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

O'REILLY®

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.

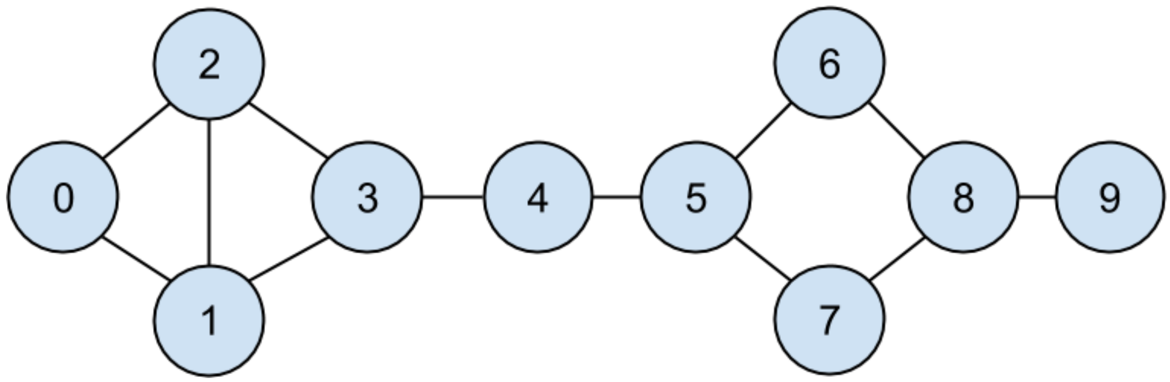

Figure 1-1. The DataSciencester network

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).

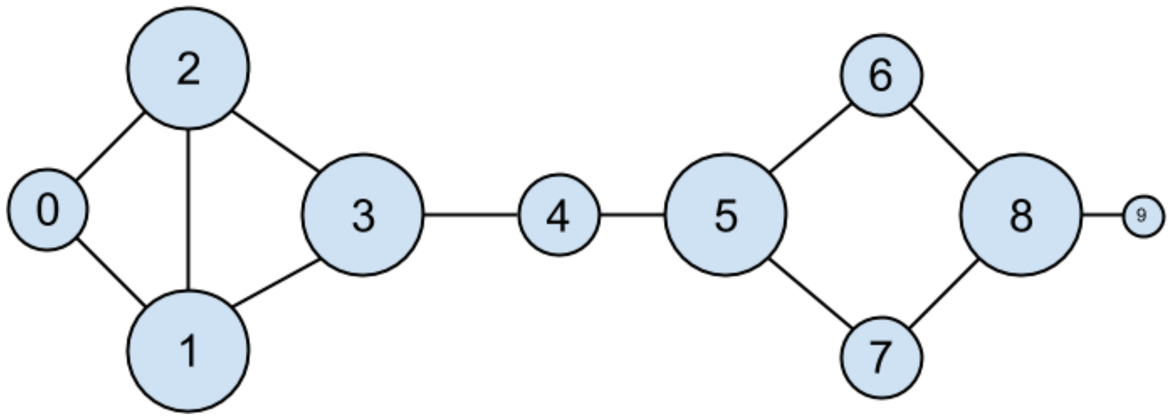

Figure 1-2. The DataSciencester network sized by degree

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:

```python
# 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 `defaultdicts`, 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 `click`ed 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-reset-able 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
```

```python
    # 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:

```python
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`ed values one at a time until none are left:

```python
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)`:

```python
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:

```python
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:

```python
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): ...
  ```

```python
# 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:

  ```python
  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.

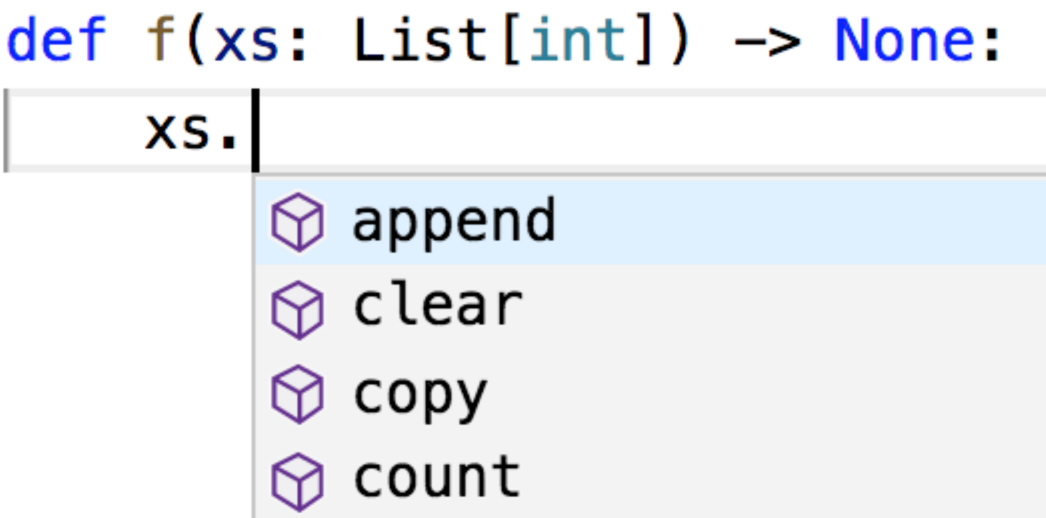

Figure 2-1. VSCode, but likely your editor does the same

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
```

```python
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:

```python
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:

```python
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.