O'REILLY®

3rd Edition

Think Stats

Exploratory Data Analysis

Allen B. Downey

"This book is a trove of deep statistical wisdom and highly practical skills. If you want to learn—and use—statistics quickly, this book is for you."

Zachary del Rosario
Assistant professor, Olin College of Engineering

Think Stats

If you know how to program, you have the skills to turn data into knowledge. This thoroughly revised edition presents statistical concepts computationally, rather than mathematically, using programs written in Python. Through practical examples and exercises based on real-world datasets, you'll learn the entire process of exploratory data analysis—from wrangling data and generating statistics to identifying patterns and testing hypotheses.

Whether you're a data scientist, software engineer, or data enthusiast, you'll get up to speed on commonly used tools including NumPy, SciPy, and Pandas. You'll explore distributions, relationships between variables, visualization, and many other concepts. And all chapters are available as Jupyter notebooks, so you can read the text, run the code, and work on exercises all in one place.

- **Analyze data distributions and visualize patterns using Python libraries**
- **Improve predictions and insights with regression models**
- **Dive into specialized topics like time series analysis and survival analysis**
- **Communicate findings with effective data visualization**
- **Troubleshoot common data analysis challenges**
- **Boost reproducibility and collaboration in data analysis projects with interactive notebooks**

Allen B. Downey is a professor emeritus at Olin College of Engineering and principal data scientist at PyMC Labs. He is the author of several books about programming and data science, including *Think Python*, *Think Bayes*, *Think Complexity*, and *Probably Overthinking It*.

DATA

US $79.99 CAN $99.99
ISBN: 978-1-098-19025-5

O'REILLY®

THIRD EDITION

Think Stats

Exploratory Data Analysis

Allen B. Downey

O'REILLY®

Think Stats
by Allen B. Downey

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Sara Hunter
Production Editor: Gregory Hyman
Copyeditor: Sonia Saruba
Proofreader: Piper Content Partners

Indexer: Sue Klefstad
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2011: First Edition
October 2014: Second Edition
April 2025: Third Edition

Revision History for the Third Edition
2025-04-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098190255* for release details.

978-1-098-19025-5

[LSI]

Table of Contents

Preface

From the earliest history of statistics, there have been two ideas about what statistics is. In one view, it is a branch of mathematics with the goal of establishing a theoretical foundation for probability and statistical inference. In another view, it is a set of tools and practices for working with data, answering questions, and making better decisions. Many introductory classes in statistics are based on the first view. This book is based on the second.

Think Stats is an introduction to practical methods for exploring and visualizing data, discovering relationships and trends, and communicating results. The organization of the book follows the process I use when I start working with a dataset:

Importing and cleaning
: Whatever format the data is in, it usually takes some time and effort to read the data, clean and transform it, and check that everything made it through the translation process intact.

Single variable explorations
: I usually start by examining one variable at a time, finding out what the variables mean, looking at distributions of the values, and choosing appropriate summary statistics.

Pair-wise explorations
: To identify possible relationships between variables, I look at tables and scatter plots, and compute correlations and linear fits.

Multivariate analysis
: If there are apparent relationships between variables, I use multiple regression to add control variables and investigate more complex relationships.

Estimation and hypothesis testing
: When reporting statistical results, it is important to answer three questions: How big is the effect? How much variability should we expect if we run the same measurement again? Is it plausible that the apparent effect is due to chance?

Visualization
: During exploration, visualization is an important tool for finding possible relationships and effects. Then if an apparent effect holds up to scrutiny, visualization is an effective way to communicate results.

This book takes a computational approach, which has several advantages over more mathematical treatments:

- I present most ideas using Python code, rather than mathematical notation. In general, Python code is more readable—also, because it is executable, the reader can run it and modify it to develop insight.
- Each chapter includes exercises readers can do to check and solidify their learning. When you write programs, you express your understanding in code—while you are debugging the program, you are also checking your understanding.
- Some exercises involve experiments to test statistical behavior. For example, you can explore the Central Limit Theorem (CLT) by generating random samples and computing their sums. The resulting visualizations show why the CLT works and when it doesn't.
- Some ideas that are hard to grasp mathematically are easy to understand by simulation. For example, we approximate p-values by running random simulations, which reinforces the meaning of hypothesis testing.
- Because the book is based on a general-purpose programming language (Python), readers can import data from almost any source. They are not limited to datasets that have been cleaned and formatted for a particular statistical tool.

To demonstrate my approach to statistical analysis, the examples and exercises use data from several sources, including:

- The National Survey of Family Growth (NSFG), conducted by the US Centers for Disease Control and Prevention (CDC) to gather "information on family life, marriage and divorce, pregnancy, infertility, use of contraception, and men's and women's health."
- The Behavioral Risk Factor Surveillance System (BRFSS), conducted by the National Center for Chronic Disease Prevention and Health Promotion to "track health conditions and risk behaviors in the United States."

- The Palmer Penguins, which includes measurements from a sample of penguins near Palmer Station in Antarctica.
- Data from the US Energy Information Administration (EIA) on electricity generation from renewable sources in the United States.

I am grateful to the people and agencies that collected this data and made it available—and I hope that working with real data from a variety of domains makes the book more engaging for readers.

What's New?

For this third edition, I started by moving the book into Jupyter notebooks. This change has one immediate benefit—you can read the text, run the code, and work on the exercises all in one place. And the notebooks are designed to work on Google Colab, so you can get started without installing anything.

The move to notebooks has another benefit—the code is more visible. In the first two editions, some of the code was in the book and some was in supporting files available online. In retrospect, it's clear that splitting the material in this way was not ideal, and it made the code more complicated than it needed to be. In the third edition, I was able to simplify the code and make it more readable.

Since the last edition was published, I've developed a library called `empiricaldist` that provides objects that represent statistical distributions. This library is more mature now, so the updated code makes better use of it.

When I started this project, NumPy and SciPy were not as widely used, and Pandas even less, so the original code used Python data structures like lists and dictionaries. This edition uses arrays and Pandas structures extensively, and makes more use of functions these libraries provide.

The third edition covers the same topics as the original, in almost the same order, but the text is substantially revised. Some of the examples are new; others are updated with new data. I've developed new exercises, revised some of the old ones, and removed a few. I think the updated exercises are better connected to the examples, and more interesting.

Since the first edition, this book has been based on the thesis that many ideas that are hard to explain with math are easier to explain with code. In this edition, I have doubled down on this idea, to the point where there is almost no mathematical notation left.

Overall, I think these changes make *Think Stats* a better book. I hope you like it!

Using the Code

The code and data used in this book are available in a Git repository on GitHub. Git is a version control system that helps to keep track of the files that make up a project. A collection of files under Git's control is called a **repository**. GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

For each chapter in this book, the repository provides a Jupyter notebook, which is a document that contains the text, code, and the results of running the code. You can use these notebooks to run the code and work on the exercises.

There are two ways you can run the notebooks. By far the easier one is to use Colab, which is a service provided by Google where you can run the notebooks in a web browser without installing anything on your computer. If you start from the *Think Stats* home page, you will find links to the notebooks, including one that introduces Colab and Jupyter notebooks.

If you don't want to use Colab, you can download the notebooks and run them on your computer, but in that case you will have to install Python, Jupyter, and the libraries the book uses, including NumPy, SciPy, and StatsModels. If you have experience installing software, setting up an environment where you can run the notebooks is not difficult. But if you don't have that experience, your first attempt can be challenging, and sometimes frustrating. In that case, it can be a barrier to getting the most out of this book. If you want to learn about exploratory data analysis in Python, you don't want to spend your time and cognitive capacity on installing software!

So I strongly recommend that you run at least the first few chapters on Colab. Then, if you want to set up your own environment, you can do it without interrupting your progress in the book. And one last suggestion: if you have any problems installing software, take advantage of tools like ChatGPT—they generally provide good guidance on these topics.

I wrote this book assuming that the reader is familiar with core Python, including object-oriented features. If you are familiar with NumPy and Pandas, that will help, but it's not necessary—I'll explain what you need to know. I assume that the reader knows basic mathematics, including logarithms, for example, and summations. You don't need to know linear algebra or calculus. There is one place where I mention derivatives and integrals, but if you are not familiar with those concepts, they are entirely optional. Finally, I don't assume you know anything about statistics.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Think Stats*, third edition, by Allen B. Downey (O'Reilly). Copyright 2025 Allen B. Downey, 978-1-098-19025-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Bold
: Indicates new technical terms, each of which has a corresponding glossary entry.

Italic
: Indicates URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/think-stats-3e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

Watch us on YouTube: *https://youtube.com/oreillymedia*

Acknowledgments

Thanks to the readers who contributed corrections and suggestion to previous editions of this book, and to the students at Olin College who suffered through the rougher drafts.

Many thanks to the technical reviewers of this edition: Peter Bruce, Zachary del Rosario, Walter R. Paczkowski, and Thomas Nield.

And thank you to everyone at O'Reilly Media, especially editors Sara Hunter and Aaron Black.

CHAPTER 1

Exploratory Data Analysis

The thesis of this book is we can use data to answer questions, resolve debates, and make better decisions.

This chapter introduces the steps we'll use to do that: loading and validating data, exploring, and choosing statistics that measure what we are interested in. As an example, we'll use data from the National Survey of Family Growth (NSFG) to answer a question I heard when my wife and I were expecting our first child: do first babies tend to arrive late?

Evidence

You might have heard that first babies are more likely to be late. If you search the web with this question, you will find plenty of discussion. Some people claim it's true, others say it's a myth, and some people say it's the other way around: first babies come early.

In many of these discussions, people provide data to support their claims. I found many examples like these:

> "My two friends that have given birth recently to their first babies, BOTH went almost 2 weeks overdue before going into labour or being induced."
>
> "My first one came 2 weeks late and now I think the second one is going to come out two weeks early!!"
>
> "I don't think that can be true because my sister was my mother's first and she was early, as with many of my cousins."

Reports like these are called **anecdotal evidence** because they are based on data that is unpublished and usually personal. In casual conversation, there is nothing wrong with anecdotes, so I don't mean to pick on the people I quoted.

But we might want evidence that is more persuasive and an answer that is more reliable. By those standards, anecdotal evidence usually fails, due to:

Small number of observations
: If pregnancy length is longer for first babies, the difference is probably small compared to natural variation. In that case, we might have to compare a large number of pregnancies to know whether there is a difference.

Selection bias
: People who join a discussion of this question might be interested because their first babies were late. In that case the process of selecting data would bias the results.

Confirmation bias
: People who believe the claim might be more likely to contribute examples that confirm it. People who doubt the claim are more likely to cite counterexamples.

Inaccuracy
: Anecdotes are often personal stories, and might be misremembered, misrepresented, repeated inaccurately, etc.

To address the limitations of anecdotes, we will use the tools of statistics, which include:

Data collection
: We will use data from a large national survey that was designed explicitly with the goal of generating statistically valid inferences about the US population.

Descriptive statistics
: We will generate statistics that summarize the data concisely, and evaluate different ways to visualize data.

Exploratory data analysis
: We will look for patterns, differences, and other features that address the questions we are interested in. At the same time we will check for inconsistencies and identify limitations.

Estimation
: We will use data from a sample to estimate characteristics of the general population.

Hypothesis testing
: Where we see apparent effects, like a difference between two groups, we will evaluate whether the effect might have happened by chance.

By performing these steps with care to avoid pitfalls, we can reach conclusions that are more justified and more likely to be correct.

The National Survey of Family Growth

Since 1973 the US Centers for Disease Control and Prevention (CDC) have conducted the National Survey of Family Growth (NSFG), which is intended to gather "information on family life, marriage and divorce, pregnancy, infertility, use of contraception, and men's and women's health. The survey results are used…to plan health services and health education programs, and to do statistical studies of families, fertility, and health."

We will use data collected by this survey to investigate whether first babies tend to be born late, and other questions. To use this data effectively, we have to understand the design of the study.

In general, the goal of a statistical study is to draw conclusions about a **population**. In the NSFG, the target population is people in the United States aged 15–44.

Ideally, surveys would collect data from every member of the population, but that's seldom possible. Instead we collect data from a subset of the population called a **sample**. The people who participate in a survey are called **respondents**.

The NSFG is a **cross-sectional study**, which means that it captures a snapshot of a population at a point in time. The NSFG has been conducted several times now; each deployment is called a **cycle**. We will use data from Cycle 6, which was conducted from January 2002 to March 2003.

In general, cross-sectional studies are meant to be **representative**, which means that the sample is similar to the target population in all ways that are important for the purposes of the study. That ideal is hard to achieve in practice, but people who conduct surveys come as close as they can.

The NSFG is not representative; instead it is **stratified**, which means that it deliberately **oversamples** some groups. The designers of the study recruited three groups—Hispanics, African-Americans and teenagers—at rates higher than their representation in the US population to make sure that the number of respondents in each group is large enough to draw valid conclusions. The drawback of oversampling is that it is not as easy to draw conclusions about the population based on statistics from the sample. We will come back to this point later.

When working with this kind of data, it is important to be familiar with the **codebook**, which documents the design of the study, the survey questions, and the encoding of the responses.

Reading the Data

Before downloading NSFG data, you have to agree to the terms of use:

> Any intentional identification or disclosure of an individual or establishment violates the assurances of confidentiality given to the providers of the information. Therefore, users will:
>
> - Use the data in this dataset for statistical reporting and analysis only.
> - Make no attempt to learn the identity of any person or establishment included in these data.
> - Not link this dataset with individually identifiable data from other NCHS or non-NCHS datasets.
> - Not engage in any efforts to assess disclosure methodologies applied to protect individuals and establishments or any research on methods of re-identification of individuals and establishments.

If you agree to comply with these terms, instructions for downloading the data are in the notebook for this chapter.

The data is stored in two files, a "dictionary" that describes the format of the data, and a data file:

```
dct_file = "2002FemPreg.dct"
dat_file = "2002FemPreg.dat.gz"
```

The notebook for this chapter defines a function that reads these files. It is called `read_stata` because this data format is compatible with a statistical software package called Stata.

Here's how we use it:

```
preg = read_stata(dct_file, dat_file)
```

The result is a `DataFrame`, which is a Pandas data structure that represents tabular data in rows and columns. This `DataFrame` contains a row for each pregnancy reported by a respondent and a column for each **variable**. A variable can contain responses to a survey question or values that are calculated based on responses to one or more questions.

In addition to the data, a `DataFrame` also contains the variable names and their types, and it provides methods for accessing and modifying the data. The `DataFrame` has an attribute called `shape` that contains the number of rows and columns:

```
preg.shape
```

```
(13593, 243)
```

This dataset has 243 variables with information about 13,593 pregnancies. The `DataFrame` provides a method called `head` that displays the first few rows:

```
preg.head()
```

	caseid	pregordr	howpreg_n	howpreg_p	moscurrp	nowprgdk	pregend1	pregend2	nbrnaliv	...
0	1	1	NaN	NaN	NaN	NaN	6.0	NaN	1.0	...
1	1	2	NaN	NaN	NaN	NaN	6.0	NaN	1.0	...
2	2	1	NaN	NaN	NaN	NaN	5.0	NaN	3.0	...
3	2	2	NaN	NaN	NaN	NaN	6.0	NaN	1.0	...
4	2	3	NaN	NaN	NaN	NaN	6.0	NaN	1.0	...

The left column is the index of the `DataFrame`, which contains a label for each row. In this case, the labels are integers starting from 0, but they can also be strings and other types.

The `DataFrame` has an attribute called `columns` that contains the names of the variables:

```
preg.columns
```

```
Index(['caseid', 'pregordr', 'howpreg_n', 'howpreg_p', 'moscurrp', 'nowprgdk',
       'pregend1', 'pregend2', 'nbrnaliv', 'multbrth',
       ...
       'poverty_i', 'laborfor_i', 'religion_i', 'metro_i', 'basewgt',
       'adj_mod_basewgt', 'finalwgt', 'secu_p', 'sest', 'cmintvw'],
      dtype='object', length=243)
```

The column names are contained in an `Index` object, which is another Pandas data structure. To access a column from a `DataFrame`, you can use the column name as a key:

```
pregordr = preg["pregordr"]
type(pregordr)
```

```
pandas.core.series.Series
```

The result is a Pandas `Series`, which represents a sequence of values. `Series` also provides `head`, which displays the first few values and their labels:

```
pregordr.head()
```

```
0    1
1    2
2    1
3    2
4    3
Name: pregordr, dtype: int64
```

The last line includes the name of the `Series` and `dtype`, which is the type of the values. In this example, `int64` indicates that the values are 64-bit integers.

The NSFG dataset contains 243 variables in total. Here are some of the ones we'll use for the explorations in this book:

`caseid`
: The integer ID of the respondent.

`pregordr`
: A pregnancy serial number: the code for a respondent's first pregnancy is 1, for the second pregnancy is 2, and so on.

`prglngth`
: The integer duration of the pregnancy in weeks.

`outcome`
: An integer code for the outcome of the pregnancy. The code 1 indicates a live birth.

`birthord`
: A serial number for live births: the code for a respondent's first child is 1, and so on. For outcomes other than live birth, this field is blank.

`birthwgt_lb` *and* `birthwgt_oz`
: Contain the pounds and ounces parts of the birth weight of the baby.

`agepreg`
: The mother's age at the end of the pregnancy.

`finalwgt`
: The statistical weight associated with the respondent. It is a floating-point value that indicates the number of people in the US population that this respondent represents.

If you read the codebook carefully, you will see that many of the variables are **recodes**, which means that they are not part of the **raw data** collected by the survey—they are calculated using the raw data.

For example, `prglngth` for live births is equal to the raw variable `wksgest` (weeks of gestation) if it is available; otherwise it is estimated using `mosgest * 4.33` (months of gestation times the average number of weeks in a month).

Recodes are often based on logic that checks the consistency and accuracy of the data. In general it is a good idea to use recodes when they are available, unless there is a compelling reason to process the raw data yourself.

Validation

When data is exported from one software environment and imported into another, errors might be introduced. And when you are getting familiar with a new dataset, you might decode data incorrectly or misunderstand its meaning. If you invest time to validate the data, you can save time later and avoid errors.

One way to validate data is to compute basic statistics and compare them with published results. For example, the NSFG codebook includes tables that summarize each variable. Here is the table for `outcome`, which encodes the outcome of each pregnancy:

Value	Label	Total
1	LIVE BIRTH	9148
2	INDUCED ABORTION	1862
3	STILLBIRTH	120
4	MISCARRIAGE	1921
5	ECTOPIC PREGNANCY	190
6	CURRENT PREGNANCY	352
Total		13593

The "Total" column indicates the number of pregnancies with each outcome. To check these totals, we'll use the `value_counts` method, which counts the number of times each value appears, and `sort_index`, which sorts the results according to the values in the `Index` (the left column):

```
preg["outcome"].value_counts().sort_index()
```

```
outcome
1    9148
2    1862
3     120
4    1921
5     190
6     352
Name: count, dtype: int64
```

Comparing the results with the published table, we can confirm that the values in `outcome` are correct. Similarly, here is the published table for `birthwgt_lb`:

Value	Label	Total
.	inapplicable	4449
0-5	UNDER 6 POUNDS	1125
6	6 POUNDS	2223
7	7 POUNDS	3049
8	8 POUNDS	1889
9-95	9 POUNDS OR MORE	799
97	Not ascertained	1
98	REFUSED	1
99	DON'T KNOW	57
Total		13593

Birth weight is only recorded for pregnancies that ended in a live birth. The table indicates that there are 4,449 cases where this variable is inapplicable. In addition, there is one case where the question was not asked, one where the respondent did not answer, and 57 cases where they did not know.

Again, we can use `value_counts` to compare the counts in the dataset to the counts in the codebook:

```
counts = preg["birthwgt_lb"].value_counts(dropna=False).sort_index()
counts
```

```
birthwgt_lb
0.0        8
1.0       40
2.0       53
3.0       98
4.0      229
5.0      697
6.0     2223
7.0     3049
8.0     1889
9.0      623
10.0     132
11.0      26
12.0      10
13.0       3
14.0       3
15.0       1
51.0       1
97.0       1
98.0       1
99.0      57
NaN     4449
Name: count, dtype: int64
```

The argument `dropna=False` means that `value_counts` does not ignore values that are "NA" or "Not applicable." These values appear in the results as `NaN`, which stands for "Not a number"—and the count of these values is consistent with the count of inapplicable cases in the codebook.

The counts for 6, 7, and 8 pounds are consistent with the codebook. To check the counts for the weight range from 0 to 5 pounds, we can use an attribute called `loc`—which is short for "location"—and a slice index to select a subset of the counts:

```
counts.loc[0:5]
```

```
birthwgt_lb
0.0      8
1.0     40
2.0     53
3.0     98
4.0    229
5.0    697
Name: count, dtype: int64
```

And we can use the `sum` method to add them up:

```
counts.loc[0:5].sum()
```

```
1125
```

The total is consistent with the codebook.

The values 97, 98, and 99 represent cases where the birth weight is unknown. There are several ways we might handle missing data. A simple option is to replace these values with `NaN`. At the same time, we will also replace a value that is clearly wrong, 51 pounds.

We can use the replace method like this:

```
preg["birthwgt_lb"] = preg["birthwgt_lb"].replace([51, 97, 98, 99], np.nan)
```

The first argument is a list of values to be replaced. The second argument, `np.nan`, gets the `NaN` value from NumPy.

When you read data like this, you often have to check for errors and deal with special values. Operations like this are called **data cleaning**.

Transformation

As another kind of data cleaning, sometimes we have to convert data into different formats, and perform other calculations.

For example, `agepreg` contains the mother's age at the end of the pregnancy. According to the codebook, it is an integer number of centiyears (hundredths of a year), as we can tell if we use the `mean` method to compute its average:

```
preg["agepreg"].mean()
```

```
2468.8151197039497
```

To convert it to years, we can divide through by 100:

```
preg["agepreg"] /= 100.0
preg["agepreg"].mean()
```

```
24.6881511970395
```

Now the average is more credible.

As another example, `birthwgt_lb` and `birthwgt_oz` contain birth weights with the pounds and ounces in separate columns. It will be more convenient to combine them into as single column that contains weights in pounds and fractions of a pound.

First we'll clean `birthwgt_oz` as we did with `birthwgt_lb`:

```
preg["birthwgt_oz"] = preg["birthwgt_oz"].replace([97, 98, 99], np.nan)
```

Now we can use the cleaned values to create a new column that combines pounds and ounces into a single quantity:

```
preg["totalwgt_lb"] = preg["birthwgt_lb"] + preg["birthwgt_oz"] / 16.0
preg["totalwgt_lb"].mean()
```

```
7.265628457623368
```

The average of the result seems plausible.

Summary Statistics

A **statistic** is a number derived from a dataset, usually intended to quantify some aspect of the data. Examples include the count, mean, variance, and standard deviation.

A `Series` object has a `count` method that returns the number of values that are not `nan`:

```
weights = preg["totalwgt_lb"]
n = weights.count()
n
```

```
9038
```

It also provides a `sum` method that returns the sum of the values—we can use it to compute the mean like this:

```
mean = weights.sum() / n
mean
```

```
7.265628457623368
```

But as we've already seen, there's also a `mean` method that does the same thing:

```
weights.mean()
```

```
7.265628457623368
```

In this dataset, the average birth weight is about 7.3 pounds.

Variance is a statistic that quantifies the spread of a set of values. It is the mean of the squared deviations, which are the distances of each point from the mean:

```
squared_deviations = (weights - mean) ** 2
```

We can compute the mean of the squared deviations like this:

```
var = squared_deviations.sum() / n
var
```

```
1.983070989750022
```

As you might expect, `Series` provides a `var` method that does *almost* the same thing:

```
weights.var()
```

```
1.9832904288326545
```

The result is slightly different because when the `var` method computes the mean of the squared deviations, it divides by `n-1` rather than `n`. That's because there are two ways to compute the variance of a sample, depending on what you are trying to do. I'll explain the difference in "Estimating Variance" on page 137—but in practice it usually doesn't matter. If you prefer the version with `n` in the denominator, you can get it by passing `ddof=0` as a keyword argument to the `var` method:

```
weights.var(ddof=0)
```

```
1.983070989750022
```

In this dataset, the variance of the birth weights is about 1.98, but that value is hard to interpret—for one thing, it is in units of pounds squared. Variance is useful in some computations, but not a good way to describe a dataset. A better option is the **standard deviation**, which is the square root of variance. We can compute it like this:

```
std = np.sqrt(var)
std
```

```
1.40821553384062
```

Or, we can use the `std` method:

```
weights.std(ddof=0)
```

```
1.40821553384062
```

In this dataset, the standard deviation of birth weights is about 1.4 pounds. Informally, values that are one or two standard deviations from the mean are common—values farther from the mean are rare.

Interpretation

To work with data effectively, you have to think on two levels at the same time: the level of statistics and the level of context. As an example, let's select the rows in the pregnancy file with `caseid` 10229. The `query` method takes a string that can contain column names, comparison operators, and numbers, among other things:

```
subset = preg.query("caseid == 10229")
subset.shape
```

```
(7, 244)
```

The result is a `DataFrame` that contains only the rows where the query is `True`. This respondent reported seven pregnancies—here are their outcomes, which are recorded in chronological order:

```
subset["outcome"].values
```

```
array([4, 4, 4, 4, 4, 4, 1])
```

The outcome code `1` indicates a live birth. Code `4` indicates a miscarriage—that is, a pregnancy loss, usually with no known medical cause.

Statistically this respondent is not unusual. Pregnancy loss is common and there are other respondents who reported as many instances. But remembering the context, this data tells the story of a woman who was pregnant six times, each time ending in miscarriage. Her seventh and most recent pregnancy ended in a live birth. If we consider this data with empathy, it is natural to be moved by the story it tells.

Each row in the NSFG dataset represents a person who provided honest answers to many personal and difficult questions. We can use this data to answer statistical questions about family life, reproduction, and health. At the same time, we have an obligation to consider the people represented by the data, and to afford them respect and gratitude.

Glossary

The end of each chapter provides a glossary of words that are defined in the chapter:

anecdotal evidence
: Data collected informally from a small number of individual cases, often without systematic sampling.

cross-sectional study
: A study that collects data from a representative sample of a population at a single point or interval in time.

cycle
: One data-collection interval in a study that collects data at multiple intervals in time.

population
: The entire group of individuals or items that is the subject of a study.

sample
: A subset of a population, often chosen at random.

respondents
: People who participate in a survey and respond to questions.

representative
: A sample is representative if it is similar to the population in ways that are important for the purposes of the study.

stratified
: A sample is stratified if it deliberately oversamples some groups, usually to make sure that enough members are included to support valid conclusions.

oversampled
: A group is oversampled if its members have a higher chance of appearing in a sample.

variable
: In survey data, a variable is a collection of responses to questions or values computed from responses.

codebook
: A document that describes the variables in a dataset, and provides other information about the data.

recode
: A variable that is computed based on other variables in a dataset.

raw data
: Data that has not been processed after collection.

data cleaning
: A process for identifying and correcting errors in a dataset, dealing with missing values, and computing recodes.

statistic
: A value that describes or summarizes a property of a sample.

standard deviation
: A statistic that quantifies the spread of data around the mean.

Exercises

The exercises for this chapter are based on the NSFG pregnancy file.

Exercise 1.1

Select the `birthord` column from `preg`, print the value counts, and compare to the results published in the NSFG Cycle 6 pregnancy codebook.

Exercise 1.2

Create a new column named `totalwgt_kg` that contains birth weight in kilograms (there are approximately 2.2 pounds per kilogram). Compute the mean and standard deviation of the new column.

Exercise 1.3

What are the pregnancy lengths for the respondent with `caseid` 2298?

What was the birth weight of the first baby born to the respondent with `caseid` 5013? Hint: You can use `and` to check more than one condition in a query.

CHAPTER 2
Distributions

This chapter introduces one of the most fundamental ideas in statistics, the distribution. We'll start with frequency tables—which represent the values in a dataset and the number of times each of them appears—and use them to explore data from the National Survey of Family Growth (NSFG). We'll also look for extreme or erroneous values, called outliers, and consider ways to handle them.

Frequency Tables

One way to describe a variable is a **frequency table**, which contains the values of the variable and their **frequencies**—that is, the number of times each value appears. This description is called the **distribution** of the variable.

To represent distributions, we'll use a library called `empiricaldist`. In this context, "empirical" means that the distributions are based on data rather than mathematical models. `empiricaldist` provides a class called `FreqTab` that we can use to compute and plot frequency tables. We can import it like this:

```
from empiricaldist import FreqTab
```

To show how it works, we'll start with a small list of values:

```
t = [1.0, 2.0, 2.0, 3.0, 5.0]
```

`FreqTab` provides a method called `from_seq` that takes a sequence and makes a `FreqTab` object:

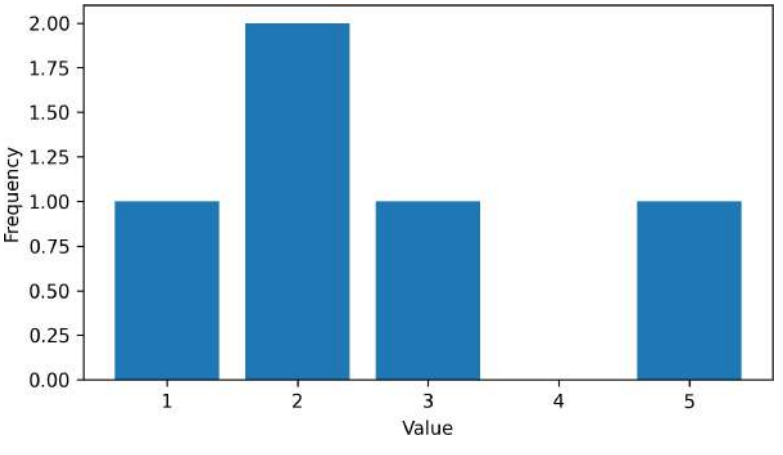
```
ftab = FreqTab.from_seq(t)
ftab
```

	freqs
1.0	1
2.0	2
3.0	1
5.0	1

A `FreqTab` object is a kind of Pandas `Series` that contains values and their frequencies. In this example, the value `1.0` corresponds to frequency 1, the value `2.0` corresponds to frequency 2, etc.

`FreqTab` provides a method called `bar` that plots the frequency table as a bar chart:

```
ftab.bar()
decorate(xlabel="Value", ylabel="Frequency")
```



Because a `FreqTab` is a Pandas `Series`, we can use the bracket operator to look up a value and get its frequency:

```
ftab[2.0]
```

```
2
```

But unlike a Pandas `Series`, we can also call a `FreqTab` object like a function to look up a value:

```
ftab(2.0)
```

```
2
```

If we look up a value that does not appear in the `FreqTab`, the function syntax returns `0`:

```
ftab(4.0)
```

```
0
```

A `FreqTab` object has an attribute called `qs` that returns an array of values—`qs` stands for quantities, although technically not all values are quantities:

```
ftab.qs
```

```
array([1., 2., 3., 5.])
```

`FreqTab` also has an attribute called `fs` that returns an array of frequencies:

```
ftab.fs
```

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

`FreqTab` provides an `items` method we can use to loop through quantity-frequency pairs:

```
for x, freq in ftab.items():
    print(x, freq)
```

```
1.0 1
2.0 2
3.0 1
5.0 1
```

We'll see more `FreqTab` methods as we go along.

NSFG Distributions

When you start working with a new dataset, I suggest you explore the variables you are planning to use one at a time, and a good way to start is by looking at frequency tables.

As an example, let's look at data from the NSFG. In the previous chapter, we downloaded this dataset, read it into a Pandas `DataFrame`, and cleaned a few of the variables. The code we used to load and clean the data is in a module called `nsfg.py`—instructions for installing this module are in the notebook for this chapter.

We can import it and read the pregnancy file like this:

```
from nsfg import read_fem_preg

preg = read_fem_preg()
```

For the examples in this chapter, we'll focus on pregnancies that ended in live birth. We can use the `query` method to select the rows where `outcome` is 1:

```
live = preg.query("outcome == 1")
```

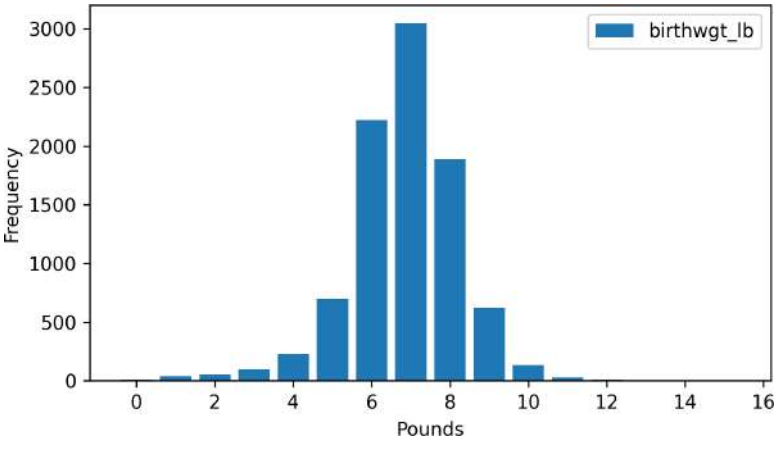
In the string that's passed to `query`, variable names like `outcome` refer to column names in the `DataFrame`. This string can also contain operators like `==` and operands like `1`.

Now we can use `FreqTab.from_seq` to count the number of times each quantity appears in `birthwgt_lb`, which is the pounds part of the birth weights. The `name` argument gives the `FreqTab` object a name, which is used as a label when we plot it:

```
ftab_lb = FreqTab.from_seq(live["birthwgt_lb"], name="birthwgt_lb")
```

Here's what the distribution looks like:

```
ftab_lb.bar()
decorate(xlabel="Pounds", ylabel="Frequency")
```



Looking at a distribution like this, the first thing we notice is the shape, which resembles the famous bell curve, more formally called a normal distribution or a Gaussian distribution. The other notable feature of the distribution is the **mode**, which is the most common value. To find the mode, we can use the method `idxmax`, which finds the quantity associated with the highest frequency:

```
ftab_lb.idxmax()
```

```
7.0
```

`FreqTab` provides a method called `mode` that does the same thing:

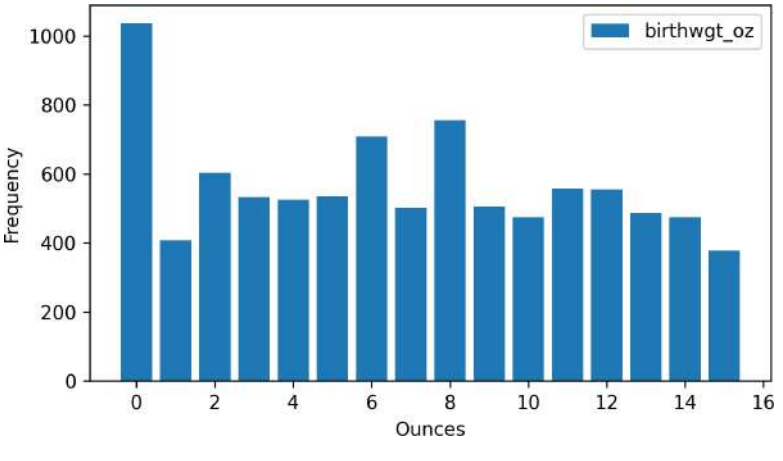
```
ftab_lb.mode()
```

```
7.0
```

In this distribution, the mode is at 7 pounds.

As another example, here's the frequency table of `birthwgt_oz`, which is the ounces part of birth weight:

```
ftab_oz = FreqTab.from_seq(live["birthwgt_oz"], name="birthwgt_oz")
ftab_oz.bar()
decorate(xlabel="Ounces", ylabel="Frequency")
```



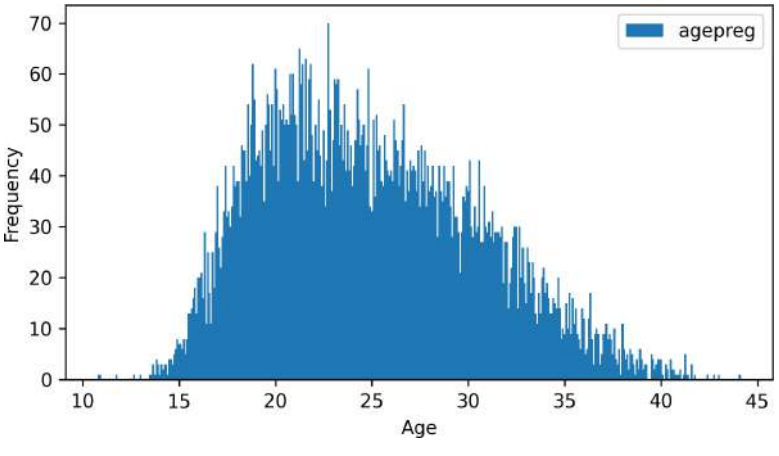
Because nature doesn't know about pounds and ounces, we might expect all values of `birthwgt_oz` to be equally likely—that is, this distribution should be **uniform**. But it looks like `0` is more common than the other quantities, and `1` and `15` are less common, which suggests that respondents round off birth weights that are close to a whole number of pounds.

As another example, let's look at the frequency table of `agepreg`, which is the mother's age at the end of pregnancy:

```
ftab_age = FreqTab.from_seq(live["agepreg"], name="agepreg")
```

In the NSFG, age is recorded in years and months, so there are more unique values than in the other distributions we've looked at. For that reason, we'll pass `width=0.1` as a keyword argument to the `bar` method, which adjusts the width of the bars so they don't overlap too much:

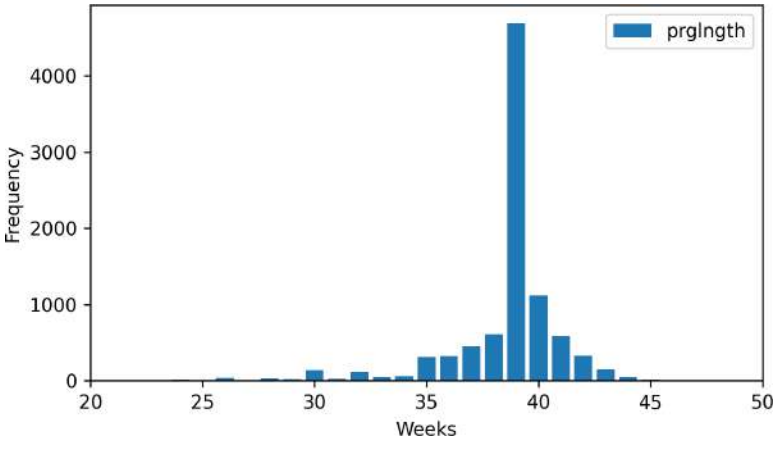
```
ftab_age.bar(width=0.1)
decorate(xlabel="Age", ylabel="Frequency")
```



The distribution is very roughly bell-shaped, but it is **skewed** to the right—that is, the tail extends farther right than left.

Finally, let's look at the frequency table of `prglngth`, which is the length of the pregnancy in weeks. The `xlim` argument sets the limit of the x-axis to the range from 20 to 50 weeks—there are not many values outside this range, and they are probably errors:

```
ftab_length = FreqTab.from_seq(live["prglngth"], name="prglngth")
ftab_length.bar()
decorate(xlabel="Weeks", ylabel="Frequency", xlim=[20, 50])
```



By far the most common quantity is 39 weeks. The left tail is longer than the right—early babies are common, but pregnancies seldom go past 43 weeks, and doctors often intervene if they do.

Outliers

Looking at frequency tables, it is easy to identify the shape of the distribution and the most common quantities, but rare quantities are not always visible. Before going on, it is a good idea to check for **outliers**, which are extreme values that might be measurement or recording errors, or might be accurate reports of rare events.

To identify outliers, the following function takes a `FreqTab` object and an integer `n`, and uses a slice index to select the `n` smallest quantities and their frequencies:

```
def smallest(ftab, n=10):
    return ftab[:n]
```

In the frequency table of `prglngth`, here are the 10 smallest values:

```
smallest(ftab_length)
```

```
prglngth
0     1
4     1
9     1
13    1
17    2
18    1
19    1
20    1
21    2
22    7
Name: prglngth, dtype: int64
```

Since we selected the rows for live births, pregnancy lengths less than 10 weeks are certainly errors. The most likely explanation is that the outcome was not coded correctly. Lengths higher than 30 weeks are probably legitimate. Between 10 and 30 weeks, it is hard to be sure—some quantities are probably errors, but some are correctly recorded preterm births.

The following function selects the largest values from a `FreqTab` object:

```
def largest(ftab, n=10):
    return ftab[-n:]
```

Here are the longest pregnancy lengths in the dataset:

```
largest(ftab_length)
```

```
prglngth
40    1116
41     587
42     328
43     148
44      46
45      10
46       1
47       1
48       7
50       2
Name: prglngth, dtype: int64
```

Again, some of these values are probably errors. Most doctors recommend induced labor if a pregnancy exceeds 41 weeks, so 50 weeks seems unlikely to be correct. But there is no clear line between values that are certainly errors and values that might be correct reports of rare events.

The best way to handle outliers depends on "domain knowledge"—that is, information about where the data come from and what they mean. And it depends on what analysis you are planning to perform.

In this example, the motivating question is whether first babies tend to be earlier or later than other babies. So we'll use statistics that are not thrown off too much by a small number of incorrect values.

First Babies

Now let's compare the distribution of pregnancy lengths for first babies and others. We can use the `query` method to select rows that represent first babies and others:

```
firsts = live.query("birthord == 1")
others = live.query("birthord != 1")
```

And make a `FreqTab` of pregnancy lengths for each group:

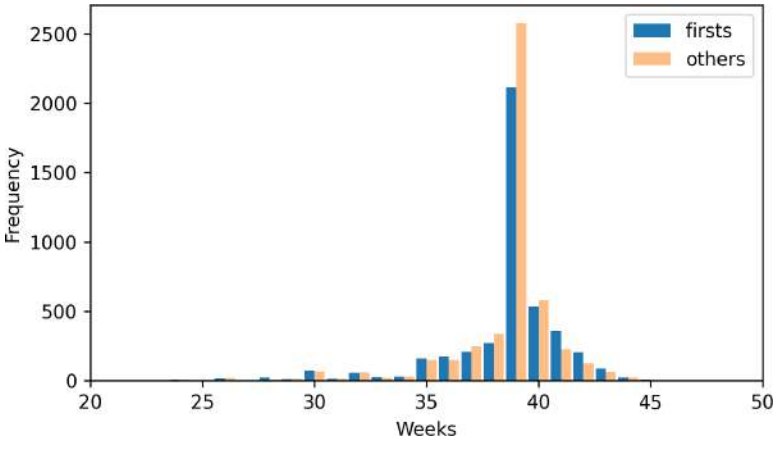
```
ftab_first = FreqTab.from_seq(firsts["prglngth"], name="firsts")
ftab_other = FreqTab.from_seq(others["prglngth"], name="others")
```

The following function plots two frequency tables side by side:

```
def two_bar_plots(ftab1, ftab2, width=0.45):
    ftab1.bar(align="edge", width=-width)
    ftab2.bar(align="edge", width=width, alpha=0.5)
```

Here's what they look like:

```
two_bar_plots(ftab_first, ftab_other)
decorate(xlabel="Weeks", ylabel="Frequency", xlim=[20, 50])
```



There is no obvious difference in the shape of the distributions or in the outliers. It looks like more of the nonfirst babies are born during week 39, but there are more nonfirst babies in the dataset, so we should not compare the counts directly:

```
firsts["prglngth"].count(), others["prglngth"].count()
```

```
(4413, 4735)
```

Comparing the means of the distributions, it looks like first babies are a little bit later on average:

```
first_mean = firsts["prglngth"].mean()
other_mean = others["prglngth"].mean()
first_mean, other_mean
```

```
(38.60095173351461, 38.52291446673706)
```

But the difference is only 0.078 weeks, which is about 13 hours:

```
diff = first_mean - other_mean
diff, diff * 7 * 24
```

```
(0.07803726677754952, 13.11026081862832)
```

There are several possible causes of this apparent difference:

- There might be an actual difference in average pregnancy length between first babies and others.
- The apparent difference we see in this dataset might be the result of bias in the sampling process—that is, the selection of survey respondents.
- The apparent difference might be the result of measurement error—for example, the self-reported pregnancy lengths might be more accurate for first babies or others.
- The apparent difference might be the result of random variation in the sampling process.

In later chapters, we will consider these possible explanations more carefully, but for now we will take this result at face value: in this dataset, there is a small difference in pregnancy length between these groups.

Effect Size

A difference like this is sometimes called an "effect." There are several ways to quantify the magnitude of an effect. The simplest is to report the difference in absolute terms—in this example, the difference is 0.078 weeks.

Another is to report the difference in relative terms. For example, we might say that first pregnancies are 0.2% longer than others, on average:

```
diff / live["prglngth"].mean() * 100
```

```
0.20237586646738304
```

Another option is to report a **standardized** effect size, which is a statistic intended to quantify the size of an effect in a way that is comparable between different quantities and different groups.

Standardizing means we express the difference as a multiple of the standard deviation. So we might be tempted to write something like this:

```
diff / live["prglngth"].std()
```

```
0.028877623375210403
```

But notice that we used both groups to compute the standard deviation. If the groups are substantially different, the standard deviation when we put them together is larger than in either group, which might make the effect size seem small.

An alternative is to use the standard deviation of just one group, but it's not clear which. So we could take the average of the two standard deviations, but if the groups are different sizes, that would give too much weight to one group and not enough to the other.

A common solution is to use **pooled standard deviation**, which is the square root of pooled variance, which is the weighted sum of the variances in the groups. To compute it, we'll start with the variances:

```
group1, group2 = firsts["prglngth"], others["prglngth"]
```

```
v1, v2 = group1.var(), group2.var()
```

Here is the weighted sum, with the group sizes as weights:

```
n1, n2 = group1.count(), group2.count()
pooled_var = (n1 * v1 + n2 * v2) / (n1 + n2)
```

Finally, here is the pooled standard deviation:

```
np.sqrt(pooled_var)
```

```
2.7022108144953862
```

The pooled standard deviation is between the standard deviations of the groups:

```
firsts["prglngth"].std(), others["prglngth"].std()
```

```
(2.7919014146687204, 2.6158523504392375)
```

A standardized effect size that uses pooled standard deviation is called **Cohen's effect size**. Here's a function that computes it:

```
def cohen_effect_size(group1, group2):
    diff = group1.mean() - group2.mean()

    v1, v2 = group1.var(), group2.var()
    n1, n2 = group1.count(), group2.count()
    pooled_var = (n1 * v1 + n2 * v2) / (n1 + n2)

    return diff / np.sqrt(pooled_var)
```

And here's the effect size for the difference in mean pregnancy lengths:

```
cohen_effect_size(firsts["prglngth"], others["prglngth"])
```

```
0.028879044654449834
```

In this example, the difference is 0.029 standard deviations, which is small. To put that in perspective, the difference in height between men and women is about 1.7 standard deviations.

Reporting Results

We have seen several ways to describe the difference in pregnancy length (if there is one) between first babies and others. How should we report these results?

The answer depends on who is asking the question. A scientist might be interested in any (real) effect, no matter how small. A doctor might only care about effects that are **practically significant**—that is, differences that matter in practice. A pregnant woman might be interested in results that are relevant to her, like the probability of delivering early or late.

How you report results also depends on your goals. If you are trying to demonstrate the importance of an effect, you might choose summary statistics that emphasize differences. If you are trying to reassure a patient, you might choose statistics that put the differences in context.

Of course your decisions should also be guided by professional ethics. It's OK to be persuasive—you *should* design statistical reports and visualizations that tell a story clearly. But you should also do your best to make your reports honest, and to acknowledge uncertainty and limitations.

Glossary

distribution
: The set of values and how frequently each value appears in a dataset.

frequency table
: A mapping from values to frequencies.

frequency
: The number of times a value appears in a sample.

skewed
: A distribution is skewed if it is asymmetrical, with extreme quantities extending farther in one direction than the other.

mode
: The most frequent quantity in a sample, or one of the most frequent quantities.

uniform distribution
: A distribution in which all quantities have the same frequency.

outlier
: An extreme quantity in a distribution.

standardized
: A statistic is standardized if it is expressed in terms that are comparable across different datasets and domains.

pooled standard deviation
: A statistic that combines data from two or more groups to compute a common standard deviation.

Cohen's effect size
: A standardized statistic that quantifies the difference in the means of two groups.

practically significant
: An effect is practically significant if it is big enough to matter in practice.

Exercises

For the exercises in this chapter, we'll load the NSFG female respondent file, which contains one row for each female respondent. Instructions for downloading the data and the codebook are in the notebook for this chapter.

The `nsfg.py` module provides a function that reads the female respondent file, cleans some of the variables, and returns a `DataFrame`:

```
from nsfg import read_fem_resp

resp = read_fem_resp()
resp.shape
```

```
(7643, 3092)
```

This `DataFrame` contains 3,092 columns, but we'll use just a few of them.

Exercise 2.1

We'll start with `totincr`, which records the total income for the respondent's family, encoded with a value from 1 to 14. You can read the codebook for the respondent file to see what income level each value represents.

Make a `FreqTab` object to represent the distribution of this variable and plot it as a bar chart.

Exercise 2.2

Make a frequency table of the `parity` column, which records the number of children each respondent has borne. How would you describe the shape of this distribution?

Use the `largest` function to find the largest values of `parity`. Are there any values you think are errors?

Exercise 2.3

Let's investigate whether people with higher income bear more children. Use the query method to select the respondents with the highest income (level 14). Plot the frequency table of `parity` for just the high income respondents.

Compare the mean `parity` for high income respondents and others.

Compute the Cohen's effect size for this difference. How does it compare with the difference in pregnancy length for first babies and others?

Do these results show that people with higher income have more children, or can you think of another explanation for the apparent difference?

CHAPTER 3

Probability Mass Functions

In the previous chapter we represented distributions using a `FreqTab` object, which contains a set of values and their frequencies—that is, the number of times each value appears. In this chapter we'll introduce another way to describe a distribution, a probability mass function (PMF).

To represent a PMF, we'll use an object called a `Pmf`, which contains a set of values and their probabilities. We'll use `Pmf` objects to compute the mean and variance of a distribution, and the skewness, which indicates whether it is skewed to the left or right. Finally, we will explore how a phenomenon called the "inspection paradox" can cause a sample to give a biased view of a distribution.

PMFs

A `Pmf` object is like a `FreqTab` that contains probabilities instead of frequencies. So one way to make a `Pmf` is to start with a `FreqTab`. For example, here's a `FreqTab` that represents the distribution of values in a short sequence:

```
from empiricaldist import FreqTab

ftab = FreqTab.from_seq([1, 2, 2, 3, 5])
ftab
```

	freqs
1	1
2	2
3	1
5	1

The sum of the frequencies is the size of the original sequence:

```
n = ftab.sum()
n
```

```
5
```

If we divide the frequencies by `n`, they represent proportions, rather than counts:

```
pmf = ftab / n
pmf
```

	probs
1	0.2
2	0.4
3	0.2
5	0.2

This result indicates that 20% of the values in the sequence are 1, 40% are 2, and so on.

We can also think of these proportions as probabilities in the following sense: if we choose a random value from the original sequence, the probability we choose the value 1 is 0.2, the probability we choose the value 2 is 0.4, and so on.

Because we divided through by `n`, the sum of the probabilities is 1, which means that this distribution is **normalized**:

```
pmf.sum()
```

```
1.0
```

A normalized `FreqTab` object represents a **probability mass function** (PMF), so-called because probabilities associated with discrete values are also called "probability masses."

The `empiricaldist` library provides a `Pmf` object that represents a probability mass function, so instead of creating a `FreqTab` object and then normalizing it, we can create a `Pmf` object directly:

```
from empiricaldist import Pmf

pmf = Pmf.from_seq([1, 2, 2, 3, 5])
pmf
```

	probs
1	0.2
2	0.4
3	0.2
5	0.2

The `Pmf` is normalized so the total probability is 1:

```
pmf.sum()
```

```
1.0
```

`Pmf` and `FreqTab` objects are similar in many ways. To look up the probability associated with a value, we can use the bracket operator:

```
pmf[2]
```

```
0.4
```

Or use parentheses to call the `Pmf` like a function:

```
pmf(2)
```

```
0.4
```

To assign a probability to a value, you have to use the bracket operator:

```
pmf[2] = 0.2
pmf(2)
```

```
0.2
```

You can modify an existing `Pmf` by incrementing the probability associated with a value:

```
pmf[2] += 0.3
pmf[2]
```

```
0.5
```

Or you can multiply a probability by a factor:

```
pmf[2] *= 0.5
pmf[2]
```

```
0.25
```

If you modify a `Pmf`, the result may not be normalized—that is, the probabilities may no longer add up to 1:

```
pmf.sum()
```

```
0.8500000000000001
```

The `normalize` method renormalizes the `Pmf` by dividing through by the sum—and returning the sum:

```
pmf.normalize()
```

```
0.8500000000000001
```

`Pmf` objects provide a `copy` method so you can make and modify a copy without affecting the original:

```
pmf.copy()
```

	probs
1	0.235294
2	0.294118
3	0.235294
5	0.235294

Like a `FreqTab` object, a `Pmf` object has a `qs` attribute that accesses the quantities and a `ps` attribute that accesses the probabilities.

It also has a `bar` method that plots the `Pmf` as a bar graph and a `plot` method that plots it as a line graph.

Summarizing a PMF

In "Summary Statistics" on page 11 we computed the mean of a sample by adding up the elements and dividing by the number of elements. Here's a simple example:

```
seq = [1, 2, 2, 3, 5]

n = len(seq)
mean = np.sum(seq) / n
mean
```

```
2.6
```

Now suppose we compute the PMF of the values in the sequence:

```
pmf = Pmf.from_seq(seq)
```

Given the `Pmf`, we can still compute the mean, but the process is different—we have to multiply the probabilities and quantities and add up the products:

```
mean = np.sum(pmf.ps * pmf.qs)
mean
```

```
2.6
```

Notice that we *don't* have to divide by `n`, because we already did that when we normalized the `Pmf`. `Pmf` objects have a `mean` method that does the same thing:

```
pmf.mean()
```

```
2.6
```

Given a `Pmf`, we can compute the variance by computing the deviation of each quantity from the mean:

```
deviations = pmf.qs - mean
```

Then we multiply the squared deviations by the probabilities and add up the products:

```
var = np.sum(pmf.ps * deviations**2)
var
```

```
1.84
```

The `var` method does the same thing:

```
pmf.var()
```

```
1.84
```

From the variance, we can compute the standard deviation in the usual way:

```
np.sqrt(var)
```

```
1.3564659966250536
```

Or the `std` method does the same thing:

```
pmf.std()
```

```
1.3564659966250536
```

`Pmf` also provides a `mode` method that finds the value with the highest probability:

```
pmf.mode()
```

```
2
```

We'll see more methods as we go along, but that's enough to get started.

The Class Size Paradox

As an example of what we can do with `Pmf` objects, let's consider a phenomenon I call "the class size paradox."

At many American colleges and universities, the student-to-faculty ratio is about 10:1. But students are often surprised that many of their classes have more than 10 students, sometimes a lot more. There are two reasons for the discrepancy:

- Students typically take four or five classes per semester, but professors often teach one or two.
- The number of students in a small class is small, and the number of students in a large class is large.

The first effect is obvious, at least once it is pointed out; the second is more subtle. Let's look at an example. Suppose that a college offers 65 classes in a given semester, and we are given the number of classes in each of the following size ranges:

```
ranges = pd.interval_range(start=5, end=50, freq=5, closed="left")
ranges.name = "class size"

data = pd.DataFrame(index=ranges)
data["count"] = [8, 8, 14, 4, 6, 12, 8, 3, 2]
data
```

class size	count
[5, 10)	8
[10, 15)	8
[15, 20)	14
[20, 25)	4
[25, 30)	6
[30, 35)	12
[35, 40)	8
[40, 45)	3
[45, 50)	2

The Pandas function `interval_range` makes an `Index` where each label represents a range of values. The notation `[5, 10)` means that `5` is included in the interval and `10` is not. Since we don't know the sizes of the classes in each interval, let's assume that all sizes are at the midpoint of the range:

```
sizes = ranges.left + 2
sizes
```

```
Index([7, 12, 17, 22, 27, 32, 37, 42, 47], dtype='int64')
```

Now let's make a `Pmf` that represents the distribution of class sizes. Because we know the sizes and their frequencies, we can create a `Pmf` directly, passing as arguments the counts, sizes, and a name. When we normalize the new `Pmf`, the result is the sum of the counts:

```
counts = data["count"]
actual_pmf = Pmf(counts, sizes, name="actual")
actual_pmf.normalize()
```

```
65
```

If you ask the college for the average class size, they report the mean of this distribution, which is 23.7:

```
actual_pmf.mean()
```

```
23.692307692307693
```

But if you survey a group of students, ask them how many students are in their classes, and compute the mean, the average is bigger. Let's see how much bigger.

The following function takes the actual `Pmf` of class sizes and makes a new `Pmf` that represents the class sizes as seen by students. The quantities in the two distributions are the same, but the probabilities in the distribution are multiplied by the quantities, because in a class with size `x`, there are `x` students who observe that class. So the probability of observing a class is proportional to its size:

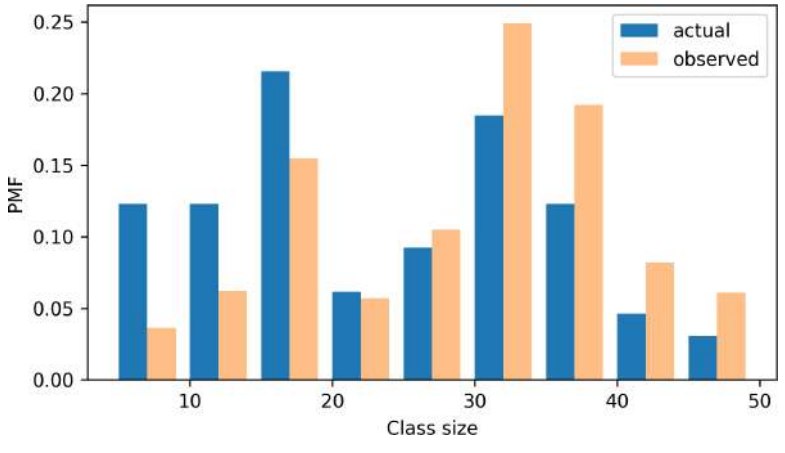
```
def bias(pmf, name):
    # multiply each probability by class size
    ps = pmf.ps * pmf.qs

    # make a new Pmf and normalize it
    new_pmf = Pmf(ps, pmf.qs, name=name)
    new_pmf.normalize()
    return new_pmf
```

Now we can compute the biased `Pmf` as observed by students:

```
observed_pmf = bias(actual_pmf, name="observed")
```

Here's what the two distributions look like:

```
from thinkstats import two_bar_plots

two_bar_plots(actual_pmf, observed_pmf, width=2)
decorate(xlabel="Class size", ylabel="PMF")
```



In the observed distribution there are fewer small classes and more large ones. And the biased mean is 29.1, almost 25% higher than the actual mean:

```
observed_pmf.mean()
```

```
29.123376623376622
```

It is also possible to invert this operation. Suppose you want to find the distribution of class sizes at a college, but you can't get reliable data. One option is to choose a random sample of students and ask how many students are in their classes.

The result would be biased for the reasons we've just seen, but you can use it to estimate the actual distribution. Here's the function that unbiases a `Pmf` by dividing the probabilities by the sizes:

```
def unbias(pmf, name):
    # divide each probability by class size
    ps = pmf.ps / pmf.qs

    new_pmf = Pmf(ps, pmf.qs, name=name)
    new_pmf.normalize()
    return new_pmf
```

And here's the result:

```
debiased_pmf = unbias(observed_pmf, "debiased")
debiased_pmf.mean()
```

```
23.692307692307693
```

The mean of the debiased `Pmf` is the same as the mean of the actual distribution we started with.

If you think this example is interesting, you might like Chapter 2 of my book *Probably Overthinking It* (University of Chicago Press, 2023), which includes this and several other examples of what's called the "inspection paradox."

NSFG Data

In the previous chapter, we plotted frequency tables of pregnancy lengths for first babies and others. But the sizes of the groups are not the same, so we can't compare the frequency tables directly. Because PMFs are normalized, we can compare them. So let's load the NSFG data again and make `Pmf` objects to represent distributions of pregnancy lengths.

The `nsfg` module provides a `read_nsfg_groups` function that reads the data, selects rows that represent live births, and partitions live births into first babies and others. It returns three `DataFrame` objects:

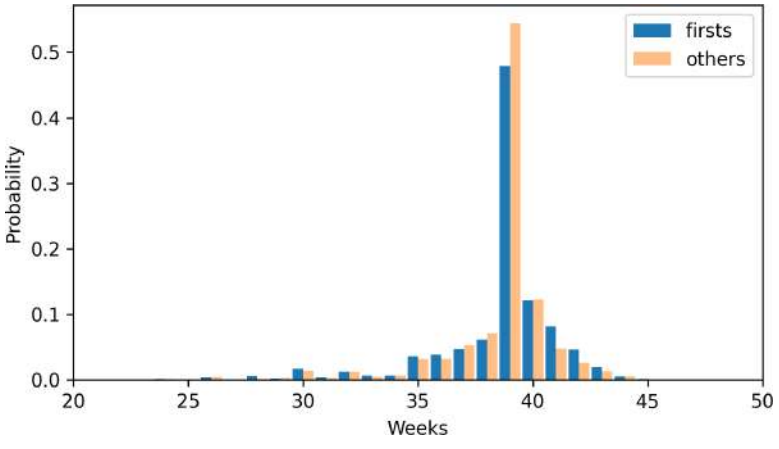
```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

We can use `firsts` and `others` to make a `Pmf` for the pregnancy lengths in each group:

```
first_pmf = Pmf.from_seq(firsts["prglngth"], name="firsts")
other_pmf = Pmf.from_seq(others["prglngth"], name="others")
```

Here are the PMFs for first babies and others, plotted as bar graphs:

```
two_bar_plots(first_pmf, other_pmf)
decorate(xlabel="Weeks", ylabel="Probability", xlim=[20, 50])
```



By plotting the PMF instead of the frequency table, we can compare the two distributions without being misled by the difference in sizes of the samples. Based on this figure, first babies seem to be less likely than others to arrive on time (week 39) and more likely to be late (weeks 41 and 42).

Other Visualizations

FreqTabs and PMFs are useful while you are exploring data and trying to identify patterns and relationships. Once you have an idea of what is going on, a good next step is to design a visualization that makes the patterns you have identified as clear as possible.

In the NSFG data, the biggest differences in the distributions are near the mode. So it makes sense to zoom in on that part of the graph, and select data from weeks 35 to 46.

When we call a `Pmf` object like a function, we can look up a sequence of quantities and get a sequence of probabilities:

```
weeks = range(35, 46)
first_pmf(weeks)
```

```
array([0.03602991, 0.03897575, 0.04713347, 0.06163608, 0.4790392,
       0.12145932, 0.08157716, 0.04645366, 0.01971448, 0.00521187,
       0.00135962])
```

```
other_pmf(weeks)
```

```
array([0.03210137, 0.03146779, 0.05216473, 0.07074974, 0.54466737,
       0.12249208, 0.04794087, 0.02597677, 0.01288279, 0.00485744,
       0.00084477])
```

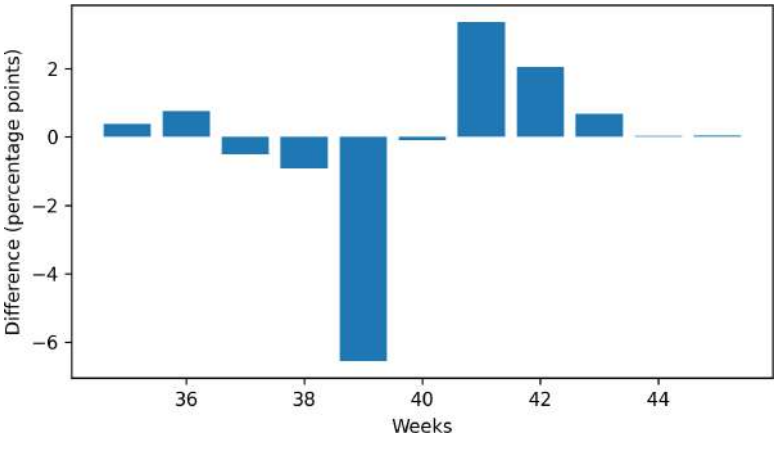
So we can compute the differences in the probabilities like this:

```
diffs = first_pmf(weeks) - other_pmf(weeks)
diffs
```

```
array([0.00392854,  0.00750796, -0.00503126, -0.00911366, -0.06562817,
       -0.00103276,  0.03363629,  0.02047689,  0.00683169,  0.00035443,
       0.00051485])
```

Here's what they look like, multiplied by 100 to express the differences in percentage points:

```
plt.bar(weeks, diffs * 100)
decorate(xlabel="Weeks", ylabel="Difference (percentage points)")
```



This figure makes the pattern clearer: first babies are less likely to be born in week 39, and somewhat more likely to be born in weeks 41 and 42.

When we see a pattern like this in a sample, we can't be sure it also holds in the population—and we don't know whether we would see it in another sample from the same population. We'll revisit this question in Chapter 9.

Glossary

There are not as many new terms in this chapter as in the previous chapters:

normalized
: A set of probabilities are normalized if they add up to 1.

probability mass function (PMF)
: A function that represents a distribution by mapping each quantity to its probability.

Exercises

For the exercises in this chapter, we'll use the NSFG respondent file, which contains one row for each respondent. Instructions for downloading the data are in the notebook for this chapter.

The `nsfg.py` module provides a function that reads the respondent file and returns a `DataFrame`:

```
from nsfg import read_fem_resp

resp = read_fem_resp()
resp.shape
```

```
(7643, 3092)
```

This `DataFrame` contains 7,643 rows and 3,092 columns.

Exercise 3.1

Select the column `numbabes`, which records the "number of babies born alive" to each respondent. Make a `FreqTab` object and display the frequencies of the values in this column. Check that they are consistent with the frequencies in the code book. Are there any special values that should be replaced with `NaN`?

Then make a `Pmf` object and plot it as a bar graph. Is the distribution symmetric, skewed to the left, or skewed to the right?

Exercise 3.2

In the same way that the mean identifies a central point in a distribution, and variance quantifies its spread, there is another statistic, called **skewness**, that indicates whether a distribution is skewed to the left or right.

Given a sample, we can compute the skewness by computing the sum of the cubed deviations and dividing by the standard deviation cubed. For example, here's how we compute the skewness of `numbabes`:

```
numbabes = resp["numbabes"].replace(97, np.nan)
```

```
deviations = numbabes - numbabes.mean()
skewness = np.mean(deviations**3) / numbabes.std(ddof=0) ** 3
skewness
```

```
1.7018914266755958
```

A positive value indicates that a distribution is skewed to the right, and a negative value indicates that it is skewed to the left.

If you are given a `Pmf`, rather than a sequence of values, you can compute skewness like this:

1. Compute the deviation of each quantity in the `Pmf` from the mean.
2. Cube the deviations, multiply by the probabilities in the `Pmf`, and add up the products.
3. Divide the sum by the standard deviation cubed.

Write a function called `pmf_skewness` that takes a `Pmf` object and returns its skewness. Use your function and the `Pmf` of `numbabes` to compute skewness, and confirm you get the result we computed above.

Exercise 3.3

Something like the class size paradox appears if you survey children and ask how many children are in their family. Families with many children are more likely to appear in your sample, and families with no children have no chance to be in the sample at all.

From `resp`, select `numkdhh`, which records the number of children under 18 in each respondent's household. Make a `Pmf` of the values in this column.

Use the `bias` function to compute the distribution we would see if we surveyed the children and asked them how many children under 18 (including themselves) are in their household.

Plot the actual and biased distributions, and compute their means.

CHAPTER 4

Cumulative Distribution Functions

Frequency tables and PMFs are the most familiar ways to represent distributions, but as we'll see in this chapter, they have limitations. An alternative is the cumulative distribution function (CDF), which is useful for computing percentiles, and especially useful for comparing distributions.

Also in this chapter, we'll compute percentile-based statistics to quantify the location, spread, and skewness of a distribution.

Percentiles and Percentile Ranks

If you have taken a standardized test, you probably got your results in the form of a raw score and a **percentile rank**. In this context, the percentile rank is the percentage of people who got the same score as you or lower. So if you are "in the 90th percentile," you did as well as or better than 90% of the people who took the exam.

To understand percentiles and percentile ranks, let's consider an example based on running speeds. Some years ago I ran the James Joyce Ramble, which is a 10 kilometer road race in Massachusetts. After the race, I downloaded the results to see how my time compared to other runners.

Instructions for downloading the data are in the notebook for this chapter. The `relay.py` module provides a function that reads the results and returns a Pandas `DataFrame`:

```
from relay import read_results

results = read_results()
results.head()
```

	Place	Div/Tot	Division	Guntime	Nettime	Min/Mile	MPH
0	1	1/362	M2039	30:43	30:42	4:57	12.121212
1	2	2/362	M2039	31:36	31:36	5:06	11.764706
2	3	3/362	M2039	31:42	31:42	5:07	11.726384
3	4	4/362	M2039	32:28	32:27	5:14	11.464968
4	5	5/362	M2039	32:52	32:52	5:18	11.320755

`results` contains one row for each of 1,633 runners who finished the race. The column we'll use to quantify performance is `MPH`, which contains each runner's average speed in miles per hour. We'll select this column and use `values` to extract the speeds as a NumPy array:

```
speeds = results["MPH"].values
```

I finished in 42:44, so we can find my row like this:

```
my_result = results.query("Nettime == '42:44'")
my_result
```

	Place	Div/Tot	Division	Guntime	Nettime	Min/Mile	MPH
96	97	26/256	M4049	42:48	42:44	6:53	8.716707

The index of my row is 96, so we can extract my speed like this:

```
my_speed = speeds[96]
```

We can use `sum` to count the number of runners at my speed or slower:

```
(speeds <= my_speed).sum()
```

```
1537
```

And we can use `mean` to compute the percentage of runners at my speed or slower:

```
(speeds <= my_speed).mean() * 100
```

```
94.12124923453766
```

The result is my percentile rank in the field, which was about 94%.

More generally, the following function computes the percentile rank of a particular value in a sequence of values:

```
def percentile_rank(x, seq):
    """Percentile rank of x.

    x: value
    seq: sequence of values

    returns: percentile rank 0-100
    """
    return (seq <= x).mean() * 100
```

In `results`, the `Division` column indicates the division each runner was in, identified by gender and age range—for example, I was in the `M4049` division, which includes male runners aged 40 to 49. We can use the `query` method to select the rows for people in my division and extract their speeds:

```
my_division = results.query("Division == 'M4049'")
my_division_speeds = my_division["MPH"].values
```

Now we can use `percentile_rank` to compute my percentile rank in my division:

```
percentile_rank(my_speed, my_division_speeds)
```

```
90.234375
```

Going in the other direction, if we are given a percentile rank, the following function finds the corresponding value in a sequence:

```
def percentile(p, seq):
     n = len(seq)
     i = (1 - p / 100) * (n + 1)
     return seq[round(i)]
```

`n` is the number of elements in the sequence; `i` is the index of the element with the given percentile rank. When we look up a percentile rank, the corresponding value is called a **percentile**:

```
percentile(90, my_division_speeds)
```

```
8.591885441527447
```

In my division, the 90th percentile was about 8.6 mph.

Now, some years after I ran that race, I am in the M5059 division. So let's see how fast I would have to run to have the same percentile rank in my new division. We can answer that question by converting my percentile rank in the M4049 division, which is about 90.2%, to a speed in the M5059 division:

```
next_division = results.query("Division == 'M5059'")
next_division_speeds = next_division["MPH"].values

percentile(90.2, next_division_speeds)
```

```
8.017817371937639
```

The person in the M5059 division with the same percentile rank as me ran just over 8 mph. We can use `query` to find him:

```
next_division.query("MPH > 8.01").tail(1)
```

	Place	Div/Tot	Division	Guntime	Nettime	Min/Mile	MPH
222	223	18/171	M5059	46:30	46:25	7:29	8.017817

He finished in 46:25 and came in 18th out of 171 people in his division.

With this introduction to percentile ranks and percentiles, we are ready for cumulative distribution functions.

CDFs

A **cumulative distribution function**, or CDF, is another way to describe the distribution of a set of values, along with a frequency table or PMF. Given a value `x`, the CDF computes the fraction of values less than or equal to `x`. As an example, we'll start with a short sequence:

```
t = [1, 2, 2, 3, 5]
```

One way to compute a CDF is to start with a PMF. Here is a `Pmf` object that represents the distribution of values in `t`:

```
from empiricaldist import Pmf

pmf = Pmf.from_seq(t)
pmf
```

	probs
1	0.2
2	0.4
3	0.2
5	0.2

As we saw in the previous chapter, we can use the bracket operator to look up a value in a `Pmf`:

```
pmf[2]
```

```
0.4
```

The result is the proportion of values in the sequence equal to the given value. In this example, two out of five values are equal to 2, so the result is 0.4. We can also think of this proportion as the probability that a randomly chosen value from the sequence equals 2.

`Pmf` has a `make_cdf` method that computes the cumulative sum of the probabilities in the `Pmf`:

```
cdf = pmf.make_cdf()
cdf
```

	probs
1	0.2
2	0.6
3	0.8
5	1.0

The result is a `Cdf` object, which is a kind of Pandas `Series`. We can use the bracket operator to look up a value:

```
cdf[2]
```

```
0.6000000000000001
```

The result is the proportion of values in the sequence less than or equal to the given value. In this example, three out of five values in the sequence are less than or equal to 2, so the result is 0.6. We can also think of this proportion as the probability that a randomly chosen value from the sequence is less than or equal to 2.

We can use parentheses to call the `Cdf` object like a function:

```
cdf(3)
```

```
array(0.8)
```

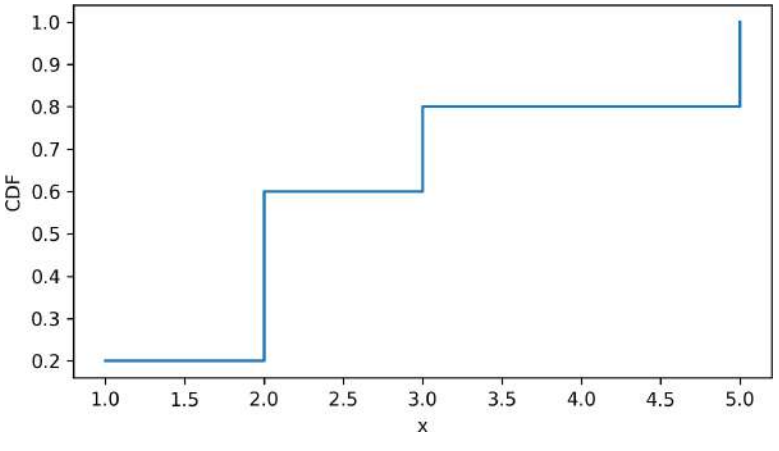
The cumulative distribution function is defined for all numbers, not just the ones that appear in the sequence:

```
cdf(4)
```

```
array(0.8)
```

To visualize the `Cdf`, we can use the `step` method, which plots the `Cdf` as a step function:

```
cdf.step()
decorate(xlabel="x", ylabel="CDF")
```

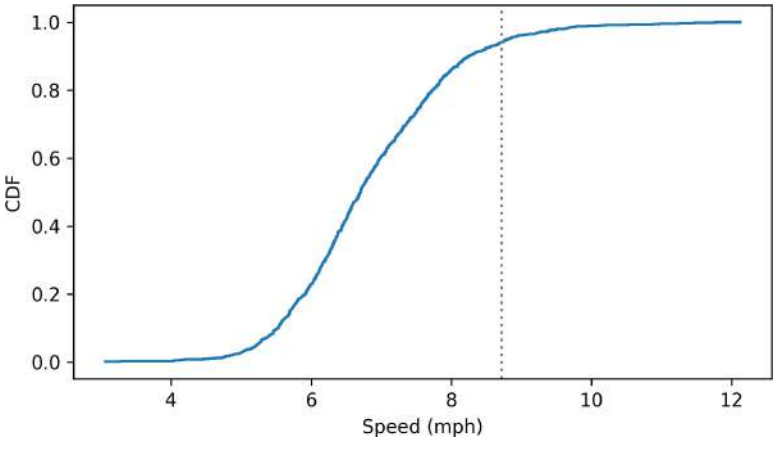


As a second example, let's make a `Cdf` that represents the distribution of running speeds from the previous section. The `Cdf` class provides a `from_seq` function we can use to create a `Cdf` object from a sequence:

```
from empiricaldist import Cdf

cdf_speeds = Cdf.from_seq(speeds)
```

And here's what it looks like—the vertical line is at my speed:

```
cdf_speeds.step()
plt.axvline(my_speed, ls=":", color="gray")
decorate(xlabel="Speed (mph)", ylabel="CDF")
```



If we look up my speed, the result is the fraction of runners at my speed or slower. If we multiply by 100, we get my percentile rank:

```
cdf_speeds(my_speed) * 100
```

```
94.12124923453766
```

So that's one way to think about the `Cdf`—given a value, it computes something like a percentile rank, except that it's a proportion between 0 and 1 rather than a percentage between 0 and 100.

`Cdf` provides an `inverse` method that computes the inverse of the cumulative distribution function—given a proportion between 0 and 1, it finds the corresponding value.

For example, if someone says they ran as fast or faster than 50% of the field, we can find their speed like this:

```
cdf_speeds.inverse(0.5)
```

```
array(6.70391061)
```

If you have a proportion and you use the inverse CDF to find the corresponding value, the result is called a **quantile**—so the inverse CDF is sometimes called the quantile function.

If you have a quantile and you use the CDF to find the corresponding proportion, the result doesn't really have a name, strangely. To be consistent with percentile and percentile rank, it could be called a "quantile rank," but as far as I can tell, no one calls it that. Most often, it is just called a "cumulative probability."

Comparing CDFs

CDFs are especially useful for comparing distributions. As an example, let's compare the distribution of birth weights for first babies and others. We'll load the NSFG dataset again, and divide it into three `DataFrames`: all live births, first babies, and others:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

From `firsts` and `others` we'll select total birth weights in pounds, using `dropna` to remove values that are `nan`:

```
first_weights = firsts["totalwgt_lb"].dropna()
first_weights.mean()
```

```
7.201094430437772
```

```
other_weights = others["totalwgt_lb"].dropna()
other_weights.mean()
```

```
7.325855614973262
```

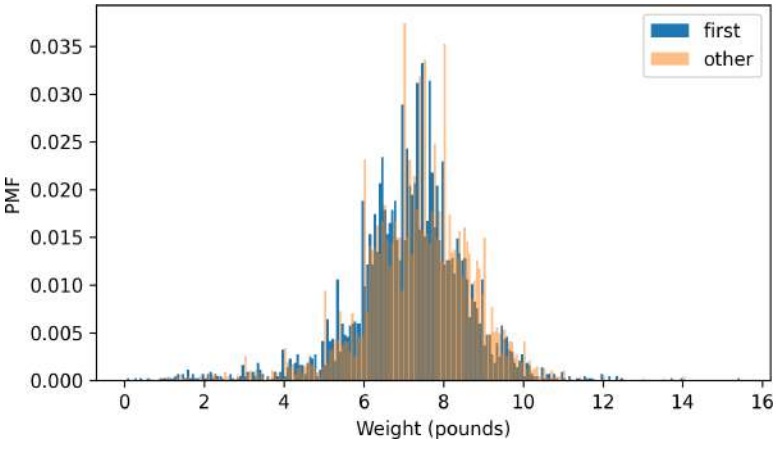
It looks like first babies are a little lighter on average. But there are several ways a difference like that could happen—for example, there might be a small number of first babies who are especially light, or a small number of other babies who are especially heavy. In those cases, the distributions would have different shapes. As another possibility, the distributions might have the same shape, but different locations.

To compare the distributions, we can try plotting the PMFs:

```
from empiricaldist import Pmf

first_pmf = Pmf.from_seq(first_weights, name="first")
other_pmf = Pmf.from_seq(other_weights, name="other")
```

But as we can see here, it doesn't work very well:

```
from thinkstats import two_bar_plots

two_bar_plots(first_pmf, other_pmf, width=0.06)
decorate(xlabel="Weight (pounds)", ylabel="PMF")
```



I adjusted the width and transparency of the bars to show the distributions as clearly as possible, but it is hard to compare them. There are many peaks and valleys, and some apparent differences, but it is hard to tell which of these features are meaningful. Also, it is hard to see overall patterns; for example, it is not visually apparent which distribution has the higher mean.

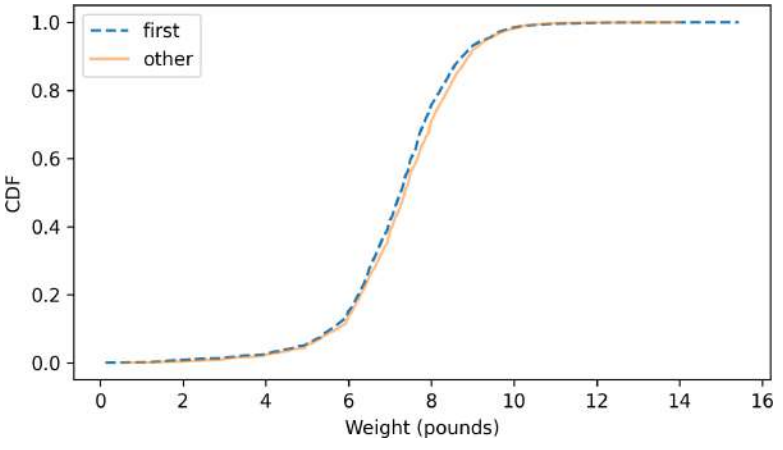
These problems can be mitigated by binning the data—that is, dividing the range of quantities into nonoverlapping intervals and counting the number of quantities in each bin. Binning can be useful, but it is tricky to get the size of the bins right. If they are big enough to smooth out noise, they might also smooth out useful information.

A good alternative is to plot the CDFs:

```
first_cdf = first_pmf.make_cdf()
other_cdf = other_pmf.make_cdf()
```

Here's what they look like:

```
first_cdf.plot(ls="--")
other_cdf.plot(alpha=0.5)
decorate(xlabel="Weight (pounds)", ylabel="CDF")
```



This figure makes the shape of the distributions, and the differences between them, much clearer. The curve for first babies is consistently to the left of the curve for others, which indicates that first babies are slightly lighter throughout the distribution—with a larger discrepancy above the midpoint.

Percentile-Based Statistics

In Chapter 3 we computed the arithmetic mean, which identifies a central point in a distribution, and the standard deviation, which quantifies how spread out the distribution is. And in a previous exercise we computed skewness, which indicates whether a distribution is skewed left or right. One drawback of all of these statistics is that they are sensitive to outliers. A single extreme value in a dataset can have a large effect on mean, standard deviation, and skewness.

An alternative is to use statistics that are based on percentiles of the distribution, which tend to be more **robust**, which means that they are less sensitive to outliers. To demonstrate, let's load the NSFG data again without doing any data cleaning:

```
from nsfg import read_stata

dct_file = "2002FemPreg.dct"
dat_file = "2002FemPreg.dat.gz"

preg = read_stata(dct_file, dat_file)
```

Recall that birth weight is recorded in two columns, one for the pounds and one for the ounces:

```
birthwgt_lb = preg["birthwgt_lb"]
birthwgt_oz = preg["birthwgt_oz"]
```

If we make a `Hist` object with the values from `birthwgt_oz`, we can see that they include the special values 97, 98, and 99, which indicate missing data:

```
from empiricaldist import Hist

Hist.from_seq(birthwgt_oz).tail(5)
```

freqs	birthwgt_oz
14.0	475
15.0	378
97.0	1
98.0	1
99.0	46

The `birthwgt_lb` column includes the same special values; it also includes the value 51, which has to be a mistake:

```
Hist.from_seq(birthwgt_lb).tail(5)
```

freqs	birthwgt_lb
15.0	1
51.0	1
97.0	1
98.0	1
99.0	57

Now let's imagine two scenarios. In one scenario, we clean these variables by replacing missing and invalid values with `nan`, and then compute total weight in pounds. Dividing `birthwgt_oz_clean` by 16 converts it to pounds in decimal:

```
birthwgt_lb_clean = birthwgt_lb.replace([51, 97, 98, 99], np.nan)
birthwgt_oz_clean = birthwgt_oz.replace([97, 98, 99], np.nan)

total_weight_clean = birthwgt_lb_clean + birthwgt_oz_clean / 16
```

In the other scenario, we neglect to clean the data and accidentally compute the total weight with these bogus values:

```
total_weight_bogus = birthwgt_lb + birthwgt_oz / 16
```

The bogus dataset contains only 49 bogus values, which is about 0.5% of the data:

```
count1, count2 = total_weight_bogus.count(), total_weight_clean.count()
diff = count1 - count2

diff, diff / count2 * 100
```

```
(49, 0.5421553441026776)
```

Now let's compute the mean of the data in both scenarios:

```
mean1, mean2 = total_weight_bogus.mean(), total_weight_clean.mean()
mean1, mean2
```

```
(7.319680587652691, 7.265628457623368)
```

The bogus values have a moderate effect on the mean. If we take the mean of the cleaned data to be correct, the mean of the bogus data is off by less than 1%:

```
(mean1 - mean2) / mean2 * 100
```

```
0.74394294099376
```

An error like that might go undetected—but now let's see what happens to the standard deviations:

```
std1, std2 = total_weight_bogus.std(), total_weight_clean.std()
std1, std2
```

```
(2.096001779161835, 1.4082934455690173)
```

```
(std1 - std2) / std2 * 100
```

```
48.83274403900607
```

The standard deviation of the bogus data is off by almost 50%, so that's more noticeable. Finally, here's the skewness of the two datasets:

```
def skewness(seq):
    deviations = seq - seq.mean()
    return np.mean(deviations**3) / seq.std(ddof=0) ** 3
```

```
skew1, skew2 = skewness(total_weight_bogus), skewness(total_weight_clean)
skew1, skew2
```

```
(22.251846195422484, -0.5895062687577697)
```

The skewness of the bogus dataset is off by a factor of almost 40, and it has the wrong sign! With the outliers added to the data, the distribution is strongly skewed to the right, as indicated by large positive skewness. But the distribution of the valid data is slightly skewed to the left, as indicated by small negative skewness.

These results show that a small number of outliers have a moderate effect on the mean, a strong effect on the standard deviation, and a disastrous effect on skewness.

An alternative is to use statistics based on percentiles. Specifically:

- The median, which is the 50th percentile, identifies a central point in a distribution, like the mean.
- The interquartile range, which is the difference between the 25th and 75th percentiles, quantifies the spread of the distribution, like the standard deviation.
- The quartile skewness uses the quartiles of the distribution (25th, 50th, and 75th percentiles) to quantify the skewness.

The `Cdf` object provides an efficient way to compute these percentile-based statistics. To demonstrate, let's make a `Cdf` object from the bogus and clean datasets:

```
cdf_total_weight_bogus = Cdf.from_seq(total_weight_bogus)
cdf_total_weight_clean = Cdf.from_seq(total_weight_clean)
```

The following function takes a `Cdf` and uses its `inverse` method to compute the 50th percentile, which is the median (at least, it is one way to define the median of a dataset):

```
def median(cdf):
    m = cdf.inverse(0.5)
    return m
```

Now we can compute the median of both datasets:

```
median(cdf_total_weight_bogus), median(cdf_total_weight_clean)
```

```
(array(7.375), array(7.375))
```

The results are identical, so in this case, the outliers have no effect on the median at all. In general, outliers have a smaller effect on the median than on the mean.

The **interquartile range (IQR)** is the difference between the 75th and 25th percentiles. The following function takes a `Cdf` and returns the IQR:

```
def iqr(cdf):
    low, high = cdf.inverse([0.25, 0.75])
    return high - low
```

And here are the interquartile ranges of the two datasets:

```
iqr(cdf_total_weight_bogus), iqr(cdf_total_weight_clean)
```

```
(1.625, 1.625)
```

In general, outliers have less effect on the IQR than on the standard deviation—in this case they have no effect at all.

Finally, here's a function that computes quartile skewness, which depends on three statistics:

- The median
- The midpoint of 25th and 75th percentiles
- The semi-IQR, which is half of the IQR

```
def quartile_skewness(cdf):
    low, median, high = cdf.inverse([0.25, 0.5, 0.75])
    midpoint = (high + low) / 2
    semi_iqr = (high - low) / 2
    return (midpoint - median) / semi_iqr
```

And here's the quartile skewness for the two datasets:

```
qskew1 = quartile_skewness(cdf_total_weight_bogus)
qskew2 = quartile_skewness(cdf_total_weight_clean)
qskew1, qskew2
```

```
(-0.07692307692307693, -0.07692307692307693)
```

The small number of outliers in these examples has no effect on the quartile skewness. These examples show that percentile-based statistics are less sensitive to outliers and errors in the data.

Random Numbers

`Cdf` objects provide an efficient way to generate random numbers from a distribution. First we generate random numbers from a uniform distribution between 0 and 1. Then we evaluate the inverse CDF at those points. The following function implements this algorithm:

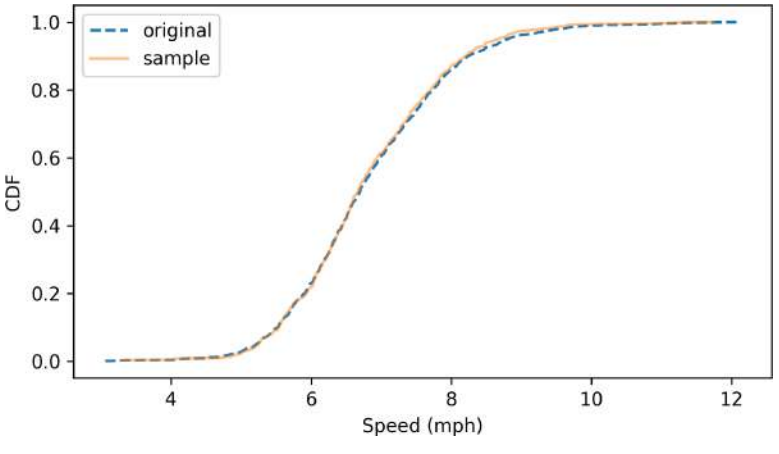
```
def sample_from_cdf(cdf, n):
    ps = np.random.random(size=n)
    return cdf.inverse(ps)
```

To demonstrate, let's generate a random sample of running speeds:

```
sample = sample_from_cdf(cdf_speeds, 1001)
```

To confirm that it worked, we can compare the CDFs of the sample and the original dataset:

```
cdf_sample = Cdf.from_seq(sample)

cdf_speeds.plot(label="original", ls="--")
cdf_sample.plot(label="sample", alpha=0.5)

decorate(xlabel="Speed (mph)", ylabel="CDF")
```



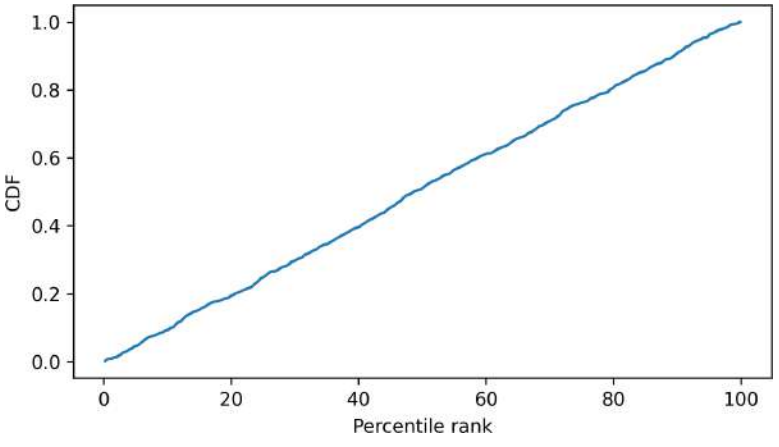
The sample follows the distribution of the original data. To understand how this algorithm works, consider this question: Suppose we choose a random sample from the population of running speeds and look up the percentile ranks of the speeds in the sample. Now suppose we compute the CDF of the percentile ranks. What do you think it will look like?

Let's find out. Here are the percentile ranks for the sample we generated:

```
percentile_ranks = cdf_speeds(sample) * 100
```

And here is the CDF of the percentile ranks:

```
cdf_percentile_rank = Cdf.from_seq(percentile_ranks)
cdf_percentile_rank.plot()

decorate(xlabel="Percentile rank", ylabel="CDF")
```



The CDF of the percentile ranks is close to a straight line between 0 and 1. And that makes sense, because in any distribution, the proportion with percentile rank less than 50% is 0.5; the proportion with percentile rank less than 90% is 0.9, and so on.

`Cdf` provides a `sample` method that uses this algorithm, so we could also generate a sample like this:

```
sample = cdf_speeds.sample(1001)
```

Glossary

percentile rank
: The percentage of values in a distribution that are less than or equal to a given quantity.

percentile
: The value in a distribution associated with a given percentile rank.

cumulative distribution function (CDF)
: A function that maps a value to the proportion of the distribution less than or equal to that value.

quantile
: The value in a distribution that is greater than or equal to a given proportion of values.

robust
: A statistic is robust if it is less affected by extreme values or outliers.

interquartile range (IQR)
: The difference between the 75th and 25th percentiles, used to measure the spread of a distribution.

Exercises

Exercise 4.1

How much did you weigh at birth? If you don't know, call your mother or someone else who knows. And if no one knows, you can use my birth weight, 8.5 pounds, for this exercise.

Using the NSFG data (all live births), compute the distribution of birth weights and use it to find your percentile rank. If you were a first baby, find your percentile rank in the distribution for first babies. Otherwise use the distribution for others. If you are in the 90th percentile or higher, call your mother back and apologize:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

Exercise 4.2

For live births in the NSFG dataset, the column `babysex` indicates whether the baby was male or female. We can use `query` to select the rows for male and female babies:

```
male = live.query("babysex == 1")
female = live.query("babysex == 2")
len(male), len(female)
```

```
(4641, 4500)
```

Make `Cdf` objects that represent the distribution of birth weights for male and female babies. Plot the two CDFs. What are the differences in the shape and location of the distributions?

If a male baby weighs 8.5 pounds, what is his percentile rank? What is the weight of a female baby with the same percentile rank?

Exercise 4.3

From the NSFG dataset pregnancy data, select the `agepreg` column and make a `Cdf` to represent the distribution of age at conception for each pregnancy. Use the CDF to compute the percentage of ages less than or equal to 20, and the percentage less than or equal to 30. Use those results to compute the percentage between 20 and 30:

```
from nsfg import read_fem_preg

preg = read_fem_preg()
```

Exercise 4.4

Here are the running speeds of the people who finished the James Joyce Ramble, described earlier in this chapter:

```
speeds = results["MPH"].values
```

Make a `Cdf` that represents the distribution of these speeds, and use it to compute the median, IQR, and quartile skewness. Does the distribution skew to the left or right?

Exercise 4.5

The numbers generated by `np.random.random` are supposed to be uniform between 0 and 1, which means that the CDF of a sample should be a straight line. Let's see if that's true. Here's a sample of 1,001 numbers. Plot the CDF of this sample. Does it look like a straight line?

```
t = np.random.random(1001)
```

CHAPTER 5

Modeling Distributions

The distributions we have used so far are called empirical distributions because they are based on empirical observations—in other words, data. Many datasets we see in the real world can be closely approximated by a theoretical distribution, which is usually based on a simple mathematical function. This chapter presents some of these theoretical distributions and datasets they can be used to model.

As examples, we'll see that:

- In a skeet shooting competition, the number of hits and misses is well modeled by a binomial distribution.
- In games like hockey and soccer (football), the number of goals in a game follows a Poisson distribution, and the time between goals follows an exponential distribution.
- Birth weights follow a normal distribution, also called a Gaussian, and adult weights follow a lognormal distribution.

If you are not familiar with these distributions—or these sports—I will explain what you need to know. For each example, we'll start with a simulation based on a simple model, and show that the simulation results follow a theoretical distribution. Then we'll see how well real data agrees with the model.

The Binomial Distribution

As a first example, we'll consider the sport of skeet shooting, in which competitors use shotguns to shoot clay disks that are thrown into the air. In international competition, including the Olympics, there are five rounds with 25 targets per round, with additional rounds as needed to determine a winner.

As a model of a skeet shooting competition, suppose that every participant has the same probability, `p`, of hitting every target. Of course, this model is a simplification—in reality, some competitors have a higher probability than others, and even for a single competitor, it might vary from one attempt to the next. But even if it is not realistic, this model makes some surprisingly accurate predictions, as we'll see.

To simulate the model, I'll use the following function, which takes the number of targets, `n`, and the probability of hitting each one, `p`, and returns a sequence of 1s and 0s to indicate hits and misses:

```
def flip(n, p):
    choices = [1, 0]
    probs = [p, 1 - p]
    return np.random.choice(choices, n, p=probs)
```

Here's an example that simulates a round of 25 targets where the probability of hitting each one is 90%:

```
flip(25, 0.9)
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
       1, 1, 1])
```

If we generate a longer sequence and compute the `Pmf` of the results, we can confirm that the proportions of 1s and 0s are correct, at least approximately:

```
from empiricaldist import Pmf

seq = flip(1000, 0.9)
pmf = Pmf.from_seq(seq)
pmf
```

	probs
0	0.101
1	0.899

Now we can use `flip` to simulate a round of skeet shooting and return the number of hits:

```
def simulate_round(n, p):
    seq = flip(n, p)
    return seq.sum()
```

In a large competition, suppose 200 competitors shoot 5 rounds each, all with the same probability of hitting the target, `p=0.9`. We can simulate a competition like that by calling `simulate_round` 1,000 times:

```
n = 25
p = 0.9
results_sim = [simulate_round(n, p) for i in range(1000)]
```

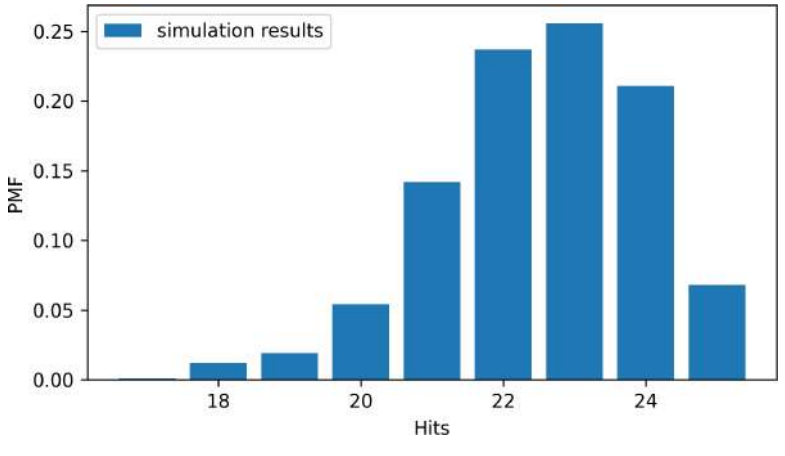
The average score is close to `22.5`, which is the product of `n` and `p`:

```
np.mean(results_sim), n * p
```

```
(22.522, 22.5)
```

Here's what the distribution of the results looks like:

```
from empiricaldist import Pmf

pmf_sim = Pmf.from_seq(results_sim, name="simulation results")

pmf_sim.bar()
decorate(xlabel="Hits", ylabel="PMF")
```



The peak is near the mean, and the distribution is skewed to the left.

Instead of running a simulation, we could have predicted this distribution. Mathematically, the distribution of these outcomes follows a **binomial distribution**, which has a PMF that is easy to compute:

```
from scipy.special import comb


def binomial_pmf(k, n, p):
   return comb(n, k) * (p**k) * ((1 - p) ** (n - k))
```

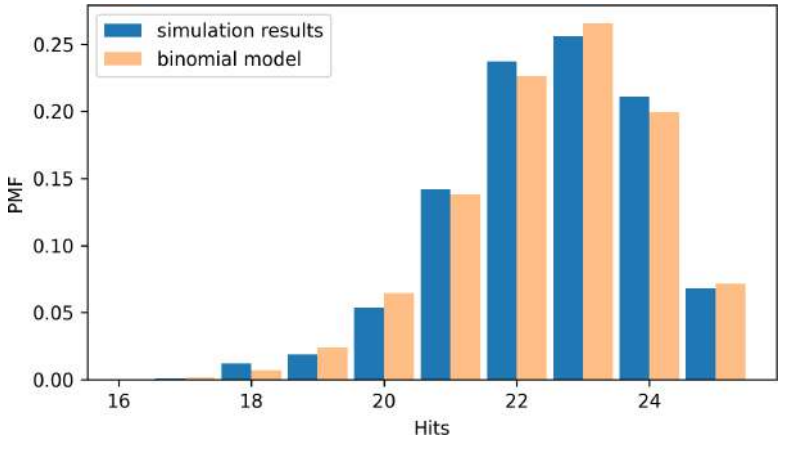
SciPy provides the `comb` function, which computes the number of combinations of `n` things taken `k` at a time, often pronounced "n choose k."

`binomial_pmf` computes the probability of getting `k` hits out of `n` attempts, given `p`. If we call this function with a range of `k` values, we can make a `Pmf` that represents the distribution of the outcomes:

```
ks = np.arange(16, n + 1)
ps = binomial_pmf(ks, n, p)
pmf_binom = Pmf(ps, ks, name="binomial model")
```

And here's what it looks like compared to the simulation results:

```
from thinkstats import two_bar_plots

two_bar_plots(pmf_sim, pmf_binom)
decorate(xlabel="Hits", ylabel="PMF")
```



They are similar, with small differences because of random variation in the simulation results. This agreement should not be surprising, because the simulation and the model are based on the same assumptions—particularly the assumption that every attempt has the same probability of success. A stronger test of a model is how it compares to real data.

From the Wikipedia page for the men's skeet shooting competition at the 2020 Summer Olympics, we can extract a table that shows the results for the qualification rounds. Instructions for downloading the data are in the notebook for this chapter.

```
filename = "Shooting_at_the_2020_Summer_Olympics_Mens_skeet"
```

```
tables = pd.read_html(filename)
table = tables[6]
table.head()
```

	Rank	Athlete	Country	1	2	3	4	5	Total[3]	Shoot-off	Notes
0	1	Éric Delaunay	France	25	25	25	24	25	124	+6	Q, OR
1	2	Tammaro Cassandro	Italy	24	25	25	25	25	124	+5	Q, OR
2	3	Eetu Kallioinen	Finland	25	25	24	25	24	123	NaN	Q
3	4	Vincent Hancock	United States	25	25	25	25	22	122	+8	Q
4	5	Abdullah Al-Rashidi	Kuwait	25	25	24	25	23	122	+7	Q

The table has one row for each competitor, with one column for each of five rounds. We'll select the columns that contain these results and use the NumPy function `flatten` to put them into a single array:

```
columns = ["1", "2", "3", "4", "5"]
results = table[columns].values.flatten()
```

With 30 competitors, we have results from 150 rounds of 25 shots each, with 3,750 hits out of a total of 3,575 attempts:

```
total_shots = 25 * len(results)
total_hits = results.sum()
n, total_shots, total_hits
```

```
(25, 3750, 3575)
```

So the overall success rate is 95.3%:

```
p = total_hits / total_shots
p
```

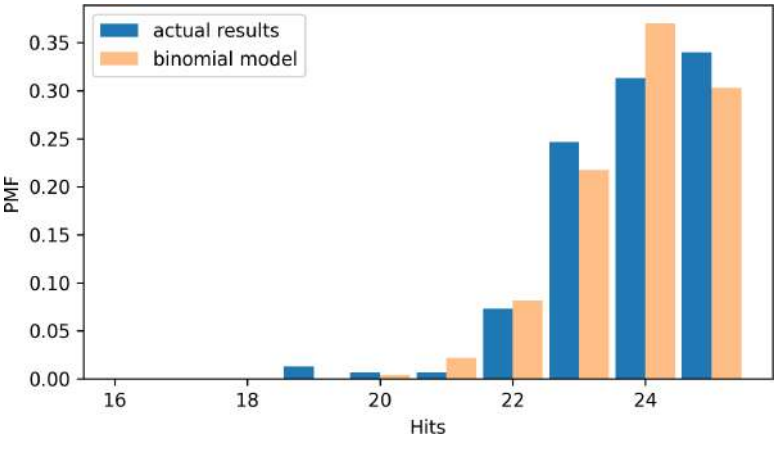
```
0.9533333333333334
```

Now let's compute a `Pmf` that represents the binomial distribution with `n=25` and the value of `p` we just computed:

```
ps = binomial_pmf(ks, n, p)
pmf_binom = Pmf(ps, ks, name="binomial model")
```

And we can compare that to the `Pmf` of the actual results:

```
pmf_results = Pmf.from_seq(results, name="actual results")

two_bar_plots(pmf_results, pmf_binom)
decorate(xlabel="Hits", ylabel="PMF")
```



The binomial model is a good fit for the distribution of the data—even though it makes the unrealistic assumption that all competitors have the same, unchanging capability.

The Poisson Distribution

As another example where the outcomes of sports events follow predictable patterns, let's look at the number of goals scored in ice hockey games.

We'll start by simulating a 60-minute game, which is 3,600 seconds, assuming that the teams score a total of 6 goals per game, on average, and that the goal-scoring probability, `p`, is the same during any second:

```
n = 3600
m = 6
p = m / 3600
p
```

```
0.0016666666666666668
```

Now we can use the following function to simulate `n` seconds and return the total number of goals scored:

```
def simulate_goals(n, p):
    return flip(n, p).sum()
```

If we simulate many games, we can confirm that the average number of goals per game is close to 6:

```
goals = [simulate_goals(n, p) for i in range(1001)]
np.mean(goals)
```

```
6.021978021978022
```

We could use the binomial distribution to model these results, but when `n` is large and `p` is small, the results are also well-modeled by a **Poisson distribution**, which is specified by a value usually denoted with the Greek letter λ, which is pronounced "lambda" and represented in code with the variable `lam` (`lambda` is not a legal variable name because it is a Python keyword). `lam` represents the goal-scoring rate, which is 6 goals per game in the example.

The PMF of the Poisson distribution is easy to compute—given `lam`, we can use the following function to compute the probability of seeing `k` goals in a game:

```
from scipy.special import factorial

def poisson_pmf(k, lam):
    return (lam**k) * np.exp(-lam) / factorial(k)
```

SciPy provides the `factorial` function, which computes the product of the integers from `1` to `k`.

If we call `poisson_pmf` with a range of `k` values, we can make a `Pmf` that represents the distribution of outcomes:

```
lam = 6
ks = np.arange(20)
ps = poisson_pmf(ks, lam)
pmf_poisson = Pmf(ps, ks, name="Poisson model")
```

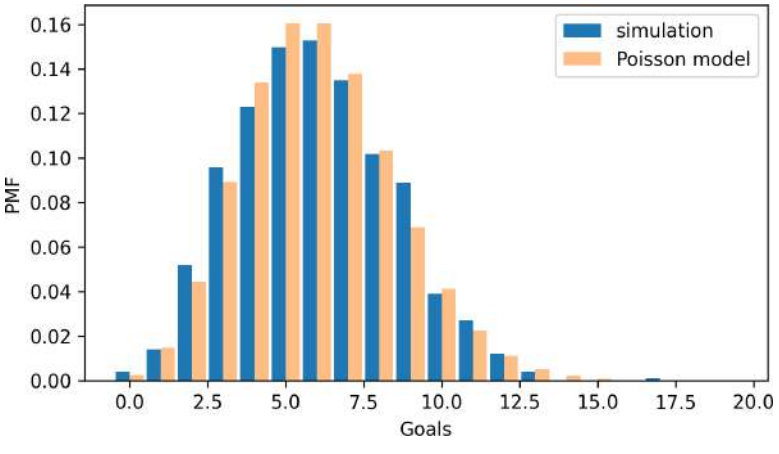
And confirm that the mean of the distribution is close to 6:

```
pmf_poisson.normalize()
pmf_poisson.mean()
```

```
5.999925498375129
```

We can compare the results from the simulation to the Poisson distribution with the same mean:

```
pmf_sim = Pmf.from_seq(goals, name="simulation")

two_bar_plots(pmf_sim, pmf_poisson)
decorate(xlabel="Goals", ylabel="PMF")
```



The distributions are similar except for small differences due to random variation. That should not be surprising, because the simulation and the Poisson model are based on the same assumption, that the probability of scoring a goal is the same during any second of the game. So a stronger test is to see how well the model fits real data.

From HockeyReference, I downloaded results of every game of the National Hockey League (NHL) 2023-2024 regular season (not including the playoffs). I extracted information about goals scored during 60 minutes of regulation play, not including overtime or tie-breaking shootouts. The results are in an HDF file with one key for each game, and a list of times, in seconds since the beginning of the game, when a goal was scored. Instructions for downloading the data are in the notebook for this chapter.

Here's how we read the keys from the file:

```
filename = "nhl_2023_2024.hdf"

with pd.HDFStore(filename, "r") as store:
    keys = store.keys()

len(keys), keys[0]
```

```
(1312, '/202310100PIT')
```

There were 1,312 games during the regular season. Each key contains the date of the game and a three-letter abbreviation for the home team. We can use `read_hdf` to look up a key and get the list of times when a goal was scored:

```
times = pd.read_hdf(filename, key=keys[0])
times
```

```
0     424
1    1916
2    2137
3    3005
4    3329
5    3513
dtype: int64
```

In the first game of the season, six goals were scored, the first after 424 seconds of play, the last after 3,513 seconds—with only 87 seconds left in the game.

The following loop reads the results for all games, counts the number of goals in each one, and stores the results in a list:

```
goals = []

for key in keys:
    times = pd.read_hdf(filename, key=key)
    n = len(times)
    goals.append(n)
```

The average number of goals per game is just over 6:

```
lam = np.mean(goals)
lam
```

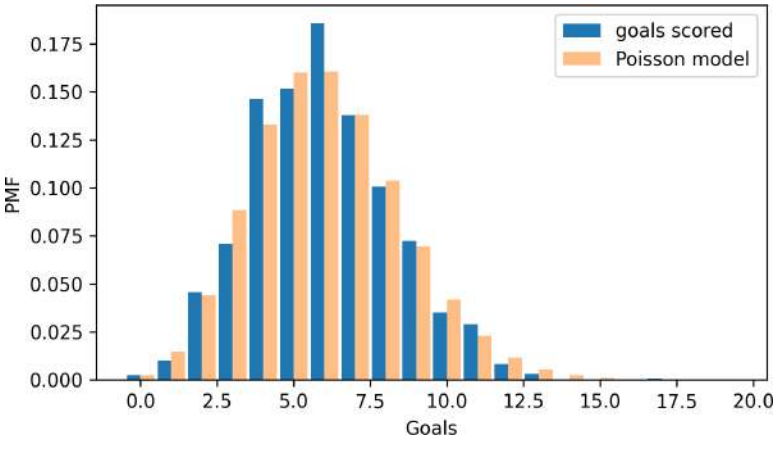
```
6.0182926829268295
```

We can use `poisson_pmf` to make a `Pmf` that represents a Poisson distribution with the same mean as the data:

```
ps = poisson_pmf(ks, lam)
pmf_poisson = Pmf(ps, ks, name="Poisson model")
```

And here's what it looks like compared to the PMF of the data:

```
pmf_goals = Pmf.from_seq(goals, name="goals scored")

two_bar_plots(pmf_goals, pmf_poisson)
decorate(xlabel="Goals", ylabel="PMF")
```



The Poisson distribution fits the data well, which suggests that it is a good model of the goal-scoring process in hockey.

The Exponential Distribution

In the previous section, we simulated a simple model of a hockey game where a goal has the same probability of being scored during any second of the game. Under the same model, it turns out, the time until the first goal follows an **exponential distribution**.

To demonstrate, let's assume again that the teams score a total of six goals, on average, and compute the probability of a goal during each second:

```
n = 3600
m = 6
p = m / 3600
p
```

```
0.0016666666666666668
```

This function simulates `n` seconds and uses `argmax` to find the time of the first goal:

```
def simulate_first_goal(n, p):
    return flip(n, p).argmax()
```

This works because the result from `flip` is a sequence of 1s and 0s, so the maximum is almost always 1. If there is at least one goal in the sequence, `argmax` returns the index of the first. If there are no goals, it returns 0, but that happens seldom enough that we'll ignore it.

We'll use `simulate_first_goal` to simulate 1,001 games and make a list of the times until the first goal:

```
first_goal_times = [simulate_first_goal(n, p) for i in range(1001)]
mean = np.mean(first_goal_times)
mean
```

```
597.7902097902098
```

The average time until the first goal is close to 600 seconds, or 10 minutes. And that makes sense—if we expect 6 goals per 60-minute game, we expect one goal every 10 minutes, on average.

When `n` is large and `p` is small, we can show mathematically that the expected time until the first goal follows an exponential distribution.

Because the simulation generates many unique time values, we'll use CDFs to compare distributions, rather than PMFs. And the CDF of the exponential distribution is easy to compute:

```
def exponential_cdf(x, lam):
    return 1 - np.exp(-lam * x)
```

The parameter, `lam`, is the average number of events per unit of time—in this example it is goals per second. We can use the mean of the simulated results to compute `lam`:

```
lam = 1 / mean
lam
```

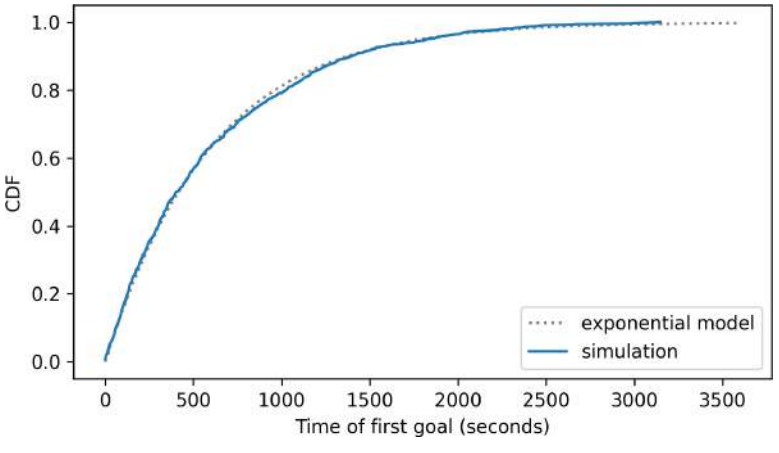
```
0.0016728276636563566
```

If we call this function with a range of time values, we can approximate the distribution of first goal times. The NumPy function `linspace` creates an array of equally-spaced values; in this example, it computes 201 values from 0 to 3,600, including both:

```
from empiricaldist import Cdf

ts = np.linspace(0, 3600, 201)
ps = exponential_cdf(ts, lam)
cdf_expo = Cdf(ps, ts, name="exponential model")
```

We can compare the simulation results to the exponential distribution we just computed:

```
cdf_sim = Cdf.from_seq(first_goal_times, name="simulation")

cdf_expo.plot(ls=":", color="gray")
cdf_sim.plot()

decorate(xlabel="Time of first goal (seconds)", ylabel="CDF")
```



The exponential model fits the results from the simulation very well—but a stronger test is to see how it does with real data.

The following loop reads the results for all games, gets the time of the first goal, and stores the result in a list. If no goals were scored, it adds `nan` to the list:

```
firsts = []

for key in keys:
    times = pd.read_hdf(filename, key=key)
    if len(times) > 0:
firsts.append(times[0])
    else:
firsts.append(np.nan)
```

To estimate the goal-scoring rate, we can use `nanmean`, which computes the mean of the times, ignoring `nan` values:

```
lam = 1 / np.nanmean(firsts)
lam
```

```
0.0015121567467720825
```

Now we can compute the CDF of an exponential distribution with the same goal-scoring rate as the data:

```
ps = exponential_cdf(ts, lam)
cdf_expo = Cdf(ps, ts, name="exponential model")
```

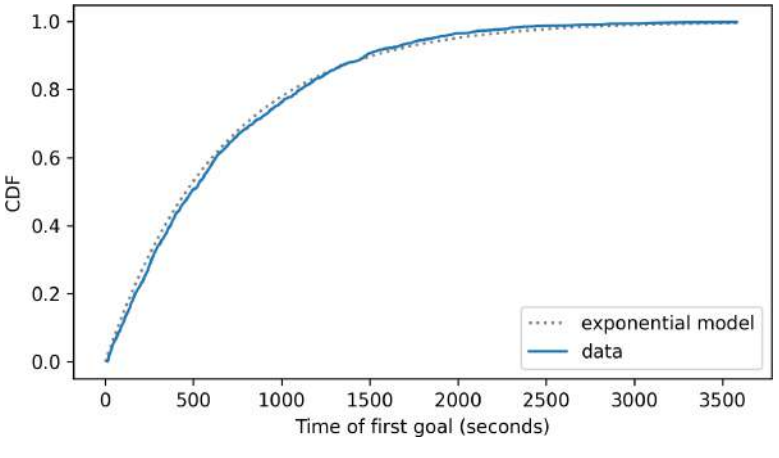
To compute the CDF of the data, we'll use the `dropna=False` argument, which includes `nan` values at the end:

```
cdf_firsts = Cdf.from_seq(firsts, name="data", dropna=False)
cdf_firsts.tail()
```

	probs
3286.0	0.996951
3581.0	0.997713
NaN	1.000000

This figure compares the exponential distribution to the distribution of the data:

```
cdf_expo.plot(ls=":", color="gray")
cdf_firsts.plot()

decorate(xlabel="Time of first goal (seconds)", ylabel="CDF")
```



The data deviate from the model in some places—it looks like there are fewer goals in the first 1,000 seconds than the model predicts. But still, the model fits the data well.

The underlying assumption of these models—the Poisson model of goals and the exponential model of times—is that a goal is equally likely during any second of a game. If you ask a hockey fan whether that's true, they would say no, and they would

be right—the real world violates assumptions like these in many ways. Nevertheless, theoretical distributions often fit real data remarkably well.

The Normal Distribution

Many things we measure in the real world follow a **normal distribution**, also known as a Gaussian distribution or a "bell curve." To see where these distributions come from, let's consider a model of the way giant pumpkins grow. Suppose that each day, a pumpkin gains one pound if the weather is bad, two pounds if the weather is fair, and three pounds if the weather is good. And suppose the weather each day is bad, fair, or good with the same probability.

We can use the following function to simulate this model for `n` days and return the total of the weight gains:

```
def simulate_growth(n):
    choices = [1, 2, 3]
    gains = np.random.choice(choices, n)
    return gains.sum()
```

NumPy's `random` module provides a `choice` function that generates an array of `n` random selections from a sequence of values, `choices` in this example.

Now suppose 1,001 people grow giant pumpkins in different places with different weather. If we simulate the growth process for 100 days, we get a list of 1,001 weights:

```
sim_weights = [simulate_growth(100) for i in range(1001)]
m, s = np.mean(sim_weights), np.std(sim_weights)
m, s
```

```
(199.37062937062936, 8.388630840376777)
```

The mean is close to 200 pounds and the standard deviation is about 8 pounds. To see whether the weights follow a normal distribution, we'll use the following function, which takes a sample and makes a `Cdf` that represents a normal distribution with the same mean and standard deviation as the sample, evaluated over the range from four standard deviations below the mean to four standard deviations above:

```
from scipy.stats import norm

def make_normal_model(data):
    m, s = np.mean(data), np.std(data)
    low, high = m - 4 * s, m + 4 * s
    qs = np.linspace(low, high, 201)
    ps = norm.cdf(qs, m, s)
    return Cdf(ps, qs, name="normal model")
```

Here's how we use it:

```
cdf_model = make_normal_model(sim_weights)
```

Now we can make a `Cdf` that represents the distribution of the simulation results:

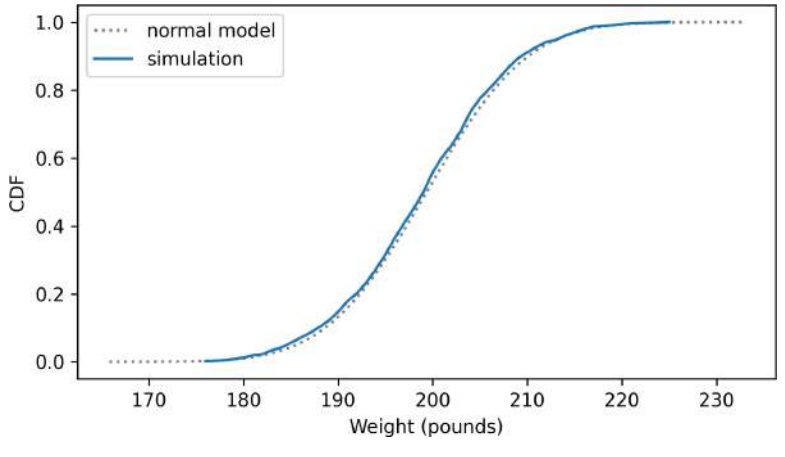
```
cdf_sim_weights = Cdf.from_seq(sim_weights, name="simulation")
```

We'll use the following function to compare the distributions. `cdf_model` and `cdf_data` are `Cdf` objects. `xlabel` is a string, and `options` is a dictionary of options that controls the way `cdf_data` is plotted:

```
def two_cdf_plots(cdf_model, cdf_data, xlabel="", **options):
    cdf_model.plot(ls=":", color="gray")
    cdf_data.plot(**options)
    decorate(xlabel=xlabel, ylabel="CDF")
```

And here are the results:

```
two_cdf_plots(cdf_model, cdf_sim_weights, xlabel="Weight (pounds)")
```



The normal model fits the distribution of the weights very well. In general, when we add up enough random factors, the sum tends to follow a normal distribution. That's a consequence of the Central Limit Theorem, which we'll come back to in Chapter 14.

But first let's see how well the normal distribution fits real data. As an example, we'll look at the distribution of birth weights in the National Survey of Family Growth (NSFG). We can use `read_fem_preg` to read the data, then select the `totalwgt_lb` column, which records birth weights in pounds:

```
import nsfg

preg = nsfg.read_fem_preg()
birth_weights = preg["totalwgt_lb"].dropna()
```

The average of the birth weights is about 7.27 pounds, and the standard deviation is 1.4 pounds, but as we've seen, there are some outliers in this dataset that are probably errors:

```
m, s = np.mean(birth_weights), np.std(birth_weights)
m, s
```

```
(7.265628457623368, 1.40821553384062)
```

To reduce the effect of the outliers on the estimated mean and standard deviation, we'll use the SciPy function `trimboth` to remove the highest and lowest values:

```
from scipy.stats import trimboth

trimmed = trimboth(birth_weights, 0.01)
m, s = np.mean(trimmed), np.std(trimmed)
m, s
```

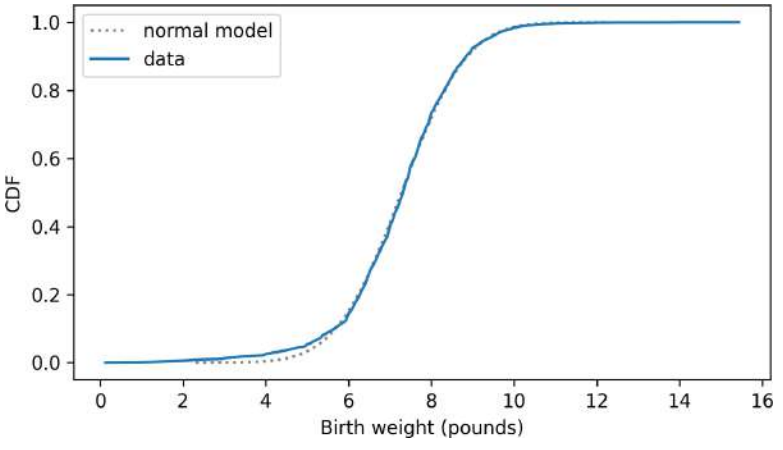
```
(7.280883100022579, 1.2430657948614345)
```

With the trimmed data, the mean is a little lower and the standard deviation is substantially lower. We'll use the trimmed data to make a normal model:

```
cdf_model = make_normal_model(trimmed)
```

And compare it to the `Cdf` of the data:

```
cdf_birth_weight = Cdf.from_seq(birth_weights, name='data')

two_cdf_plots(cdf_model, cdf_birth_weight, xlabel="Birth weight (pounds)")
```



The normal model fits the data well except below five pounds, where the distribution of the data is to the left of the model—that is, the lightest babies are lighter than we'd expect in a normal distribution. The real world is usually more complicated than simple mathematical models.

The Lognormal Distribution

In the previous section, we simulated pumpkin growth under the assumption that pumpkins grow one to three pounds per day, depending on the weather. Instead, let's suppose their growth is proportional to their current weight, so big pumpkins gain more weight per day than small pumpkins—which is probably more realistic.

The following function simulates this kind of proportional growth, where a pumpkin gains 3% of its weight if the weather is bad, 5% if the weather is fair, and 7% if the weather is good. Again, we'll assume that the weather is bad, fair, or good on any given day with equal probability:

```
def simulate_proportionate_growth(n):
    choices = [1.03, 1.05, 1.07]
    gains = np.random.choice(choices, n)
    return gains.prod()
```

If a pumpkin gains 3% of its weight, the final weight is the product of the initial weight and the factor 1.03. So we can compute the weight after 100 days by choosing random factors and multiplying them together.

We'll call this function 1,001 times to simulate 1,001 pumpkins and save their weights:

```
sim_weights = [simulate_proportionate_growth(100) for i in range(1001)]
np.mean(sim_weights), np.std(sim_weights)
```

```
(130.80183363824722, 20.956047434921466)
```

The average weight is about 131 pounds; the standard deviation is about 21 pounds. So the pumpkins in this model are smaller but more variable than in the previous model.

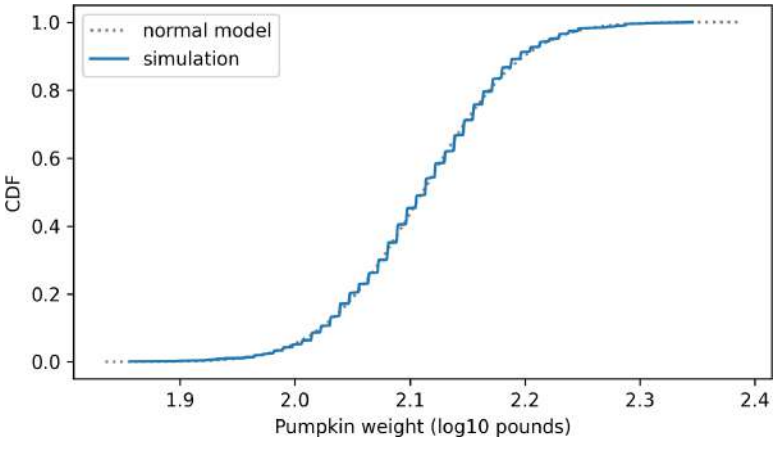
And we can show mathematically that they follow a **lognormal distribution**, which means that the logarithms of the weights follow a normal distribution. To check, we'll compute the logs of the weights and their mean and standard deviation. We could use logarithms with any base, but I'll use base 10 because it makes the results easier to interpret:

```
log_sim_weights = np.log10(sim_weights)
m, s = np.mean(log_sim_weights), np.std(log_sim_weights)
m, s
```

```
(2.1111299372609933, 0.06898607064749827)
```

Now let's compare the distribution of the logarithms to a normal distribution with the same mean and standard deviation:

```
cdf_model = make_normal_model(log_sim_weights)
cdf_log_sim_weights = Cdf.from_seq(log_sim_weights, name="simulation")

two_cdf_plots(
    cdf_model, cdf_log_sim_weights, xlabel="Pumpkin weight (log10 pounds)"
)
```



The model fits the simulation result very well, which is what we expected.

If people are like pumpkins, where the change in weight from year to year is proportionate to their current weight, we might expect the distribution of adult weights to follow a lognormal distribution. Let's find out.

The National Center for Chronic Disease Prevention and Health Promotion conducts an annual survey as part of the Behavioral Risk Factor Surveillance System (BRFSS). In 2008, they interviewed 414,509 respondents and asked about their demographics, health, and health risks. Among the data they collected are the weights, in kilograms, of 398,484 respondents. Instructions for downloading the data are in the notebook for this chapter.

The `thinkstats` module provides a function that reads BRFSS data and returns a Pandas `DataFrame`:

```
from thinkstats import read_brfss

brfss = read_brfss()
```

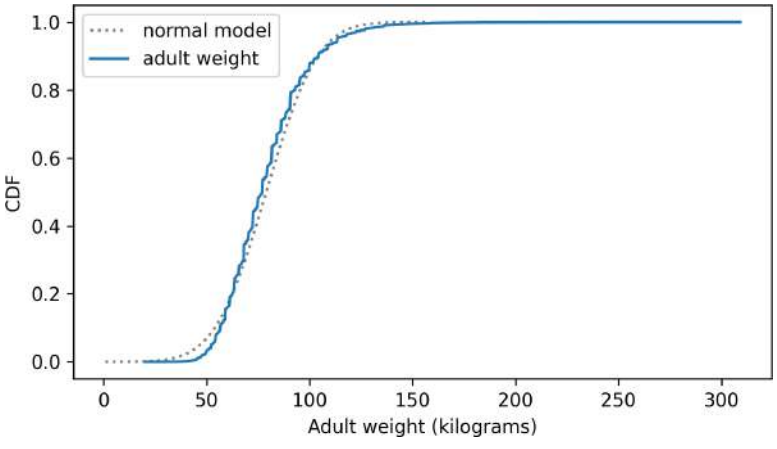
Adult weights in kilograms are recorded in the `wtkg2` column:

```
adult_weights = brfss["wtkg2"].dropna()
m, s = np.mean(adult_weights), np.std(adult_weights)
m, s
```

```
(78.9924529968581, 19.546132387397257)
```

The mean is about 79 kg. Before we compute logarithms, let's see if the weights follow a normal distribution:

```
cdf_model = make_normal_model(adult_weights)
cdf_adult_weights = Cdf.from_seq(adult_weights, name="adult weight")

two_cdf_plots(cdf_model, cdf_adult_weights, xlabel="Adult weight (kilograms)")
```

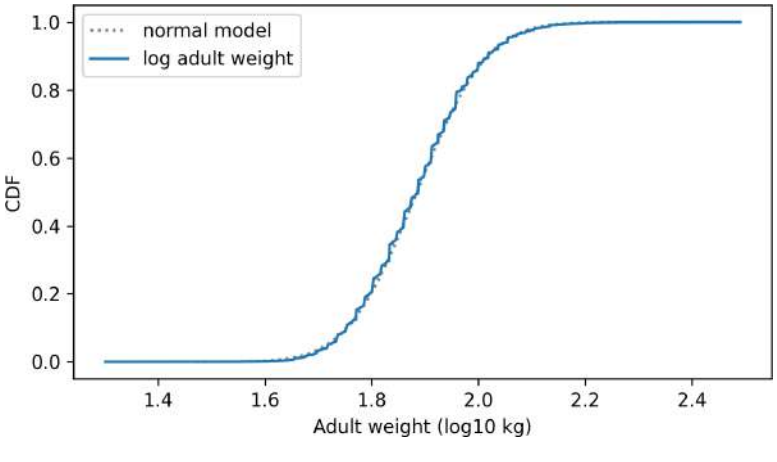


The normal distribution might be a good enough model for this data, for some purposes—but let's see if we can do better.

Here's the distribution of the log-transformed weights and a normal model with the same mean and standard deviation:

```
log_adult_weights = np.log10(adult_weights)
cdf_model = make_normal_model(log_adult_weights)

cdf_log_adult_weights = Cdf.from_seq(log_adult_weights, name="log adult weight")
```

```
two_cdf_plots(cdf_model, cdf_log_adult_weights, xlabel="Adult weight (log10 kg)")
```



The normal model fits the logarithms better than it fits the weights themselves, which suggests that proportional growth is a better model of weight gain than additive growth.

Why Model?

At the beginning of this chapter, I said that many real-world phenomena can be modeled with theoretical distributions. But it might not have been clear why we should care.

Like all models, theoretical distributions are abstractions, which means they leave out details that are considered irrelevant. For example, an observed distribution might have measurement errors or quirks that are specific to the sample; theoretical models ignore these idiosyncrasies.

Theoretical models are also a form of data compression. When a model fits a dataset well, a small set of values can effectively summarize a large amount of data.

It is sometimes surprising when data from a natural phenomenon fit a theoretical distribution, but these observations can provide insight into physical systems. Sometimes we can explain why an observed distribution has a particular form. For example, in the previous section we found that adult weight is well modeled by a lognormal distribution, which suggests that changes in weight from year to year might be proportional to current weight.

Also, theoretical distributions lend themselves to mathematical analysis, as we'll see in Chapter 14.

But it is important to remember that all models are imperfect. Data from the real world never fit a theoretical distribution perfectly. People sometimes talk as if data are generated by models; for example, they might say that the distribution of human heights is normal, or the distribution of income is lognormal. Taken literally, these claims cannot be true—there are always differences between the real world and mathematical models.

Models are useful if they capture the relevant aspects of the real world and leave out unneeded details. But what is relevant or unneeded depends on what you are planning to use the model for.

Glossary

binomial distribution
: A theoretical distribution often used to model the number of successes or hits in a sequence of hits and misses

Poisson distribution
: A theoretical distribution often used to model the number of events that occur in an interval of time

exponential distribution
: A theoretical distribution often used to model the time between events

normal distribution
: A theoretical distribution often used to model data that follow a symmetric, bell-like curve

lognormal distribution
: A theoretical distribution often used to model data that follow a bell-like curve that is skewed to the right

Exercises

Exercise 5.1

In the NSFG respondent file, the `numfmhh` column records the "number of family members in" each respondent's household. We can use `read_fem_resp` to read the file, and `query` to select respondents who were 25 or older when they were interviewed:

```
from nsfg import read_fem_resp

resp = read_fem_resp()
```

```
older = resp.query("age >= 25")
num_family = older["numfmhh"]
```

Compute the `Pmf` of `numfmhh` for these older respondents and compare it with a Poisson distribution with the same mean. How well does the Poisson model fit the data?

Exercise 5.2

Earlier in this chapter we saw that the time until the first goal in a hockey game follows an exponential distribution. If our model of goal scoring is correct, a goal is equally likely at any time, regardless of how long it has been since the previous goal. And if that's true, we expect the time between goals to follow an exponential distribution, too.

The following loop reads the hockey data again, computes the time between successive goals, if there is more than one in a game, and collects the inter-goal times in a list:

```
intervals = []

for key in keys:
    times = pd.read_hdf(filename, key=key)
    if len(times) > 1:
intervals.extend(times.diff().dropna())
```

Use `exponential_cdf` to compute the CDF of an exponential distribution with the same mean as the observed intervals and compare this model to the CDF of the data.

Exercise 5.3

Is the distribution of human height more like a normal or a lognormal distribution? To find out, we can select height data from the BRFSS like this:

```
adult_heights = brfss["htm3"].dropna()
m, s = np.mean(adult_heights), np.std(adult_heights)
m, s
```

```
(168.82518961012298, 10.35264015645592)
```

Compute the CDF of these values and compare it to a normal distribution with the same mean and standard deviation. Then compute the logarithms of the heights and compute the distribution of the logarithms to a normal distribution. Based on a visual comparison, which model fits the data better?

CHAPTER 6

Probability Density Functions

In the previous chapter, we modeled data with theoretical distributions including the binomial, Poisson, exponential, and normal distributions.

The binomial and Poisson distributions are **discrete**, which means that the outcomes have to be distinct or separate elements, like an integer number of hits and misses, or goals scored. In a discrete distribution, each outcome is associated with a probability mass.

The exponential and normal distribution are **continuous**, which means the outcomes can be at any point in a range of possible values. In a continuous distribution, each outcome is associated with a **probability density**. Probability density is an abstract idea, and many people find it difficult at first, but we'll take it one step at a time. As a first step, let's think again about comparing distributions.

Comparing Distributions

In the previous chapter, when we compared discrete distributions, we used a bar plot to show their probability mass functions (PMFs). When we compared continuous distributions, we used a line plot to show their cumulative distribution functions (CDFs).

For the discrete distributions, we could also have used CDFs. For example, here's the PMF of a Poisson distribution with `lam=2.2`, which is a good model for the distribution of household size in the NSFG data.

We can use `read_fem_resp` to read the respondent data file:

```
from nsfg import read_fem_resp

resp = read_fem_resp()
```

Next we'll select household sizes for people 25 and older:

```
older = resp.query("age >= 25")
num_family = older["numfmhh"]
```

And make a `Pmf` that represents the distribution of responses:

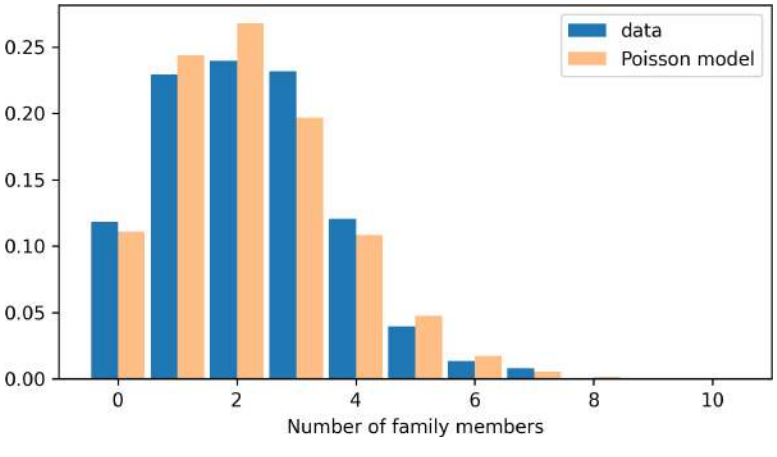
```
from empiricaldist import Pmf

pmf_family = Pmf.from_seq(num_family, name="data")
```

Here's another `Pmf` that represents a Poisson distribution with the same mean:

```
from thinkstats import poisson_pmf

lam = 2.2
ks = np.arange(11)
ps = poisson_pmf(ks, lam)

pmf_poisson = Pmf(ps, ks, name="Poisson model")
```

And here's how the distribution of the data compares to the Poisson model:

```
from thinkstats import two_bar_plots

two_bar_plots(pmf_family, pmf_poisson)
decorate(xlabel="Number of family members")
```



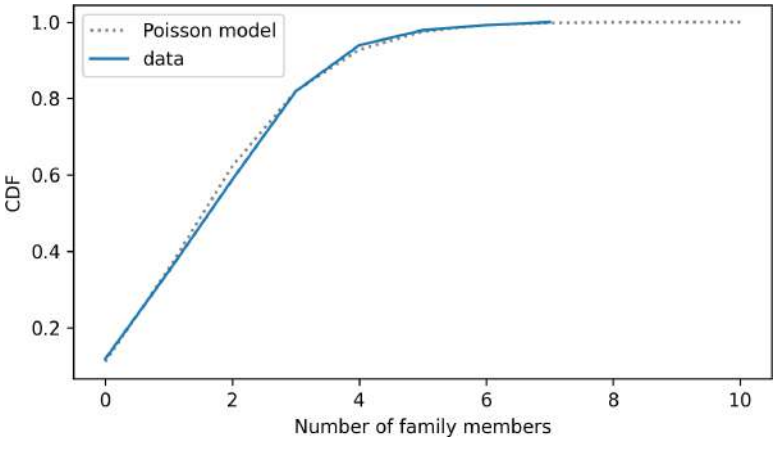
Comparing the PMFs, we can see that the model fits the data well, but with some deviations.

To get a sense of how substantial those deviations are, it can be helpful to compare CDFs. We can use `make_cdf` to compute the CDFs of the data and the model:

```
cdf_family = pmf_family.make_cdf()
cdf_poisson = pmf_poisson.make_cdf()
```

Here's what they look like:

```
from thinkstats import two_cdf_plots

two_cdf_plots(cdf_poisson, cdf_family)
decorate(xlabel="Number of family members")
```



When we compare CDFs, the deviations are less prominent, but we can see where and how the distributions differ. PMFs tend to emphasize small differences—sometimes CDFs provide a better sense of the big picture.

CDFs also work well with continuous data. As an example, let's look at the distribution of birth weights again, which is in the NSFG pregnancy file:

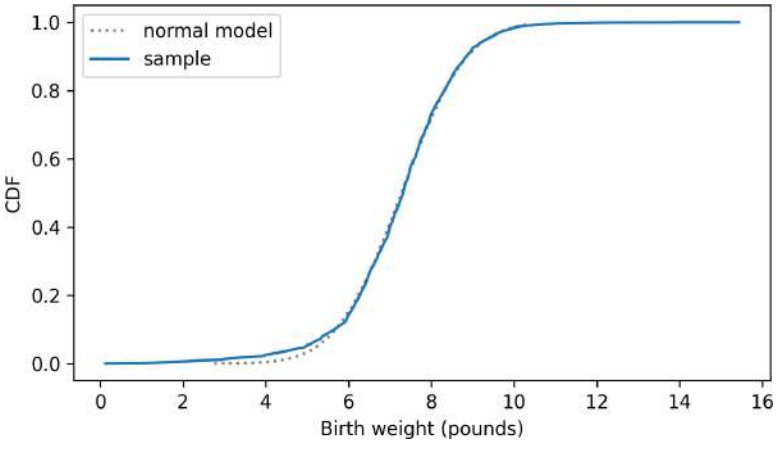
```
from nsfg import read_fem_preg

preg = read_fem_preg()
birth_weights = preg["totalwgt_lb"].dropna()
```

Here is the code we used in the previous chapter to fit a normal model to the data:

```
from scipy.stats import trimboth
from thinkstats import make_normal_model

trimmed = trimboth(birth_weights, 0.01)
cdf_model = make_normal_model(trimmed)
```

And here's the distribution of the data compared to the normal model:

```
from empiricaldist import Cdf

cdf_birth_weight = Cdf.from_seq(birth_weights, name="sample")
two_cdf_plots(cdf_model, cdf_birth_weight, xlabel="Birth weight (pounds)")
```



As we saw in the previous chapter, the normal model fits the data well except in the range of the lightest babies.

In my opinion, CDFs are usually the best way to compare data to a model. But for audiences that are not familiar with CDFs, there is one more option: probability density functions.

Probability Density

We'll start with the **probability density function (PDF)** of the normal distribution, which computes the density for the quantities, `xs`, given `mu` and `sigma`:

```
def normal_pdf(xs, mu, sigma):
    z = (xs - mu) / sigma
    return np.exp(-(z**2) / 2) / sigma / np.sqrt(2 * np.pi)
```

For `mu` and `sigma`, we'll use the mean and standard deviation of the trimmed birth weights:

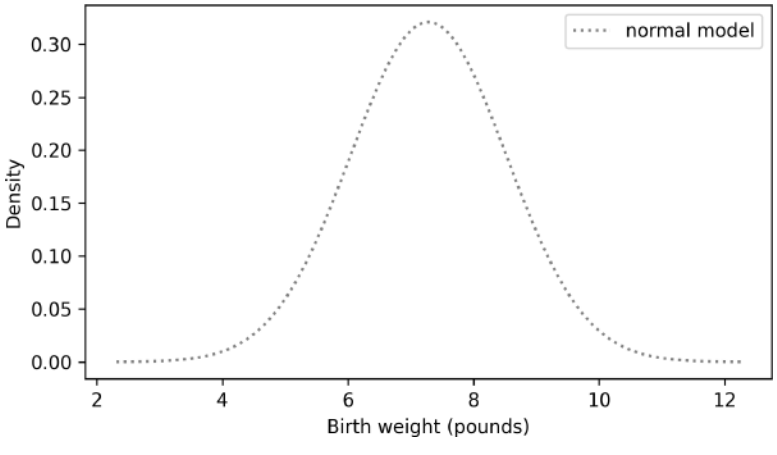
```
m, s = np.mean(trimmed), np.std(trimmed)
```

Now we'll evaluate `normal_pdf` for a range of weights:

```
low = m - 4 * s
high = m + 4 * s
qs = np.linspace(low, high, 201)
ps = normal_pdf(qs, m, s)
```

And plot it:

```
plt.plot(qs, ps, label="normal model", ls=":", color="gray")
decorate(xlabel="Birth weight (pounds)", ylabel="Density")
```



The result looks like a bell curve, which is characteristic of the normal distribution.

When we evaluate `normal_pdf`, the result is a probability density. For example, here's the density function evaluated at the mean, which is where the density is highest:

```
normal_pdf(m, m, s)
```

```
0.32093416297880123
```

By itself, a probability density doesn't mean much—most importantly, it is *not* a probability. It would be incorrect to say that the probability is 32% that a randomly chosen birth weight equals `m`. In fact, the probability that a birth weight is truly, exactly, and precisely equal to `m`—or any other specific value—is zero.

However, we can use the probability densities to compute the probability that an outcome falls in an interval between two values, by computing the area under the curve.

We could do that with the `normal_pdf` function, but it is more convenient to use the `NormalPdf` class, which is defined in the `thinkstats` module. Here's how we create a `NormalPdf` object with the same mean and standard deviation as the birth weights in the NSFG dataset:

```
from thinkstats import NormalPdf

pdf_model = NormalPdf(m, s, name="normal model")
pdf_model
```

```
NormalPdf(7.280883100022579, 1.2430657948614345, name='normal model')
```

If we call this object like a function, it evaluates the normal PDF:

```
pdf_model(m)
```

```
0.32093416297880123
```

Now, to compute the area under the PDF, we can use the following function, which takes a `NormalPdf` object and the bounds of an interval, `low` and `high`. It evaluates the normal PDF at equally spaced quantities between `low` and `high`, and uses the SciPy function `simpson` to estimate the area under the curve (`simpson` is so named because it uses an algorithm called Simpson's method):

```
from scipy.integrate import simpson


def area_under(pdf, low, high):
    qs = np.linspace(low, high, 501)
    ps = pdf(qs)
    return simpson(y=ps, x=qs)
```

If we compute the area under the curve from the lowest to the highest point in the graph, the result is close to 1:

```
area_under(pdf_model, 2, 12)
```

```
0.9999158086616793
```

If we extend the interval from negative infinity to positive infinity, the total area is exactly 1.

If we start from 0—or any value far below the mean—we can compute the fraction of birth weights less than or equal to 8.5 pounds:

```
area_under(pdf_model, 0, 8.5)
```

```
0.8366380335513807
```

You might recall that the "fraction less than or equal to a given value" is the definition of the CDF. So we could compute the same result using the CDF of the normal distribution:

```
from scipy.stats import norm

norm.cdf(8.5, m, s)
```

```
0.8366380358092718
```

Similarly, we can use the area under the density curve to compute the fraction of birth weights between 6 and 8 pounds:

```
area_under(pdf_model, 6, 8)
```

```
0.5671317752927691
```

Or we can get the same result using the CDF to compute the fraction less than 8 and then subtracting off the fraction less than 6:

```
norm.cdf(8, m, s) - norm.cdf(6, m, s)
```

```
0.5671317752921801
```

So the CDF is the area under the curve of the PDF. If you know calculus, another way to say the same thing is that the CDF is the integral of the PDF. And conversely, the PDF is the derivative of the CDF.

The Exponential PDF

To get your head around probability density, it might help to see another example. In the previous chapter, we used an exponential distribution to model the time until the first goal in a hockey game. We used the following function to compute the exponential CDF, where `lam` is the rate in goals per unit of time:

```
def exponential_cdf(x, lam):
    return 1 - np.exp(-lam * x)
```

We can compute the PDF of the exponential distribution like this:

```
def exponential_pdf(x, lam):
    return lam * np.exp(-lam * x)
```

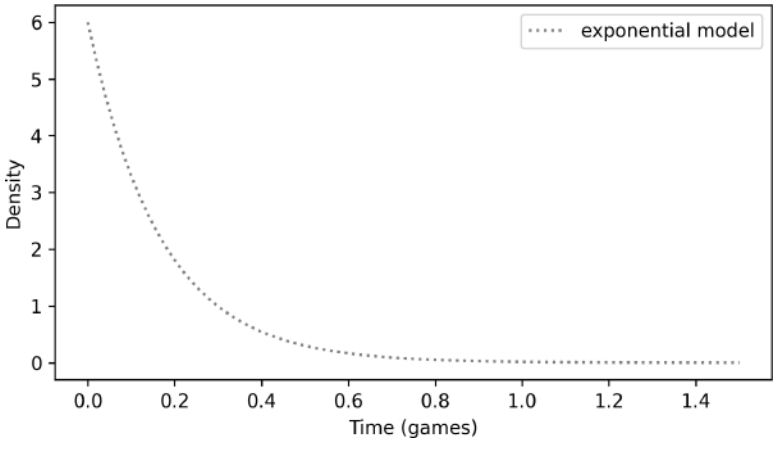
`thinkstats` provides an `ExponentialPdf` object that uses this function to compute the exponential PDF. We can use one to represent an exponential distribution with a rate of six goals per game:

```
from thinkstats import ExponentialPdf

lam = 6
pdf_expo = ExponentialPdf(lam, name="exponential model")
pdf_expo
```

```
ExponentialPdf(6, name='model')
```

`ExponentialPdf` provides a `plot` function we can use to plot the PDF—notice that the unit of time is games here, rather than seconds as in the previous chapter:

```
qs = np.linspace(0, 1.5, 201)
pdf_expo.plot(qs, ls=":", color="gray")
decorate(xlabel="Time (games)", ylabel="Density")
```



Looking at the y-axis, you might notice that some of these densities are greater than 1, which is a reminder that a probability density is not a probability. But the area under a density curve is a probability, so it should never be greater than 1.

If we compute the area under this curve from 0 to 1.5 games, we can confirm that the result is close to 1:

```
area_under(pdf_expo, 0, 1.5)
```

```
0.999876590779019
```

If we extend the interval much farther, the result is slightly greater than 1, but that's because we're approximating the area numerically. Mathematically, it is exactly 1, as we can confirm using the exponential CDF:

```
from thinkstats import exponential_cdf

exponential_cdf(7, lam)
```

```
1.0
```

We can use the area under the density curve to compute the probability of a goal during any interval. For example, here is the probability of a goal during the first minute of a 60-minute game:

```
area_under(pdf_expo, 0, 1 / 60)
```

```
0.09516258196404043
```

We can compute the same result using the exponential CDF:

```
exponential_cdf(1 / 60, lam)
```

```
0.09516258196404048
```

In summary, if we evaluate a PDF, the result is a probability density—which is not a probability. However, if we compute the area under the PDF, the result is the probability that a quantity falls in an interval. Or we can find the same probability by evaluating the CDF at the beginning and end of the interval and computing the difference.

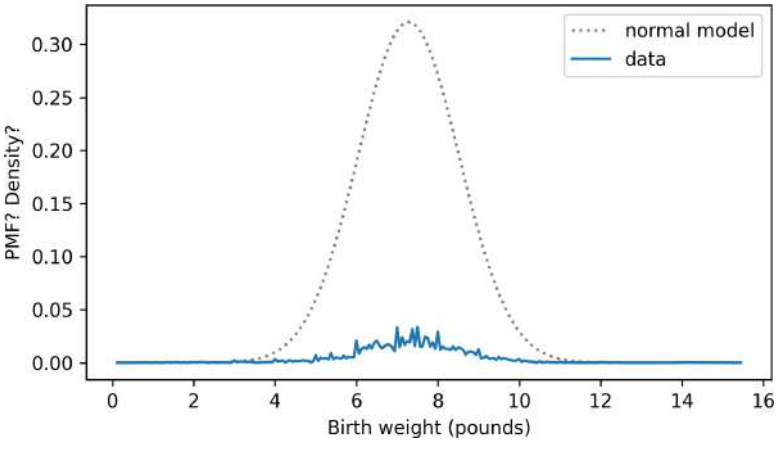
Comparing PMFs and PDFs

It is a common error to compare the PMF of a sample with the PDF of a theoretical model. For example, suppose we want to compare the distribution of birth weights to a normal model. Here's a `Pmf` that represents the distribution of the data:

```
pmf_birth_weight = Pmf.from_seq(birth_weights, name="data")
```

And we already have `pdf_model`, which represents the PDF of the normal distribution with the same mean and standard deviation. Here's what happens if we plot them on the same axis:

```
pdf_model.plot(ls=":", color="gray")
pmf_birth_weight.plot()

decorate(xlabel="Birth weight (pounds)", ylabel="PMF? Density?")
```



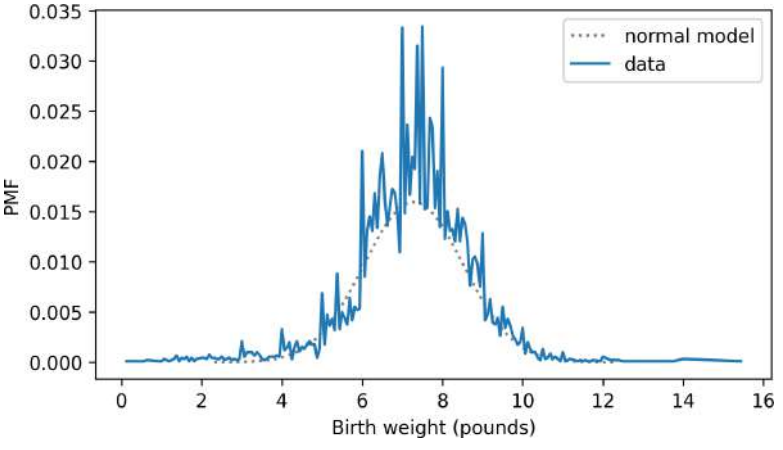
It doesn't work very well. One reason is that they are not in the same units. A PMF contains probability masses and a PDF contains probability densities, so we can't compare them, and we shouldn't plot them on the same axes.

As a first attempt to solve the problem, we can make a `Pmf` that approximates the normal distribution by evaluating the PDF at a discrete set of points. `NormalPdf` provides a `make_pmf` method that does that:

```
pmf_model = pdf_model.make_pmf()
```

The result is a normalized `Pmf` that contains probability masses, so we can at least plot it on the same axes as the PMF of the data:

```
pmf_model.plot(ls=":", color="gray")
pmf_birth_weight.plot()

decorate(xlabel="Birth weight (pounds)", ylabel="PMF")
```



But this is still not a good way to compare distributions. One problem is that the two `Pmf` objects contain different numbers of quantities, and the quantities in `pmf_birth_weight` are not equally spaced, so the probability masses are not really comparable:

```
len(pmf_model), len(pmf_birth_weight)
```

```
(201, 184)
```

The other problem is that the `Pmf` of the data is noisy. So let's try something else.

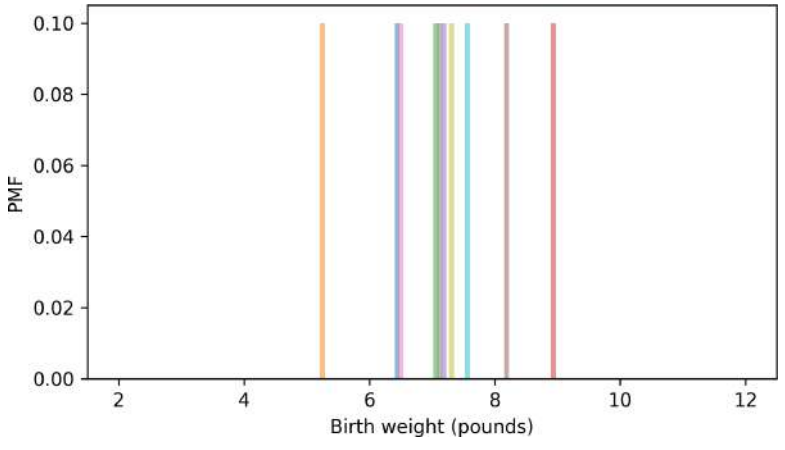
Kernel Density Estimation

Instead of using the model to make a PMF, we can use the data to make a PDF. To show how that works, I'll start with a small sample of the data:

```
n = 10
sample = birth_weights.sample(n)
```

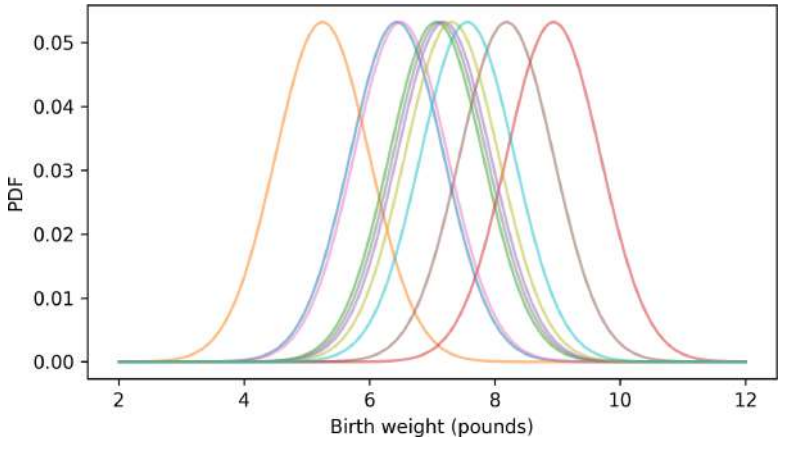
The `Pmf` of this sample looks like this:

```
for weight in sample:
    pmf = Pmf.from_seq([weight]) / n
    pmf.bar(width=0.08, alpha=0.5)

xlim = [1.5, 12.5]
decorate(xlabel="Birth weight (pounds)", ylabel="PMF", xlim=xlim)
```



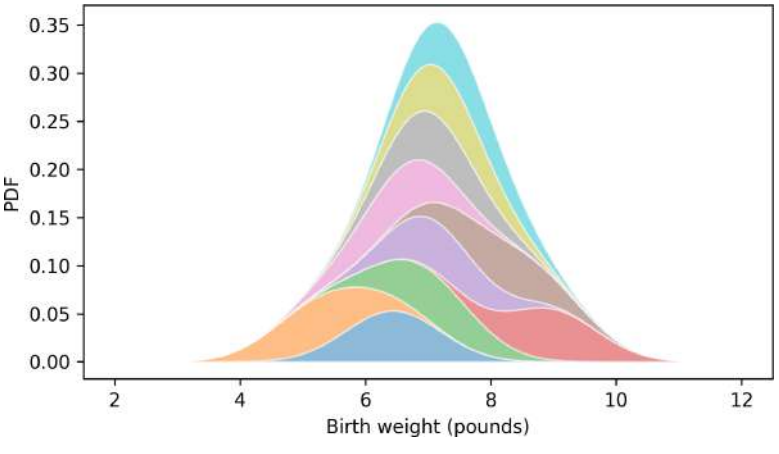
This way of representing the distribution treats the data as if it is discrete, so each probability mass is stacked up on a single point. But birth weight is actually continuous, so the quantities between the measurements are also possible. We can represent that possibility by replacing each discrete probability mass with a continuous probability density, like this:

```
qs = np.linspace(2, 12, 201)

for weight in sample:
    ps = NormalPdf(weight, 0.75)(qs) / n
    plt.plot(qs, ps, alpha=0.5)

decorate(xlabel="Birth weight (pounds)", ylabel="PDF", xlim=xlim)
```



For each weight in the sample, we create a `NormalPdf` with the observed weight as the mean—now let's add them up:

```
low_ps = np.zeros_like(qs)

for weight in sample:
    ps = NormalPdf(weight, 0.75)(qs) / n
    high_ps = low_ps + ps
    plt.fill_between(qs, low_ps, high_ps, alpha=0.5, lw=1, ec="white")
    low_ps = high_ps

decorate(xlabel="Birth weight (pounds)", ylabel="PDF", xlim=xlim)
```



When we add up the probability densities for each data point, the result is an estimate of the probability density for the whole sample. This process is called **kernel density estimation** or KDE. In this context, a "kernel" is one of the small density functions we added up. Because the kernels we used are normal distributions—also known as Gaussians—we could say more specifically that we computed a Gaussian KDE.

SciPy provides a function called `gaussian_kde` that implements this algorithm. Here's how we can use it to estimate the distribution of birth weights:

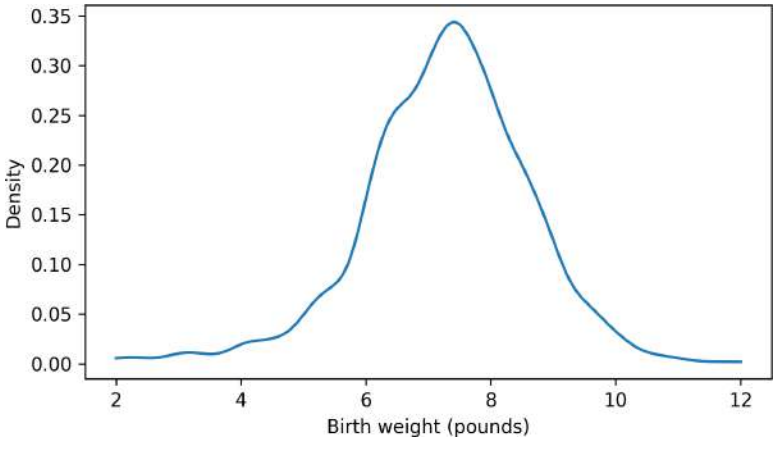
```
from scipy.stats import gaussian_kde

kde = gaussian_kde(birth_weights)
```

The result is an object that represents the estimated PDF, which we can evaluate by calling it like a function:

```
ps = kde(qs)
```

Here's what the result looks like:

```
plt.plot(qs, ps)

decorate(xlabel="Birth weight (pounds)", ylabel="Density")
```

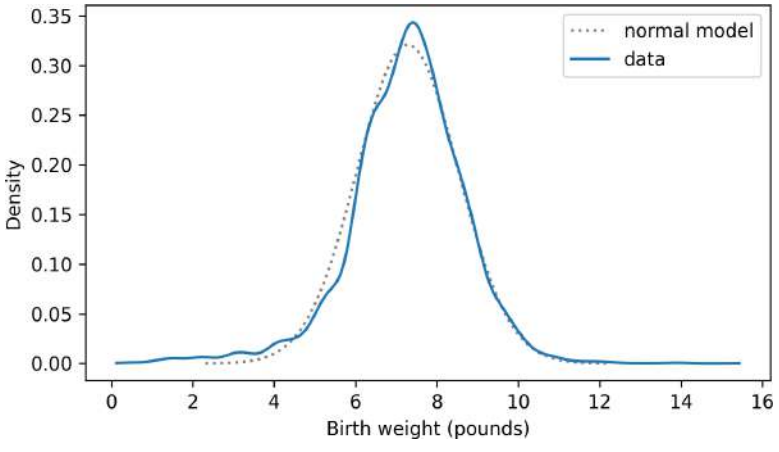


`thinkstats` provides a `Pdf` object that takes the result from `gaussian_kde`, and a domain that indicates where the density should be evaluated. Here's how we make one:

```
from thinkstats import Pdf

domain = np.min(birth_weights), np.max(birth_weights)
kde_birth_weights = Pdf(kde, domain, name="data")
```

`Pdf` provides a `plot` method we can use to compare the estimated PDF of the sample to the PDF of a normal distribution:

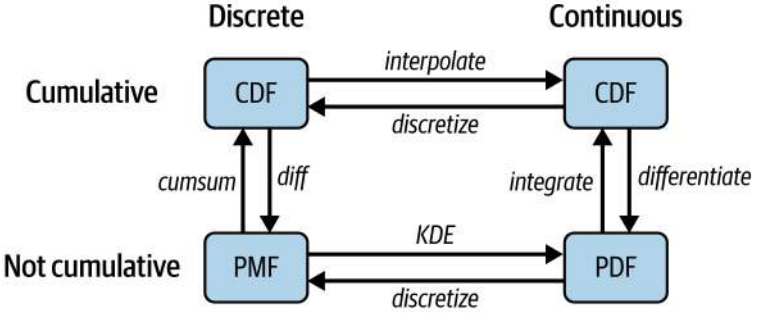
```
pdf_model.plot(ls=":", color="gray")
kde_birth_weights.plot()

decorate(xlabel="Birth weight (pounds)", ylabel="Density")
```



Kernel density estimation makes it possible to compare the distribution of a dataset to a theoretical model, and for some audiences, this is a good way to visualize the comparison. But for audiences that are familiar with CDFs, comparing CDFs is often better.

The Distribution Framework

At this point we have a complete set of ways to represent distributions: PMFs, CDFs, and PDFs The following figure shows these representations and the transitions from one to another. For example, if we have a `Pmf`, we can use the `cumsum` function to compute the cumulative sum of the probabilities and get a `Cdf` that represents the same distribution:



To demonstrate these transitions, we'll use a new dataset that "contains the time of birth, sex, and birth weight for each of 44 babies born in one 24-hour period at a Brisbane, Australia, hospital," according to the description. Instructions for downloading the data are in the notebook for this chapter.

We can read the data like this:

```
from thinkstats import read_baby_boom

boom = read_baby_boom()
boom.head()
```

	time	sex	weight_g	minutes
0	5	1	3837	5
1	104	1	3334	64
2	118	2	3554	78
3	155	2	3838	115
4	257	2	3625	177

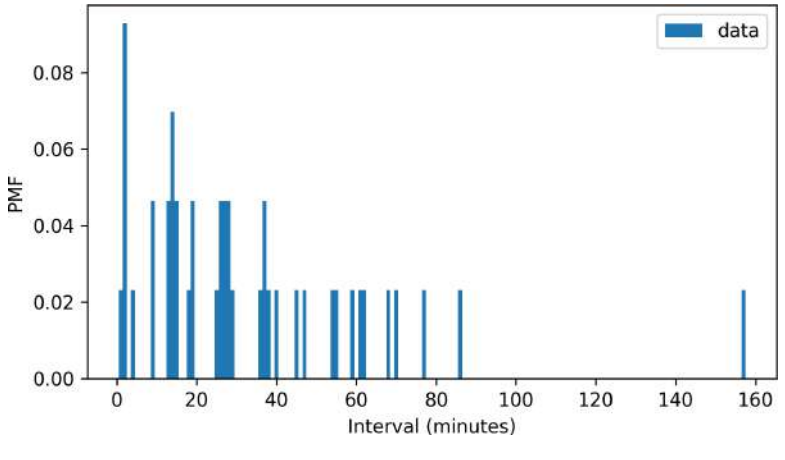
The `minutes` column records "the number of minutes since midnight for each birth." So we can use the `diff` method to compute the interval between each successive birth:

```
diffs = boom["minutes"].diff().dropna()
```

If births happen with equal probability during any minute of the day, we expect these intervals to follow an exponential distribution. In reality, that assumption is not precisely true, but the exponential distribution might still be a good model for the data.

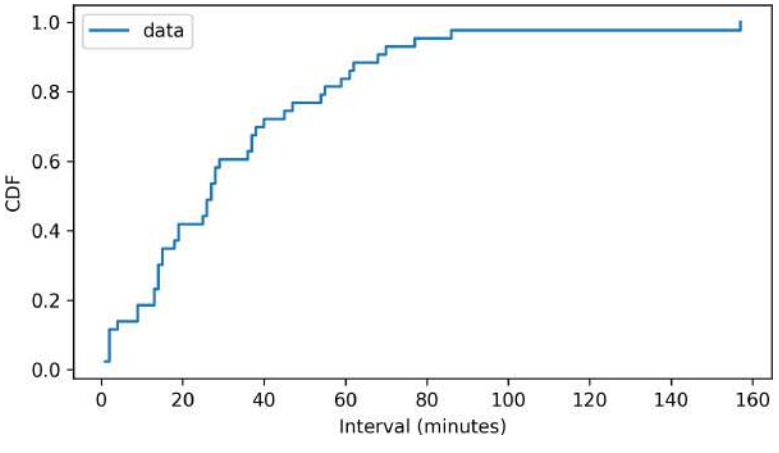
To find out, we'll start by making a `Pmf` that represents the distribution of intervals:

```
pmf_diffs = Pmf.from_seq(diffs, name="data")
pmf_diffs.bar(width=1)

decorate(xlabel="Interval (minutes)", ylabel="PMF")
```



Then we can use `make_cdf` to compute the cumulative probabilities and store them in a `Cdf` object:

```
cdf_diffs = pmf_diffs.make_cdf()
cdf_diffs.step()

decorate(xlabel="Interval (minutes)", ylabel="CDF")
```



The `Pmf` and `Cdf` are equivalent in the sense that if we are given either one, we can compute the other. To demonstrate, we'll use the `make_pmf` method, which computes the differences between successive probabilities in a `Cdf` and returns a `Pmf`:

```
pmf_diffs2 = cdf_diffs.make_pmf()
```

The result should be identical to the original `Pmf`, but there might be small floating-point errors. We can use `allclose` to check that the result is close to the original `Pmf`:

```
np.allclose(pmf_diffs, pmf_diffs2)
```

```
True
```

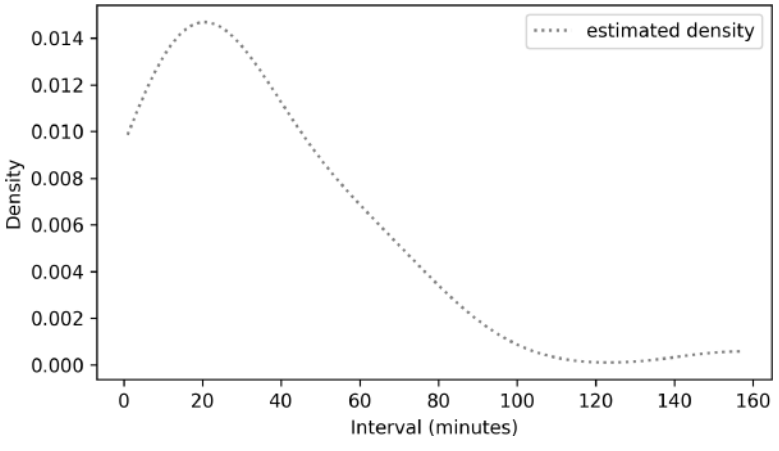
And it is.

From a `Pmf`, we can estimate a density function by calling `gaussian_kde` with the probabilities from the `Pmf` as weights:

```
kde = gaussian_kde(pmf_diffs.qs, weights=pmf_diffs.ps)
```

To plot the results, we can use `kde` to make a `Pdf` object, and call the `plot` method:

```
domain = np.min(pmf_diffs.qs), np.max(pmf_diffs.qs)
kde_diffs = Pdf(kde, domain=domain, name="estimated density")

kde_diffs.plot(ls=":", color="gray")
decorate(xlabel="Interval (minutes)", ylabel="Density")
```

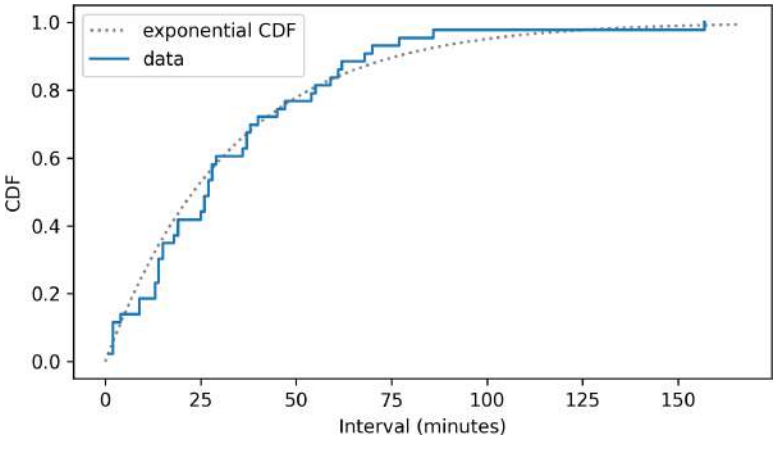


To see whether the estimated density follows an exponential model, we can make an `ExponentialCdf` with the same mean as the data:

```
from thinkstats import ExponentialCdf

m = diffs.mean()
lam = 1 / m
cdf_model = ExponentialCdf(lam, name="exponential CDF")
```

Here's what it looks like compared to the CDF of the data:

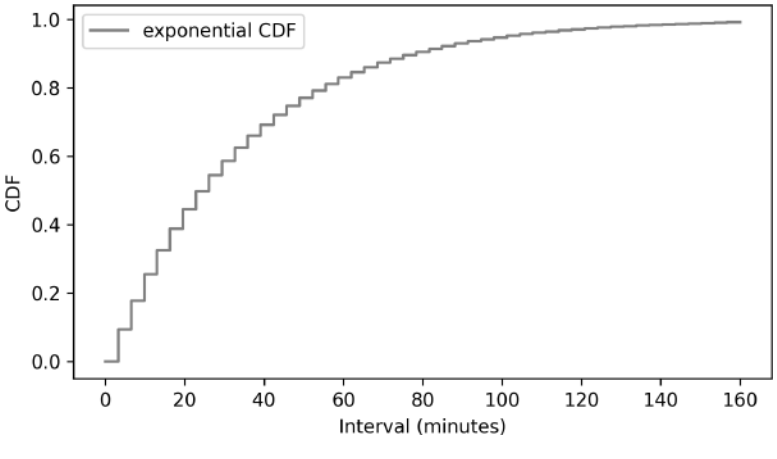
```
cdf_model.plot(ls=":", color="gray")
cdf_diffs.step()

decorate(xlabel="Interval (minutes)", ylabel="CDF")
```



The exponential model fits the CDF of the data well.

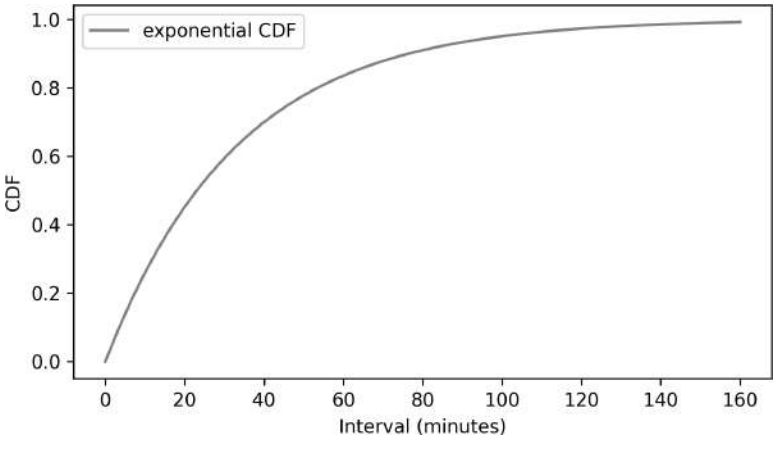
Given an `ExponentialCdf`, we can use `make_cdf` to **discretize** the CDF—that is, to make a discrete approximation by evaluating the CDF at a sequence of equally spaced quantities:

```
discrete_cdf_model = cdf_model.make_cdf(qs)
discrete_cdf_model.step()

decorate(xlabel="Interval (minutes)", ylabel="CDF")
```



Finally, to get from a discrete CDF to a continuous CDF, we can interpolate between the steps, which is what we see if we use the `plot` method instead of the `step` method:

```
discrete_cdf_model.plot(color="gray")
decorate(xlabel="Interval (minutes)", ylabel="CDF")
```



Finally, a PDF is the derivative of a continuous CDF, and a CDF is the integral of a PDF.

This example shows how we use `Pmf`, `Cdf`, and `Pdf` objects to represent PMFs, CDFs, and PDFs, and demonstrates the process for converting from each to the others.

Glossary

continuous
: A quantity is continuous if it can have any value in a range on the number line. Most things we measure in the world—like weight, distance, and time—are continuous.

discrete
: A quantity is discrete if it can have a limited set of values, like integers or categories. Exact counts are discrete, as well as categorical variables.

probability density function (PDF)
: A function that shows how density (not probability) is spread across the values of a continuous variable. The area under the PDF within an interval gives the probability that the variable falls in that interval.

probability density
: The value of a PDF at a specific point; it's not a probability itself, but it can be used to compute a probability.

kernel density estimation (KDE)
: A method for estimating a PDF based on a sample.

discretize
: To approximate a continuous quantity by dividing its range into discrete levels or categories.

Exercises

Exercise 6.1

In World Cup soccer (football), suppose the time until the first goal is well modeled by an exponential distribution with rate `lam=2.5` goals per game. Make an `ExponentialPdf` to represent this distribution and use `area_under` to compute the probability that the time until the first goal is less than half of a game. Then use an `ExponentialCdf` to compute the same probability and check that the results are consistent.

Use `ExponentialPdf` to compute the probability the first goal is scored in the second half of the game. Then use an `ExponentialCdf` to compute the same probability and check that the results are consistent.

Exercise 6.2

To join Blue Man Group, you have to be male between 5′10″ and 6′1″, which is roughly 178 to 185 centimeters. Let's see what fraction of the male adult population in the United States meets this requirement.

The heights of male participants in the BRFSS are well modeled by a normal distribution with mean 178 cm and standard deviation 7 cm:

```
from thinkstats import read_brfss

brfss = read_brfss()
male = brfss.query("sex == 1")
heights = male["htm3"].dropna()
```

```
from scipy.stats import trimboth

trimmed = trimboth(heights, 0.01)
```

```
m, s = np.mean(trimmed), np.std(trimmed)
m, s
```

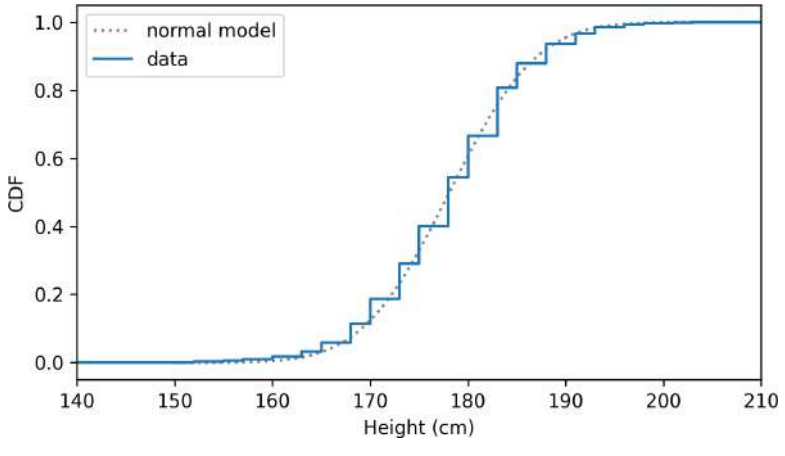
```
(178.10278947124948, 7.017054887136004)
```

Here's a `NormalCdf` object that represents a normal distribution with the same mean and standard deviation as the trimmed data:

```
from thinkstats import NormalCdf

cdf_normal_model = NormalCdf(m, s, name='normal model')
```

And here's how it compares to the CDF of the data:

```
cdf_height = Cdf.from_seq(heights, name="data")
cdf_normal_model.plot(ls=":", color="gray")
cdf_height.step()

xlim = [140, 210]
decorate(xlabel="Height (cm)", ylabel="CDF", xlim=xlim)
```



Use `gaussian_kde` to make a `Pdf` that approximates the PDF of male height. Hint: investigate the `bw_method` argument, which can be used to control the smoothness of the estimated density. Plot the estimated density and compare it to a `NormalPdf` with mean `m` and standard deviation `s`.

Use a `NormalPdf` and `area_under` to compute the fraction of people in the normal model that are between 178 and 185 centimeters. Use a `NormalCdf` to compute the same fraction, and check that the results are consistent. Finally, use the empirical `Cdf` of the data to see what fraction of people in the dataset are in the same range.

CHAPTER 7

Relationships Between Variables

So far we have only looked at one variable at a time. In this chapter we start looking at relationships between variables. Two variables are related if knowing one gives you information about the other. For example, height and weight are related—people who are taller tend to be heavier. Of course, it is not a perfect relationship: there are short heavy people and tall light ones. But if you are trying to guess someone's weight, you will be more accurate if you know their height than if you don't.

This chapter presents several ways to visualize relationships between variables, and one way to quantify the strength of a relationship, correlation.

Scatter Plots

If you meet someone who is unusually good at math, do you expect their verbal skills to be better or worse than average? On the one hand, you might imagine that people specialize in one area or the other, so someone who excels at one might be less good at the other. On the other hand, you might expect someone who is generally smart to be above average in both areas. Let's find out which it is.

We'll use data from the National Longitudinal Survey of Youth 1997 (NLSY97), which "follows the lives of a sample of 8,984 American youth born between 1980-84." The public data set includes the participants' scores on several standardized tests, including the tests most often used in college admissions, the SAT and ACT. Because test-takers get separate scores for the math and verbal sections, we can use this data to explore the relationship between mathematical and verbal ability.

Instructions for downloading the data are in the notebook for this chapter. We can use `read_csv` to read the data and `replace` to replace the special codes for missing data with `np.nan`:

```
missing_codes = [-1, -2, -3, -4, -5]
nlsy = pd.read_csv("nlsy97-extract.csv.gz").replace(missing_codes, np.nan)
nlsy.shape
```

```
(8984, 34)
```

The `DataFrame` contains one row for each of the 8,984 participants in the survey and one column for each of the 34 variables I selected. The column names don't mean much by themselves, so let's replace the ones we'll use with more interpretable names:

```
nlsy["sat_verbal"] = nlsy["R9793800"]
nlsy["sat_math"] = nlsy["R9793900"]
```

Both columns contain a few values less than 200, which is not possible because 200 is the lowest score, so we'll replace them with `np.nan`:

```
columns = ["sat_verbal", "sat_math"]

for column in columns:
    invalid = nlsy[column] < 200
    nlsy.loc[invalid, column] = np.nan
```

Next we'll use `dropna` to select only rows where both scores are valid:

```
nlsy_valid = nlsy.dropna(subset=columns).copy()
nlsy_valid.shape
```

```
(1398, 36)
```

SAT scores are standardized so the mean is 500 and the standard deviation is 100. In the NLSY sample, the means and standard deviations are close to these values:

```
sat_verbal = nlsy_valid["sat_verbal"]
sat_verbal.mean(), sat_verbal.std()
```

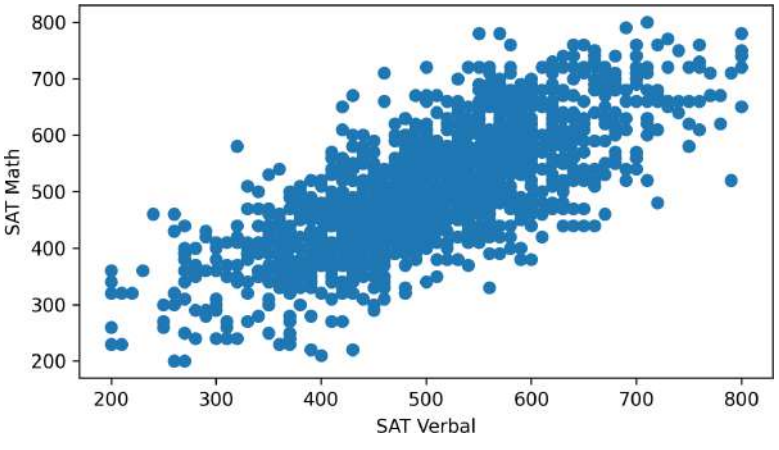
```
(501.80972818311875, 108.36562024213643)
```

```
sat_math = nlsy_valid["sat_math"]
sat_math.mean(), sat_math.std()
```

```
(503.0829756795422, 109.8329973731453)
```

Now, to see whether there is a relationship between these variables, let's look at a **scatter plot**:

```
plt.scatter(sat_verbal, sat_math)

decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```

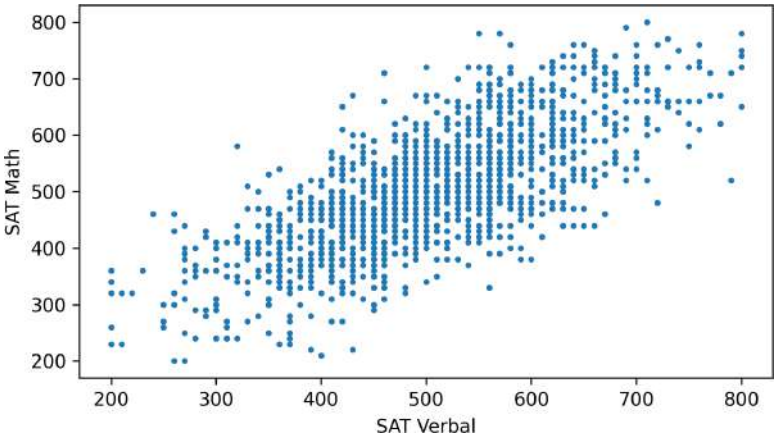


Using the default options of the `scatter` function, we can see the general shape of the relationship. People who do well on one section of the test tend to do better on the other, too.

However, this version of the figure is **overplotted**, which means there are a lot of overlapping points, which can create a misleading impression of the relationship. The center, where the density of points is highest, is not as dark as it should be—by comparison, the extreme values are darker than they should be. Overplotting tends to give too much visual weight to outliers.

We can improve the plot by reducing the size of the markers so they overlap less:

```
plt.scatter(sat_verbal, sat_math, s=5)

decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```



Now we can see that the markers are aligned in rows and columns, because scores are rounded off to the nearest multiple of 10. Some information is lost in the process.

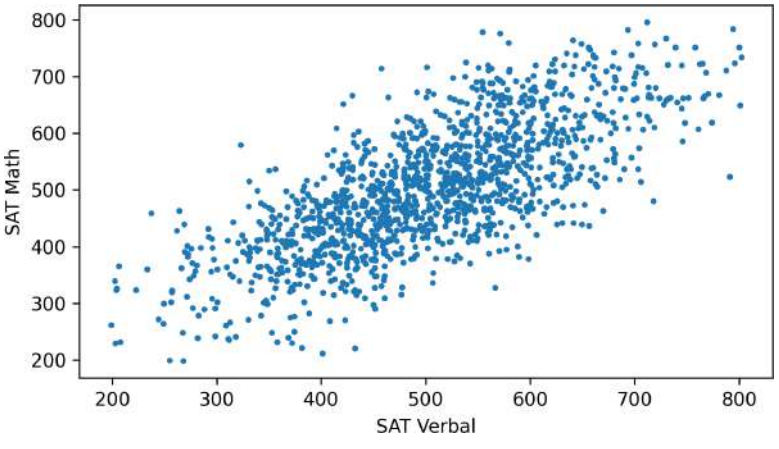
We can't get that information back, but we can minimize the effect on the scatter plot by **jittering** the data, which means adding random noise to reverse the effect of rounding off. The following function takes a sequence and jitters it by adding random values from a normal distribution with mean 0 and the given standard deviation. The result is a NumPy array:

```
def jitter(seq, std=1):
    n = len(seq)
    return np.random.normal(0, std, n) + seq
```

If we jitter the scores with a standard deviation of 3, the rows and columns are no longer visible in the scatter plot:

```
sat_verbal_jittered = jitter(sat_verbal, 3)
sat_math_jittered = jitter(sat_math, 3)
```

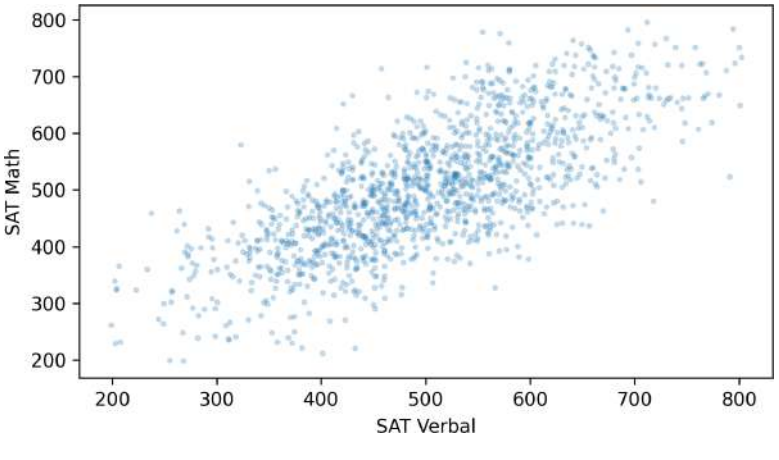
```
plt.scatter(sat_verbal_jittered, sat_math_jittered, s=5)

decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```



Jittering reduces the visual effect of rounding and makes the shape of the relationship clearer. But in general you should only jitter data for purposes of visualization and avoid using jittered data for analysis.

In this example, even after adjusting the marker size and jittering the data, there is still some overplotting. So let's try one more thing: we can use the `alpha` parameter to make the markers partly transparent:

```
plt.scatter(sat_verbal_jittered, sat_math_jittered, s=5, alpha=0.2)

decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```



With transparency, overlapping data points look darker, so darkness is proportional to density.

Although scatter plots are a simple and widely used visualization, they can be hard to get right. In general, it takes some trial and error to adjust marker sizes, transparency, and jittering to find the best visual representation of the relationship between variables.

Decile Plots

Scatter plots provide a general impression of the relationship between variables, but there are other visualizations that provide more insight into the nature of the relationship. One of them is a **decile plot**.

To generate a decile plot, we'll sort the respondents by verbal score and divide them into 10 groups, called **deciles**. We can use the `qcut` method to compute the deciles:

```
deciles = pd.qcut(nlsy_valid["sat_verbal"], 10, labels=False) + 1
deciles.value_counts().sort_index()
```

```
sat_verbal
1     142
2     150
3     139
4     140
5     159
6     130
7     148
8     121
9     138
10    131
Name: count, dtype: int64
```

The number of respondents in each decile is roughly equal.

Now we can use the `groupby` method to divide the `DataFrame` into groups by `decile`:

```
df_groupby = nlsy_valid.groupby(deciles)
df_groupby
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f369c918a60>
```

The result is a `DataFrameGroupBy` object that represents the groups. We can select the `sat_math` column from it:

```
series_groupby = df_groupby["sat_math"]
series_groupby
```

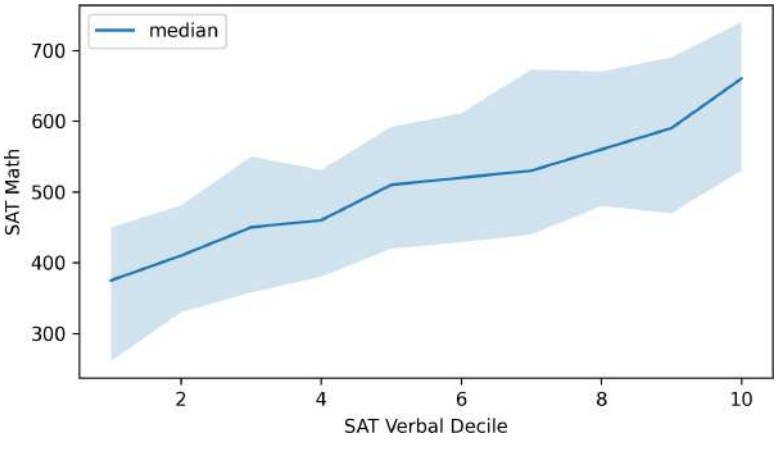
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f369fdb3760>
```

The result is a `SeriesGroupBy` object that represents the math scores in each decile. We can use the `quantile` function to compute the 10th, 50th, and 90th percentiles in each group:

```
low = series_groupby.quantile(0.1)
median = series_groupby.quantile(0.5)
high = series_groupby.quantile(0.9)
```

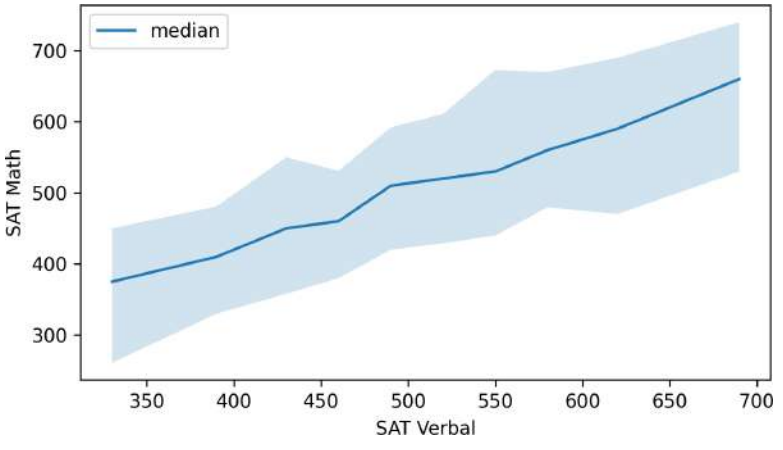
A decile plot shows these percentiles for each decile group. In the following figure, the line shows the median, and the shaded region shows the area between the 10th and 90th percentiles:

```
xs = median.index
plt.fill_between(xs, low, high, alpha=0.2)
plt.plot(xs, median, label="median")

decorate(xlabel="SAT Verbal Decile", ylabel="SAT Math")
```



As an alternative, we can compute the median verbal score in each group and plot those values on the x-axis, rather than the decile numbers:

```
xs = df_groupby["sat_verbal"].median()

plt.fill_between(xs, low, high, alpha=0.2)
plt.plot(xs, median, color="C0", label="median")

decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```



It looks like the relationship between these variables is linear—that is, each increase in the median verbal scores corresponds to a roughly equal increase in median math scores.

More generally, we could divide the respondents into any number of groups, not necessarily 10, and we could compute other summary statistics in each group, not just these percentiles.

Correlation

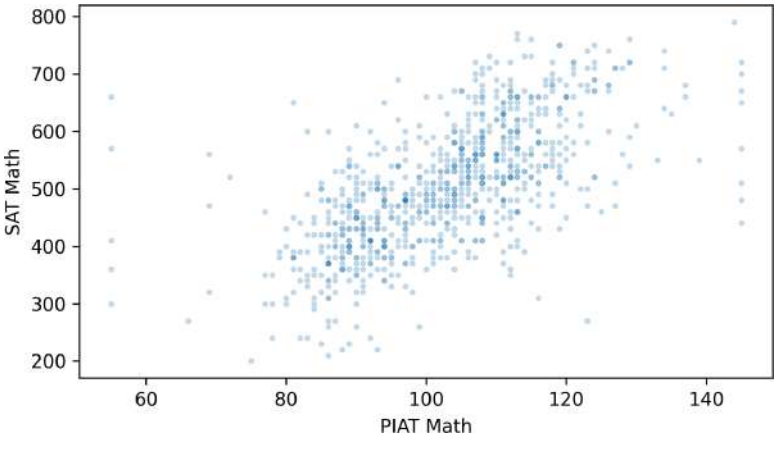
When the NLSY participants were in ninth grade, many of them took the mathematics section of the Peabody Individual Achievement Test (PIAT). Let's give the column that contains the results a more interpretable name:

```
nlsy["piat_math"] = nlsy["R1318200"]
nlsy["piat_math"].describe()
```

```
count    6044.000000
mean       93.903706
std        14.631148
min        55.000000
25%        84.000000
50%        92.000000
75%       103.000000
max       145.000000
Name: piat_math, dtype: float64
```

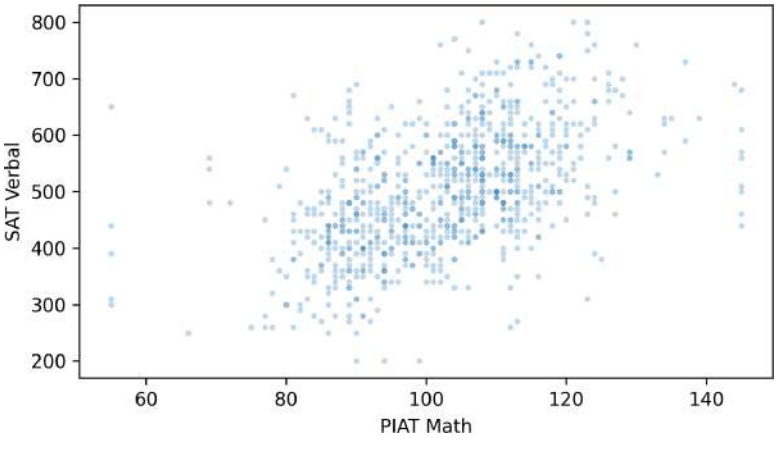
Students who do well on the PIAT in ninth grade are likely to do well on the SAT math section in twelfth grade. For the NLSY participants who took both tests, the following scatter plot shows the relationship between their scores. It uses the `scatter` function in `thinkstats`, which adjusts the marker size and transparency, and optionally jitters the data:

```
from thinkstats import scatter

scatter(nlsy, "piat_math", "sat_math")

decorate(xlabel="PIAT Math", ylabel="SAT Math")
```



As expected, students who do well on the PIAT are likely to do well on the SAT math. And if math and verbal ability are related, we expect them do well on the SAT verbal section, too. The following figure shows the relationship between the PIAT and SAT verbal scores:

```
scatter(nlsy, "piat_math", "sat_verbal")

decorate(xlabel="PIAT Math", ylabel="SAT Verbal")
```



Students with higher PIAT scores also have higher SAT verbal scores, on average.

Comparing the scatter plots, the points in the first figure might be more compact, and the points in the second figure more dispersed. If so, that means that the PIAT math scores predict SAT math scores more accurately than they predict SAT verbal scores—and it makes sense if they do.

To quantify the strength of these relationships, we can use the **Pearson correlation coefficient**, often just called "correlation." To understand correlation, let's start with standardization.

To standardize a variable, we subtract off the mean and divide through by the standard deviation, as in this function:

```
def standardize(xs):
    return (xs - np.mean(xs)) / np.std(xs)
```

To show how it's used, we'll select the rows where `piat_math` and `sat_math` are valid:

```
valid = nlsy.dropna(subset=["piat_math", "sat_math"])
piat_math = valid["piat_math"]
sat_math = valid["sat_math"]
```

And standardize the PIAT math scores:

```
piat_math_standard = standardize(piat_math)
np.mean(piat_math_standard), np.std(piat_math_standard)
```

```
(-2.4321756236287047e-16, 1.0)
```

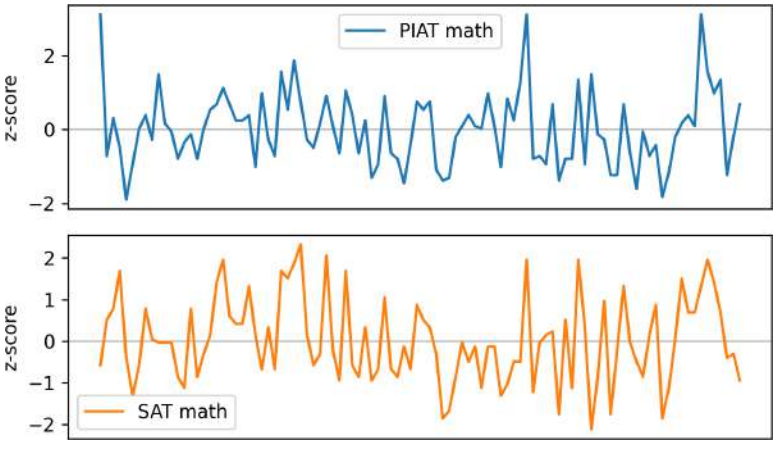
The results are **standard scores**, also called "z-scores." Because of the way the standard scores are calculated, the mean is close to 0 and the standard deviation is close to 1.

Let's also standardize the SAT math scores:

```
sat_math_standard = standardize(sat_math)
np.mean(sat_math_standard), np.std(sat_math_standard)
```

```
(-1.737268302591932e-16, 0.9999999999999998)
```

This figure shows sequences of these scores for the first 100 participants:



These variables are clearly related: when one is above the mean, the other is likely to be above the mean, too. To quantify the strength of this relationship, we'll multiply the standard scores element-wise and compute the average of the products.

When both scores are positive, their product is positive, so it tends to increase the average product. And when both scores are negative, their product is positive, so it also tends to increase the average product. When the scores have opposite signs, the product is negative, so it decreases the average product. As a result, the average product measures the similarity between the sequences:

```
np.mean(piat_math_standard * sat_math_standard)
```

```
0.639735816517885
```

The result, which is about 0.64, is the correlation coefficient. Here's one way to interpret it: if someone's PIAT math score is 1 standard deviation above the mean, we expect their SAT math score to be 0.64 standard deviations above the mean, on average.

The result is the same if we multiply the elements in the other order:

```
np.mean(sat_math_standard * piat_math_standard)
```

```
0.639735816517885
```

So the correlation coefficient is symmetric: if someone's SAT math score is 1 standard deviation above the mean, we expect their PIAT math score to be 0.64 standard deviations above the mean, on average.

Correlation is a commonly used statistic, so NumPy provides a function that computes it:

```
np.corrcoef(piat_math, sat_math)
```

```
array([[1.        , 0.63973582],
       [0.63973582, 1.        ]])
```

The result is a **correlation matrix**, with one row and one column for each variable. The value in the upper left is the correlation of `piat_math` with itself. The value in the lower right is the correlation of `sat_math` with itself. The correlation of any variable with itself is 1, which indicates perfect correlation.

The values in the upper right and lower left are the correlation of `piat_math` with `sat_math` and the correlation of `sat_math` with `piat_math`, which are necessarily equal.

`thinkstats` provides a `corrcoef` function that takes a `DataFrame` and two column names, selects the rows where both columns are valid, and computes their correlation:

```
from thinkstats import corrcoef

corrcoef(nlsy, "piat_math", "sat_math")
```

```
0.6397358165178849
```

We can use this function to compute the correlation of `piat_math` and `sat_verbal`:

```
corrcoef(nlsy, "piat_math", "sat_verbal")
```

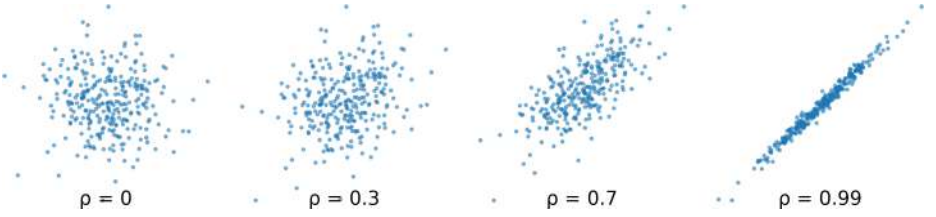
```
0.509413914696731
```

The correlation is about 0.51, so if someone's PIAT math score is one standard deviation above the mean, we expect their SAT verbal score to be 0.51 standard deviations above the mean, on average.

As we might expect, PIAT math scores predict SAT math scores better than they predict SAT verbal scores.

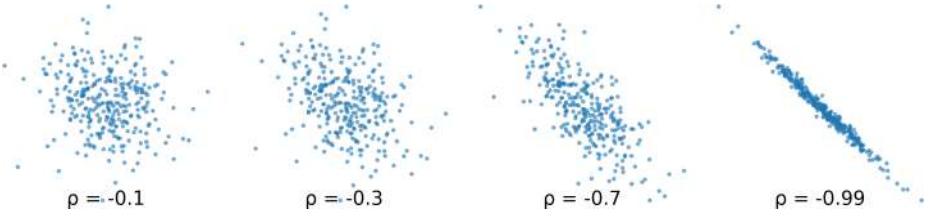
Strength of Correlation

As you look at more scatter plots, you will get a sense of what different correlations look like. To help you develop this sense, the following figure shows scatter plots for randomly generated data with the different correlations:



The Greek letter ρ, which is spelled "rho" and pronounced "row," is the conventional symbol for the correlation coefficient.

Correlation can also be negative. Here are scatter plots for random data with a range of negative correlations:



The correlation coefficient is always between –1 and 1. If there is no relationship between two variables, their correlation is 0—but if the correlation is 0, that doesn't necessarily mean there is no relationship.

In particular, if there is a nonlinear relationship, the correlation coefficient can be close to 0. In each of the following examples, there is a clear relationship between the variables in the sense that if we are given one of the values, we can make a substantially better prediction of the other. But in each case the correlation coefficient is close to 0:

Correlation quantifies the strength of a *linear* relationship between variables. If there is a nonlinear relationship, the correlation coefficient can be misleading. And if the correlation is close to 0, that does *not* mean there is no relationship.

Rank Correlation

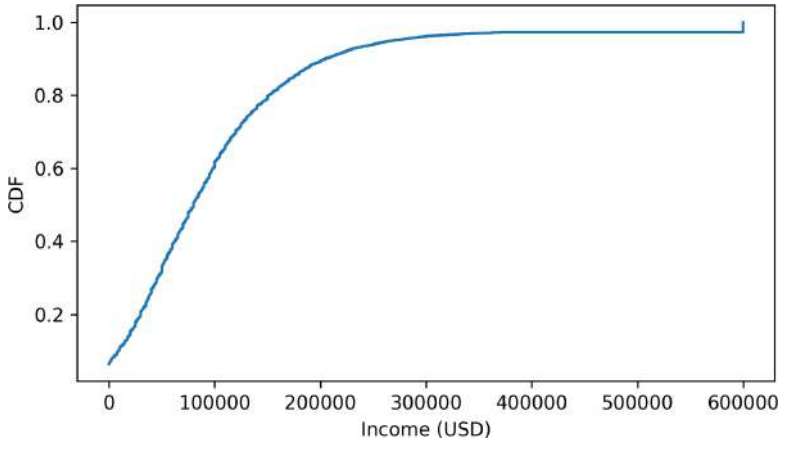
The NLSY is longitudinal, which means that it follows the same group of people over time. The group we've been studying includes people born between 1980 and 1984. The ones who took the SAT probably took it in the late 1990s, when they were about 18 years old. So when they were asked about their income in 2021, they were in their late 30s or early 40s. Let's give the column with the income data a more interpretable name:

```
nlsy["income"] = nlsy["U4949700"]
nlsy["income"].describe()
```

```
count      6051.000000
mean     104274.239960
std      108470.571497
min           0.000000
25%       38000.000000
50%       80000.000000
75%      134157.000000
max      599728.000000
Name: income, dtype: float64
```

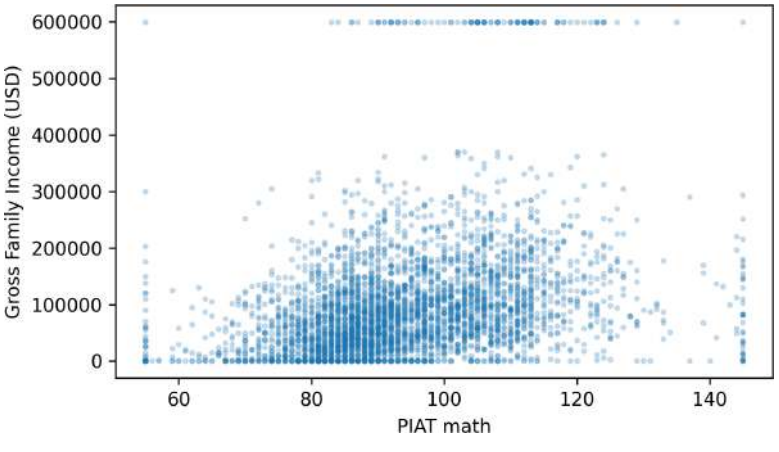
The values in this column are gross family income, which is total income of the respondent and the other members of their household, from all sources, reported in US dollars (USD). Here's what the distribution of income looks like:

```
cdf_income = Cdf.from_seq(nlsy["income"])
cdf_income.step()

decorate(xlabel="Income (USD)", ylabel="CDF")
```



Notice the step near $600,000—values above this threshold were capped to protect the anonymity of the participants. Now here's a scatter plot of the respondents' SAT math scores and their income later in life:

```
scatter(nlsy, "piat_math", "income")

decorate(xlabel="PIAT math", ylabel="Gross Family Income (USD)")
```



It looks like there is a relationship between these variables. Here is the correlation:

```
corrcoef(nlsy, "piat_math", "income")
```

```
0.30338587288641233
```

The correlation is about 0.3, which means that if someone gets a PIAT math score one standard deviation above the mean when they are 15 years old, we expect their income to be about 0.3 standard deviations above the mean when they are 40. That's not as strong as the correlation between PIAT scores and SAT scores, but considering the number of factors that affect income, it's pretty strong.

In fact, Pearson's correlation coefficient might understate the strength of the relationship. As we can see in the previous scatter plot, both variables have an apparent excess of values at the extremes. Because the correlation coefficient is based on the product of deviations from the mean, it is sensitive to these extreme values.

A more robust alternative is the **rank correlation**, which is based on the ranks of the scores rather than standardized scores. We can use the Pandas method `rank` to compute the rank of each score and each income:

```
valid = nlsy.dropna(subset=["piat_math", "income"])

piat_math_rank = valid["piat_math"].rank(method="first")
income_rank = valid["income"].rank(method="first")
```

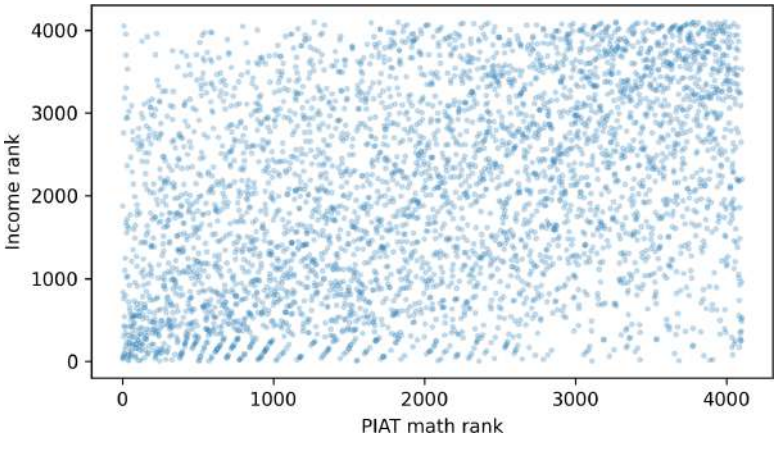
With the `method="first"` argument, `rank` assigns ranks from 1 to the length of the sequence, which is 4,101:

```
income_rank.min(), income_rank.max()
```

```
(1.0, 4101.0)
```

Here's a scatter plot of income ranks versus math score ranks:

```
plt.scatter(piat_math_rank, income_rank, s=5, alpha=0.2)

decorate(xlabel="PIAT math rank", ylabel="Income rank")
```



And here's the correlation of the ranks:

```
np.corrcoef(piat_math_rank, income_rank)[0, 1]
```

```
0.38148396696764847
```

The result is about 0.38, somewhat higher than the Pearson correlation, which is 0.30. Because rank correlation is less sensitive to the effect of extreme values, it is probably a better measure of the strength of the relationship between these variables.

`thinkplot` provides a `rankcorr` function that encapsulates the code in this section:

```
from thinkstats import rankcorr

rankcorr(nlsy, "piat_math", "income")
```

```
0.38474681505344815
```

As an exercise, you'll have a chance to compute the correlation between SAT verbal scores and income, using both the Pearson correlation and rank correlation.

Correlation and Causation

If variables A and B are correlated, the apparent correlation might be due to random sampling, or it might be the result of nonrepresentative sampling, or it might indicate a real correlation between quantities in the population.

If the correlation is real, there are three possible explanations: A causes B, or B causes A, or some other set of factors causes both A and B. These explanations are called "causal relationships."

Correlation alone does not distinguish between these explanations, so it does not tell you which ones are true. This rule is often summarized with the phrase "Correlation does not imply causation," which is so pithy it has its own Wikipedia page.

So what can you do to provide evidence of causation?

Use time.
: If A comes before B, then A can cause B but not the other way around. The order of events can help us infer the direction of causation, but it does not preclude the possibility that something else causes both A and B.

Use randomness.
: If you divide a large sample into two groups at random and compute the means of almost any variable, you expect the difference to be small. If the groups are nearly identical in all variables but A and B, you can rule out the possibility that something else causes both A and B.

These ideas are the motivation for the **randomized controlled trial**, in which subjects are assigned randomly to two (or more) groups: a **treatment group** that receives some kind of intervention, like a new medicine, and a **control group** that receives no intervention, or another treatment whose effects are known. A randomized controlled trial is the most reliable way to demonstrate a causal relationship, and the foundation of evidence-based medicine.

Unfortunately, controlled trials are sometimes impossible or unethical. An alternative is to look for a **natural experiment**, where similar groups are exposed to different conditions due to circumstances beyond the control of the experimenter.

Identifying and measuring causal relationships is the topic of a branch of statistics called **causal inference**.

Glossary

scatter plot
: A visualization that shows the relationship between two variables by plotting one point for each observation in the dataset.

overplotted
: A scatter plot is overplotted if many markers overlap, making it hard to distinguish areas of different density, which can misrepresent the relationship.

jitter
: Random noise added to data points in a plot to make overlapping values more visible.

decile plot
: A plot that divides data into deciles (ten groups) based on one variable, then summarizes another variable for each group.

decile
: One of the groups created by sorting data and dividing it into ten roughly equal parts.

Pearson correlation coefficient
: A statistic that measures the strength and sign (positive or negative) of the linear relationship between two variables.

standard score
: A quantity that has been standardized so that it is expressed in standard deviations from the mean.

correlation matrix
: A table showing the correlation coefficients for each pair of variables in a dataset.

rank correlation
: A robust way to quantify the strength of a relationship by using the ranks of values instead of the actual values.

randomized controlled trial
: An experiment where subjects are randomly assigned to groups that receive different treatments.

treatment group
: In an experiment, the group that receives the intervention being tested.

control group
: In an experiment, the group that does not receive the intervention, or receives a treatment whose effect is known.

natural experiment
: An experiment that uses naturally occurring groups, which can sometimes mimic random assignment.

causal inference
: Methods for identifying and quantifying cause-and-effect relationships.

Exercises

Exercise 7.1

The `thinkstats` module provides a function called `decile_plot` that encapsulates the code from earlier in this chapter. We can call it like this to visualize the relationship between SAT verbal and math scores:

```
from thinkstats import decile_plot

decile_plot(nlsy, "sat_verbal", "sat_math")
decorate(xlabel="SAT Verbal", ylabel="SAT Math")
```

Make a decile plot of PIAT math scores and income. Does it appear to be a linear relationship?

Exercise 7.2

Make a scatter plot of income versus SAT math scores. Compute the Pearson correlation and rank correlation. Are they substantially different?

Make a scatter plot of income versus SAT verbal scores, and compute both correlations. Which is a stronger prediction of future income, math or verbal scores?

Exercise 7.3

Let's see how a student's high school grade point average (GPA) is correlated with their SAT scores. Here's the variable in the NLSY dataset that encodes GPA:

```
missing_codes = [-6, -7, -8, -9]
nlsy["gpa"] = nlsy["R9871900"].replace(missing_codes, np.nan) / 100
nlsy["gpa"].describe()
```

```
count    6004.000000
mean        2.818408
std         0.616357
min         0.100000
25%         2.430000
50%         2.860000
75%         3.260000
max         4.170000
Name: gpa, dtype: float64
```

Make a scatter plot that shows the relationship between GPA and SAT math scores and compute the correlation coefficient. Do the same for the relationship between GPA and SAT verbal scores. Which SAT score is a better predictor of GPA?

Exercise 7.4

Let's investigate the relationship between education and income. The NLSY dataset includes a column that reports the highest degree earned by each respondent. The values are encoded as integers:

```
nlsy["degree"] = nlsy["Z9083900"]
nlsy["degree"].value_counts().sort_index()
```

```
degree
0.0     877
1.0    1167
2.0    3531
3.0     766
4.0    1713
5.0     704
6.0      64
7.0     130
Name: count, dtype: int64
```

But we can use these lists to decode them:

```
positions = [0, 1, 2, 3, 4, 5, 6, 7]
labels = [
    "None",
    "GED",
    "High school diploma",
    "Associate's degree",
    "Bachelor's degree",
    "Master's degree",
    "PhD",
    "Professional degree",
]
```

Make a scatter plot of `income` versus `degree`. To avoid overplotting, jitter the values of `degree` and adjust the marker size and transparency.

Use the `groupby` method to group respondents by `degree`. From the `DataFrame GroupBy` object, select the `income` column; then use the `quantile` method to compute the median, 10th and 90th percentiles in each group. Use `fill_between` to plot the region between the 10th and 90th percentiles, and use `plot` to plot the medians.

What can you say about the income premium associated with each additional degree?

Exercise 7.5

The Behavioral Risk Factor Surveillance System (BRFSS) dataset includes self-reported heights and weights for about 400,000 respondents. Instructions for downloading the data are in the notebook for this chapter.

Make a scatter plot that shows the relationship between height and weight. You might have to jitter the data to blur the visible rows and columns due to rounding. And with such a large sample, you will have to adjust the marker size and transparency to avoid overplotting. Also, because there are outliers in both measurements, you might want to use `xlim` and `ylim` to zoom in on a region that covers most of the respondents.

Make a decile plot of weight versus height. Does the relationship seem to be linear? Compute the correlation coefficient and rank correlation. Are they substantially different? Which one do you think better quantifies the relationship between these variables?

CHAPTER 8
Estimation

Suppose you live in a town with a population of 10,000 people, and you want to predict who will win an upcoming election. In theory, you could ask everyone in town who they plan to vote for, and if they all answered honestly, you could make a reliable prediction.

But even in a small town, it is probably not practical to survey the entire population. Fortunately, is it not necessary. If you survey a random subset of the people, you can use the sample to infer the voting preferences of the population. This process—using a sample to make inferences about a population—is called statistical inference.

Statistical inference includes estimation, which is the topic of this chapter, and hypothesis testing, which is the topic of the next chapter.

Weighing Penguins

Suppose you are a researcher in Antarctica, studying local populations of penguins. One of your tasks is to monitor the average weight of the penguins as it varies over the course of the year. It would be impractical to weigh every penguin in the environment, so your plan is to collect a random sample of 10 penguins each week, weigh them, and use the sample to estimate the mean of the entire population—which is called the **population mean**.

There are many ways you could use the sample to estimate the population mean, but we'll consider just two: the sample mean and the sample median. They are both reasonable choices, but let's see which is better—and think about what we mean by "better."

For purposes of demonstration, we'll assume that penguin weights are drawn from a normal distribution with known mean and standard deviation, which I'll denote as `mu` and `sigma` and assign values in kilograms:

```
mu = 3.7
sigma = 0.46
```

These values are the **parameters** of the normal distribution, which means that they specify a particular distribution. Given these parameters, we can use NumPy to simulate the sampling process and generate a sample of any size. For example, here's a hypothetical sample of 10 weights:

```
sample = np.random.normal(mu, sigma, size=10)
sample
```

```
array([4.44719887, 3.41859205, 3.45704099, 3.20643443, 4.09808751,
       2.6412922 , 4.50261341, 3.34984483, 3.84675798, 3.58528963])
```

And here are the mean and median of the sample:

```
np.mean(sample), np.median(sample)
```

```
(3.6553151902291945, 3.521165310619601)
```

The mean and median are different enough that we should wonder which is a better estimate. To find out, we'll use the following function to generate hypothetical samples with the given size, `n`:

```
def make_sample(n):
    return np.random.normal(mu, sigma, size=n)
```

As a first experiment, let's see how the sample mean and sample median behave as the sample size increases. We'll use the NumPy function `logspace` to make a range of `ns` from 10 to 100,000, equally spaced on a logarithmic scale:

```
ns = np.logspace(1, 5).astype(int)
```

We can use a list comprehension to generate a hypothetical sample for each value of `n`, compute the mean, and collect the results:

```
means = [np.mean(make_sample(n)) for n in ns]
```

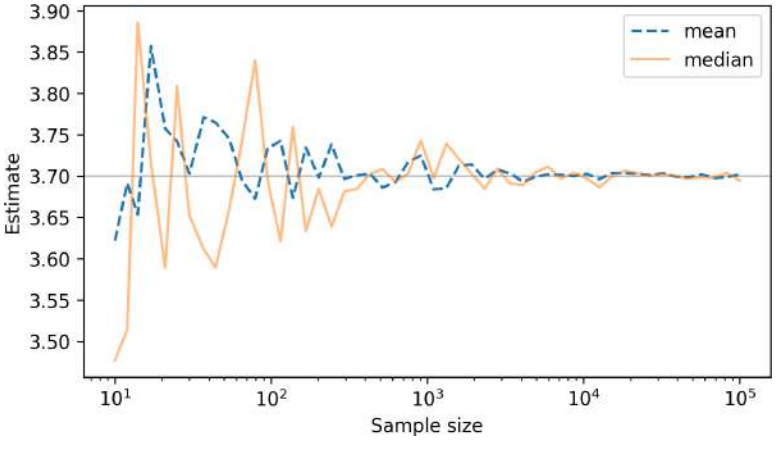
And we'll do the same for the median:

```
medians = [np.median(make_sample(n)) for n in ns]
```

A statistic, like the sample mean or median, that's used to estimate a property of a population is called an **estimator**.

The following figure shows how these estimators behave as we increase the sample size. The horizontal line shows the actual mean in the population:

```
plt.axhline(mu, color="gray", lw=1, alpha=0.5)
plt.plot(ns, means, "--", label="mean")
plt.plot(ns, medians, alpha=0.5, label="median")

decorate(xlabel="Sample size", xscale="log", ylabel="Estimate")
```



For both estimators, the estimates converge to the actual value as the sample size increases. This demonstrates that they are **consistent**, which is one of the properties a good estimator should have. Based on this property, the mean and median seem equally good.

In the previous figure, you might notice that the estimates are sometimes too high and sometimes too low—and it looks like the variation is roughly symmetric around the true value. That suggests another experiment: if we collect many samples with the same size and compute many estimates, what is the average of the estimates?

The following loop simulates this scenario by generating 10,001 samples of 10 penguins and computing the mean of each sample:

```
means = [np.mean(make_sample(n=10)) for i in range(10001)]
np.mean(means)
```

```
3.70034508492869
```

The average of the means is close to the actual mean we used to generate the samples: 3.7 kg.

The following loop simulates the same scenario, but this time it computes the median of each sample:

```
medians = [np.median(make_sample(n=10)) for i in range(10001)]
np.mean(medians)
```

```
3.701214089907223
```

The average of these hypothetical medians is also very close to the actual population mean.

These results demonstrate that the sample mean and median are **unbiased** estimators, which means that they are correct on average. The word "bias" means different things in different contexts, which can be a source of confusion. In this context, "unbiased" means that the average of the estimates is the actual value.

So far, we've shown that both estimators are consistent and unbiased, but it's still not clear which is better. Let's try one more experiment: let's see which estimator is more accurate. The word "accurate" also means different things in different contexts—as one way to quantify it, let's consider the **mean squared error** (MSE). The following function computes the differences between the estimates and the actual value, and returns the mean of the squares of these errors:

```
def mse(estimates, actual):
    """Mean squared error of a sequence of estimates."""
    errors = np.asarray(estimates) - actual
    return np.mean(errors**2)
```

Notice that we can only compute the MSE if we know the actual value. In practice, we usually don't—after all, if we knew the actual value, we wouldn't have to estimate it. But in our experiment, we know that the actual population mean is 3.7 kg, so we can use it to compute the MSE of the sample means:

```
mse(means, mu)
```

```
0.020871984891289382
```

If we have samples with size 10 and we use the sample mean to estimate the population mean, the average squared error is about 0.021 kilograms squared. Now here's the MSE of the sample medians:

```
mse(medians, mu)
```

```
0.029022273128644173
```

If we use the sample medians to estimate the population mean, the average squared error is about 0.029 kilograms squared. In this example, the sample mean is better than the sample median; and in general, if the data are drawn from a normal

distribution, it is the *best* unbiased estimator of the population mean, in the sense that it minimizes the MSE.

Minimizing the MSE is a good property for an estimator to have, but the MSE is not always the best way to summarize errors. For one thing, it is hard to interpret. In this example, the units of MSE are kilograms squared, so it's hard to say what that means.

One solution is to use the square root of the MSE, called "root mean squared error," or RMSE. Another option is to use the average of the absolute values of the errors, called the "mean absolute error" or MAE. The following function computes the MAE for a sequence of estimates:

```
def mae(estimates, actual):
    """Mean absolute error of a sequence of estimates."""
    errors = np.asarray(estimates) - actual
    return np.mean(np.abs(errors))
```

Here's the MAE of the sample means:

```
mae(means, mu)
```

```
0.11540433749505272
```

And the sample medians:

```
mae(medians, mu)
```

```
0.13654429774596036
```

On average, we expect the sample mean to be off by about 0.115 kg, and the sample median to be off by 0.137 kg. So the sample mean is probably the better choice, at least for this example.

Robustness

Now let's consider a different scenario. Suppose that 2% of the time, when you try to weigh a penguin, it accidentally presses the units button on the scale and the weight gets recorded in pounds instead of kilograms. Assuming that the error goes unnoticed, it introduces an outlier in the sample.

The following function simulates this scenario, multiplying 2% of the weights by the conversion factor 2.2 pounds per kilogram:

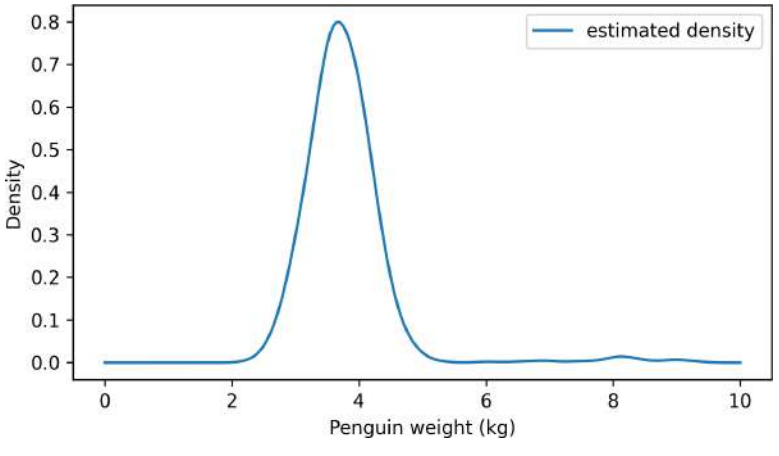
```
def make_sample_with_errors(n):
    sample = np.random.normal(mu, sigma, size=n)
    factor = np.random.choice([1, 2.2], p=[0.98, 0.02], size=n)
    return sample * factor
```

To see what effect this has on the distribution, we'll generate a large sample:

```
sample = make_sample_with_errors(n=1000)
```

To plot the distribution of the sample, we'll use KDE and the `Pdf` object from "Kernel Density Estimation" on page 97:

```
from scipy.stats import gaussian_kde
from thinkstats import Pdf

kde = gaussian_kde(sample)
domain = 0, 10
pdf = Pdf(kde, domain)
pdf.plot(label='estimated density')
decorate(xlabel="Penguin weight (kg)", ylabel="Density")
```



In addition to the mode near 3.7 kg, the measurement errors introduce a second mode near 8 kilograms.

Now let's repeat the previous experiment, simulating many samples with size 10, computing the mean of each sample, and then computing the average of the sample means:

```
means = [np.mean(make_sample_with_errors(n=10)) for i in range(10001)]
np.mean(means)
```

```
3.786352945690677
```

The measurement errors cause the sample mean to be higher, on average, than 3.7 kg.

Now here's the same experiment using sample medians:

```
medians = [np.median(make_sample_with_errors(n=10)) for i in range(10001)]
np.mean(medians)
```

```
3.7121869836715353
```

The average of the sample medians is also higher than 3.7 kg, but it is not off by nearly as much. If we compare the MSE of the estimates, we see that the sample medians are substantially more accurate:

```
mse(means, mu), mse(medians, mu)
```

```
(0.06853430354724438, 0.031164467796883758)
```

If measurements actually come from a normal distribution, the sample mean minimizes MSE, but this scenario violates that assumption, so the sample mean doesn't minimize MSE. The sample median is less sensitive to outliers, so it is less biased and its MSE is smaller. Estimators that deal well with outliers—and similar violations of assumptions—are said to be **robust**.

Estimating Variance

As another example, suppose we want to estimate variance in the penguins' weights. In "Summary Statistics" on page 11, we saw that there are two ways to compute the variance of a sample. I promised to explain the difference later—and later is now.

The reason there are two ways to compute the variance of a sample is that one is a biased estimator of the population variance, and the other is unbiased. The following function computes the biased estimator, which is the sum of the squared deviations divided by `n`:

```
def biased_var(xs):
    # Compute variance with n in the denominator
    n = len(xs)
    deviations = xs - np.mean(xs)
    return np.sum(deviations**2) / n
```

To test it, we'll simulate many samples with size 10, compute the biased variance of each sample, and then compute the average of the variances:

```
biased_vars = [biased_var(make_sample(n=10)) for i in range(10001)]
np.mean(biased_vars)
```

```
0.19049277659404473
```

The result is about 0.19, but in this case, we know that the actual population variance is about 0.21, so this version of the sample variance is too low on average—which confirms that it is biased.

The following function computes the unbiased estimator, which is the sum of the squared deviations divided by `n-1`:

```
def unbiased_var(xs):
    # Compute variance with n-1 in the denominator
    n = len(xs)
    deviations = xs - np.mean(xs)
    return np.sum(deviations**2) / (n - 1)
```

We can test it by generating many samples and computing the unbiased variance for each one:

```
unbiased_vars = [unbiased_var(make_sample(n=10)) for i in range(10001)]
np.mean(unbiased_vars)
```

```
0.21159109492300626
```

The average of the unbiased sample variances is very close to the actual value—which is what we expect if it is unbiased.

With sample size 10, the difference between the biased and unbiased estimators is about 10%, which might not be negligible. With sample size 100, the difference is only 1%, which is small enough that it probably doesn't matter in practice.

Sampling Distributions

So far we've been working with simulated data, assuming that penguin weights are drawn from a normal distribution with known parameters. Now let's see what happens with real data.

Between 2007 and 2010, researchers at Palmer Station in Antarctica measured and weighed 342 penguins from local populations. The data they collected is freely available—instructions for downloading it are in the notebook for this chapter.

We can use Pandas to read the data:

```
penguins = pd.read_csv("penguins_raw.csv").dropna(subset=["Body Mass (g)"])
penguins.shape
```

```
(342, 17)
```

The dataset includes three penguin species:

```
penguins["Species"].value_counts()
```

```
Species
Adelie Penguin (Pygoscelis adeliae)         151
Gentoo penguin (Pygoscelis papua)           123
Chinstrap penguin (Pygoscelis antarctica)    68
Name: count, dtype: int64
```

For the first example we'll select just the Chinstrap penguins:

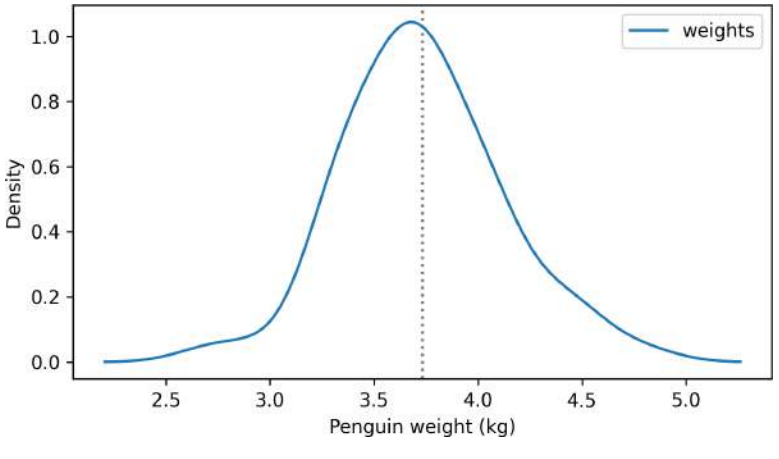
```
chinstrap = penguins.query('Species.str.startswith("Chinstrap")')
```

We'll use this function to plot estimated PDFs:

```
def plot_kde(sample, name="estimated density", **options):
    kde = gaussian_kde(sample)
    m, s = np.mean(sample), np.std(sample)
    plt.axvline(m, color="gray", ls=":")

    domain = m - 4 * s, m + 4 * s
    pdf = Pdf(kde, domain, name)
    pdf.plot(**options)
```

Here's the distribution of chinstrap penguin weights in kilograms. The vertical dotted line shows the sample mean:

```
weights = chinstrap["Body Mass (g)"] / 1000
plot_kde(weights, "weights")
decorate(xlabel="Penguin weight (kg)", ylabel="Density")
```



The sample mean is about 3.7 kg:

```
sample_mean = np.mean(weights)
sample_mean
```

```
3.733088235294118
```

If you are asked to estimate the population mean, 3.7 kg is a reasonable choice—but how precise is that estimate?

One way to answer that question is to compute the **sampling distribution** of the mean, which shows how much the estimated mean varies from one sample to another. If we knew the actual mean and standard deviation in the population, we could model the sampling process and compute the sampling distribution. But if we knew the actual population mean, we wouldn't have to estimate it!

Fortunately, there's a simple way to approximate the sampling distribution, called **resampling**. The core idea is to use the sample to make a model of the population, then use the model to simulate the sampling process.

More specifically, we'll use **parametric resampling**, which means we'll use the sample to estimate the parameters of the population and then use a theoretical distribution to generate new samples.

The following function implements this process with a normal distribution. Notice that the new samples are the same size as the original:

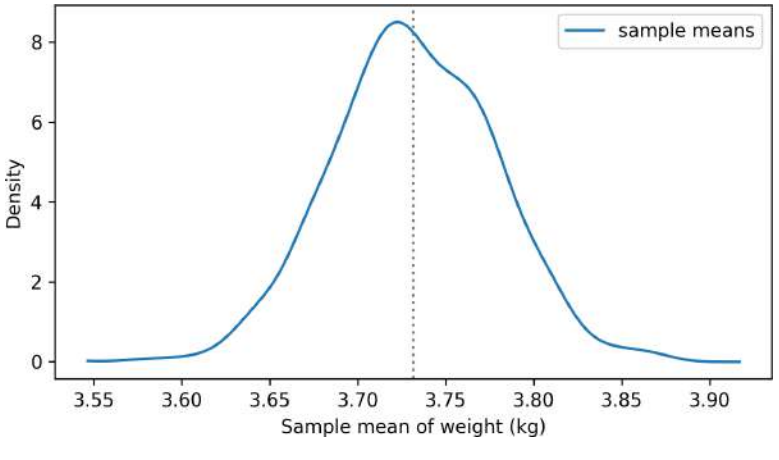
```
def resample(sample):
    m, s = np.mean(sample), np.std(sample)
    return np.random.normal(m, s, len(sample))
```

This loop uses `resample` to generate many samples and compute the mean of each one:

```
sample_means = [np.mean(resample(weights)) for i in range(1001)]
```

The following figure shows the distribution of these sample means:

```
plot_kde(sample_means, "sample means")
decorate(xlabel="Sample mean of weight (kg)", ylabel="Density")
```



This result approximates the sampling distribution of the sample mean. It shows how much we expect the sample mean to vary if we collect many samples of the same size—assuming that our model of the population is accurate.

Informally, we can see that the sample mean could be as low as 3.55, if we collected another sample with the same size, or as high as 3.9.

Standard Error

To quantify the width of the sampling distribution, one option is to compute its standard deviation—the result is called the **standard error**:

```
standard_error = np.std(sample_means)
standard_error
```

```
0.04626531069684985
```

In this case, the standard error is about 0.045 kg—so if we collect many samples, we expect the sample means to vary by about 0.045 kg, on average.

People often confuse standard error and standard deviation. Remember:

- Standard deviation quantifies variation in measurements.
- Standard error quantifies the precision of an estimate.

In this dataset, the standard deviation of penguin weights is about 0.38 kg for chinstrap penguins:

```
np.std(weights)
```

```
0.3814986213564681
```

The standard error of the average weight is about 0.046 kg:

```
np.std(sample_means)
```

```
0.04626531069684985
```

Standard deviation tells you how much penguins differ in weight. Standard error tells you how precise an estimate is. They are answers to different questions.

However, there is a relationship between them. If we know the standard deviation and sample size, we can approximate the standard error of the means like this:

```
def approximate_standard_error(sample):
    n = len(sample)
    return np.std(sample) / np.sqrt(n)
```

```
approximate_standard_error(weights)
```

```
0.046263503290595163
```

This result is close to what we got by resampling.

Confidence Intervals

Another way to summarize the sampling distribution is to compute a **confidence interval**. For example, a 90% confidence interval contains 90% of the values in the sampling distribution, which we can find by computing the 5th and 95th percentiles. Here's the 90% confidence interval for the average weight of chinstrap penguins:

```
ci90 = np.percentile(sample_means, [5, 95])
ci90
```

```
array([3.6576334 , 3.80737506])
```

To interpret a confidence interval, it is tempting to say that there is a 90% chance that the true value of the population parameter falls in the 90% confidence interval. In this example, we would say there is a 90% chance that the population mean for chinstrap penguins is between 3.66 and 3.81 kg.

Under a strict philosophy of probability called **frequentism**, this interpretation would not be allowed, and in many statistics books, you will be told that this interpretation is wrong.

In my opinion, this prohibition is unnecessarily strict. Under reasonable philosophies of probability, a confidence interval means what people expect it to mean: there is a 90% chance that the true value falls in the 90% confidence interval.

However, confidence intervals only quantify variability due to sampling—that is, measuring only part of the population. The sampling distribution does not account for other sources of error, notably sampling bias and measurement error, which are the topics of the next section.

Sources of Error

Suppose that instead of the average weight of penguins in Antarctica, you want to know the average weight of women in the city where you live. You can't randomly choose a representative sample of women and weigh them.

A simple alternative would be "telephone sampling"—that is, you could choose random numbers from the phone book, call and ask to speak to an adult woman, and ask how much she weighs. But telephone sampling has obvious problems.

For example, the sample is limited to people whose telephone numbers are listed, so it eliminates people without phones (who might be poorer than average) and people with unlisted numbers (who might be richer). Also, if you call home telephones during the day, you are less likely to sample people with jobs. And if you only sample the person who answers the phone, you are less likely to sample people who share a phone line.

If factors like income, employment, and household size are related to weight—and it is plausible that they are—the results of your survey would be affected one way or another. This problem is called **sampling bias** because it is a property of the sampling process.

This sampling process is also vulnerable to self-selection, which is a kind of sampling bias. Some people will refuse to answer the question, and if the tendency to refuse is related to weight, that would affect the results.

Finally, if you ask people how much they weigh, rather than weighing them, the results might not be accurate. Even helpful respondents might round up or down if they are uncomfortable with their actual weight. And not all respondents are helpful. These inaccuracies are examples of **measurement error**.

When you report an estimated quantity, it is useful to quantify variability due to sampling by reporting a standard error or a confidence interval. But remember that this variability is only one source of error, and often it is not the biggest.

Glossary

population mean
: The true mean of a quantity in an entire population, as opposed to the sample mean, which is calculated from a subset.

parameter
: One of the values that specify a particular distribution in a set of distributions—for example, the parameters of a normal distribution are the mean and standard deviation.

estimator
: A statistic calculated from a sample that is used to estimate a parameter of a population.

consistent
: An estimator is consistent if it converges to the actual value of the parameter as the sample size increases.

unbiased
: An estimator is unbiased if, for a particular sample size, the average of the sample estimates is the actual value of the parameter.

mean squared error (MSE)
: A measure of the accuracy of an estimator—it's the average squared difference between estimated and true parameter values, assuming the true value is known.

robust
: An estimator is robust if it remains accurate even when a dataset contains outliers or errors—or does not perfectly follow a theoretical distribution.

resampling
: A way to approximate the sampling distribution of an estimate by simulating the sampling process.

sampling distribution
: The distribution of a statistic across possible samples from the same population.

parametric resampling
: A kind of resampling that estimates population parameters from sample data and then uses a theoretical distribution to simulate the sampling process.

standard error
: The standard deviation of a sampling distribution, which quantifies the variability of an estimate due to random sampling (but not measurement error or nonrepresentative sampling).

confidence interval
: An interval that contains the most likely values in a sampling distribution.

sampling bias
: A flaw in the way a sample is collected that makes it unrepresentative of the population.

measurement error
: Inaccuracy in how data are observed, measured, or recorded.

Exercises

Exercise 8.1

One of the strengths of resampling methods is that they are easy to extend to other statistics. In this chapter, we computed the sample mean of penguin weights and then used resampling to approximate the sampling distribution of the mean. Now let's do the same for standard deviation.

Compute the sample standard deviation of weights for chinstrap penguins. Then use `resample` to approximate the sampling distribution of the standard deviation. Use the sampling distribution to compute the standard error of the estimate and a 90% confidence interval.

Exercise 8.2

The Behavioral Risk Factor Surveillance System (BRFSS) dataset includes self-reported heights and weights for a sample of adults in the United States. Use this data to estimate the average height of male adults. Use resample to approximate the sampling distribution and compute a 90% confidence interval.

Because the sample size is very large, the confidence interval is very small, which means that variability due to random sampling is small. But other sources of error might be bigger—what other sources of error do you think affect the results?

Exercise 8.3

In games like soccer and hockey, the time between goals tends to follow an exponential distribution (as we saw in "The Exponential PDF" on page 93). Suppose we observe a sample of times between goals. If we assume that the sample came from an exponential distribution, how can we estimate the actual mean of the distribution? We might consider using either the sample mean or the sample median. Let's see if either of them is a consistent, unbiased estimator. For the experiments, we'll assume that the actual mean time between goals is 10 minutes:

```
actual_mean = 10
```

The following function generates a sample from an exponential distribution with this mean and the given sample size:

```
def make_exponential(n):
    return np.random.exponential(actual_mean, size=n)
```

Use this function to generate samples with a range of sizes and compute the mean of each one. As `n` increases, do the sample means converge to the actual mean?

Next, generate samples with a range of sizes and compute the median of each one. Do the sample medians converge to the actual median?

Next, generate many samples with size 10 and check whether the sample mean is an unbiased estimator of the population mean.

Finally, check whether the sample median is an unbiased estimator of the population median.

Exercise 8.4

In this chapter we tested a biased estimator of variance and showed that it is, in fact, biased. And we showed that the unbiased estimator is unbiased. Now let's try standard deviation.

To estimate the standard deviation of a population, we can compute the square root of the biased or unbiased estimator of variance, like this:

```
def biased_std(sample):
    # Square root of the biased estimator of variance
    var = biased_var(sample)
    return np.sqrt(var)
```

```
def unbiased_std(sample):
    # Square root of the unbiased estimator of variance
    var = unbiased_var(sample)
    return np.sqrt(var)
```

Use `make_sample` to compute many samples of size 10 from a normal distribution with mean 3.7 and standard deviation 0.46. Check whether either of these is an unbiased estimator of standard deviation.

Exercise 8.5

This exercise is based on the German tank problem, which is a simplified version of an actual analysis performed by the Economic Warfare Division of the American Embassy in London during World War II.

Suppose you are an Allied spy and your job is to estimate how many tanks the Germans have built. As data, you have serial numbers recovered from `k` captured tanks.

If we assume that the Germans have `N` tanks numbered from 1 to `N`, and that all tanks in this range were equally likely to be captured, we can estimate `N` like this:

```
def estimate_tanks(sample):
    m = np.max(sample)
    k = len(sample)
    return m + (m - k) / k
```

As an example, suppose `N` is 122:

```
N = 122
tanks = np.arange(1, N + 1)
```

We can use the following function to generate a random sample of `k` tanks:

```
def sample_tanks(k):
    return np.random.choice(tanks, replace=False, size=k)
```

Here's an example:

```
sample = sample_tanks(5)
sample
```

```
array([74, 71, 95, 10, 17])
```

And here is the estimate based on this sample:

```
estimate_tanks(sample)
```

```
113.0
```

Check whether this estimator is biased.

Exercise 8.6

In several sports—especially basketball—many players and fans believe in a phenomenon called the "hot hand," which implies that a player who has hit several consecutive shots is more likely to hit the next, and a player who has missed several times is more likely to miss.

A famous paper proposed a way to test whether the hot hand is real or an illusion by looking at sequences of hits and misses from professional basketball games.[1] For each player, the authors computed the overall probability of making a shot, and the conditional probability of making a shot after three consecutive hits. For eight out of nine players, they found that the probability of making a shot was *lower* after three hits. Based on this and other results, they concluded that there is "no evidence for a positive correlation between the outcomes of successive shots." And for several decades, many people believed that the hot hand had been debunked.

However, this conclusion is based on a statistical error, at least in part. A 2018 paper showed that the statistic used in the first paper—the probability of making a shot

1 T. Gilovich, R. Vallone, and A. Tversky, "The Hot Hand in Basketball: On the Misperception of Random Sequences," *Cognitive Psychology* 17(3): 295-314.

after three hits—is biased.[2] Even if the probability of making every shot is exactly 0.5, and there is actually no correlation between the outcomes, the probability of making a shot after three hits is *less than 0.5*.

It is not obvious why that's true, which is why the error went undetected for so long, and I won't try to explain it here. But we can use the methods from this chapter to check it. We'll use the following function to generate a sequence of 0s and 1s with probability 0.5 and no correlation:

```
def make_hits_and_misses(n):
    # Generate a random sequence of 0s and 1s
    return np.random.choice([0, 1], size=n)
```

In the notebook for this chapter, I provide a function that finds all subsequences of three hits (1s) and returns the element of the sequence that follows.

Generate a large number of sequences with length 100 and for each sequence, find each shot that follows three hits. Compute the percentage of these shots that are hits. Hint: if the sequence does not contain three consecutive hits, the function returns an empty sequence, so your code will have to handle that.

If you run this simulation many times, what is the average percentage of hits? How does this result vary as you increase or decrease the length of the sequence?

2 J. B. Miller and A. Sanjurjo, "Surprised by the Hot Hand Fallacy? A Truth in the Law of Small Numbers," *Econometrica* 86(6): 2019-2047.

CHAPTER 9

Hypothesis Testing

In the datasets we have explored in this book, we've seen differences between groups of people—and penguins—correlations between variables, and slopes of regression lines. Results like these are called **observed effects** because they appear in a sample, as contrasted with actual effects in the population, which we usually can't observe directly. When we see an apparent effect, we should consider whether it is likely to be present in the larger population or whether it might appear in the sample by chance.

There are several ways to formulate this question, including Fisher null hypothesis testing, Neyman-Pearson decision theory, and Bayesian hypothesis testing. What I present here is a mixture of these approaches that is often used in practice.

Flipping Coins

We'll start with a simple example:[1] When Euro coins were introduced in 2002, a curious coin enthusiast spun a Belgian one-Euro coin on edge 250 times and noted that it landed with the heads side up 140 times and tails side up 110 times. If the coin is perfectly balanced, we expect only 125 heads, so this data suggests the coin is biased. On the other hand, we don't expect to get exactly 125 heads every time, so it's possible that the coin is actually fair, and the apparent deviation from the expected value is due to chance. To see whether that's plausible, we can perform a hypothesis test.

1 Based on an example in D. J. MacKay, *Information Theory, Inference and Learning Algorithms* (Cambridge University Press, 2003).

We'll use the following function to compute the absolute difference between the observed number and the expected number if the coin is fair:

```
n = 250
p = 0.5

def abs_deviation(heads):
    expected = n * p
    return np.abs(heads - expected)
```

In the observed data, this deviation is 15:

```
heads = 140
tails = 110

observed_stat = abs_deviation(heads)
observed_stat
```

```
15.0
```

If the coin is actually fair, we can simulate the coin-spinning experiment by generating a sequence of random strings—either `'H'` or `'T'` with equal probability—and counting the number of times `'H'` appears:

```
def simulate_flips():
    flips = np.random.choice(["H", "T"], size=n)
    heads = np.sum(flips == "H")
    return heads
```

Each time we call this function, we get the outcome of a simulated experiment:

```
simulate_flips()
```

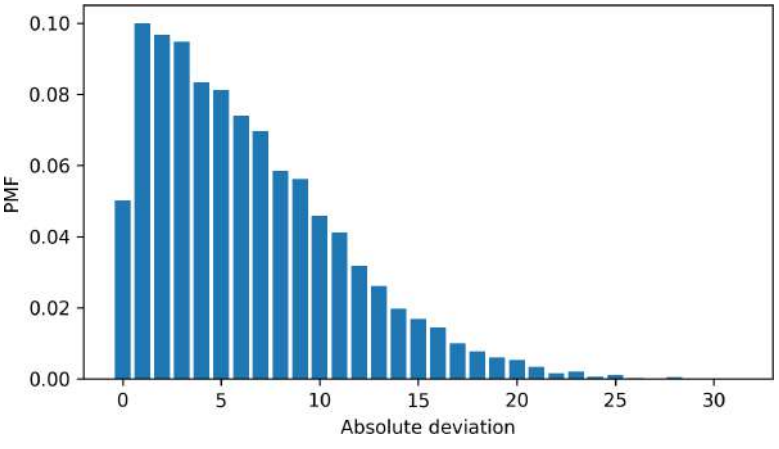
```
119
```

The following loop simulates the experiment many times, computes the deviation for each one, and uses a list comprehension to collect the results in a list:

```
simulated_stats = [abs_deviation(simulate_flips()) for i in range(10001)]
```

The result is a sample from the distribution of deviations under the assumption that the coin is fair. Here's what the distribution of these values looks like:

```
from empiricaldist import Pmf

pmf_effects = Pmf.from_seq(simulated_stats)
pmf_effects.bar()

decorate(xlabel="Absolute deviation", ylabel="PMF")
```



Values near 0 are the most common; values greater than 10 are less common. Remembering that the deviation in the observed data is 15, we see that deviations of that magnitude are rare, but not impossible. In this example, the simulated results equal or exceed 15 about 7.1% of the time:

```
(np.array(simulated_stats) >= 15).mean() * 100
```

```
7.079292070792921
```

So, if the coin is fair, we expect a deviation as big as the one we saw about 7.1% of the time, just by chance.

We can conclude that an effect of this size is not common, but it is certainly not impossible, even if the coin is fair. On the basis of this experiment, we can't rule out the possibility that the coin is fair.

This example demonstrates the logic of statistical hypothesis testing:

- We started with an observation, 140 heads out of 250 spins, and the hypothesis that the coin is biased—that is, that the probability of heads differs from 50%.
- We chose a **test statistic** that quantifies the size of the observed effect. In this example, the test statistic is the absolute deviation from the expected outcome.
- We defined a **null hypothesis**, which is a model based on the assumption that the observed effect is due to chance. In this example, the null hypothesis is that the coin is fair.
- Next, we computed a **p-value**, which is the probability of seeing the observed effect if the null hypothesis is true. In this example, the p-value is the probability of a deviation as big as 15 or bigger.

The last step is to interpret the result. If the p-value is small, we conclude that the effect would be unlikely to happen by chance. If it is large, we conclude that the effect could plausibly be explained by chance. And if it falls somewhere in the middle, as in this example, we can say that the effect is unlikely to happen by chance, but we can't rule out the possibility.

All hypothesis tests are based on these elements—a test statistic, a null hypothesis, and a p-value.

Testing a Difference in Means

In the NSFG data, we saw that the average pregnancy length for first babies is slightly longer than for other babies. Now let's see if that difference could be due to chance.

The function `get_nsfg_groups` reads the data, selects live births, and groups live births into first babies and others:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

Now we can select pregnancy lengths, in weeks, for both groups:

```
data = firsts["prglngth"].values, others["prglngth"].values
```

The following function takes the data as a tuple of two sequences, and computes the absolute difference in means:

```
def abs_diff_means(data):
    group1, group2 = data
    diff = np.mean(group1) - np.mean(group2)
    return np.abs(diff)
```

Between first babies and others, the observed difference in pregnancy length is 0.078 weeks:

```
observed_diff = abs_diff_means(data)
observed_diff
```

```
0.07803726677754952
```

So the hypothesis we'll test is whether pregnancy length is generally longer for first babies. The null hypothesis is that pregnancy lengths are actually the same for both groups, and the apparent difference is due to chance. If pregnancy lengths are the same for both groups, we can combine the two groups into a single pool. To simulate the experiment, we can use the NumPy function `shuffle` to put the pooled values in

random order, and then use slice indexes to select two groups with the same sizes as the original:

```
def simulate_groups(data):
    group1, group2 = data
    n, m = len(group1), len(group2)

    pool = np.hstack(data)
    np.random.shuffle(pool)
    return pool[:n], pool[-m:]
```

Each time we call this function, it returns a tuple of sequences, which we can pass to `abs_diff_means`:

```
abs_diff_means(simulate_groups(data))
```

```
0.031193045602279312
```

The following loop simulates the experiment many times and computes the difference in means for each simulated dataset:

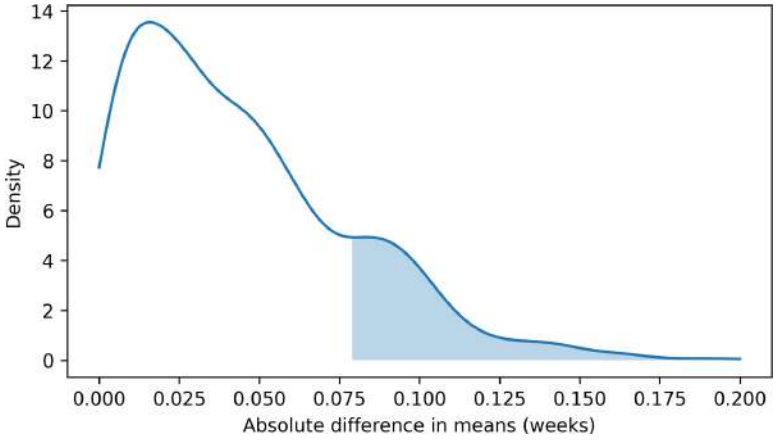
```
simulated_diffs = [abs_diff_means(simulate_groups(data)) for i in range(1001)]
```

To visualize the results, we'll use the following function, which takes a sample of simulated results and makes a `Pmf` object that approximates its distribution:

```
from scipy.stats import gaussian_kde
from empiricaldist import Pmf


def make_pmf(sample, low, high):
    kde = gaussian_kde(sample)
    qs = np.linspace(low, high, 201)
    ps = kde(qs)
    return Pmf(ps, qs)
```

Here's what the distribution of the simulated results looks like. The shaded region shows the cases where the difference in means under the null hypothesis exceeds the observed difference. The area of this region is the p-value:

```
from thinkstats import fill_tail

pmf = make_pmf(simulated_diffs, 0, 0.2)
pmf.plot()
fill_tail(pmf, observed_diff, "right")
decorate(xlabel="Absolute difference in means (weeks)", ylabel="Density")
```



The following function computes the p-value, which is the fraction of simulated values that are as big as or bigger than the observed value:

```
def compute_p_value(simulated, observed):
    """Fraction of simulated values as big or bigger than the observed value."""
    return (np.asarray(simulated) >= observed).mean()
```

In this example, the p-value is about 18%, which means it is plausible that a difference as big as 0.078 weeks could happen by chance:

```
compute_p_value(simulated_diffs, observed_diff)
```

```
0.1838161838161838
```

Based on this result, we can't be sure that pregnancy lengths are generally longer for first babies—it's possible that the difference in this dataset is due to chance.

Notice that we've seen the same elements in both examples of hypothesis testing: a test statistic, a null hypothesis, and a model of the null hypothesis. In this example, the test statistic is the absolute difference in the means. The null hypothesis is that the distribution of pregnancy lengths is actually the same in both groups. And we modeled the null hypothesis by combining the data from both groups into a single pool, shuffling the pool, and splitting it into two groups with the same sizes as the originals. This process is called **permutation**, which is another word for shuffling.

This computational approach to hypothesis testing makes it easy to combine these elements to test different statistics.

Other Test Statistics

We might wonder whether pregnancy lengths for first babies are not just longer, but maybe more variable. To test that hypothesis, we can use as a test statistic the absolute difference between the standard deviations of the two groups. The following function computes this test statistic:

```
def abs_diff_stds(data):
    group1, group2 = data
    diff = np.std(group1) - np.std(group2)
    return np.abs(diff)
```

In the NSFG dataset, the difference in standard deviations is about 0.18:

```
observed_diff = abs_diff_stds(data)
observed_diff
```

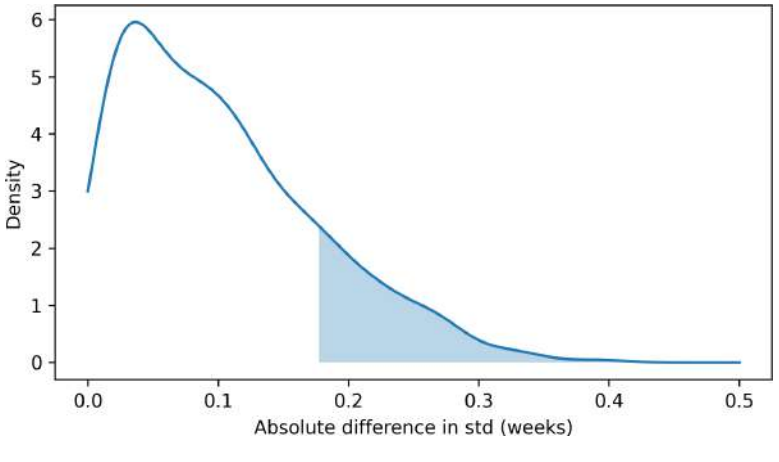
```
0.17600895913991677
```

To see whether this difference might be due to chance, we can use permutation again. The following loop simulates the null hypothesis many times and computes the difference in standard deviation for each simulated dataset:

```
simulated_diffs = [abs_diff_stds(simulate_groups(data)) for i in range(1001)]
```

Here's what the distribution of the results looks like. Again, the shaded region shows where the test statistic under the null hypothesis exceeds the observed difference:

```
pmf = make_pmf(simulated_diffs, 0, 0.5)
pmf.plot()
fill_tail(pmf, observed_diff, "right")
decorate(xlabel="Absolute difference in std (weeks)", ylabel="Density")
```



We can estimate the area of this region by computing the fraction of results that are as big as or bigger than the observed difference:

```
compute_p_value(simulated_diffs, observed_diff)
```

```
0.17082917082917082
```

The p-value is about 0.17, so it is plausible that we could see a difference this big even if the two groups are the same. In conclusion, we can't be sure that pregnancy lengths are generally more variable for first babies—the difference we see in this dataset could be due to chance.

Testing a Correlation

We can use the same framework to test correlations. For example, in the NSFG data set, there is a correlation between birth weight and mother's age—older mothers have heavier babies, on average. But could this effect be due to chance?

To find out, we'll start by preparing the data. From live births, we'll select cases where the age of the mother and birth weight are known:

```
valid = live.dropna(subset=["agepreg", "totalwgt_lb"])
valid.shape
```

```
(9038, 244)
```

Then we'll select the relevant columns:

```
ages = valid["agepreg"]
birthweights = valid["totalwgt_lb"]
```

The following function takes a tuple of `xs` and `ys` and computes the magnitude of the correlation, positive or negative:

```
def abs_correlation(data):
    xs, ys = data
    corr = np.corrcoef(xs, ys)[0, 1]
    return np.abs(corr)
```

In the NSFG dataset, the correlation is about 0.07:

```
data = ages, birthweights
observed_corr = abs_correlation(data)
observed_corr
```

```
0.0688339703541091
```

The null hypothesis is that there is no correlation between mother's age and birth weight. By shuffling the observed values, we can simulate a world where the distributions of age and birth weight are the same, but where the variables are unrelated.

The following function takes a tuple of `xs` and `ys`, shuffles `xs` and returns a tuple containing the shuffled `xs` and the original `ys`. It would also work if we shuffled the `ys` instead, or shuffled both:

```
def permute(data):
    xs, ys = data
    new_xs = xs.values.copy()
    np.random.shuffle(new_xs)
    return new_xs, ys
```

The correlation of the shuffled values is usually close to 0:

```
abs_correlation(permute(data))
```

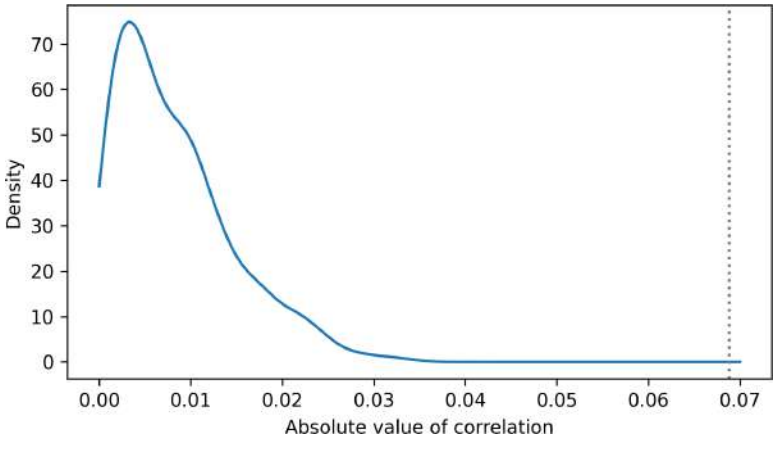
```
0.0019269515502894237
```

The following loop generates many shuffled datasets and computes the correlation of each one:

```
simulated_corrs = [abs_correlation(permute(data)) for i in range(1001)]
```

Here's what the distribution of the results looks like. The vertical dotted line shows the observed correlation:

```
pmf = make_pmf(simulated_corrs, 0, 0.07)
pmf.plot()
plt.axvline(observed_corr, color="gray", ls=":")
decorate(xlabel="Absolute value of correlation", ylabel="Density")
```



We can see that the observed correlation is in the tail of the distribution, with no visible area under the curve. If we try to compute a p-value, the result is 0, indicating that the correlation in the shuffled data did not exceed the observed value in any of the simulations:

```
compute_p_value(simulated_corrs, observed_corr)
```

```
0.0
```

Based on this calculation, we can conclude that the p-value is probably less that 1 per 1,000, but it is not actually zero. It is unlikely for the correlation of the shuffled data to exceed the observed value—but it is not impossible.

When the p-value is small, traditionally less than 0.05, we can say that the result is **statistically significant**. But this way of interpreting p-values has always been problematic, and it is slowly becoming less widely used.

One problem is that the traditional threshold is arbitrary and not appropriate for all applications. Another problem is that this use of "significant" is misleading because it suggests that the effect is important in practice. The correlation between mother's age and birth weight is a good example—it is statistically significant, but so small that it is not important.

An alternative is to interpret p-values qualitatively:

- If a p-value is large, it is plausible that the observed effect could happen by chance.
- If the p-value is small, we can often rule out the possibility that the effect is due to chance—but we should remember that it could still be due to nonrepresentative sampling or measurement errors.

Testing Proportions

As a final example, let's consider a case where the choice of the test statistic takes some thought. Suppose you run a casino and you suspect that a customer is using a crooked die—that is, one that has been modified to make one of the faces more likely than the others. You apprehend the alleged cheater and confiscate the die, but now you have to prove that it is crooked. You roll the die 60 times and record the frequency of each outcome from 1 to 6. Here are the results in a `Hist` object:

```
from empiricaldist import Hist

qs = np.arange(1, 7)
freqs = [8, 9, 19, 5, 8, 11]
observed = Hist(freqs, qs)
observed.index.name = "outcome"
observed
```

freqs	outcome
1	8
2	9
3	19
4	5
5	8
6	11

On average you expect each value to appear 10 times. In this dataset, the value 3 appears more often than expected, and the value 4 appears less often. But could these differences happen by chance?

To test this hypothesis, we'll start by computing the expected frequency for each outcome:

```
num_rolls = observed.sum()
outcomes = observed.qs
expected = Hist(num_rolls / 6, outcomes)
```

The following function takes the observed and expected frequencies and computes the sum of the absolute differences:

```
def total_abs_deviation(observed):
    return np.sum(np.abs(observed - expected))
```

In the observed dataset, this test statistic is 20:

```
observed_dev = total_abs_deviation(observed)
observed_dev
```

```
20.0
```

The following function takes the observed data, simulates rolling a fair die the same number of times, and returns a `Hist` object that contains the simulated frequencies:

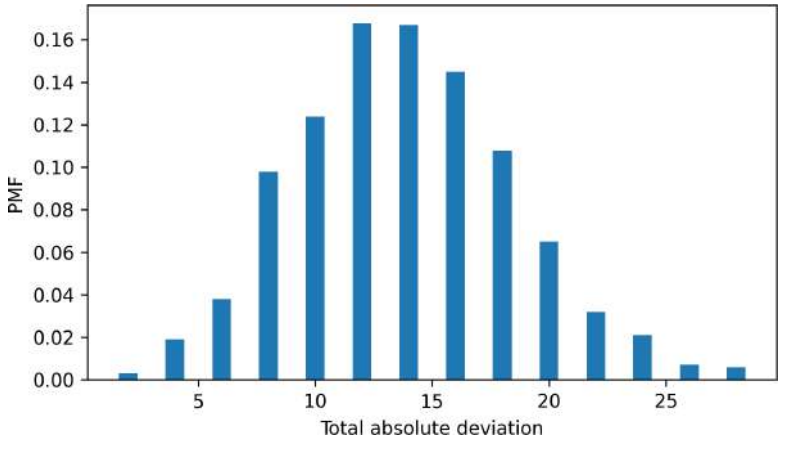
```
def simulate_dice(observed):
    num_rolls = np.sum(observed)
    rolls = np.random.choice(observed.qs, num_rolls, replace=True)
    hist = Hist.from_seq(rolls)
    return hist
```

The following loop simulates the experiment many times and computes the total absolute deviation for each one:

```
simulated_devs = [total_abs_deviation(simulate_dice(observed))
                  for i in range(1001)]
```

Here's what the distribution of the test statistic looks like under the null hypothesis. Notice that the total is always even, because every time an outcome appears more often than expected, another outcome has to appear less often:

```
pmf_devs = Pmf.from_seq(simulated_devs)
pmf_devs.bar()

decorate(xlabel="Total absolute deviation", ylabel="PMF")
```



We can see that a total deviation of 20 is not unusual—the p-value is about 13%, which means that we can't be sure the die is crooked:

```
compute_p_value(simulated_devs, observed_dev)
```

```
0.13086913086913088
```

But the test statistic we chose was not the only option. For a problem like this, it would be more conventional to use the chi-squared statistic, which we can compute like this:

```
def chi_squared_stat(observed):
    diffs = (observed - expected) ** 2
    return np.sum(diffs / expected)
```

Squaring the deviations (rather than taking absolute values) gives more weight to large deviations. Dividing through by `expected` standardizes the deviations—although in this case it has no effect on the results because the expected frequencies are all equal:

```
observed_chi2 = chi_squared_stat(observed)
observed_chi2
```

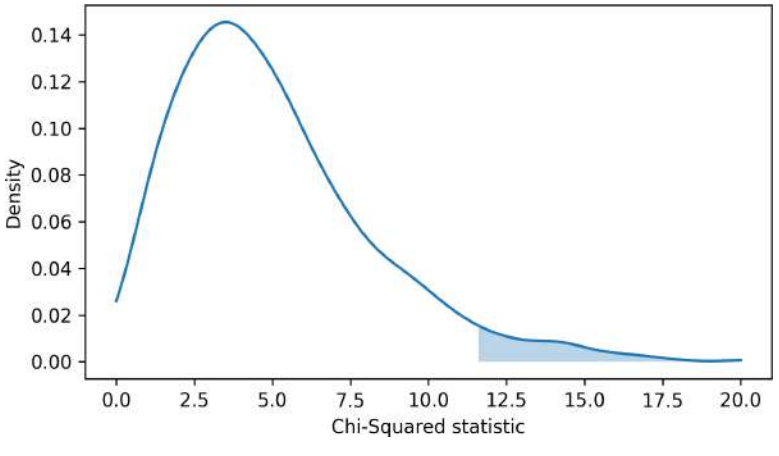
```
11.6
```

The chi-squared statistic of the observed data is 11.6. By itself, this number doesn't mean very much, but we can compare it to the results from the simulated rolls. The following loop generates many simulated datasets and computes the chi-squared statistic for each one:

```
simulated_chi2 = [chi_squared_stat(simulate_dice(observed)) for i in range(1001)]
```

Here's what the distribution of this test statistic looks like under the null hypothesis. The shaded region shows the results that exceed the observed value:

```
pmf = make_pmf(simulated_chi2, 0, 20)
pmf.plot()
fill_tail(pmf, observed_chi2, "right")
decorate(xlabel="Chi-Squared statistic", ylabel="Density")
```



Again, the area of the shaded region is the p-value:

```
compute_p_value(simulated_chi2, observed_chi2)
```

```
0.04495504495504495
```

The p-value using the chi-squared statistic is about 0.04, substantially smaller than what we got using total deviation, 0.13. If we take the 5% threshold seriously, we would consider this effect statistically significant. But considering the two tests together, I would say that the results are inconclusive. I would not rule out the possibility that the die is crooked, but I would not convict the accused cheater.

This example demonstrates an important point: the p-value depends on the choice of test statistic and the model of the null hypothesis, and sometimes these choices determine whether an effect is statistically significant or not.

Glossary

hypothesis testing
: A set of methods for checking whether an observed effect could plausibly be due to random sampling.

test statistic
: A statistic used in a hypothesis test to quantify the size of an observed effect.

null hypothesis
: A model of a system based on the assumption that an effect observed in a sample does not exist in the population.

permutation
: A way to simulate a null hypothesis by randomly shuffling a dataset.

p-value
: The probability of an effect as big as the observed effect, under a null hypothesis.

statistically significant
: An effect is statistically significant if the p-value is smaller than a chosen threshold, often 5%. In a large dataset, an observed effect can be statistically significant even if it is too small to matter in practice.

Exercises

Exercise 9.1

Let's try hypothesis testing with the penguin data from "Sampling Distributions" on page 138. Instructions for downloading the data are in the notebook for this chapter.

Here's how we read the data and select the Chinstrap penguins:

```
penguins = pd.read_csv("penguins_raw.csv").dropna(subset=["Body Mass (g)"])
chinstrap = penguins.query('Species.str.startswith("Chinstrap")')
chinstrap.shape
```

```
(68, 17)
```

And here's how we can extract the weights for male and female penguins in kilograms:

```
male = chinstrap.query("Sex == 'MALE'")
weights_male = male["Body Mass (g)"] / 1000
weights_male.mean()
```

```
3.9389705882352937
```

```
female = chinstrap.query("Sex == 'FEMALE'")
weights_female = female["Body Mass (g)"] / 1000
weights_female.mean()
```

```
3.5272058823529413
```

Use `abs_diff_means` and `simulate_groups` to generate a large number of simulated datasets under the null hypothesis that the two groups have the same distribution of weights, and compute the difference in means for each one. Compare the simulation results to the observed difference and compute a p-value. Is it plausible that the apparent difference between the groups is due to chance?

Exercise 9.2

Using the penguin data from the previous exercise, we can extract the culmen depths and lengths for the female penguins (the culmen is the top ridge of the bill):

```
data = female["Culmen Depth (mm)"], female["Culmen Length (mm)"]
```

The correlation between these variables is about 0.26:

```
observed_corr = abs_correlation(data)
observed_corr
```

```
0.2563170802728449
```

Let's see whether this correlation could happen by chance, even if there is actually no correlation between the measurements. Use `permute` to generate many permutations of this data and `abs_correlation` to compute the correlation for each one. Plot the distribution of the correlations under the null hypothesis and compute a p-value for the observed correlation. How do you interpret the result?

CHAPTER 10
Least Squares

This chapter and the next introduce the idea of fitting a model to data. In this context, a **model** consists of a mathematical description of the relationship between variables—like a straight line—and a description of random variation—like a normal distribution.

When we say that a model fits data, we usually mean that it minimizes errors, which are the distances between the model and the data. We'll start with one of the most widely used ways of fitting a model, minimizing the sum of the squared errors, which is called a least squares fit.

We'll also start with models that work with just two variables at a time. The next chapter introduces models that can handle more than two variables.

Least Squares Fit

As a first example, let's return to the scenario from "Weighing Penguins" on page 131. Suppose you are a researcher in Antarctica, studying local populations of penguins. As part of your data collection, you capture a sample of penguins, measure and weigh them—and then release them unharmed.

As you would soon learn, it can be difficult to get penguins to stay on the scale long enough to get an accurate measurement. Suppose that for some penguins we have measurements like flipper and bill sizes, but no weights. Let's see if we can use the other measurements to fill in the missing data—this process is called **imputation**.

We'll start by exploring the relationship between the weights and measurements, using data collected between 2007 and 2010 by researchers at Palmer Station in Antarctica. The data they collected is freely available—instructions for downloading it are in the notebook for this chapter.

We can use `read_csv` to read the data:

```
penguins = pd.read_csv("penguins_raw.csv").dropna(subset=["Body Mass (g)"])
penguins.shape
```

```
(342, 17)
```

The dataset includes measurements of 151 Adélie penguins. We can use `query` to select the rows that contain this data:

```
adelie = penguins.query('Species.str.startswith("Adelie")')
len(adelie)
```

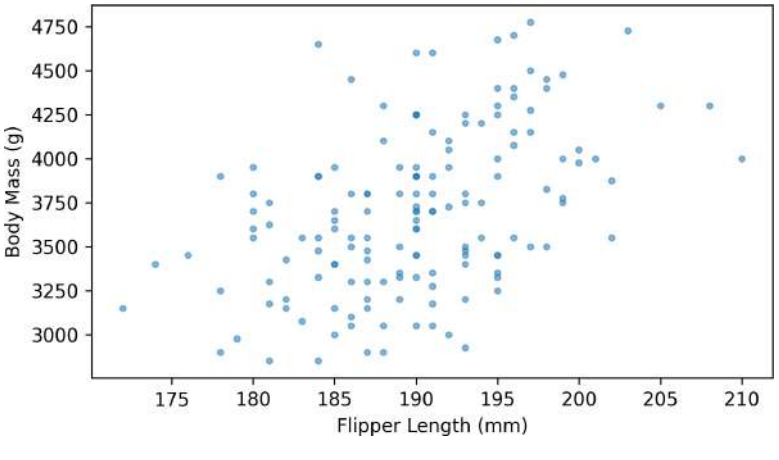
```
151
```

Now suppose we know the flipper length of an Adélie penguin—let's see how well we can predict its weight. First we'll select these columns from the `DataFrame`:

```
xvar = "Flipper Length (mm)"
yvar = "Body Mass (g)"

flipper_length = adelie[xvar]
body_mass = adelie[yvar]
```

Here's a scatter plot showing the relationship between these quantities:

```
plt.scatter(flipper_length, body_mass, marker=".", alpha=0.5)
decorate(xlabel=xvar, ylabel=yvar)
```



It looks like they are related—we can quantify the strength of the relationship by computing the coefficient of correlation:

```
np.corrcoef(flipper_length, body_mass)[0, 1]
```

```
0.4682016942179394
```

The correlation is about 0.47, so penguins with longer flippers tend to be heavier. That's useful because it means we can guess a penguin's weight more accurately if we know its flipper length—but correlation alone doesn't tell us how to make those guesses. For that, we need to choose a **line of best fit**.

There are many ways to define the "best" line, but for data like this a common choice is a **linear least squares fit**, which is the straight line that minimizes the mean squared error (MSE).

SciPy provides a function called `linregress` that computes a least squares fit. The name is short for **linear regression**, which is another term for a model like this. The arguments of `linregress` are the `x` values and the `y` values, in that order:

```
from scipy.stats import linregress

result = linregress(flipper_length, body_mass)
result
```

```
LinregressResult(slope=32.83168975115009, intercept=-2535.8368022002514,
rvalue=0.46820169421793933, pvalue=1.3432645947790051e-09,
stderr=5.076138407990821, intercept_stderr=964.7984274994059)
```

The result is a `LinregressResult` object that contains the slope and intercept of the fitted line, along with other information we'll unpack soon. The slope is about 32.8, which means that each additional millimeter of flipper length is associated with an additional 32.8 grams of body weight.

The intercept is –2535 grams, which might seem nonsensical, since a measured weight can't be negative. It might make more sense if we use the slope and intercept to evaluate the fitted line at the average flipper length:

```
x = flipper_length.mean()
y = result.intercept + result.slope * x
x, y
```

```
(189.95364238410596, 3700.662251655629)
```

For a penguin with the average flipper length, about 190 mm, the expected body weight is about 3,700 grams.

The following function takes the result from `linregress` and a sequence of `xs` and finds the point on the fitted line for each value of `x`:

```
def predict(result, xs):
    ys = result.intercept + result.slope * xs
    return ys
```

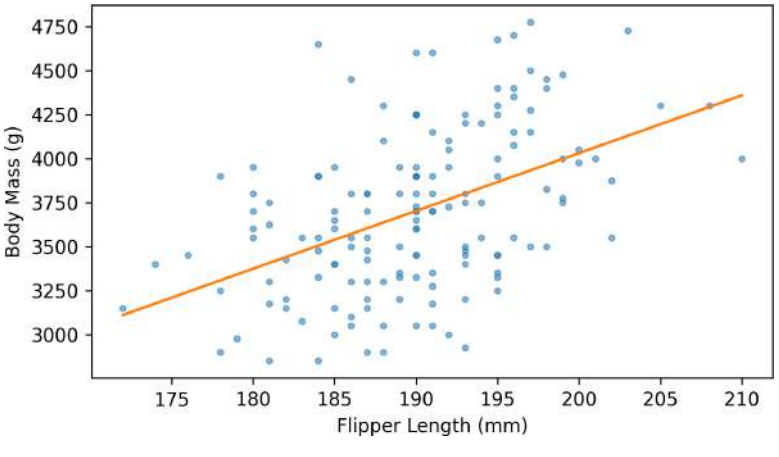
The name `predict` might seem odd here—in natural language, a **prediction** usually pertains to something happening in the future, but in the context of regression, the points on the fitted line are also called predictions.

We can use `predict` to compute the points on the line for a range of flipper sizes:

```
fit_xs = np.linspace(np.min(flipper_length), np.max(flipper_length))
fit_ys = predict(result, fit_xs)
```

Here's the fitted line along with the scatter plot of the data:

```
plt.scatter(flipper_length, body_mass, marker=".", alpha=0.5)
plt.plot(fit_xs, fit_ys, color="C1")
decorate(xlabel=xvar, ylabel=yvar)
```



As expected, the fitted line goes through the center of the data and follows the trend. And some of the predictions are accurate—but many of the data points are far from the line. To get a sense of how good (or bad) the predictions are, we can compute the prediction error, which is the vertical distance of each point from the line. The following function computes these errors, which are also called **residuals**:

```
def compute_residuals(result, xs, ys):
    fit_ys = predict(result, xs)
    return ys - fit_ys
```

Here are the residuals for body mass as a function of flipper length:

```
residuals = compute_residuals(result, flipper_length, body_mass)
```

As an example, we can look at the results for the first penguin in the dataset:

```
x = flipper_length[0]
y = predict(result, x)
x, y
```

```
(181.0, 3406.699042757914)
```

The flipper length of the selected penguin is 181 mm and the predicted body mass is 3,407 grams. Now let's see what the actual mass is:

```
body_mass[0], residuals[0]
```

```
(3750.0, 343.30095724208604)
```

The actual mass of this penguin is 3,750 grams and the residual—after subtracting away the prediction—is 343 grams.

The average of the squared residuals is the mean squared error (MSE) of the predictions:

```
mse = np.mean(residuals**2)
mse
```

```
163098.85902884745
```

By itself, this number doesn't mean very much. We can make more sense of it by computing the coefficient of determination.

Coefficient of Determination

Suppose you want to guess the weight of a penguin. If you know its flipper length, you can use the least squares fit to inform your guess, and the MSE quantifies the accuracy of your guesses, on average.

But what if you don't know the flipper length—what would you guess? It turns out that guessing the mean is the best strategy in the sense that it minimizes the MSE. If we always guess the mean, the prediction errors are the deviations from the mean:

```
deviations = body_mass - np.mean(body_mass)
```

And the MSE is the mean squared deviation:

```
np.mean(deviations**2)
```

```
208890.28989956583
```

You might remember that the mean squared deviation is the variance:

```
np.var(body_mass)
```

```
208890.28989956583
```

So we can think of the variance of the masses as the MSE if we always guess the mean, and the variance of the residuals as the MSE if we use the regression line. If we compute the ratio of these variances and subtract it from 1, the result indicates how much the MSE is reduced if we use flipper lengths to inform our guesses.

The following function computes this value, which is technically called the **coefficient of determination**, but because it is denoted R^2, most people call it "R squared."

```
def coefficient_of_determination(ys, residuals):
    return 1 - np.var(residuals) / np.var(ys)
```

In the example, R^2 is about 0.22, which means that the fitted line reduces MSE by 22%:

```
R2 = coefficient_of_determination(body_mass, residuals)
R2
```

```
0.21921282646854912
```

It turns out that there's a relationship between the coefficient of determination, R^2, and the coefficient of correlation, r. As you might guess based on the notation, $r^2 = R^2$. We can show that's true by computing the square root of R^2:

```
r = np.sqrt(R2)
r
```

```
0.4682016942179397
```

And comparing it to the correlation we computed earlier:

```
corr = np.corrcoef(flipper_length, body_mass)[0, 1]
corr
```

```
0.4682016942179394
```

They are the same except for a small difference due to floating-point approximation.

The `linregress` function also computes this value and returns it as an attribute in the `RegressionResult` object:

```
result.rvalue
```

```
0.46820169421793933
```

The coefficients of determination and correlation convey mostly the same information, but they are interpreted differently:

- Correlation quantifies the strength of the relationship on a scale from –1 to 1.
- R^2 quantifies the ability of the fitted line to reduce MSE.

Also, R^2 is always positive, so it doesn't indicate whether the correlation is positive or negative.

Minimizing MSE

Earlier I said that the least squares fit is the straight line that minimizes the mean squared error (MSE). We won't prove that, but we can test it by adding small random values to the intercept and slope, and checking whether the MSE gets worse:

```
intercept = result.intercept + np.random.normal(0, 1)
slope = result.slope + np.random.normal(0, 1)
```

To run the test, we need to make an object with `intercept` and `slope` attributes—we'll use the `SimpleNamespace` object provided by the `types` module:

```
from types import SimpleNamespace

fake_result = SimpleNamespace(intercept=intercept, slope=slope)
fake_result
```

```
namespace(intercept=-2535.738911382989, slope=34.24509022936497)
```

We can pass this object to `compute_residuals` and use the residuals to compute the MSE:

```
fake_residuals = compute_residuals(fake_result, flipper_length, body_mass)
fake_mse = np.mean(fake_residuals**2)
```

If we compare the result to the MSE of the least squares line, it is always worse:

```
mse, fake_mse, fake_mse > mse
```

```
(163098.85902884745, 235318.11301937344, True)
```

Minimizing MSE is nice, but it's not the only definition of "best." One alternative is to minimize the absolute values of the errors. Another is to minimize the shortest distance from each point to the fitted line, which is called the "total error." In some contexts, guessing too high might be better (or worse) than guessing too low. In that case you might want to compute a cost function for each residual, and minimize total cost.

But the least squares fit is much more widely used than these alternatives, primarily because it is efficient to compute. The following function shows how:

```
def least_squares(xs, ys):
    xbar = np.mean(xs)
    ybar = np.mean(ys)

    xdev = xs - xbar
    ydev = ys - ybar

    slope = np.sum(xdev * ydev) / np.sum(xdev**2)
    intercept = ybar - slope * xbar

    return intercept, slope
```

To test this function, we'll use flipper length and body mass again:

```
intercept, slope = least_squares(flipper_length, body_mass)
intercept, slope
```

```
(-2535.8368022002524, 32.831689751150094)
```

And we can confirm that we get the same results we got from `linregress`:

```
np.allclose([intercept, slope], [result.intercept, result.slope])
```

```
True
```

Minimizing MSE made sense when computational efficiency was more important than choosing the method most appropriate to the problem at hand. But that's no longer the case, so it is worth considering whether squared residuals are the right thing to minimize.

Estimation

The parameters `slope` and `intercept` are estimates based on a sample. Like other estimates, they are vulnerable to nonrepresentative sampling, measurement error, and variability due to random sampling. As usual, it's hard to quantify the effect of nonrepresentative sampling and measurement error. It's easier to quantify the effect of random sampling.

One way to do that is a kind of resampling called **bootstrapping**: we'll treat the sample as if it were the whole population and draw new samples, with replacement, from the observed data. The following function takes a `DataFrame` and uses the `sample` method to resample the rows and return a new `DataFrame`:

```
def resample(df):
    n = len(df)
    return df.sample(n, replace=True)
```

And the following function takes a `DataFrame`, finds the least squares fit, and returns the slope of the fitted line:

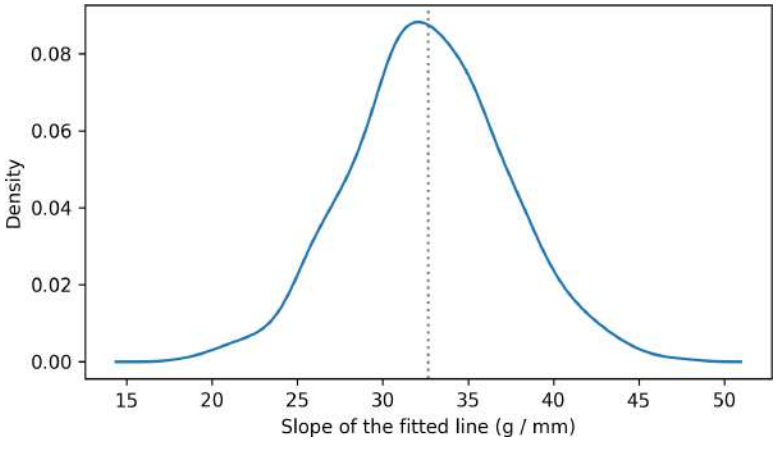
```
def estimate_slope(df):
    xs, ys = df["Flipper Length (mm)"], df["Body Mass (g)"]
    result = linregress(xs, ys)
    return result.slope
```

We can use these functions to generate many simulated datasets and compute the slope for each one:

```
resampled_slopes = [estimate_slope(resample(adelie)) for i in range(1001)]
```

The result is a sample from the sampling distribution of the slope. Here's what it looks like:

```
from thinkstats import plot_kde

plot_kde(resampled_slopes)
decorate(xlabel="Slope of the fitted line (g / mm)", ylabel="Density")
```



We can use `percentile` to compute a 90% confidence interval:

```
ci90 = np.percentile(resampled_slopes, [5, 95])
print(result.slope, ci90)
```

```
32.83168975115009 [25.39604591 40.21054526]
```

So we could report that the estimated slope is 33 grams/mm with a 90% CI [25, 40] grams/mm.

The standard error of the estimate is the standard deviation of the sampling distribution:

```
stderr = np.std(resampled_slopes)
stderr
```

```
4.570238986584832
```

The `RegressionResult` object we got from `linregress` provides an approximation of the standard error, based on some assumptions about the shape of the distribution:

```
result.stderr
```

```
5.076138407990821
```

The standard error we computed by resampling is a little smaller, but the difference probably doesn't matter in practice.

Visualizing Uncertainty

Each time we resample the dataset, we get a different fitted line. To see how much variation there is in the lines, one option is to loop through them and plot them all. The following function takes a resampled `DataFrame`, computes a least squares fit, and generates predicted values for a sequence of `xs`:

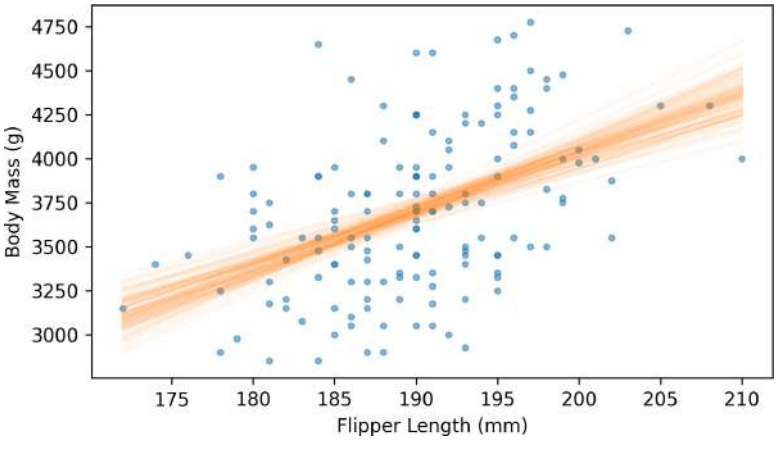
```
def fit_line(df, fit_xs):
    xs, ys = df["Flipper Length (mm)"], df["Body Mass (g)"]
    result = linregress(xs, ys)
    fit_ys = predict(result, fit_xs)
    return fit_ys
```

Here's the sequence of `xs` we'll use:

```
xs = adelie["Flipper Length (mm)"]
fit_xs = np.linspace(np.min(xs), np.max(xs))
```

And here's what the fitted lines look like, along with a scatter plot of the data:

```
plt.scatter(flipper_length, body_mass, marker=".", alpha=0.5)

for i in range(101):
    fit_ys = fit_line(resample(adelie), fit_xs)
    plt.plot(fit_xs, fit_ys, color="C1", alpha=0.05)

decorate(xlabel=xvar, ylabel=yvar)
```



Near the middle, the fitted lines are close together—at the extremes, they are farther apart.

Another way to represent the variability of the fitted lines is to plot a 90% confidence interval for each predicted value. We can do that by collecting the fitted lines as a list of arrays:

```
fitted_ys = [fit_line(resample(adelie), fit_xs) for i in range(1001)]
```

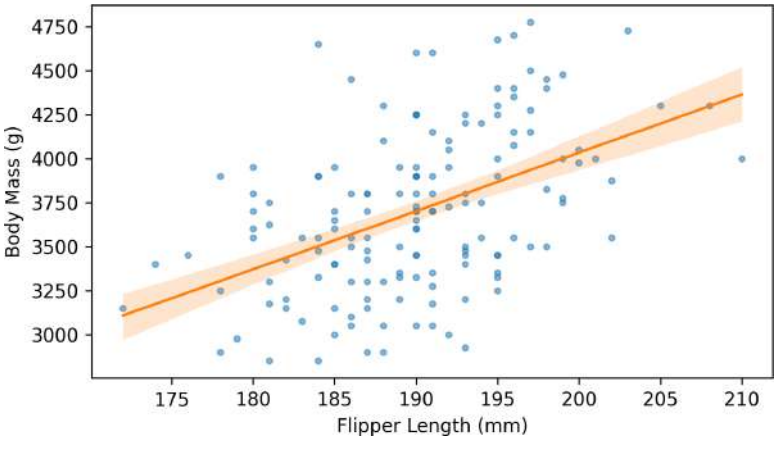
We can think of this list of arrays as a two-dimensional array with one row for each fitted line and one column corresponding to each of the values in `xs`.

We can use `percentile` with the `axis=0` argument to find the 5th, 50th, and 95th percentiles of the `ys` corresponding to each of the values in `xs`:

```
low, median, high = np.percentile(fitted_ys, [5, 50, 95], axis=0)
```

Now we'll use `fill_between` to plot a region between the 5th and 95 percentiles, which represents the 90% CI, along with the median value in each column and a scatter plot of the data:

```
plt.scatter(flipper_length, body_mass, marker=".", alpha=0.5)

plt.fill_between(fit_xs, low, high, color="C1", lw=0, alpha=0.2)
plt.plot(fit_xs, median, color="C1")

decorate(xlabel=xvar, ylabel=yvar)
```



This is my favorite way to represent the variability of a fitted line due to random sampling.

Transformation

Before fitting a line to data, it is sometimes useful to transform one or both variables, for example by computing the squares of the values, their square roots, or their logarithms. To demonstrate, we'll use heights and weights from the Behavioral Risk Factor Surveillance System (BRFSS), described in "The Lognormal Distribution" on page 79.

We can load the BRFSS data like this:

```
from thinkstats import read_brfss

brfss = read_brfss()
```

Next we'll find the rows with valid data and select the columns containing heights and weights:

```
valid = brfss.dropna(subset=["htm3", "wtkg2"])
heights, weights = valid["htm3"], valid["wtkg2"]
```

We can use `linregress` to compute the slope and intercept of the least squares fit:

```
result_brfss = linregress(heights, weights)
result_brfss.intercept, result_brfss.slope
```

```
(-82.65926054409877, 0.957074585033226)
```

The slope is about 0.96, which means that an increase of 1 centimeter corresponds to an increase of almost 1 kilogram, on average. We can use `predict` again to generate predicted values for a range of `xs`:

```
fit_xs = np.linspace(heights.min(), heights.max())
fit_ys = predict(result_brfss, fit_xs)
```

Before we make a scatter plot of the data, it's useful to jitter the heights and weights:

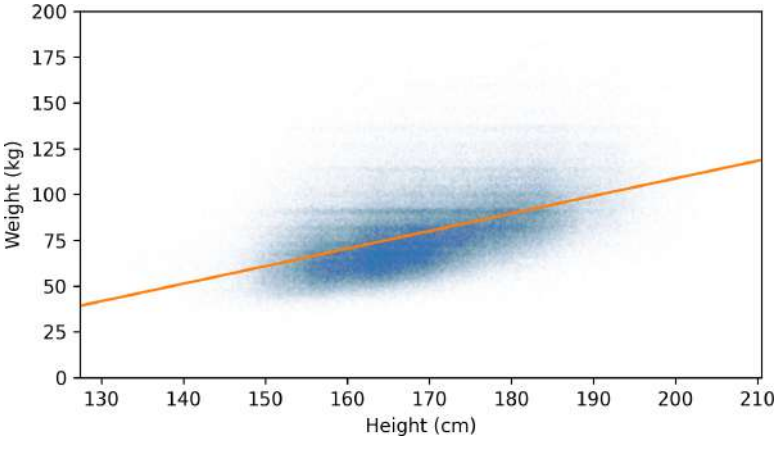
```
from thinkstats import jitter

jittered_heights = jitter(heights, 2)
jittered_weights = jitter(weights, 1.5)
```

And we'll use the mean and standard deviation of the heights to choose the limits of the x-axis:

```
m, s = heights.mean(), heights.std()
xlim = m - 4 * s, m + 4 * s
ylim = 0, 200
```

Here's a scatter plot of the jittered data along with the fitted line:

```
plt.scatter(jittered_heights, jittered_weights, alpha=0.01, s=0.1)
plt.plot(fit_xs, fit_ys, color="C1")
decorate(xlabel="Height (cm)", ylabel="Weight (kg)", xlim=xlim, ylim=ylim)
```

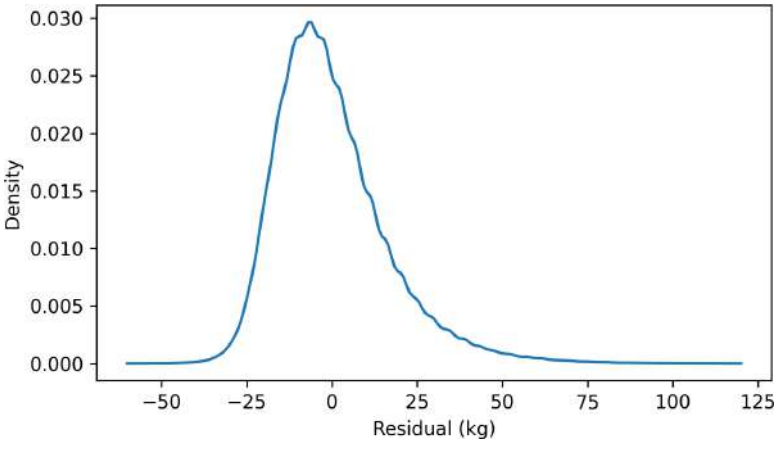


The fitted line doesn't pass through the densest part of the scatter plot. That's because the weights don't follow a normal distribution. As we saw in "The Lognormal Distribution" on page 79, adult weights tend to follow a lognormal distribution, which is skewed toward larger values—and those values pull the fitted line up.

Another cause for concern is the distribution of the residuals, which looks like this:

```
residuals = compute_residuals(result_brfss, heights, weights)
```

```
from thinkstats import make_pmf

pmf_kde = make_pmf(residuals, -60, 120)
pmf_kde.plot()

decorate(xlabel="Residual (kg)", ylabel="Density")
```



The distribution of the residuals is skewed to the right. By itself, that's not necessarily a problem, but it suggests that the least squares fit has not characterized the relationship between these variables properly.

If the weights follow a lognormal distribution, their logarithms follow a normal distribution. So let's see what happens if we fit a line to the logarithms of weight as a function of height:

```
log_weights = np.log10(weights)
result_brfss2 = linregress(heights, log_weights)
result_brfss2.intercept, result_brfss2.slope
```

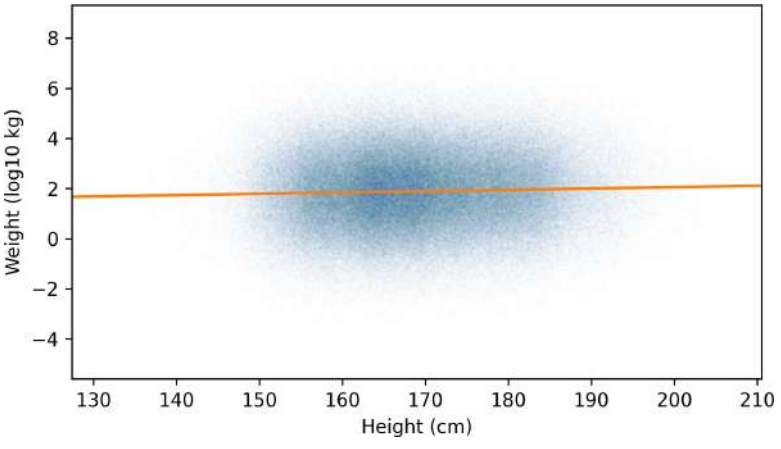
```
(0.9930804163932876, 0.005281454169417777)
```

Because we transformed one of the variables, the slope and intercept are harder to interpret. But we can use `predict` to compute the fitted line:

```
fit_xs = np.linspace(heights.min(), heights.max())
fit_ys = predict(result_brfss2, fit_xs)
```

And then plot it along with a scatter plot of the transformed data:

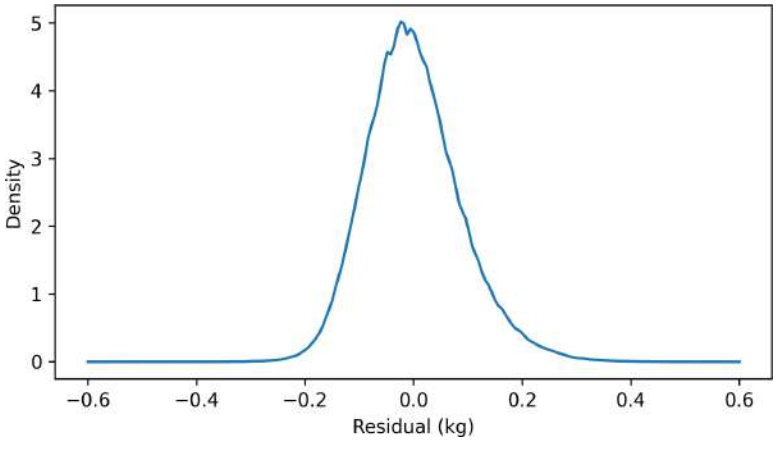
```
jittered_log_weights = jitter(log_weights, 1.5)
plt.scatter(jittered_heights, jittered_log_weights, alpha=0.01, s=0.1)
plt.plot(fit_xs, fit_ys, color="C1")
decorate(xlabel="Height (cm)", ylabel="Weight (log10 kg)", xlim=xlim)
```



The fitted line passes through the densest part of the plot, and the actual values extend about the same distance above and below the line—so the distribution of the residuals is roughly symmetric:

```
residuals = compute_residuals(result_brfss2, heights, log_weights)
```

```
pmf_kde = make_pmf(residuals, -0.6, 0.6)
pmf_kde.plot()

decorate(xlabel="Residual (kg)", ylabel="Density")
```



The appearance of the scatter plot and the distribution of the residuals suggest that the relationship of height and log-transformed weight is well described by the fitted

line. If we compare the r values of the two regressions, we see that the correlation of height with log-transformed weights is slightly higher:

```
result_brfss.rvalue, result_brfss2.rvalue
```

```
(0.5087364789734582, 0.5317282605983435)
```

Which means that the R^2 value is slightly higher, too:

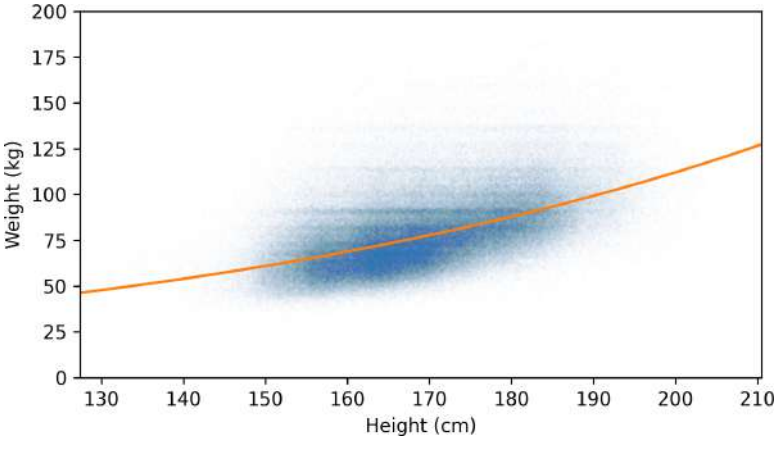
```
result_brfss.rvalue**2, result_brfss2.rvalue**2
```

```
(0.2588128050383119, 0.28273494311893993)
```

If we use heights to guess weights, the guesses are a little better if we work with the log-transformed weights.

However, transforming the data makes the parameters of the model harder to interpret—it can help to invert the transformation before presenting the results. For example, the inverse of a logarithm in base 10 is exponentiation with base 10. Here's what the fitted line looks like after the inverse transformation, along with the untransformed data:

```
plt.scatter(jittered_heights, jittered_weights, alpha=0.01, s=0.1)
plt.plot(fit_xs, 10**fit_ys, color="C1")
decorate(xlabel="Height (cm)", ylabel="Weight (kg)", xlim=xlim, ylim=ylim)
```



A fitted line that's straight with respect to the transformed data is curved with respect to the original data.

Glossary

model
: In the context of regression, a model is a mathematical description of the relationship between variables—such as a straight line—along with a description of random variation—such as a normal distribution.

imputation
: A process for estimating and filling in missing values in a dataset.

line of best fit
: A line (or curve) that best describes a relationship between variables, by some definition of "best."

linear regression
: A method for finding a line of best fit.

prediction
: A point on a line of best fit—in the context of regression, it is not necessarily a claim about the future.

residual
: The difference between an observed value and a value predicted by a line of best fit.

linear least squares fit
: A line that minimizes the sum of squared residuals.

coefficient of determination
: A statistic, denoted R^2 and often pronounced "R squared," that quantifies how well a model fits the data.

bootstrap resampling
: A way of resampling by treating the sample as a population and drawing new samples with the same size as the original, with replacement.

Exercises

Exercise 10.1

In this chapter we computed a least squares fit for penguin weights as a function of flipper length. There are two other measurements in the dataset we can also consider: culmen length and culmen depth (the culmen is the top ridge of the bill).

Compute the least squares fit for weight as a function of culmen length. Make a scatter plot of these variables and plot the fitted line.

Based on the `rvalue` attribute of the `RegressionResult` object, what is the correlation of these variables? What is the coefficient of determination? Which is a better predictor of weight, culmen length or flipper length?

Exercise 10.2

In this chapter we used resampling to approximate the sampling distribution for the slope of a fitted line. We can approximate the sampling distribution of the intercept the same way.

1. Write a function called `estimate_intercept` that takes a resampled `DataFrame` as an argument, computes the least squares fit of penguin weight as a function of flipper length, and returns the intercept.
2. Call the function with many resampled versions of `adelie` and collect the intercepts.
3. Use `plot_kde` to plot the sampling distribution of the intercept.
4. Compute the standard error and a 90% confidence interval.
5. Check that the standard error you get from resampling is consistent with the `intercept_stderr` attribute in the `RegressionResult` object—it might be a little smaller.

Exercise 10.3

A person's Body Mass Index (BMI) is their weight in kilograms divided by their height in meters raised to the second power. In the BRFSS dataset, we can compute BMI like this, after converting heights from centimeters to meters:

```
heights_m = heights / 100
bmis = weights / heights_m**2
```

In this definition, heights are squared, rather than raised to some other exponent, because of the observation—early in the history of statistics—that average weight increases roughly in proportion to height squared.

To see whether that's true, we can use data from the BRFSS, a least squares fit, and a little bit of math. Suppose weight is proportional to height raised to an unknown exponent, a. In that case, we can write:

$$w = bh^a$$

where w is weight, h is height, and b is an unknown constant of proportionality.

Taking logarithms of both sides:

$$\log w = \log b + a \log h$$

So, if we compute a least squares fit for log-transformed weights as a function of log-transformed heights, the slope of the fitted line estimates the unknown exponent *a*.

Compute the logarithms of height and weight. You can use any base for the logarithms, as long as it's the same for both transformations. Compute a least squares fit. Is the slope close to 2?

CHAPTER 11

Multiple Regression

The linear least squares fit in the previous chapter is an example of **regression**, which is the more general problem of modeling the relationship between one set of variables, called **response variables** or dependent variables, and another set of variables, called **explanatory variables** or independent variables.

In the examples in the previous chapter, there is only one response variable and one explanatory variable, which is called **simple regression**. In this chapter, we move on to **multiple regression**, with more than one explanatory variable, but still only one response variable. If there is more than one response variable, that's multivariate regression, which we won't cover in this book.

StatsModels

In the previous chapter we used the SciPy function `linregress` to compute least squares fit. This function performs simple regression, but not multiple regression. For that, we'll use `StatsModels`, a package that provides several forms of regression and other analyses.

As a first example, we'll continue exploring the penguin data. When we load the data, we'll use the following dictionary to give the columns names that don't contain spaces, which will make them easier to use with StatsModels:

```
columns = {
    "Body Mass (g)": "mass",
    "Flipper Length (mm)": "flipper_length",
    "Culmen Length (mm)": "culmen_length",
    "Culmen Depth (mm)": "culmen_depth",
}
```

Now we can load the data, drop the rows with missing body mass, and rename the columns:

```
penguins = (
    pd.read_csv("penguins_raw.csv")
    .dropna(subset=["Body Mass (g)"])
    .rename(columns=columns)
)
penguins.shape
```

```
(342, 17)
```

The dataset contains three species of penguins. We'll work with just the Adélie penguins:

```
adelie = penguins.query('Species.str.startswith("Adelie")').copy()
len(adelie)
```

```
151
```

In the previous chapter, we computed a least squares fit between the penguins' flipper lengths and weights:

```
flipper_length = adelie["flipper_length"]
body_mass = adelie["mass"]
```

As a reminder, here's how we did that with `linregress`:

```
from scipy.stats import linregress

result_linregress = linregress(flipper_length, body_mass)
result_linregress.intercept, result_linregress.slope
```

```
(-2535.8368022002514, 32.83168975115009)
```

`StatsModels` provides two interfaces (APIs)—we'll use the "formula" API, which uses the Patsy formula language to specify the response and explanatory variables. The following formula string specifies that the response variable, `mass`, is a linear function of one explanatory variable, `flipper_length`:

```
formula = "mass ~ flipper_length"
```

We can pass this formula to the `StatsModels` function `ols`, along with the data:

```
import statsmodels.formula.api as smf

model = smf.ols(formula, data=adelie)
type(model)
```

```
statsmodels.regression.linear_model.OLS
```

The name `ols` stands for "ordinary least squares," which indicates that this function computes a least squares fit under the most common, or "ordinary," set of assumptions.

The result is an `OLS` object that represents the model. In general, a model is a simplified description of the relationship between variables. In this example, it's a linear model, which means that it assumes that the response variable is a linear function of the explanatory variables.

The `fit` method fits the model to the data and returns a `RegressionResults` object that contains the result:

```
result_ols = model.fit()
```

The `RegressionResults` object contains a lot of information, so `thinkstats` provides a function that just displays the information we need for now:

```
from thinkstats import display_summary

display_summary(result_ols)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2535.8368	964.798	-2.628	0.009	-4442.291	-629.382
flipper_length	32.8317	5.076	6.468	0.000	22.801	42.862

R-squared: 0.2192

The first column contains the intercept and slope, which are the **coefficients** of the model. We can confirm that they are the same as the coefficients we got from `linregress`:

```
result_linregress.intercept, result_linregress.slope
```

```
(-2535.8368022002514, 32.83168975115009)
```

The second column contains the standard errors of the coefficients—again, they are the same as the values we got from `linregress`:

```
result_linregress.intercept_stderr, result_linregress.stderr,
```

```
(964.7984274994059, 5.076138407990821)
```

The next column reports *t* statistics, which are used to compute p-values—we can ignore them, because the p-values are in the next column, labeled `P>|t|`. The p-value for `flipper_length` is rounded down to 0, but we can display it like this:

```
result_ols.pvalues["flipper_length"]
```

```
1.3432645947789321e-09
```

And confirm that `linregress` computed the same result:

```
result_linregress.pvalue
```

```
1.3432645947790051e-09
```

The p-value is very small, which means that if there were actually no relationship between weight and flipper length, it is very unlikely we would see a slope as big as the estimated value by chance.

The last two columns, labeled `[0.025` and `0.975]`, report 95% confidence intervals for the intercept and slope. So the 95% CI for the slope is [22.8, 42.9].

The last line reports the R^2 value of the model, which is about 0.22—that means we can reduce MSE by about 22% if we use flipper length to predict weight, compared to using only the average weight.

The R^2 value we get from simple correlation is the square of the correlation coefficient, r. So we can compare `rsquared` computed by `ols` with the square of the `rvalue` computed by `linregress`:

```
result_ols.rsquared, result_linregress.rvalue**2
```

```
(0.21921282646854878, 0.21921282646854875)
```

They are the same except for a small difference due to floating-point approximation.

Before we go on to multiple regression, let's compute one more simple regression, with `culmen_length` as the explanatory variable (the culmen is the top ridge of the bill):

```
formula = "mass ~ culmen_length"
result = smf.ols(formula, data=adelie).fit()
display_summary(result)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	34.8830	458.439	0.076	0.939	-870.998	940.764
culmen_length	94.4998	11.790	8.015	0.000	71.202	117.798

R-squared: 0.3013

Again, the p-value of the slope is very small, which means that if there were actually no relationship between mass and culmen length, it is unlikely we would see a slope this big by chance. You might notice that the p-value associated with the intercept is large, but that's not a problem because we are not concerned about whether the intercept might be zero. In this model, the intercept is close to zero, but that's just a coincidence—it doesn't indicate a problem with the model.

The R^2 value for this model is about 0.30, so the reduction in MSE is a little higher if we use culmen length rather than flipper length as an explanatory variable (the R^2 value with flipper length is 0.22). Now, let's see what happens if we combine them.

On to Multiple Regression

Here's the Patsy formula for a multiple regression model where mass is a linear function of both flipper length and culmen length:

```
formula = "mass ~ flipper_length + culmen_length"
```

And here is the result of fitting this model to the data:

```
result = smf.ols(formula, data=adelie).fit()
display_summary(result)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-3573.0817	866.739	-4.122	0.000	-5285.864	-1860.299
flipper_length	22.7024	4.742	4.787	0.000	13.331	32.074
culmen_length	76.3402	11.644	6.556	0.000	53.331	99.350

R-squared: 0.3949

This model has three coefficients: the intercept and two slopes. The slope associated with flipper length is 22.7, which means we expect a penguin with a longer flipper, by one millimeter, to weight more, by 22.7 grams—assuming that culmen length is the same. Similarly, we expect a penguin with a longer culmen, by one millimeter, to weight more, by 76.3 grams—assuming that flipper length is the same.

The p-values associated with both slopes are small, which means that the contribution of both explanatory variables would be unlikely to happen by chance.

And the R^2 value is 0.39, higher than the model with only culmen length (0.30) and the model with only flipper length (0.22). So predictions based on both explanatory variables are better than predictions based on either one alone.

But they are not as much better as we might have hoped. If flipper length reduces MSE by 22% and culmen length reduces it by 30%, why don't the two of them together reduce it by a total of 52%? The reason is that the explanatory variables are correlated with each other:

```
from thinkstats import corrcoef

corrcoef(adelie, "flipper_length", "culmen_length")
```

```
0.32578471516515944
```

A penguin with a longer flipper also has a longer culmen, on average. The explanatory variables contain some information about each other, which means that they contain some of the same information about the response variable. When we add an explanatory variable to the model, the improvement in R^2 reflects only the new information provided by the new variable.

We see the same pattern if we add culmen depth as a third explanatory variable:

```
formula = "mass ~ flipper_length + culmen_length + culmen_depth"
result = smf.ols(formula, data=adelie).fit()
display_summary(result)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-4341.3019	795.117	-5.460	0.000	-5912.639	-2769.964
flipper_length	17.4215	4.385	3.973	0.000	8.756	26.087
culmen_length	55.3676	11.133	4.973	0.000	33.366	77.369
culmen_depth	140.8946	24.216	5.818	0.000	93.037	188.752

R-squared: 0.5082

This model has four coefficients. All of the p-values are small, which means that the contribution of each explanatory variable would be unlikely to happen by chance. And the R^2 value is about 0.51, somewhat better than the previous model with two explanatory variables (0.39), and better than either model with a single variable (0.22 and 0.30).

But again, the incremental improvement is smaller than we might have hoped, because culmen depth is correlated with the other two measurements:

```
[
    corrcoef(adelie, "culmen_depth", "flipper_length"),
    corrcoef(adelie, "culmen_depth", "culmen_length"),
]
```

```
[0.30762017939668534, 0.39149169183587634]
```

This example demonstrates a common use of multiple regression, combining multiple explanatory variables to make better predictions. Another common use is to quantify the contribution of one set of variables while controlling for the contribution of another set.

Control Variables

In "Comparing CDFs" on page 52 we saw that first babies are lighter than other babies, on average. And in "Testing a Correlation" on page 156 we saw that birth weight is correlated with the mother's age—older mothers have heavier babies on average.

These results might be related. If mothers of first babies are younger than mothers of other babies—which seems likely—that might explain why their babies are lighter. We can use multiple regression to test this conjecture, by estimating the difference in birth weight between first babies and others while controlling for the mothers' ages.

Instructions for downloading the NSFG data are in the notebook for this chapter.

We can use `get_nsfg_groups` to read the data, select live births, and group live births into first babies and others:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

We'll use `dropna` to select the rows with valid birth weights, birth order, and mother's ages:

```
valid = live.dropna(subset=["agepreg", "birthord", "totalwgt_lb"]).copy()
```

Now we can use `StatsModels` to confirm that birth weight is correlated with age, and to estimate the slope—assuming that it is a linear relationship:

```
formula = "totalwgt_lb ~ agepreg"
result_age = smf.ols(formula, data=valid).fit()
display_summary(result_age)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	6.8304	0.068	100.470	0.000	6.697	6.964
agepreg	0.0175	0.003	6.559	0.000	0.012	0.023

R-squared: 0.004738

The slope is small, only 0.0175 pounds per year. So if two mothers differ in age by a decade, we expect their babies to differ in weight by 0.175 pounds. But the p-value is small, so this slope—small as it is—would be unlikely if there were actually no relationship.

The R^2 value is also small, which means that the mother's age is not very useful as a predictive variable. If we know the mother's age, our ability to predict the baby's weight is hardly improved at all.

This combination of a small p-value and a small R^2 value is a common source of confusion, because it seems contradictory—if the relationship is statistically significant, it seems like it should be predictive. But this example shows that there is no contradiction—a relationship can be statistically significant but not very useful for prediction. If we visualize the results, we'll see why. First let's select the relevant columns:

```
totalwgt = valid["totalwgt_lb"]
agepreg = valid["agepreg"]
```

To compute the fitted line, we could extract the intercept and slope from `result_age`, but we don't have to. The `RegressionResults` object provides a `predict` method we can use instead. First we'll compute a range of values for `agepreg`:

```
agepreg_range = np.linspace(agepreg.min(), agepreg.max())
```

To use `predict`, we have to put values for the explanatory variables in a `DataFrame`:

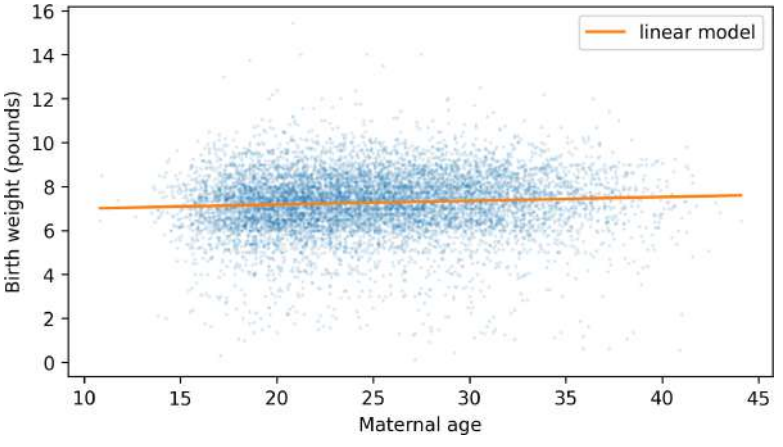
```
df = pd.DataFrame({"agepreg": agepreg_range})
```

The columns in the `DataFrame` have to have the same names as the explanatory variables. Then we can pass it to `predict`:

```
fit_ys = result_age.predict(df)
```

The result is a `Series` containing the predicted values. Here's what they look like, along with a scatter plot of the data:

```
plt.scatter(agepreg, totalwgt, marker=".", alpha=0.1, s=5)
plt.plot(agepreg_range, fit_ys, color="C1", label="linear model")

decorate(xlabel="Maternal age", ylabel="Birth weight (pounds)")
```



Because the slope of the fitted line is small, we can barely see the difference in the expected birth weight between the youngest and oldest mothers. The variation in birth weight, at every maternal age, is much larger.

Next we'll use `StatsModels` to confirm that first babies are lighter than others. To make that work, we'll create a Boolean `Series` that is `True` for first babies and `False` for others, and add it as a new column called `is_first`:

```
valid["is_first"] = valid["birthord"] == 1
```

Here's the formula for a model with birth weight as the response variable and `is_first` as the explanatory variable. In the Patsy formula language, `C` and the parentheses around the variable name indicate that it is **categorical**—that is, it represents categories like "first baby" rather than measurements like birth weight:

```
formula = "totalwgt_lb ~ C(is_first)"
```

Now we can fit the model and display the results, as usual:

```
result_first = smf.ols(formula, data=valid).fit()
display_summary(result_first)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	7.3259	0.021	356.007	0.000	7.286	7.366
C(is_first)[T.True]	-0.1248	0.030	-4.212	0.000	-0.183	-0.067

R-squared: 0.00196

In the results, the label `C(is_first)[T.True]` indicates that `is_first` is a categorical variable and the coefficient is associated with the value `True`. The `T` before `True` stands for "treatment"—in the language of a controlled experiment, first babies are considered the treatment group and other babies are considered the reference group. These designations are arbitrary—we could consider first babies to be the reference group and others to be the treatment group. But, to interpret the results, we need to know which is which.

The intercept is about 7.3, which means that the average weight of the reference group is 7.3 pounds. The coefficient of `is_first` is –0.12, which means that the average weight of the treatment group—first babies—is 0.12 pounds lighter. We can check both of these results by computing them directly:

```
others["totalwgt_lb"].mean()
```

```
7.325855614973262
```

```
diff_weight = firsts["totalwgt_lb"].mean() - others["totalwgt_lb"].mean()
diff_weight
```

```
-0.12476118453549034
```

In addition to these coefficients, `StatsModels` also computes p-values, confidence intervals, and R^2. The p-value associated with first babies is small, which means that the difference between the groups is statistically significant. And the R^2 value is small, which means that if we're trying to guess the weight of a baby, it doesn't help much to know whether it is a first baby.

Now let's see if it's plausible that the difference in birth weight is due to the difference in maternal age. On average, mothers of first babies are about 3.6 years younger than other mothers:

```
diff_age = firsts["agepreg"].mean() - others["agepreg"].mean()
diff_age
```

```
-3.5864347661500275
```

And the slope of birth weight as a function of age is 0.0175 pounds per year:

```
slope = result_age.params["agepreg"]
slope
```

```
0.017453851471802638
```

If we multiply the slope by the difference in ages, we get the expected difference in birth weight for first babies and others, due to the mother's age:

```
slope * diff_age
```

```
-0.0625970997216918
```

The result is 0.063 pounds, which is about half of the observed difference. So it seems like the observed difference in birth weight can be partly explained by the difference in mother's age.

Using multiple regression, we can estimate coefficients for maternal age and first babies at the same time:

```
formula = "totalwgt_lb ~ agepreg + C(is_first)"
result = smf.ols(formula, data=valid).fit()
display_summary(result)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	6.9142	0.078	89.073	0.000	6.762	7.066
C(is_first)[T.True]	-0.0698	0.031	-2.236	0.025	-0.131	-0.009
agepreg	0.0154	0.003	5.499	0.000	0.010	0.021

R-squared: 0.005289

The coefficient of `is_first` is –0.0698, which means that first babies are 0.0698 pounds lighter than others, on average, after accounting for the difference due to maternal age. That's about half of the difference we get without accounting for maternal age.

And the p-value is 0.025, which is still considered statistically significant but is in the borderline range where we can't exclude the possibility that a difference this size could happen by chance.

Because this model takes into account the weight difference due to maternal age, we can say that it **controls for** maternal age. But it assumes that the relationship between weight and maternal age is linear. So let's see if that's true.

Nonlinear Relationships

To check whether the contribution of `agepreg` might be nonlinear, we can add a new column to the dataset, which contains the values of `agepreg` squared:

```
valid["agepreg2"] = valid["agepreg"] ** 2
```

Now we can define a model that includes a linear relationship *and* a quadratic relationship:

```
formula = "totalwgt_lb ~ agepreg + agepreg2"
```

We can fit the model in the usual way:

```
result_age2 = smf.ols(formula, data=valid).fit()
display_summary(result_age2)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	5.5720	0.275	20.226	0.000	5.032	6.112
agepreg	0.1186	0.022	5.485	0.000	0.076	0.161
agepreg2	-0.0019	0.000	-4.714	0.000	-0.003	-0.001

R-squared: 0.00718

The p-value associated with the quadratic term, `agepreg2`, is very small, which suggests that it contributes more information about birth weight than we would expect by chance. And the R^2 value for this model is 0.0072, higher than for the linear model (0.0047).

By estimating coefficients for `agepreg` and `agepreg2`, we are effectively fitting a parabola to the data. To see that, we can use the `RegressionResults` object to generate predictions for a range of maternal ages.

First we'll create a temporary `DataFrame` that contains columns named `agepreg` and `agepreg2`, based on the range of ages in `agepreg_range`:

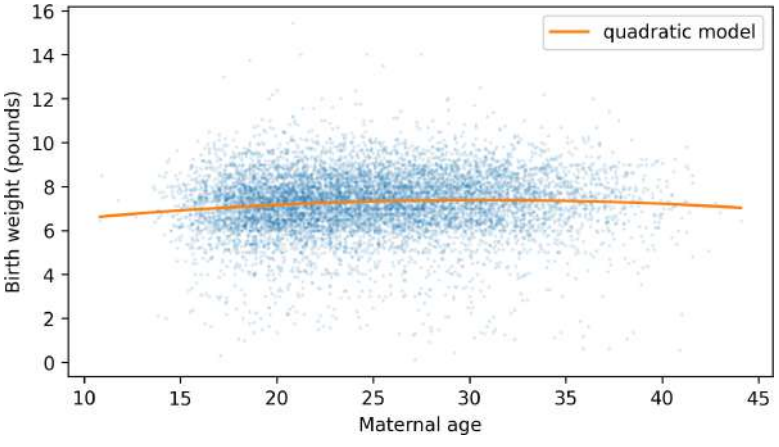
```
df = pd.DataFrame({"agepreg": agepreg_range})
df["agepreg2"] = df["agepreg"] ** 2
```

Now we can use the `predict` method, passing the `DataFrame` as an argument and getting back a `Series` of predictions:

```
fit_ys = result_age2.predict(df)
```

Here's what the fitted parabola looks like, along with a scatter plot of the data:

```
plt.scatter(agepreg, totalwgt, marker=".", alpha=0.1, s=5)
plt.plot(agepreg_range, fit_ys, color="C1", label="quadratic model")

decorate(xlabel="Maternal age", ylabel="Birth weight (pounds)")
```



The curvature is subtle, but it suggests that birth weights are lower for the youngest and oldest mothers, and higher in the middle.

The quadratic model captures the relationship between these variables better than the linear model, which means it can account more effectively for the difference in birth weight due to maternal age. So let's see what happens when we add `is_first` to the quadratic model:

```
formula = "totalwgt_lb ~ agepreg + agepreg2 + C(is_first)"
result = smf.ols(formula, data=valid).fit()
display_summary(result)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	5.6923	0.286	19.937	0.000	5.133	6.252
C(is_first)[T.True]	-0.0504	0.031	-1.602	0.109	-0.112	0.011
agepreg	0.1124	0.022	5.113	0.000	0.069	0.155
agepreg2	-0.0018	0.000	-4.447	0.000	-0.003	-0.001

R-squared: 0.007462

With a more effective control for maternal age, the estimated difference between first babies and others is 0.0504 pounds, smaller than the estimate with just the linear model (0.0698 pounds). And the p-value associated with `is_first` is 0.109, which mean it is plausible that the remaining difference between these groups is due to chance.

We can conclude that the difference in birth weight is explained—at least in part and possibly in full—by the difference in mother's age.

Logistic Regression

Linear regression is based on a model where the expected value of the response variable is the weighted sum of the explanatory variables and an intercept. This model is appropriate when the response variable is a continuous quantity like birth weight or penguin mass, but not when the response variable is a discrete quantity like a count or a category.

For these kinds of response variables, we can use **generalized linear models** or GLMs. For example:

- If the response variable is a count, we can use Poisson regression.
- If it's categorical with only two categories, we can use logistic regression.
- If it's categorical with more than two categories, we can use multinomial logistic regression.
- If it's categorical and the categories can be arranged in order, we can use ordered logistic regression.

We won't cover all of them in this book—just **logistic regression**, which is the most widely used. As an example, we'll use the penguin dataset again, and see if we can tell whether a penguin is male or female, based on its weight and other measurements.

`StatsModels` provides a function that does logistic regression—it's called `logit` because that's the name of a mathematical function that appears in the definition of logistic regression. Before we can use the `logit` function, we have to transform the response variable so the values are `0` and `1`:

```
adelie["y"] = (adelie["Sex"] == "MALE").astype(int)
```

We'll start with a simple model with `y` as the response variable and `mass` as the explanatory variable. Here's how we make and fit the model—the argument `disp=False` suppresses messages about the fitting process:

```
model = smf.logit("y ~ mass", data=adelie)
result = model.fit(disp=False)
```

And here are the results:

```
display_summary(result)
```

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-25.9871	4.221	-6.156	0.000	-34.261	-17.713
mass	0.0070	0.001	6.138	0.000	0.005	0.009

Pseudo R-squared: 0.5264

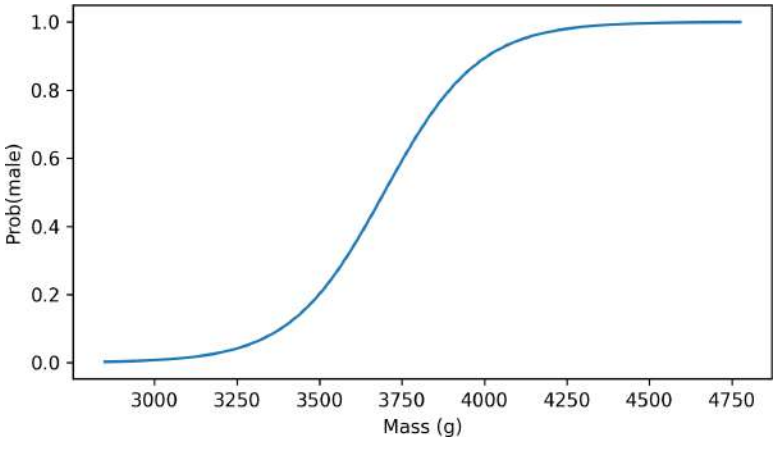
The coefficient of determination, R^2, does not apply to logistic regression, but there are several alternatives that are used as "pseudo R^2 values." The pseudo R^2 value for this model is about 0.526, which doesn't mean much by itself, but we will use it to compare models.

The coefficient of `mass` is positive, which means that heavier penguins are more likely to be male. Other than that, the coefficients are not easy to interpret—we can understand the model better by plotting the predictions. We'll make a `DataFrame` with a range of values for `mass`, and use `predict` to compute a `Series` of predictions:

```
mass = adelie["mass"]
mass_range = np.linspace(mass.min(), mass.max())
df = pd.DataFrame({"mass": mass_range})
fit_ys = result.predict(df)
```

Each predicted value is the probability a penguin is male as a function of its weight. Here's what the predicted values look like:

```
plt.plot(mass_range, fit_ys)

decorate(xlabel="Mass (g)", ylabel="Prob(male)")
```



The lightest penguins are almost certain to be female, and the heaviest are likely to be male—in the middle, a penguin weighing 3750 grams is about equally likely to be male or female.

Now let's see what happens if we add the other measurements as explanatory variables:

```
formula = "y ~ mass + flipper_length + culmen_length + culmen_depth"
model = smf.logit(formula, data=adelie)
result = model.fit(disp=False)
display_summary(result)
```

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-60.6075	13.793	-4.394	0.000	-87.642	-33.573
mass	0.0059	0.001	4.153	0.000	0.003	0.009
flipper_length	-0.0209	0.052	-0.403	0.687	-0.123	0.081
culmen_length	0.6208	0.176	3.536	0.000	0.277	0.965
culmen_depth	1.0111	0.349	2.896	0.004	0.327	1.695

Pseudo R-squared: 0.6622

The pseudo R^2 value of this model is 0.662, higher than the previous model (0.526)—so the additional measurements contain additional information that distinguishes male and female penguins.

The p-values for culmen length and depth are small, which indicates that they contribute more information than we expect by chance. The p-value for flipper length is large, which suggests that if you know a penguin's weight and bill dimensions, flipper length doesn't contribute additional information.

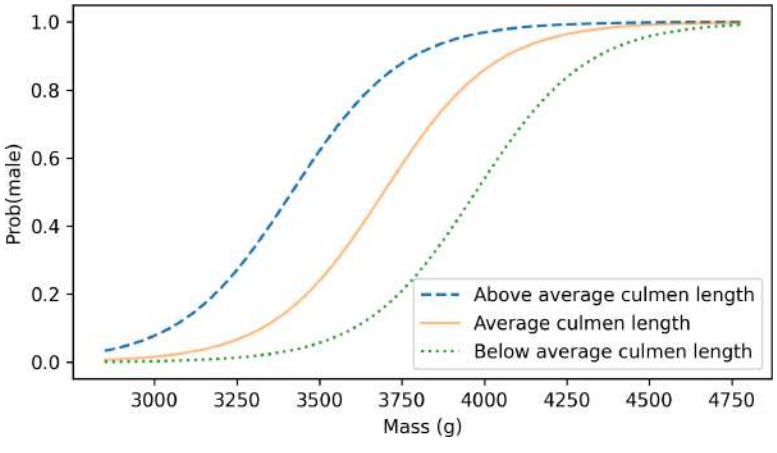
To understand this model, let's look at some of its predictions. We'll use the following function, which takes a sequence of masses and a specific value for `culmen_length`. It sets the other measurements to their mean values, computes predicted probabilities as a function of mass, and plots the results:

```
def plot_predictions(mass_range, culmen_length, **options):
    """Plot predicted probabilities as a function of mass."""
    df = pd.DataFrame({"mass": mass_range})
    df["flipper_length"] = adelie["flipper_length"].mean()
    df["culmen_length"] = culmen_length
    df["culmen_depth"] = adelie["culmen_depth"].mean()
    fit_ys = result.predict(df)
    plt.plot(mass_range, fit_ys, **options)
```

Here's what the results look like for three values of `culmen_length`: one standard deviation above average, average, and one standard deviation below average:

```
culmen_length = adelie["culmen_length"]
m, s = culmen_length.mean(), culmen_length.std()
```

```
plot_predictions(mass_range, m + s, ls="--", label="Above average culmen length")
plot_predictions(mass_range, m, alpha=0.5, label="Average culmen length")
plot_predictions(mass_range, m - s, ls=":", label="Below average culmen length")

decorate(xlabel="Mass (g)", ylabel="Prob(male)")
```



As we saw in the simpler model, heavier penguins are more likely to be male. Also, at any weight, a penguin with a longer bill is more likely to be male.

This model is more useful than it might seem—in fact, it is similar to models that were used in the original research paper this dataset was collected for. The primary topic of that research is sexual dimorphism, which is the degree to which male and female bodies differ. One way to quantify dimorphism is to use measurements to classify males and females. In a species with higher dimorphism, we expect these classifications to be more accurate.

To test this methodology, let's try the same model on a different species. In addition to the Adélie penguins we've worked with so far, the dataset also contains measurements from 123 Gentoo penguins. We'll use the following function to select them:

```
def get_species(penguins, species):
    df = penguins.query(f'Species.str.startswith("{species}")').copy()
    df["y"] = (df["Sex"] == "MALE").astype(int)
    return df
```

```
gentoo = get_species(penguins, "Gentoo")
len(gentoo)
```

```
123
```

Here are the results of the logistic regression model:

```
formula = "y ~ mass + flipper_length + culmen_length + culmen_depth"
model = smf.logit(formula, data=gentoo)
result = model.fit(disp=False)
display_summary(result)
```

	coef	std err	z	P>|z|	[0.025	0.975]
Intercept	-173.9123	62.326	-2.790	0.005	-296.069	-51.756
mass	0.0105	0.004	2.948	0.003	0.004	0.017
flipper_length	0.2839	0.183	1.549	0.121	-0.075	0.643
culmen_length	0.2734	0.285	0.958	0.338	-0.286	0.833
culmen_depth	3.0843	1.291	2.389	0.017	0.554	5.614

Pseudo R-squared: 0.848

The pseudo R^2 value is 0.848, higher than what we got with Adélie penguins (0.662). That means Gentoo penguins can be classified more accurately using physical measurements, compared to Adélie penguins, which suggests that Gentoo penguins are more dimorphic.

Glossary

regression
: A method for estimating coefficients that fit a model to data.

response variables
: The variables a regression model tries to predict, also known as dependent variables.

explanatory variables
: The variables a model uses to predict the response variables, also known as independent variables.

simple regression
: A regression with one response variable and one explanatory variable.

multiple regression
: A regression with multiple explanatory variables, but only one response variable.

coefficients
: In a regression model, the coefficients are the intercept and the estimated slopes for the explanatory variables.

categorical variable
: A variable that can have one of a discrete set of values, usually not numerical.

control variable
: A variable included in a regression to separate the direct effect of an explanatory variable from an indirect effect.

generalized linear models
: A set of regression models based on different mathematical relationships between the explanatory and response variables.

logistic regression
: A generalized linear model used when the response variable has only two possible values.

Exercises

Exercise 11.1

Are baby boys heavier than baby girls? To answer this question, we'll use the NSFG data again.

Fit a linear regression model with `totalwgt_lb` as the response variable and `babysex` as a categorical explanatory variable—the value of this variable is `1` for boys and `2` for girls. What is the estimated difference in weight? Is it statistically significant? What if you control for the mother's age—does maternal age account for some or all of the apparent difference?

Exercise 11.2

The Trivers-Willard hypothesis suggests that for many mammals the sex ratio depends on "maternal condition"—that is, factors like the mother's age, size, health, and social status. Some studies have shown this effect among humans, but results are mixed.

Let's see if there is a relationship between mother's age and the probability of having a boy. Fit a logistic regression model with the baby's sex as the response variable and mother's age as an explanatory variable. Are older mothers more or less likely to have boys? What if you use a quadratic model of maternal age?

Exercise 11.3

For the Adelie penguins, fit a linear regression model that predicts penguin weights as a function of `flipper_length`, `culmen_depth`, and `Sex` as a categorical variable. If we control for flipper length and culmen depth, how much heavier are male penguins? Generate and plot predictions for a range of flipper lengths, for male and female penguins, with `culmen_depth` set to its average value.

Exercise 11.4

Let's see if Chinstrap penguins are more or less dimorphic than the other penguin species in the dataset, as quantified by the pseudo R^2 value of the model. Use `get_species` to select the Chinstrap penguins, then use logistic regression to fit a logistic regression model with sex as the response variable and all four measurements as explanatory variables. How does the pseudo R^2 value compare to the other models?

CHAPTER 12

Time Series Analysis

A **time series** is a sequence of measurements from a system that varies in time. Many of the tools we used in previous chapters, like regression, can also be used with time series. But there are additional methods that are particularly useful for this kind of data.

As examples, we'll look at two datasets: renewable electricity generation in the United States from 2001 to 2024, and weather data over the same interval. We will develop methods to decompose a time series into a long-term trend and a repeated seasonal component. We'll use linear regression models to fit and forecast trends. And we'll try out a widely used model for analyzing time series data, with the formal name "autoregressive integrated moving average" and the easier-to-say acronym ARIMA.

Electricity

As an example of time-series data, we'll use a dataset from the US Energy Information Administration—it includes total electricity generation per month from renewable sources from 2001 to 2024. Instructions for downloading the data are in the notebook for this chapter.

After loading the data, we have to make some transformations to get it into a format that's easy to work with:

```
elec = (
    pd.read_csv("Net_generation_for_all_sectors.csv", skiprows=4)
    .drop(columns=["units", "source key"])
    .set_index("description")
    .replace("--", np.nan)
    .transpose()
    .astype(float)
)
```

In the reformatted dataset, each column is a sequence of monthly totals in gigawatt hours (GWh). Here are the column labels, showing the different sources of electricity, or "sectors":

```
elec.columns
```

```
Index(['Net generation for all sectors', 'United States',
       'United States : all fuels (utility-scale)', 'United States : nuclear',
       'United States : conventional hydroelectric',
       'United States : other renewables', 'United States : wind',
       'United States : all utility-scale solar', 'United States : geothermal',
       'United States : biomass',
       'United States : hydro-electric pumped storage',
       'United States : all solar',
       'United States : small-scale solar photovoltaic'],
      dtype='object', name='description')
```

The labels in the index are strings indicating months and years—here are the first 12:

```
elec.index[:12]
```

```
Index(['Jan 2001', 'Feb 2001', 'Mar 2001', 'Apr 2001', 'May 2001', 'Jun 2001',
       'Jul 2001', 'Aug 2001', 'Sep 2001', 'Oct 2001', 'Nov 2001', 'Dec 2001'],
      dtype='object')
```

It will be easier to work with this data if we replace these strings with Pandas `Timestamp` objects. We can use the `date_range` function to generate a sequence of `Timestamp` objects, starting in January 2001 with the frequency code `"ME"`, which stands for "month end," so it fills in the last day of each month:

```
elec.index = pd.date_range(start="2001-01", periods=len(elec), freq="ME")
elec.index[:6]
```

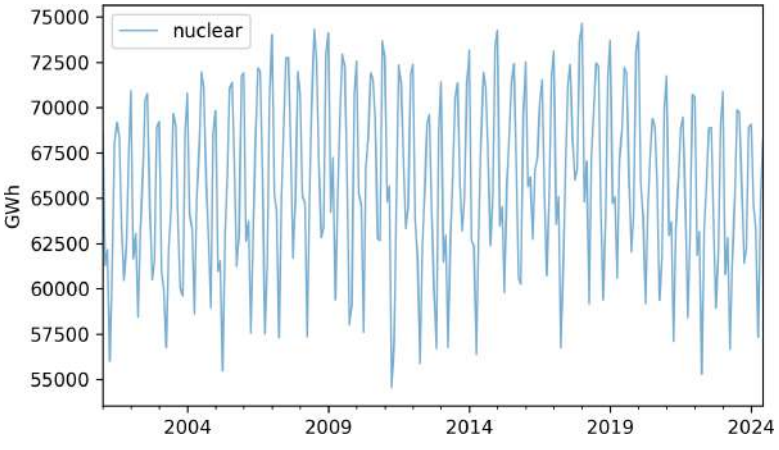
```
DatetimeIndex(['2001-01-31', '2001-02-28', '2001-03-31', '2001-04-30',
               '2001-05-31', '2001-06-30'],
              dtype='datetime64[ns]', freq='ME')
```

Now the index is a `DataTimeIndex` with the data type `datetime64[ns]`, which is defined in NumPy—`64` means each label uses 64 bits, and `ns` means it has nanosecond precision.

Decomposition

As a first example, we'll look at how electricity generation from nuclear reactors has changed over the interval from January 2001 to June 2024, and we'll decompose the time series into a long-term trend and a periodic component. Here are monthly totals of electricity generation from nuclear reactors in the United States:

```
nuclear = elec["United States : nuclear"]
nuclear.plot(label="nuclear", **actual_options)

decorate(ylabel="GWh")
```



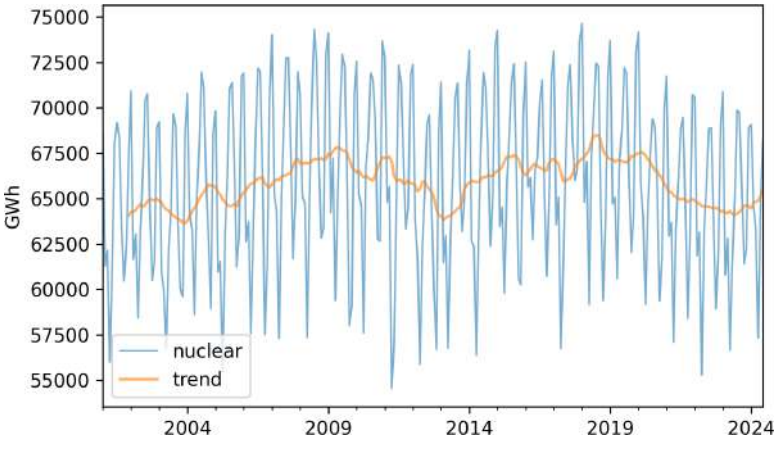
It looks like there are some increases and decreases, but they are hard to see clearly because there are large variations from month to month. To see the long-term trend more clearly, we can use the `rolling` and `mean` methods to compute a **moving average**:

```
trend = nuclear.rolling(window=12).mean()
```

The `window=12` argument selects overlapping intervals of 12 months, so the first interval contains 12 measurements starting with the first, the second interval contains 12 measurements starting with the second, and so on. For each interval, we compute the mean production.

Here's what the results look like, along with the original data:

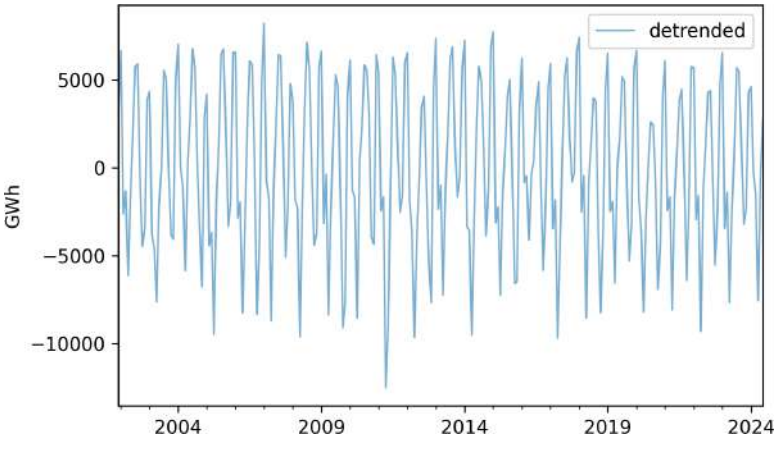
```
nuclear.plot(label="nuclear", **actual_options)
trend.plot(label="trend", **trend_options)
decorate(ylabel="GWh")
```



The trend is still quite variable. We could smooth it more by using a longer **window**, but we'll stick with the 12-month window for now.

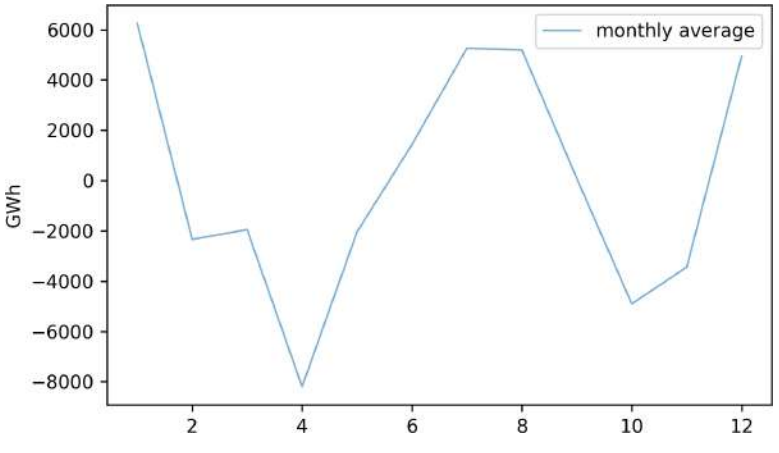
If we subtract the trend from the original data, the result is a "detrended" time series, which means that the long-term mean is close to constant. Here's what it looks like:

```
detrended = (nuclear - trend).dropna()
detrended.plot(label="detrended", **actual_options)
decorate(ylabel="GWh")
```



It seems like there is a repeating annual pattern, which makes sense because demand for electricity varies from one season to another, as it is used to generate heat in the winter and run air conditioning in the summer. To describe this annual pattern we can select the month part of the `datetime` objects in the index, group the data by month, and compute average production. Here's what the monthly averages look like:

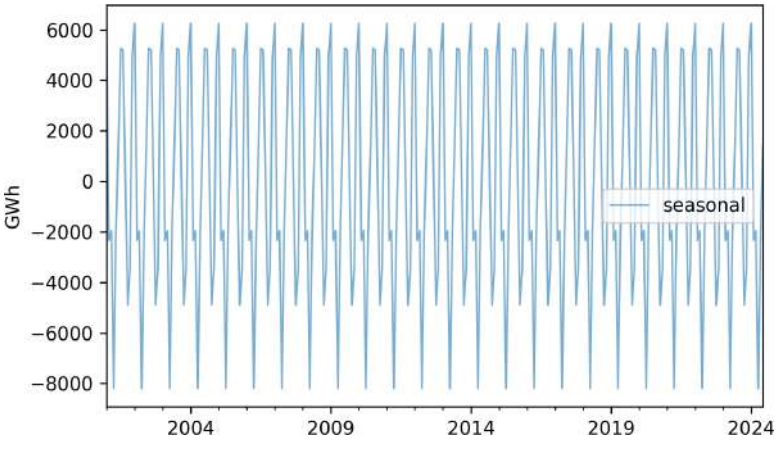
```
monthly_averages = detrended.groupby(detrended.index.month).mean()
monthly_averages.plot(label="monthly average", **actual_options)
decorate(ylabel="GWh")
```



On the x-axis, month 1 is January and month 12 is December. Electricity production is highest during the coldest and warmest months, and lowest during April and October.

We can use `monthly_averages` to construct the seasonal component of the data, which is a series the same length as `nuclear`, where the element for each month is the average for that month. Here's what it looks like:

```
seasonal = monthly_averages[nuclear.index.month]
seasonal.index = nuclear.index
seasonal.plot(label="seasonal", **actual_options)
decorate(ylabel="GWh")
```



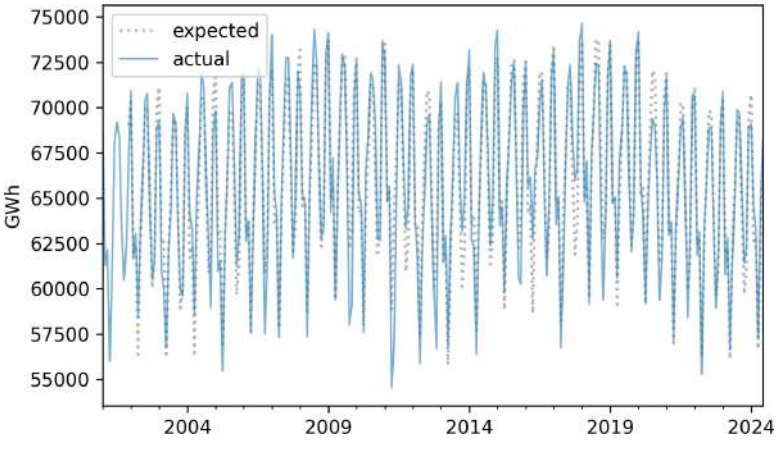
Each 12-month period is identical to the others.

The sum of the trend and the seasonal component represents the expected value for each month:

```
expected = trend + seasonal
```

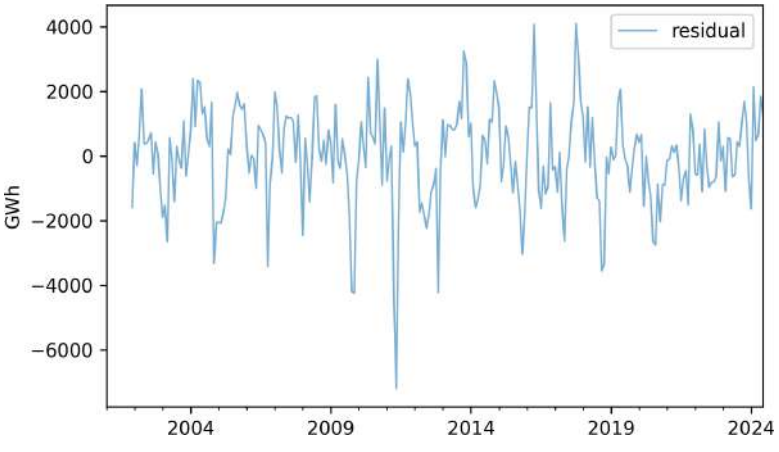
Here's what it looks like compared to the original series:

```
expected.plot(label="expected", **pred_options)
nuclear.plot(label="actual", **actual_options)
decorate(ylabel="GWh")
```



If we subtract this sum from the original series, the result is the residual component, which represents the departure from the expected value for each month:

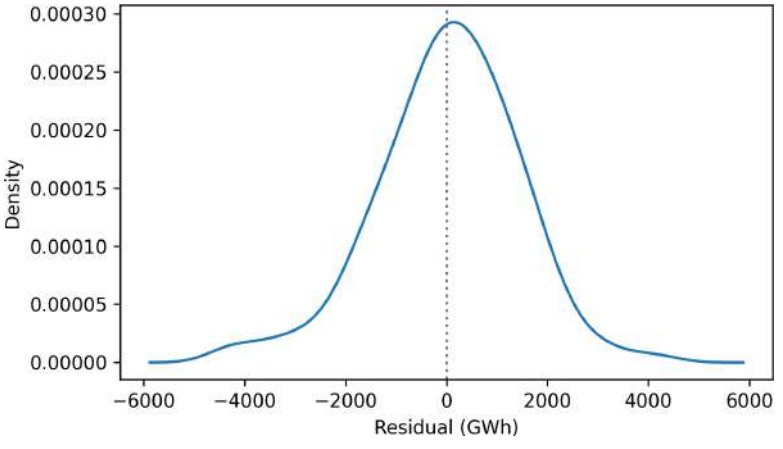
```
resid = nuclear - expected
resid.plot(label="residual", **actual_options)
decorate(ylabel="GWh")
```



We can think of the residual as the sum of everything in the world that affects energy production, but is not explained by the long-term trend or the seasonal component. Among other things, that sum includes weather, equipment that's down for maintenance, and changes in demand due to specific events. Since the residual is the sum of many unpredictable, and sometimes unknowable, factors, we often treat it as a random quantity.

Here's what the distribution of the residuals look like:

```
from thinkstats import plot_kde

plot_kde(resid.dropna())
decorate(xlabel="Residual (GWh)")
```



It resembles the bell curve of the normal distribution, which is consistent with the assumption that it is the sum of many random contributions.

To quantify how well this model describes the original series, we can compute the coefficient of determination, which indicates how much smaller the variance of the residuals is, compared to the variance of the original series:

```
rsquared = 1 - resid.var() / nuclear.var()
rsquared
```

```
0.9054559977517084
```

The R^2 value is about 0.92, which means that the long-term trend and seasonal component account for 92% of the variability in the series. This R^2 is substantially higher than the ones we saw in the previous chapter, but that's common with time series data—especially in a case like this where we've constructed the model to resemble the data.

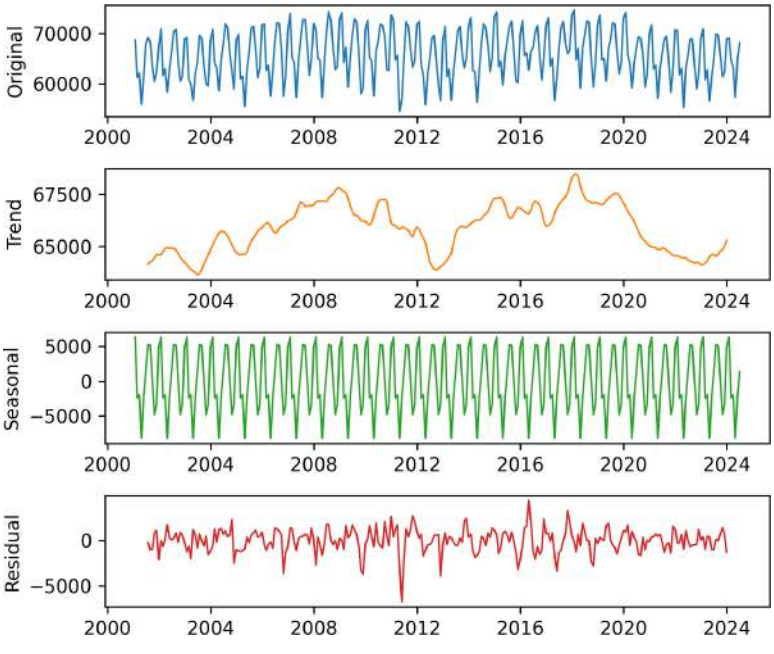
The process we've just walked through is called **seasonal decomposition**. StatsModels provides a function that does it, called `seasonal_decompose`:

```
from statsmodels.tsa.seasonal import seasonal_decompose

decomposition = seasonal_decompose(nuclear, model="additive", period=12)
```

The `model="additive"` argument indicates the additive model, so the series is decomposed into the sum of a trend, seasonal component, and residual. We'll see the multiplicative model soon. The `period=12` argument indicates that the duration of the seasonal component is 12 months.

The result is an object that contains the three components. The notebook for this chapter provides a function that plots them:

```
plot_decomposition(nuclear, decomposition)
```



The results are similar to those we computed ourselves, with small differences due to the details of the implementation.

This kind of seasonal decomposition provides insight into the structure of a time series. As we'll see in the next section, it is also useful for making forecasts.

Prediction

We can use the results from seasonal decomposition to predict the future. To demonstrate, we'll use the following function to split the time series into a **training series**, which we'll use to generate predictions, and a **test series**, which we'll use to see whether they are accurate:

```
def split_series(series, n=60):
    training = series.iloc[:-n]
    test = series.iloc[-n:]
    return training, test
```

With `n=60`, the duration of the test series is five years, starting in July 2019:

```
training, test = split_series(nuclear)
test.index[0]
```

```
Timestamp('2019-07-31 00:00:00')
```

Now, suppose it's June 2019 and you are asked to generate a five-year forecast for electricity production from nuclear generators. To answer this question, we'll use the training data to make a model and then use the model to generate predictions. We'll start with a seasonal decomposition of the training data:

```
decomposition = seasonal_decompose(training, model="additive", period=12)
trend = decomposition.trend
```

Now we'll fit a linear model to the trend. The explanatory variable, `months`, is the number of months from the beginning of the series:

```
import statsmodels.formula.api as smf

months = np.arange(len(trend))
data = pd.DataFrame({"trend": trend, "months": months}).dropna()
results = smf.ols("trend ~ months", data=data).fit()
```

Here is a summary of the results:

```
from thinkstats import display_summary

display_summary(results)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	6.482e+04	131.524	492.869	0.000	6.46e+04	6.51e+04
months	10.9886	1.044	10.530	0.000	8.931	13.046

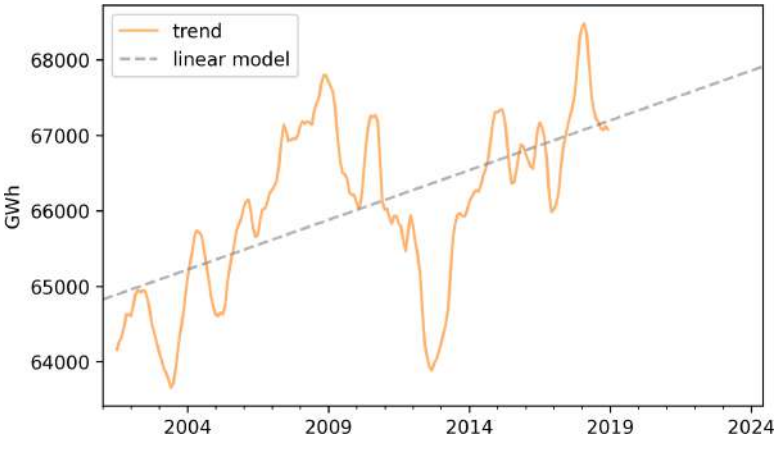
R-squared: 0.3477

The R^2 value is about 0.35, which suggests that the model does not fit the data particularly well. We can get a better sense of that by plotting the fitted line. We'll use the `predict` method to compute expected values for the training and test data:

```
months = np.arange(len(training) + len(test))
df = pd.DataFrame({"months": months})
pred_trend = results.predict(df)
pred_trend.index = nuclear.index
```

Here's the trend component and the linear model:

```
trend.plot(**trend_options)
pred_trend.plot(label="linear model", **model_options)
decorate(ylabel="GWh")
```



There's a lot going on that's not captured by the linear model, but it looks like there is a generally increasing trend.

Next we'll use the seasonal component from the decomposition to compute a `Series` of monthly averages:

```
seasonal = decomposition.seasonal
monthly_averages = seasonal.groupby(seasonal.index.month).mean()
```

We can predict the seasonal component by looking up the dates from the fitted line in `monthly_averages`:

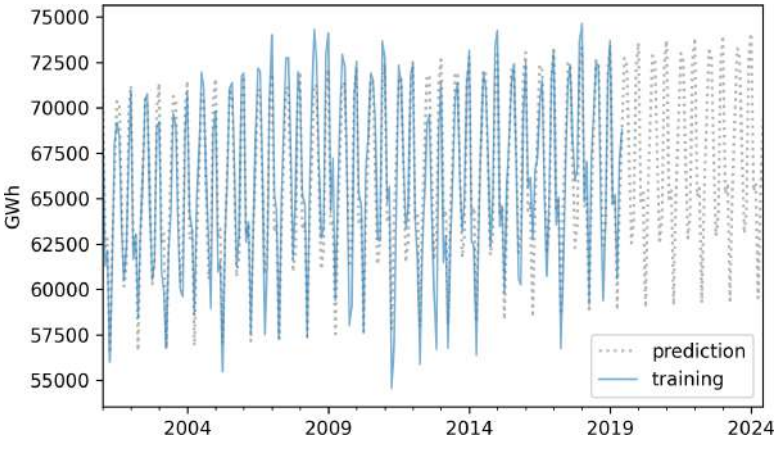
```
pred_seasonal = monthly_averages[pred_trend.index.month]
pred_seasonal.index = pred_trend.index
```

Finally, to generate predictions, we'll add the seasonal component to the trend:

```
pred = pred_trend + pred_seasonal
```

Here's the training data and the predictions:

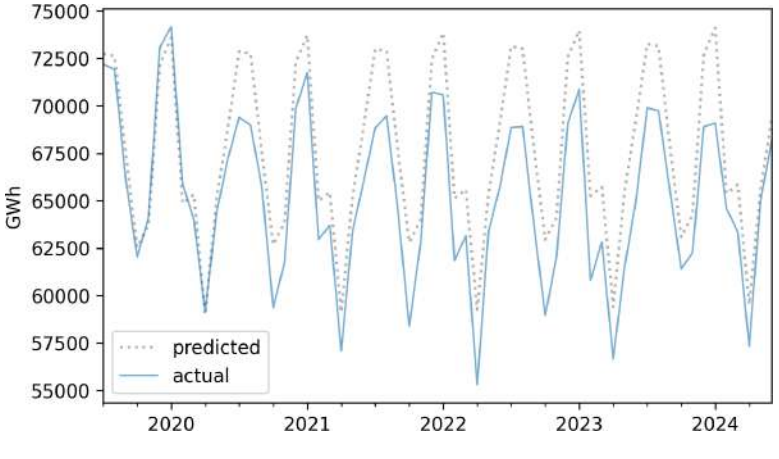
```
pred.plot(label="prediction", **pred_options)
training.plot(label="training", **actual_options)
decorate(ylabel="GWh")
```



The predictions fit the training data reasonably well, and the forecast looks like a reasonable projection, based on the assumption that the long-term trend will continue.

Now, from the vantage point of the future, let's see how accurate this forecast turned out to be. Here are the predicted and actual values for the five-year interval from July 2019:

```
forecast = pred[test.index]
forecast.plot(label="predicted", **pred_options)
test.plot(label="actual", **actual_options)
decorate(ylabel="GWh")
```



The first year of the forecast was pretty good, but production from nuclear reactors in 2020 was lower than expected—possibly due to the COVID-19 pandemic—and it never returned to the long-term trend.

To quantify the accuracy of the predictions, we'll use the mean absolute percentage error (MAPE), which the following function computes:

```
def MAPE(predicted, actual):
    ape = np.abs(predicted - actual) / actual
    return np.mean(ape) * 100
```

In this example, the predictions are off by 3.81% on average:

```
MAPE(forecast, test)
```

```
3.811940747879257
```

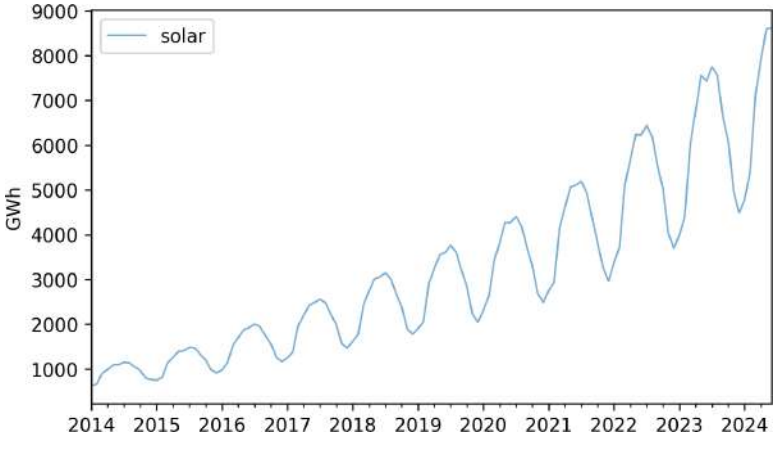
We'll come back to this example later in the chapter and see if we can do better with a different model.

Multiplicative Model

The additive model we used in the previous section assumes that the time series is the *sum* of a long-term trend, a seasonal component, and a residual—which implies that the magnitude of the seasonal component and the residuals does not vary over time.

As an example that violates this assumption, let's look at small-scale solar electricity production since 2014:

```
solar = elec["United States : small-scale solar photovoltaic"].dropna()
solar.plot(label="solar", **actual_options)
decorate(ylabel="GWh")
```



Over this interval, total production has increased several times over. And it's clear that the magnitude of seasonal variation has increased as well.

If we suppose that the magnitudes of seasonal and random variation are proportional to the magnitude of the trend, that suggests an alternative to the additive model in which the time series is the *product* of the three components.

To try out this multiplicative model, we'll split this series into training and test sets:

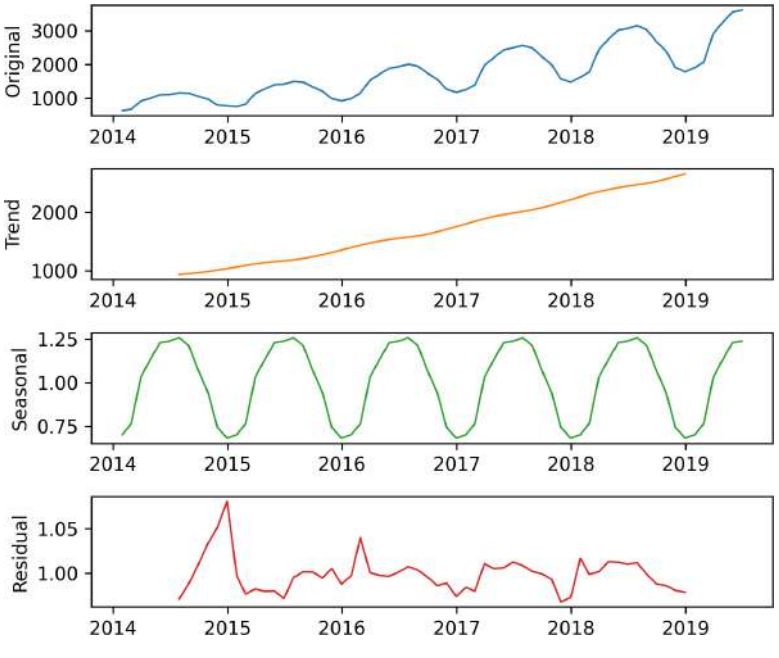
```
training, test = split_series(solar)
```

And call `seasonal_decompose` with the `model="multiplicative"` argument:

```
decomposition = seasonal_decompose(training, model="multiplicative", period=12)
```

Here's what the results look like:

```
plot_decomposition(training, decomposition)
```



Now the seasonal and residual components are multiplicative factors. So, it looks like the seasonal component varies from about 25% below the trend to 25% above. And the residual component is usually less than 5% either way, with the exception of some larger factors in the first period. We can extract the components of the model like this:

```
trend = decomposition.trend
seasonal = decomposition.seasonal
resid = decomposition.resid
```

The R^2 value of this model is very high:

```
rsquared = 1 - resid.var() / training.var()
rsquared
```

```
0.9999999992978134
```

The production of a solar panel is largely a function of the sunlight it's exposed to, so it makes sense that production follows an annual cycle so closely.

To predict the long-term trend, we'll use a quadratic model:

```
months = range(len(training))
data = pd.DataFrame({"trend": trend, "months": months}).dropna()
results = smf.ols("trend ~ months + I(months**2)", data=data).fit()
```

In the Patsy formula, the substring `I(months**2)` adds a quadratic term to the model, so we don't have to compute it explicitly. Here are the results:

```
display_summary(results)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	766.1962	13.494	56.782	0.000	739.106	793.286
months	22.2153	0.938	23.673	0.000	20.331	24.099
I(months ** 2)	0.1762	0.014	12.480	0.000	0.148	0.205

R-squared: 0.9983

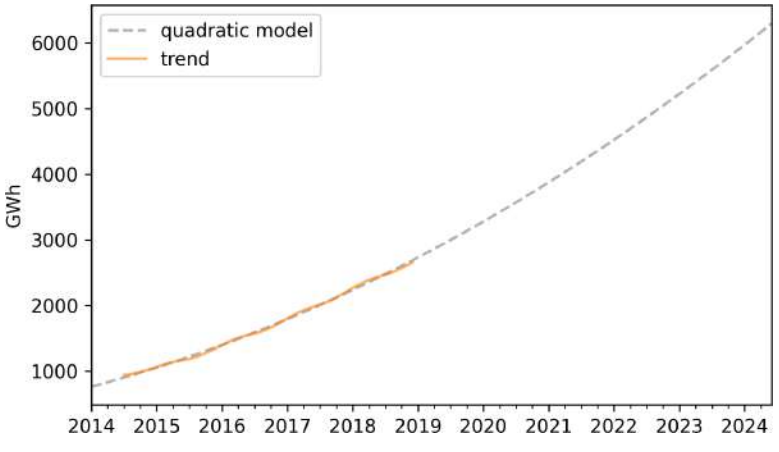
The p-values of the linear and quadratic terms are very small, which suggests that the quadratic model captures more information about the trend than a linear model would—and the R^2 value is very high.

Now we can use the model to compute the expected value of the trend for the past and future:

```
months = range(len(solar))
df = pd.DataFrame({"months": months})
pred_trend = results.predict(df)
pred_trend.index = solar.index
```

Here's what it looks like:

```
pred_trend.plot(label="quadratic model", **model_options)
trend.plot(**trend_options)
decorate(ylabel="GWh")
```



The quadratic model fits the past trend well. Now we can use the seasonal component to predict future seasonal variation:

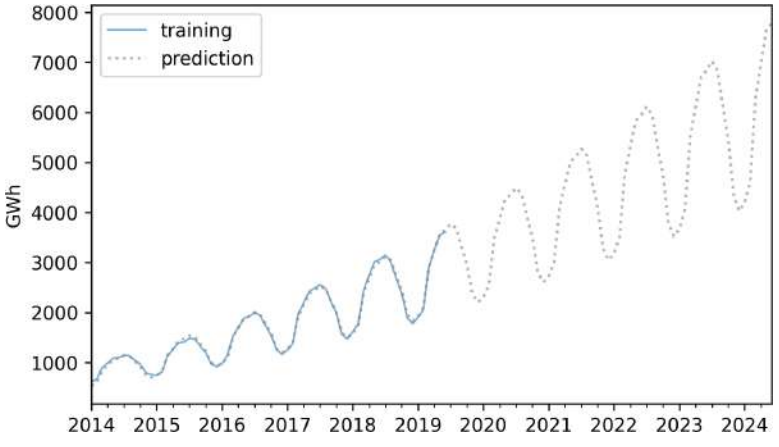
```
monthly_averages = seasonal.groupby(seasonal.index.month).mean()
pred_seasonal = monthly_averages[pred_trend.index.month]
pred_seasonal.index = pred_trend.index
```

Finally, to compute **retrodictions** for past values and predictions for the future, we multiply the trend and the seasonal component:

```
pred = pred_trend * pred_seasonal
```

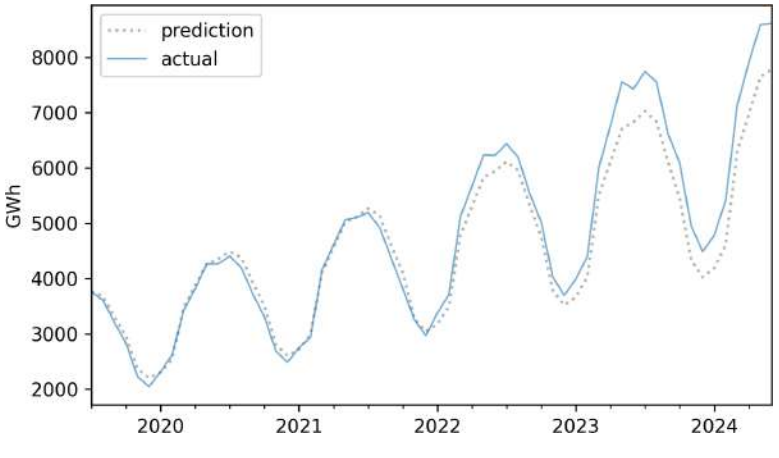
Here is the result along with the training data:

```
training.plot(label="training", **actual_options)
pred.plot(label="prediction", **pred_options)
decorate(ylabel="GWh")
```



The retrodictions fit the training data well and the predictions seem plausible—now let's see if they turned out to be accurate. Here are the predictions along with the test data:

```
future = pred[test.index]

future.plot(label="prediction", **pred_options)
test.plot(label="actual", **actual_options)
decorate(ylabel="GWh")
```



For the first three years, the predictions are very good. After that, it looks like actual growth exceeded expectations.

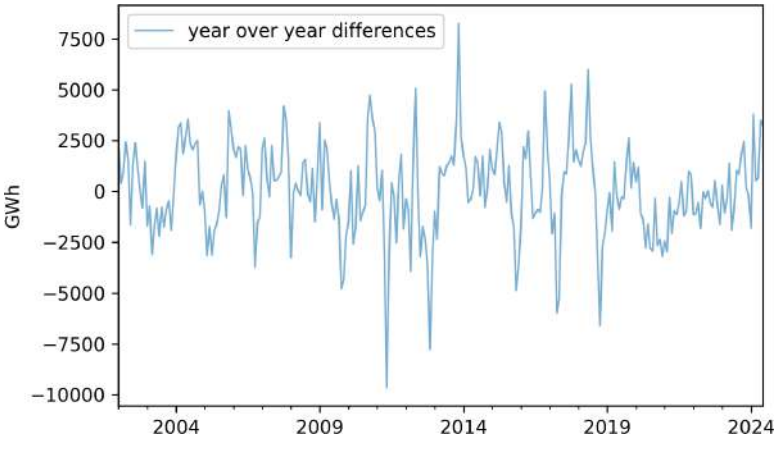
In this example, seasonal decomposition worked well for modeling and predicting solar production, but in the previous example, it was not very effective for nuclear production. In the next section, we'll try a different approach, autoregression.

Autoregression

The first idea of autoregression is that the future will be like the past. For example, in the time series we've looked at so far, there is a clear annual cycle. So if you are asked to make a prediction for next June, a good starting place would be last June.

To see how well that might work, let's go back to `nuclear`, which contains monthly electricity production from nuclear generators, and compute differences between the same month in successive years, which are called "year-over-year" differences:

```
diff = (nuclear - nuclear.shift(12)).dropna()
diff.plot(label="year over year differences", **actual_options)
decorate(ylabel="GWh")
```



The magnitudes of these differences are substantially smaller than the magnitudes of the original series, which suggests the second idea of autoregression, which is that it might be easier to predict these differences, rather than the original values.

Toward that end, let's see if there are correlations between successive elements in the series of differences. If so, we could use those correlations to predict future values based on previous values.

I'll start by making a `DataFrame`, putting the differences in the first column and putting the same differences—shifted by 1, 2, and 3 months—into successive columns. These columns are named `lag1`, `lag2`, and `lag3`, because the series they contain have been **lagged** or delayed:

```
df_ar = pd.DataFrame({"diff": diff})
for lag in [1, 2, 3]:
    df_ar[f"lag{lag}"] = diff.shift(lag)

df_ar = df_ar.dropna()
```

Here are the correlations between these columns:

```
df_ar.corr()[["diff"]]
```

	diff
diff	1.000000
lag1	0.562212
lag2	0.292454
lag3	0.222228

These correlations are called lagged correlations or **autocorrelations**—the prefix "auto" indicates that we're taking the correlation of the series with itself. As a special case, the correlation between `diff` and `lag1` is called **serial correlation** because it is the correlation between successive elements in the series.

These correlation are strong enough to suggest that they should help with prediction, so let's put them into a multiple regression. The following function uses the columns from the `DataFrame` to make a Patsy formula with the first column as the response variable and the other columns as explanatory variables:

```
def make_formula(df):
    """Make a Patsy formula from column names."""
    y = df.columns[0]
    xs = " + ".join(df.columns[1:])
    return f"{y} ~ {xs}"
```

Here are the results of a linear model that predicts the next value in a sequence based on the previous three values:

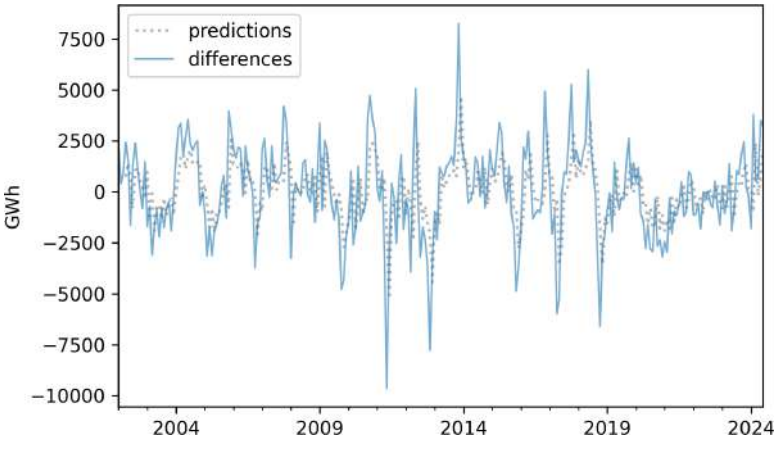
```
formula = make_formula(df_ar)
results_ar = smf.ols(formula=formula, data=df_ar).fit()
display_summary(results_ar)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	24.2674	114.674	0.212	0.833	-201.528	250.063
lag1	0.5847	0.061	9.528	0.000	0.464	0.706
lag2	-0.0908	0.071	-1.277	0.203	-0.231	0.049
lag3	0.1026	0.062	1.666	0.097	-0.019	0.224

R-squared: 0.3239

Now we can use the `predict` method to generate predictions for the past values in the series. Here's what these retrodictions look like compared to the data:

```
pred_ar = results_ar.predict(df_ar)
pred_ar.plot(label="predictions", **pred_options)
diff.plot(label="differences", **actual_options)
decorate(ylabel="GWh")
```



The predictions are good in some places, but the R^2 value is only about 0.319, so there is room for improvement:

```
resid_ar = (diff - pred_ar).dropna()
R2 = 1 - resid_ar.var() / diff.var()
R2
```

```
0.3190252265690783
```

One way to improve the predictions is to compute the residuals from this model and use another model to predict the residuals—which is the third idea of autoregression.

Moving Average

Suppose it's June 2019, and you are asked to make a prediction for June 2020. Your first guess might be that this year's value will be repeated next year.

Now suppose it's May 2020, and you are asked to revise your prediction for June 2020. You could use the results from the last three months, and the autocorrelation model from the previous section, to predict the year-over-year difference.

Finally, suppose you check the predictions for the last few months, and see that they have been consistently too low. This suggests that the prediction for next month

might also be too low, so you could revise it upward. The underlying assumption is that recent prediction errors predict future prediction errors.

To see whether they do, we can make a `DataFrame` with the residuals from the autoregression model in the first column, and lagged versions of the residuals in the other columns. For this example, I'll use lags of 1 and 6 months:

```
df_ma = pd.DataFrame({"resid": resid_ar})

for lag in [1, 6]:
    df_ma[f"lag{lag}"] = resid_ar.shift(lag)

df_ma = df_ma.dropna()
```

We can use `ols` to make an autoregression model for the residuals. This part of the model is called a "moving average" because it reduces variability in the predictions in a way that's analogous to the effect of a moving average. I don't find that term particularly helpful, but it is conventional.

Anyway, here's a summary of the autoregression model for the residuals:

```
formula = make_formula(df_ma)
results_ma = smf.ols(formula=formula, data=df_ma).fit()
display_summary(results_ma)
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-14.0016	114.697	-0.122	0.903	-239.863	211.860
lag1	0.0014	0.062	0.023	0.982	-0.120	0.123
lag6	-0.1592	0.063	-2.547	0.011	-0.282	-0.036

R-squared: 0.0247

The R^2 is quite small, so it looks like this part of the model won't help very much. But the p-value for the six-month lag is small, which suggests that it contributes more information than we'd expect by chance.

Now we can use the model to generate retrodictions for the residuals:

```
pred_ma = results_ma.predict(df_ma)
```

Then, to generate retrodictions for the year-over-year differences, we add the adjustment from the second model to the retrodictions from the first:

```
pred_diff = pred_ar + pred_ma
```

The R^2 value for the sum of the two models is about 0.332, which is just a little better than the result without the moving average adjustment (0.319):

```
resid_ma = (diff - pred_diff).dropna()
R2 = 1 - resid_ma.var() / diff.var()
R2
```

```
0.3315101001391231
```

Next we'll use these year-over-year differences to generate retrodictions for the original values.

Retrodiction with Autoregression

To generate retrodictions, we'll start by putting the year-over-year differences in a `Series` that's aligned with the index of the original:

```
pred_diff = pd.Series(pred_diff, index=nuclear.index)
```

Using `isna` to check for `NaN` values, we find that the first 21 elements of the new `Series` are missing:

```
n_missing = pred_diff.isna().sum()
n_missing
```

```
21
```

That's because we shifted the `Series` by 12 months to compute year-over-year differences, then we shifted the differences 3 months for the first autoregression model, and we shifted the residuals of the first model by 6 months for the second model. Each time we shift a `Series` like this, we lose a few values at the beginning, and the sum of these shifts is 21.

So before we can generate retrodictions, we have to prime the pump by copying the first 21 elements from the original into a new `Series`:

```
pred_series = pd.Series(index=nuclear.index, dtype=float)
pred_series.iloc[:n_missing] = nuclear.iloc[:n_missing]
```

Now we can run the following loop, which fills in the elements from index 21 (which is the 22nd element) to the end. Each element is the sum of the value from the previous year and the predicted year-over-year difference:

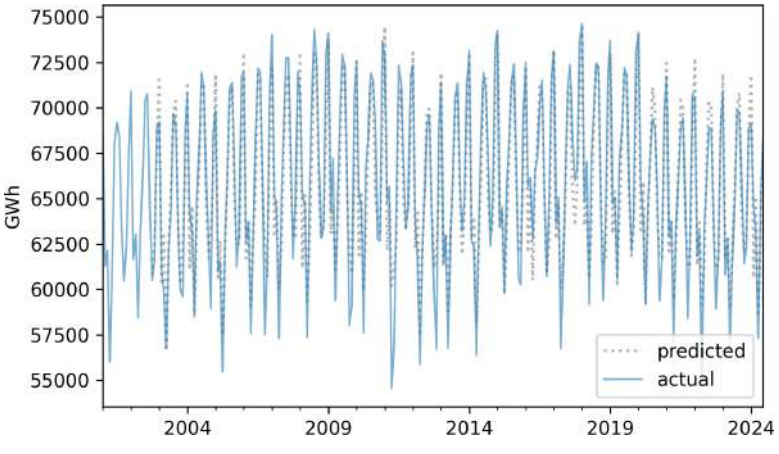
```
for i in range(n_missing, len(pred_series)):
    pred_series.iloc[i] = pred_series.iloc[i - 12] + pred_diff.iloc[i]
```

Now we'll replace the elements we copied with `NaN` so we don't get credit for "predicting" the first 21 values perfectly:

```
pred_series[:n_missing] = np.nan
```

Here's what the retrodictions look like compared to the original:

```
pred_series.plot(label="predicted", **pred_options)
nuclear.plot(label="actual", **actual_options)
decorate(ylabel="GWh")
```



They look pretty good, and the R^2 value is about 0.86:

```
resid = (nuclear - pred_series).dropna()
R2 = 1 - resid.var() / nuclear.var()
R2
```

```
0.8586566911201015
```

The model we used to compute these retrodictions is called SARIMA, which is one of a family of models called ARIMA. Each part of these acronyms refers to an element of the model:

- *S* stands for seasonal, because the first step was to compute differences between values separated by one seasonal period.
- *AR* stands for autoregression, which we used to model lagged correlations in the differences.
- *I* stands for integrated, because the iterative process we used to compute `pred_series` is analogous to integration in calculus.

- *MA* stands for moving average, which is the conventional name for the second autoregression model we ran with the residuals from the first.

ARIMA models are powerful and versatile tools for modeling time series data.

ARIMA

StatsModel provides a library called `tsa`, which stands for "time series analysis"—it includes a function called `ARIMA` that fits ARIMA models and generates forecasts.

To fit the SARIMA model we developed in the previous sections, we'll call this function with two tuples as arguments: `order` and `seasonal_order`. Here are the values in `order` that correspond to the model we used in the previous sections:

```
order = ([1, 2, 3], 0, [1, 6])
```

The values in `order` indicate:

- Which lags should be included in the AR model—in this example it's the first three.
- How many times it should compute differences between successive elements—in this example it's 0 because we computed a seasonal difference instead, and we'll get to that in a minute.
- Which lags should be included in the MA model—in this example it's the first and sixth.

Now here are the values in `seasonal_order`:

```
seasonal_order = (0, 1, 0, 12)
```

The first and third elements are 0, which means that this model does not include seasonal AR or seasonal MA. The second element is 1, which means it computes seasonal differences—and the last element is the seasonal period.

Here's how we use `ARIMA` to make and fit this model:

```
import statsmodels.tsa.api as tsa

model = tsa.ARIMA(nuclear, order=order, seasonal_order=seasonal_order)
results_arima = model.fit()
display_summary(results_arima)
```

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.0458	0.379	0.121	0.904	-0.697	0.788
ar.L2	-0.0035	0.116	-0.030	0.976	-0.230	0.223
ar.L3	0.0375	0.049	0.769	0.442	-0.058	0.133
ma.L1	0.2154	0.382	0.564	0.573	-0.533	0.964
ma.L6	-0.0672	0.019	-3.500	0.000	-0.105	-0.030
sigma2	3.473e+06	1.9e-07	1.83e+13	0.000	3.47e+06	3.47e+06

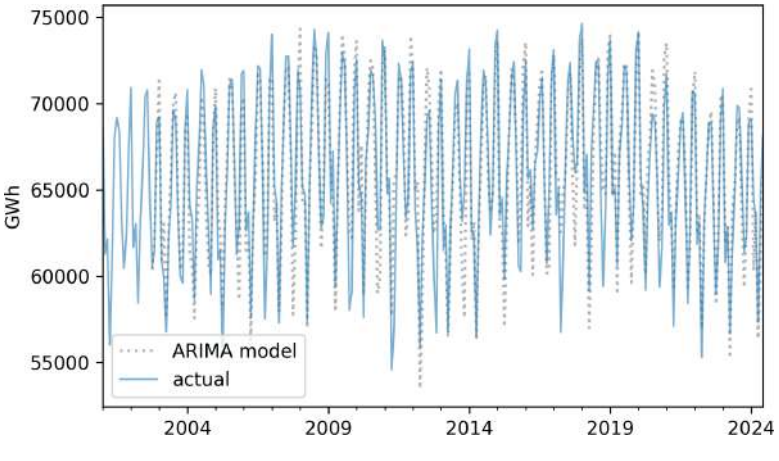
The results include estimated coefficients for the three lags in the AR model, the two lags in the MA model, and `sigma2`, which is the variance of the residuals.

From `results_arima` we can extract `fittedvalues`, which contains the retrodictions. For the same reason there were missing values at the beginning of the retrodictions we computed, there are incorrect values at the beginning of `fittedvalues`, which we'll drop:

```
fittedvalues = results_arima.fittedvalues[n_missing:]
```

The fitted values are similar to the ones we computed, but not exactly the same—probably because `ARIMA` handles the initial conditions differently:

```
fittedvalues.plot(label="ARIMA model", **pred_options)
nuclear.plot(label="actual", **actual_options)
decorate(ylabel="GWh")
```



The R^2 value is also similar but not precisely the same:

```
resid = fittedvalues - nuclear
R2 = 1 - resid.var() / nuclear.var()
R2
```

```
0.8262717330784233
```

The `ARIMA` function makes it easy to experiment with different versions of the model.

Prediction with ARIMA

The object returned by `ARIMA` provides a method called `get_forecast` that generates predictions. To demonstrate, we'll split the time series into a training and test set, and fit the same model to the training set:

```
training, test = split_series(nuclear)
model = tsa.ARIMA(training, order=order, seasonal_order=seasonal_order)
results_training = model.fit()
```

We can use the result to generate a forecast for the test set:

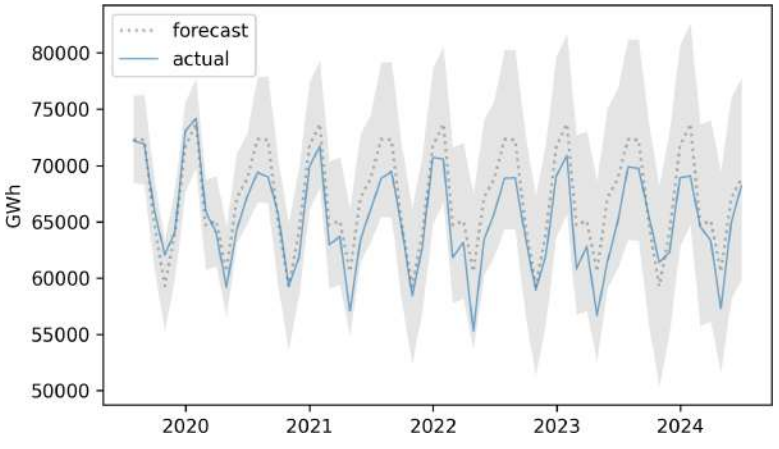
```
forecast = results_training.get_forecast(steps=len(test))
```

The result is an object that contains an attribute called `forecast_mean` and a function that returns a confidence interval:

```
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()
forecast_ci.columns = ["lower", "upper"]
```

We can plot the results like this and compare them to the actual time series:

```
plt.fill_between(
    forecast_ci.index,
    forecast_ci.lower,
    forecast_ci.upper,
    lw=0,
    color="gray",
    alpha=0.2,
)
plt.plot(forecast_mean.index, forecast_mean, label="forecast", **pred_options)
plt.plot(test.index, test, label="actual", **actual_options)
decorate(ylabel="GWh")
```



The actual values fall almost entirely within the confidence interval of the predictions. Here's the MAPE of the predictions:

```
MAPE(forecast_mean, test)
```

```
3.381754924564627
```

The predictions are off by 3.38% on average, somewhat better than the results we got from seasonal decomposition (3.81%).

ARIMA is more versatile than seasonal decomposition, and can often make better predictions. In this time series, the autocorrelations are not especially strong, so the advantage of ARIMA is modest.

Glossary

time series
: A dataset where each value is associated with a specific time, often representing measurements taken at regular intervals

seasonal decomposition
: A method of splitting a time series into a long-term trend, a repeating seasonal component, and a residual component

training series
: Part of a time series used to fit a model

test series
: Part of a time series used to check the accuracy of predictions generated by a model

retrodiction
: A prediction for a value observed in the past, often used to test or validate a model

window
: A sequence of consecutive values in a time series, used to compute a moving average

moving average
: A time series computed by averaging values in overlapping windows to smooth fluctuations

serial correlation
: The correlation between successive elements of a time series

autocorrelation
: A correlation between a time series and a shifted or lagged version of itself

lag
: The size of the shift in a serial correlation or autocorrelation

Exercises

Exercise 12.1

As an example of seasonal decomposition, let's model monthly average surface temperatures in the United States. We'll use a dataset from Our World in Data that includes "temperature [in Celsius] of the air measured 2 meters above the ground, encompassing land, sea, and in-land water surfaces," for most countries in the world from 1950 to 2024. Instructions for downloading the data are in the notebook for this chapter.

We can read the data like this:

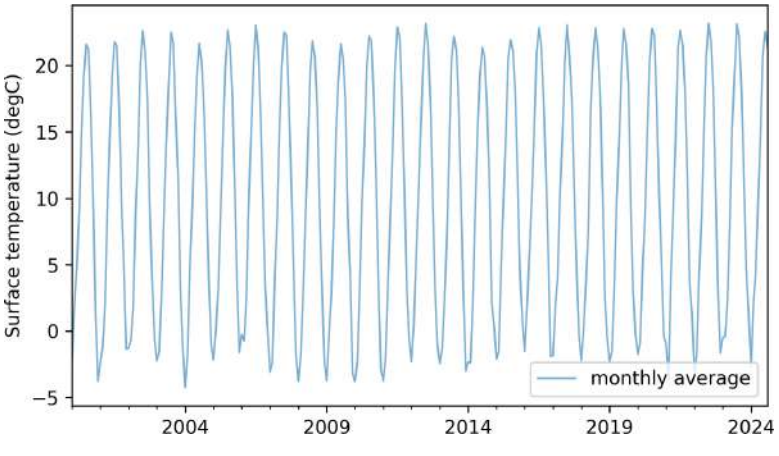
```
temp = pd.read_csv("monthly-average-surface-temperatures-by-year.csv")
```

The following cell selects data for the United States from 2001 to the end of the series and packs it into a Pandas `Series`:

```
temp_us = temp.query("Code == 'USA'")
columns = [str(year) for year in range(2000, 2025)]
temp_series = temp_us.loc[:, columns].transpose().stack()
temp_series.index = pd.date_range(start="2000-01", periods=len(temp_series),
                                  freq="ME")
```

Here's what it looks like:

```
temp_series.plot(label="monthly average", **actual_options)
decorate(ylabel="Surface temperature (degC)")
```

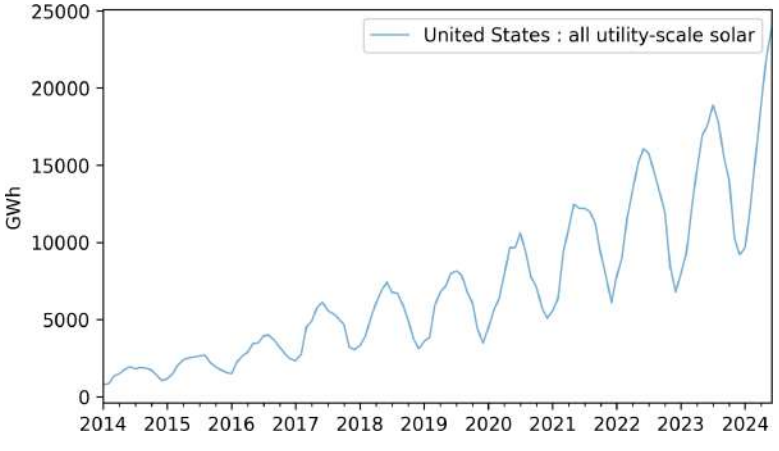


Not surprisingly, there is a strong seasonal pattern. Compute an additive seasonal decomposition with a period of 12 months. Fit a linear model to the trend line. What is the average annual increase in surface temperature during this interval? If you are curious, repeat this analysis with other intervals or data from other countries.

Exercise 12.2

Earlier in this chapter we used a multiplicative seasonal decomposition to model electricity production from small-scale solar power from 2014 to 2019 and forecast production from 2019 to 2024. Now let's do the same with utility-scale solar power. Here's what the time series looks like:

```
util_solar = elec["United States : all utility-scale solar"].dropna()
util_solar = util_solar[util_solar.index.year >= 2014]
util_solar.plot(**actual_options)
decorate(ylabel="GWh")
```

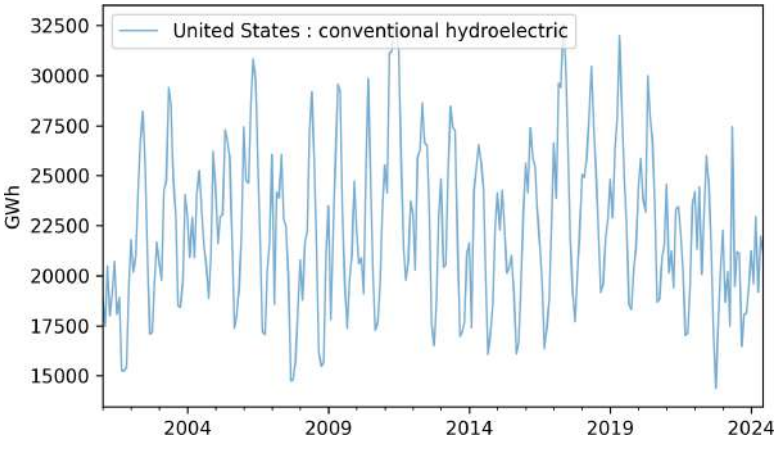


Use `split_series` to split this data into a training and test series. Compute a multiplicative decomposition of the training series with a 12-month period. Fit a linear or quadratic model to the trend and generate a five-year forecast, including a seasonal component. Plot the forecast along with the test series, and compute the mean absolute percentage error (MAPE).

Exercise 12.3

Let's see how well an ARIMA model fits production from hydroelectric generators in the United States. Here's what the time series looks like from 2001 to 2024:

```
hydro = elec["United States : conventional hydroelectric"]
hydro.plot(**actual_options)
decorate(ylabel="GWh")
```



Fit a SARIMA model to this data with a seasonal period of 12 months. Experiment with different lags in the autoregression and moving average parts of the model and see if you can find a combination that maximizes the R^2 value of the model. Generate a five-year forecast and plot it along with its confidence interval.

CHAPTER 13
Survival Analysis

Survival analysis is a way to describe how long things last. It is often used to study human lifetimes, but it also applies to "survival" of mechanical and electronic components, or more generally to an interval in time before any kind of event—or even an interval in space.

We'll start with a simple example, the lifespans of light bulbs, and then consider a more substantial example, age at first marriage and how it has changed in the United States over the last 50 years.

Survival Functions

A fundamental concept in survival analysis is the **survival function**, which is the fraction of a population that survives longer than a given duration. As a first example, we'll compute a survival function for the lifespans of light bulbs.

We'll use data from an experiment conducted in 2007. Researchers installed 50 new light bulbs and left them on continuously. They checked on the bulbs every 12 hours and recorded the lifespan of any that expired—running the experiment until all 50 bulbs expired. Instructions for downloading the data are in the notebook for this chapter.

We can read the data like this:

```
df = pd.read_csv("lamps.csv", index_col=0)
df.tail()
```

	h	f	K	i
28	1812	1	4
29	1836	1	3
30	1860	1	2
31	1980	1	1
32	2568	1	0

The `h` column contains lifespans in hours. The `f` column records the number of bulbs that expired at each value of `h`. To represent the distribution of lifespans, we'll put these values in a `Pmf` object and normalize it:

```
from empiricaldist import Pmf

pmf_bulblife = Pmf(df["f"].values, index=df["h"])
pmf_bulblife.normalize()
```

```
50
```

We can use `make_cdf` to compute the CDF, which indicates the fraction of bulbs that expire at or before each value of `h`. For example, 78% of the bulbs expire at or before 1,656 hours:

```
cdf_bulblife = pmf_bulblife.make_cdf()
cdf_bulblife[1656]
```

```
0.7800000000000002
```

The survival function is the fraction of bulbs that expire *after* each value of `h`, which is the complement of the CDF. So we can compute it like this:

```
complementary_cdf = 1 - cdf_bulblife
complementary_cdf[1656]
```

```
0.21999999999999975
```

Twenty-two percent of the bulbs expire after 1,656 hours.

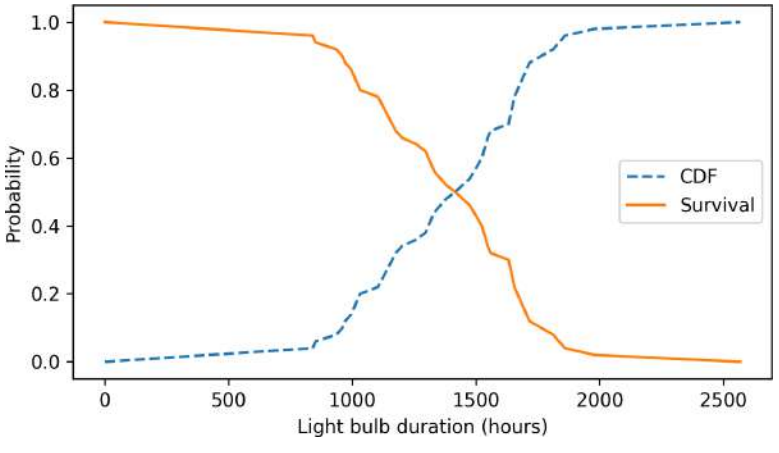
The `empiricaldist` library provides a `Surv` object that represents a survival function, and a method called `make_surv` that makes one:

```
surv_bulblife = cdf_bulblife.make_surv()
surv_bulblife[1656]
```

```
0.21999999999999997
```

If we plot the CDF and the survival function, we can see that they are complementary—that is, their sum is 1 at all values of h:

```
cdf_bulblife.plot(ls="--", label="CDF")
surv_bulblife.plot(label="Survival")

decorate(xlabel="Light bulb duration (hours)", ylabel="Probability")
```



In that sense, the CDF and survival function are equivalent—if we are given either one, we can compute the other—but in the context of survival analysis it is more common to work with survival curves. And computing a survival curve is a step toward the next important concept, the hazard function.

Hazard Function

In the light bulb dataset, each value of h represents a 12-hour interval ending at hour h—which I will call "interval h." Suppose we know that a light bulb has survived up to interval h, and we would like to know the probability that it expires during interval h. To answer this question, we can use the survival function, which indicates the fraction of bulbs that survive past interval h, and the PMF, which indicates the fraction that expire during interval h. The sum of these is the fraction of bulbs that *could* expire during interval h, which are said to be "at risk." As an example, 26% of the bulbs were at risk during interval 1656:

```
at_risk = pmf_bulblife + surv_bulblife
at_risk[1656]
```

```
0.25999999999999995
```

And 4% of all bulbs expired during interval 1656:

```
pmf_bulblife[1656]
```

```
0.04
```

The **hazard** is the ratio of `pmf_bulblife` and `at_risk`:

```
hazard = pmf_bulblife / at_risk
hazard[1656]
```

```
0.15384615384615388
```

Of all bulbs that survived up to interval 1656, about 15% expired during interval 1656.

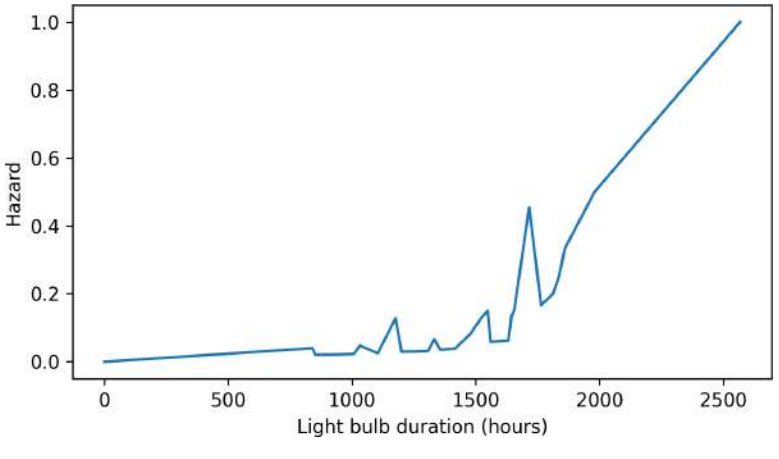
Instead of computing the hazard function ourselves, we can use `empiricaldist`, which provides a `Hazard` object that represents a hazard function, and a `make_hazard` method that computes it:

```
hazard_bulblife = surv_bulblife.make_hazard()
hazard_bulblife[1656]
```

```
0.15384615384615397
```

Here's what the hazard function looks like for the light bulbs:

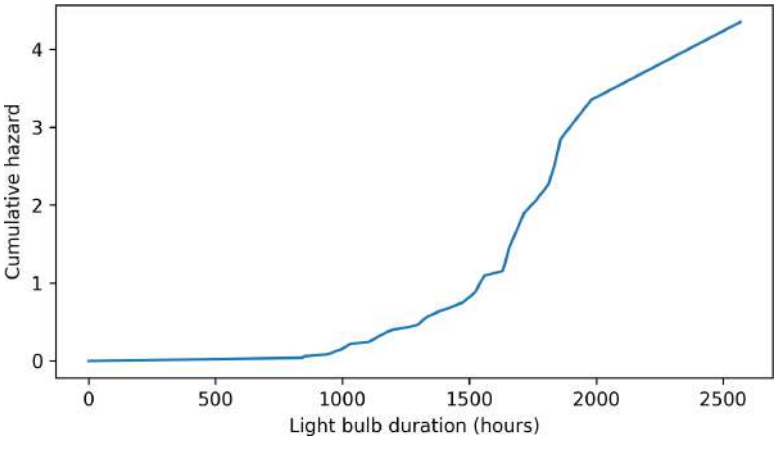
```
hazard_bulblife.plot()
decorate(xlabel="Light bulb duration (hours)", ylabel="Hazard")
```



We can see that the hazard is higher in some places than others, but this way of visualizing the hazard function can be misleading, especially in parts of the range where

we don't have much data. A better alternative is to plot the **cumulative hazard function**, which is the cumulative sum of the hazards:

```
cumulative_hazard = hazard_bulblife.cumsum()
cumulative_hazard.plot()

decorate(xlabel="Light bulb duration (hours)", ylabel="Cumulative hazard")
```



Where the probability of expiring is high, the cumulative hazard function is steep. Where the probability of expiring is low, the cumulative hazard function is flat. In this example, we can see that the hazard is highest between 1,500 and 2,000 hours. After that, the hazard decreases—although this outcome is based on just one unusually long-lived bulb, so it might look different in another dataset.

Now that we have the general idea of survival and hazard functions, let's apply them to a more substantial dataset.

Marriage Data

In many countries, people are getting married later than they used to, and more people stay unmarried. To explore these trends in the United States, we'll use the tools of survival analysis and data from the National Survey of Family Growth (NSFG).

The NSFG dataset we used in previous chapters is the pregnancy file, which contains one row for each pregnancy reported by the survey respondents. In this chapter, we'll work with the respondent file, which contains information about the respondents themselves.

I have compiled responses from nine iterations of the survey, conducted between 1982 and 2019, and selected data related to marriage. Instructions for downloading this excerpt are in the notebook for this chapter.

We can read the data like this:

```
resp = pd.read_csv("marriage_nsfg_female.csv.gz")
resp.shape
```

```
(70183, 34)
```

The excerpt includes one row for each of more than 70,000 respondents, and has the following variables related to age and marriage:

`cmbirth`
: The respondent's date of birth, known for all respondents

`cmintvw`
: The date the respondent was interviewed, known for all respondents

`cmmarrhx`
: The date the respondent was first married, if applicable and known

`evrmarry`
: 1 if the respondent had been married prior to the date of interview, 0 otherwise

The first three variables are encoded in "century-months"—that is, the integer number of months since December 1899. So century-month 1 is January 1900.

To explore generational changes, we'll group respondents by their decade of birth. We'll use the following function, which takes a value of `cmbirth` and computes the corresponding decade of birth. It uses the integer division operator `//` to divide by 10 and round down:

```
month0 = pd.to_datetime("1899-12-31")


def decade_of_birth(cmbirth):
    date = month0 + pd.DateOffset(months=cmbirth)
    return date.year // 10 * 10
```

We can use this function and the `apply` method to compute each respondent's decade of birth and assign it to a new column called `cohort`. In this context, a **cohort** is a group of people with something in common—like the decade they were born—who are treated as a group for purposes of analysis.

The result from `value_counts` shows the number of people in each cohort:

```
from thinkstats import value_counts

resp["cohort"] = resp["cmbirth"].apply(decade_of_birth)
value_counts(resp["cohort"])
```

```
cohort
1930      325
1940     3608
1950    10631
1960    14953
1970    16438
1980    14271
1990     8552
2000     1405
Name: count, dtype: int64
```

The dataset includes more than 10,000 respondents born in each decade from the 1950s to the 1980s, and fewer respondents in the earlier and later decades.

Next we'll compute each respondent's age when married (if applicable) and their age when interviewed:

```
resp["agemarr"] = (resp["cmmarrhx"] - resp["cmbirth"]) / 12
resp["age"] = (resp["cmintvw"] - resp["cmbirth"]) / 12
```

To get started with this data, we'll use the following function, which takes as arguments a `DataFrame` and a list of cohorts, and returns a dictionary that maps from each cohort to a `Surv` object. For each cohort, it selects their ages at first marriage and uses `Surv.from_seq` to compute a survival function. The `dropna=False` argument includes `NaN` values in the survival function, so the result includes people who have not married:

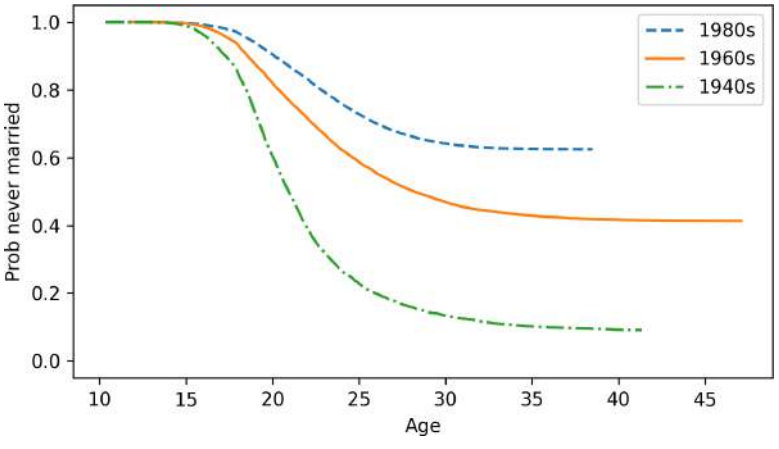
```
from empiricaldist import Surv


def make_survival_map(resp, cohorts):
    surv_map = {}

    grouped = resp.groupby("cohort")
    for cohort in cohorts:
        group = grouped.get_group(cohort)
        surv_map[cohort] = Surv.from_seq(group["agemarr"], dropna=False)

    return surv_map
```

Here's how we use this function:

```
cohorts = [1980, 1960, 1940]
surv_map = make_survival_map(resp, cohorts)
```

And here are the results for people born in the 1940s, 1960s, and 1980s:

```
for cohort, surv in surv_map.items():
    surv.plot(label=f"{cohort}s")

ylim = [-0.05, 1.05]
decorate(xlabel="Age", ylabel="Prob never married", ylim=ylim)
```



If we take these results at face value, they show that people in earlier generations got married younger, and more of them got married eventually. However, we should not interpret these results yet, because they are not correct. There are two problems we have to address:

- As discussed in "The National Survey of Family Growth" on page 3, the NSFG uses stratified sampling, which means that it deliberately oversamples some groups.
- Also, this way of computing the survival function does not properly take into account people who are not married yet.

For the first problem, we'll use a kind of resampling called a **weighted bootstrap**. For the second problem, we'll use a method called Kaplan-Meier estimation. We'll start with resampling.

Weighted Bootstrap

The NSFG dataset includes a column called `finalwgt` that contains each respondent's sampling weight, which is the number of people in the population they represent. We can use these weights during the resampling process to correct for stratified sampling. The following function takes a `DataFrame` and the name of the column that contains the sampling weights. It resamples the rows of the `DataFrame`, taking the sampling weights into account, and returns a new `DataFrame`:

```
def resample_rows_weighted(df, column="finalwgt"):
    n = len(df)
    weights = df[column]
    return df.sample(n, weights=weights, replace=True)
```

The current dataset includes respondents from several iterations of the survey, called cycles, so to resample, we have to group the respondents by cycle, resample each group, and then put the groups back together. That's what the following function does:

```
def resample_cycles(resp):
    grouped = resp.groupby("cycle")
    samples = [resample_rows_weighted(group) for _, group in grouped]
    return pd.concat(samples)
```

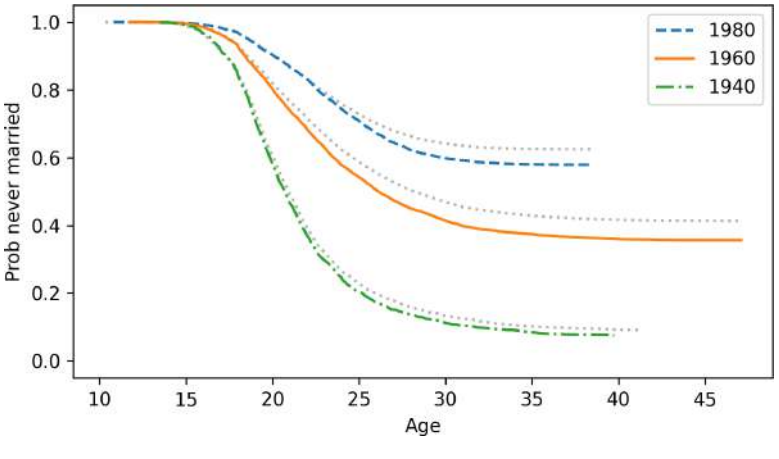
To get started, we'll resample the data once:

```
sample = resample_cycles(resp)
```

Later we'll resample the data several times, so we can see how much variation there is due to random sampling.

The following figure shows the results with resampling, compared to the results from the previous section without resampling, shown as dotted lines:

```
for label, surv in surv_map.items():
    surv.plot(ls=":", color="gray", alpha=0.6)

survs_resampled = make_survival_map(sample, cohorts)

for label, surv in survs_resampled.items():
    surv.plot(label=label)

decorate(xlabel="Age", ylabel="Prob never married", ylim=ylim)
```



The difference, with and without resampling, is substantial, which shows that we need to correct for stratified sampling to get accurate results.

Now let's get to the second problem, dealing with incomplete data.

Estimating Hazard Functions

In the light bulb example, we know the life spans for all 50 bulbs, so we can compute the survival function directly—and we can use the survival function to compute the hazard function.

In the marriage example, we know the age at first marriage for some respondents, the ones who had been married before they were interviewed. But for respondents who had never married, we don't know at what age they would marry in the future—or if they will.

This kind of missing data is said to be **censored**. That term might seem odd, because censored information is usually hidden deliberately, but in this case it is hidden just because we don't know the future.

However, we have partial information we can work with: if someone is unmarried when they are surveyed, we know that the age when they get married (if they do) must be greater than their current age.

We can use this partial information to estimate the hazard function; then we can use the hazard function to compute the survival function. This process is called **Kaplan-Meier estimation**.

To demonstrate, I'll select just one cohort from the resampled data:

```
resp60 = sample.query("cohort == 1960")
```

For respondents who were married when they were surveyed, we'll select their age at first marriage. There are 9,921 of them, which we'll call "complete" cases:

```
complete = resp60.query("evrmarry == 1")["agemarr"]
complete.count()
```

```
9921
```

For respondents who had not married, we'll select their age when they were surveyed. There are 5,468 of them, which we'll call the "ongoing" cases:

```
ongoing = resp60.query("evrmarry == 0")["age"]
ongoing.count()
```

```
5468
```

Now, to estimate the hazard function, we'll compute the total number of cases that were "at risk" at each age, including everyone who was unmarried up to that age. It will be convenient to make a `FreqTab` object that counts the number of complete and ongoing cases at each age:

```
from empiricaldist import FreqTab

ft_complete = FreqTab.from_seq(complete)
ft_ongoing = FreqTab.from_seq(ongoing)
```

As an example, there are 58 respondents who reported that they were married for the first time at age 25:

```
ft_complete[25]
```

```
58
```

And another 5 respondents who were surveyed at age 25 and reported that they had never married:

```
ft_ongoing[25]
```

```
5
```

From these `FreqTab` objects, we can compute unnormalized `Surv` objects that contain the number of complete and ongoing cases that exceed each age:

```
surv_complete = ft_complete.make_surv()
surv_ongoing = ft_ongoing.make_surv()
```

For example, there are 2,848 people who reported getting married after age 25:

```
surv_complete[25]
```

```
2848
```

And 2,273 people surveyed after age 25 who had never married:

```
surv_ongoing[25]
```

```
2273
```

The sum of the four numbers we just computed is the number of respondents who were at risk—that is, people who could have married at age 25. The term "at risk" is a legacy of survival analysis in medicine, where it often refers to risk of disease or death. It might seem incongruent in the context of marriage, which is generally considered a positive milestone. That said, here's how we compute it:

```
at_risk = ft_complete[25] + ft_ongoing[25] + surv_complete[25] + surv_ongoing[25]
at_risk
```

```
5184
```

Of those, the number who actually married at age 25 is `ft_complete[25]`. So we can compute the hazard function at age 25 like this:

```
hazard = ft_complete[25] / at_risk
hazard
```

```
0.011188271604938271
```

That's how we can compute the hazard function at a single age. Now let's compute the whole function, for all ages. We'll use the `union` method of the `Index` class to compute a Pandas `Index` that contains all of the ages from `ft_complete` and `ft_ongoing`, in order:

```
ts = pd.Index.union(ft_complete.index, ft_ongoing.index)
```

Now we can compute the number of people at risk at every age, by looking up the ages in `ts` in each of the `FreqTab` and `Surv` objects:

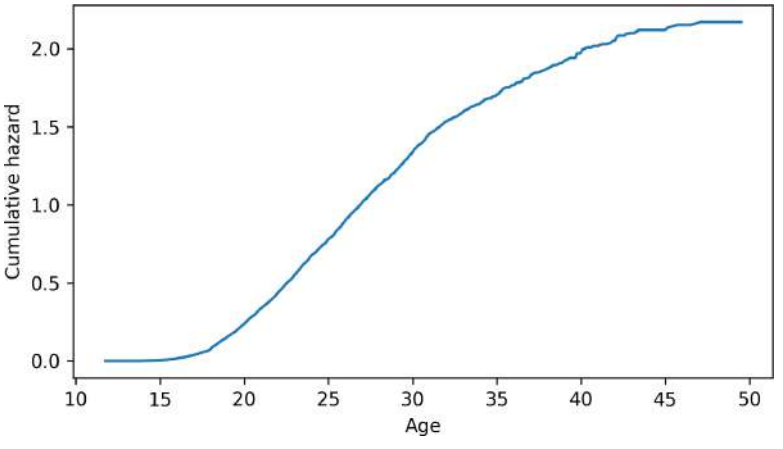
```
at_risk = ft_complete(ts) + ft_ongoing(ts) + surv_complete(ts) + surv_ongoing(ts)
```

Finally, we can compute the hazard function at each age, and put the results into a `Hazard` object:

```
from empiricaldist import Hazard

hs = ft_complete(ts) / at_risk
hazard = Hazard(hs, ts)
```

Here's what the cumulative hazard function looks like:

```
hazard.cumsum().plot()

decorate(xlabel="Age", ylabel="Cumulative hazard")
```



It is steepest between ages 20 and 30, which means that an unmarried person is at the greatest "risk" of getting married at these ages. After that, the cumulative hazard levels off, which means that the hazard gradually decreases.

Estimating Survival Functions

If we are given a survival function, we know how to compute the hazard function. Now let's go in the other direction.

Here's one way to think of it. The hazard function indicates the probability of getting married at each age, if you have not already married. So the complement of the hazard function is the probability of staying unmarried at each age.

To "survive" past a given age, `t`, you have to stay unmarried at every age up to and including `t`. And the probability of doing that is the product of the complementary hazard function, which we can compute like this:

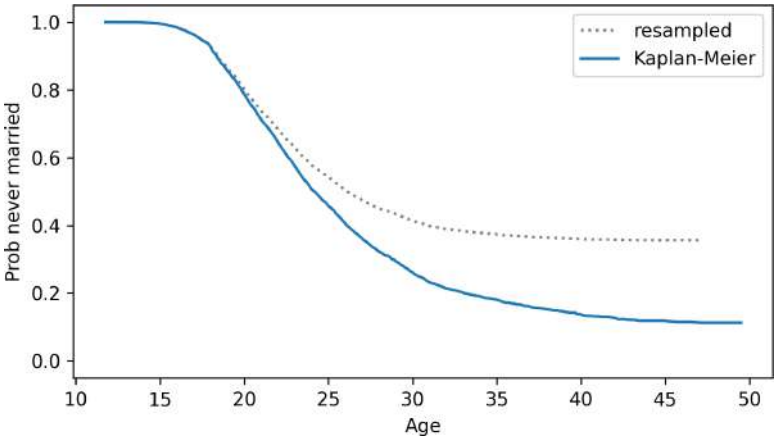
```
ps = (1 - hazard).cumprod()
```

The `Hazard` object has a `make_surv` method that does this computation:

```
surv = hazard.make_surv()
```

Here's what the result looks like, compared to the previous result (dotted line), which corrected for stratified resampling, but did not handle censored data:

```
survs_resampled[1960].plot(ls=":", color="gray", label="resampled")
surv.plot(label="Kaplan-Meier")

decorate(xlabel="Age", ylabel="Prob never married", ylim=ylim)
```



We can see how important it is to handle censored data correctly.

A survival function like this was the basis of a famous magazine article in 1986—*Newsweek* reported that a 40-year-old unmarried woman was "more likely to be killed by a terrorist" than get married. That claim was widely reported and became part of popular culture, but it was wrong then (because it was based on faulty analysis) and turned out to be even more wrong (because of cultural changes that were already in progress). In 2006, *Newsweek* ran another article acknowledging their error.

I encourage you to read more about this article, the statistics it was based on, and the reaction. It should remind you of the ethical obligation to perform statistical analysis with care, interpret the results with appropriate skepticism, and present them to the public accurately and honestly.

The following function encapsulates the steps of Kaplan-Meier estimation. It takes as arguments sequences of survival times for complete and ongoing cases, and returns a `Hazard` object:

```
def estimate_hazard(complete, ongoing):
    """Kaplan-Meier estimation."""
    ft_complete = FreqTab.from_seq(complete)
    ft_ongoing = FreqTab.from_seq(ongoing)

    surv_complete = ft_complete.make_surv()
    surv_ongoing = ft_ongoing.make_surv()

    ts = pd.Index.union(ft_complete.index, ft_ongoing.index)
    at_risk = (
        ft_complete(ts) + ft_ongoing(ts) +
        surv_complete(ts) + surv_ongoing(ts)
    )

    hs = ft_complete(ts) / at_risk
    return Hazard(hs, ts)
```

And here's a function that takes a group of respondents, extracts survival times, calls `estimate_hazard` to get the hazard function, and then computes the corresponding survival function:

```
def estimate_survival(group):
    """Estimate the survival function."""
    complete = group.query("evrmarry == 1")["agemarr"]
    ongoing = group.query("evrmarry == 0")["age"]
    hf = estimate_hazard(complete, ongoing)
    sf = hf.make_surv()
    return sf
```

Soon we'll use these functions to compute confidence intervals for survival functions. But first let's see another way to compute Kaplan-Meier estimates.

Lifelines

A Python package called `lifelines` provides tools for survival analysis, including functions that compute Kaplan-Meier estimates.

We can use it to confirm that the result in the previous section is correct. First we'll compute the survival function using `estimate_survival`:

```
surv = estimate_survival(resp60)
```

Next we'll compute it using `lifelines`. First we'll get the data into the format `lifelines` requires:

```
complete = complete.dropna()
durations = np.concatenate([complete, ongoing])
event_observed = np.concatenate([np.ones(len(complete)), np.zeros(len(ongoing))])
```

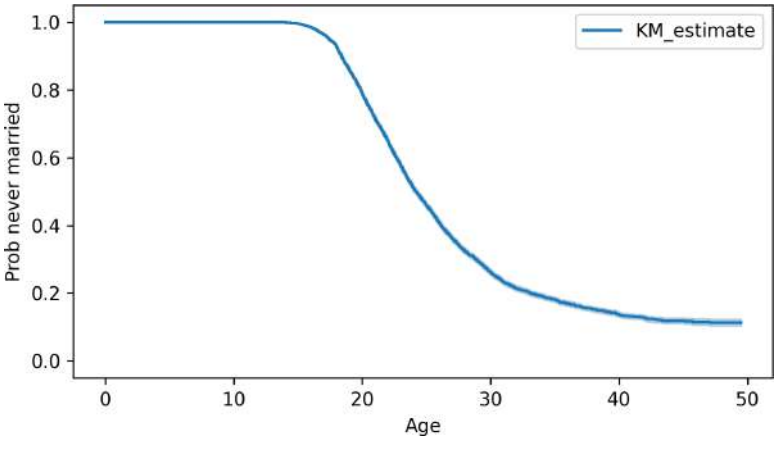
Now we can make a `KaplanMeierFitter` object and fit the data:

```
from lifelines import KaplanMeierFitter

kmf = KaplanMeierFitter()
kmf.fit(durations=durations, event_observed=event_observed)
```

```
<lifelines.KaplanMeierFitter:"KM_estimate", fitted with 15389 total observations,
5468 right-censored observations>
```

After fitting the data, we can call the `plot` function to display the results, which include the estimated survival function and a confidence interval—although the confidence interval is not correct in this case because it doesn't correct for stratified sampling:

```
kmf.plot()

decorate(xlabel="Age", ylabel="Prob never married", ylim=ylim)
```



Unlike the survival function we computed, the one from `lifelines` starts from 0. But the rest of the function is the same, within floating-point error:

```
ps = kmf.survival_function_["KM_estimate"].drop(0)
np.allclose(ps, surv)
```

```
True
```

In the next section, we'll use weighted resampling to compute confidence intervals that take stratified sampling into account.

Confidence Intervals

The Kaplan-Meier estimate we computed is based on a single resampling of the dataset. To get an idea of how much variation there is due to random sampling, we'll run the analysis with several resamplings and plot the results.

We'll use the following function, which takes a `DataFrame` and a list of cohorts, estimates the survival function for each cohort, and returns a dictionary that maps from each integer cohort to a `Surv` object.

This function is identical to `make_survival_map`, except that it calls `estimate_survival`, which uses Kaplan-Meier estimation, rather than `Surv.from_seq`, which only works if there is no censored data:

```
def estimate_survival_map(resp, cohorts):
    """Make a dictionary from cohorts to Surv objects."""
    surv_map = {}

    grouped = resp.groupby("cohort")
    for cohort in cohorts:
        group = grouped.get_group(cohort)
        surv_map[cohort] = estimate_survival(group)

    return surv_map
```

The following loop generates 101 random resamplings of the dataset and makes a list of 101 dictionaries containing the estimated survival functions:

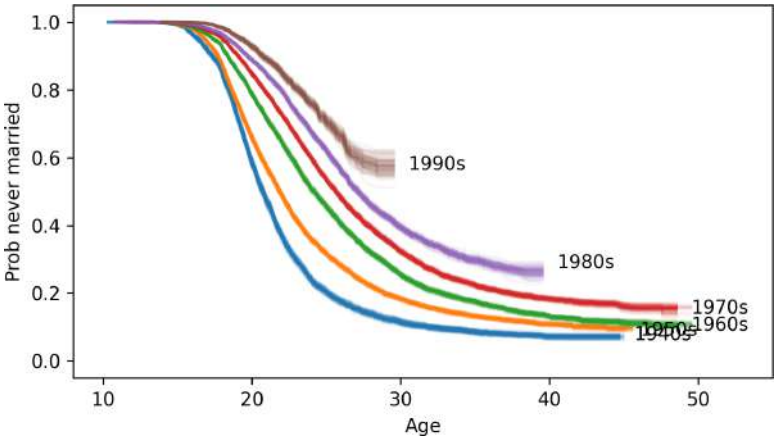
```
cohorts = [1940, 1950, 1960, 1970, 1980, 1990]

surv_maps = [estimate_survival_map(resample_cycles(resp), cohorts)
             for i in range(101)]
```

To plot the results, we'll use the following function, which takes that list of dictionaries, an integer cohort, and a color string. It loops through the dictionaries, selects the survival function for the given cohort, and plots it with a nearly transparent line—which is one way to visualize the variability between resamplings:

```
def plot_cohort(surv_maps, cohort, color):
    """Plot results for a single cohort."""
    survs = [surv_map[cohort] for surv_map in surv_maps]
    for surv in survs:
        surv.plot(color=color, alpha=0.05)

    x, y = surv.index[-1], surv.iloc[-1]
    plt.text(x + 1, y, f"{cohort}s", ha="left", va="center")
```

Here are the results for birth cohorts from the 1940s to the 1990s:

```
colors = [f"C{i}" for i in range(len(cohorts))]

for cohort, color in zip(cohorts, colors):
    plot_cohort(surv_maps, cohort, color)

xlim = [8, 55]
decorate(xlabel="Age", ylabel="Prob never married", xlim=xlim, ylim=ylim)
```



This visualization is good enough for exploration, but the lines look blurry and some of the labels overlap. More work might be needed to make a publication-ready figure, but we'll keep it simple for now.

Several patterns are visible:

- Women born in the 1940s married earliest—cohorts born in the 1950s and 1960s married later, but about the same fraction stayed unmarried.
- Women born in the 1970s married later *and* stayed unmarried at higher rates than previous cohorts.
- Cohorts born in the 1980s and 1990s are marrying even later, and are on track to stay unmarried at even higher rates—although these patterns could change in the future.

We'll have to wait for the next data release from the NSFG to learn more.

Expected Remaining Lifetime

Given a distribution, we can compute the expected remaining lifetime as a function of elapsed time. For example, given the distribution of pregnancy lengths, we can compute the expected time until delivery. To demonstrate, we'll use pregnancy data from the NSFG.

We'll use `get_nsfg_groups` to read the data and divide it into first babies and others:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

We'll start with a single resampling of the data:

```
sample = resample_rows_weighted(live, "finalwgt")
```

Here's the PMF of pregnancy durations:

```
pmf_durations = Pmf.from_seq(sample["prglngth"])
```

Now suppose it's the beginning of the 36th week of pregnancy. Remembering that the most common pregnancy length is 39 weeks, we expect the remaining time to be 3–4 weeks. To make that estimate more precise, we can identify the values in the distribution that equal or exceed 36 weeks:

```
t = 36
is_remaining = pmf_durations.qs >= t
```

Next we'll make a new `Pmf` object that contains only those values, shifted left so the current time is at 0:

```
ps = pmf_durations.ps[is_remaining]
qs = pmf_durations.qs[is_remaining] - t

pmf_remaining = Pmf(ps, qs)
```

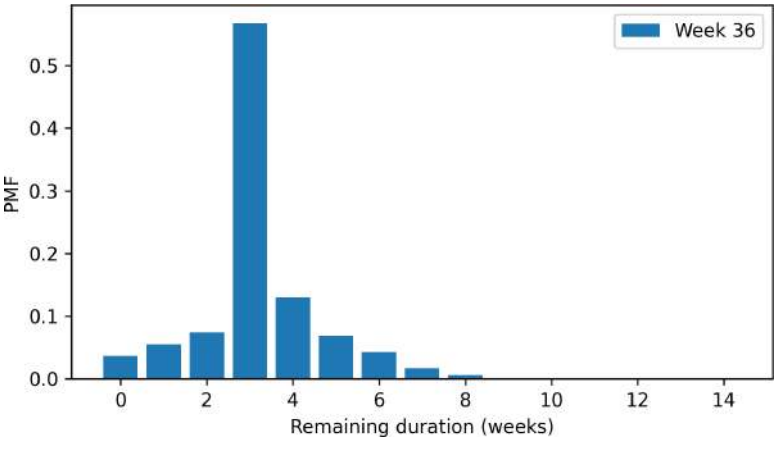
Because we selected a subset of the values in the `Pmf`, the probabilities no longer add up to 1, but we can normalize the `Pmf` so they do:

```
pmf_remaining.normalize()
```

```
0.9155006558810669
```

Here's the result, which shows the distribution of remaining time at the beginning of the 36th week:

```
pmf_remaining.bar(label="Week 36")
decorate(xlabel="Remaining duration (weeks)", ylabel="PMF")
```



The mean of this distribution is the expected remaining time:

```
pmf_remaining.mean()
```

```
3.2145671641791043
```

The following function encapsulates these steps and computes the distribution of remaining time for a given `Pmf` at a given time, `t`:

```
def compute_pmf_remaining(pmf, t):
    """Distribution of remaining time."""
    is_remaining = pmf.qs >= t
    ps = pmf.ps[is_remaining]
    qs = pmf.qs[is_remaining] - t
    pmf_remaining = Pmf(ps, qs)
    pmf_remaining.normalize()
    return pmf_remaining
```

The following function takes a `Pmf` of pregnancy lengths and computes the expected remaining time at the beginning of each week from the 36th to the 43rd:

```
def expected_remaining(pmf):
    index = range(36, 44)
    expected = pd.Series(index=index)

    for t in index:
        pmf_remaining = compute_pmf_remaining(pmf, t)
        expected[t] = pmf_remaining.mean()

    return expected
```

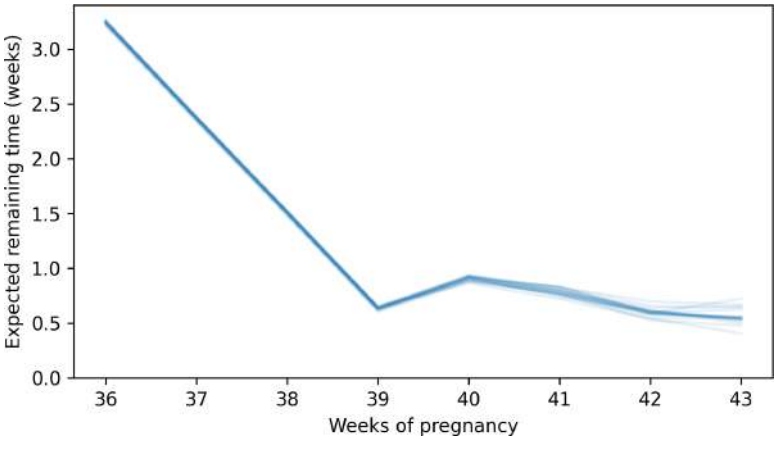
Here are the results for a single resampling of the data:

```
expected = expected_remaining(pmf_durations)
expected
```

```
36    3.214567
37    2.337714
38    1.479095
39    0.610133
40    0.912517
41    0.784211
42    0.582301
43    0.589372
dtype: float64
```

To see how much variation there is due to random sampling, we can run this analysis with several resamplings and plot the results:

```
for i in range(21):
    sample = resample_rows_weighted(live, "finalwgt")
    pmf_durations = Pmf.from_seq(sample["prglngth"])
    expected = expected_remaining(pmf_durations)
    expected.plot(color="C0", alpha=0.1)

decorate(
    xlabel="Weeks of pregnancy",
    ylabel="Expected remaining time (weeks)",
    ylim=[0, 3.4]
)
```



Between weeks 36 and 39, the expected remaining time decreases until, at the beginning of the 39th week, it is about 0.6 weeks. But after that, the curve levels off. At the beginning of the 40th week, the expected remaining time is still close to 0.6 weeks, and at the beginning of the 41st, 42nd, and 43rd, it is almost the same. For people waiting anxiously for a baby to be born, this behavior seems quite cruel.

Glossary

survival analysis
: A set of methods for describing and predicting the time until an event of interest, often focused on lifetimes or durations

survival function
: A function that maps from a time, *t*, to the probability of surviving beyond *t*

hazard function
: A function that maps from *t* to the fraction of cases that experience the event at *t*, out of all cases that survive until *t*

cumulative hazard function
: The cumulative sum of the hazard function, often useful for visualization

weighted bootstrap
: A form of resampling that uses sampling weights to correct for stratified sampling by simulating a representative sample

censored data
: Data that is only partially known because the event of interest has not yet occurred or was unobserved

Kaplan-Meier estimation
: A method for estimating survival and hazard functions in datasets with censored observations

cohort
: A group of subjects with a shared characteristic like a diagnosis or decade of birth

Exercises

Exercise 13.1

We can use the methods in this chapter to estimate hazard and survival functions for the duration of a marriage. To keep things simple, we'll consider only first marriages, and we'll focus on divorce as the endpoint, rather than separation or death.

In the NSFG data, the `cmdivorcx` column contains the date of divorce for each respondent's first marriage, if applicable, encoded in century-months. Compute the duration of marriages that have ended in divorce, and the duration, so far, of marriages that are ongoing:

- For complete cases, compute the elapsed time between `cmdivorcx` and `cmmarrhx`. If both values are valid—not `NaN`—that means the respondent's first marriage ended in divorce.
- To identify ongoing cases, select people who have only married once and who are still married. You can use `fmarno`, which records the number of times each respondent has married, and `fmarital`, which encodes their marital status—the value 1 indicates that the respondent is married.

In some cases the values of these variables are only approximate, so you might find a small number of negative differences, but they should not be more than one year.

Estimate the hazard and survival functions for the duration of marriage. Plot the cumulative hazard function—when is the danger of divorce highest? Plot the survival function—what fraction of marriages end in divorce?

Exercise 13.2

In 2012, a team of demographers at the University of Southern California estimated life expectancy for people born in Sweden in the early 1800s and 1900s. For ages from 0 to 91 years, they estimated the age-specific mortality rate, which is the fraction of people who die at a given age, out of all who survive until that age—which you might recognize as the hazard function.

I used an online graph digitizer to get the data from the figure in their paper and stored it in a CSV file. Instructions for downloading the data are in the notebook for this chapter.

We can load the data like this:

```
mortality = pd.read_csv("mortality_rates_beltran2012.csv", header=[0, 1]).dropna()
```

The following function interpolates the data to make a hazard function with approximate mortality rates for each age from 0 to 99:

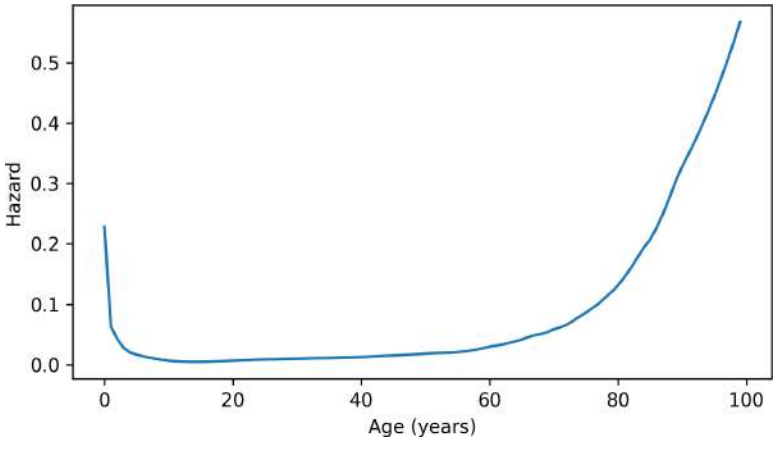
```
from scipy.interpolate import interp1d
from empiricaldist import Hazard

def make_hazard(ages, rates):
    interp = interp1d(ages, rates, fill_value="extrapolate")
    xs = np.arange(0, 100)
    ys = np.exp(interp(xs))
    return Hazard(ys, xs)
```

Now we can make a `Hazard` object like this:

```
ages = mortality["1800", "X"].values
rates = mortality["1800", "Y"].values
hazard = make_hazard(ages, rates)
```

Here's what the mortality rates look like:

```
hazard.plot()

decorate(xlabel="Age (years)", ylabel="Hazard")
```



Use `make_surv` to make a survival function based on these rates, and `make_cdf` to compute the corresponding CDF. Plot the results.

Then use `make_pmf` to make a `Pmf` object that represents the distribution of lifetimes, and plot it. Finally, use `compute_pmf_remaining` to compute the average remaining lifetime at each age from 0 to 99. Plot the result.

In the remaining lifetime curve, you should see a counterintuitive pattern—for the first few years of life, remaining lifetime increases. Because infant mortality was so high in the early 1800s, an older child was expected to live longer than a younger child. After about age 5, life expectancy returns to the pattern we expect—young people are expected to live longer than old people.

If you are interested in this topic, you might like Chapter 5 of my book *Probably Overthinking It*, which presents similarly counterintuitive results from many areas of statistics.

CHAPTER 14

Analytic Methods

This book has focused on computational methods like simulation and resampling, but some of the problems we solved have analytic solutions that can be much faster to compute.

This chapter presents some of these methods and explains how they work. At the end of the chapter, I make suggestions for integrating computational and analytic methods for data analysis.

Normal Probability Plots

Many analytic methods are based on the properties of the normal distribution, for two reasons: distributions of many measurements in the real world are well-approximated by normal distributions, and normal distributions have mathematical properties that make them useful for analysis.

To demonstrate the first point, we'll look at some of the measurements in the penguin dataset. Then we'll explore the mathematical properties of the normal distribution. Instructions for downloading the data are in the notebook for this chapter.

We can read the data like this:

```
penguins = pd.read_csv("penguins_raw.csv")
penguins.shape
```

```
(344, 17)
```

The dataset contains measurements from three penguin species. For this example, we'll select the Adélie penguins:

```
adelie = penguins.query('Species.str.startswith("Adelie")').copy()
len(adelie)
```

```
152
```

To see if penguin weights follow a normal distribution, we'll compute the empirical CDF of the data:

```
from empiricaldist import Cdf

weights = adelie["Body Mass (g)"].dropna()
cdf_weights = Cdf.from_seq(weights)
```

And we'll compute the analytic CDF of a normal distribution with the same mean and standard deviation:

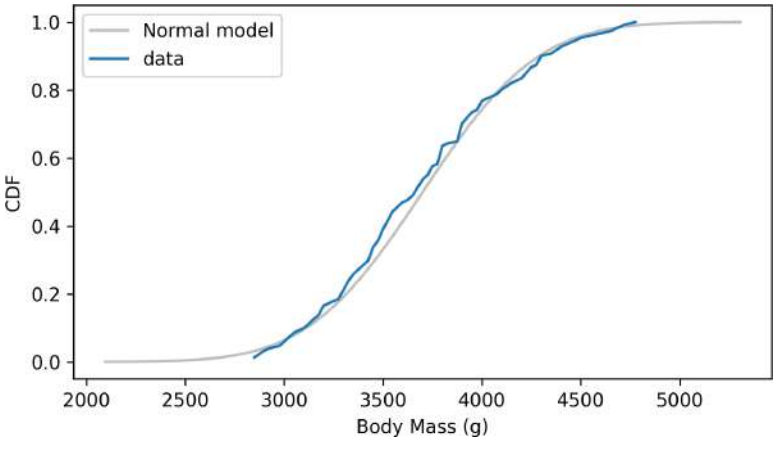
```
m, s = weights.mean(), weights.std()
m, s
```

```
(3700.662251655629, 458.5661259101348)
```

```
from scipy.stats import norm

dist = norm(m, s)
qs = np.linspace(m - 3.5 * s, m + 3.5 * s)
ps = dist.cdf(qs)
```

Here's what the CDF of the data looks like compared to the normal model:

```
model_options = dict(color="gray", alpha=0.5, label="model")
plt.plot(qs, ps, **model_options)
cdf_weights.plot(label="data")

decorate(ylabel="CDF")
```



The normal distribution might be a good enough model of this data, but it's certainly not a perfect fit.

In general, plotting the CDF of the data and the CDF of a model is a good way to evaluate how well the model fits the data. But one drawback of this method is that it depends on how well we estimate the parameters of the model—in this example, the mean and standard deviation.

An alternative is a **normal probability plot**, which does not depend on our ability to estimate parameters. In a normal probability plot the *y* values are the sorted measurements:

```
ys = np.sort(weights)
```

And the *x* values are the corresponding percentiles of a normal distribution, computed using the `ppf` method of the `norm` object, which computes the "percent point function," which is the inverse CDF:

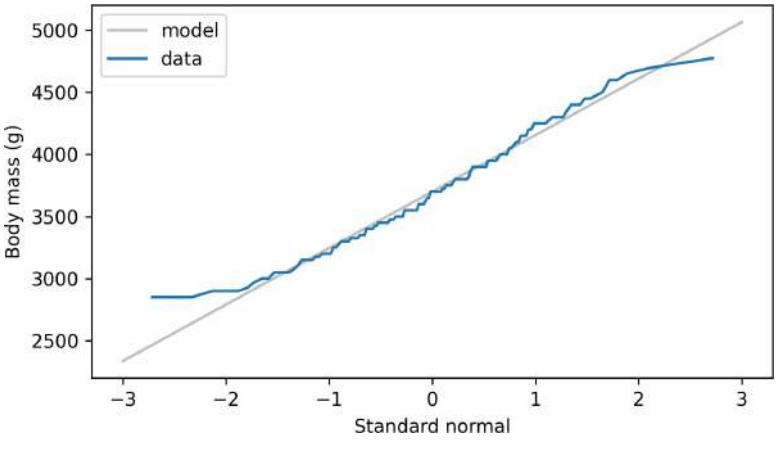
```
n = len(weights)
ps = (np.arange(n) + 0.5) / n
xs = norm.ppf(ps)
```

If the measurements are actually drawn from a normal distribution, the y and x values should fall on a straight line. To see how well they do, we can use `linregress` to fit a line:

```
from scipy.stats import linregress

results = linregress(xs, ys)
intercept, slope = results.intercept, results.slope

fit_xs = np.linspace(-3, 3)
fit_ys = intercept + slope * fit_xs
```

The following figure shows the x and y values along with the fitted line:

```
plt.plot(fit_xs, fit_ys, **model_options)
plt.plot(xs, ys, label="data")

decorate(xlabel="Standard normal", ylabel="Body mass (g)")
```



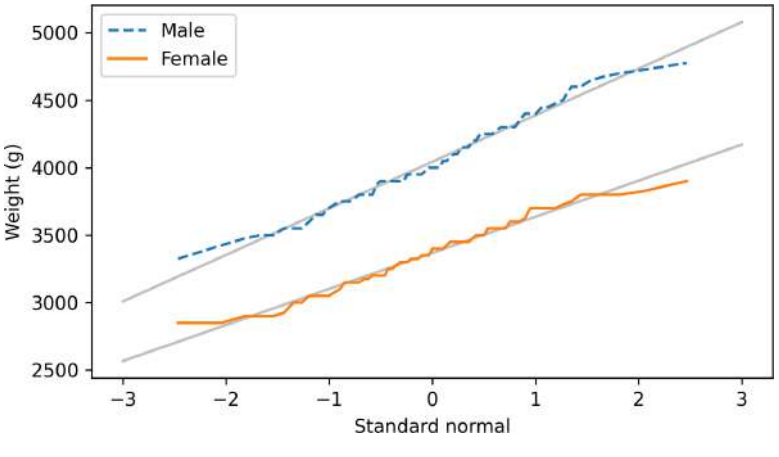
The normal probability plot is not a perfectly straight line, which indicates that the normal distribution is not a perfect model for this data.

One reason is that the dataset includes male and female penguins, and the two groups have different means—let's see what happens if we plot the groups separately. The following function encapsulates the steps we used to make a normal probability plot:

```
def normal_probability_plot(sample, **options):
    """Make a normal probability plot with a fitted line."""
    n = len(sample)
    ps = (np.arange(n) + 0.5) / n
    xs = norm.ppf(ps)
    ys = np.sort(sample)

    results = linregress(xs, ys)
    intercept, slope = results.intercept, results.slope

    fit_xs = np.linspace(-3, 3)
    fit_ys = intercept + slope * fit_xs

    plt.plot(fit_xs, fit_ys, color="gray", alpha=0.5)
    plt.plot(xs, ys, **options)
    decorate(xlabel="Standard normal")
```

Here's what the results look like for male and female penguins separately:

```
grouped = adelie.groupby("Sex")

weights_male = grouped.get_group("MALE")["Body Mass (g)"]
normal_probability_plot(weights_male, ls="--", label="Male")

weights_female = grouped.get_group("FEMALE")["Body Mass (g)"]
normal_probability_plot(weights_female, label="Female")

decorate(ylabel="Weight (g)")
```



The normal probability plots for both groups are close to a straight line, which indicates that the distributions of weight follow normal distributions. When we put the groups together, the distribution of their weights is a mixture of two normal distributions with different means—and a mixture like that is not always well modeled by a normal distribution.

Now let's consider some of the mathematical properties of normal distributions that make them so useful for analysis.

Normal Distributions

The following class defines an object that represents a normal distribution. It contains as attributes the parameters `mu` and `sigma2`, which represent the mean and variance of the distribution. The name `sigma2` is a reminder that variance is the square of the standard deviation, which is usually denoted `sigma`:

```
class Normal:
    """Represents a Normal distribution"""

    def __init__(self, mu, sigma2):
        self.mu = mu
        self.sigma2 = sigma2

    def __repr__(self):
        return f"Normal({self.mu}, {self.sigma2})"

    __str__ = __repr__
```

As an example, we'll create a `Normal` object that represents a normal distribution with the same mean and variance as the weights of the male penguins:

```
m, s = weights_male.mean(), weights_male.std()
dist_male = Normal(m, s**2)
dist_male
```

```
Normal(4043.4931506849316, 120278.25342465754)
```

And another `Normal` object with the same mean and variance as the weights of the female penguins:

```
m, s = weights_female.mean(), weights_female.std()
dist_female = Normal(m, s**2)
dist_female
```

```
Normal(3368.8356164383563, 72565.63926940637)
```

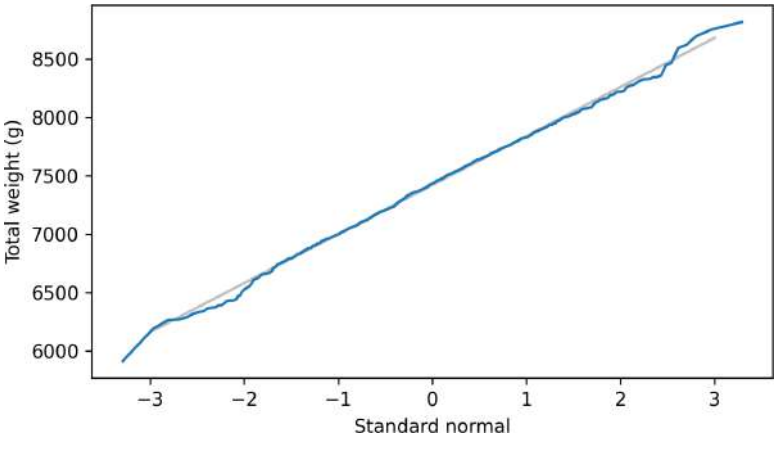
Next we'll add a method to the `Normal` class that generates a random sample from a normal distribution. To add methods to an existing class, we'll use a Jupyter magic command, `add_method_to`, which is defined in the `thinkstats` module. This command is not part of Python—it only works in Jupyter notebooks:

```
%%add_method_to Normal

def sample(self, n):
    sigma = np.sqrt(self.sigma2)
    return np.random.normal(self.mu, sigma, n)
```

We'll use `sample` to demonstrate the first useful property of a normal distribution: if you draw values from two normal distributions and add them, the distribution of the sum is also normal.

As an example, we'll generate samples from the `Normal` objects we just made, add them together, and make a normal probability plot of the sums:

```
sample_sum = dist_male.sample(1000) + dist_female.sample(1000)
normal_probability_plot(sample_sum)

decorate(ylabel="Total weight (g)")
```



The normal probability plot looks like a straight line, which indicates that the sums follow a normal distribution. And that's not all—if we know the parameters of the two distributions, we can compute the parameters of the distribution of the sum. The following method shows how:

```
%%add_method_to Normal

def __add__(self, other):
    """Distribution of the sum of two normal distributions."""
    return Normal(self.mu + other.mu, self.sigma2 + other.sigma2)
```

In the distribution of the sum, the mean is the sum of the means and the variance is the sum of the variances. Now that we've defined the special method `__add__`, we can use the + operator to "add" two distributions—that is, to compute the distribution of their sum:

```
dist_sum = dist_male + dist_female
dist_sum
```

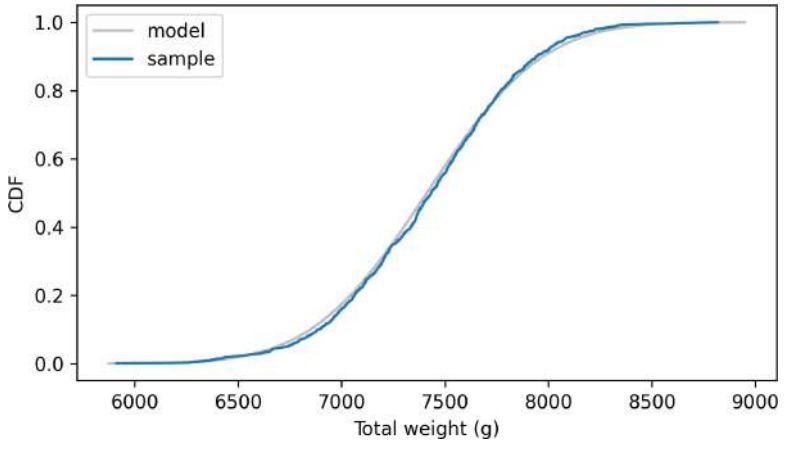
```
Normal(7412.328767123288, 192843.8926940639)
```

To confirm that this result is correct, we'll use the following method, which plots the analytic CDF of a normal distribution:

```
%%add_method_to Normal

def plot_cdf(self, n_sigmas=3.5, **options):
    mu, sigma = self.mu, np.sqrt(self.sigma2)
    low, high = mu - n_sigmas * sigma, mu + n_sigmas * sigma
    xs = np.linspace(low, high, 101)
    ys = norm.cdf(xs, mu, sigma)
    plt.plot(xs, ys, **options)
```

Here's the result along with the empirical CDF of the sum of the random samples:

```
dist_sum.plot_cdf(**model_options)
Cdf.from_seq(sample_sum).plot(label="sample")

decorate(xlabel="Total weight (g)", ylabel="CDF")
```



It looks like the parameters we computed are correct, which confirms that we can add two normal distributions by adding their means and variances.

As a corollary, if we generate `n` values from a normal distribution and add them up, the distribution of the sum is also a normal distribution. To demonstrate, we'll start by generating 73 values from the distribution of male weights and adding them up. The following loop does that 1,001 times, so the result is a sample from the distribution of sums:

```
n = len(weights_male)
sample_sums_male = [dist_male.sample(n).sum() for i in range(1001)]
n
```

```
73
```

The following method makes a `Normal` object that represents the distribution of the sums. To compute the parameters, we multiply both the mean and variance by `n`:

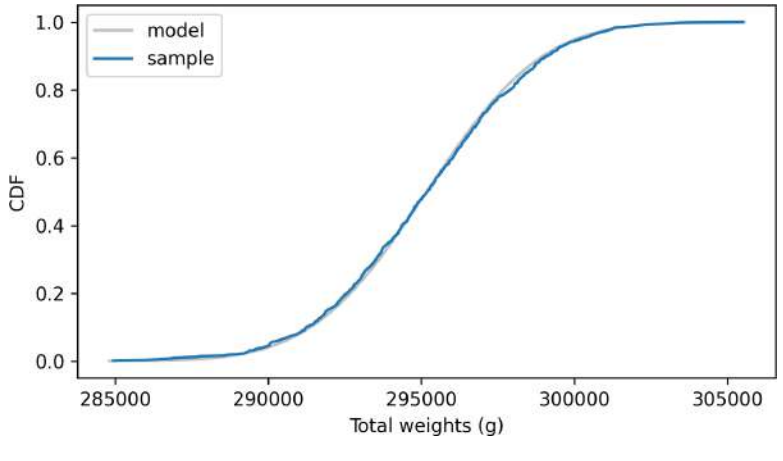
```
%%add_method_to Normal

def sum(self, n):
    """Distribution of the sum of n values."""
    return Normal(n * self.mu, n * self.sigma2)
```

Here's the distribution of the sum of `n` weights:

```
dist_sums_male = dist_male.sum(n)
```

And here's how it compares to the empirical distribution of the random sample:

```
dist_sums_male.plot_cdf(**model_options)
Cdf.from_seq(sample_sums_male).plot(label="sample")

decorate(xlabel="Total weights (g)", ylabel="CDF")
```



The analytic distribution fits the distribution of the sample, which confirms that the `sum` method is correct. So if we collect a sample of `n` measurements, we can compute the distribution of their sum.

Distribution of Sample Means

If we can compute the distribution of a sample sum, we can also compute the distribution of a sample mean. To do that, we'll use a third property of a normal distribution: if we multiply or divide by a constant, the result is a normal distribution. The following methods show how we compute the parameters of the distribution of a product or quotient:

```
%%add_method_to Normal


def __mul__(self, factor):
    """Multiply by a scalar."""
    return Normal(factor * self.mu, factor**2 * self.sigma2)
```

```
%%add_method_to Normal

def __truediv__(self, factor):
    """Divide by a scalar."""
    return self * (1/factor)
```

To compute the distribution of the product we multiply the mean by `factor` and the variance by the square of `factor`. We can use this property to compute the distribution of the sample means:

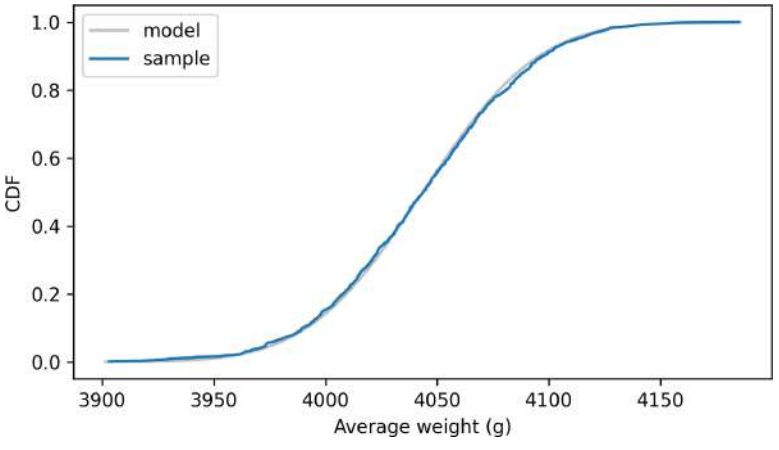
```
dist_mean_male = dist_sums_male / n
```

To see if the result is correct, we'll also compute the means of the random samples:

```
sample_means_male = np.array(sample_sums_male) / n
```

And compare the normal model to the empirical CDF of the sample means:

```
dist_mean_male.plot_cdf(**model_options)
Cdf.from_seq(sample_means_male).plot(label="sample")

decorate(xlabel="Average weight (g)", ylabel="CDF")
```



The model and the simulation results agree, which shows that we can compute the distribution of the sample means analytically—which is very fast, compared to resampling.

Now that we know the sampling distribution of the mean, we can use it to compute the standard error, which is the standard deviation of the sampling distribution:

```
standard_error = np.sqrt(dist_mean_male.sigma2)
standard_error
```

```
40.591222045992765
```

This result suggests a shortcut we can use to compute the standard error directly, without computing the sampling distribution. In the sequence of steps we followed, we multiplied the variance by `n` and then divided by `n**2`—the net effect was to divide the variance by `n`, which means we divided the standard deviation by the square root of `n`.

So we can compute the standard error of the sample mean like this:

```
standard_error = weights_male.std() / np.sqrt(n)
standard_error
```

```
40.59122204599277
```

Now let's consider one more result we can compute with normal distributions, the distribution of differences.

Distribution of Differences

Putting together the steps from the previous section, here's how we can compute the distribution of sample means for the weights of the female penguins:

```
n = len(weights_female)
dist_mean_female = dist_female.sum(n) / n
dist_mean_female
```

```
Normal(3368.835616438356, 994.0498530055667)
```

Now that we have sampling distributions for the average weight of male and female penguins—let's compute the distribution of the differences. The following method computes the distribution of the difference between values from two normal distributions:

```
%%add_method_to Normal

def __sub__(self, other):
    """The distribution of a difference."""
    return Normal(self.mu - other.mu, self.sigma2 + other.sigma2)
```

As you might expect, the mean of the differences is the difference of the means. But as you might not expect, the variance of the differences is not the difference of the variances—it's the sum! To see why, imagine we perform subtraction in two steps:

- If we negate the second distribution, the mean is negated but the variance is the same.
- Then if we add in the first distribution, the variance of the sum is the sum of the variances.

If that doesn't convince you, let's test it. Here's the analytic distribution of the differences:

```
dist_diff_means = dist_mean_male - dist_mean_female
dist_diff_means
```

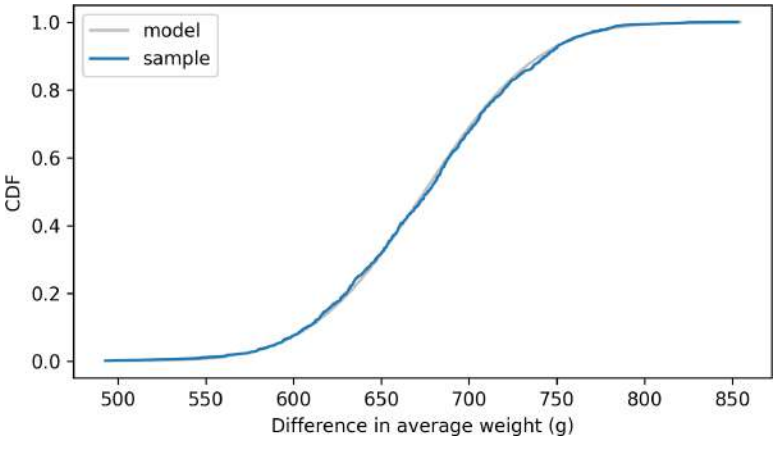
```
Normal(674.6575342465753, 2641.697160192656)
```

And here's a random sample of differences:

```
sample_sums_female = [dist_female.sample(n).sum() for i in range(1001)]
sample_means_female = np.array(sample_sums_female) / n
sample_diff_means = sample_means_male - sample_means_female
```

The following figure shows the empirical CDF of the random sample and the analytic CDF of the normal distribution:

```
dist_diff_means.plot_cdf(**model_options)
Cdf.from_seq(sample_diff_means).plot(label="sample")

decorate(xlabel="Difference in average weight (g)", ylabel="CDF")
```



They agree, which confirms that we found the distribution of the differences correctly. We can use this distribution to compute a confidence interval for the difference in weights. We'll use the following method to compute the inverse CDF:

```
%%add_method_to Normal

def ppf(self, xs):
    sigma = np.sqrt(self.sigma2)
    return norm.ppf(xs, self.mu, sigma)
```

The 5th and 95th percentiles form a 90% confidence interval:

```
ci90 = dist_diff_means.ppf([0.05, 0.95])
ci90
```

```
array([590.1162635, 759.19880499])
```

We get approximately the same results from the random sample:

```
np.percentile(sample_diff_means, [5, 95])
```

```
array([589.01470284, 760.1276391 ])
```

The analytic method is faster than resampling, and it is deterministic—that is, not random.

However, everything we've done so far is based on the assumption that the distribution of measurements is normal. That's not always true—in fact, with real data it is

never exactly true. But even if the distribution of the measurements isn't normal, if we add up many measurements, the distribution of their sum is often close to normal. That is the power of the Central Limit Theorem.

Central Limit Theorem

As we saw in the previous sections, if we add values drawn from normal distributions, the distribution of the sum is normal. Most other distributions don't have this property—for example, if we add values drawn from an exponential distribution, the distribution of the sum is not exponential.

But for many distributions, if we generate `n` values and add them up, the distribution of the sum converges to normal as `n` increases. More specifically, if the distribution of the values has mean `m` and variance `s2` the distribution of the sum converges to a normal distribution with mean `n * m` and variance `n * s2`.

That conclusion is the Central Limit Theorem (CLT). It is one of the most useful tools for statistical analysis, but it comes with caveats:

- The values have to come from the same distribution (although this requirement can be relaxed).
- The values have to be drawn independently. If they are correlated, the CLT doesn't apply (although it can still work if the correlation is not too strong).
- The values have to be drawn from a distribution with finite mean and variance. So the CLT doesn't apply to some long-tailed distributions.

The Central Limit Theorem explains the prevalence of normal distributions in the natural world. Many characteristics of living things are affected by genetic and environmental factors whose effect is additive. The characteristics we measure are the sum of a large number of small effects, so their distribution tends to be normal.

To see how the Central Limit Theorem works, and when it doesn't, let's try some experiments, starting with an exponential distribution. The following loop generates samples from an exponential distribution, adds them up, and makes a dictionary that maps from each sample size, `n`, to a list of 1,001 sums:

```
lam = 1
df_sample_expo = pd.DataFrame()
for n in [1, 10, 100]:
    df_sample_expo[n] = [np.sum(np.random.exponential(lam, n))
                         for _ in range(1001)]
```

Here are the averages for each list of sums:

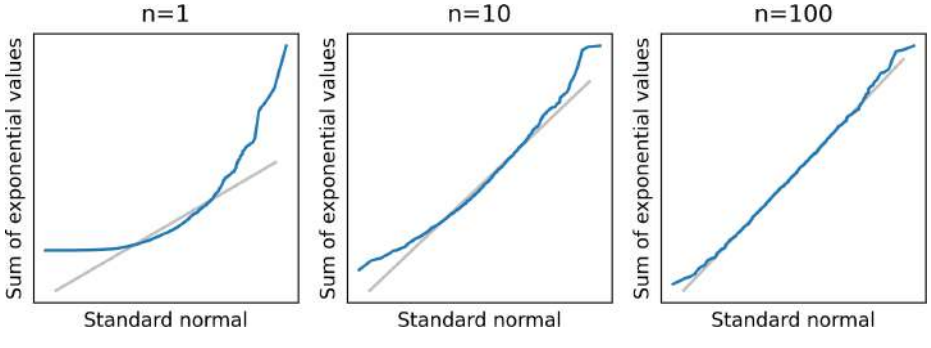
```
df_sample_expo.mean()
```

```
1        1.010255
10       9.993695
100    100.416396
dtype: float64
```

The average value from this distribution is 1, so if we add up 10 values, the average of the sum is close to 10, and if we add up 100 values the average of the sum is close to 100.

The following figure shows normal probability plots for the three lists of sums (the definition of `normal_plot_samples` is in the notebook for this chapter):

```
normal_plot_samples(df_sample_expo, ylabel="Sum of exponential values")
```



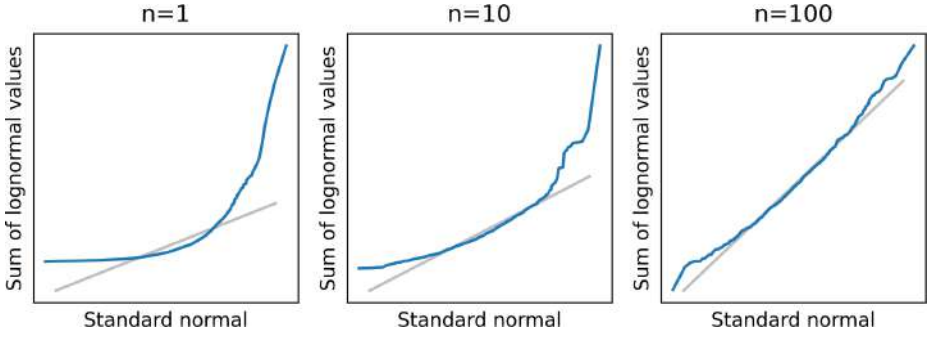
When `n=1`, the distribution of the sum is exponential, so the normal probability plot is not a straight line. But with `n=10` the distribution of the sum is approximately normal, and with `n=100` it is almost indistinguishable from normal.

For distributions that are less skewed than an exponential, the distribution of the sum converges to normal more quickly—that is, for smaller values of `n`. For distributions that are more skewed, it takes longer. As an example, let's look at sums of values from a lognormal distribution:

```
mu, sigma = 3.0, 1.0
df_sample_lognormal = pd.DataFrame()
for n in [1, 10, 100]:
    df_sample_lognormal[n] = [
        np.sum(np.random.lognormal(mu, sigma, n)) for _ in range(1001)
    ]
```

Here are the normal probability plots for the same range of sample sizes:

```
normal_plot_samples(df_sample_lognormal, ylabel="Sum of lognormal values")
```



When `n=1`, a normal model does not fit the distribution, and it is not much better with `n=10`. Even with `n=100`, the tails of the distribution clearly deviate from the model.

The mean and variance of the lognormal distribution are finite, so the distribution of the sum converges to normal eventually. But for some highly skewed distributions, it might not converge at any practical sample size. And in some cases, it doesn't happen at all.

The Limits of the Central Limit Theorem

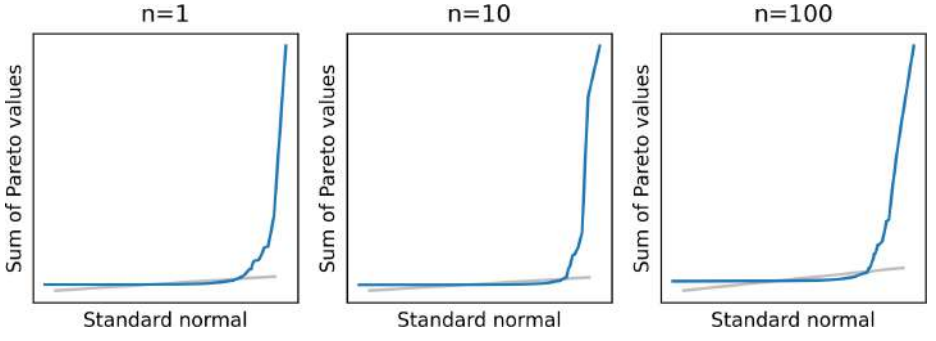
Pareto distributions are even more skewed than lognormal. Depending on the parameters, some Pareto distributions do not have finite mean and variance—in those cases, the Central Limit Theorem does not apply.

To demonstrate, we'll generate values from a Pareto distribution with parameter `alpha=1`, which has infinite mean and variance:

```
alpha = 1.0
df_sample = pd.DataFrame()
for n in [1, 10, 100]:
    df_sample[n] = [np.sum(np.random.pareto(alpha, n)) for _ in range(1001)]
```

Here's what the normal probability plots look like for a range of sample sizes:

```
normal_plot_samples(df_sample, ylabel="Sum of Pareto values")
```



Even with `n=100`, the distribution of the sum is nothing like a normal distribution.

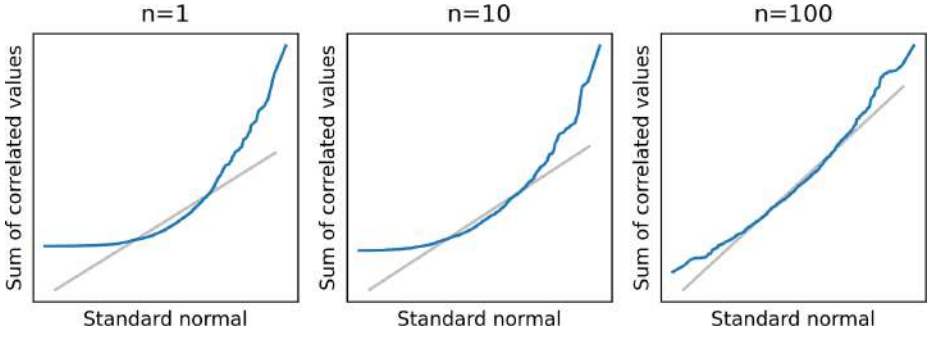
I also mentioned that the CLT does not apply if the values are correlated. To test that, we'll use a function called `generate_expo_correlated` to generate values from an exponential distribution where the serial correlation—that is, the correlation between successive elements in the sample—is the given value, `rho`. This function is defined in the notebook for this chapter.

The following loop makes a `DataFrame` with one column for each sample size and 1,001 sums in each column:

```
rho = 0.8
df_sample = pd.DataFrame()
for n in [1, 10, 100]:
    df_sample[n] = [np.sum(generate_expo_correlated(n, rho)) for _ in range(1001)]
```

Here are the normal probability plots for the distribution of these sums:

```
normal_plot_samples(df_sample, ylabel="Sum of correlated values")
```



With `rho=0.8`, there is a strong correlation between successive elements, and the distribution of the sum converges slowly. If there is also a strong correlation between distant elements of the sequence, it might not converge at all.

The previous section shows that the Central Limit Theorem works, and this section shows what happens when it doesn't. Now let's see how we can use it.

Applying the CLT

To see why the Central Limit Theorem is useful, let's get back to the example in "Testing a Difference in Means" on page 152: testing the apparent difference in mean pregnancy length for first babies and others. We'll use the NSFG data again—instructions for downloading it are in the notebook for this chapter.

We'll use `get_nsfg_groups` to read the data and divide it into first babies and others:

```
from nsfg import get_nsfg_groups

live, firsts, others = get_nsfg_groups()
```

As we've seen, first babies are born a little later, on average—the apparent difference is about 0.078 weeks:

```
delta = firsts["prglngth"].mean() - others["prglngth"].mean()
delta
```

```
0.07803726677754952
```

To see whether this difference might have happened by chance, we'll assume as a null hypothesis that the mean and variance of pregnancy lengths is actually the same for both groups, so we can estimate it using all live births:

```
all_lengths = live["prglngth"]
m, s2 = all_lengths.mean(), all_lengths.var()
```

The distribution of pregnancy lengths does not follow a normal distribution—nevertheless, we can use a normal distribution to approximate the sampling distribution of the mean.

The following function takes a sequence of values and returns a `Normal` object that represents the sampling distribution of the mean of a sample with the given size, `n`, drawn from a normal distribution with the same mean and variance as the data:

```
def sampling_dist_mean(data, n):
    mean, var = data.mean(), data.var()
    dist = Normal(mean, var)
    return dist.sum(n) / n
```

Here's a normal approximation to the sampling distribution of mean weight for first births, under the null hypothesis:

```
n1 = firsts["totalwgt_lb"].count()
dist_firsts = sampling_dist_mean(all_lengths, n1)
```

And here's the sampling distribution for other babies:

```
n2 = others["totalwgt_lb"].count()
dist_others = sampling_dist_mean(all_lengths, n2)
```

We can compute the sampling distribution of the difference like this:

```
dist_diff = dist_firsts - dist_others
dist_diff
```

```
Normal(0.0, 0.003235837567930557)
```

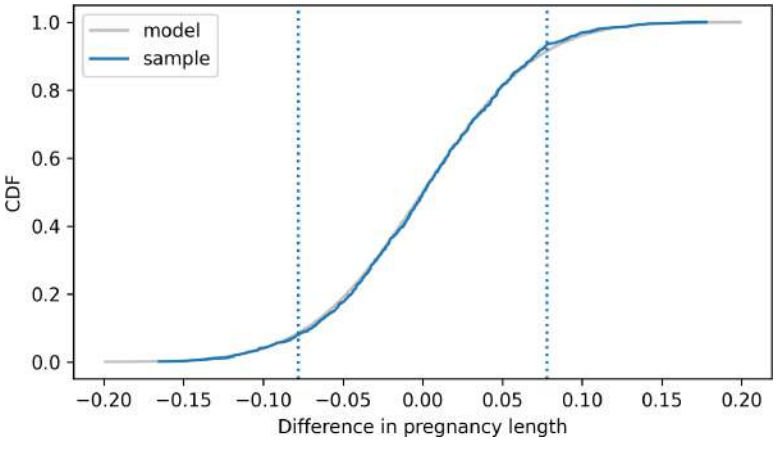
The mean is 0, which makes sense because if we draw two samples from the same distribution, we expect the difference in means to be 0, on average. The variance of the sampling distribution is 0.0032, which indicates how much variation we expect in the difference due to chance.

To confirm that this distribution approximates the sampling distribution, we can also estimate it by resampling:

```
sample_firsts = [np.random.choice(all_lengths, n1).mean() for i in range(1001)]
sample_others = [np.random.choice(all_lengths, n2).mean() for i in range(1001)]
sample_diffs = np.subtract(sample_firsts, sample_others)
```

Here's the empirical CDF of the resampled differences compared to the normal model. The vertical dotted lines show the observed difference, positive and negative:

```
dist_diff.plot_cdf(**model_options)
Cdf.from_seq(sample_diffs).plot(label="sample")
plt.axvline(delta, ls=":")
plt.axvline(-delta, ls=":")

decorate(xlabel="Difference in pregnancy length", ylabel="CDF")
```



In this example, the sample sizes are large and the skewness of the measurements is modest, so the sampling distribution is well approximated by a normal distribution. Therefore, we can use the normal CDF to compute a p-value. The following method computes the CDF of a normal distribution:

```
%%add_method_to Normal

def cdf(self, xs):
    sigma = np.sqrt(self.sigma2)
    return norm.cdf(xs, self.mu, sigma)
```

Here's the probability of a difference as large as `delta` under the null hypothesis, which is the area under the right tail of the sampling distribution:

```
right = 1 - dist_diff.cdf(delta)
right
```

```
0.08505405315526993
```

And here's the probability of a difference as negative as `-delta`, which is the area under the left tail:

```
left = dist_diff.cdf(-delta)
left
```

```
0.08505405315526993
```

`left` and `right` are the same because the normal distribution is symmetric. The sum of the two is the probability of a difference as large as `delta`, positive or negative:

```
left + right
```

```
0.17010810631053985
```

The resulting p-value is 0.170, which is consistent with the estimate we computed by resampling in "Testing a Difference in Means" on page 152.

The way we computed this p-value is similar to an **independent sample *t* test**. SciPy provides a function called `ttest_ind` that takes two samples and computes a p-value for the difference in their means:

```
from scipy.stats import ttest_ind

result = ttest_ind(firsts["prglngth"], others["prglngth"])
result.pvalue
```

```
0.16755412639415004
```

When the sample sizes are large, the result of the *t* test is close to what we computed with normal distributions. The *t* test is so called because it is based on a ***t* distribution** rather than a normal distribution. The *t* distribution is also useful for testing whether a correlation is statistically significant, as we'll see in the next section.

Correlation Test

In "Testing a Correlation" on page 156 we used a permutation test for the correlation between birth weight and mother's age, and found that it is statistically significant, with p-value less than 0.001.

Now we can do the same thing analytically. The method is based on this mathematical result: if we generate two samples with size `n` from normal distributions, compute Pearson's correlation, `r`, and then transform the correlation with this function:

```
def transform_correlation(r, n):
    return r * np.sqrt((n - 2) / (1 - r**2))
```

The transformed correlations follow a *t* distribution with parameter `n-2`. To see what that looks like, we'll use the following function to generate uncorrelated samples from a standard normal distribution:

```
def generate_data(n):
    """Uncorrelated sequences from a standard normal."""
    xs = np.random.normal(0, 1, n)
    ys = np.random.normal(0, 1, n)
    return xs, ys
```

And the following function to compute their correlation:

```
def correlation(data):
    xs, ys = data
    return np.corrcoef(xs, ys)[0, 1]
```

The following loop generates many pairs of samples, computes their correlation, and puts the results in a list:

```
n = 100
rs = [correlation(generate_data(n)) for i in range(1001)]
```

Next we'll compute the transformed correlations:

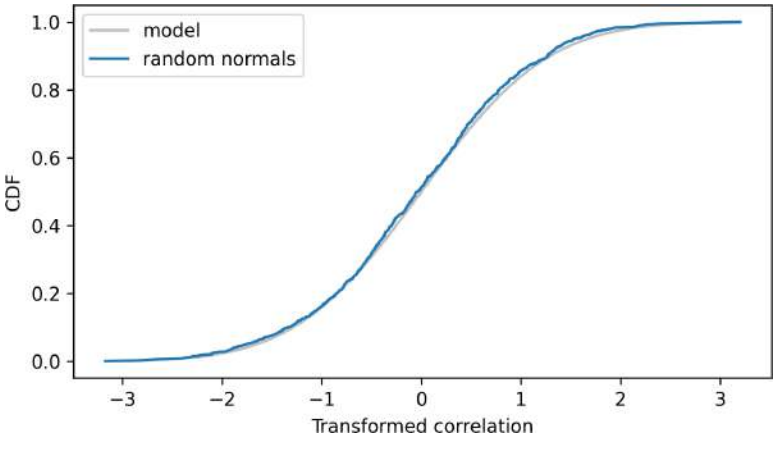
```
ts = transform_correlation(np.array(rs), n)
```

To check whether these `ts` follow a *t* distribution, we'll use the following function, which makes an object that represents the CDF of a *t* distribution:

```
from scipy.stats import t as student_t

def make_student_cdf(df):
    """Compute the CDF of a Student t distribution."""
    ts = np.linspace(-3, 3, 101)
    ps = student_t.cdf(ts, df=df)
    return Cdf(ps, ts)
```

The parameter of the *t* distribution is called `df`, which stands for "degrees of freedom." The following figure shows the CDF of a *t* distribution with parameter `n-2` along with the empirical CDF of the transformed correlations:

```
make_student_cdf(df=n - 2).plot(**model_options)

cdf_ts = Cdf.from_seq(ts)
cdf_ts.plot(label="random normals")

decorate(xlabel="Transformed correlation", ylabel="CDF")
```



This shows that if we draw uncorrelated samples from normal distributions, their transformed correlations follow a *t* distribution.

If we draw samples from other distributions, their transformed correlations don't follow a *t* distribution exactly, but they converge to a *t* distribution as the sample size increases. Let's see if this applies to the correlation of maternal age and birth weight. From the `DataFrame` of live births, we'll select the rows with valid data:

```
valid = live.dropna(subset=["agepreg", "totalwgt_lb"])
n = len(valid)
n
```

```
9038
```

The actual correlation is about 0.07:

```
data = valid["agepreg"].values, valid["totalwgt_lb"].values
r_actual = correlation(data)
r_actual
```

```
0.0688339703541091
```

As we did in "Testing a Correlation" on page 156, we can simulate the null hypothesis by permuting the samples:

```
def permute(data):
    """Shuffle the x values."""
    xs, ys = data
    new_xs = xs.copy()
    np.random.shuffle(new_xs)
    return new_xs, ys
```

If we generate many permutations and compute their correlations, the result is a sample from the distribution of correlations under the null hypothesis:

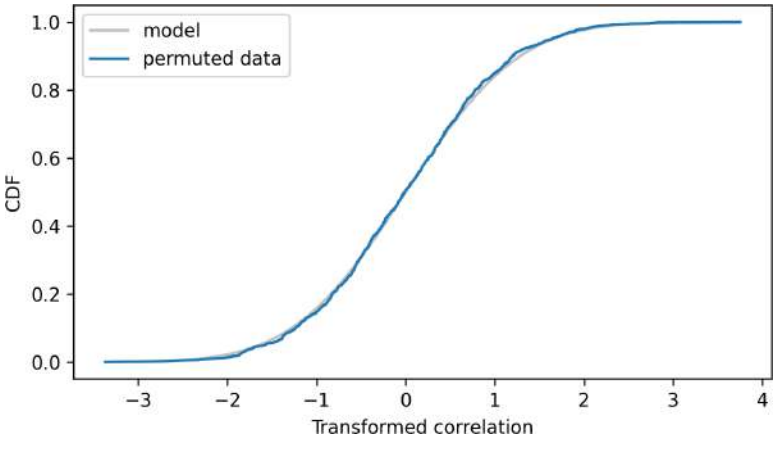
```
permuted_corrs = [correlation(permute(data)) for i in range(1001)]
```

And we can compute the transformed correlations like this:

```
ts = transform_correlation(np.array(permuted_corrs), n)
```

The following figure shows the empirical CDF of the `ts` along with the CDF of the *t* distribution with parameter `n-2`:

```
make_student_cdf(n - 2).plot(**model_options)
Cdf.from_seq(ts).plot(label="permuted data")

decorate(xlabel="Transformed correlation", ylabel="CDF")
```



The model fits the empirical distribution well, which means we can use it to compute a p-value for the observed correlation. First we'll transform the observed correlation:

```
t_actual = transform_correlation(r_actual, n)
```

Now we can use the CDF of the *t* distribution to compute the probability of a value as large as `t_actual` under the null hypothesis:

```
right = 1 - student_t.cdf(t_actual, df=n - 2)
right
```

```
2.861466619208386e-11
```

We can also compute the probability of a value as negative as `-t_actual`:

```
left = student_t.cdf(-t_actual, df=n - 2)
left
```

```
2.8614735536574016e-11
```

The sum of the two is the probability of a correlation as big as `r_actual`, positive or negative:

```
left + right
```

```
5.722940172865787e-11
```

SciPy provides a function that does the same calculation and returns the p-value of the observed correlation:

```
from scipy.stats import pearsonr

corr, p_value = pearsonr(*data)
p_value
```

```
5.722947107314431e-11
```

The results are nearly the same.

Based on the resampling results, we concluded that the p-value was less than 0.001, but we could not say how much less without running a very large number of resamplings. With analytic methods, we can compute small p-values quickly.

However, in practice it might not matter. Generally, if a p-value is smaller than 0.001, we can conclude that the observed effect is unlikely to be due to chance. It is not usually important to know precisely how unlikely.

Chi-squared Test

In "Testing Proportions" on page 158 we tested whether a die is crooked, based on this set of observed outcomes:

```
from empiricaldist import Hist

qs = np.arange(1, 7)
freqs = [8, 9, 19, 5, 8, 11]
observed = Hist(freqs, qs)
observed.index.name = "outcome"
observed
```

freqs	outcome
1	8
2	9
3	19
4	5
5	8
6	11

First we computed the expected frequency for each outcome:

```
num_rolls = observed.sum()
outcomes = observed.qs
expected = Hist(num_rolls / 6, outcomes)
```

Then we used the following function to compute the chi-squared statistic:

```
def chi_squared_stat(observed, expected):
    diffs = (observed - expected) ** 2
    ratios = diffs / expected
    return np.sum(ratios.values.flatten())
```

```
observed_chi2 = chi_squared_stat(observed, expected)
```

The **chi-squared statistic** is widely used for this kind of data because its sampling distribution under the null hypothesis converges to a distribution we can compute efficiently—not coincidentally, it is called a **chi-squared distribution**. To see what it looks like, we'll use the following function, which simulates rolling a fair die:

```
def simulate_dice(observed):
    n = np.sum(observed)
    rolls = np.random.choice(observed.qs, n, replace=True)
    hist = Hist.from_seq(rolls)
    return hist
```

The following loop runs the simulation many times and computes the chi-squared statistic of the outcomes:

```
simulated_chi_squared = [
    chi_squared_stat(simulate_dice(observed), expected) for i in range(1001)
]
cdf_simulated = Cdf.from_seq(simulated_chi_squared)
```

To check whether the results follow a chi-squared distribution, we'll use the following function, which computes the CDF of a chi-squared distribution with parameter `df`:

```
from scipy.stats import chi2 as chi2_dist

def chi_squared_cdf(df):
```

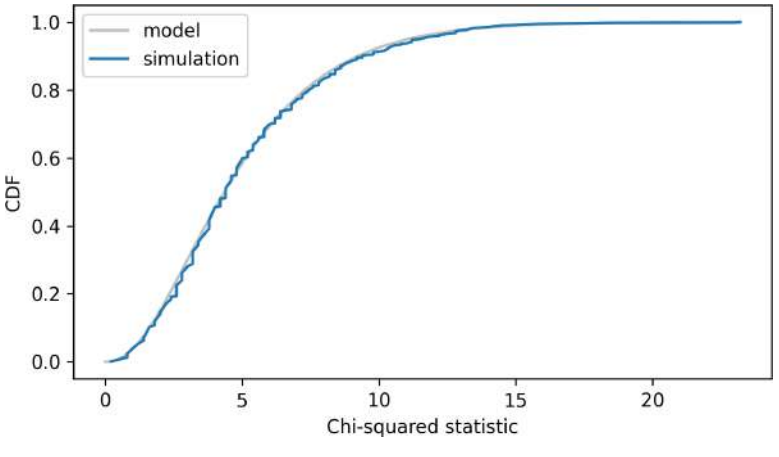
```
    """Discrete approximation of the chi-squared CDF."""
    xs = np.linspace(0, 21, 101)
    ps = chi2_dist.cdf(xs, df=df)
    return Cdf(ps, xs)
```

With `n` possible outcomes, the simulated chi-squared statistics should follow a chi-squared distribution with parameter `n-1`:

```
n = len(observed)
cdf_model = chi_squared_cdf(df=n - 1)
```

Here's the empirical CDF of the simulated chi-squared statistics along with the CDF of the chi-squared distribution:

```
cdf_model.plot(**model_options)
cdf_simulated.plot(label="simulation")

decorate(xlabel="Chi-squared statistic", ylabel="CDF")
```



The model fits the simulation results well, so we can use it to compute the probability of a value as large as `observed_chi2` under the null hypothesis:

```
p_value = 1 - chi2_dist.cdf(observed_chi2, df=n - 1)
p_value
```

```
0.04069938850404997
```

SciPy provides a function that does the same computation:

```
from scipy.stats import chisquare

chi2_stat, p_value = chisquare(f_obs=observed, f_exp=expected)
```

The result is nearly the same as the p-value we computed:

```
p_value
```

```
0.040699388504049985
```

The advantage of the chi-squared statistic is that its distribution under the null hypothesis can be computed efficiently. But in context, it might not be the statistic that best quantifies the difference between the observed and expected outcomes.

Computation and Analysis

This book focuses on computational methods like resampling and permutation. These methods have several advantages over analysis:

They are easier to explain and understand.
: For example, one of the most difficult topics in an introductory statistics class is hypothesis testing. Many students don't really understand what p-values are. I think the approach we took in Chapter 9—simulating the null hypothesis and computing test statistics—makes the fundamental idea clearer.

They are robust and versatile.
: Analytic methods are often based on assumptions that don't hold in practice. Computational methods require fewer assumptions, and can be adapted and extended more easily.

They are debuggable.
: Analytic methods are often like a black box: you plug in numbers and they spit out results. But it's easy to make subtle errors, hard to be confident that the results are right, and hard to diagnose the problem if they are not. Computational methods lend themselves to incremental development and testing, which fosters confidence in the results.

But there are drawbacks:

- Computational methods can be slow.
- Randomized methods like resampling don't produce the same results every time, which makes it harder to check that they are correct.

Taking into account these pros and cons, I recommend the following process:

1. Use computational methods during exploration. If you find a satisfactory answer and the run time is acceptable, you can stop.
2. If run time is not acceptable, look for opportunities to optimize. Using analytic methods is one of several methods of optimization.

3. If replacing a computational method with an analytic method is appropriate, use the computational method as a basis of comparison, providing mutual validation between the computational and analytic results.

For many practical problems, the run time of computational methods is not a problem, and we don't have to go past the first step.

Glossary

normal probability plot
: A plot that compares observed values with the quantiles of a normal distribution to see how closely the data follow a normal distribution

independent sample t test
: A method for computing the p-value of an observed difference between the means of two independent groups

t distribution
: A distribution used to model the sampling distribution of a difference in means under the null hypothesis that the difference is 0, as well as the sampling distribution of transformed correlations

chi-squared distribution
: A distribution used to model the sampling distribution of the chi-squared statistic

chi-squared statistic
: A test statistic that quantifies the magnitude of the difference between two discrete distributions

Exercises

Exercise 14.1

In this chapter we compared the weights of male and female penguins and computed a confidence interval for the difference. Now let's do the same for flipper length. The observed difference is about 4.6 mm:

```
grouped = adelie.groupby("Sex")

lengths_male = grouped.get_group("MALE")["Flipper Length (mm)"]
lengths_female = grouped.get_group("FEMALE")["Flipper Length (mm)"]
observed_diff = lengths_male.mean() - lengths_female.mean()
observed_diff
```

```
4.616438356164366
```

Use `sampling_dist_mean` to make `Normal` objects that represent sampling distributions for the mean flipper length in the two groups—noting that the groups are not the same size. Then compute the sampling distribution of the difference and a 90% confidence interval.

Exercise 14.2

Using the NSFG data, we computed the correlation between a baby's birth weight and the mother's age, and we used a *t* distribution to compute a p-value. Now let's do the same with birth weight and father's age, which is recorded in the `hpagelb` column:

```
valid = live.dropna(subset=["hpagelb", "totalwgt_lb"])
n = len(valid)
n
```

```
8933
```

The observed correlation is about 0.065:

```
data = valid["hpagelb"].values, valid["totalwgt_lb"].values
r_actual = correlation(data)
r_actual
```

```
0.06468629895432174
```

Compute the transformed correlation, `t_actual`. Use the CDF of the *t* distribution to compute a p-value—is this correlation statistically significant? Use the SciPy function `pearsonr` to check your results.

Exercise 14.3

In one of the exercises in Chapter 9 we considered the Trivers-Willard hypothesis, which suggests that for many mammals the sex ratio depends on "maternal condition"—that is, factors like the mother's age, size, health, and social status. Some studies have shown this effect among humans, but results are mixed.

As an example, and a chance to practice a chi-squared test, let's see if there's a relationship between the sex of a baby and the mother's marital status. The notebook for this chapter has instructions to help you get started.

Exercise 14.4

The method we used in this chapter to analyze differences between groups can be extended to analyze "differences in differences," which is a common experimental design. As an example, we'll use data from a 2014 paper that investigates the effects of an intervention intended to mitigate gender-stereotypical task allocation within student engineering teams.

Before and after the intervention, students responded to a survey that asked them to rate their contribution to each aspect of class projects on a seven-point scale.

Before the intervention, male students reported higher scores for the programming aspect of the projects than female students: men reported an average score of 3.57 with standard error 0.28; women reported an average score of 1.91 with standard error 0.32.

After the intervention, the gender gap was smaller: the average score for men was 3.44 (SE 0.16); the average score for women was 3.18 (SE 0.16).

1. Make four `Normal` objects to represent the sampling distributions of the estimated means before and after the intervention, for both male and female students. Because we have standard errors for the estimated means, we don't need to know the sample size to get the parameters of the sampling distributions.
2. Compute the sampling distributions of the gender gap—the difference in means—before and after the intervention.
3. Then compute the sampling distribution of the difference in differences—that is, the change in the size of the gap. Compute a 95% confidence interval and a p-value.

Is there evidence that the size of the gender gap decreased after the intervention?

Index

C

D

E

F

G

H

I

N

Q

R

S

T

U

V

W

Z

About the Author

Allen Downey is a professor emeritus at Olin College and a consultant specializing in data science and Bayesian statistics. He is the author of several books—including *Think Python* (O'Reilly 2024), *Think Bayes* (O'Reilly 2021), and *Probably Overthinking It*—and a blog about programming and data science. He received a Ph.D. in computer science from the University of California, Berkeley, and bachelor's and master's degrees from MIT.

Colophon

The animal on the cover of *Think Stats*, third edition, is an archerfish, or spinner fish (*Toxotidae*). This family of fish preys on land-based insects and small animals, using their specialized mouths to shoot them down with water droplets. This family consists of seven species, which can be found from India to the Philippines, Australia, and Polynesia.

The archerfish has a deep body; the space between the dorsal fin and mouth forms a straight line. The protractile mouth has a lower jaw that juts out. The shape of its mouth lends itself directly to feeding: the narrow groove in the roof of its mouth allows it to squirt a jet of water at its victim by pressing its tongue against the groove and contracting its gills to force out the powerful jet of water, which can travel up to five meters. Archerfish learn how to shoot when they reach 2.5 cm long. Often they are inaccurate at first and hunt in small schools, eventually learning from experience.

The archerfish's eyes are also valuable tools for feeding. It has particularly good eyesight and is able to compensate for light refraction as it passes through the air-water interface when aiming at prey. Once it spots its prey, the archerfish rotates its eye so the image of the prey falls on a particular portion of the eye. Often, the archerfish will leap out of the water to grab the insect in its mouth, if within reach.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *Dover*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.