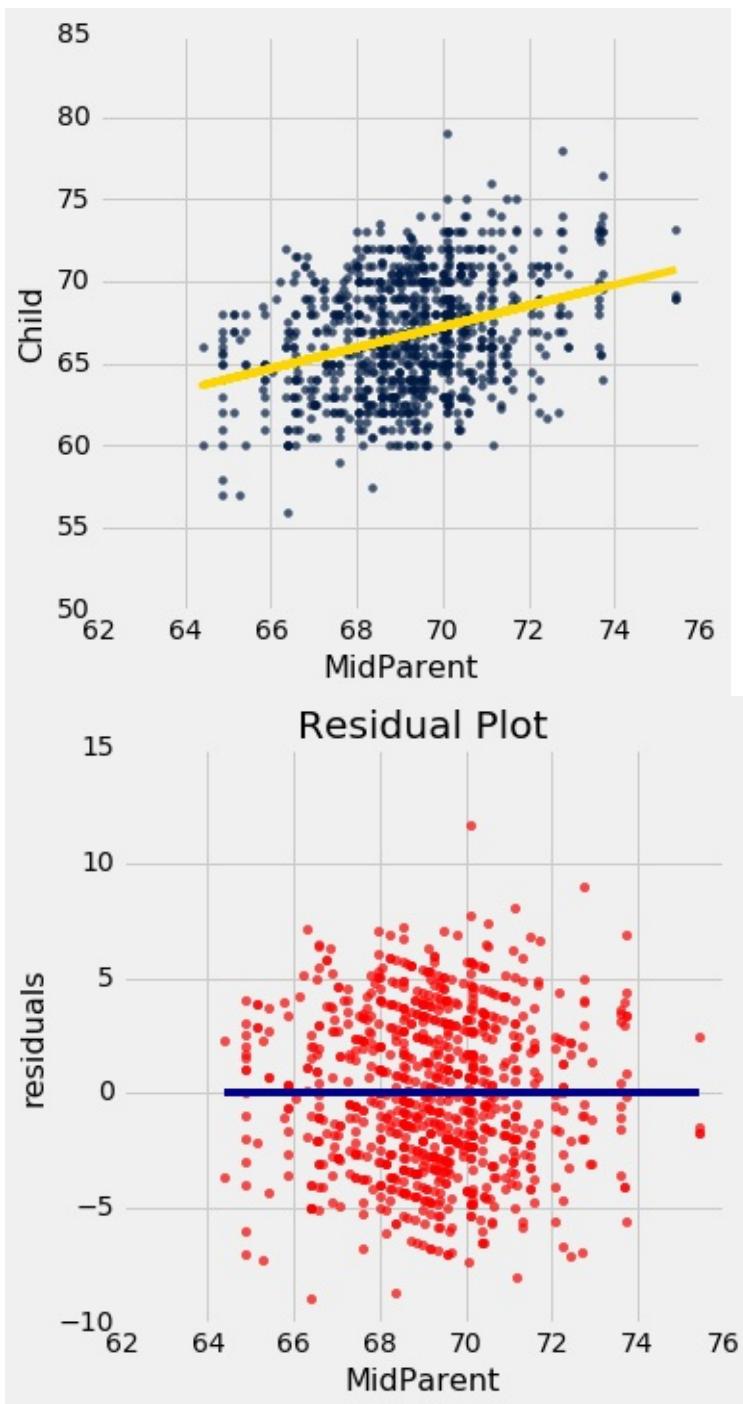
Regression Diagnostics¶

Residual plots help us make visual assessments of the quality of a linear regression analysis. Such assessments are called *diagnostics*. The function

`regression_diagnostic_plots` draws the original scatter plot as well as the residual plot for ease of comparison.

```
def regression_diagnostic_plots(table, x, y):
    scatter_fit(table, x, y)
    residual_plot(table, x, y)
```

```
regression_diagnostic_plots(heights, 'MidParent', 'Child')
```



This residual plot indicates that linear regression was a reasonable method of estimation. Notice how the residuals are distributed fairly symmetrically above and below the horizontal line at 0, corresponding to the original scatter plot being roughly symmetrical above and below. Notice also that the vertical spread of the plot is fairly even across the most common values of the children's heights. In other words, apart from a few outlying points, the plot isn't narrower in some places and wider in others.

In other words, the accuracy of the regression appears to be about the same across the observed range of the predictor variable.

The residual plot of a good regression shows no pattern. The residuals look about the same, above and below the horizontal line at 0, across the range of the predictor variable.

Detecting Nonlinearity

Drawing the scatter plot of the data usually gives an indication of whether the relation between the two variables is non-linear. 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.



Our data are a [dataset](#) on the age and length of dugongs, which are marine mammals related to manatees and sea cows (image from [Wikimedia Commons](#)). The data are in a table called `dugong`. Age is measured in years and length in meters. Because dugongs tend not to keep track of their birthdays, ages are estimated based on variables such as the condition of their teeth.

```
dugong =  
Table.read_table('http://www.statsci.org/data/oz/dugongs.txt')  
dugong = dugong.move_to_start('Length')  
dugong
```

Length	Age
1.8	1
1.85	1.5
1.87	1.5
1.77	1.5
2.02	2.5
2.27	4
2.15	5
2.26	5
2.35	7
2.47	8
... (17 rows omitted)	

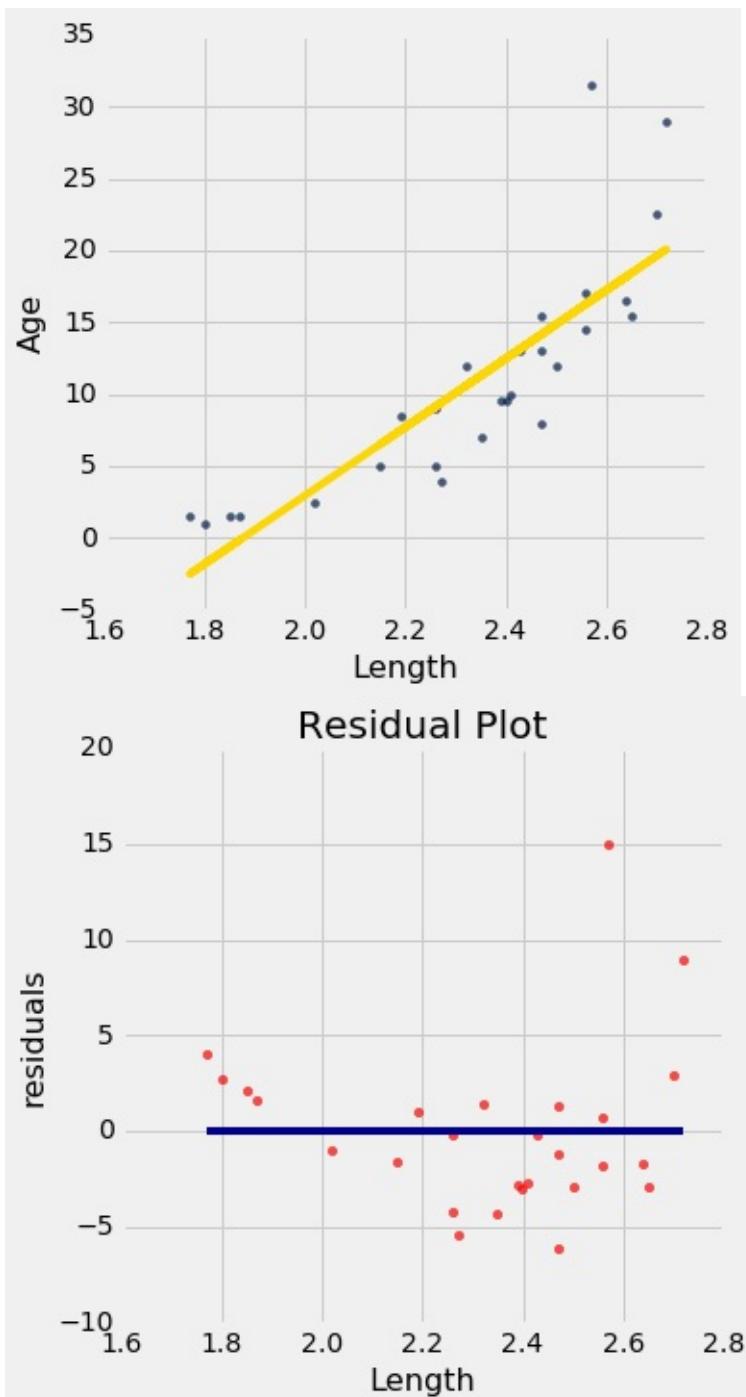
If we could measure the length of a dugong, what could we say about its age? Let's examine what our data say. Here is a regression of age (the response) on length (the predictor). The correlation between the two variables is substantial, at 0.83.

```
correlation(dugong, 'Length', 'Age')
```

```
0.82964745549057139
```

High correlation notwithstanding, the plot shows a curved pattern that is much more visible in the residual plot.

```
regression_diagnostic_plots(dugong, 'Length', 'Age')
```



While you can spot the non-linearity in the original scatter, it is more clearly evident in the residual plot.

At the low end of the lengths, the residuals are almost all positive; then they are almost all negative; then positive again at the high end of lengths. In other words the regression estimates have a pattern of being too high, then too low, then too high. That means it would have been better to use a curve instead of a straight line to estimate the ages.

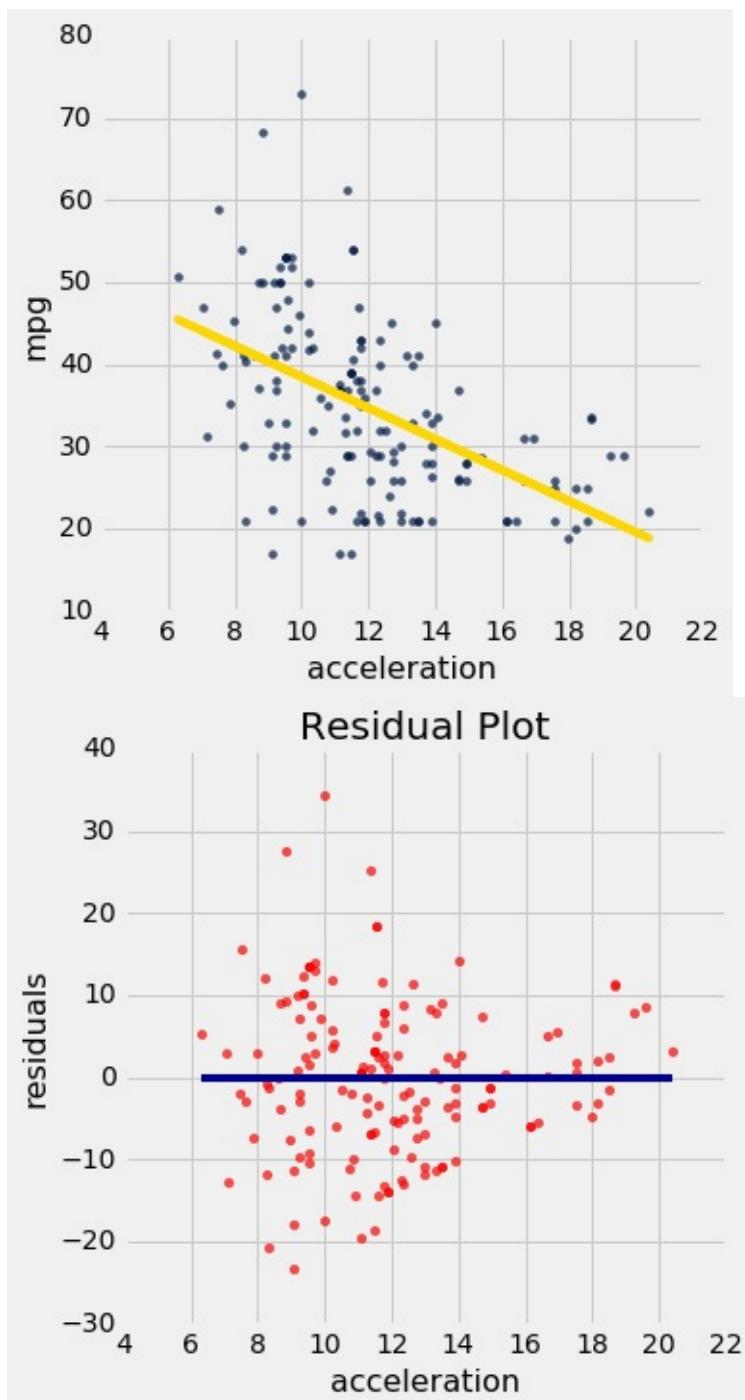
When a residual plot shows a pattern, there may be a non-linear relation between the variables.

Detecting 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".

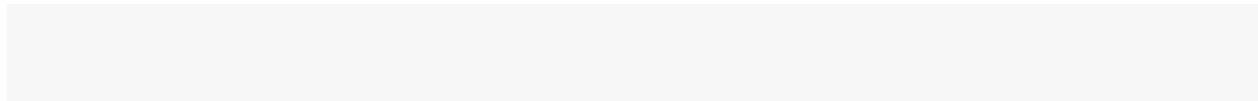
Recall the table `hybrid` that contains data on hybrid cars in the U.S. Here is a regression of fuel efficiency on the rate of acceleration. The association is negative: cars that accelerate quickly tend to be less efficient.

```
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. Uneven variation is often more easily noticed in a residual plot than in the original scatter plot.

If the residual plot shows uneven variation about the horizontal line at 0, the regression estimates are not equally accurate across the range of the predictor variable.



[Interact](#)

Numerical Diagnostics

In addition to visualization, we can use numerical properties of residuals to assess the quality of regression. We will not prove these properties mathematically. Rather, we will observe them by computation and see what they tell us about the regression.

All of the facts listed below hold for all shapes of scatter plots, whether or not they are linear.

Residual Plots Show No Trend

For every linear regression, whether good or bad, the residual plot shows no trend. Overall, it is flat. In other words, the residuals and the predictor variable are uncorrelated.

You can see this in all the residual plots above. We can also calculate the correlation between the predictor variable and the residuals in each case.

```
correlation(heights, 'MidParent', 'Residual')
```

```
-2.7196898076470642e-16
```

That doesn't look like zero, but it is a tiny number that is 0 apart from rounding error due to computation. Here it is again, correct to 10 decimal places. The minus sign is because of the rounding that above.

```
round(correlation(heights, 'MidParent', 'Residual'), 10)
```

```
-0.0
```

```
dugong = dugong.with_columns(  
    'Fitted Value', fit(dugong, 'Length', 'Age'),  
    'Residual', residual(dugong, 'Length', 'Age'))  
round(correlation(dugong, 'Length', 'Residual'), 10)
```

0 . 0

Average of Residuals

No matter what the shape of the scatter diagram, the average of the residuals is 0.

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.

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 children's heights based on parents' heights in Galton's dataset.

```
round(np.mean(heights.column('Residual'))), 10)
```

0 . 0

The same is true of the average of the residuals in the regression of the age of dugongs on their length. The mean of the residuals is 0, apart from rounding error.

```
round(np.mean(dugong.column('Residual'))), 10)
```

0 . 0

SD of the Residuals

No matter what the shape of the scatter plot, the SD of the residuals is a fraction of the SD of the response variable. The fraction is $\sqrt{1 - r^2}$.

$$\text{SD of residuals} = \sqrt{1 - r^2} \cdot \text{SD of } y$$

We will soon see how this measures the accuracy of the regression estimate. But first, let's confirm it by example.

In the case of children's heights and midparent heights, the SD of the residuals is about 3.39 inches.

```
np.std(heights.column('Residual'))
```

3.3880799163953426

That's the same as $\sqrt{1 - r^2}$ times the SD of response variable:

```
r = correlation(heights, 'MidParent', 'Child')
np.sqrt(1 - r**2) * np.std(heights.column('Child'))
```

3.3880799163953421

The same is true for the regression of mileage on acceleration of hybrid cars. The correlation r is negative (about -0.5), but r^2 is positive and therefore $\sqrt{1 - r^2}$ is a fraction.

```
r = correlation(hybrid, 'acceleration', 'mpg')
r
```

-0.5060703843771186

```
hybrid = hybrid.with_columns(
    'fitted mpg', fit(hybrid, 'acceleration', 'mpg'),
    'residual', residual(hybrid, 'acceleration', 'mpg'))
np.std(hybrid.column('residual')), np.sqrt(1 -
r**2)*np.std(hybrid.column('mpg'))
```

(9.4327368334302903, 9.4327368334302903)

Now let us see how the SD of the residuals is a measure of how good the regression is. Remember that the average of the residuals is 0. Therefore the smaller the SD of the residuals is, the closer the residuals are to 0. In other words, if the SD of the residuals is small, the overall size of the errors in regression is small.

The extreme cases are when $r = 1$ or $r = -1$. In both cases, $\sqrt{1 - r^2} = 0$. Therefore the residuals have an average of 0 and an SD of 0 as well, and therefore the residuals are all equal to 0. The regression line does a perfect job of estimation. As we saw earlier in this chapter, if $r = \pm 1$, the scatter plot is a perfect straight line and is the same as the regression line, so indeed there is no error in the regression estimate.

But usually r is not at the extremes. If r is neither ± 1 nor 0, then $\sqrt{1 - r^2}$ is a proper fraction, and the rough overall size of the error of the regression estimate is somewhere between 0 and the SD of y .

The worst case is when $r = 0$. Then $\sqrt{1 - r^2} = 1$, and the SD of the residuals 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 the average of y . In this situation, the root mean square error of regression is the root mean squared deviation from the average of y , which is the SD of y . In practical terms, if $r = 0$ then there is no linear association between the two variables, so there is no benefit in using linear regression.

Another Way to Interpret r

We can rewrite the result above to say that no matter what the shape of the scatter plot,

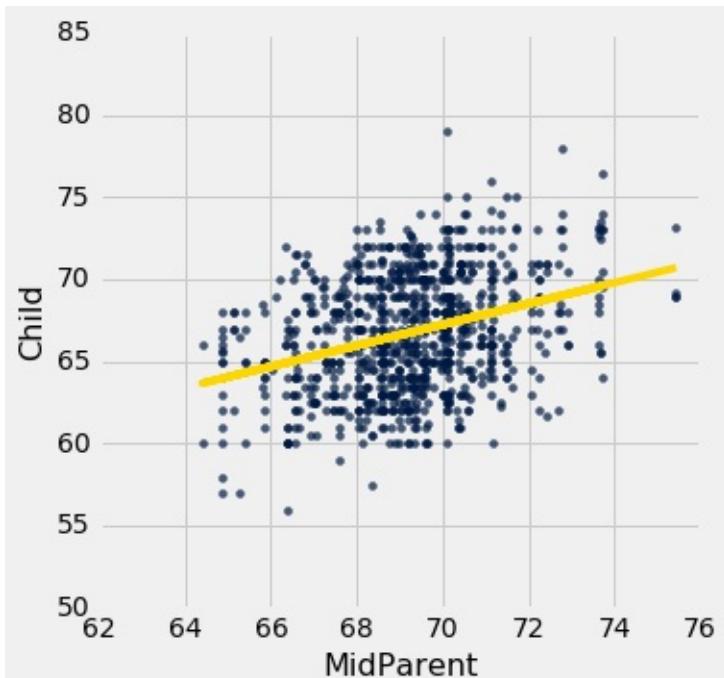
$$\frac{\text{SD of residuals}}{\text{SD of } y} = \sqrt{1 - r^2}$$

A complementary result is that no matter what the shape of the scatter plot, the SD of the fitted values is a fraction of the SD of the observed values of y . The fraction is $|r|$.

$$\frac{\text{SD of fitted values}}{\text{SD of } y} = |r|$$

To see where the fraction comes in, notice 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(heights, 'MidParent', 'Child')
```



The fitted values range from about 64 to about 71, whereas the heights of all the children are quite a bit more variable, ranging from about 55 to 80.

To verify the result numerically, we just have to calculate both sides of the identity.

```
correlation(heights, 'MidParent', 'Child')
```

```
0.32094989606395924
```

Here is ratio of the SD of the fitted values and the SD of the observed values of birth weight:

```
np.std(heights.column('Fitted
Value'))/np.std(heights.column('Child'))
```

```
0.32094989606395957
```

The ratio is equal to r , confirming our result.

Where does the absolute value come in? First note that as SDs can't be negative, nor can a ratio of SDs. So what happens when r is negative? The example of fuel efficiency and acceleration will show us.

```
correlation(hybrid, 'acceleration', 'mpg')
```

```
-0.5060703843771186
```

```
np.std(hybrid.column('fitted mpg'))/np.std(hybrid.column('mpg'))
```

```
0.5060703843771186
```

The ratio of the two SDs is $|r|$.

A more standard way to express this result is to recall that

$$\begin{aligned}\text{variance} &= \text{mean squared deviation from average} \\ &= \text{SD}^2\end{aligned}$$

and therefore, by squaring both sides of our result,

$$\frac{\text{variance of fitted values}}{\text{variance of } y} = r^2$$

[Interact](#)

Inference for Regression

Thus far, our analysis of the relation between variables has been purely descriptive. We know how to find the best straight line to draw through a scatter plot. The line is the best in the sense that it has the smallest mean squared error of estimation among all straight lines.

But what if our data were only a sample from a larger population? If in the sample we found a linear relation between the two variables, would the same be true for the population? Would it be exactly the same linear relation? Could we predict the response of a new individual who is not in our sample?

Such questions of inference and prediction 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*.

[Interact](#)

A Regression Model

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 it exists.
- The scatter plot is created by taking points on the line and pushing them off the line vertically, either above or below, as follows:
 - For each x , find the corresponding point on the true line (that's the signal), and then generate the noise or error.
 - The errors are drawn at random with replacement from a population of errors that has a normal distribution with mean 0.
 - Create a point whose horizontal coordinate is x and whose vertical coordinate is "the height of the true line at x , plus the error".
- Finally, erase the true line from the scatter, and display just the points created.

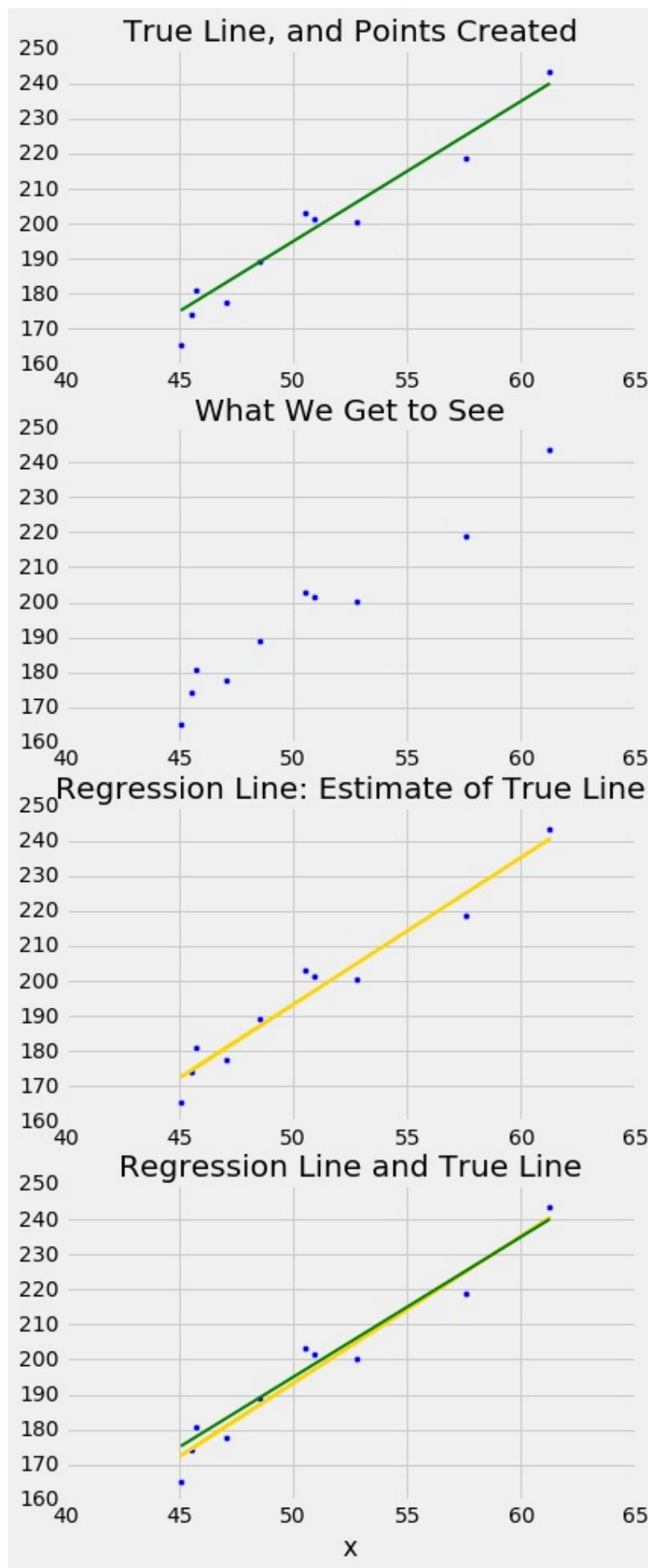
Based on this scatter plot, how should we estimate the true line? The best line that we can put through a scatter plot is the regression line. So the regression line is a natural estimate of the true line.

The simulation below shows how close the regression line is to the true line. The first panel shows how the scatter plot is generated from the true line. The second shows the scatter plot that we see. The third shows the regression line through the plot. 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.

```
# The true line,  
# the points created,  
# and our estimate of the true line.  
# Arguments: true slope, true intercept, number of points  
  
draw_and_compare(4, -5, 10)
```



In reality, of course, 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.



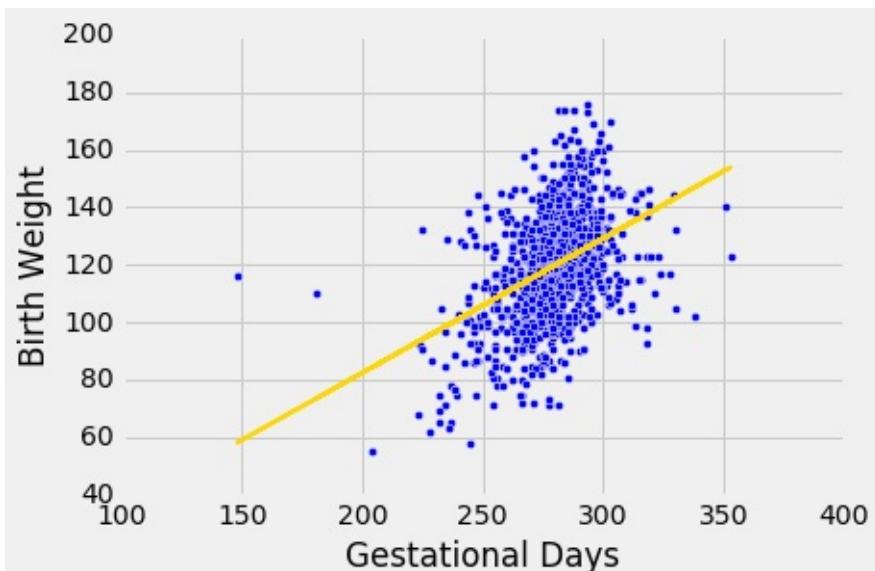
[Interact](#)

Inference for the True Slope

Our simulations show that if the regression model holds and the sample size is large, then the regression line is likely to be close to the true line. This allows us to estimate the slope of the true line.

We will use our familiar sample of mothers and their newborn babies to develop a method of estimating the slope of the true line. First, let's see if we believe that the regression model is an appropriate set of assumptions for describing the relation between birth weight and the number of gestational days.

```
scatter_fit(baby, 'Gestational Days', 'Birth Weight')
```



```
correlation(baby, 'Gestational Days', 'Birth Weight')
```

```
0.40754279338885108
```

By and large, the scatter looks fairly evenly distributed around the line, though there are some points that are scattered on the outskirts of the main cloud. The correlation is 0.4 and the regression line has a positive slope.

Does this reflect the fact that the true line has a positive slope? To answer this question, let us see if we can estimate the true slope. We certainly have one estimate of it: the slope of our regression line. That's about 0.47 ounces per day.

```
slope(baby, 'Gestational Days', 'Birth Weight')
```

```
0.46655687694921522
```

But had the scatter plot come out differently, the regression line would have been different and might have had a different slope. How do we figure out how different the slope might have been?

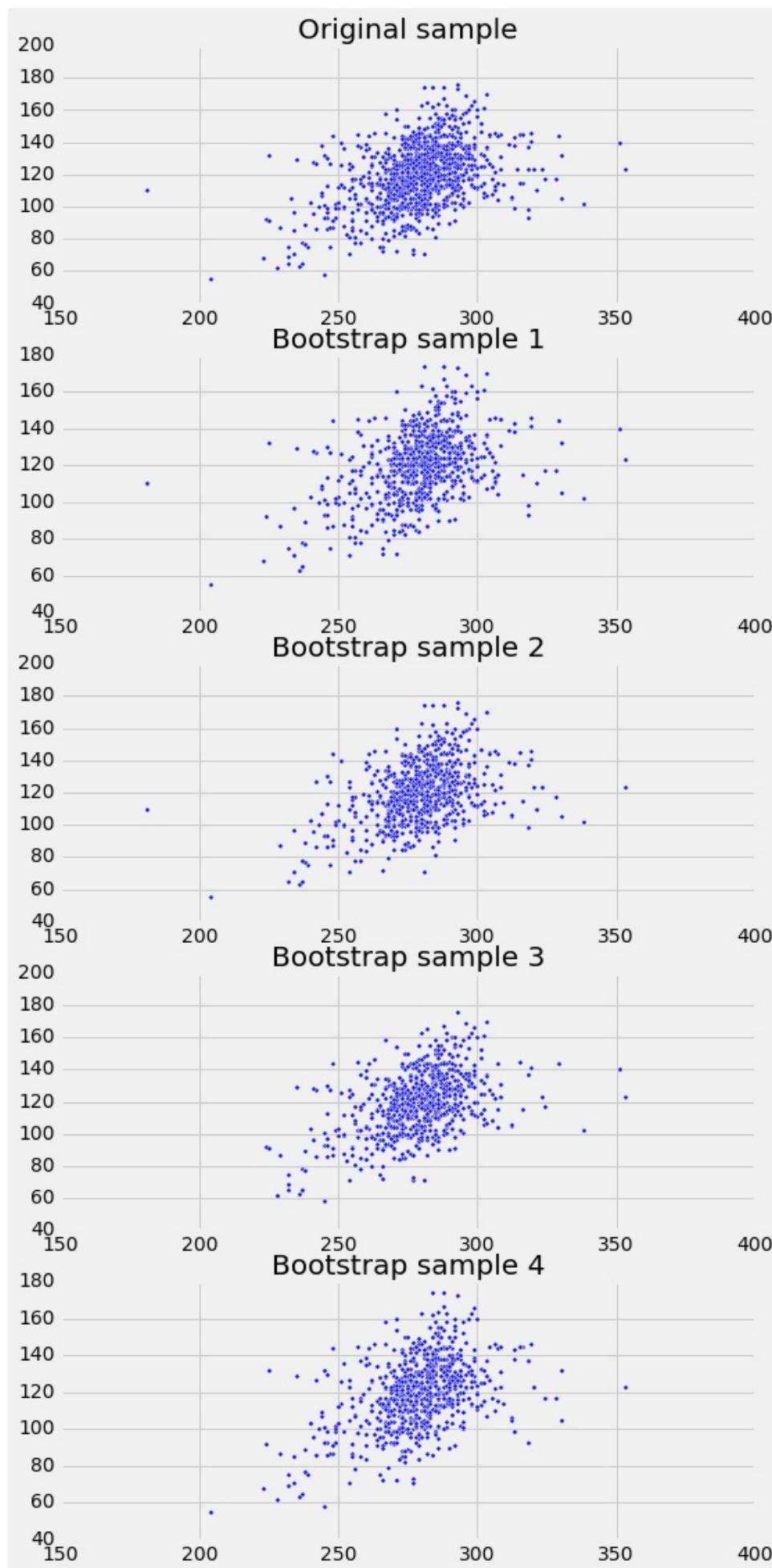
We need another sample of points, so that we can draw the regression line through the new scatter plot and find its slope. But from where will get another sample?

You have guessed it – we will *bootstrap our original sample*. That will give us a bootstrapped scatter plot, through which we can draw a regression line.

Bootstrapping the Scatter Plot

We can simulate new samples by random sampling with replacement from the original sample, as many times as the original sample size. Each of these new samples will give us a scatter plot. We will call that a *bootstrapped scatter plot*, and for short, we will call the entire process *bootstrapping the scatter plot*.

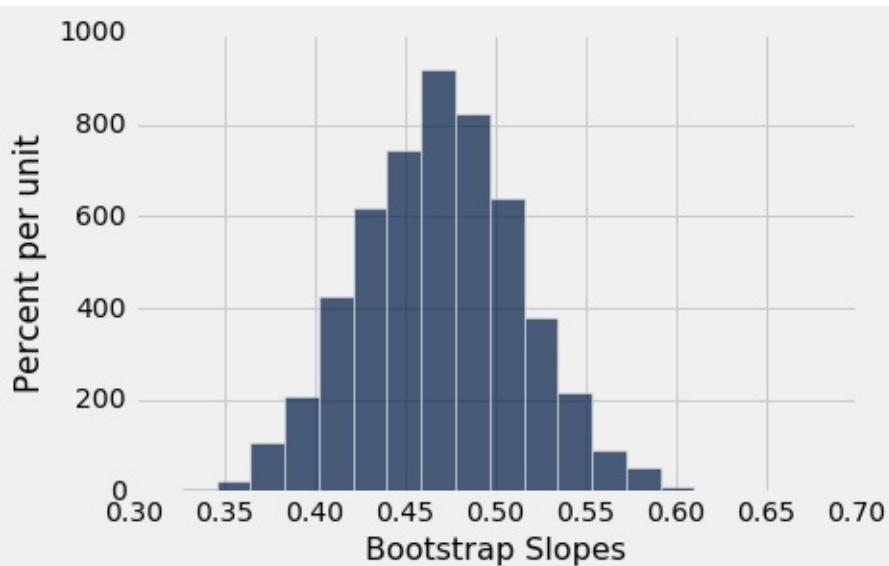
Here is the original scatter diagram from the sample, and four replications of the bootstrap resampling procedure. Notice how the resampled scatter plots are in general a little more sparse than the original. That is because some of the original points do not get selected in the samples.



Estimating the True Slope

We can bootstrap the scatter plot a large number of times, and draw a regression line through each bootstrapped plot. Each of those lines has a slope. We can simply collect all the slopes and draw their empirical histogram. Recall that by default, the `sample` method draws at random with replacement, the same number of times as the number of rows in the table. That is, `sample` generates a bootstrap sample by default.

```
slopes = make_array()
for i in np.arange(5000):
    bootstrap_sample = baby.sample()
    bootstrap_slope = slope(bootstrap_sample, 'Gestational
Days', 'Birth Weight')
    slopes = np.append(slopes, bootstrap_slope)
Table().with_column('Bootstrap Slopes', slopes).hist(bins=20)
```



We can then construct an approximate 95% confidence interval for the slope of the true line, using the bootstrap percentile method. The confidence interval extends from the 2.5th percentile to the 97.5th percentile of the 5000 bootstrapped slopes.

```
left = percentile(2.5, slopes)
right = percentile(97.5, slopes)
left, right
```

```
(0.38209399211893086, 0.56014757838023777)
```

An approximate 95% confidence interval for the true slope extends from about 0.38 ounces per day to about 0.56 ounces per day.

A Function to Bootstrap the Slope

Let us collect all the steps of our method of estimating the slope and define a function `bootstrap_slope` that carries them out. 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.

```

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 = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = table.sample()
        bootstrap_slope = slope(bootstrap_sample, x, y)
        slopes = np.append(slopes, bootstrap_slope)

    # Find the endpoints of the 95% confidence interval for the
    true slope
    left = percentile(2.5, slopes)
    right = percentile(97.5, slopes)

    # Slope of the regression line from the original sample
    observed_slope = slope(table, x, y)

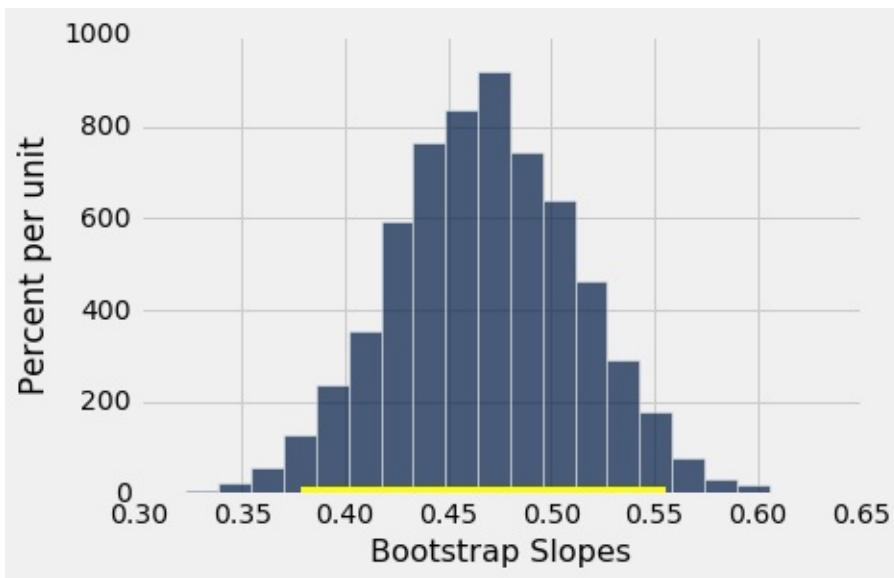
    # Display results
    Table().with_column('Bootstrap Slopes',
    slopes).hist(bins=20)
    plots.plot(make_array(left, right), make_array(0, 0),
    color='yellow', lw=8);
    print('Slope of regression line:', observed_slope)
    print('Approximate 95%-confidence interval for the true
    slope:')
    print(left, right)

```

When we call `bootstrap_slope` to find a confidence interval for the true slope when the response variable is birth weight and the predictor is gestational days, we get an interval very close to the one we obtained earlier: approximately 0.38 ounces per day to 0.56 ounces per day.

```
bootstrap_slope(baby, 'Gestational Days', 'Birth Weight', 5000)
```

```
Slope of regression line: 0.466556876949
Approximate 95%-confidence interval for the true slope:
0.378663152966 0.555005146304
```

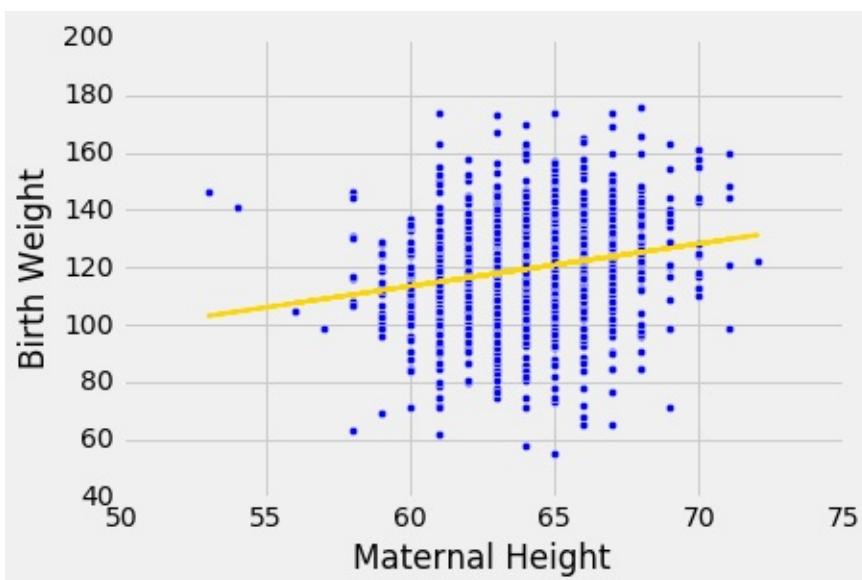


Now that we have a function that automates our process of estimating the slope of the true line in a regression model, we can use it on other variables as well.

For example, let's examine the relation between birth weight and the mother's height. Do taller women tend to have heavier babies?

The regression model seems reasonable, based on the scatter plot, but the correlation is not high. It's just about 0.2.

```
scatter_fit(baby, 'Maternal Height', 'Birth Weight')
```



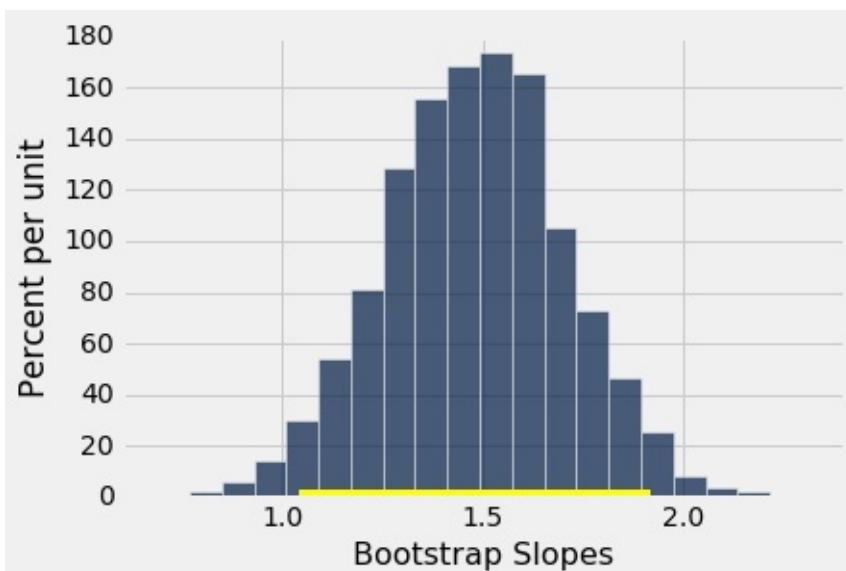
```
correlation(baby, 'Maternal Height', 'Birth Weight')
```

```
0.20370417718968034
```

As before, we can use `bootstrap_slope` to estimate the slope of the true line in the regression model.

```
bootstrap_slope(baby, 'Maternal Height', 'Birth Weight', 5000)
```

Slope of regression line: 1.47801935193
 Approximate 95%-confidence interval for the true slope:
 1.0403083964 1.91576886223



A 95% confidence interval for the true slope extends from about 1 ounce per inch to about 1.9 ounces per inch.

Could the True Slope Be 0?

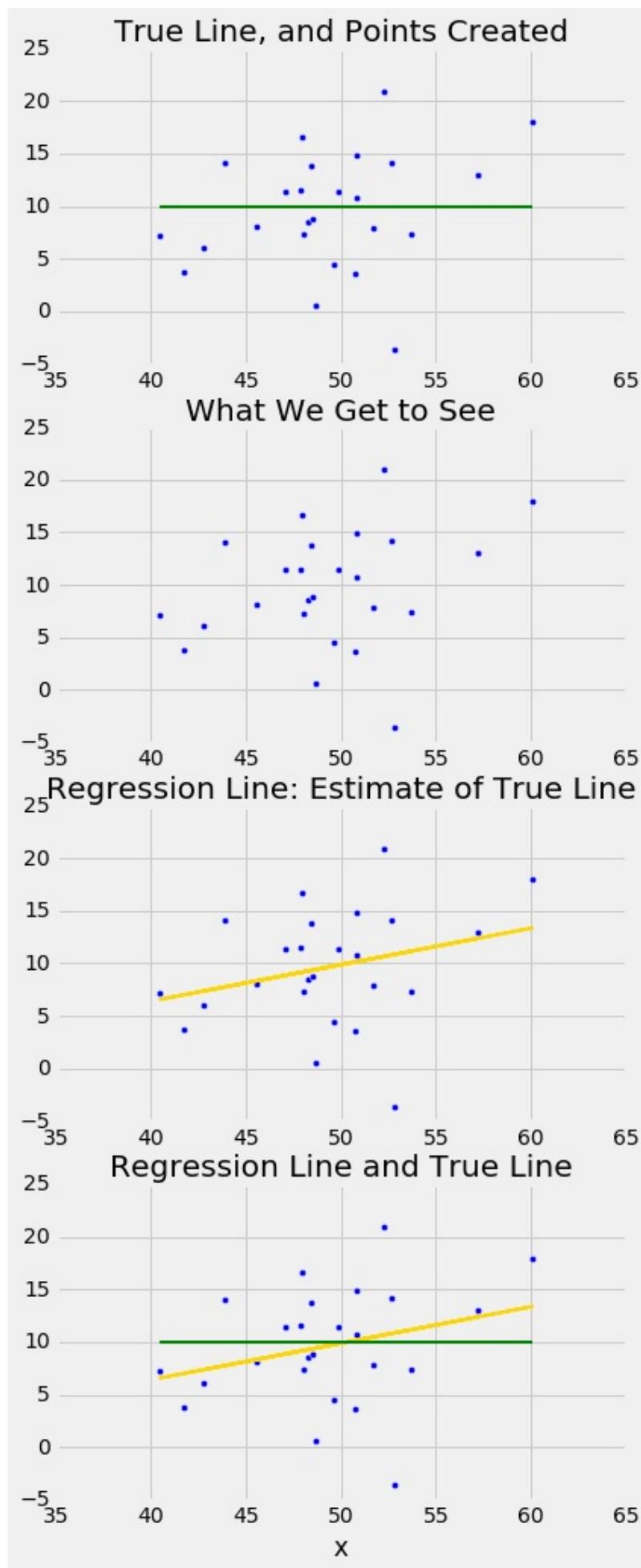
Suppose we believe that our data follow the regression model, and we fit the regression line to estimate the true line. If the regression line isn't perfectly flat, as is almost invariably the case, we will be observing some linear association in the scatter plot.

But what if that observation is spurious? In other words, what if the true line was flat – that is, there was no linear relation between the two variables – and the association that we observed was just due to randomness in generating the points that form our sample?

Here is a simulation that illustrates why this question arises. We will once again call the function `draw_and_compare`, this time requiring the 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 the true line, and the number of points to be generated.

```
draw_and_compare(0, 10, 25)
```



Run the simulation a few times, keeping the slope of the true line 0 each time. You will notice that while the slope of the true line is 0, the slope of the regression line is typically not 0. The regression line sometimes slopes upwards, and sometimes downwards, each time giving us a false impression that the two variables are correlated.

To decide whether or not the slope that we are seeing is real, we would like to test the following hypotheses:

Null Hypothesis. The slope of the true line is 0.

Alternative Hypothesis. The slope of the true line is not 0.

We are well positioned to do this. Since we can construct a 95% confidence interval for the true slope, all we have to do is see whether the interval contains 0.

If it doesn't, then we can reject the null hypothesis (with the 5% cutoff for the P-value).

If the confidence interval for the true slope does contain 0, then we don't have enough evidence to reject the null hypothesis. Perhaps the slope that we are seeing is spurious.

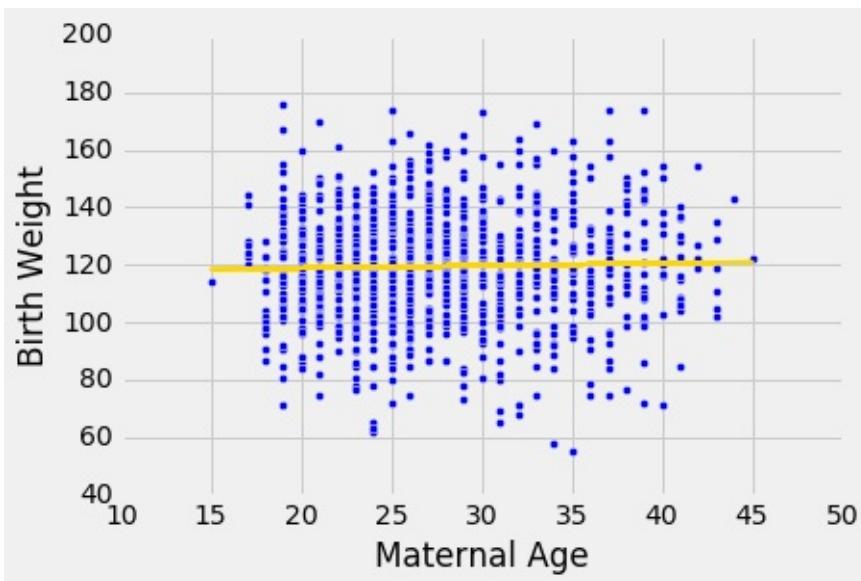
Let's use this method in an example. Suppose we try to estimate the birth weight of the baby based on the mother's age. Based on the sample, the slope of the regression line for estimating birth weight based on maternal age is positive, about 0.08 ounces per year.

```
slope(baby, 'Maternal Age', 'Birth Weight')
```

```
0.085007669415825132
```

Though the slope is positive, it's pretty small. The regression line is so close to flat that it raises the question of whether the true line is flat.

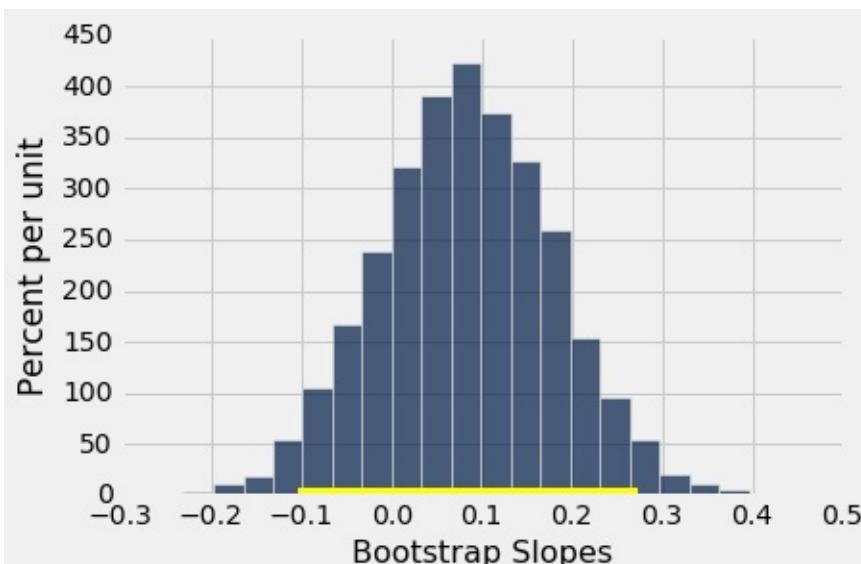
```
scatter_fit(baby, 'Maternal Age', 'Birth Weight')
```



We can use `bootstrap_slope` to estimate the slope of the true line. The calculation shows that 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.

```
bootstrap_slope(baby, 'Maternal Age', 'Birth Weight', 5000)
```

```
Slope of regression line: 0.0850076694158
Approximate 95%-confidence interval for the true slope:
-0.104335243815 0.272791852339
```



Because the interval contains 0, we cannot reject the null hypothesis that the slope of the true linear relation between maternal age and baby's birth weight is 0. Based on this analysis, it would be unwise to predict birth weight based on the regression model with maternal age as the predictor.



[Interact](#)

Prediction Intervals

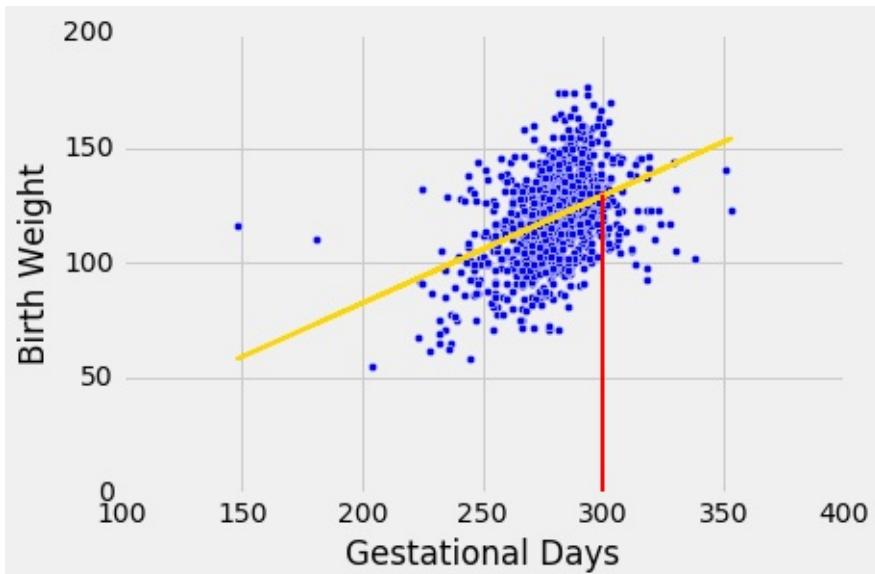
One of the primary uses of regression is to make predictions for a new individual who was not part of our original sample but is similar to the sampled individuals. In the language of the model, we want to estimate y for a new value of x .

Our estimate is the height of the true line at x . Of course, we don't know the true line. What we have as a substitute is the regression line through our sample of points.

The **fitted value** at a given value of x is the regression 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 .

Suppose we try to predict a baby's birth weight based on the number of gestational days. As we saw in the previous section, the data fit the regression model fairly well and a 95% confidence interval for the slope of the true line doesn't contain 0. So it seems reasonable to carry out our prediction.

The figure below shows where the prediction lies on the regression line. The red line is at $x = 300$.



The height of the point where the red line hits the regression line is the fitted value at 300 gestational days.

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. In other words, for a pregnancy that has a duration of 300 gestational days, our estimate for the baby's weight is about 129.2 ounces.

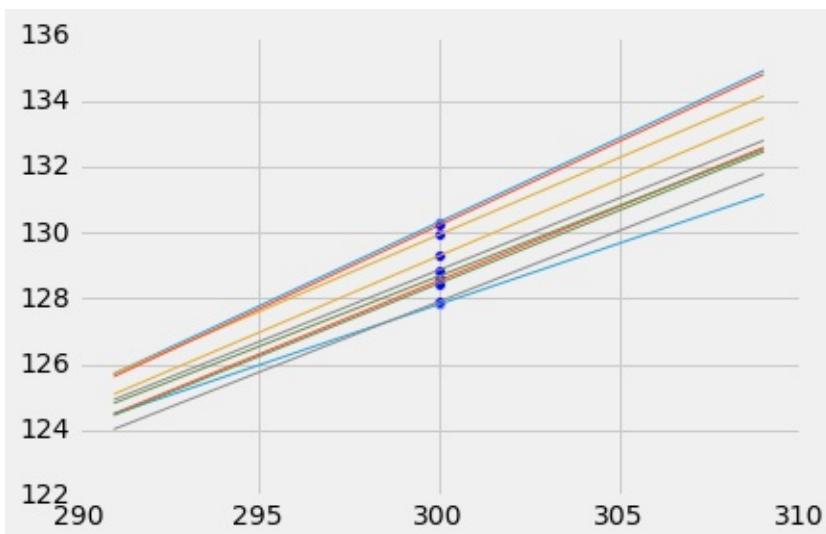
```
fit_300 = fitted_value(baby, 'Gestational Days', 'Birth Weight',
300)
fit_300
```

129.2129241703143

The Variability of the Prediction

We have developed a method making one prediction of a new baby's birth weight based on the number of gestational days, using the data in our sample. 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.

To do this, we must generate new samples. We can do that by bootstrapping the scatter plot as in the previous section. We will then 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.



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 $x=300$
0.503931	-21.6998	129.479
0.53227	-29.5647	130.116
0.518771	-25.363	130.268
0.430556	-1.06812	128.099
0.470229	-11.7611	129.308
0.48713	-16.5314	129.608
0.51241	-23.2954	130.428
0.52473	-27.2053	130.214
0.409943	5.22652	128.21
0.468065	-11.6967	128.723

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 an interval of predictions, using the same percentile method that we used to create a bootstrap confidence interval for the slope.

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 finds 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 find the fitted value at the specified x .

Finally, it draws the empirical histogram of all the predicted 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
scatter plot

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 = make_array()
    for i in np.arange(repetitions):
        bootstrap_sample = table.sample()
        bootstrap_prediction = fitted_value(bootstrap_sample, x,
y, new_x)
        predictions = np.append(predictions,
bootstrap_prediction)

    # Find the ends of the approximate 95% prediction interval
    left = percentile(2.5, predictions)
    right = percentile(97.5, predictions)

    # Prediction based on original sample
    original = fitted_value(table, x, y, new_x)

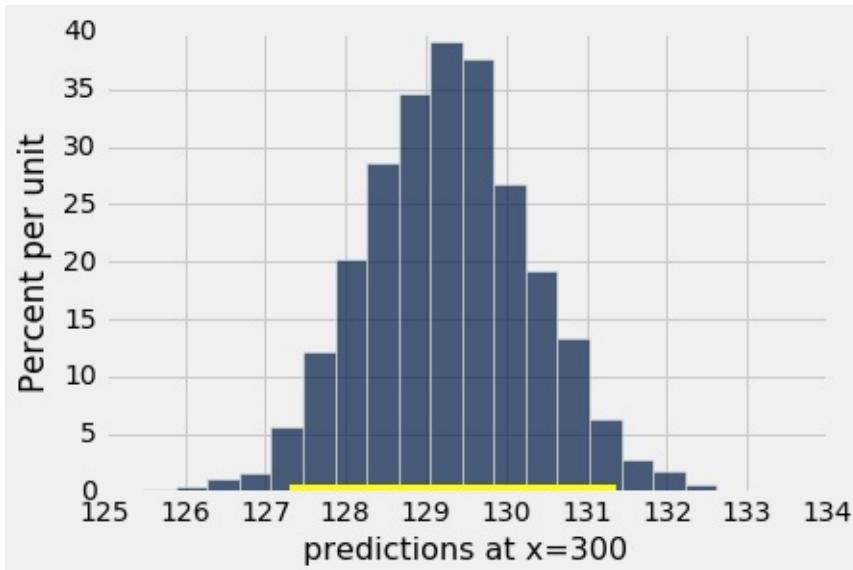
    # Display results
    Table().with_column('Prediction', predictions).hist(bins=20)
    plots.xlabel('predictions at x=' + str(new_x))
    plots.plot(make_array(left, right), make_array(0, 0),
color='yellow', lw=8);
    print('Height of regression line at x=' + str(new_x) + ':',
original)
    print('Approximate 95%-confidence interval:')
    print(left, right)
```

```
bootstrap_prediction(baby, 'Gestational Days', 'Birth Weight',
300, 5000)
```

Height of regression line at $x=300$: 129.21292417

Approximate 95%-confidence interval:

127.300774171 131.361729528



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.

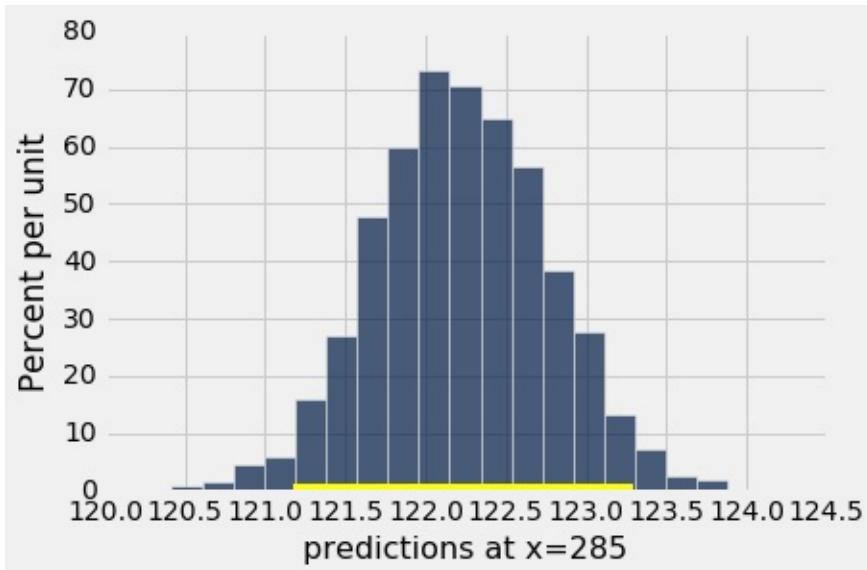
An approximate 95% prediction interval of scores has been constructed by taking the "middle 95%" of the predictions, that is, the interval from the 2.5th percentile to the 97.5th percentile of the predictions. The interval ranges from about 127 to about 131. The prediction based on the original sample was about 129, which is close to the center of the interval.

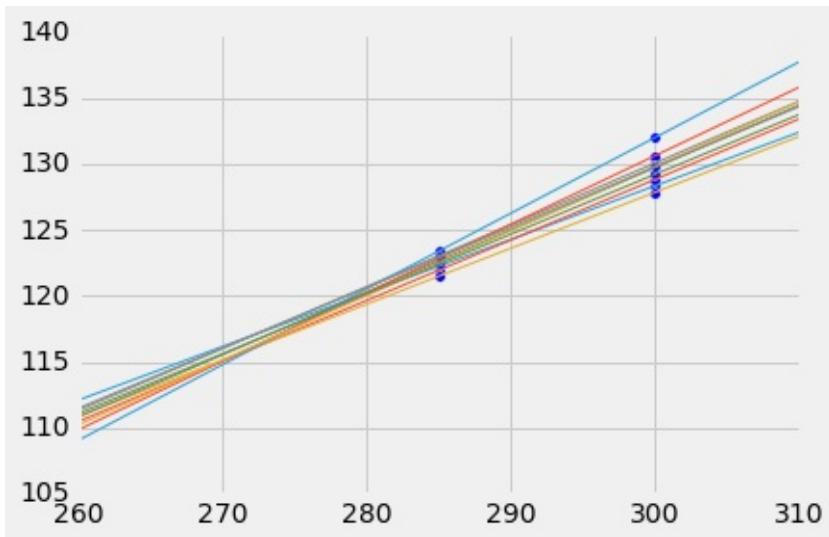
The Effect of Changing the Value of the Predictor

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, 5000)
```

```
Height of regression line at x=285: 122.214571016
Approximate 95%-confidence interval:
121.177089926 123.291373304
```





Words of caution

All of the predictions and tests that we have performed in this chapter 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 normal 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. We should also run the diagnostics we developed in the previous section using the residual plot.

[Interact](#)

Classification

David Wagner is the primary author of this chapter.

Machine learning is a class of techniques for automatically finding patterns in data and using it to draw inferences or make predictions. You have already seen linear regression, which is one kind of machine learning. This chapter introduces a new one: *classification*.

Classification is about learning how to make predictions from past examples. We are 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 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 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, each individual or situation where we'd like to make a prediction is called an *observation*. We ordinarily have many observations. Each observation has multiple *attributes*, which are known (for example, the total value of the order on Amazon, or the voter's annual salary). Also, each observation has a *class*, which is the answer to the question we care about (for example, fraudulent or not, or voting for you or not).

When Amazon is predicting whether orders are fraudulent, each order corresponds to a single observation. Each observation has several attributes: 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. When a customer makes a new order, we do not observe whether it is fraudulent, but we do observe its attributes, and we will try to predict its class using those attributes.

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.

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.

[Interact](#)

Nearest Neighbors¶

In this section we'll develop the *nearest neighbor* method of classification. Just focus on the ideas for now and don't worry if some of the code is mysterious. Later in the chapter we'll see how to organize our ideas into code that performs the classification.

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.

```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose  
Random', 'Glucose')  
ckd
```

Age	Blood Pressure	Specific Gravity	Albumin	Sugar	Red Blood Cells	Pus Cell	Pus Cell Clump
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	not present
48	80	1.025	4	0	normal	abnormal	not present
69	70	1.01	3	4	normal	abnormal	not present
73	70	1.005	0	0	normal	normal	not present
73	80	1.02	2	0	abnormal	abnormal	not present
46	60	1.01	1	0	normal	normal	not present

... (148 rows omitted)

Some of the variables are categorical (words like "abnormal"), and some quantitative. The quantitative variables all have different scales. We're going to want to make comparisons and estimate distances, often by eye, so let's select just a few of the variables and work in standard units. Then we won't have to worry about the scale of each of the different variables.

```
ckd = Table().with_columns(
    'Hemoglobin', standard_units(ckd.column('Hemoglobin')),
    'Glucose', standard_units(ckd.column('Glucose')),
    'White Blood Cell Count', standard_units(ckd.column('White
Blood Cell Count')),
    'Class', ckd.column('Class')
)
```

```
ckd
```

Hemoglobin	Glucose	White Blood Cell Count	Class
-0.865744	-0.221549	-0.569768	1
-1.45745	-0.947597	1.16268	1
-1.00497	3.84123	-1.27558	1
-2.81488	0.396364	0.809777	1
-2.08395	0.643529	0.232293	1
-1.35303	-0.561402	-0.505603	1
-0.413266	2.04928	0.360623	1
-1.28342	-0.947597	3.34429	1
-1.10939	1.87936	-0.409356	1
-1.35303	0.489051	1.96475	1
... (148 rows omitted)			

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).

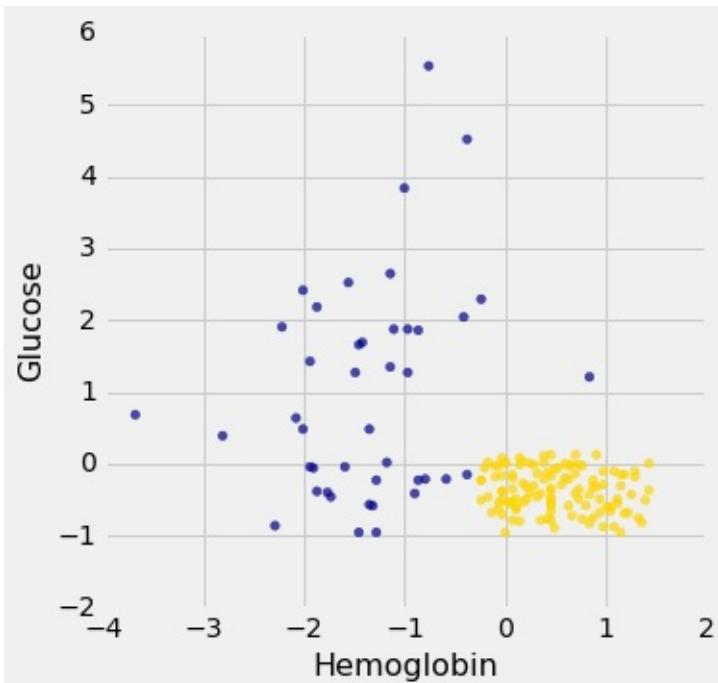
We'll draw a scatter plot to visualize the relation between the two variables. Blue dots are patients with CKD; gold dots are patients without CKD. What kind of medical test results seem to indicate CKD?

```

color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
ckd = ckd.join('Class', color_table)

```

```
ckd.scatter('Hemoglobin', 'Glucose', colors='Color')
```



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?

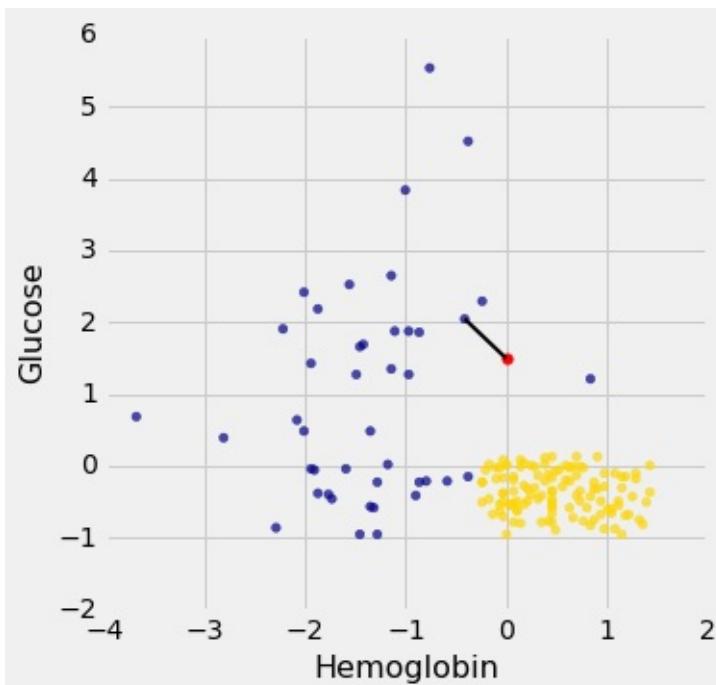
A Nearest Neighbor Classifier

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 blue or gold; 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.

In the graph below, the red dot represents Alice. It is joined with a black line to the point that is nearest to it – its *nearest neighbor* in the training set. The figure is drawn by a function called `show_closest`. It takes an array that represents the x and y coordinates of Alice's point. Vary those to see how the closest point changes! Note especially when the closest point is blue and when it is gold.

```
# In this example, Alice's Hemoglobin attribute is 0 and her
Glucose is 1.5.
alice = make_array(0, 1.5)
show_closest(alice)
```



Thus our *nearest neighbor classifier* works like this:

- Find the point in the training set that is nearest to the new point.
- If that nearest point is a "CKD" point, classify the new point as "CKD". If the nearest point is a "not CKD" point, classify the new point as "not CKD".

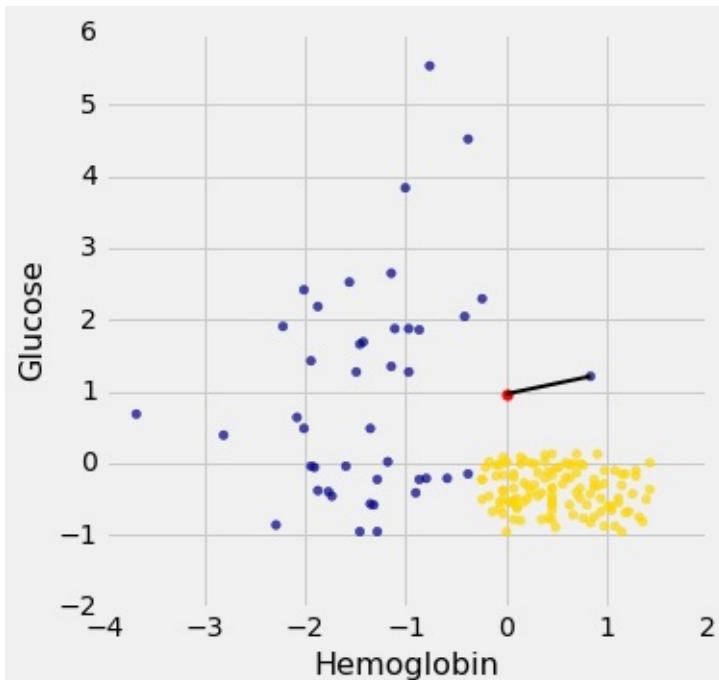
The scatterplot suggests that this nearest neighbor classifier should be pretty accurate. Points in the lower-right will tend to receive a "no CKD" diagnosis, as their nearest neighbor will be a gold point. The rest of the points will tend to receive a "CKD" diagnosis, as their nearest neighbor will be a blue point. So the nearest neighbor strategy seems to capture our intuition pretty well, for this example.

Decision boundary

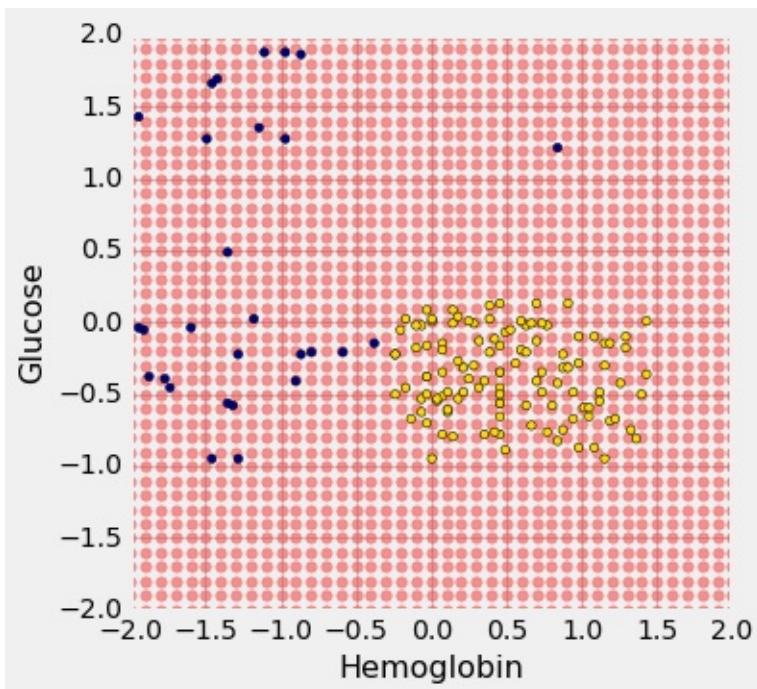
Sometimes a helpful way to visualize a classifier is to map out the kinds of attributes where the classifier would predict 'CKD', and the kinds where it would predict 'not CKD'. We end up with some boundary between the two, where points on one side of the boundary will be classified 'CKD' and points 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.

For example, suppose the coordinates of Alice's point are (0, 1.5). Notice that the nearest neighbor is blue. Now try reducing the height (the y -coordinate) of the point. You'll see that at around $y = 0.95$ the nearest neighbor turns from blue to gold.

```
alice = make_array(0, 0.97)
show_nearest(alice)
```



Here are hundreds of new unclassified points, all in red.

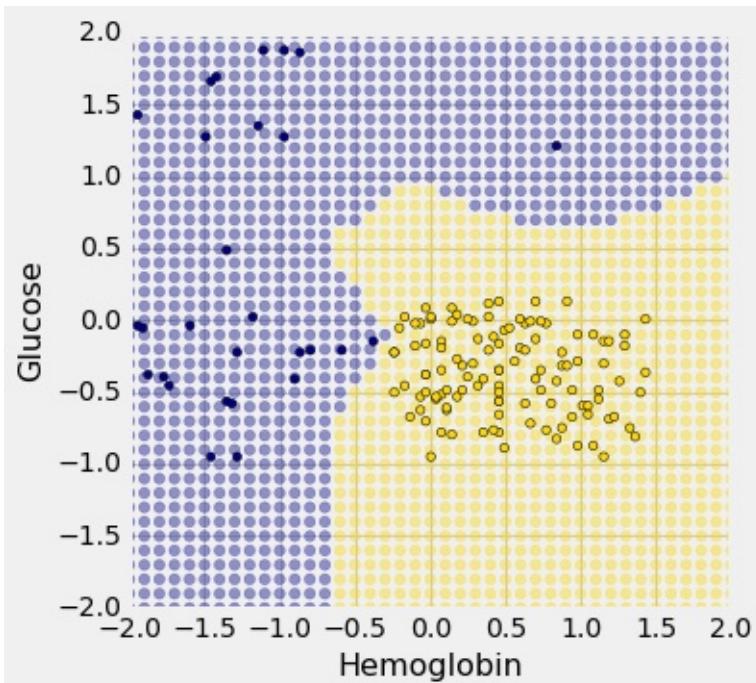


Each of the red points has a nearest neighbor in the training set (the same blue and gold points as before). For some red points you can easily tell whether the nearest neighbor is blue or gold. For others, it's a little more tricky to make the decision by eye. Those are the points near the decision boundary.

But the computer can easily determine the nearest neighbor of each point. So let's get it to apply our nearest neighbor classifier to each of the red points:

For each red point, it must find the closest point in the training set; it must then change the color of the red point to become the color of the nearest neighbor.

The resulting graph shows which points will get classified as 'CKD' (all the blue ones), and which as 'not CKD' (all the gold ones).

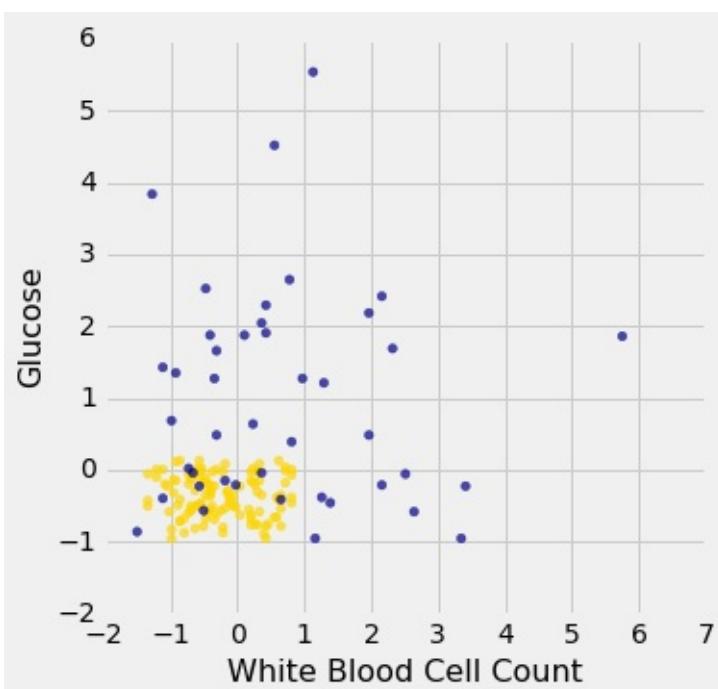


The decision boundary is where the classifier switches from turning the red points blue to turning them gold.

k-Nearest Neighbors

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', colors='Color')
```



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 blue dots scattered among the gold 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 scatter plot; 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 blue 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.

[Interact](#)

Training and Testing

How good is our nearest neighbor classifier? To answer this we'll need to find out how frequently our classifications are correct. If a patient has chronic kidney disease, how likely is our classifier to pick that up?

If the patient is in our training set, we can find out immediately. We already know what class the patient is in. So we can just compare our prediction and the patient's true class.

But the point of the classifier is to make predictions for *new* patients not in our training set. We don't know what class these patients are in but we can make a prediction based on our classifier. How to find out whether the prediction is correct?

One way is to wait for further medical tests on the patient and then check whether or not our prediction agrees with the test results. With that approach, by the time we can say how likely our prediction is to be accurate, it is no longer useful for helping the patient.

Instead, we will try our classifier on some patients whose true classes are known. Then, we will compute the proportion of the time our classifier was correct. This proportion will serve as an estimate of the proportion of all new patients whose class our classifier will accurately predict. This is called *testing*.

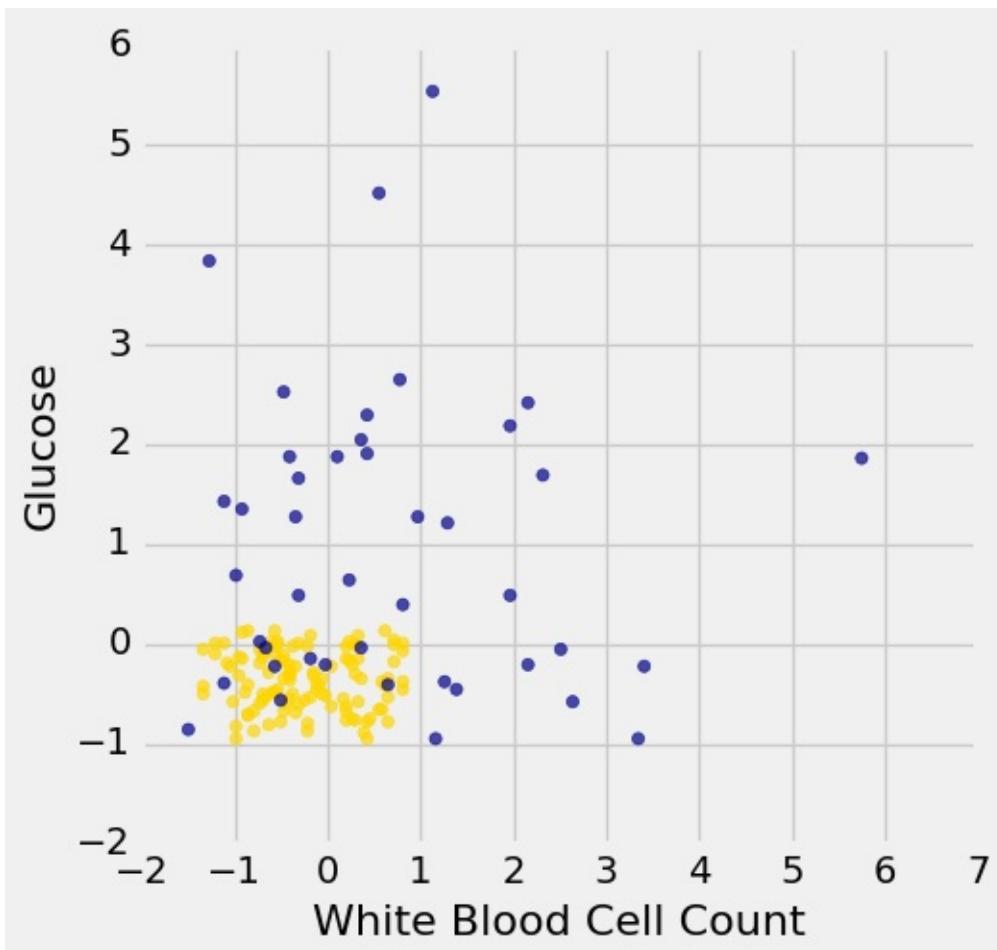
Overly Optimistic "Testing"

The training set offers a very tempting set of patients on whom to test out our classifier, because we know the class of each patient in the training set.

But let's be careful ... there will be pitfalls ahead if we take this path. An example will show us why.

Suppose we use a 1-nearest neighbor classifier to predict whether a patient has chronic kidney disease, based on glucose and white blood cell count.

```
ckd.scatter('White Blood Cell Count', 'Glucose', colors='Color')
```



Earlier, we said that we expect to get some classifications wrong, because there's some intermingling of blue and gold points in the lower-left.

But what about the points in the training set, that is, the points already on the scatter? Will we ever mis-classify them?

The answer is no. Remember that 1-nearest neighbor classification looks for the point *in the training set* that is nearest to the point being classified. Well, if the point being classified is already in the training set, then its nearest neighbor in the training set is itself! And therefore it will be classified as its own color, which will be correct because each point in the training set is already correctly colored.

In other words, **if we use our training set to "test" our 1-nearest neighbor classifier, the classifier will pass the test 100% of the time.**

Mission accomplished. What a great classifier!

No, not so much. A new point in the lower-left might easily be mis-classified, as we noted earlier. "100% accuracy" was a nice dream while it lasted.

The lesson of this example is *not* to use the training set to test a classifier that is based on it.

Generating a Test Set

In earlier chapters, we saw that random sampling could be used to estimate the proportion of individuals in a population that met some criterion. Unfortunately, we have just seen that the training set is not like a random sample from the population of all patients, in one important respect: Our classifier guesses correctly for a higher proportion of individuals in the training set than it does for individuals in the population.

When we computed confidence intervals for numerical parameters, we wanted to have many new random samples from a population, but we only had access to a single sample. We solved that problem by taking bootstrap resamples from our sample.

We will use an analogous idea to test our classifier. We will *create two samples out of the original training set*, use one of the samples as our training set, and *the other one for testing*.

So we will have three groups of individuals:

- a training set on which we can do any amount of exploration to build our classifier;
- a separate testing set on which to try out our classifier and see what fraction of times it classifies correctly;
- the underlying population of individuals for whom we don't know the true classes; the hope is that our classifier will succeed about as well for these individuals as it did for our testing set.

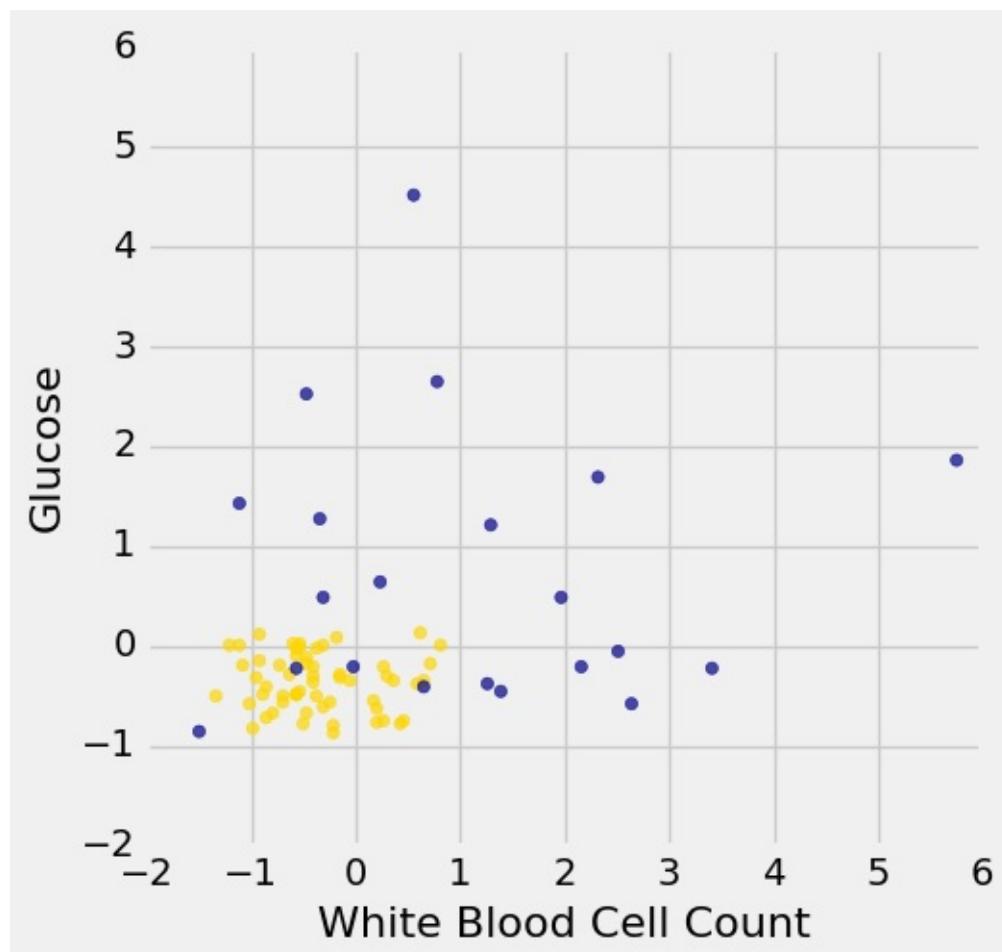
How to generate the training and testing sets? You've guessed it – we'll select at random.

There are 158 individuals in `ckd`. Let's use a random half of them for training and the other half for testing. To do this, we'll shuffle all the rows, take the first 79 as the training set, and the remaining 79 for testing.

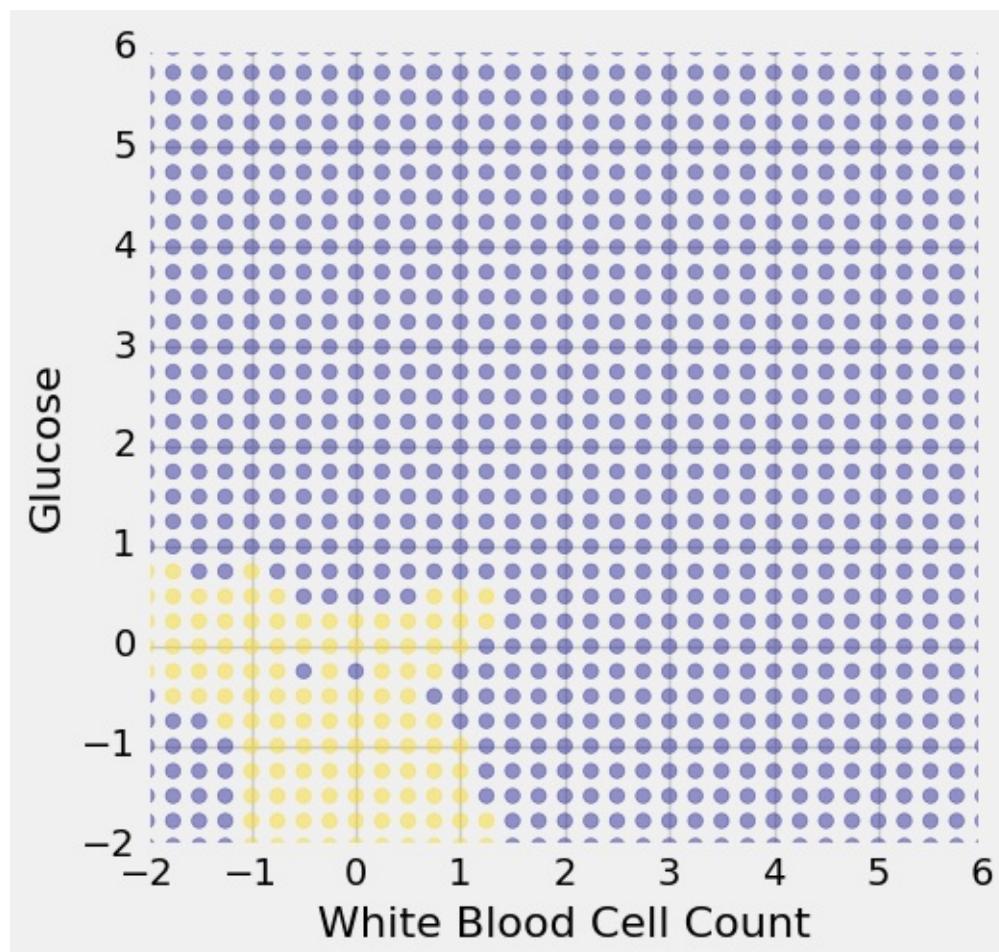
```
shuffled_ckd = ckd.sample(with_replacement=False)
training = shuffled_ckd.take(np.arange(79))
testing = shuffled_ckd.take(np.arange(79, 158))
```

Now let's construct our classifier based on the points in the training sample:

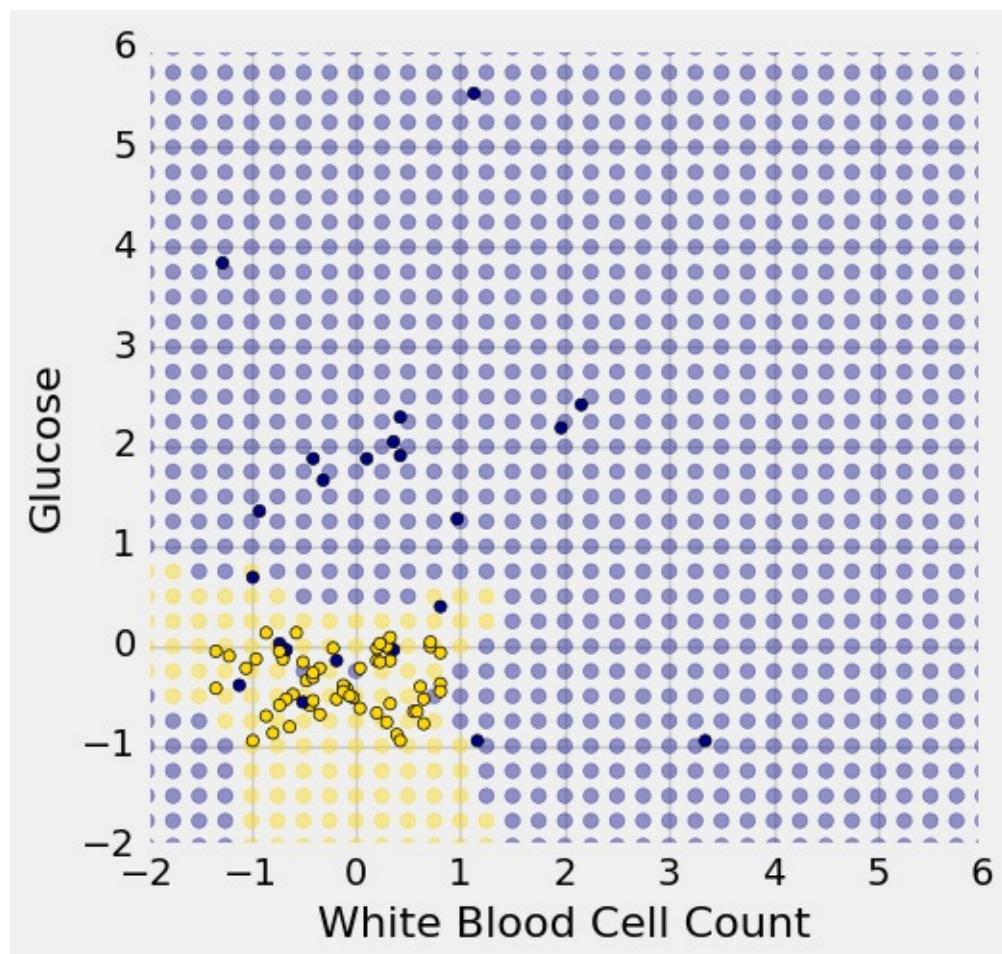
```
training.scatter('White Blood Cell Count', 'Glucose',
colors='Color')
plt.xlim(-2, 6)
plt.ylim(-2, 6);
```



We get the following classification regions and decision boundary:



Place the *test* data on this graph and you can see at once that while the classifier got almost all the points right, there are some mistakes. For example, some blue points of the test set fall in the gold region of the classifier.



Some errors notwithstanding, it looks like the classifier does fairly well on the test set.

Assuming that the original sample was drawn randomly from the underlying population, the hope is that the classifier will perform with similar accuracy on the overall population, since the test set was chosen randomly from the original sample.

[Interact](#)

Rows of Tables

Now that we have a qualitative understanding of nearest neighbor classification, it's time to implement our classifier.

Until this chapter, we have worked mostly with single columns of tables. But now we have to see whether one *individual* is "close" to another. Data for individuals are contained in *rows* of tables.

So let's start by taking a closer look at rows.

Here is the original table `ckd` containing data on patients who were tested for chronic kidney disease.

```
ckd = Table.read_table('ckd.csv').relabeled('Blood Glucose  
Random', 'Glucose')
```

The data corresponding to the first patient is in row 0 of the table, consistent with Python's indexing system. The Table method `row` accesses the row by taking the index of the row as its argument:

```
ckd.row(0)
```

```
Row(Age=48, Blood Pressure=70, Specific  
Gravity=1.004999999999999, Albumin=4, Sugar=0, Red Blood  
Cells='normal', Pus Cell='abnormal', Pus Cell clumps='present',  
Bacteria='notpresent', Glucose=117, Blood Urea=56, Serum  
Creatinine=3.7999999999998, Sodium=111, Potassium=2.5,  
Hemoglobin=11.1999999999999, Packed Cell Volume=32, White  
Blood Cell Count=6700, Red Blood Cell Count=3.899999999999999,  
Hypertension='yes', Diabetes Mellitus='no', Coronary Artery  
Disease='no', Appetite='poor', Pedal Edema='yes', Anemia='yes',  
Class=1)
```

Rows have their very own data type: they are *row objects*. Notice how the display shows not only the values in the row but also the labels of the corresponding columns.

Rows are in general **not arrays**, as their elements can be of different types. For example, some of the elements of the row above are strings (like `'abnormal'`) and some are numerical. So the row can't be converted into an array.

However, rows share some characteristics with arrays. You can use `item` to access a particular element of a row. For example, to access the Albumin level of Patient 0, we can look at the labels in the printout of the row above to find that it's item 3:

```
ckd.row(0).item(3)
```

4

Converting Rows to Arrays (When Possible)¶

Rows whose elements are all numerical (or all strings) can be converted to arrays.

Converting a row to an array gives us access to arithmetic operations and other nice NumPy functions, so it is often useful.

Recall that in the previous section we tried to classify the patients as 'CKD' or 'not CKD', based on two attributes `Hemoglobin` and `Glucose`, both measured in standard units.

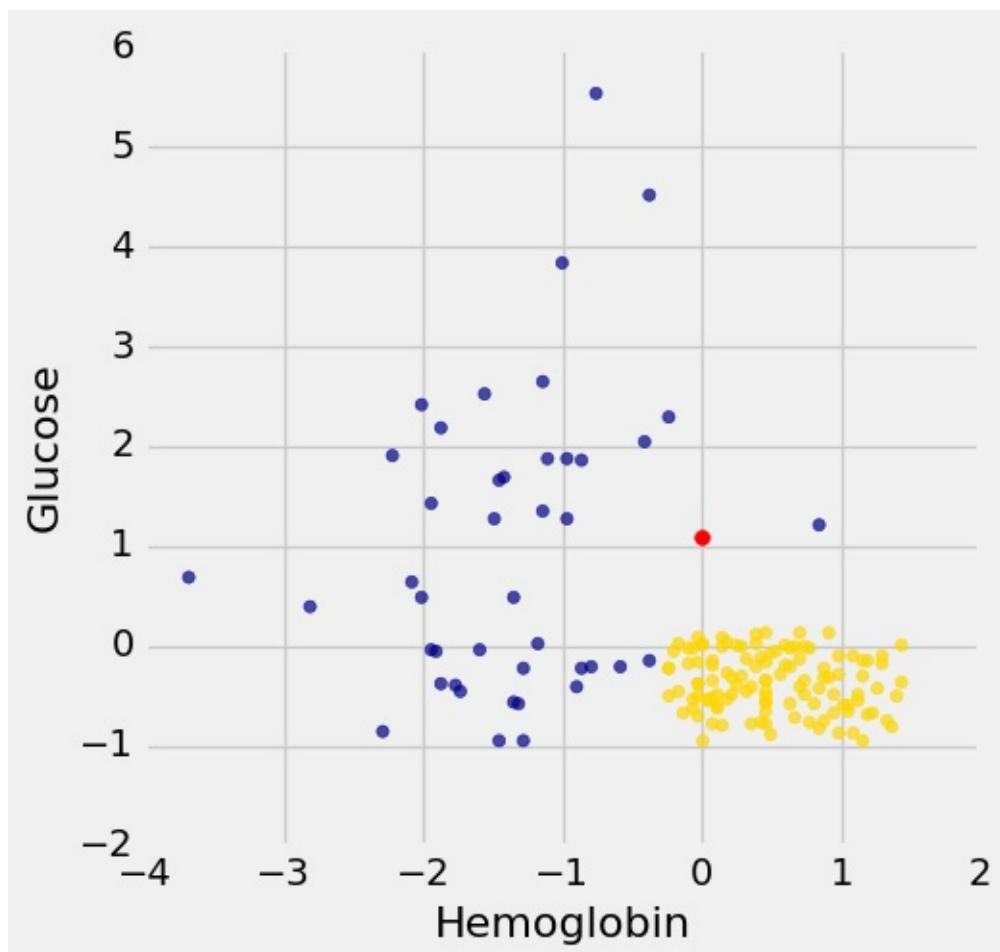
```
ckd = Table().with_columns(
    'Hemoglobin', standard_units(ckd.column('Hemoglobin')),
    'Glucose', standard_units(ckd.column('Glucose')),
    'Class', ckd.column('Class')
)

color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
ckd = ckd.join('Class', color_table)
ckd
```

Class	Hemoglobin	Glucose	Color
0	0.456884	0.133751	gold
0	1.153	-0.947597	gold
0	0.770138	-0.762223	gold
0	0.596108	-0.190654	gold
0	-0.239236	-0.49961	gold
0	-0.0304002	-0.159758	gold
0	0.282854	-0.00527964	gold
0	0.108824	-0.623193	gold
0	0.0740178	-0.515058	gold
0	0.83975	-0.422371	gold
... (148 rows omitted)			

Here is a scatter plot of the two attributes, along with a red point corresponding to Alice, a new patient. Her value of hemoglobin is 0 (that is, at the average) and glucose 1.1 (that is, 1.1 SDs above average).

```
alice = make_array(0, 1.1)
ckd.scatter('Hemoglobin', 'Glucose', colors='Color')
plots.scatter(alice.item(0), alice.item(1), color='red', s=30);
```



To find the distance between Alice's point and any of the other points, we only need the values of the attributes:

```
ckd_attributes = ckd.select('Hemoglobin', 'Glucose')
```

```
ckd_attributes
```

Hemoglobin	Glucose
0.456884	0.133751
1.153	-0.947597
0.770138	-0.762223
0.596108	-0.190654
-0.239236	-0.49961
-0.0304002	-0.159758
0.282854	-0.00527964
0.108824	-0.623193
0.0740178	-0.515058
0.83975	-0.422371

... (148 rows omitted)

Each row consists of the coordinates of one point in our training sample. **Because the rows now consist only of numerical values**, it is possible to convert them to arrays. For this, we use the function `np.array`, which converts any kind of sequential object, like a row, to an array. (Our old friend `make_array` is for *creating* arrays, not for *converting* other kinds of sequences to arrays.)

```
ckd_attributes.row(3)
```

```
Row(Hemoglobin=0.59610766482326683,
Glucose=-0.19065363034327712)
```

```
np.array(ckd_attributes.row(3))
```

```
array([ 0.59610766, -0.19065363])
```

This is very handy because we can now use array operations on the data in each row.

Distance Between Points When There are Two Attributes

The main calculation we need to do is to find the distance between Alice's point and any other point. For this, the first thing we need is a way to compute the distance between any pair of 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 hypotenuse.)

In the next section we'll see that this formula has a straightforward extension when there are more than two attributes. For now, let's use the formula and array operations to find the distance between Alice and the patient in Row 3.

```
patient3 = np.array(ckd_attributes.row(3))
alice, patient3
```

```
(array([ 0. ,  1.1]), array([ 0.59610766, -0.19065363]))
```

```
distance = np.sqrt(np.sum((alice - patient3)**2))
distance
```

```
1.4216649188818471
```

We're going to need the distance between Alice and a bunch of points, so let's write a function called `distance` that computes the distance between any pair of points. The function will take two arrays, each containing the (x, y) coordinates of a point. (Remember, those are really the Hemoglobin and Glucose levels of a patient.)

```
def distance(point1, point2):
    """Returns the Euclidean distance between point1 and point2.

    Each argument is an array containing the coordinates of a
    point."""
    return np.sqrt(np.sum((point1 - point2)**2))
```

```
distance(alice, patient3)
```

```
1.4216649188818471
```

We have begun to build our classifier: the `distance` function is the first building block. Now let's work on the next piece.

Using `apply` on an Entire Row

Recall that if you want to apply a function to each element of a column of a table, one way to do that is by the call `table_name.apply(function_name, column_label)`. This evaluates to an array consisting of the values of the function when we call it on each element of the column. So each entry of the array is based on the corresponding row of the table.

If you use `apply` without specifying a column label, then the entire row is passed to the function. Let's see how this works on a very small table `t` containing the information about the first five patients in the training sample.

```
t = ckd_attributes.take(np.arange(5))
t
```

Hemoglobin	Glucose
0.456884	0.133751
1.153	-0.947597
0.770138	-0.762223
0.596108	-0.190654
-0.239236	-0.49961

Just as an example, suppose that for each patient we want to know how unusual their most unusual attribute is. Concretely, if a patient's hemoglobin level is further from the average than her glucose level, we want to know how far it is from the average. If her glucose level is further from the average than her hemoglobin level, we want to know how far that is from the average instead.

That's the same as taking the maximum of the absolute values of the two quantities. To do this for a particular row, we can convert the row to an array and use array operations.

```
def max_abs(row):
    return np.max(np.abs(np.array(row)))
```

```
max_abs(t.row(4))
```

```
0.49961028259186968
```

And now we can apply `max_abs` to each row of the table `t`:

```
t.apply(max_abs)
```

```
array([ 0.4568837 ,  1.15300352,  0.77013762,  0.59610766,
       0.49961028])
```

This way of using `apply` will help us create the next building block of our classifier.

Alice's k Nearest Neighbors

If we want to classify Alice using a k -nearest neighbor classifier, we have to identify her k nearest neighbors. What are the steps in this process? Suppose $k = 5$. Then the steps are:

- **Step 1.** Find the distance between Alice and each point in the training sample.
- **Step 2.** Sort the data table in increasing order of the distances.
- **Step 3.** Take the top 5 rows of the sorted table.

Steps 2 and 3 seem straightforward, provided we have the distances. So let's focus on Step 1.

Here's Alice:

```
alice
```

```
array([ 0. ,  1.1])
```

What we need is a function that finds the distance between Alice and another point whose coordinates are contained in a row. The function `distance` returns the distance between any two points whose coordinates are in arrays. We can use that to define

`distance_from_alice`, which takes a row as its argument and returns the distance between that row and Alice.

```
def distance_from_alice(row):
    """Returns distance between Alice and a row of the
    attributes table"""
    return distance(alice, np.array(row))
```

```
distance_from_alice(ckd_attributes.row(3))
```

```
1.4216649188818471
```

Now we can `apply` the function `distance_from_alice` to each row of `ckd_attributes`, and augment the table `ckd` with the distances. Step 1 is complete!

```
distances = ckd_attributes.apply(distance_from_alice)
ckd_with_distances = ckd.with_column('Distance from Alice',
    distances)
```

```
ckd_with_distances
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
0	0.456884	0.133751	gold	1.06882
0	1.153	-0.947597	gold	2.34991
0	0.770138	-0.762223	gold	2.01519
0	0.596108	-0.190654	gold	1.42166
0	-0.239236	-0.49961	gold	1.6174
0	-0.0304002	-0.159758	gold	1.26012
0	0.282854	-0.00527964	gold	1.1409
0	0.108824	-0.623193	gold	1.72663
0	0.0740178	-0.515058	gold	1.61675
0	0.83975	-0.422371	gold	1.73862

... (148 rows omitted)

For Step 2, let's sort the table in increasing order of distance:

```
sorted_by_distance = ckd_with_distances.sort('Distance from Alice')
sorted_by_distance
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
1	0.83975	1.2151	darkblue	0.847601
1	-0.970162	1.27689	darkblue	0.986156
0	-0.0304002	0.0874074	gold	1.01305
0	0.14363	0.0874074	gold	1.02273
1	-0.413266	2.04928	darkblue	1.03534
0	0.387272	0.118303	gold	1.05532
0	0.456884	0.133751	gold	1.06882
0	0.178436	0.0410639	gold	1.07386
0	0.00440582	0.025616	gold	1.07439
0	-0.169624	0.025616	gold	1.08769

... (148 rows omitted)

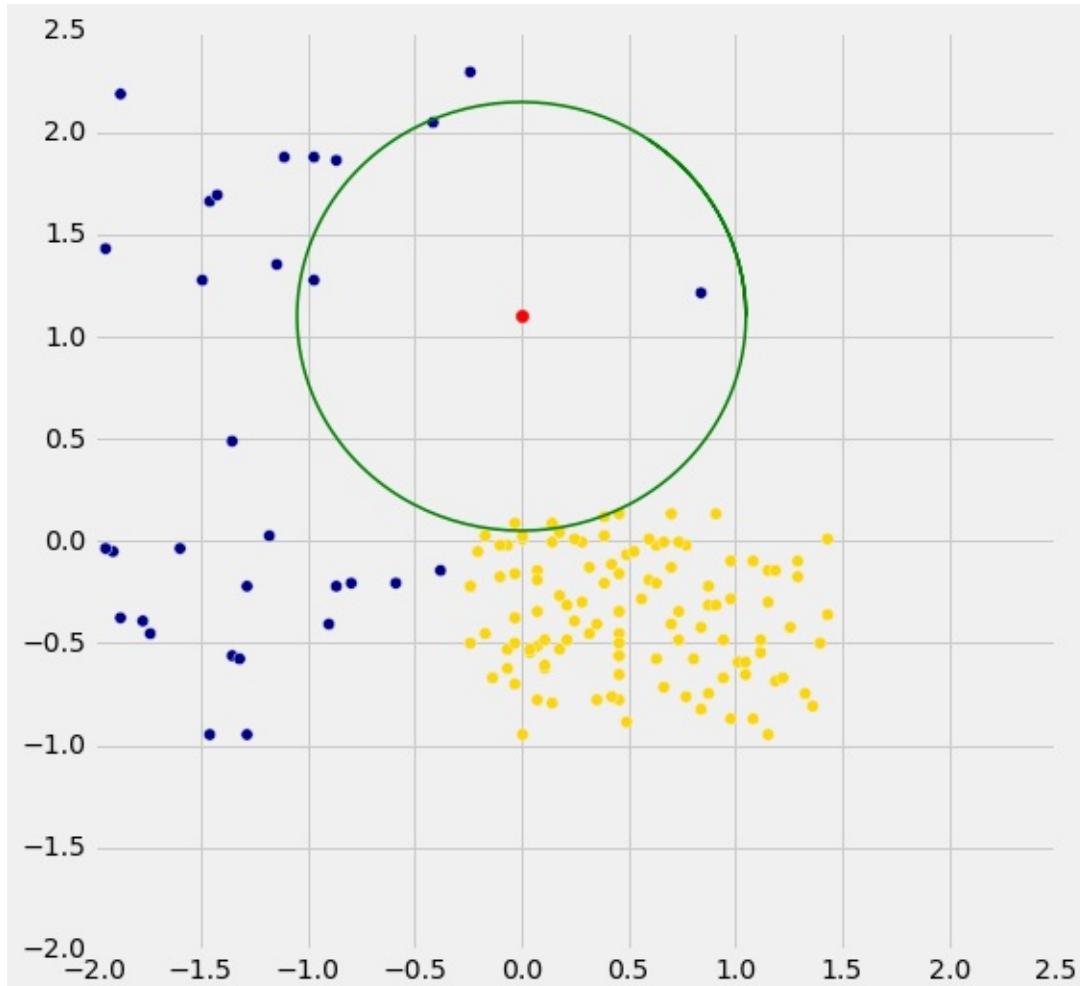
Step 3: The top 5 rows correspond to Alice's 5 nearest neighbors; you can replace 5 by any other positive integer.

```
alice_5_nearest_neighbors =
sorted_by_distance.take(np.arange(5))
alice_5_nearest_neighbors
```

Class	Hemoglobin	Glucose	Color	Distance from Alice
1	0.83975	1.2151	darkblue	0.847601
1	-0.970162	1.27689	darkblue	0.986156
0	-0.0304002	0.0874074	gold	1.01305
0	0.14363	0.0874074	gold	1.02273
1	-0.413266	2.04928	darkblue	1.03534

Three of Alice's five nearest neighbors are blue points and two are gold. So a 5-nearest neighbor classifier would classify Alice as blue: it would predict that Alice has chronic kidney disease.

The graph below zooms in on Alice and her five nearest neighbors. The two gold ones just inside the circle directly below the red point. The classifier says Alice is more like the three blue ones around her.



We are well on our way to implementing our k-nearest neighbor classifier. In the next two sections we will put it together and assess its accuracy.

[Interact](#)

Implementing the Classifier

We are now ready to implement a k -nearest neighbor classifier based on multiple attributes. We have used only two attributes so far, for ease of visualization. But usually predictions will be based on many attributes. Here is an example that shows how multiple attributes can be better than pairs.

Banknote authentication

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

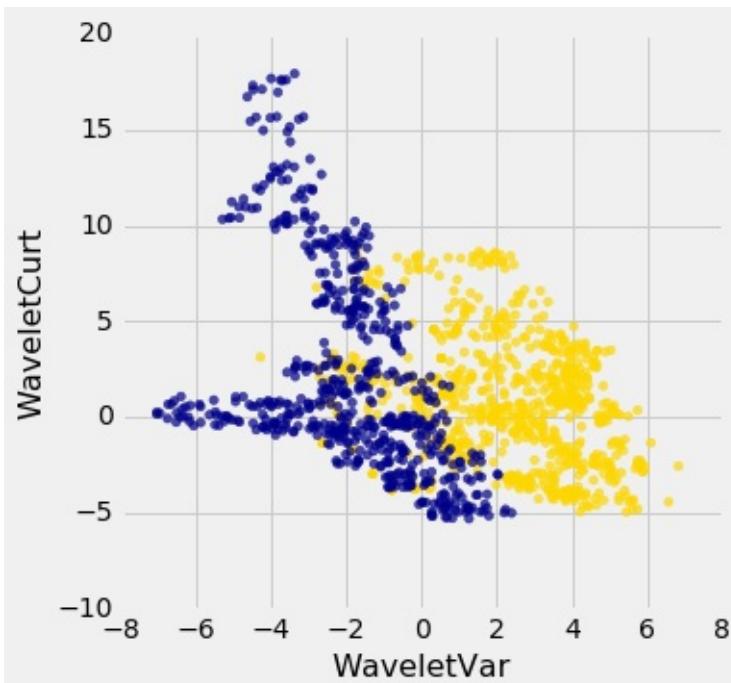
... (1362 rows omitted)

Let's look at whether the first two numbers tell us anything about whether the banknote is counterfeit or not. Here's a scatterplot:

```
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
```

```
banknotes = banknotes.join('Class', color_table)
```

```
banknotes.scatter('WaveletVar', 'WaveletCurt', colors='Color')
```

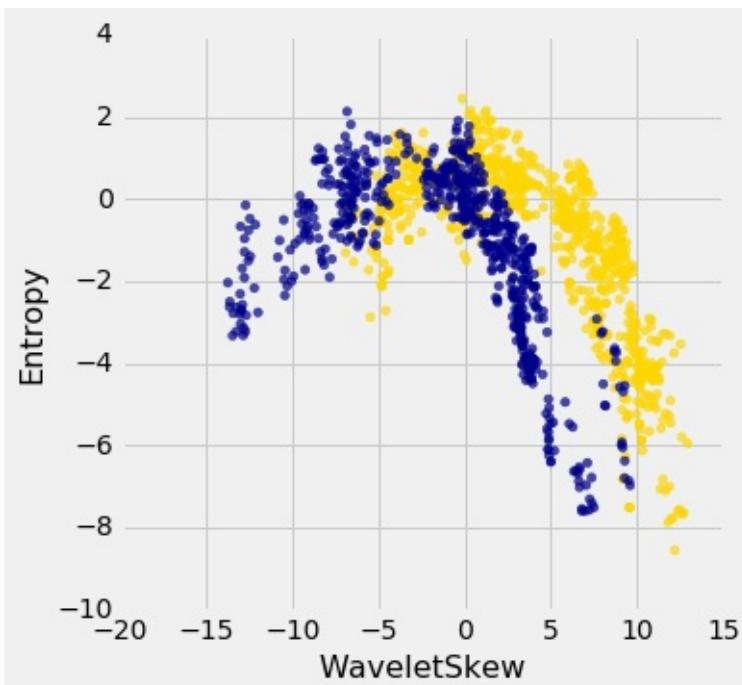


Pretty interesting! Those two measurements do seem helpful for predicting whether the banknote is counterfeit or not. However, in this example you can now see that there is some overlap between the blue cluster and the gold cluster. This indicates that there will be some images where it's hard to tell 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', colors='Color')
```



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

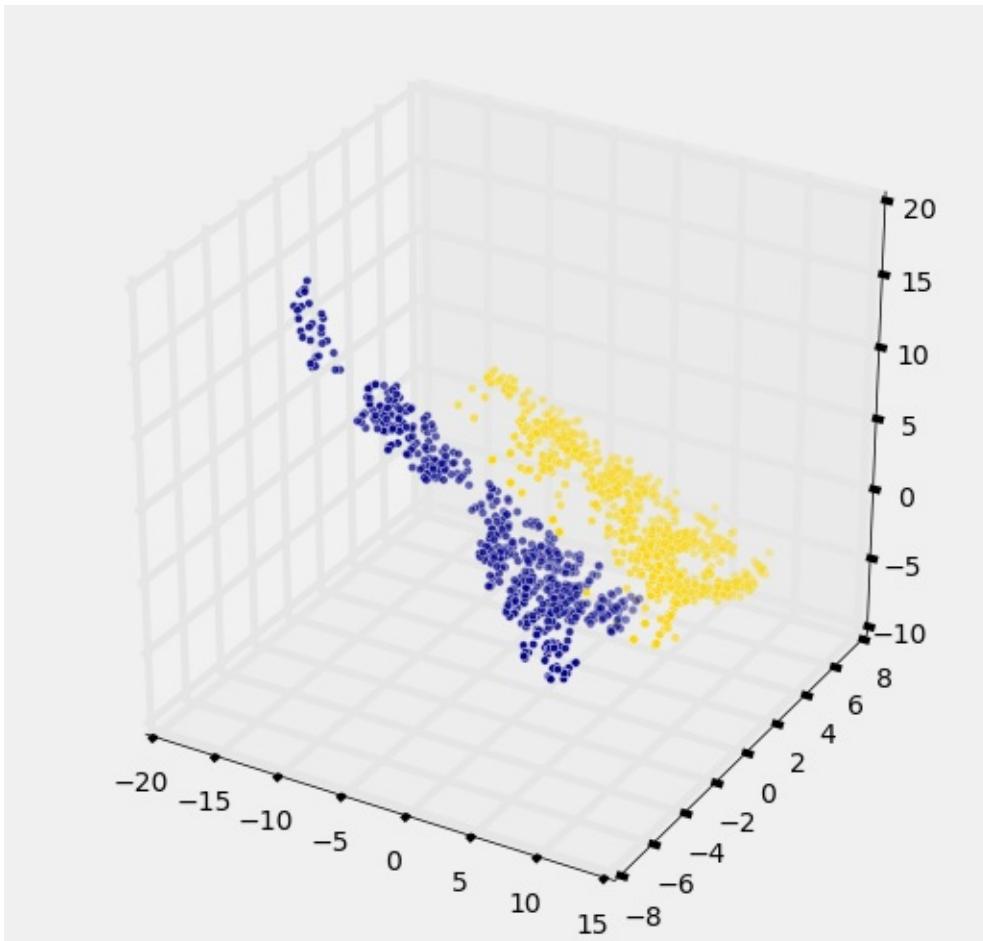
So far I've been assuming that we have exactly 2 attributes that we can use to help us make our prediction. What if we have more than 2? For instance, what if we have 3 attributes?

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('Color'));
```



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.

Distance in Multiple Dimensions

We know how to compute distance in 2-dimensional space. If we have a point at coordinates (x_0, y_0) and another at (x_1, y_1) , the distance between them is

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}.$$

In 3-dimensional space, the points are (x_0, y_0, z_0) and (x_1, y_1, z_1) , and the formula for the distance between them is

$$D = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

In n -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.

In the last section, we defined the function `distance` which returned the distance between two points. We used it in two-dimensions, but the great news is that the function doesn't care how many dimensions there are! It just subtracts the two arrays of coordinates (no matter how long the arrays are), squares the differences and adds up, and then takes the square root. To work in multiple dimensions, we don't have to change the code at all.

```
def distance(point1, point2):
    """Returns the distance between point1 and point2
    where each argument is an array
    consisting of the coordinates of the point"""
    return np.sqrt(np.sum((point1 - point2)**2))
```

Let's use this on a [new dataset](#). The table `wine` contains the chemical composition of 178 different Italian wines. The classes are the grape species, called cultivars. There are three classes but let's just see whether we can tell Class 1 apart from the other two.

```
wine = Table.read_table('wine.csv')

# For converting Class to binary

def is_one(x):
    if x == 1:
        return 1
    else:
        return 0

wine = wine.with_column('Class', wine.apply(is_one, 0))
```

wine

Class	Alcohol	Malic Acid	Ash	Alcalinity of Ash	Magnesium	Total Phenols	Flavane
1	14.23	1.71	2.43	15.6	127	2.8	3.06
1	13.2	1.78	2.14	11.2	100	2.65	2.76
1	13.16	2.36	2.67	18.6	101	2.8	3.24
1	14.37	1.95	2.5	16.8	113	3.85	3.49
1	13.24	2.59	2.87	21	118	2.8	2.69
1	14.2	1.76	2.45	15.2	112	3.27	3.39
1	14.39	1.87	2.45	14.6	96	2.5	2.52
1	14.06	2.15	2.61	17.6	121	2.6	2.51
1	14.83	1.64	2.17	14	97	2.8	2.98
1	13.86	1.35	2.27	16	98	2.98	3.15

... (168 rows omitted)

The first two wines are both in Class 1. To find the distance between them, we first need a table of just the attributes:

```
wine_attributes = wine.drop('Class')
```

```
distance(np.array(wine_attributes.row(0)),  
np.array(wine_attributes.row(1)))
```

```
31.265012394048398
```

The last wine in the table is of Class 0. Its distance from the first wine is:

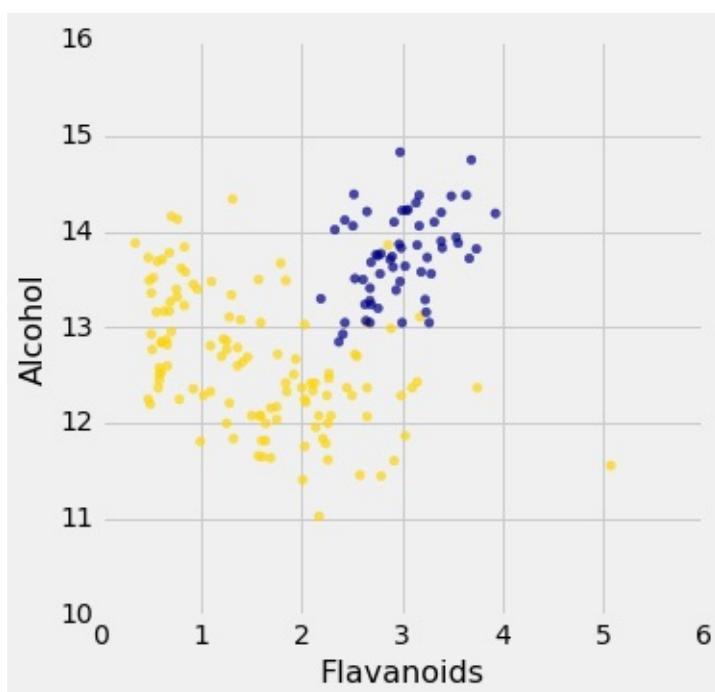
```
distance(np.array(wine_attributes.row(0)),  
np.array(wine_attributes.row(177)))
```

```
506.05936766351834
```

That's quite a bit bigger! Let's do some visualization to see if Class 1 really looks different from Class 0.

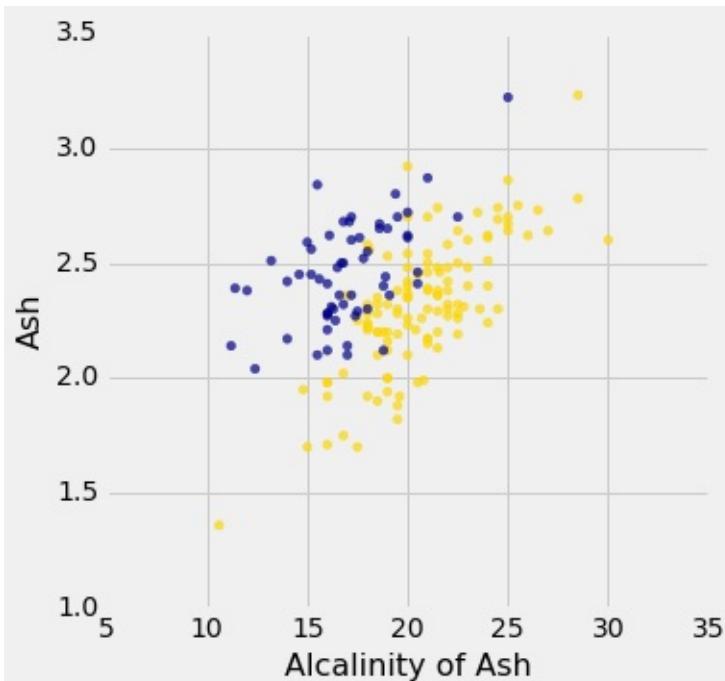
```
wine_with_colors = wine.join('Class', color_table)
```

```
wine_with_colors.scatter('Flavanoids', 'Alcohol',  
colors='Color')
```



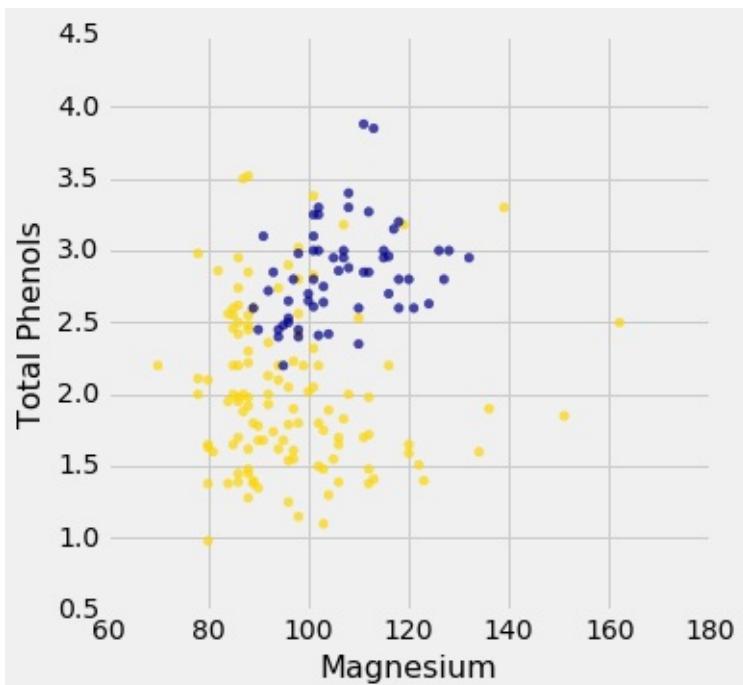
The blue points (Class 1) are almost entirely separate from the gold ones. That is one indication of why the distance between two Class 1 wines would be smaller than the distance between wines of two different classes. We can see a similar phenomenon with a different pair of attributes too:

```
wine_with_colors.scatter('Alcalinity of Ash', 'Ash',  
colors='Color')
```



But for some pairs the picture is more murky.

```
wine_with_colors.scatter('Magnesium', 'Total Phenols',  
colors='Color')
```



Let's see if we can implement a classifier based on all of the attributes. After that, we'll see how accurate it is.

A Plan for the Implementation

It's time to write some code to implement the classifier. The input is a `point` that we want to classify. The classifier works by finding the k nearest neighbors of `point` from the training set. So, our approach will go like this:

1. Find the closest k neighbors of `point`, i.e., the k wines from the training set that are most similar to `point`.
2. Look at the classes of those k neighbors, and take the majority vote to find the most-common class of wine. Use that as our predicted class for `point`.

So that will guide the structure of our 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)
```

Implementation Step 1

To implement the first step for the kidney disease data, we had to compute the distance from each patient in the training set to `point`, sort them by distance, and take the k closest patients in the training set.

That's what we did in the previous section with the point corresponding to Alice. Let's generalize that code. We'll redefine `distance` here, just for convenience.

```
def distance(point1, point2):
    """Returns the distance between point1 and point2
    where each argument is an array
    consisting of the coordinates of the point"""
    return np.sqrt(np.sum((point1 - point2)**2))

def all_distances(training, new_point):
    """Returns an array of distances
    between each point in the training set
    and the new point (which is a row of attributes)"""
    attributes = training.drop('Class')
    def distance_from_point(row):
        return distance(np.array(new_point), np.array(row))
    return attributes.apply(distance_from_point)

def table_with_distances(training, new_point):
    """Augments the training table
    with a column of distances from new_point"""
    return training.with_column('Distance',
        all_distances(training, new_point))

def closest(training, new_point, k):
    """Returns a table of the k rows of the augmented table
    corresponding to the k smallest distances"""
    with_dists = table_with_distances(training, new_point)
    sorted_by_distance = with_dists.sort('Distance')
    topk = sorted_by_distance.take(np.arange(k))
    return topk
```

Let's see how this works on our `wine` data. We'll just take the first wine and find its five nearest neighbors among all the wines. Remember that since this wine is part of the dataset, it is its own nearest neighbor. So we should expect to see it at the top of the list, followed by four others.

First let's extract its attributes:

```
special_wine = wine.drop('Class').row(0)
```

And now let's find its 5 nearest neighbors.

```
closest(wine, special_wine, 5)
```

Class	Alcohol	Malic Acid	Ash	Alcalinity of Ash	Magnesium	Total Phenols	Flavane
1	14.23	1.71	2.43	15.6	127	2.8	3.06
1	13.74	1.67	2.25	16.4	118	2.6	2.9
1	14.21	4.04	2.44	18.9	111	2.85	2.65
1	14.1	2.02	2.4	18.8	103	2.75	2.92
1	14.38	3.59	2.28	16	102	3.25	3.17

Bingo! The first row is the nearest neighbor, which is itself – there's a 0 in the `Distance` column as expected. All five nearest neighbors are of Class 1, which is consistent with our earlier observation that Class 1 wines appear to be clumped together in some dimensions.

Implementation Steps 2 and 3

Next we need to take a "majority vote" of the nearest neighbors and assign our point the same class as the majority.

```
def majority(topkclasses):
    ones = topkclasses.where('Class', are.equal_to(1)).num_rows
    zeros = topkclasses.where('Class', are.equal_to(0)).num_rows
    if ones > zeros:
        return 1
    else:
        return 0

def classify(training, new_point, k):
    closestk = closest(training, new_point, k)
    topkclasses = closestk.select('Class')
    return majority(topkclasses)
```

```
classify(wine, special_wine, 5)
```

```
1
```

If we change `special_wine` to be the last one in the dataset, is our classifier able to tell that it's in Class 0?

```
special_wine = wine.drop('Class').row(177)
classify(wine, special_wine, 5)
```

```
0
```

Yes! The classifier gets this one right too.

But we don't yet know how it does with all the other wines, and in any case we know that testing on wines that are already part of the training set might be over-optimistic. In the final section of this chapter, we will separate the wines into a training and test set and then measure the accuracy of our classifier on the test set.

[Interact](#)

The Accuracy of the Classifier

To see how well our classifier does, 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. We've been calling that the *test set*. 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.

Measuring the Accuracy of Our Wine Classifier

OK, so let's apply the hold-out method to evaluate the effectiveness of the k -nearest neighbor classifier for identifying wines. The data set has 178 wines, so we'll randomly permute the data set and put 89 of them in the training set and the remaining 89 in the test set.

```
shuffled_wine = wine.sample(with_replacement=False)
training_set = shuffled_wine.take(np.arange(89))
test_set   = shuffled_wine.take(np.arange(89, 178))
```

We'll train the classifier using the 89 wines 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 wine in the test set:

```
def count_zero(array):
    """Counts the number of 0's in an array"""
    return len(array) - np.count_nonzero(array)

def count_equal(array1, array2):
    """Takes two numerical arrays of equal length
    and counts the indices where the two are equal"""
    return count_zero(array1 - array2)

def evaluate_accuracy(training, test, k):
    test_attributes = test.drop('Class')
    def classify_testrow(row):
        return classify(training, row, k)
    c = test_attributes.apply(classify_testrow)
    return count_equal(c, test.column('Class')) / test.num_rows
```

Now for the grand reveal -- let's see how we did. We'll arbitrarily use $k = 5$.

```
evaluate_accuracy(training_set, test_set, 5)
```

```
0.9213483146067416
```

The accuracy rate isn't bad at all for a simple classifier.

Breast Cancer Diagnosis

Now I want to do an example based on diagnosing breast cancer. I was inspired by Brittany Wenger, who won the Google national science fair in 2012 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 the use of 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).

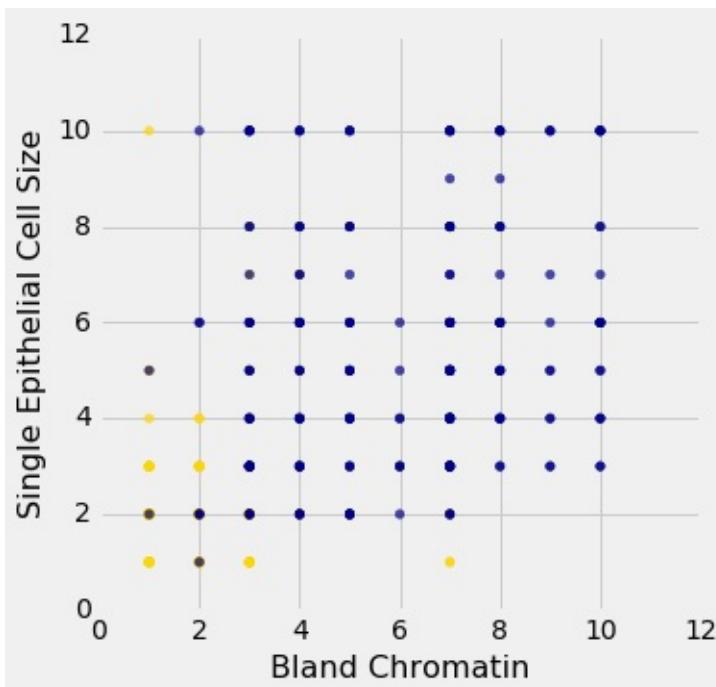
```
patients = Table.read_table('breast-cancer.csv').drop('ID')
patients
```

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)						

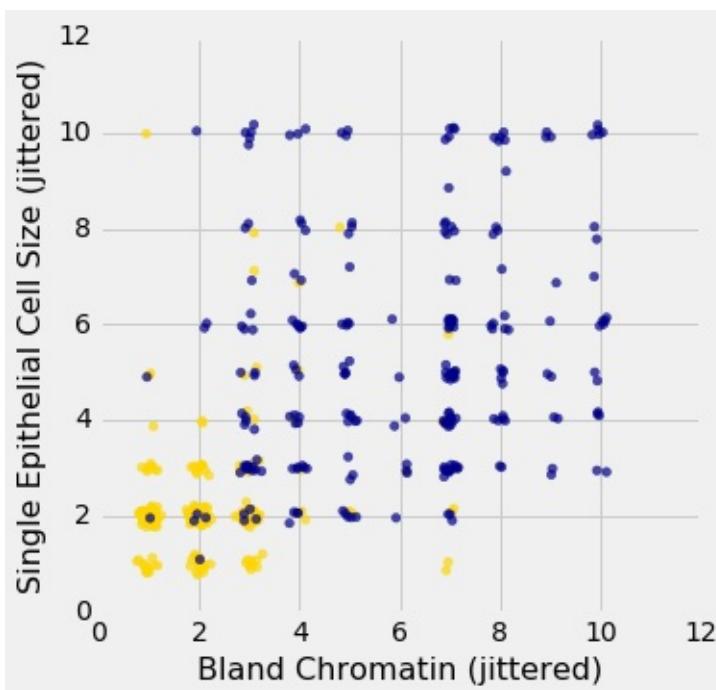
So we have 9 different attributes. I don't know how to make a 9-dimensional scatterplot of all of them, so I'm going to pick two and plot them:

```
color_table = Table().with_columns(
    'Class', make_array(1, 0),
    'Color', make_array('darkblue', 'gold')
)
patients_with_colors = patients.join('Class', color_table)
```

```
patients_with_colors.scatter('Bland Chromatin', 'Single
Epithelial Cell Size', colors='Color')
```



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:



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. We're ready to work with the data now, and we'll use the original (unjittered) data.

First we'll create a training set and a test set. 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.

```
shuffled_patients = patients.sample(683, with_replacement=False)
training_set = shuffled_patients.take(np.arange(342))
test_set = shuffled_patients.take(np.arange(342, 683))
```

Let's stick with 5 nearest neighbors, and see how well our classifier does.

```
evaluate_accuracy(training_set, test_set, 5)
```

```
0.967741935483871
```

Over 96% accuracy. Not bad! Once again, 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.

Interact

Now that we have explored ways to use multiple attributes to predict a categorical variable, let us return to predicting a quantitative variable. Predicting a numerical quantity is called regression, and a commonly used method to use multiple attributes for regression is called *multiple linear regression*.

Home Prices

The following dataset of house prices and attributes was collected over several years for the city of Ames, Iowa. A [description of the dataset appears online](#). We will focus only a subset of the columns. We will try to predict the sale price column from the other columns.

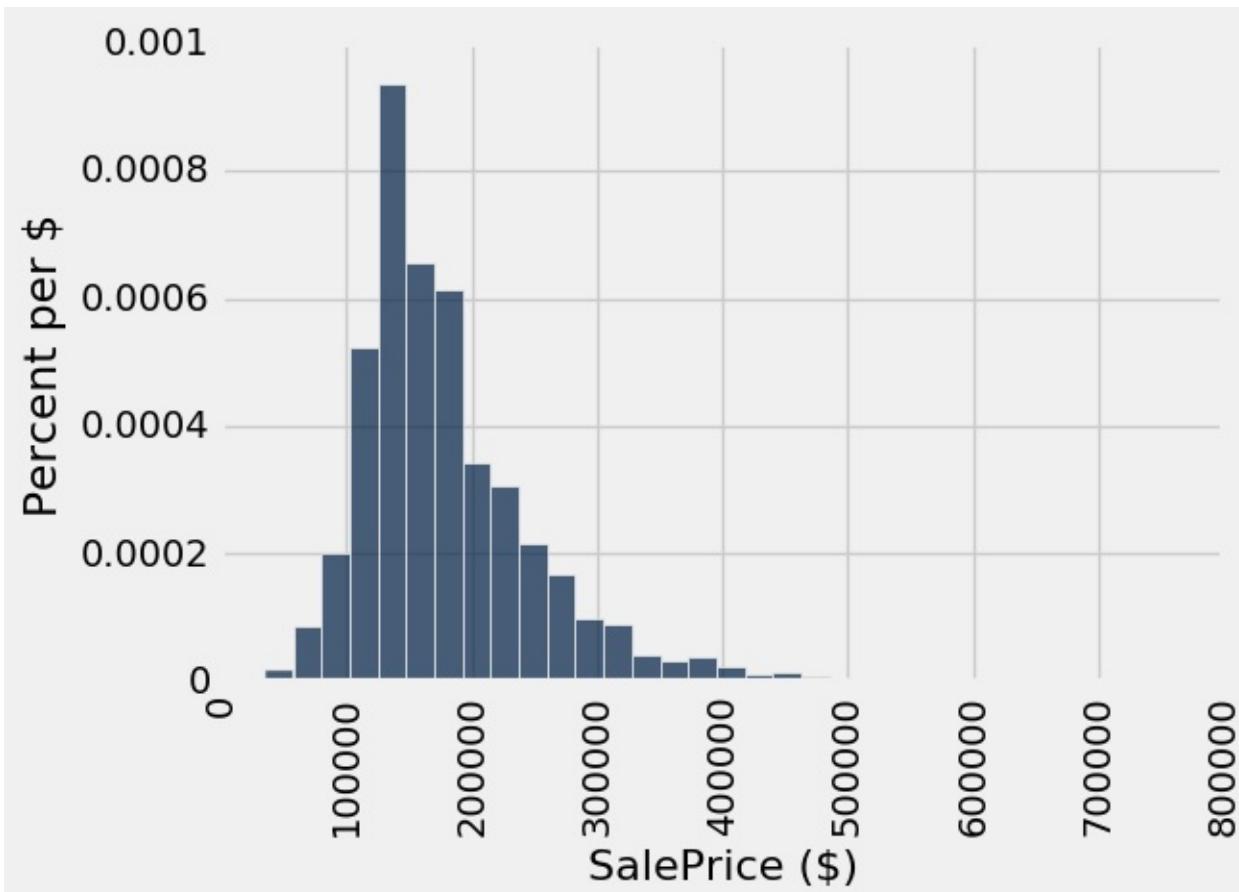
```
all_sales = Table.read_table('house.csv')
sales = all_sales.where('Bldg Type', '1Fam').where('Sale
Condition', 'Normal').select(
    'SalePrice', '1st Flr SF', '2nd Flr SF',
    'Total Bsmt SF', 'Garage Area',
    'Wood Deck SF', 'Open Porch SF', 'Lot Area',
    'Year Built', 'Yr Sold')
sales.sort('SalePrice')
```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Wood Deck SF	Open Porch SF	Lot Area	Year Built	
35000	498	0	498	216	0	0	8088	1922	:
39300	334	0	0	0	0	0	5000	1946	:
40000	649	668	649	250	0	54	8500	1920	:
45000	612	0	0	308	0	0	5925	1940	:
52000	729	0	270	0	0	0	4130	1935	:
52500	693	0	693	0	0	20	4118	1941	:
55000	723	363	723	400	0	24	11340	1920	:
55000	796	0	796	0	0	0	3636	1922	:
57625	810	0	0	280	119	24	21780	1910	:
58500	864	0	864	200	0	0	8212	1914	:

... (1992 rows omitted)

A histogram of sale prices shows a large amount of variability and a distribution that is clearly not normal. A long tail to the right contains a few houses that had very high prices. The short left tail does not contain any houses that sold for less than \$35,000.

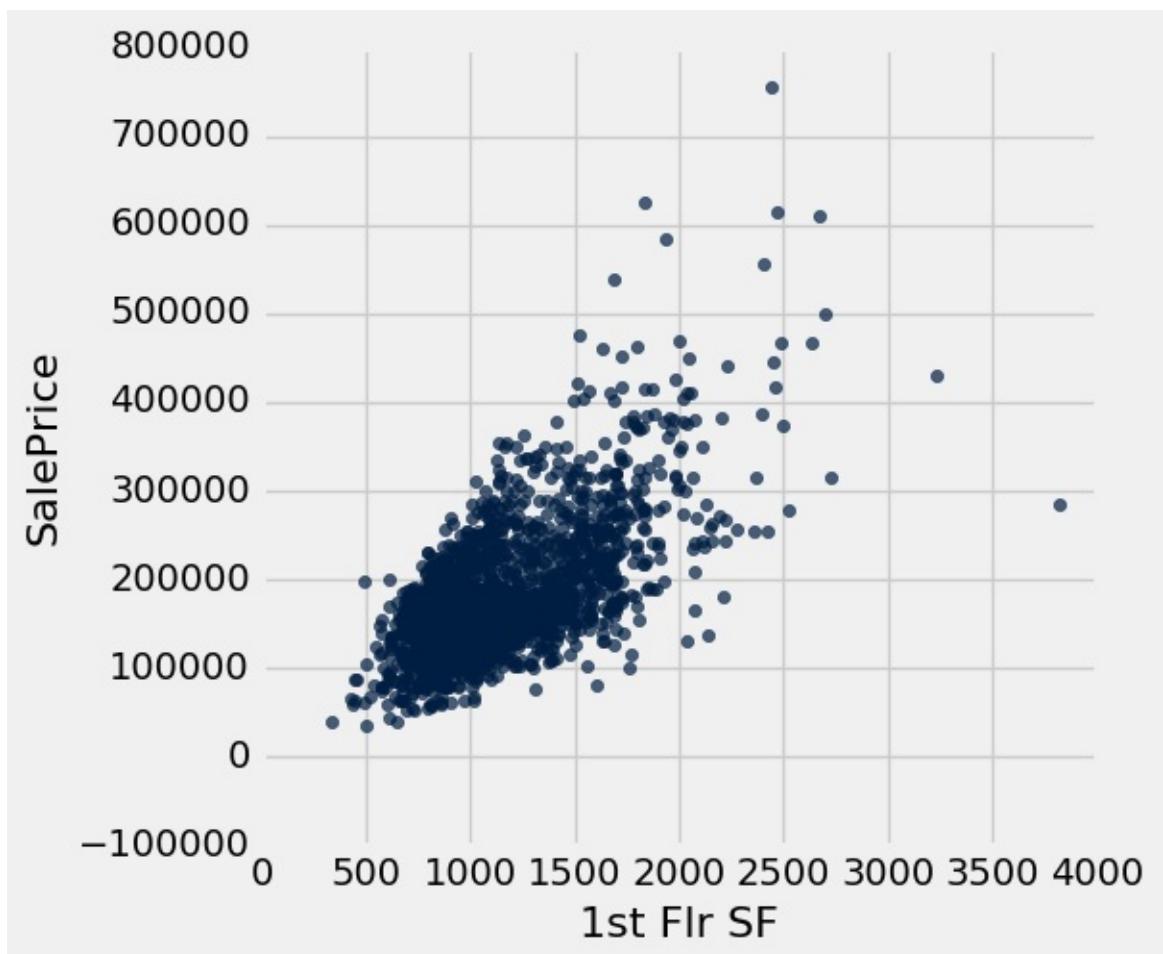
```
sales.hist('SalePrice', bins=32, unit='$')
```



Correlation

No single attribute is sufficient to predict the sale price. For example, the area of first floor, measured in square feet, correlates with sale price but only explains some of its variability.

```
sales.scatter('1st Flr SF', 'SalePrice')
```



```
correlation(sales, 'SalePrice', '1st Flr SF')
```

```
0.64246625410302249
```

In fact, none of the individual attributes have a correlation with sale price that is above 0.7 (except for the sale price itself).

```
for label in sales.labels:  
    print('Correlation of', label, 'and SalePrice:\t',  
correlation(sales, label, 'SalePrice'))
```

```

Correlation of SalePrice and SalePrice:      1.0
Correlation of 1st Flr SF and SalePrice:     0.642466254103
Correlation of 2nd Flr SF and SalePrice:      0.35752189428
Correlation of Total Bsmt SF and SalePrice:   0.652978626757
Correlation of Garage Area and SalePrice:    0.638594485252
Correlation of Wood Deck SF and SalePrice:    0.352698666195
Correlation of Open Porch SF and SalePrice:   0.336909417026
Correlation of Lot Area and SalePrice:       0.290823455116
Correlation of Year Built and SalePrice:     0.565164753714
Correlation of Yr Sold and SalePrice:        0.0259485790807

```

However, combining attributes can provide higher correlation. In particular, if we sum the first floor and second floor areas, the result has a higher correlation than any single attribute alone.

```

both_floors = sales.column(1) + sales.column(2)
correlation(sales.with_column('Both Floors', both_floors),
            'SalePrice', 'Both Floors')

```

```
0.7821920556134877
```

This high correlation indicates that we should try to use more than one attribute to predict the sale price. In a dataset with multiple observed attributes and a single numerical value to be predicted (the sale price in this case), multiple linear regression can be an effective technique.

Multiple Linear Regression

In multiple linear regression, a numerical output is predicted from numerical input attributes by multiplying each attribute value by a different slope, then summing the results. In this example, the slope for the `1st Flr SF` would represent the dollars per square foot of area on the first floor of the house that should be used in our prediction.

Before we begin prediction, we split our data randomly into a training and test set of equal size.

```
train, test = sales.split(1001)
print(train.num_rows, 'training and', test.num_rows, 'test
instances.')
```

1001 training and 1001 test instances.

The slopes in multiple regression is an array that has one slope value for each attribute in an example. Predicting the sale price involves multiplying each attribute by the slope and summing the result.

```
def predict(slopes, row):
    return sum(slopes * np.array(row))

example_row = test.drop('SalePrice').row(0)
print('Predicting sale price for:', example_row)
example_slopes = np.random.normal(10, 1, len(example_row))
print('Using slopes:', example_slopes)
print('Result:', predict(example_slopes, example_row))
```

```
Predicting sale price for: Row(1st Flr SF=1092, 2nd Flr SF=1020,
Total Bsmt SF=952.0, Garage Area=576.0, Wood Deck SF=280, Open
Porch SF=0, Lot Area=11075, Year Built=1969, Yr Sold=2008)
Using slopes: [  9.99777721   9.019661   11.13178317
  9.40645585  11.07998556
  11.03830075  10.26908341  10.42534332  11.00103437]
Result: 195583.275784
```

The result is an estimated sale price, which can be compared to the actual sale price to assess whether the slopes provide accurate predictions. Since the `example_slopes` above were chosen at random, we should not expect them to provide accurate predictions at all.

```
print('Actual sale price:', test.column('SalePrice').item(0))
print('Predicted sale price using random slopes:',
predict(example_slopes, example_row))
```

```
Actual sale price: 206900
Predicted sale price using random slopes: 195583.275784
```

Least Squares Regression¶

The next step in performing multiple regression is to define the least squares objective. We perform the prediction for each row in the training set, and then compute the root mean squared error (RMSE) of the predictions from the actual prices.

```
train_prices = train.column(0)
train_attributes = train.drop(0)

def rmse(slopes, attributes, prices):
    errors = []
    for i in np.arange(len(prices)):
        predicted = predict(slopes, attributes.row(i))
        actual = prices.item(i)
        errors.append((predicted - actual) ** 2)
    return np.mean(errors) ** 0.5

def rmse_train(slopes):
    return rmse(slopes, train_attributes, train_prices)

print('RMSE of all training examples using random slopes:',
      rmse_train(example_slopes))
```

```
RMSE of all training examples using random slopes: 69653.9880638
```

Finally, we use the `minimize` function to find the slopes with the lowest RMSE. Since the function we want to minimize, `rmse_train`, takes an array instead of a number, we must pass the `array=True` argument to `minimize`. When this argument is used, `minimize` also requires an initial guess of the slopes so that it knows the dimension of the input array. Finally, to speed up optimization, we indicate that `rmse_train` is a smooth function using the `smooth=True` attribute. Computation of the best slopes may take several minutes.

```
best_slopes = minimize(rmse_train, start=example_slopes,
smooth=True, array=True)
print('The best slopes for the training set:')
Table(train_attributes.labels).with_row(list(best_slopes)).show()

print('RMSE of all training examples using the best slopes:', 
rmse_train(best_slopes))
```

The best slopes for the training set:

1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Wood Deck SF	Open Porch SF	Lot Area	Year Built
73.7779	72.3057	51.8885	46.5581	39.3267	11.996	0.451265	538.24

RMSE of all training examples using the best slopes:
31146.4442711

Interpreting Multiple Regression

Let's interpret these results. The best slopes give us a method for estimating the price of a house from its attributes. A square foot of area on the first floor is worth about \$75 (the first slope), while one on the second floor is worth about \$70 (the second slope). The final negative value describes the market: prices in later years were lower on average.

The RMSE of around \$30,000 means that our best linear prediction of the sale price based on all of the attributes is off by around \$30,000 on the training set, on average. We find a similar error when predicting prices on the test set, which indicates that our prediction method will generalize to other samples from the same population.

```
test_prices = test.column(0)
test_attributes = test.drop(0)

def rmse_test(slopes):
    return rmse(slopes, test_attributes, test_prices)

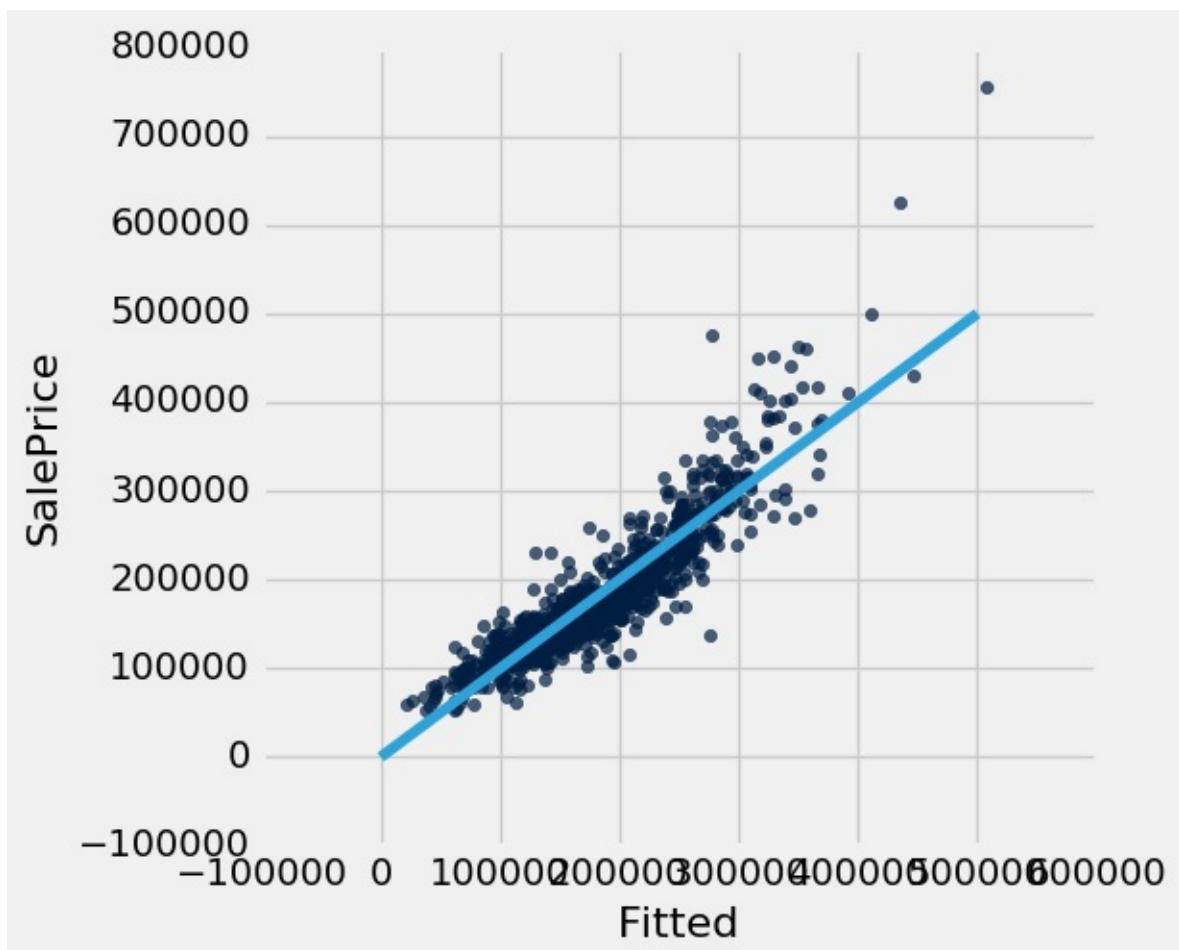
rmse_linear = rmse_test(best_slopes)
print('Test set RMSE for multiple linear regression:', rmse_linear)
```

```
Test set RMSE for multiple linear regression: 31105.4799398
```

If the predictions were perfect, then a scatter plot of the predicted and actual values would be a straight line with slope 1. We see that most dots fall near that line, but there is some error in the predictions.

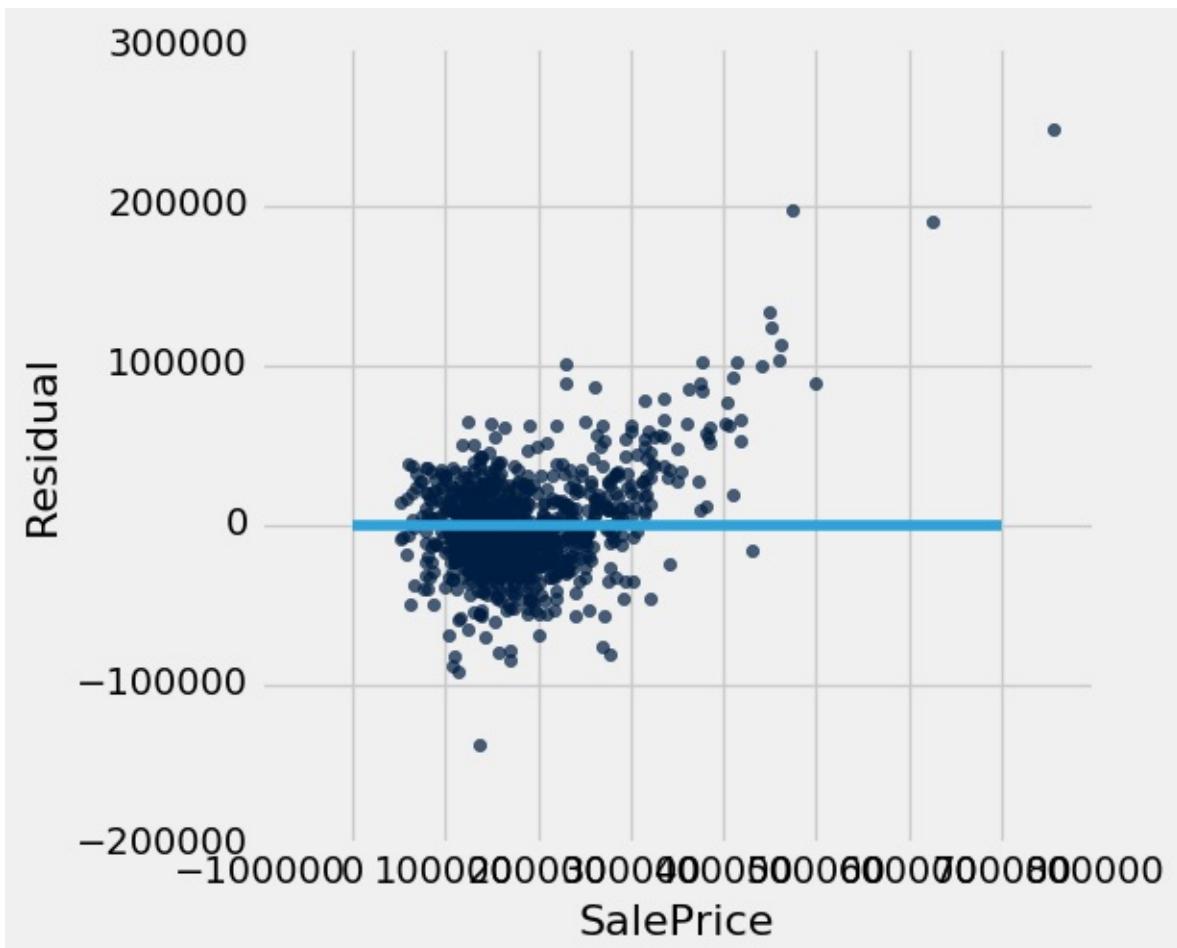
```
def fit(row):
    return sum(best_slopes * np.array(row))

test.with_column('Fitted',
test.drop(0).apply(fit)).scatter('Fitted', 0)
plots.plot([0, 5e5], [0, 5e5]);
```



A residual plot for multiple regression typically compares the errors (residuals) to the actual values of the predicted variable. We see in the residual plot below that we have systematically underestimated the value of expensive houses, shown by the many positive residual values on the right side of the graph.

```
test.with_column('Residual', test_prices-
test.drop(0).apply(fit)).scatter(0, 'Residual')
plots.plot([0, 7e5], [0, 0]);
```



As with simple linear regression, interpreting the result of a predictor is at least as important as making predictions. There are many lessons about interpreting multiple regression that are not included in this textbook. A natural next step after completing this text would be to study linear modeling and regression in further depth.

Nearest Neighbors for Regression

Another approach to predicting the sale price of a house is to use the price of similar houses. This *nearest neighbor* approach is very similar to our classifier. To speed up computation, we will only use the attributes that had the highest correlation with the sale price in our original analysis.

```
train_nn = train.select(0, 1, 2, 3, 4, 8)
test_nn = test.select(0, 1, 2, 3, 4, 8)
train_nn.show(3)
```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Year Built
240000	1710	0	1710	550	2004
229000	1302	735	672	472	1996
136500	864	0	864	336	1978

... (998 rows omitted)

The computation of closest neighbors is identical to a nearest-neighbor classifier. In this case, we will exclude the `'SalePrice'` rather than the `'Class'` column from the distance computation. The five nearest neighbors of the first test row are shown below.

```

def distance(pt1, pt2):
    """The distance between two points, represented as
arrays."""
    return np.sqrt(sum((pt1 - pt2) ** 2))

def row_distance(row1, row2):
    """The distance between two rows of a table."""
    return distance(np.array(row1), np.array(row2))

def distances(training, example, output):
    """Compute the distance from example for each row in
training."""
    dists = []
    attributes = training.drop(output)
    for row in attributes.rows:
        dists.append(row_distance(row, example))
    return training.with_column('Distance', dists)

def closest(training, example, k, output):
    """Return a table of the k closest neighbors to example."""
    return distances(training, example,
output).sort('Distance').take(np.arange(k))

example_nn_row = test_nn.drop(0).row(0)
closest(train_nn, example_nn_row, 5, 'SalePrice')

```

SalePrice	1st Flr SF	2nd Flr SF	Total Bsmt SF	Garage Area	Year Built	Distance
150000	1299	0	967	494	1954	51.9711
144000	1344	0	1024	484	1958	60.8358
183500	1299	0	1001	486	1979	68.6003
140000	1283	0	931	506	1962	76.5049
173000	1287	0	957	541	1977	77.2464

One simple method for predicting the price is to average the prices of the nearest neighbors.

```
def predict_nn(example):
    """Return the majority class among the k nearest
neighbors."""
    return np.average(closest(train_nn, example, 5,
'SalePrice').column('SalePrice'))

predict_nn(example_nn_row)
```

158100.0

Finally, we can inspect whether our prediction is close to the true sale price for our one test example. Looks reasonable!

```
print('Actual sale price:', test_nn.column('SalePrice').item(0))
print('Predicted sale price using nearest neighbors:',
predict_nn(example_nn_row))
```

Actual sale price: 146000

Predicted sale price using nearest neighbors: 158100.0

Evaluation

To evaluate the performance of this approach for the whole test set, we apply `predict_nn` to each test example, then compute the root mean squared error of the predictions. Computation of the predictions may take several minutes.

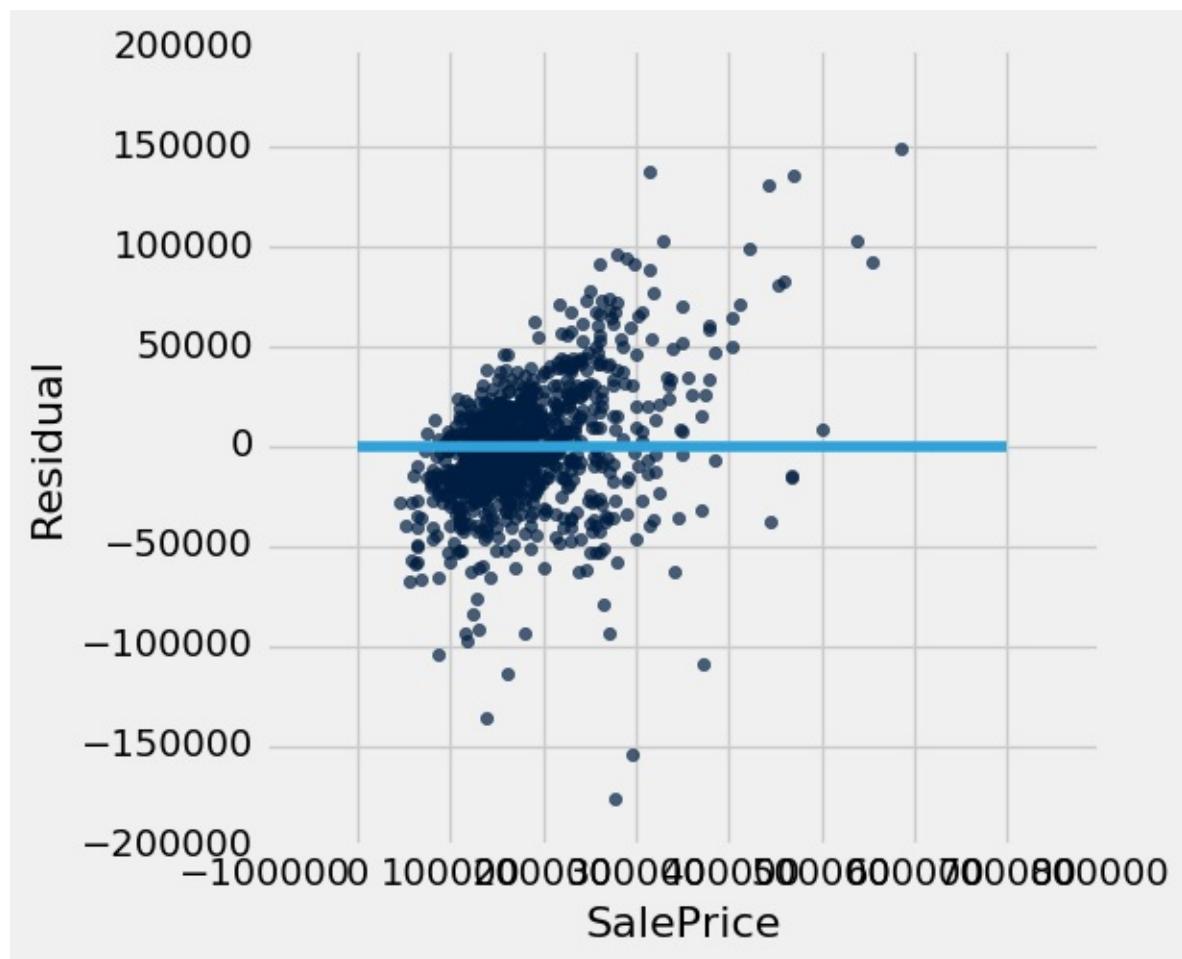
```
nn_test_predictions =  
test_nn.drop('SalePrice').apply(predict_nn)  
rmse_nn = np.mean((test_prices - nn_test_predictions) ** 2) **  
0.5  
  
print('Test set RMSE for multiple linear regression: ',  
rmse_linear)  
print('Test set RMSE for nearest neighbor regression:', rmse_nn)
```

```
Test set RMSE for multiple linear regression: 30232.0744208  
Test set RMSE for nearest neighbor regression: 31210.6572877
```

For these data, the errors of the two techniques are quite similar! For different data sets, one technique might outperform another. By computing the RMSE of both techniques on the same data, we can compare methods fairly. One note of caution: the difference in performance might not be due to the technique at all; it might be due to the random variation due to sampling the training and test sets in the first place.

Finally, we can draw a residual plot for these predictions. We still underestimate the prices of the most expensive houses, but the bias does not appear to be as systematic. However, fewer residuals are very close to zero, indicating that fewer prices were predicted with very high accuracy.

```
test.with_column('Residual', test_prices -  
nn_test_predictions).scatter(0, 'Residual')  
plots.plot([0, 7e5], [0, 0]);
```



[Interact](#)

Updating Predictions

We know how to use training data to classify a point into one of two categories. Our classification is just a prediction of the class, based on the most common class among the training points that are nearest our new point.

Suppose that we eventually find out the true class of our new point. Then we will know whether we got the classification right. Also, we will have a new point that we can add to our training set, because we know its class. This *updates* our training set. So, naturally, we will want to *update our classifier* based on the new training set.

This chapter looks at some simple scenarios where new data leads us to update our predictions. While the examples in the chapter are simple in terms of calculation, the method of updating can be generalized to work in complex settings and is one of the most powerful tools used for machine learning.

[Interact](#)

A "More Likely Than Not" Binary Classifier

Let's try to use data to classify a point into one of two categories, choosing the category that we think is more likely than not. To do this, we not only need the data but also a clear description of how chances are involved.

We will start out in a simple artificial setting just to develop the main technique, and then move to a more intriguing example.

Suppose there is a university class with the following composition:

- 60% of the students are Second Years and the remaining 40% are Third Years
- 50% of the Second Years have declared their major
- 80% of the Third Years have declared their major

Now suppose I pick a student at random from the class. Can you classify the student as Second Year or Third Year, using our "more likely than not" criterion?

You can, because the student is picked at random and so you know that the chance that the student is a Second Year is 60%. That's greater than the 40% chance of being a Third Year, so you would classify the student as Second Year.

The information about the majors is irrelevant, as we already know the proportions of Second and Third Years in the class.

We have a pretty simple classifier! But now suppose I give you some additional information about the student who was picked:

The student has declared a major.

Would this knowledge change your classification?

Updating the Prediction Based on New Information

Now that we know the student has declared a major, it becomes important to look at the relation between year and major declaration. It's still true that more students are Second Years than Third Years. But it's also true that among the Third Years, a much higher percent have declared their major than among the Second Years. Our classifier has to take both of these observations into account.

To visualize this, we will use a table `students` that consists of one row for each of 100 students whose years and majors have the same proportions as given in the data.

```
students.show(3)
```

Year	Major
Second	Undeclared
Second	Undeclared
Second	Undeclared

... (97 rows omitted)

To check that the proportions are correct, let's use `pivot` to cross-classify each student according to the two variables.

```
students.pivot('Major', 'Year')
```

Year	Declared	Undeclared
Second	30	30
Third	32	8

The total count is 100 students, of whom 60 are Second Years and 40 are Third Years. Among the Second Years, 50% are in each of the Major categories. Among the 40 Third Years, 20% are Undeclared and 80% Declared. So this population of 100 students has the same proportions as the class in our problem, and we can assume that our student has been picked at random from among all 100 students.

We have to pick which row the student is most likely to be in. When we knew nothing more about the student, he or she could be in any of the four cells, and therefore were more likely to be in the top row (Second Year) because that contains more students.

But now we know that the student has declared a major, so the space of possible outcomes has decreased: now the student can only be in one of the two Declared cells.

There are 62 students in those cells, and 32 out of the 62 are Third Years. That's more than half, even though not by much.

So, in the light of the new information about the student's major, we have to update our prediction and now classify the student as a Third Year.

What is the chance that our classification is correct? We will be right for all the 32 Third Years who are Declared, and wrong for the 30 Second Years who are Declared. The chance that we are correct is therefore about 0.516.

In other words, the chance that we are correct is **the proportion of Third Years among the students who have Declared**.

```
32/(30+32)
```

```
0.5161290322580645
```

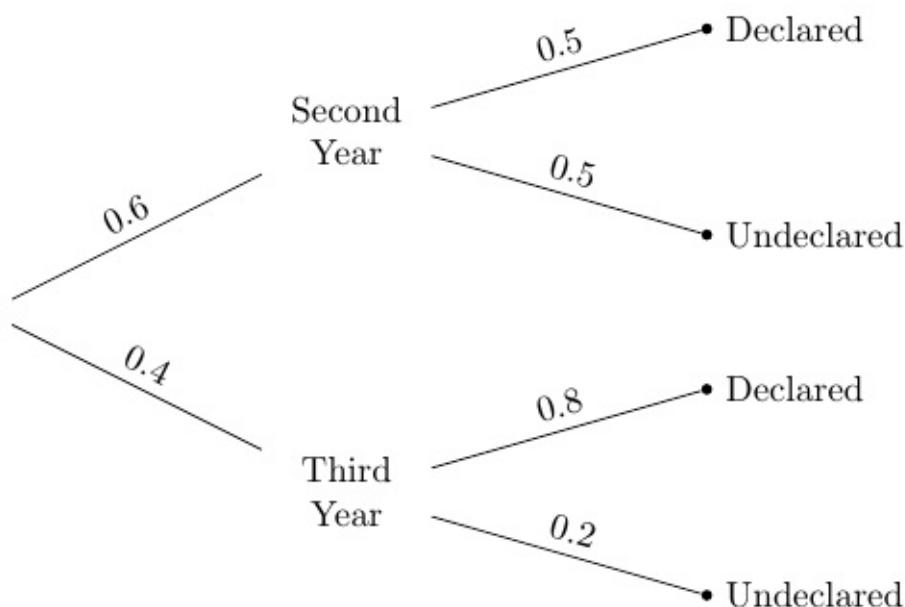
Tree Diagram¶

The proportion that we have just calculated was based on a class of 100 students. But there's no reason the class couldn't have had 200 students, for example, as long as all the proportions in the cells were correct. Then our calculation would just have been $64/(60 + 64)$ which is 0.516 as before.

So the calculation depends only on the proportions in the different categories, not on the counts. The proportions can be visualized in a *tree diagram*, shown directly below the pivot table for ease of comparison.

```
students.pivot('Major', 'Year')
```

Year	Declared	Undeclared
Second	30	30
Third	32	8



Like the pivot table, this diagram *partitions* the students into four distinct groups known as "branches". Notice that the "Third Year, Declared" branch contains the proportion $0.4 \times 0.8 = 0.32$ of the students, corresponding to the 32 students in the "Third Year, Declared" cell of the pivot table. The "Second Year, Declared" branch contains $0.6 \times 0.5 = 0.3$ of the students, corresponding to the 30 in the "Second Year, Declared" cell of the pivot table.

We know that the student who was picked belongs to a "Declared" branch; that is, the student is either in the top branch or the third from top. Those two branches now form our reduced space of possibilities, and all chances have to be calculated relative to the total chance of this reduced space.

So, given that the student is Declared, the chance of them being a Third Year can be calculated directly from the tree. The answer is the proportion in the "Third Year, Declared" branch relative to the total proportion in the two "Declared" branches.

That is, the answer is **the proportion of Third Years among students who are Declared**, as before.

$$(0.4 * 0.8) / (0.6 * 0.5 + 0.4 * 0.8)$$

$$0.5161290322580645$$

Bayes' Rule

The method that we have just used is due to the Reverend [Thomas Bayes](#) (1701-1761). His method solved what was called an "inverse probability" problem: given new data, how can you update chances you had found earlier? Though Bayes lived three centuries ago, his method is [widely used now](#) in machine learning.

We will state the rule in the context of our population of students. First, some terminology:

Prior probabilities. Before we knew the chosen student's major declaration status, the chance that the student was a Second Year was 60% and the chance that the student was a Third Year was 40%. These are the *prior* probabilities of the two categories.

Likelihoods. These are the chances of the Major status, given the category of student; thus they can be read off the tree diagram. For example, the likelihood of Declared status given that the student is a Second Year is 0.5.

Posterior probabilities. These are the chances of the two Year categories, *after* we have taken into account information about the Major declaration status. We computed one of these:

The *posterior probability* that the student is a Third Year, given that the student has Declared, is denoted $P(\text{Third Year} \mid \text{Declared})$ and is calculated as follows.

$$\begin{aligned} P(\text{Third Year} \mid \text{Declared}) &= \frac{0.4 \times 0.8}{0.6 \times 0.5 + 0.4 \times 0.8} \\ &= \frac{(\text{prior probability of Third Year}) \times (\text{likelihood})}{\text{total probability of De}} \end{aligned}$$

The other posterior probability is

$$\begin{aligned} P(\text{Second Year} \mid \text{Declared}) &= \frac{0.6 \times 0.5}{0.6 \times 0.5 + 0.4 \times 0.8} \\ &= \frac{(\text{prior probability of Second Year}) \times (\text{likelihood})}{\text{total probability of}} \end{aligned}$$

$$(0.6 * 0.5) / (0.6 * 0.5 + 0.4 * 0.8)$$

$$0.4838709677419354$$

That's about 0.484, which is less than half, consistent with our classification of Third Year.

Notice that both the posterior probabilities have the same denominator: the chance of the new information, which is that the student has Declared.

Because of this, Bayes' method is sometimes summarized as a statement about proportionality:

$$\text{posterior} \propto \text{prior} \times \text{likelihood}$$

Formulas are great for efficiently describing calculations. But in settings like our example about students, it is simpler not to think in terms of formulas. Just use the tree diagram.

[Interact](#)

Making Decisions

A primary use of Bayes' Rule is to make decisions based on incomplete information, incorporating new information as it comes in. This section points out the importance of keeping your assumptions in mind as you make decisions.

Many medical tests for diseases return Positive or Negative results. A Positive result means that according to the test, the patient has the disease. A Negative result means the test concludes that the patient doesn't have the disease.

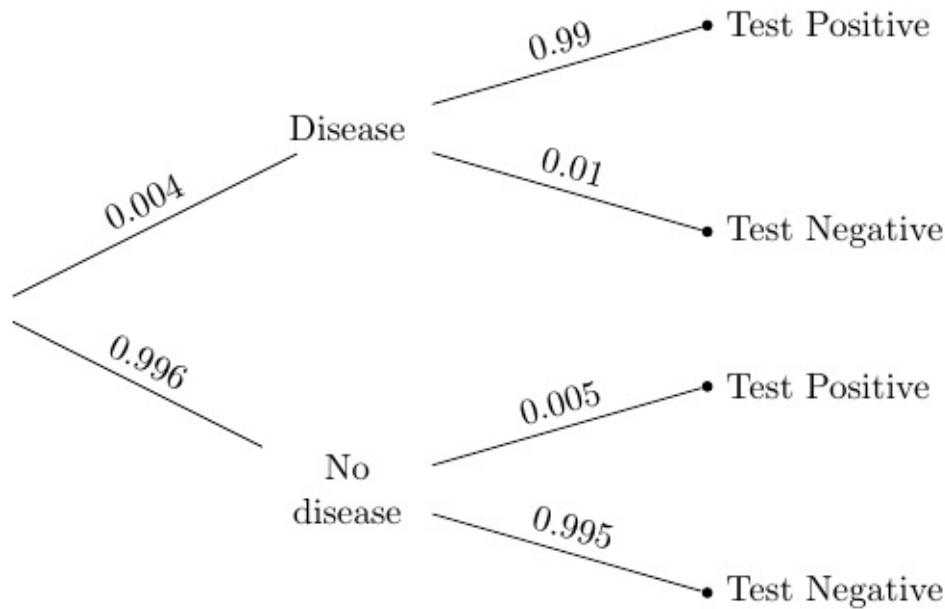
Medical tests are carefully designed to be very accurate. But few tests are accurate 100% of the time. Almost all tests make errors of two kinds:

- A **false positive** is an error in which the test concludes Positive but the patient doesn't have the disease.
- A **false negative** is an error in which the test concludes Negative but the patient does have the disease.

These errors can affect people's decisions. False positives can cause anxiety and unnecessary treatment (which in some cases is expensive or dangerous). False negatives can have even more serious consequences if the patient doesn't receive treatment because of their Negative test result.

A Test for a Rare Disease

Suppose there is a large population and a disease that strikes a tiny proportion of the population. The tree diagram below summarizes information about such a disease and about a medical test for it.



Overall, only 4 in 1000 of the population has the disease. The test is quite accurate: it has a very small false positive rate of 5 in 1000, and a somewhat larger (though still small) false negative rate of 1 in 100.

Individuals might or might not know whether they have the disease; typically, people get tested to find out whether they have it.

So suppose a person is picked at random from the population and tested. If the test result is Positive, how would you classify them: Disease, or No disease?

We can answer this by applying Bayes' Rule and using our "more likely than not" classifier. Given that the person has tested Positive, the chance that he or she has the disease is the proportion in the top branch, relative to the total proportion in the Test Positive branches.

$$(0.004 * 0.99) / (0.004 * 0.99 + 0.996 * 0.005)$$

$$0.44295302013422816$$

Given that the person has tested Positive, the chance that he or she has the disease is about 44%. So we will classify them as: No disease.

This is a strange conclusion. We have a pretty accurate test, and a person who has tested Positive, and our classification is ... that they **don't** have the disease? That doesn't seem to make any sense.

When faced with a disturbing answer, the first thing to do is to check the calculations. The arithmetic above is correct. Let's see if we can get the same answer in a different way.

The function `population` returns a table of outcomes for 100,000 patients, with columns that show the `True Condition` and `Test Result`. The test is the same as the one described in the tree. But the proportion who have the disease is an argument to the function.

We will call `population` with 0.004 as the argument, and then pivot to cross-classify each of the 100,000 people.

```
population(0.004).pivot('Test Result', 'True Condition')
```

True Condition	Negative	Positive
Disease	4	396
No Disease	99102	498

The cells of the table have the right counts. For example, according to the description of the population, 4 in 1000 people have the disease. There are 100,000 people in the table, so 400 should have the disease. That's what the table shows: $4 + 396 = 400$. Of these 400, 99% get a Positive test result: $0.99 \times 400 = 396$.

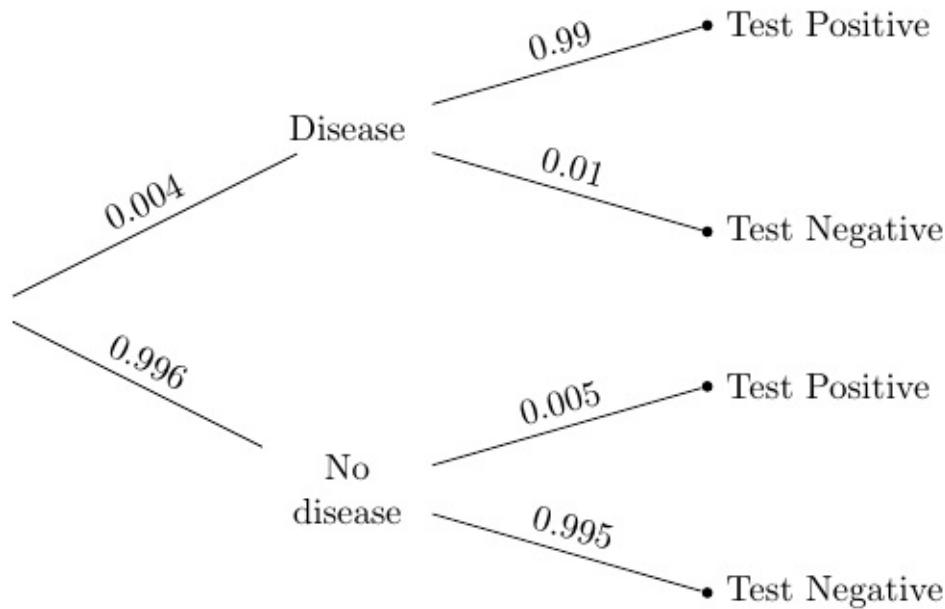
Among the Positives, the proportion that have the disease is:

```
396 / (396 + 498)
```

```
0.4429530201342282
```

That's the answer we got by using Bayes' Rule. The counts in the Positives column show why it is less than 1/2. Among the Positives, more people **don't** have the disease than do have the disease.

The reason is that a huge fraction of the population doesn't have the disease in the first place. The tiny fraction of those that falsely test Positive are still greater in number than the people who correctly test Positive. This is easier to visualize in the tree diagram:



- The proportion of true Positives is a large fraction (0.99) of a tiny fraction (0.004) of the population.
- The proportion of false Positives is a tiny fraction (0.005) of a large fraction (0.996) of the population.

These two proportions are comparable; the second is a little larger.

So, given that the randomly chosen person tested positive, we were right to classify them as more likely than not to **not** have the disease.

A Subjective Prior

Being right isn't always satisfying. Classifying a Positive patient as not having the disease still seems somehow wrong, for such an accurate test. Since the calculations are right, let's take a look at the basis of our probability calculation: the assumption of randomness.

Our assumption was that a randomly chosen person was tested and got a Positive result. But this doesn't happen in reality. People go in to get tested because they think they might have the disease, or because their doctor thinks they might have the disease. **People getting tested are not randomly chosen members of the population.**

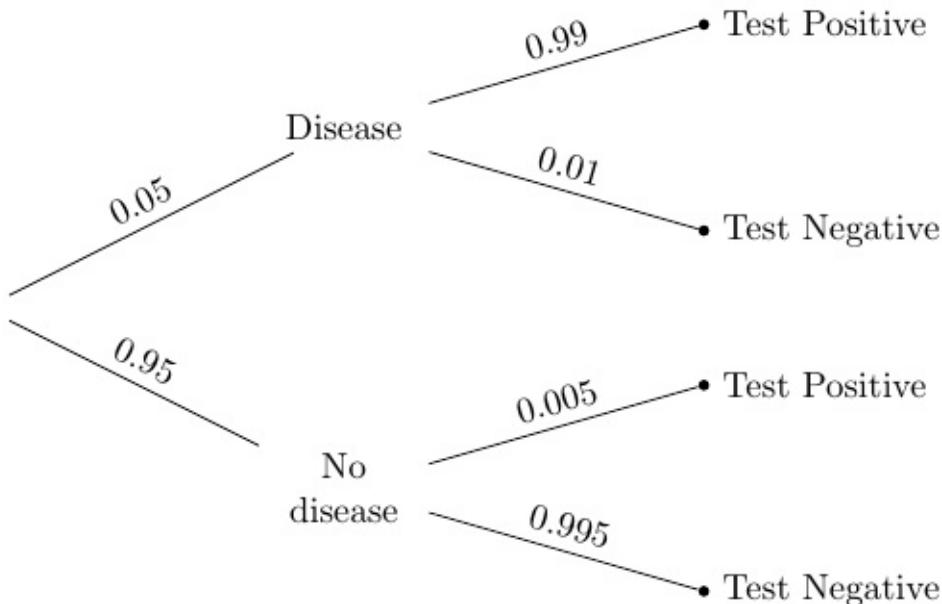
That is why our intuition about people getting tested was not fitting well with the answer that we got. We were imagining a realistic situation of a patient going in to get tested because there was some reason for them to do so, whereas the calculation was based on a randomly chosen person being tested.

So let's redo our calculation under the more realistic assumption that the patient is getting tested because the doctor thinks there's a chance the patient has the disease.

Here it's important to note that "the doctor thinks there's a chance" means that the chance is the doctor's opinion, not the proportion in the population. It is called a *subjective probability*. In our context of whether or not the patient has the disease, it is also a *subjective prior* 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. All calculations involving them thus have a subjective element too.

Suppose the doctor's subjective opinion is that there is a 5% chance that the patient has the disease. Then just the prior probabilities in the tree diagram will change:



Given that the patient tests Positive, the chance that he or she has the disease is given by Bayes' Rule.

$$(0.05 * 0.99) / (0.05 * 0.99 + 0.95 * 0.005)$$

$$0.9124423963133641$$

The effect of changing the prior is stunning. Even though the doctor has a pretty low prior probability (5%) that the patient has the disease, once the patient tests Positive the posterior probability of having the disease shoots up to more than 91%.

If the patient tests Positive, it would be reasonable for the doctor to proceed as though the patient has the disease.

Confirming the Answer

Though the doctor's opinion is subjective, we can generate an artificial population in which 5% of the people have the disease and are tested using the same test. Then we can count people in different categories to see if the counts are consistent with the answer we got by using Bayes' Rule.

We can use `population(0.05)` and `pivot` to construct the corresponding population and look at the counts in the four cells.

```
population(0.05).pivot('Test Result', 'True Condition')
```

True Condition	Negative	Positive
Disease	50	4950
No Disease	94525	475

In this artificially created population of 100,000 people, 5000 people (5%) have the disease, and 99% of them test Positive, leading to 4950 true Positives. Compare this with 475 false Positives: among the Positives, the proportion that have the disease is the same as what we got by Bayes' Rule.

```
4950 / (4950 + 475)
```

```
0.9124423963133641
```

Because we can generate a population that has the right proportions, we can also use simulation to confirm that our answer is reasonable. The table `pop_05` contains a population of 100,000 people generated with the doctor's prior disease probability of 5% and the error rates of the test. We take a simple random sample of size 10,000 from the population, and extract the table `positive` consisting only of those in the sample that had Positive test results.

```
pop_05 = population(0.05)

sample = pop_05.sample(10000, with_replacement=False)

positive = sample.where('Test Result', are.equal_to('Positive'))
```

Among these Positive results, what proportion were true Positives? That's the proportion of Positives that had the disease:

```
positive.where('True Condition',
are.equal_to('Disease')).num_rows/positive.num_rows
```

```
0.9131205673758865
```

Run the two cells a few times and you will see that the proportion of true Positives among the Positives hovers around the value of 0.912 that we calculated by Bayes' Rule.

You can also use the `population` function with a different argument to change the prior disease probability and see how the posterior probabilities are affected.