The Crystal Ball Instruction Manual

Volume One: Introduction to Data Science

version 1.2

Stephen Davies, Ph.D.
Computer Science Department
University of Mary Washington

University of Mary Washington
Department of Computer Science
James Farmer Hall
1301 College Avenue
Fredericksburg, VA 22401

The LaTeXsource for this book is available from: https://github.com/rockladyeagles/crystal-ball-1.

Contents

Chapter 1

Introduction

If this marks your first exposure to the new and exciting discipline of *data science*, you occupy an enviable position. Still in front of you is all the cool stuff, even the first few sparks of magic when you learn how to plug data into electrical sockets, perform automated prediction, and write the first gems of code to probe the depths of an interesting data set. I'm a bit jealous, tbh, but am also excited to explore it all again with you, which is the next best thing!

This field has changed the world like hardly any other has, and on an incredibly short time scale, too. Just a couple decades ago, businesses and organizations were routinely making major decisions based on gut feelings and anecdotal observations. Doctors eyeballed sets of symptoms and diagnosed patients largely based on what conditions they themselves had seen before, or seen recently. Online sellers gave product recommendations that made sense to *them*, completely missing patterns and trends that would become apparent if the characteristics and purchasing patterns of past customers were taken into account.

Part of the reason decision makers made these suboptimal choices was because it wasn't yet clear how much punch data science would pack. Another reason was that the technology wasn't there yet: the processing power and storage capacity to work with extremely large data sets wasn't commonly available, and of course the data itself hadn't all been gathered yet. No more! All these parts are here

now. And somewhat incredibly, they're all at your disposal for low (or even no) cost.

This is the era of data science. If you want to understand and make an impact on your world, I can honestly think of no better field to dive into than this one, no matter what your sphere of interest. The ability to command these techniques and tools gives you both great insight and great power to influence how life on planet Earth proceeds from this day forward.

1.1 Defining Data Science

When people ask me what data science *is*, here's my go-to definition: **deriving knowledge from data**. But interpreting that phrase entails dissecting the difference between "knowledge" and "data," two related but different terms. And that brings me to the **data-to-wisdom hierarchy**, depicted in Figure 1.1. Let's break it down.

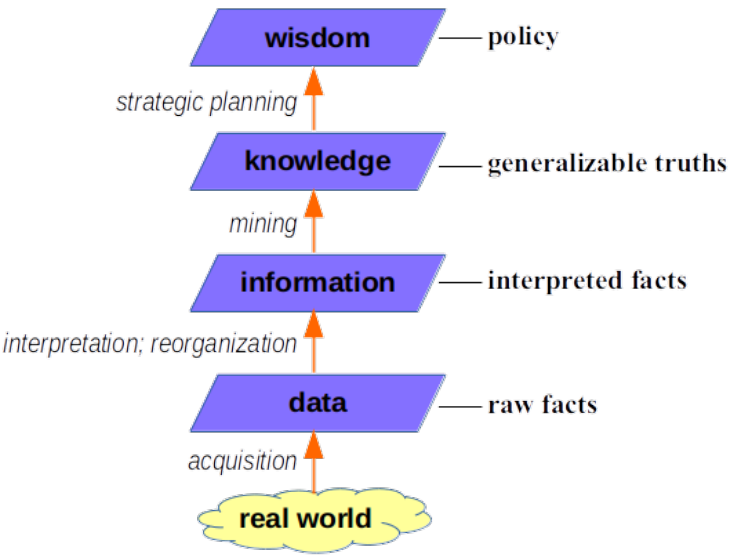


Figure 1.1: The data-to-wisdom hierarchy.

The real world

Ultimately, what we're interested in is not data, but aspects of the **real world** – album sales and video views, stock prices and employment rates, hurricane trajectories and virus hot spots, or whatever. Data science can't really get off the ground until some sort of **data acquisition** takes place that records measurements of the real world in electronic form.

This sounds obvious, but it's important to keep in mind, actually. No matter how much time we spend working with data, *it's never the data that actually matters – it's the real-world phenomenon the data represents.* It might seem strange to say that "data" is merely incidental to a data scientist, but it's true. And I've definitely seen more than one data scientist get so locked on to the data that they forget this basic truth.

One important observation is that decisions about exactly *which* data to acquire from the real world are often crucial in how things are interpreted later on. To take an example close to home, let's say we're gathering information on college professors so we can gauge which universities have the highest performing faculty, and how this might be changing over the years. We choose some representative set of criteria to measure for each faculty member to get a rough assessment of their performance. Let's say we choose three things: the number of research papers the professor publishes each year, the total amount of research funding they've been granted, and the average student evaluation score of the courses they teach. That seems like a good first cut at assessing "faculty performance." We then go on our merry data science way, finding correlations, making data visualizations, and drawing conclusions.

This is all fine and dandy, provided we always keep in mind that it was those three qualities, and *only* those three, that we gathered in the first place. If our study gains any traction, and university professors find they have a vested interest in being ranked high in our yearly study, we'll discover that they act to maximize *only* the categories that are being collected. We didn't gather data on how many university committees they served on, or how many independent studies they supervised, or how many advisees they had, *etc.*

Those metrics will inevitably become minimized in importance, because they weren't part of what we lifted out of the real world and onto the bottom rung of our lofty chain.

The moral is: what we measure matters, often more than we realize. Our country's GDP and the Dow Jones Industrial Average are easy things to quantify, and so we often do. And thus they gain great importance in analyses of the economy. But are they actually the most important indicators? Does focusing on them leave out other, perhaps more vital, benchmarks? I'll just leave you with that question for now.

Data

Have you ever gotten blood work done, say for an annual physical? I have. I like to look over the numbers when the doctor hands me the results, just to chuckle and wonder what they all mean. To me, a non-physician, they're all pretty much gobbledy-gook. They tell me my TBC is 4.93 x10E6/μL, that I have 5.7 Absolute Neutrophils, and a slightly out-of-range NT-proBNP (just 53.49 pg/mL, whatever the heck that means).

When I use the word **data** in the context of the hierarchy, this is what I mean: recorded measurements, often (but not always) quantitative, that have not yet been **interpreted**. They may be very precise, but they're also quite meaningless without the context in which to understand them. They'd even be meaningless to a *physician* if I didn't provide the labels; try telling your doctor that you have 4.93 "something" and see whether he/she freaks out.

The good news is that when we're at the data stage of the hierarchy, we at least have the stuff in an electronic form so we can start to *do* something with it. We also often make choices at this stage about how to **organize** the data, choosing the appropriate type of atomic and/or aggregate data structures that we'll discuss in detail in Chapters 3 and beyond. This will allow us to bring our analysis equipment to bear on the problem in powerful ways.

Information

Data becomes **information** when it *informs* us of something; *i.e.*, when we know what it means. Getting large amounts of data organized, formatted, and labeled the right way are jobs for the data scientist, since turning that morass into useful knowledge is impossible without those steps. When the aspects of the real world that we've collected are properly structured and conceptually meaningful, we're in business.

Knowledge

Now **knowledge** is where the real action is. As shown in Figure 1.1, knowledge consists of *generalizable* truths.

Here's what I mean. Information is about specific individuals or occurrences. When we say "Chandra is a female bank teller, and earns $48,000 a year," or "Austin is a male bank teller, and earns $69,000 a year," we have in our information repository some individual facts. They can be looked up and consulted when necessary, as you'll learn in the first part of this book.

But if we say "women make less money than men do, even at the same jobs," we're in a different realm entirely. We have now generalized from specific facts to more wide-reaching tendencies. In the language of our discipline, we've moved from information to knowledge.

Properly gleaning knowledge from information is a trickier business than interpreting individual data points. There are established rules, some of them mathematical, for determining when an apparent pattern is actually reliable, what kinds of relationships can be detected with data, whether a relationship is causal, and so forth. We'll build some important foundations with this kind of reasoning in this *Crystal Ball* volume and its follow-on companion. For now, I only want to make the point that *knowledge* – as opposed to mere information – opens up a whole new world of understanding. No longer is the world limited to a chaotic collection of individual observations: we can now begin to understand the general ways in which the world works...and perhaps even to change them.

Wisdom

Wisdom is the gold standard. It represents what we *do* with our knowledge. Let's say we indeed determine that on average men are paid higher than women in our country, even for the same jobs. What do we do with that realization? Is it okay? Do we want to try and fix it, and if so, how? With laws? Education? Government subsidies? Revolution?

You'll remember my definition of Data Science on p. 2: deriving *knowledge* from *data*. This implies that the "wisdom" level of the hierarchy is really outside the discipline, and belongs to other disciplines instead. And that's partially true: in some sense, the data scientist's job stops when the deep truths about the real world are ferreted out and illustrated, leaving it to CEOs, directors, and other policy makers to act on them. But the data scientist is often involved here too, for a simple reason: a decision maker wants to know what's likely to *happen* if a particular policy is implemented. Most non-trivial interventions will have results that are hard to predict in advance, as well as unintended side effects. One set of tools in the data scientist's toolkit is for making principled, calculated predictions about such things, as well as quantifying the level of **uncertainty** in the predictions. Sometimes, the technique of **simulation** is used – carrying out experiments on virtual societies or systems to see the likely aggregate effects of different interventions. It's like having a high-dimensional, multi-faceted crystal ball that lets you play out various scenarios to their logical conclusions.

Starting with the rough and tumble real world and helping produce wise decisions about how humankind can deal with it all: that's the grand promise of the data science enterprise. And those are the mighty waters you're about to dip your toes in! I hope you'll find it as exhilarating as I do.

1.2 A word of warning

Before we dive into the nitty gritty, let me leave you with one more general thought. It's actually an application of something

Spiderman once said: "with great power comes great responsibility."

Here's the deal. The skills you'll learn in this book are so powerful and (still!) so rare, that when you demonstrate them, people will think you can walk on water. If you continue in the discipline, you'll become highly sought-after (and well paid). People will constantly be asking you to work with new data, to produce plots, predictions, and insights, and basically to do your magic. You'll be treated as a guru: the oracle people go to when they want the scoop.

This is ultra-cool, but also dangerous. Why dangerous? One simple reason: because *when you make a data-related claim,* ***people will believe you.*** Pretty much unquestioningly. Most of your colleagues won't have the expertise or understanding to double-check your snazzy results. And it wouldn't occur to them to do that anyway – after all, you're the wizard.

The truth of the matter is that data science lives on the knife edge of **uncertainty**. With our crystal ball, we can make non-obvious assertions about the past or present and even predict the future, but as with all "knowledge," we must always hold it tentatively. We may be 95% confident that men are paid more than women...but that's only 95% confidence, not 100%. We may have reason to believe that raising the minimum wage in a city will decrease poverty by 3%...but there's a 1 in 20 chance that it might decrease it by as much as 6%, or even *increase* it by 1%.

The abiding principle is that you should always be forthright about the limits of your bold claims, the caveats behind your beautiful plots, and the level of likelihood that your hypotheses will turn out to be wrong. Admittedly, doing so will make you seem a little less magic. There are lots of talking heads on television who deliberately *obscure* the level of uncertainty in their analyses so that they seem more certain (and more impressive) than they really are. To be responsible data scientists, though, we're going to do the Spiderman thing and be up front and transparent about exactly what we've found, and what we might be missing.

Believe me, this will make you powerful enough!

Chapter 2

A trip to Jupyter

Python is a ridiculously popular **language** for programming and data science (currently the third most widely used in the world[1]) which is one of many reasons we're using it for this course. The language itself is different from the **programming environment** used to write code in it, just as "English" is different from "Microsoft Word" and "Google Docs." A programming environment is just a fancy name for a tool or application used to write programs. At a minimum, it must include a way to **edit** (write and revise) code, and a way to **execute** (run) it.

There are many different programming environments data scientists use to write Python code, just as there are many different word processing apps people use to write English. The choice largely comes down to personal preference. Some use full-blown **IDE**s ("integrated development environments") like Spyder or Atom; some use text-based tools like Notepad++ or `vim`. In this class, we're going to use the friendly and minimalistic "**Jupyter Notebooks**" environment since it's appropriate for an intro experience.

2.1 Jupyter Notebooks

The concept of a Jupyter Notebook is simple: it's a Web page with editable "**cell**s." Each cell is a little text window you can type in.

[1] See `https://www.tiobe.com/tiobe-index/`.

There are three kinds of cells in Jupyter Notebooks:

Raw. "Raw" is dumb. Never use it.

Markdown. "Markdown" cells are for *English text*, not Python code. They're mostly used to describe and annotate what you're doing in the code cells, like a running commentary. You can type plain-ol' text in a Markdown cell, plus various cutesy formatting adornments like boldface (putting double-splats (`**`) around a word or phrase), italics (single splats), outline headings (prefacing a line with one or more hashtags (like `#` or `###`), and so forth.[2] When you type in a Markdown cell, you see the raw text and formatting; to actually get Jupyter to **render** your cells and make them pretty, you choose "Run All" from the "Cell" menu.

Code. The most important cells are "Code" cells which contain (duh) code. When executed (again, by choosing "Run All" from the "Cell" menu) they actually carry out the Python instructions you have typed in that cell, and display any results.

By the way, a common snafu is to somehow accidentally click in a way that changes the type of a cell from "Code" to one of the other types. If you do this, the Python code in that cell won't execute until you change the type back to "Code" (more on this below).

Figure 2.1 shows a Jupyter Notebook hosted by the **CoCalc** cloud computing platform, which we'll use this semester. It has two cells, one Markdown and one Code. Note carefully the **cell-type dropdown** which is kind of hidden in the middle of the page: it currently reads "Code" because the second cell is the one that's highlighted. (If we clicked to highlight and edit the top cell, that dropdown would change to "Markdown.")

The top figure shows the two cells before the user has done a "Run All" from the "Cell" menu: all the Markdown is unrendered (see the literal splats and hashtags) and the code is just sitting there. After

[2] For a complete list of formatting options, see `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`.

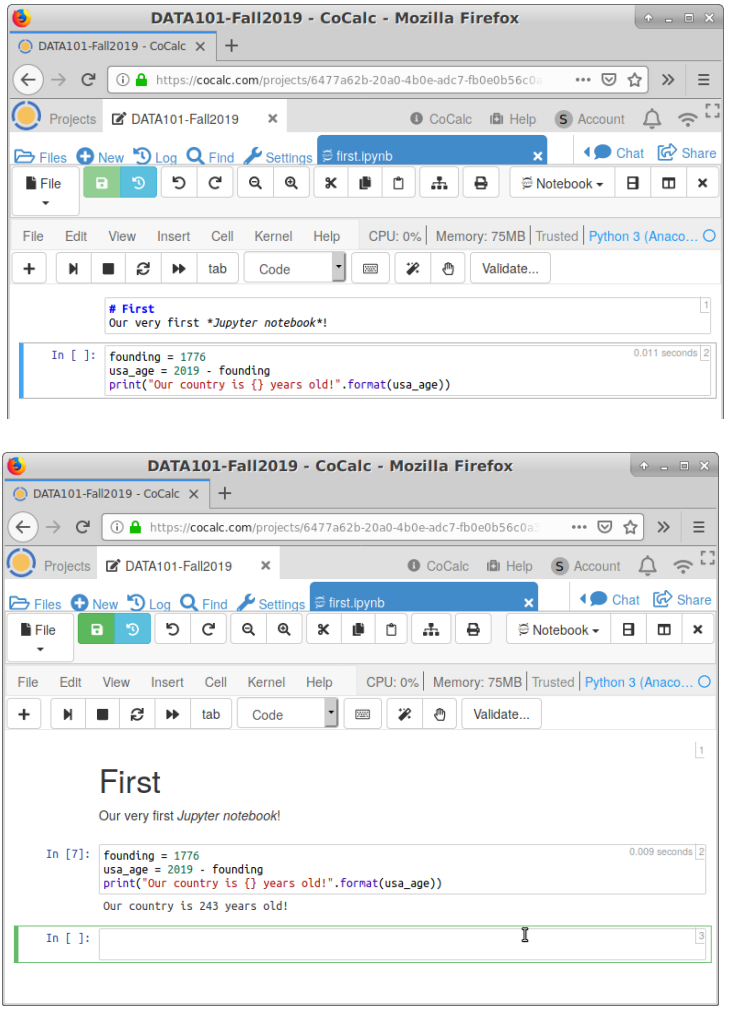


Figure 2.1: A Jupyter Notebook with one Markdown cell and one Code cell. In the top image, the two cells have been edited but not yet "run" – hence the Markdown formatting is unrendered and the code has not been executed. The bottom pane shows both cells after the use has chosen "Run All" from the "Cell" menu.

"Run All," the picture changes: you see the formatted message in the top cell, and the **output** of the Python code snippet after it runs. (The latter is easy to miss; stare at that bottom picture and

find the "`Our country is 249 years old!`" message. That's the "output.") We haven't yet covered what that Python code means (that's the main subject of this book) but you can probably guess some of what it's doing.

2.2 Code and output

Incredibly, that's about it. Everything else in this book is going to concern what to type in those Code cells and how to interpret its output.

From now on, whenever I give example Python code in this book, I'll write it in a box like this:

```
founding = 1776
usa_age = 2025 - founding
print("Our country is {} years old!".format(usa_age))
```

That box means "this stuff goes in a Code cell of a Notebook."

When I write the corresponding output (*i.e.*, what gets printed on the page immediately below the code cell when "Run All" is chosen from the "Cell" menu), I'll write it like this:

```
Our country is 249 years old!
```

That vertical bar means "this stuff is the printed result of executing the code cell."

Easy enough. Onward!

Chapter 3

Three kinds of atomic data

3.1 Atomic data

When we say that some data is "**atomic**," we don't mean it's radioactive; we mean it's *indivisible*.

The ancients spoke of "atoms" as the smallest possible bits of matter. If you divide up any physical object – say, an apple – into parts, you get its components: a stalk, a stem, skin, seeds, and the sweet juicy stuff. Cut up any of *those* pieces with a knife and you get smaller pieces. If you continue to split and split and split, philosophers like Democritus reasoned, you'll eventually get to tiny indivisible bits that cannot be further dissected. This is where the physical world bottoms out at the finest degree of granularity.

Similarly, a piece of atomic data is typically treated as an entire unit, not as something with internal structure that can be broken down. In the next chapter we'll learn about various ways that these atoms of data can be strung together and organized into larger wholes; for now, though, we're just looking at the atoms themselves.

3.2 Environments and variables

A data analysis program – of which we will write many in this course – makes use of an **environment** as it runs. "Environment" just means "all the data that is currently in view, and which the

program can access."[1] The environment consists of **variables**, each of which (usually) has a **name** and a **value**. For example, we might have a variable named `age` whose value is 21, and a variable named `slogan` whose value is `"Finger lickin' good"`.

Each variable in the environment must have a *distinct* name (*i.e.*, no two variables can share the same name). Also, importantly, the reason these building blocks are called "variables" is that their value can *change* as the program executes. Although we may initially create an `age` variable with the value `21`, later on in the program the variable's value might change to `22`, or `50`, or `0`. The variable's *name* never changes, though.

3.3 Atomic data types

There's one other thing that a variable has in addition to its name and value: a **type**.[2] In a programming language like Python, every piece of data has a specific type, which is necessary for determining how it behaves and what all you can do to it. A question you should ask yourself a lot is: "okay, I've got a variable in my environment called `x`...now what is its type?" You might have guessed (correctly) that our `age` and `slogan` variables from the previous section are of different types: one is a number, and the other is a phrase.

In this course, we'll principally deal with three types of atomic data, all of which will be familiar to you.

Whole numbers

One very common type of data is whole numbers, or integers. These are usually positive, but can be negative, and have no decimal point. Things like a person's birth year, a candidate's vote total, or a social media post's number of "likes" are represented with this data type.

[1] Confusingly, this use of the term "environment" is different from the term "programming environment" I introduced on p.9.

[2] Strictly speaking, although in languages like Java variables indeed have types, in Python the *values* have types, not the variables. This distinction will never be important for us though.

Real (fractional) numbers

You may remember from high school math that the so-called "real numbers" include not only integers, but also numbers with digits after the decimal point. This type can therefore be used to store interest rates, temperature readings, and average movie ratings on a 1-to-5 scale.

Since all whole numbers are themselves real numbers, you might wonder why we bother to define two different types for these. Why not just give both kinds of variables the same real number type? Basically, the answer is that something "feels wrong" about that to the Data Science community. A Facebook user might have 240 friends, or 241, but it would never make sense for her to have 240.3 friends. A consensus has thus arisen: variables that would only ever store whole numbers really ought to be of a type that's devoted to only whole numbers. You can violate this convention, but you'll be thought weird by your fellow developers if you do so.

Text

Lastly, some values obviously aren't numeric at all, like a customer's name, a show title, or a tweet. So our third type of data is textual. Variables of this type have a sequence of characters as values. These characters are most often English letters, but can also include spaces, punctuation, and characters from other alphabets.

By the way, this third data type can tiptoe right up to the "atomic" line and sometimes cross it. In other words, we will occasionally work with text values *non*-atomically, by splitting them up into their constituent words or even letters. Most of the time, though, we'll treat a character sequence like `"Avengers: Endgame"` as a single, indivisible chunk of data in the same way we treat a number like `42`.

But what about...?

What about other things a computer can store: images, song files, videos? It turns out that through clever tricks, all these kinds of media and more can be boiled down to a large number of integers,

and stored in an aggregate data structure like those discussed in the next chapter. At the atomic level, we'll really only ever need to deal with the three types of this section.

3.4 The three kinds in Python

Now the three kinds of atomic data described above are **language-general**: this means that they're conceptual, not tied to any specific programming language or analysis tool. *Any* technology used for Data Science will have the ability to deal with those three basic types. The specific ways they do so will differ somewhat from language to language. Let's learn about how Python implements them.

Whole numbers: `int`

One of the most basic Python data types is the "`int`," which stands for "integer." It's what we use to represent whole numbers.

In Python, you create a variable by simply typing its name, an equals sign, and then its initial value, like so:

```
revolution = 1776
```

This is our first **line of code**[3]. As we'll see, lines of code are **executed** one by one – there is a time before, and a time after, each line is actually carried out. This will turn out to be very important. (Oh, and a "line of code" is sometimes also called a **statement**.)

Python variable names can be as long as you like, provided they consist only of upper and lower case letters, digits, and underscores. (You do have to be consistent with your capitalization and your

[3]By the way, the word **code** is grammatically a mass noun, not a count noun. Hence it is proper to say "I wrote some code last night," not "I wrote some codes last night." If you misuse this, it will brand you as a newbie right away.

spelling: you can't call a variable `Movie` in one line of code and `movie` in another.) Underscores are often used as pseudo-spaces, but no other weird punctuation marks are allowed in a variable's name.[4]

And while we're on the subject, let me encourage you to *name your variables well.* This means that each variable name should reflect *exactly* what the value that it stores represents. Example: if a variable is meant to store the rating (in "stars") that an IMDB user gave to a movie, don't name it `movie`. Name it `rating`. (Or even better, `movie_rating`.) Trust me: when you're working on a complex program, there's enough hard stuff to think about without confusing yourself (and your colleagues) by close-but-not-exact variable names.[5]

Now remember that a variable has three things – a name, value, and type. The first two explicitly appear in the line of code itself. As for the type, how does Python know that `revolution` should be an "`int`?" Simple: it's *a number with no decimal point.*

As a sanity check, we can ask Python to tell us the variable's type explicitly, by writing this code:

```
type(revolution)
```

If this line of code is executed after the previous one is executed, Python responds with:

```
int
```

So there you go.

Here's another "**code snippet**" (a term that just means "some lines of code I'm focusing on, which are generally only part of a larger program"):

[4] Oh, and another rule: a variable name can't *start* with a digit. So `r2d2` is a legal variable name, but not `007bond`.

[5] And I fully own up to the fact that the `revolution` variable isn't named very well. I chose it to make a different point shortly.

```
revolution = 1776
moon_landing = 1969
revolution = 1917
```

Now if this were a math class, that set of equations would be nonsensical. How could the same variable (`revolution`) have two contradictory values? But in a *program*, this is perfectly legit: it just means that immediately after the first line of code executes, `revolution` has the value `1776`, and moments later, after the third line executes, its value has changed to `1917`. Its value depends entirely on "where the program is" during its execution.

Real (fractional) numbers: `float`

The only odd thing about the second data type in Python is its name. In some other universe it might have been called a "real" or a "decimal" or a "fractional" variable, but for some bizarre historical reasons it is called a `float`.[6]

All the same rules and regulations pertain to `float`s as they do to `int`s; the only difference is you type a decimal point. So:

```
GPA = 3.17
price_of_Christian_Louboutin_shoes = 895.95
interest_rate = 6.
```

Note that the `interest_rate` variable is indeed a `float` type (even though it has no fractional part) because we typed a period:

[6] If you're curious, this is because in computer programming parlance a "floating-point number" means a number where the decimal point might be anywhere. With an integer like -52, the decimal point is implicitly at the far right-hand side of the sequence of digits. But with numbers like -5.2 or -.52 or -.000052 or even 520000, the decimal point has "floated" away from this fixed position.

```
type(interest_rate)
```

```
float
```

Text: `str`

Speaking of weird names, a Python text variable is of type `str`, which stands for "**string**." You could think of it as a bunch of letters "strung" together like a beaded necklace.

Important: when specifying a `str` value, you must use **quotation marks** (either single or double). For one thing, this is how Python know that you intend to create a `str` as opposed to some other type. Examples:

```
slang = 'lit'
grade = "3rd"
donut_store = "Paul's Bakery"
url = 'http://umweagles.com'
```

Notice, by the way, that a *string of digits* is not the same as an integer. To wit:

```
schwarzenegger_weight = 249
action_movie = "300"

type(schwarzenegger_weight)
```

```
int
```

```
type(action_movie)
```

```
str
```

See? The quotes make all the difference.

The length of a string

We'll do many things with strings in this book. Probably the most basic is simply to inquire as to a string's **length**, or the number of characters it contains. To do this, we enclose the variable's name in parentheses after the word `len`:

```
len(slang)
```

```
3
```

```
len(donut_store)
```

```
13
```

As we'll see, the `len()` operation (and many others like it) is an example of a **function** in Python. In proper lingo, when we write a line of code like `len(donut_store)` we say we are "**calling the function**," which simply means to invoke or trigger it.

More lingo: for obscure reasons, the value inside the bananas (here, `donut_store`) is called an **argument** to the function. And we say that we "**pass**" one or more arguments to a function when we call it.

All these terms may seem pedantic, but they are precise and universally-used, so be sure to learn them. The preceding line of code can be completely summed up by saying:

> "We are **calling** the `len()` **function**, and **passing** it the `donut_store` **variable** as an **argument**."

I recommend you say that sentence out loud at least four times in a row to get used to its rhythm.

Note, by the way, that the `len()` function expects a `str` argument. You can't call `len()` with an `int` or a `float` variable as an argument:

```
schwarzenegger_weight = 249

len(schwarzenegger_weight)
```

```
TypeError: object of type 'int' has no len()
```

(You might think that the "length" of an `int` would be its number of digits, but nope.)

One thing that students often get confused is the difference between a named string *variable* and that of an (unnamed) string *value*. Consider the difference in outputs of the following:

```
slang = 'lit'
len(slang)
```

```
3
```

```
len('slang')
```

```
5
```

In the first example, we asked "how long is the value being held in the `slang` variable?" The answer was 3, since "`lit`" is three characters long. In the second example, we asked "how long is the word `'slang'`?" and the answer is 5. Remember: variable names never go in quotes. If something is in quotes, it's being taken *literally*.

Combining and printing variables

There's a whole lot of stuff you can do with variables other than just creating them. One thing you'll want to do frequently is **print** a variable, which means to dump its value to the page so you can see it. This is easily done by calling the `print()` function:

```
print(donut_store)
print(price_of_Christian_Louboutin_shoes)
print("slang")
print(slang)
```

```
Paul's Bakery
895.95
slang
lit
```

Again, don't miss the crucial difference between printing `"slang"` and printing `slang`. The former is literal and the latter is not. In the first of these, we're passing the *word* "`slang`" as the argument, not the variable `slang`.

Often we'll want to combine bits of information into a single print statement. Typically one of the variables is a string that contains the overall message. There are several ways to accomplish this, but the most flexible will turn out to be the `.format()` **method**.

I hate to hit you with so much new lingo. A **method** is very similar to a function, but not exactly. The difference is in the syntax used to call it. When you call a function (like `type()` or `len()`) you simply type its name, followed by a pair of bananas inside of which you put the arguments (separated by commas, if there's more than one). But when you "call a method," you put *a variable* before a dot ("`.`") and the method name, then the bananas. This is referred to as "calling the method **on** the variable."

It sounds more confusing than it is. Here's an example of `.format()` in action:

```
price_of_Christian_Louboutin_shoes = 895.95
message = "Honey, I spent ${} today!"
print(message.format(price_of_Christian_Louboutin_shoes))
```

Take note of how we write "`message.format`" instead of just "`format`". This is because `.format()` is a method, not a function. We say that we are calling `.format()` "on" `message`, and passing `price_of_Christian_Louboutin_shoes` as an argument.[7] Also be sure to notice the *double* bananas "`))`" at the end of that last line. We need both of them because in programming, every left-banana must match a corresponding right-banana. Since we're calling two functions/methods on one line (`print()` and `.format()`), we had two left-bananas on that line. Each one needs a partner.

As for the specifics of how `.format()` works, you'll see that the string variable you call it on may include pairs of curlies (squiggly braces). These are placeholders for where to stick the values of other variables in the output. Those variables are then included as arguments to the `.format()` method. The above code produces this output:

```
Honey, I spent $895.95 today!
```

Often, instead of creating a new variable name to hold the preformatted string, we'll just `print()` it literally, like this:

```
print("Honey, I spent ${} today!".format(
    price_of_Christian_Louboutin_shoes))
```

We're still actually calling `.format()` on a variable here, it's just that we haven't bothered to name the variable. Also, notice that

[7] Btw, in this book, whenever I refer to a method, I'll be sure to put a dot before its name. For example, it's not the "`format()`" method, but the "`.format()`" method.

our code was too long to fit on one line nicely, so we broke it in two, and indented the second line to make it clear that "`price_of_...`" wasn't starting its own new line. Crucially, all the bananas are still paired up, two-by-two, even though the left bananas are on a different line than the corresponding right bananas.

Finally, here's a longer example with more variables:

```
name = "Pedro Pascal"
num_items = 3
cost = 91.73
print("Customer {} bought {} items worth ${}.".format(name,
    num_items, cost))
```

```
Customer Pedro Pascal bought 3 items worth $91.73.
```

You can see how we can pass more than one argument to a function/method simply by separating them with commas inside the bananas.

Chapter 4

Memory pictures

Now that we've talked about the three important kinds of atomic variables, let's consider the question of *where they live*. It might sound like a strange question. Aren't they "in" the Jupyter Notebook cell in which they were typed?

Actually, no. And that brings me to the first mission-critical lesson of the semester, which is a bane to all students who don't deeply grasp it. The lesson is:

> **The code itself is only a means to an end. The purpose of the code is to read or write what's in memory.**

Memory is the part of the computer in which variables and their values are stored. To use the terminology of Chapter 3, memory is where the **environment** lives. It is *invisible* to the programmer, but it is also *very much there*. The single most important trick to learning how to write correct code is being able to summon to mind what memory looks like at any point in time. The code you must write is a natural consequence of that.

4.1 A picture of memory

It's easier with pictures at first, so we'll draw plenty of them. Our **memory pictures** will have a very specific format, and this is crucially important: don't get creative with how things are labeled or where things are drawn. In order for your code to work *you must have this picture exactly right.* It's not art; it's science.

Our memory pictures will always be divided into exactly two "realms," one on the left and one on the right, labeled as follows:

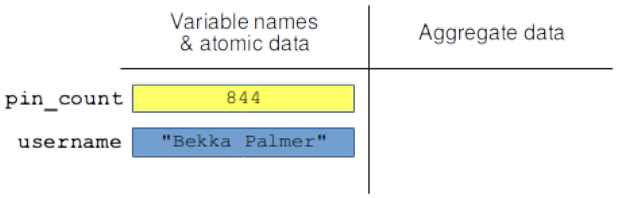
The left column's name should be recognizable, since that's exactly what we covered last chapter. The right column won't have anything in it for a couple chapters.

Writing to memory

When we create atomic variables in a Code cell, a la:

```
pin_count = 844
username = 'Bekka Palmer'
```

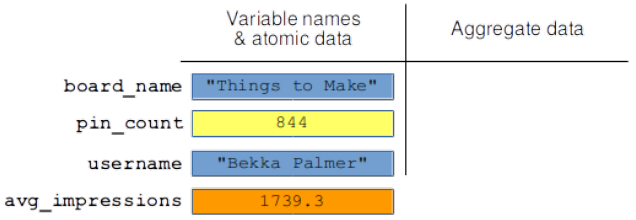
each one gets put on the left-hand side of the diagram as a **named box**. The name of the box is the variable's name, and the thing inside of the box is its value.



It doesn't matter which boxes are higher or lower on the page, only that the names stick with their boxes and don't get mixed up. As a bonus, I have colored the boxes differently, indicating that `pin_count` (an `int`) is a different type than `username` (a `str`).[1]

Creating more variables just adds more named boxes:

```
...
avg_num_impressions = 1739.3
board_name = "Things to Make"
```



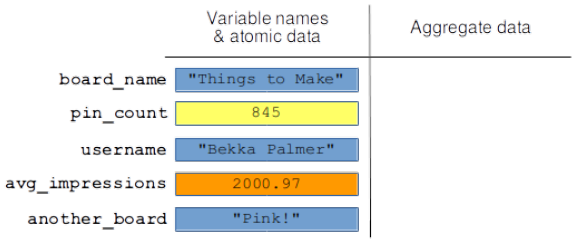
I'm deliberately shuffling around the order of the boxes just to mess with you. Python makes no guarantee of what "order" it will store variables in anyway, and in reality it actually does become a big jumbled mess like this under the hood. All Python guarantees is

[1] One other tiny detail you might notice: even though our code had single quotes to delimit Bekka Palmer's name, I put double quotes in the box in the memory picture. This is to emphasize that no matter how you create a string in the code – whether with single quotes or double – the underlying "thing" that gets written to memory is the same. In fact, what's stored are actually the characters `Bekka Palmer` *without* the quotes. I like putting quotes in the memory pictures, though, just to emphasize the string nature of the value.

that it will consistently store a name, value, and a type for each variable.

When we change the value of a variable (rather than creating a new one), the value in the appropriate box gets updated:

```
...
avg_num_impressions = 2000.97
pin_count = 845
another_board = 'Pink!'
```

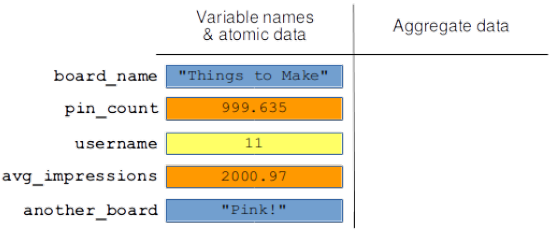


Note carefully that *the previous value in the box is completely obliterated* and there is absolutely no way to ever get it back. There's no way, in fact, to know that there even *was* a previous value different than the current one. Unless specifically orchestrated to do so, computer programs only keep track of the present, not the past.

One other thing: unlike in some programming languages (so-called "strongly typed" languages like Java or C++) even the *type* of value that a variable holds can change if you want it to. Even though the following example doesn't make much sense, suppose we wrote this code next:

```
...
pin_count = 999.635
username = 11
```

This causes not only the contents of the boxes to change, but even their colors. The `username` variable was a `str` a moment ago, but now it's an `int`.



Reading from memory

"Reading from memory" just means referring to a variable in order to retrieve its value. So far, we don't know how to do much with that other than print:

```
print("The {} board has {} pins.".format(another_board,
    pin_count))
```

```
The Pink! board has 999.635 pins.
```

The important point is that the memory picture is the (only) current, reliable record of what memory looks like at any point in a program. Think of it as reflecting a snapshot in time: immediately after some line of code executes – and right before the following one does – we can consult the picture to obtain the value of each variable. This is exactly what Python does under the hood.

I stress this point because I've seen many students stare at complicated code and try to "think out" what value each variable will have as it runs. That's hard to do with anything more than a few lines. To keep track of what-has-changed-to-what-and-when, you really need to maintain an up-to-date list of each variable's value as the program executes...which is in fact exactly what the memory picture is.

So if you're trying to figure out "what will this program output if I print the `odometer` variable immediately after line 12?" don't stare at the code and try to reconstruct its behavior from scratch. Instead, draw a memory picture, update it accordingly as you walk through each line of code, and then look at it for the answer.

Tip

By the way, investing in a small whiteboard and a couple of markers is a great way to help you learn programming. They're perfect for drawing and updating memory pictures as they evolve.

Hopefully this chapter was straightforward. These memory pictures will be getting increasingly complex as we learn more kinds of things to store, however, so stay sharp!

Chapter 5

Calculations

Our discipline obviously involves a lot of computation – in fact, I expect the first image that comes to mind when most people hear the words "data science" is one of numerical calculation. In this chapter, I'll lay out the Python syntax for performing various mathematical operations on numbers, as well as manipulating strings. These things appear in every program, and you'll find it all straightforward.

And then I'll drop a bomb on you. I'll unveil a Python behavior which you'll probably find completely unexpected, which flummoxes nearly every student who first sees it, and yet which you must understand and master to succeed in Python or any programming language.

5.1 Mathematical operations

First, the easy part. Python has a number of built-in **operators** to do the familiar math stuff. Figure 5.1 has a table of the ones we'll use. A few are mildly surprising (`*` instead of `X` for multiplication; `/` instead of ÷ for division, which I'll bet you couldn't find on your keyboard anyway), and you have to remember to use only bananas (not boxies `[]`, curlies `{}`, or wakkas `<>`) for grouping sub-expressions within a larger expression. Otherwise, it's a piece of cake.

Operator	Operation
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`**`	exponentiation ("to the power of")
`()`	grouping

Figure 5.1: Python's basic math operators.

All this stuff has to appear on the **right-hand side** of an equals sign, by the way, never on the left. That may seem surprising, since in mathematics the equations "$x = y + 3$" and "$y + 3 = x$" mean the same thing. Why does it matter which order you write it in? The answer, you'll recall, is that in a program the symbol "=" doesn't mean "*is* equal to" but rather "*make* equal to." It's not an equation; it's a command. And you can't command "$y + 3$" to be equal to anything. Therefore the only thing permitted on the left-hand side of an equals sign is a single, plain-jane variable name.

To test your understanding of the syntax, see if you agree that the following math expression:

$$\text{gpa} = \frac{\text{creds}_1 \cdot \text{gpts}_1 + \text{creds}_2 \cdot \text{gpts}_2}{\text{creds}_1 + \text{creds}_2}$$

should look like this in Python:

```
gpa = (creds1 * gpts1 + creds2 * gpts2) / (creds1 + creds2)
```

and that this one:

$$a = \frac{[x^2 y(4 - z) + (x + q) \cdot y] \times 2^{15y+2z}}{19x^3 - (yz)^{(y-1)^2}}$$

should look like this:

```
a = (((x**2)*y*(4-z) + (x+q)*y) * 2**(15*y+2*z)) / (19*(x**3) - (y*z)**((y-1)**2))
```

If so, you're good to go. It's tedious, but not complicated.

Python also has plenty of functions for absolute value, sine and cosine, logarithms, square roots, and anything else you can think of. We'll learn all those at the proper time (or they're all eminently Google-able if you want to look them up now).

A common pattern: cumulative totals

Here's a technique we'll use over and over in our code, but which can seem a bit jarring the first time you see it. Check out this line of code:

```
balance = balance + 50
```

Now there is no universe where that statement is true *mathematically*. (Think about it: can you come up with any number that is equal to itself plus fifty? I thought not.) But again, this is programming, not algebra. We're *commanding* the `balance` variable to have a new value. And what is that new value? Simple: whatever its *previous* value was, plus 50.

The net effect is to increase `balance`'s value by 50. Follow this:

```
balance = 1000
print("In July, I had ${}.".format(balance))
balance = balance + 50
print("In August, I had ${}.".format(balance))
balance = balance - 200
balance = balance + 120
print("In September, I had ${}.".format(balance))
```

```
In July, I had $1000.
In August, I had $1050.
In September, I had $970.
```

You get the idea. This approach will become especially useful when we get to **loops** in Chapter 14, because we'll be able to repeatedly **increment** a variable's value by a desired amount in automated fashion.

A couple other things. First, a very common special case of the above is to increment a variable by exactly *one*:

```
number_of_home_runs = number_of_home_runs + 1
```

This allows us to count the occurrences of various things: every time somebody hits a home run (or whatever), the above line of code will increase the appropriate **counter variable**'s value by one.

Second, Python has a special alternative syntax for this incrementing operation. It looks weird:

```
balance += 50
number_of_home_runs += 1
```

The two characters "+" and "=" (pronounced "plus-equals") allow us to shorthand this operation and avoid typing the variable name twice. The above two lines of code are *exact* synonyms for these:

```
balance = balance + 50
number_of_home_runs = number_of_home_runs + 1
```

You can use whichever one you wish, although be aware that your fellow programmers may well choose the former one, so you need to understand what it means.

5.2 String operations

Text data, too, has many things that can be done to it. For now, let's just learn a few techniques for **concatenating** strings (tacking

one onto the end of another) **trimming** strings (removing **whitespace**[1] from the ends) and changing their **case** (upper/lower). See Figure 5.2 for a list.

Method/operator	Operation
`+`	concatenate two strings
`.lstrip()`	remove leading whitespace
`.rstrip()`	remove trailing whitespace
`.strip()`	remove leading and trailing whitespace
`.upper()`	convert to all uppercase
`.lower()`	convert to all lowercase
`.title()`	convert to "title case" (capitalize each word)

Figure 5.2: A few of Python's string methods.

The plus sign is an operator, like the mathematical ones in Figure 5.1: it's used to concatenate (append) one string to another. Example:

```
x = "Lady"
y = "Gaga"
z = x + y
print(z)
```

```
LadyGaga
```

The second one is slapped right on the end of the first; there's no spaces or punctuation. If you wanted to insert a space, you'd have to do that explicitly with a string-that-consists-of-only-a-space (written as the three characters: quote, space, quote), like this:

```
first = 'Dwayne'
last = "Johnson"
full = first + ' ' + last
print(full)
```

[1]The word "whitespace" is a catch-all for spaces, tabs, newline characters, and most anything else invisible.

```
Dwayne Johnson
```

Punctuation marks, too, have to be included literally, and it can be tricky to get everything typed in the right way:

```
first = 'Dwayne'
last = "Johnson"
nick = 'The Rock'
full = first + ' "' + nick + '" ' + last
print("Don't ya just love {}?".format(full))
```

```
Don't ya just love Dwayne "The Rock" Johnson?
```

Stare at that line beginning with "`full =`" and see if you can figure out why each punctuation mark is where it is, and why there are spaces between some of them and not between others.

By the way, here's a bit of a head-scratcher at first:

```
matriculation_year = "2024"
graduation_year = matriculation_year + 4
print("Imma graduate in {}!".format(graduation_year))
```

```
Imma graduate in 20244!
```

Whoa – wut? That's a lot of tuition. The problem here is that `matriculation_year` was defined as a *string*, not an integer (note the quotes). So the `+` sign meant concatenation, not addition. Remember: a string-consisting-only-of-digits is not the same as a number. (If you remove the quotes from the first line, your mom will breathe easier and you'll get the result you expect.)

The other items in Figure 5.2 are methods: they have an initial dot ("`.`") and they must be called "**on** a string" (meaning, a string variable name must immediately precede them). They also take

no arguments, which means that a lonely, empty pair of bananas comes after their name when they are called. Examples:

```
shop_title = "               carl's ICE cream      "
print(shop_title)
print(shop_title.strip())
print(shop_title.upper())
print(shop_title.lower())
print(shop_title.title())
```

```
               carl's ICE cream
carl's ICE cream
               CARL'S ICE CREAM
               carl's ice cream
               Carl'S Ice Cream
```

(You can't see the trailing spaces in the output, but you can see the leading ones.)

You can even combine method calls back to back like this:

```
print(shop_title.strip().upper())
```

```
Carl'S Ice Cream
```

These operations are for more than mere prettiness. They're also used for **data cleansing**, which is often needed when dealing with messy, real-world data sets. If, say, you asked a bunch of people on a Web-based survey which Fredericksburg ice cream store they prefer, lots of them will name Carl's: but they'll type the capitalization every which way, forget the apostrophe, clumsily add spaces to one end (or even both, or even in the middle), yet they'll all have in mind the same luscious vanilla cones. One step towards conflating all these different expressions to the same root answer would be trimming the whitespace off the ends and converting everything to all lower-case. More surgical operations like removing punctuation or spaces in the middle is a bit trickier; stay tuned.

5.3 Return values

Okay, you've been in suspense long enough. Time for the bomb.

First, we're going to add another phrase to our already lengthy function-calling mantra. You'll recall that we summarized this code (a function call):

```
len(movie_title)
```

with this English:

> "We are calling the `len()` function, and passing it `movie_title` as an argument."

And we summarized this code (a method call):

```
message.format(name, age)
```

with this English:

> "We are calling the `.format()` method **on** the `message` variable, and passing it `name` and `age` as arguments."

Now, a third thing. We can use the equals sign with a variable name to capture the output of the function or method, instead of just printing it. The output of a function is called its **return value**. We say that "the `.upper()` method **returns** an upper-case version of the string it was called **on**." We can capture it like this:

```
big_and_loud = shop_title.upper()
```

The variable `big_and_loud` now holds the value `"CARL'S ICE CREAM"`. Functions work similarly:

```
width_of_sign = len(shop_title)
```

The `width_of_sign` int now has the value 40 (remember all those extraneous spaces); if we'd trimmed first, we'd have gotten 16:

```
true_width_of_sign = len(shop_title.strip())
print(true_width_of_sign)
```

```
16
```

The bomb

I've probably built this up too much, but I think you'll agree that the following output is pretty surprising:

```
diva = "Ariana Grande"
diva.upper()
print("I just love {}!".format(diva))
```

```
I just love Ariana Grande!
```

Wait...did the "`diva.upper()`" part just not work? Did it get skipped? Did we do it wrong somehow?

Even more confusing, putting the "`.upper()`" call directly in the `print` statement seems to work...but only temporarily. Accessing `diva` a moment later appears to revert it back to its old value:

```
diva = "Ariana Grande"
print("I just love {}!".format(diva.upper()))
print("When does the next {} album come out?".format(diva))
```

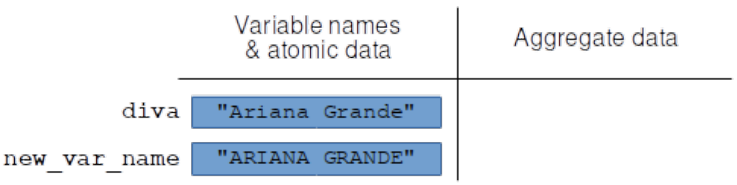
```
I just love ARIANA GRANDE!
When does the next Ariana Grande album come out?
```

The root cause of this and practically all perplexing Python printing can be discovered by consulting the memory picture. Here's how it starts out when we first define `diva`:

Now say we do this:

```
new_var_name = diva.upper()
```

The result is this picture:



And now we see the reason for it all. The contents of the `diva` variable itself are ***unchanged*** by the method call. Calling "`.upper()`" on `diva` didn't change the string value in `diva`: it merely *returned* a modified *copy* of the string.

Think of it this way: if I asked you, "what is your name in Pig Latin?" and you told me, that would not intrinsically *change* your actual name to be in Pig Latin. You would simply be "returning" to me the Pig Latin version of it in response to my query.

You could argue this behavior of Python's is dumb, or at best misleading, and I'm actually inclined to agree with you in this case. But of course beggars can't be choosers: someone took the time to write the `.upper()` method for us, so if we want to take advantage of it we have to use his/her owner's manual. And the fact is that many (perhaps even most) Python functions/methods – including many of the ones from Pandas, which we'll use extensively – are coded with this style: not actually modifying the variables they are passed, but instead returning to you a modified copy which you must store.

Now given that this is the case, it would at least be nice if I could tell you that it always, consistently worked this way. Then you could simply accept it and get used to it. Alas, no. There *are* functions/methods (lots of them) which *do* modify a parameter or the variable they were called on. So sometimes, our naïve approach of calling the method and expecting the variable to change is exactly what we need to do. The bottom line is: *there's no way of knowing without being told, or else reading the documentation.* We'll learn how to do the latter in a future chapter. For now, I'm simply telling you for the record that the methods in Figure 5.2 are all of the "return a modified copy" type, and giving you a heads up that both styles of method do exist out there in abundance.

A couple more things. First, as a corollary of the above, realize that the following statement (on a line by itself) is officially 100% useless:

```
name_of_pet.lstrip()
```

You called the `.lstrip()` method, and then....did nothing with the return value. If you don't store it in a variable – or else do something with it right away like `print` it before it slips out of your fingers – it's irrevocably lost: it doesn't even show up on the memory picture because there's no variable name. (Think about that.)

Second, note the following pattern which is very often used:

```
name_of_pet = name_of_pet.lstrip()
```

Here, we're calling `.lstrip()` on the `name_of_pet` variable *and then storing the return value back in the* `name_of_pet` *variable.* This might be what you thought would have happened in the first place – the author of the previous, useless line, probably wanted the variable itself to permanently have its leading spaces removed. Simply calling `.lstrip()` on the variable won't do that, but putting the revised value back in the same blue box on the memory diagram will.

Chapter 6

Scales of measure

In the last chapter, we learned the Python verbiage for how to do arithmetic operations. In this one, we zoom out and ask: when does it actually make *sense* to use those operations? The answer turns out to be: not always.

Another way to phrase this distinction is in terms of **syntax** vs. **semantics**. Syntax concerns the rules for combining various symbols in a programming (or other) language. Semantics concerns the *meaning* of those symbols. This isn't something a programming language can tell us. Only a human who understands what all those symbols refer to can determine when a particular combination actually relates to something meaningful.

6.1 The four scales of measure

Every variable[1] we collect can have various values, and the nature of information it contains can be described by its **scale of measure**.

[1]Note that our use of the term **variable** in this chapter is different than how we used it in chapter 3 (*e.g.*, p. 14) and throughout chapter 5. In this chapter, a variable is normally some measurable aspect of *every* object in our study. We might recruit participants to a research experiment, and record their race, weight, and favorite breakfast cereal. These would be our three variables. Each of the three will constitute *many* values, since our group of participants will have many races, weights, and cereals. In programming terms, they will eventually become **aggregate** data types of some kind.

There are four such scales of measure[2], and each one determines which kinds of operations are "legal" (*i.e.*, sensible) with that variable.

Categorical/nominal

The first kind is the simplest, although it actually has two different names in common use: they're called both **categorical** variables and **nominal** variables. These variables represent one of a set of predefined choices, where no choice is "higher" or "greater" than any other.

An example would be a `fave_color` variable that holds the value of a child's favorite color: legal values are `"red"`, `"blue"`, `"green"` or `"yellow"`. We know it's categorical from, among other things, the fact that there's no one right way to **order** those values. (Alphabetical, most-popular-first, and ordering according to the sequence of the rainbow are three possibilities. You might think of others.)

Political affiliation would be another categorical variable. Its values (like `"Democrat"`, `"Republican"`, and `"Green"`) aren't in any particular order. (Although you might think of the traditional left-to-right political spectrum, that's only one dimension of political party, and perhaps not even the most important one.) Other examples include a film's genre, a student's nationality, and a football player's position.

Now you might be tempted to think, "hmm...all the categorical examples so far are textual, not numeric. Perhaps this scales of measure thing is just another way of stating the variable type?" Alas, no. For one, we'll see text variables in the next category as well. For another, even data that on its surface seems numeric can actually be categorical in disguise.

Consider the uniform number of an athlete. I might be interested in asking, "which uniform number had the greatest professional athletes who chose it?" #24 is a good candidate: Willie Mays, Ken Griffey Jr., and Kobe Bryant all wore that jersey number. Or maybe

[2] According to psychologist Stanley Smith Stevens in 1946. Other researchers have developed related, but different, scales of measure.

#7 is the winner, with Mickey Mantle, John Elway, and Cristiano Ronaldo. Either way, though, all that matters in this analysis is *which* uniform number an athlete chose, not how high that number is compared to others. No one in their right mind would say that Peyton Manning (#18) was "twice the player" Mia Hamm was (#9), because uniform numbers aren't really *numbers* at all: they're more like labels.

Legal operations for categorical/nominal variables

When a variable is on a categorical scale, about the only things you can do are compare for equality/inequality, count the occurrences of different values, and compute something called the **mode** of the values.

The mode simply means the value that occurs *the most often*. It's the first of the "**measures of central tendency**" we'll see: such measures are a way of capturing something about the "typical" value of a variable. For categorical variables, the only typical-ness is "which one occurs the most often?" If we ask a bunch of people for their `fave_color`, and we get the answers `"blue"`, `"red"`, `"blue"`, `"blue"`, and `"yellow"`, then the mode is `"blue"`. It's that simple.

To wrap things up, these things make sense to ask of a *categorical* variable:

- 👍 "Is his favorite color the same as her favorite color?"
- 👍 "How many people have `"red"` as their favorite color?"
- 👍 "What's the most popular favorite color?"

while these do *not*:

- 👎 "Is his favorite color greater than her favorite color?" (??)
- 👎 "What's Caitlin's favorite color minus Hannah's?" (??)
- 👎 "What's the 'average' favorite color in this data set?" (??)

Ordinal

One step up on the food chain is an **ordinal** variable, which means that its different possible values *do* have some meaningful order.

Consider `education_level`, a variable that contains the highest degree a survey respondent has earned. Its values can be any of the following: `"HS"`, `"Associates"`, `"Bachelors"`, `"Masters"`, and `"PhD"`. In some ways, this is like `fave_color`: the variable must take on one of a set of specific, prescribed values. However, it's pretty clear that a High School degree is closer to (more similar to) an Associates degree than it is to a Ph.D. Each successive value represents more education, and so unlike categorical variables, it *does* make sense to compare them along greater-than-or-less-than lines.

In addition to the mode, another measure of central tendency available for ordinal variables is the **median**. I think of the median as the "middlest" value: if you line up all the occurrences in a row – in order of the values – it's the one that lies in the exact middle. Suppose our survey respondents give these answers: `"Bachelors"`, `"HS"`, `"HS"`, `"Masters"`, `"Masters"`, `"Bachelors"`, and `"HS"`. To compute the median, we line them all up in order:

```
"HS" "HS" "HS" "Bachelors" "Bachelors" "Masters" "Masters"
```

and find the middlest one, which is `"Bachelors"`. So `"HS"` is the mode of this variable, and `"Bachelors"` is the median.

Other examples of ordinal variables include an NCAA basketball team's top-25 ranking, a taxpayer's tax bracket, and survey questions asking whether you `"strongly disagree"`, `"disagree"`, are `"neutral"`, `"agree"`, or `"strongly agree"` with a certain statement.

Again, a list of do's and don't's. For *ordinal* variables, these are okay:

- 👍 "Is his education level the same as her education level?"
- 👍 "How many people answered `"strongly disagree"` to this question?"
- 👍 "Is UMW basketball ranked higher or lower than Messiah?"
- 👍 "What's the median tax bracket for this group of employees?"

while these are *not*:

- "Which looks like the bigger mismatch on paper: Duke v. Kentucky, or Villanova v. Gonzaga?" (??)
- "What's Caitlin's education level minus Hannah's?" (??)
- "What's the 'average' tax bracket for this group of employees?"

It's worth commenting on that second list, because you might have thought some of those items were completely reasonable. For example, suppose that in the latest AP poll, Duke is currently ranked #1, Kentucky #3, Villanova #4, and Gonzaga #23. You might think that clearly the Villanova/Gonzaga matchup is the most lopsided, since there's nineteen places between them, whereas Duke and Kentucky are separated by just two.

But not necessarily. We know Duke is considered *stronger* than Kentucky, but not *how much stronger*. It is almost certainly not the case that the teams are exactly evenly spaced all the way down the list from #1 to #25. Real life doesn't work like that. Instead, it might be the case that Duke and Georgetown, the #1 and #2 teams in the country, are considered *far and away* the best two teams. And perhaps the next five or even twenty teams on the list are considered very close, to the point where experts disagree wildly on what order they should be in. If this is the case, then mighty Duke vs. (comparatively) lowly Kentucky might be an enormous mismatch, while Villanova and Gonzaga might be considered a tossup.

The bottom line is: although an ordinal variable's values are *ordered*, there is no information at all about the *spacing* between them. I'll tell you from personal experience that the difference between a Bachelors and a Masters degree is nuthin' compared to that between a Masters and a Ph.D. (You can ask anyone who has earned the latter for confirmation.)

This leads into the second item on the no-no list: subtracting two ordinal values. All you're going to get is "the number of positions in the sequence by which they differ," which tells you next to nothing. If I ask people to rate a movie on a scale of `"POOR"`,

`"FAIR"`, `"GOOD"`, and `"EXCELLENT"`, the difference between `"POOR"` and `"GOOD"` is likely to be a lot greater than that between `"FAIR"` and `"EXCELLENT"`. This is true even though the "difference" between them seems exactly the same: two ranking's worth. The fact is that humans don't interpret those four adjectives as exactly equally spaced, and therefore it's a fallacy to interpret their results as though they did.

Which leads to the third and last item: trying to take the "average" (adding up all the scores and dividing by the total). It's tempting to say, "let's treat `"POOR"` as a 1, `"FAIR"` as a 2, `"GOOD"` as a 3, and `"EXCELLENT"` as a 4. Then, we can just take the mean of all the results to get the average rating! What's not to like?" Here's what's not to like. By assigning those numbers, you added spurious information and thereby twisted the respondent's meaning into something they didn't necessarily intend. They very likely didn't think of the four options as equally-spaced numerically, and so this average is quite bogus. Instead, take the median.

Interval

Onward. Our next scale of measure is the **interval** scale, which fulfills what was missing with ordinal variables. An interval variable *does* have meaningful and reliable differences between values, which can be computed and analyzed.

Unlike the previous two scales, interval variables are always numeric by nature. You can't subtract two words from one another, but you can do so with numbers, and unlike our uniform number and NCAA hoops ranking examples, that subtraction is a *meaningful* operation.

An example of an interval variable might be the longitude (or latitude) of a city. Not only can we ask whether two cities have the same longitude (as with categorical), and whether one is east or west of another (as with ordinal), we can now ask *how far* east. Subtract one longitude from the other, and boom. We have a reliable degree of difference.

This allows us to ask questions like "are Dallas and Fort Worth far-

ther apart than Minneapolis and St. Paul are?" or "is the temperature swing between daytime and nighttime wider in Colorado than in Virginia?" (Hint: yes.) Note that we couldn't legally ask such questions of an ordinal variable, since there was no way to really know how large the difference between `"GOOD"` and `"EXCELLENT"` was, as opposed to that between `"FAIR"` and `"GOOD"`.

Another example of an interval scale variable, besides the aforementioned temperature, is the year an event takes place. We can say, for example, that nearly two-thirds of our nation's history has occurred *after* the Civil War ($2021 - 1865 = 156$ years, versus $1861 - 1776 = 85$ years).

The quintessential measure of central tendency for interval scale is the arithmetic **mean**. Both the median and the mode are still permitted, and they are sometimes quite useful. But often we're going to fall back on the add-'em-up-and-divide-by-the-number-of-elements thing you learned in grade school. In this case, it makes sense, because the values are at fixed, meaningful, numerical positions and so adding them up is okay.

Here's our list of goods (for interval scale variables):

- 👍 "Was today's high temperature the same as yesterday's?"
- 👍 "Was Beethoven born before or after Napoleon?"
- 👍 "How many cities are at 40° latitude?"
- 👍 "What's the median year of birth for current U.S. Senators?"
- 👍 "Which is experiencing more global warming (temperature difference) – Greenland or France?"
- 👍 "What's London's latitude minus Boston's? How much farther north is it?"
- 👍 "What was the average high temperature in Fredericksburg in September?"

and bads:

- 👎 "Which cities are at least 20% more east than Chicago?" (??)
- 👎 "When was the first fall day which was half as hot as it was on July 4th?" (??)
- 👎 "Was Lincoln born 5% later than Washington?" (??)

Let's consider that bads list. With an interval scale variable, we can ask almost anything we want to about it. Almost. The one fly in the ointment is questions that have phrases like "twice as" or "10% less than." Those, we cannot do. The reason is that an interval scale variable *has no meaningful* ***zero point***.

In an interval scale, values have *relative* distances from each other, but not *absolute* differences from some fixed reference point. Consider years. Saying that the Cubs finally won the World Series 146 years after their franchise was born is meaningful: the difference between 1870 and 2016 can be measured. But what if we said "they won the World Series 7.8% later than their franchise was born"? Could such a sentence possibly say anything useful?

The answer is no, and here's why. The "zero point" of our calendar system is **arbitrary**. By that I mean that the year we might consider "year zero" has nothing to do with the Cubs or baseball or America or anything else: it was a guess as to the birth year of Jesus Christ, and a wrong one at that.[3]

We could, of course, have chosen to measure time relative to any other point instead, like the birth of our own nation, the founding of Rome, the Cubs franchise being founded, or anything else. If we had done that, all of the *relative* differences between years would have been the same: there would still have been 85 years between the Declaration of Independence and the Civil War, Barack Obama would still have been President for 8 years, and you would still be the same age. But all the *absolute* calculations that implicitly make reference to the zero point – like "what percent later did the Cubs won the Series than their franchise began?" would suddenly become radically different. If we measured years relative to 1776, then the Cubs' victory would have been "155.3% later" than their origin, instead of "7.8% later!" That betrays the fact that this is an

[3]Later historical discoveries have demonstrated that Herod the Great died in what we now call 4 B.C. If you went to Sunday School, you might recall that in a fit of jealousy, King Herod the Great ordered all the baby boys in Bethlehem (two years old or younger) to be killed. (See Matthew 2:13-18.) He chose "two years or younger" as the cutoff because his goal was to kill Jesus, who was about two years old at the time. Hence Jesus was most likely born in the year which we have (incorrectly, it turns out) labeled as "6 B.C." Fun facts.

utterly meaningless calculation.

Same thing with longitude. While latitude plausibly has a meaningful zero point – the equator – and thus perhaps "twice as north" has some meaning to it ("twice as far from the center of the planet") longitude clearly does not. Saying a city is "twice as east" as another is plain nonsense. That's because the zero point for longitude is arbitrary: it's set at the Greenwich, England, of all things. Clearly only relative differences between longitude have any meaning.

And the same thing with temperature. If yesterday's high was 40°, and today's is 80°, it's tempting to say "whew! It's twice as hot today!" To see that this is gibberish, though, consider what would happen if we changed to use the metric system like the rest of the civilized world does, and measured temperature in Celsius. Now if we did that, clearly we wouldn't start experiencing heat waves or cold spells as a result! Hey we're just changing our units, bro, not influencing the atmosphere. But realize that in Celsius, yesterday's 40°F day would become 4.4°C, and today's 80°F would be 26.7°C. So now, by changing our units, we would have to say "oh golly, I guess it's actually over *six times* as hot today!" This is why multiplying and dividing with interval scale variables leads to madness.

Ratio

Which brings us to our last of the four scales: the ratio scale. In some ways this is the easiest to understand, because of all the mathematical questions we might want to ask, we *can* ask them. Multiply, divide, make absolute statements like "25% greater than" – go crazy, man.

Salary has a meaningful, absolute zero point: namely, an unemployed (or volunteer) worker earning *zero* dollars. Since we have that non-arbitrary standard, it makes perfect sense to say things like "he makes twice as much as she does.

The height of a person has a meaningful zero point as well: the ground. If Tyrion Lannister rises $3\frac{1}{2}$ feet from the floor, and Gregor Clegane stands a full 7 feet from that same floor, it makes all the sense in the world to say "Gregor is twice as tall as Tyrion."

As with interval scale variables, we often use the arithmetic **mean** as our measure of central tendency.[4]

6.2 Final word

The lesson of this chapter is that Python will *not* prevent you from doing any of the above stupid things – if we have an ordinal scale variable, for instance, we can subtract values from one other until we're blue in the face, not recognizing that the results we're producing are gibberish. It's all on us to be responsible data citizens, and to only use operations that give meaningful results.

[4] Interestingly, there are actually two different kinds of means, one of which, called the "geometric mean" is only applicable on the ratio data scale. It involves multiplying and taking roots instead of adding and dividing, and is a useful operation in some niche contexts.

Chapter 7

Three kinds of aggregate data

Now it's time to consider some loftier goals for our lowly atomic bits of data. Most anything interesting in Data Science comes from arranging them together in various ways to form more complex structures. This chapter is the subject of these.

7.1 Aggregate data types

The number of ways in which pieces of data can be arranged is far greater than the number of different atomic types. These various ways all have names, some of them nerdy and/or exotic like "hash tables," "binary search trees," and "skip lists." Nevertheless, there are again three basic ones which will form the basis of our study: they're called **arrays**, **associative arrays**, and **tables**. As before, we'll consider each one conceptually first, and then look at how to use them in Python.

Arrays

An **array** is simply a sequence of items, all in a row. We call those items the "**elements**" of the array. So an array could have ten whole numbers as its elements, or a thousand strings of text, or a million real numbers.

Normally, we will deal with **homogeneous** arrays, in which all the elements are of the same type; this turns out to be what you want

99% of the time. Some languages (including Python) do permit creating a **heterogeneous** array, which could hold (say) three whole numbers, sixteen reals, and four strings of text all mixed together. But usually you're using an array to contain a bunch of related values, like the current balances of all the accounts in your bank, or the Twitter screen names of all a user's followees.

Figure 7.1 shows what those two examples would look like conceptually. One has four strings of text, and the other five real numbers. Note that each *entire set* of elements is *one* variable. We might call the left one "`followees`" and the right one "`balances`."

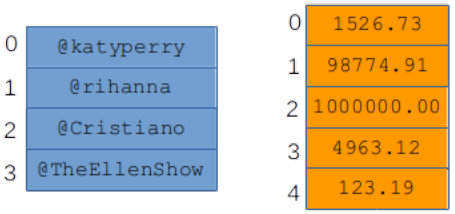


Figure 7.1: Two arrays.

Worthy of special note are the numbers on the left-hand side. These numbers are called the **indices** (singular: **index**) of the array. They exist simply so we have a way to talk about the individual elements. I could say "element #2 of the `followees` array" to refer to `@Cristiano`.

And yes, you noticed that the index numbers start with 0, not 1. Yes, this is weird. The reason I did that it is because nearly all programming languages (including Python) number their array elements starting with zero, so you might as well just start getting used to it now. It's really not hard once you get past the initial weirdness.

Arrays are the most basic kind of aggregate data there is, and they are the workhorse of a whole lot of Data Science processing. Sometimes they're called **lists**, **vectors**, or **sequences**, by the way. (When a particular concept has lots of different names, you know it's important.)

Associative arrays

An **associative array**, by contrast, has no index numbers. And its elements are slightly more complicated: instead of just bare values, an associative array contains **key-value pairs**. Figure 7.2 shows a couple of examples. The left-hand side of each picture shows the keys, and the right-hand side the corresponding value.

With an associative array, you don't ask "what's element #3?" like you do with a regular array. Instead, you ask "what value is associated with the `"Baltimore"` key?" And out pops your answer (`"Ravens"`).

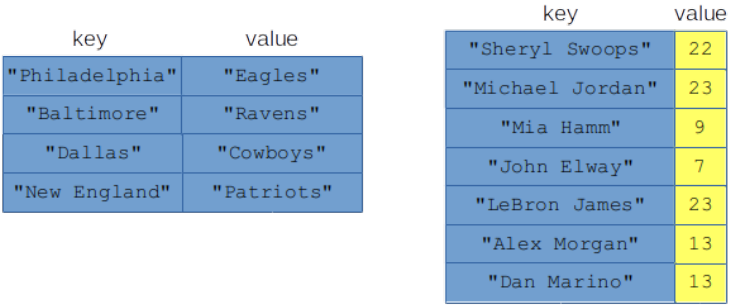


Figure 7.2: Two associative arrays.

All access to the associative array is through the keys: you can change the value that goes with a key, retrieve the value that goes with a key, or even retrieve and process *all* the keys and their associated values sequentially.[1] In that third case, the order in which you'll receive the key-value pairs is **undefined** (which means "not guaranteed to be consistent" or "not necessarily what you'd expect.") This underscores the fact that there isn't any reliable "first" key-value pair, or second, or last. They're just kind of all equally "in there." Your mental model of an associative array should just think of keys that are **mapped** to values (we say that `"Dallas"` is "mapped" to `"Cowboys"`) without any implied order. (Sure, the `"Philadelphia"`/`"Eagles"` pair is at the top of the picture, but that's only because I had to put *something* at the top of the picture,

[1] Using something called a "loop," which we'll learn about a little later in the book.

and I chose Philadelphia at random. It doesn't have any meaningful primacy though.)

Note a couple things about Figure 7.2. First, the keys in an associative array will almost always (and for us, *always*) be homogeneous. Similarly, the values will be homogeneous. But the keys might not be of the same type as the values. In the left picture, both keys and values are text, but in the right picture, the keys are text and the values (uniform numbers of famous athletes) are whole numbers. This is perfectly healthy and good.

Second, realize that the *keys* in an associative array must be **unique** – this means that there can be no duplicate keys. If we tried to create a second `"Alex Morgan"` (oh, if only...) with a different value, that new value would *replace* her current value, not sit alongside the first one as an additional key-value pair.

The reverse is not true, however: the *values* of an associative array may very well not be unique. In the left-hand picture they are, but in the right-hand picture there are duplicates: both Jordan and LeBron wore #23 in their stellar careers, while Hall of Famer quarterback Dan Marino once chose the same uniform number that Alex wears today. This isn't a problem, because we always access the information in an associative array *through the keys*. Asking "what number did Mia Hamm wear?" gives us a well-defined answer. Asking "which famous athlete wore #23?" does not. That's why we can't ask that second question (and aren't meant to).

Tables

Lastly, we have the table, which in Data Science is positively ubiquitous. In Figure 7.3 we return to the pinterest.com example, with a table of their most popular users. As you can see, it has more going on that than the previous two aggregate data types. Still, it's pretty straightforward to wrap your head around.

Unlike the other two aggregate data types, tables are full-on two-dimensional. There's (theoretically) no limit to how many **rows** and how many **columns** they can have. By the way, it's important to get those two terms straight: rows go across, and columns go up

screenname	pins	followers	real name	followers per board
"@cathiehong"	21122	7975315	"Cathie Hong"	134136.3
"@bekkapalmer"	13763	8479803	"Bekka Palmer"	229641.8
"@poppytalk"	21122	8129040	"Jan Halvarson"	11617.3
"@ohjoy"	16097	12645798	"Joy Cho"	99074.2
"@maryannrizzo"	110927	8951932	"Maryann Rizzo"	122019.8
"@stylemepretty"	162613	5879142	"Abby Leif"	80943.0

Figure 7.3: A table.

and down. (Think of the columns in the rotunda in James Farmer Hall.) Also, the typical table has many, many more rows than columns, so they're super tall and skinny, not short and fat.

Although the *rows* of a table are often heterogeneous, each *column* must be homogeneous. You can see that with a glance at Figure 7.3. Each column represents a specific type of data – in this case, some statistic or piece of information about a Pinterest user. Clearly all screen names should be of a text type, all "number of pins or followers" should be whole numbers, *etc.* It doesn't make sense otherwise.

As with the other types, the whole dog-gone table – no matter how many millions of rows it might have – is *one* variable with a single name. Also, just like with associative arrays, there normally isn't any implied *order* to the rows. Many implementations of these data types (including Python/Pandas) will actually let you specify "the first row" or "the 53rd row," but that always makes me cringe because conceptually, there isn't any such thing. They're just "rows" that are all "in there."

"Querying" tables (and other things)

Now you might be wondering how to actually "get at" the individual values of a table. Unlike arrays, there's no index number. And unlike associative arrays, there's no key. How then to address, say, the `@poppytalk` row?

The answer will turn out to be something called a **query**, which is a geeky way of saying "a set of criteria which will match some, but

not all, of the rows and/or columns." For instance, we might say "tell me the pin count for `@ohjoy`." Or, "give me all the information for any user who has more than 100,000 followers per board *and* at least 20,000 pins." Those specific requirements will restrict the table to a subset of its rows and/or columns. We'll learn the syntax for that later. It's a bit tricky but very powerful.

By the way, it turns out we'll actually be using the concept of a query for arrays and associative arrays as well. So strictly speaking, a query isn't just a "table thing." However, they're especially invaluable for tables, since they're essentially the *only* way to access individual elements.

7.2 Aggregate data and memory pictures

Recall from chapter 4 (p. 26) that the right-hand side of our memory pictures bore the label "Aggregate data." You may have anticipated that that's where the stuff in this chapter will live, and you're right. But there's a catch. Remember that variable *names* live on the left-hand side, and that's true even if the variable is of an aggregate type! This turns out to be crucially important, so I'm going to make a big deal about it.

You must draw your memory pictures (either on a whiteboard, or in your head) in a very specific way, and that way is illustrated in Figure 7.4.

Study this picture carefully, and notice several vitally important things. First, *all* the variable names are on the left-hand side – whether aggregate or not. This is always, always true.

Second, the actual array, associative array, and table depicted in this diagram are on the right-hand side. The variable name on the left "**points to**" the data in question with a little arrow. The technical name for this arrow is a **pointer** or a **reference**. The rule is simple: each atomic variable (like `gpa` or `course`) contains *the colored box itself.* Aggregate variables (like the other four) contain *a pointer to the group of colored boxes.*

Finally, mull over the fact that two *different* variables in this mem-

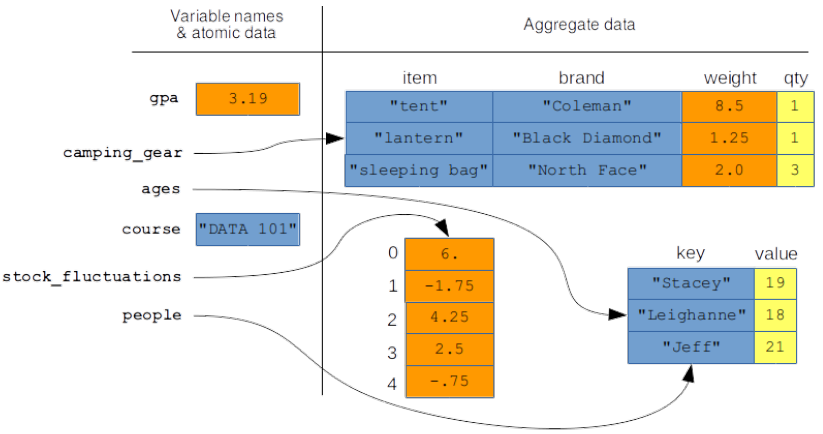


Figure 7.4: Where aggregate data variables – and their variable names – live in memory.

ory picture are pointing to the same thing! (`ages` and `people`) Believe it or not, this is a normal occurrence. The consequence is that if Stacey had a birthday, and we increased her age from 19 to 20 in the associative array, *both* `ages` *and* `people` would automatically see the new value. There is only one copy of that associative array in memory, and both variable names point at it.

It may seem like I'm being pedantic with this left-side-right-side stuff and all the little arrows. **I promise you I'm not.** The moment your data analysis program gets even mildly complicated, you will do the *wrong* thing and get the *wrong* answers if you don't think of it exactly like this. So take your time and commit it to memory. (See what I did there?)

Chapter 8

Arrays in Python (1 of 2)

There are several candidates in the Python language for representing the type of array structure we introduced in chapter 7. One is the plain-ol' Python `list`, which you may have used if you've taken a computer science course in Python. Turns out, `lists` are going to be too slow for us once we start dealing with a lot of data, plus there are a lot of things that it won't do for us automatically that are handy to have. Another choice is the Pandas `Series` which we'll actually introduce in chapter 11 – oddly, that one turns out to do too *much*, rather than too little, for our purposes here. A happy medium is the `ndarray` from the NumPy package[1]. Before we do that, however, we need to learn what a "package" actually is, and how to use one.

8.1 Packages

Back in my day (circa 1990's) when someone wanted to write a computer program, they wrote the entire thing themselves, line by line. Everything you needed to do – from something complex like making a remote network connection to something simple like computing the average of some numbers – was up to you to build. Code sharing over the Internet just wasn't much of a thing.

[1]Most people seem to pronounce this "NUM-pie," although I've heard "NUM-pee" as well. Pick your poison.

Today, the reverse is true. When you write a complex data analysis program, *most* of the code will actually be written by others, if you do it right. This is because many, many smart people across the globe have written snippets of code to do all the common (and some not-so-common) things you'll want to do, and your job is to string them all together. Put another way: you're given most of the Legos® – and even a bunch of pre-assembled chunks made with dozens of Legos® each – and your job is to construct your masterpiece out of those building blocks.

In Python, a **package** is a repository of useful functions and methods that someone else has written. By **import**ing a package into your program, you're making all those useful things available to you. Your own code can then call those functions/methods whenever you see fit. It's the modular, organized, and elegant way to do things, in addition to saving a ton of time.

The first package we'll use is called **NumPy**, which stands for "Numerical Python." To import it, you should include this exact line of code in the *first* Code cell of your Notebook:

```
import numpy as np
```

Note that it's in all lower-case letters. Once that cell has been executed, you now have access to all the NumPy "stuff," which is the subject of this chapter.

8.2 The NumPy `ndarray`

The actual data type that the NumPy package provides is called an `ndarray`, which stands for "n-dimensional array." If that sounds heady, it kind of is, although in this course we're only ever going to use a *one*-dimensional array, which is super simple to understand. In fact, it looks exactly like the examples in Figure 7.1 (p. 54).

"One-dimensional" just means that there is a single index number, and the elements are all in a line.[2]

Creating ndarrays

There are many different ways to create an `ndarray`. We'll learn four of them.

Way 1: `np.array([])`

The first is to use the `array()` function of the NumPy package, and give it all the values explicitly. Here's the code to reproduce the Figure 7.1 examples:

```
followees = np.array(['@katyperry','@rihanna','@TheEllenShow'])
balances = np.array([1526.73, 98774.91, 1000000, 4963.12, 123.19])
```

It's simple, but don't miss the syntactical gotcha: *you must include a pair of boxies inside the bananas.* Why? Reasons.[3] For now, just memorize that for this function – and this function only – we use "`([`*...stuff...*`])`" instead of "`(`*...stuff...*`)`" when we call it.

By the way, the attentive reader might object to me calling `array()` a function, instead of a method. Isn't there a word-and-a-dot before it, and isn't that a "method thing?" Shrewd of you to think that, but actually no, and the reason is that "`np`" isn't the name of a variable, but the name of a *package*. When we say "`np.array()`" what we're saying is: "Python, please call the `array()` function *from the* ***`np`*** *package*." The word-and-dot syntax does double-duty.

We can call the `type()` function, as we did back on p. 17, to verify that yes indeed we have created `ndarray`s:

[2] A two-dimensional array is a spreadsheety-looking thing also called a **matrix**. Each element has *two* index numbers: a row and a column. A three-dimensional array is a cube, with three index numbers needed to specify an element. *Etc.*

[3] For the experienced reader, what we're actually doing here is creating a plain-ol' Python list (with the boxies), and then calling the `array()` function with that list as an argument.

```
print(type(followees))
print(type(balances))
```

```
numpy.ndarray
numpy.ndarray
```

This is useful, but sometimes we want to know what underlying atomic data type the array is comprised of. To do that, we attach "`.dtype`" (confusingly, *without* bananas this time) to the end of the variable name. "`.dtype`" stands for "data type." Here goes:

```
print(followees.dtype)
print(balances.dtype)
```

```
dtype('<U13')
dtype('float64')
```

Whoa, what does *that* stuff mean? It's a bit hard on the eyes, but let me explain. The underlying data type of `followees` is (bizarrely) "`<U13`" which in English means "strings of Unicode characters[4], each of which is 13 characters long or less." (If you bother to count, you'll discover that the longest string in our `followees` array is the last one, `'@TheEllenShow'`, which is exactly 13 characters long.) The "`float64`" thing means "`float`s, each of which is represented with 64 bits[5] in memory.

You don't need to worry about any of those details. All you need to know is: if an array's `dtype` has "`<U`" in it, then it's composed of strings; and if it has the word "`int`" or "`float`" in it, it means one of those two old friends from chapter 3.

[4] A "Unicode character" is just a fancy way of saying "a character, which might not be English." NumPy is capable of storing more than just a-b-c's in its strings; it can store symbols from Greek, Arabic, Chinese, *etc.* as well.

[5] A "bit" – which is short for "binary digit" – is the tiniest piece of information a computer can store: it's a single 0 or 1.

Incidentally, you'll recall from chapter 7 that an array is *homogeneous*, which means all its elements are of the same type. NumPy enforces this. If you try to combine them:

```
weird = np.array([3, 4.9, 8])
strange = np.array([18, 73.0, 'bob', 22.8])
```

you'll discover that NumPy converts them to all be of the same type:

```
print(weird)
print(weird.dtype)
print(strange)
print(strange.dtype)
```

```
[ 3.   4.9  8. ]
dtype('float64')
['18' '73.0' 'bob' '22.8']
dtype('<U4')
```

See how the `int`s `3` and `8` from the first array were converted into the `float`s `3.` and `8.`; meanwhile, all of the numerical elements of the second array got converted to `str`s. (If you think about it, that's the only direction the conversions could go.)

Way 2: `np.zeros()`

It will often be useful to create an array, possibly a large one, with all elements equal to *zero* initially. Among other scenarios, we often need to use a bunch of counter variables to, well, count things. (Recall our incrementing technique from Section 5.1 on p. 33.) Suppose, for example, that we had a giant array that held the numbers of likes that each Instagram photo had. When someone likes a photo, that photo's appropriate element in the array should be **incremented** (raised in value) by one. Similarly, if someone unlikes it, then its value in the array should be **decremented** by one.

An easy way to do this is NumPy's `zeros()` function:

```
photo_likes = np.zeros(40000000000)
```

(although I'll bet you don't have enough memory on your laptop to actually store an array this size! Instagram sure has a lot of pics...) When I do this on my Data Science cluster, I get this:

```
print(photo_likes)
print(photo_likes.dtype)
```

```
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
float64
```

Don't miss the "`...`" in the middle of that first line! It means "there are (potentially) a lot of elements here that we're not showing, for conciseness." Also notice that `zeros()` makes an array of `float`s, not `int`s.

Way 3: `np.arange()`

Sometimes we need to create an array with regularly-spaced values, like "all the numbers from one to a million" or "all even numbers between 20 and 50." We can use NumPy's `arange()` function for this.

Normally we pass this function two arguments, like so:

```
usa_years = np.arange(1776, 2025)
print(usa_years)
print(usa_years.dtype)
```

```
[1776 1777 1778 1779 ... 2021 2022 2023 2024]
int64
```

If you read that code and output carefully, you should be surprised. We asked for elements in the range of 1776 to 2025, and we got...1776 through *2024*. Huh?

Welcome to one of several little Python idiosyncrasies. When you use `arange()` you get an array of elements starting with the first argument, and going up through *but not including* the last number. There's a reason Python and NumPy decided to do it this way[6], but for now it's just another random thing to memorize. If you forget, you're likely to get an "OBOE" – which stands for "off-by-one error" – a common programming error where you do *almost* the right thing but perform one fewer, or one more, operation than you meant to.

Anyways, other than that glitch, you can see that the function did a useful thing. We can quickly generate regularly-spaced arrays of any range of values we like. By including a third argument, we can even specify the **step size** (the interval between each pair of values):

```
twentieth_century_decades = np.arange(1900, 2010, 10)
prez_elections = np.arange(1788, 2028, 4)
print(twentieth_century_decades)
print(prez_elections)
```

```
[1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000]
[1788 1792 1796 1800 ... 2012 2016 2020 2024]
```

Notice we had to specify 2010 and 2028 as the second argument to these function calls in order for the arrays to include 2000, and 2024, respectively. This is the same "up to but not including the end point" behavior, but extended to step sizes of greater than one.

[6]Certain common operations are claimed to be "simpler" when you make a range function work this way. I personally don't buy it: I think it should work in the way you probably expected (including the second argument). I didn't get a vote, though.

Way 4: `np.loadtxt()`

Most of the data that we analyze will come from external **file**s, rather than being typed in by hand in Python. For one thing, this is how it will be provided by external sources; for another, it's infeasible to actually type in anything very large.

Let me say a word about files. You probably work with them every day on your own computer, but what you might not realize is that fundamentally, they all contain the same kind of "data." You might think of a Microsoft Word document as a completely different kind of thing than a GIF image, or an MP3 song file, or a saved HTML page. But they're actually more alike than they are different. They all just contain a bunch of bits. Those bits are organized to conform to a particular kind of file specification, so that a program (like Word, Photoshop, or Spotify) can recognize and understand them. But it's still just "information." The difference between a Word doc and a GIF is like the difference between a book written in English and one written in Spanish; it's not like the difference between a bicycle and a fish.

In this course, we'll be working with **plain-text files**. This is how most of the open data sources on the Internet provide their information. A plain-text file is one that consists of readable characters, but which doesn't contain any additional formatting (like boldface, colors, margin settings, *etc.*). You can actually open up a plain-text file in any text editor (including Microsoft Word) and see what it contains.

In your CoCalc account, you have your own little group of files which, like those on your own computer, can be organized into **directories** (or **folders**[7]). *It is critically important that the data file you read, and the Jupyter Notebook that reads it, are in the same directory.* The #1 trouble students experience when trying to read from a text file is not having the text file itself located in the same directory as the code that reads it. If you make this mistake, Python will simply claim to not recognize the filename you give it.

[7]The words "directory" and "folder" are exact synonyms, and mean just what you think they mean. They are named containers which can contain files and/or other directories

That doesn't mean your file doesn't exist! It's just not in the right place.

An example of doing this *correctly* is in Figure 8.1. We're in a directory called "`filePractice`" (stare at the middle of the figure until you find those words) which is contained within the **home directory** that's denoted by a little house icon. Your home directory is just the starting point of your own private little CoCalc world. The slash mark between the house and the word `filePractice` indicates that `filePractice` is contained directly within, or "under," the home directory.

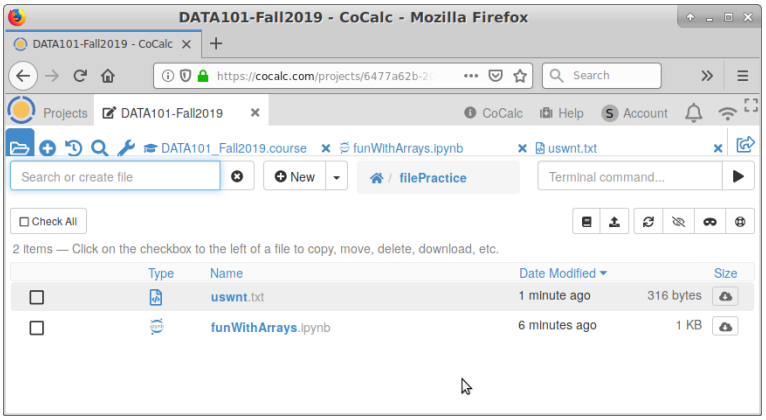


Figure 8.1: A directory (folder) on CoCalc, which contains two files: a plain-text file (called `uswnt.txt`) and a Jupyter Notebook which will read from it (`funWithArrays.ipynb`).

The two entries listed are a plain-text file (called `uswnt.txt`) and a Jupyter Notebook (`funWithArrays.ipynb`). You can tell that the former is a plain-text file because of the **filename extension** "`.txt`".[8] If we clicked on `uswnt.txt`, we'll bring up the contents of the file, as shown in Figure 8.2. In this case, we have the current

[8]Some operating systems like Windows, unfortunately, tend to "hide" the extension of the filenames it presents to users. You may think you have a file called "`nfl2024`" when you actually have one called "`nfl2024.txt`" or "`nfl2024.csv`," and Windows thinks it's being helpful (?!) by simply not showing you the part after the period. There are ways to tell Windows you're smarter than that, and that you want to see extensions, but these change with every version of Windows so I'll leave you to Google to figure that one out.

roster on the US Women's National Soccer team, one name per line. Perhaps the most important thing to see is that the file itself, which we will read into Python in a moment, is nothing strange or scary: you could type it yourself into Notepad or Word.[9]

This is a good time to mention that *spaces and other funny characters in filenames are considered evil.* You might think it looks better to call the notebook file "`fun with arrays.ipynb`" and the data file "`US Women's National Team roster.txt`", but I promise you it will lead to pain in the end, for a variety of fiddly reasons. It's better to use **camel case** for filenames, which is simply capitalizingEachSuccessiveWordInAPhrase.

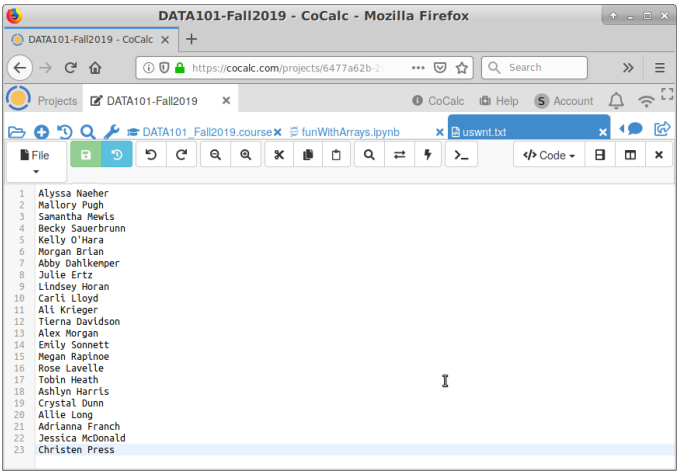


Figure 8.2: The contents of a plain-text file, as rendered by CoCalc.

Okay, finally back to NumPy code. If all the stars are aligned, we can write this code in a `funWithArrays.ipynb` cell to read the soccer roster into an `ndarray`:

[9] If you do ever create a plain-text file using Microsoft Word or similar word processing program, be sure to choose "Save as..." and save the file in **plain-text mode**. If you don't, Word will save a ton of extraneous formatting information (page settings, fonts, italics, and so forth) which will utterly pollute the raw information and make it impossible to read into Python.

```
roster = np.loadtxt("uswnt.txt", dtype=object, delimiter="###")
```

There's a lot of weird stuff in that line, so follow me here. The first argument is easy enough: the name of the file that contains our data. (Again, I stress that the file must be located *in the same directory* as the notebook!) The second argument is bizarre: we know what `dtype` means, but "`object`"? Ugh, another fiddly detail. When you read from a file into a NumPy array, you will be reading one of our three atomic types. Here are the rules:

If you want to read an array of...	...then set `dtype` to:
`int`s	`int`
`float`s	`float`
`str`s	`object`

So basically, you set `dtype` to the type of data you want in your `ndarray`...unless you want strings, in which case you put the word `object`. Sorry about that.

The last of the three arguments is even nuttier, and you actually don't need to include it at *all* if you're reading `int`s or `float`s. If you're reading `str`s, however, you need to set the **delimiter** to something *that doesn't appear in any of the* `str`*s*. I chose three-hashtags-in-a-row since that rarely appears in any set of text data.

Bottom line: once we've done all this, we get:

```
print(roster)
```

```
['Alyssa Naeher' 'Mallory Pugh' 'Sam Mewis' 'Becky Sauerbrunn'
 "Kelly O'Hara" 'Morgan Brian' 'Abby Dahlkemper' 'Julie Ertz'
 'Lindsey Horan' 'Carli Lloyd' 'Ali Krieger' 'Tierna Davidson'
 'Alex Morgan' 'Emily Sonnett' 'Megan Rapinoe' 'Rose Lavelle'
 'Tobin Heath' 'Ashlyn Harris' 'Crystal Dunn' 'Allie Long'
 'Adrianna Franch' 'Jessica McDonald' 'Christen Press']
```

which is pretty cool.

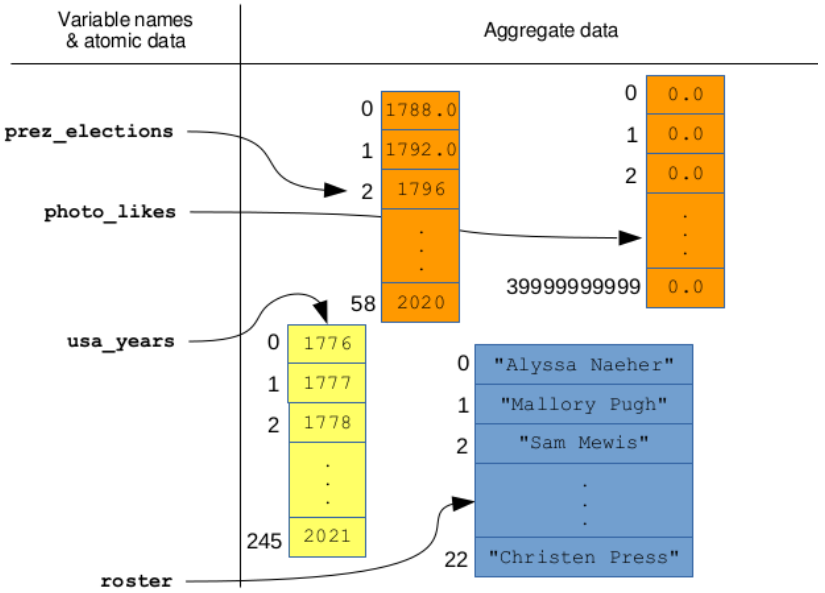


Figure 8.3: The memory picture of the four arrays we created in section 8.2.

8.3 Arrays in memory pictures

Before we leave this chapter and move on to actually *using* the `ndarrays` we've created, let me once again emphasize the memory picture and where arrays live in it. The four arrays we created in the previous section are depicted in Figure 8.3 on the following page. In each case, the variable name appears in the left half, with a pointer to the array itself which lives in the right half. Each of the four arrays starts at index 0, of course, and is numbered up to its length minus 1.

Learn to love these pictures!

Chapter 9

Arrays in Python (2 of 2)

Now that we know several options for how to *create* `ndarrays`, what can we do with them? Many and sundry things.

9.1 Getting the array size

To learn how long an array is (*i.e.*, how many elements) we use the `len()` function, kind of like we did for strings. Refer back to Figure 8.3 (p. 72) and consider this code:

```
num_players = len(roster)
sam_len = len(roster[2])
big_number = len(photo_likes)
print("There are {} players on the USWNT.".format(num_players))
print("Sam Mewis has {} characters in her name.".format(sam_len))
print(big_number)
print("We've had {} elections.".format(len(prez_elections)))
```

```
There are 24 players on the USWNT.
Sam Mewis has 9 characters in her name.
40000000000
We've had 59 elections.
```

This is an example of Python **overloading** function names, which just means that the same name is used for two different functions. When you pass a string to `len()`, you get the number of characters; but when you pass an array to `len()`, you get the number of elements it has. (The `roster` array had way more than 24 *letters* in it, notice – but `len()` returned 24 since that was the number of strings.)

9.2 Accessing individual elements

Retrieving an element

To get the value of a specific element from an array, we use "boxie notation" with the index number:

```
print(prez_elections[0])
third_year = usa_years[2]
print("{} was the 3rd year of U.S.A.".format(third_year))
print("The highest-numbered player is {}".format(
    roster[len(roster)-1]))
```

```
1788.0
1778 was the 3rd year of U.S.A.
The highest-numbered player is Christen Press.
```

Remember, indices start at zero (not one) so that's why the first line has a `0` in it.

Now examine that last line, which is kind of tricky. Whenever you have boxies, you have to first evaluate the code *inside* them to get a number. That number is then the index number Python will look up in the array. In the last line above, the code *inside* the boxies is:

...`len(roster)-1`...

Breaking it down, we know that `len(roster)` is 24, which means `len(roster)-1` must be 23, and so `roster[len(roster)-1]` is

`Christen Press`. It's a common pattern to get the last element of an array.[1]

To test your understanding, figure out what the following code will print:

```
q = 2
r = np.array([45,17,39,99])
s = 1
print(r[q-s+1]+3)
```

The answer is at the end of the chapter.

Changing an element

To modify an element of an array, just use the equals sign like we do for variables:

```
stooges = np.array(['Larry','Beavis','Moe'])
print(stooges)
stooges[1] = 'Curly'
print(stooges)
```

```
['Larry' 'Beavis' 'Moe']
['Larry' 'Curly' 'Moe']
```

After all, an individual element like `stooges[1]` is itself a variable pretty much like any other.

[1] Fun fact: you can also use *negative* indices to mean "from the end of the array, rather than the beginning." So `roster[-1]` will also give you `Christen Press`, `roster[-2]` is `Jessica McDonald`, `roster[-5]` is `Crystal Dunn`, *etc.* (see p. 71 for the values). I find this a bit obscure, though, so I don't normally use this feature. (Negative indices also mean a completely different thing in the R language, which is another reason I eschew them in both R and Python.)

Slices

Sometimes, instead of getting just one element from an array, we want to get a whole chunk of them at a time. We're interested in the first ten elements, say, or the last twenty, or the fourteen elements starting at index 100. To do this sort of thing, we use a **slice**.

Suppose we had a list of states in alphabetical order, and wanted to snag a chunk of consecutive entries out of the middle – say, Arizona through Colorado. Consider this code:

```
states = np.array(["AL","AK","AZ","AR","CA","CO","CT",
    "DE","FL","GA","HI"])
print(states[2:6])
```

```
['AZ' 'AR' 'CA' 'CO']
```

The "`2:6`" in the boxies tells Python that we want a slice with elements 2 through 5 (not through 6, as you'd expect). This is the same behavior we saw for `np.arange()` (p. 67) where the range goes up to, but *not* including, the last value. Just get used to it.

We can also omit the number before the colon, which tells Python to start at the beginning of the array:

```
print(states[:5])
```

```
['AL' 'AK' 'AZ' 'AR' 'CA']
```

or omit the number after the colon, which says to go until the end:

```
print(states[8:])
```

```
['FL' 'GA' 'HI']
```

We can even include a second colon, after which a third number specifies a step size, or stride length. Consider:

```
print(states[2:9:3])
```

```
['AZ' 'CO' 'FL']
```

This tells Python: "start the slice at element #2, and go up to (but not including) element #9, ***by threes***." If you count out the states by hand, you'll see that Arizona is at index 2, Colorado is at index 5, and Florida is at index 8. Hence these are the three elements included in the slice.

This slice stuff may seem esoteric, but it comes up surprisingly often.

9.3 "Vectorized" arithmetic operators

Recall our table of Python math operators (Figure 5.1 on p. 32). What do those things do if we use them on aggregate, instead of atomic data? The answer is: something super cool and useful.

Operating on an array and a single value

Consider the following code:

```
num_likes_today = np.array([6,61,0,0,14])
num_likes_tomorrow = num_likes_today + 3
print(num_likes_tomorrow)
```

```
[ 9 64 3 3 17 ]
```

See what happened? "Adding 3" to the *array* means adding 3 *to each element*. All in one compact line of code, we can do five – or even five billion – operations. This works for all the other Figure 5.1 operators as well.

For somewhat geeky reasons, this sort of thing is called a **vectorized** operation. All you need to know is that this means **fast**. And that's "fast" in two different ways: fast to write the code (since instead of using a **loop**, which we'll cover in 14, you just write a single statement with + and = signs), and more importantly, fast to *execute*. For more geeky reasons, the above code will run lightning fast even if `num_likes_today` had five hundred million elements instead of just five. As you'll learn if you ever try it, a Python loop is much slower.[2]

Don't get me wrong: there are times we'll have to use a loop because we have no choice. But the general rule with Python is: if you can figure out how to perform a calculation without using a loop, always do it!

Operating on two arrays

Possibly even cooler, we can even "+" (or "-", or "*", or...) two entire *arrays* together. Example:

```
salaries = np.array([38000, 102000, 55750, 29500, 250000])
raises = np.array([1000, 4000, 2000, 1000, 2000])
salaries = salaries + raises
print(salaries)
```

This code produces:

```
[ 39000 106000 57750 30500 252000]
```

[2] I just ran that comparison on my laptop, and here are the results. Using the plain-ol' "+" vectorized operator, my machine added the number 3 to an array with five hundred million elements in just 1.51 seconds. A loop, by contrast, took 2.8 *minutes* to do the same thing.

Can you see why? "Adding" the two arrays together performed addition *element-by-element*. The result is a new array with 38000+ 1000 as the first element, 102000 + 4000 as the second, *etc.* This, too, is a lightning-fast, vectorized operation, and it too works with all the other math operators.

Just to re-emphasize one point before we go on. In the example back on p. 77, we assigned the result of the operation to a new variable, `num_likes_tomorrow`. This means that `num_likes_today` itself was *unchanged* by the code. In contrast, in the example we just did, we assigned the result of the operation back into an *existing* variable (`salaries`). So `salaries` has itself been updated as a result of that code.

9.4 Copying – and *not* copying – arrays

Now, a surprise for the unwary. Suppose I write this code:

```
stooges = np.array(['Larry','Beavis','Moe'])
funny_people = stooges
stooges[1] = 'Curly'
print("The stooges are: {}.".format(stooges))
print("The funny people are: {}.".format(funny_people))
```

Take a moment and predict what you think the output will be. Then, read it and (possibly) weep:

```
The stooges are: ['Larry' 'Curly' 'Moe'].
The funny people are: ['Larry' 'Curly' 'Moe'].
```

Note carefully: *no* ***Beavis***.

Now the question is why. To understand this (and virtually any other tricky programming problem) you have to return once again to the memory picture. Figure 9.1 shows the situation immediately before, and after, the line "`stooges[1] = 'Curly'`" executes. Crucially, *there is only one array* in memory. Both variables – `stooges` and `funny_people` – are pointing at it.

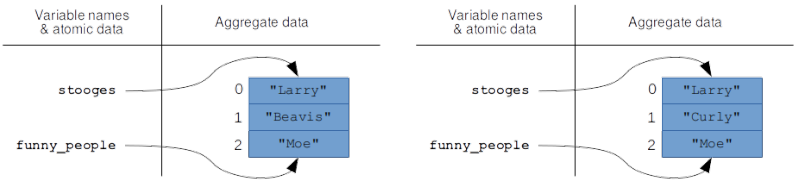


Figure 9.1: The code on p. 79 immediately before (left side) and after (right side) the line "`stooges[1] = 'Curly'`" is reached.

You see, if `y` contains *aggregate* (instead of atomic) data, the line "`x = y`" does not perform a copy operation. Instead, it just points the `x` variable name to the same place `y` is pointing to.

Once you grasp this, it's easy to see why `"Beavis"` completely disappeared. There's only one array at all, so changing `stooges` has the side effect of implicitly changing `funny_people` as well.

Actually copying

The "point the variable to the same thing, but don't do a copy" behavior is the default, because such copy operations are expensive (in terms of memory usage and time to execute). They're normally not what you want anyway. Sometimes, however, you *do* want to produce an entire separate copy of an array, so you can modify the copy yet preserve the original. To do so, you use the `.copy()` method:

```
orig_beatles = np.array(['John', 'Paul', 'George', 'Pete'])
beatles = orig_beatles.copy()
beatles[3] = 'Ringo'
print("The Beatles were originally {}.".format(orig_beatles))
print("But the ones we all know were {}.".format(beatles))
```

Look carefully at that second line: it makes all the difference. Instead of making the new variable `beatles` point to the same array in memory that `orig_beatles` did, we explicitly copied the array and made `beatles` point to that new copy. The final memory picture is thus as per Figure 9.2, and the output is of course:

```
The Beatles were originally ['John' 'Paul' 'George' 'Pete'].
But the ones we all know were ['John' 'Paul' 'George' 'Ringo'].
```

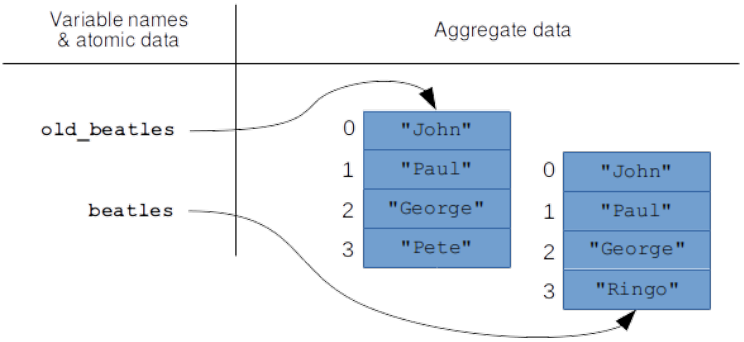


Figure 9.2: The memory picture after calling the `.copy()` method, instead of simply assigning to a new variable.

9.5 Sorting arrays

A common operation in Data Science is to **sort** an array, either numerically (if the array contains `int`s or `float`s) or alphabetically (if strings). There are two ways to do this, which turn out to differ in the same way as the operations in the previous section.

One way is to call the `.sort()` method directly **on** an array. This sorts the array **in place**, which means that the actual data in memory is rearranged right then and there. As an important side effect, any *other* variable that points to the same array will *also* be sorted.

Here's an example:

```
gpas = np.array([2.86, 3.99, 3.12, 1.17])
gpas2 = gpas.copy()
gpas3 = gpas
gpas.sort()
print("gpas has: {}".format(gpas))
print("gpas2 has: {}".format(gpas2))
print("gpas3 has: {}".format(gpas3))
```

```
gpas has: [1.17 2.86 3.12 3.99]
gpas2 has: [2.86 3.99 3.12 1.17]
gpas3 has: [1.17 2.86 3.12 3.99]
```

Do you see why that output was produced? It's because the memory picture after the "`gpas.sort()`" line looks like Figure 9.3. The `gpas` variable really *is* the `gpas3` variable, so when one is sorted, the other automatically is. They're both distinct from `gpas2`, though.

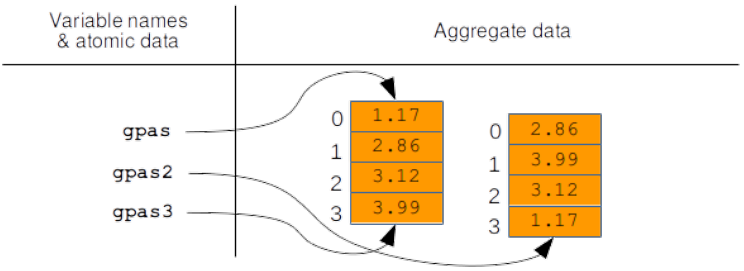


Figure 9.3: The state of affairs after `.sort()`ing the `gpas` array in place.

The second option is to call the `np.sort()` function and pass an array as an object. Like many Python functions, including the ones in the next section, `np.sort()` *returns a modified copy* of its argument rather than changing it in place. To illustrate:

```
nfl_teams = np.array(["Ravens", "Patriots", "Broncos",
    "Chargers", "Steelers"])
sorted_teams = np.sort(nfl_teams)
print(nfl_teams)
print(sorted_teams)
```

```
['Ravens' 'Patriots' 'Broncos' 'Chargers' 'Steelers']
['Broncos' 'Chargers' 'Patriots' 'Ravens' 'Steelers']
```

Observe that the `nfl_teams` variable, even though we passed it to `np.sort()`, was not *itself* sorted. The `sorted_teams` variable, on

the other hand, *is* alphabetically sorted, because we assigned the return value from `np.sort()` to it. Again, the memory picture is shown in Figure 9.4.

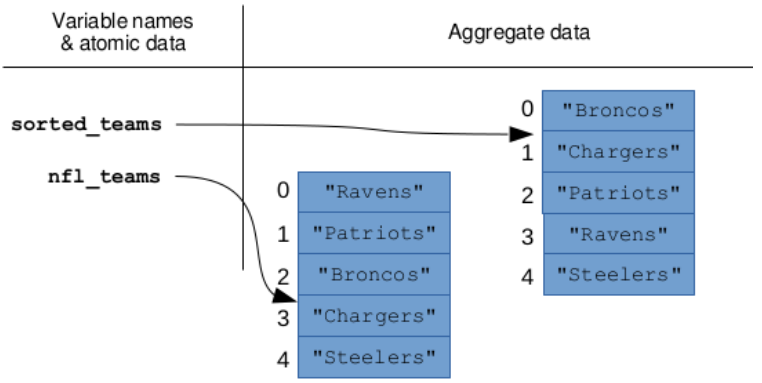


Figure 9.4: Calling the `np.sort()` function (as opposed to calling the `.sort()` method **on** the array) returns a sorted copy.

To be clear, either one of these techniques can be used on *any* `ndarray`: whole numbers, real numbers, or text. I just chose to do real numbers in the first example and text in the second. The difference between the two is merely in what is affected: in one, the actual array in memory is modified, and in the other, a modified copy is returned.

9.6 More exotic array modifications

There are lots of additional things you can do to an array to either modify its structure or rearrange its contents. Here's a few. **Important: all of the functions in this section return a modified copy of the array you pass to it. They do *not* change the array in place.**

- `np.append()` can be used to add a single element to the end of an array, or to add an entire second array of elements to it. (In the latter case, this is really **concatenation** of arrays.)

- `np.insert()` is like the first form of `np.append()`, except it inserts in the middle (or the beginning).
- `np.delete()` will remove an element of an array *by position*. In other words, you tell it which *index number* to remove, not which element.
- `np.flip()` reverses the order of elements in an array.

These functions are all summarized in Figure 9.5.

Remember that when you're calling a function like this – which returns a modified copy – it is perfectly acceptable to store the return value *in the same variable* that you passed it. This is common if you don't actually want to keep around the original:

```
ice_cream_flavors = np.flip(ice_cream_flavors)
```

In this pattern, the net effect *is* effectively to modify the array in place, since you're making a reversed copy, and then assigning that reversed copy to the same variable.

Anyway, here's some example code to illustrate the functions in this section:

```
clowns = np.array(["Bozo", "Krusty"])
more_clowns = np.array(["Pennywise", "Skelton"])
more_clowns = np.insert(more_clowns, 1, "Happy Slappy")
all_clowns = np.append(clowns, more_clowns)
all_clowns = np.append(all_clowns, "Ronald McDonald")
all_clowns = np.flip(all_clowns)
all_clowns = np.delete(all_clowns, 2)
print("clowns is: {}".format(clowns))
print("more_clowns is: {}".format(more_clowns))
print("all_clowns is: {}".format(all_clowns))
```

```
clowns is: ['Bozo' 'Krusty']
more_clowns is: ['Pennywise' 'Happy Slappy' 'Skelton']
all_clowns is: ['Ronald' 'Skelton' 'Pennywise' 'Krusty' 'Bozo']
```

An *excellent* exercise to help cement your understanding of the ideas in this chapter would be to go through the above "clowns" code line by line, drawing the memory picture as you go, and then confirm that your output matches the actual output.

Yes, an *excellent* exercise indeed!

9.7 Postlude: characters within a string

These two chapters have dealt with arrays, but let me say a word at this point about *strings*, and how they can be made to act like arrays in some respects.

I mentioned earlier in the book (p. 15) that strings, though normally treated as atomic, sometimes tiptoe up to the "atomic/aggregate" line and even cross it. In other words, we will occasionally look at individual parts of a string variable rather than the entire thing as one lump.

The way we access individual characters within a string is actually the same boxie notation we use for arrays. So this code:

```
antihero = "Light Yagami"
print(antihero[0])
letter1 = antihero[6]
letter2 = antihero[7]
print("{}{}{}".format(letter1,letter2,letter1))
```

will give this output:

```
L
YaY
```

As you can see, string indexes use the same starting-at-zero nonsense that arrays do. Hey, at least it's consistent.

This is actually another example of overloading. Just as the `len()` function means two different things, depending on whether you're asking for the length of an array or the length of a string (recall

p. 73), so the boxie notation means two different things. You're either getting a specific element out of an array, or a specific character out of a string.

9.8 Summary

The table in Figure 9.5 gives the promised summary of the array functions, methods, and operators in this chapter.

Function	Description
`len(`*arr*`)`	Get the number of elements in the array *arr*.
arr`[17]`	Get a specific element's value from the array *arr*.
arr`[8] =` (*something*)	Set a specific element of the array *arr*.
arr `+ 91`	Add a value to each element of *arr*, yielding a new array. (Also works with `-`, `*`, `/`, *etc.*)
arr1 `+` *arr2*	Add each pair of values in two arrays, yielding a new array. (Also works with `-`, `*`, `/`, *etc.*)
arr1 = *arr2*	Make *arr1* point to the same data that *arr2* points to. (*Not* a copy!)
arr1 = *arr2*`.copy()`	Make *arr1* point to a new, independent copy of *arr2*.
arr`.sort()`	Sort the array *arr* **in place**. (Numerical or alphabetical, depending on the `.dtype`.)
`np.sort(`*arr*`)`	Return a new array with the sorted elements of *arr*. (Numerical or alphabetical, depending on the `.dtype`.)
`np.append(`*arr*, *elem*`)`	Return a new array with *elem* tacked on to the end.
`np.append(`*arr1*, *arr2*`)`	Return a new array with the two arrays *arr1* and *arr2* concatenated.
`np.insert(`*arr*, *ind*, *val*`)`	Return a new array with the new value *val* inserted into position *ind* of *arr*.
`np.delete(`*arr*, *ind*`)`	Return a new array with the element at index *ind* removed from *arr*.
`np.flip(`*arr*`)`	Return a new array with *arr* in reverse order.

Figure 9.5: Handy NumPy functions, methods, and operators.

The answer

Oh, and the answer to the puzzle on p. 75 – and also the answer to Life, the Universe, and Everything, as it turns out – is 42.

Chapter 10

Interpreting Data

Let's take an intermission from the nitty-gritty Python stuff and talk about how to properly *interpret* the data we're working with; specifically, how to draw correct conclusions from what we've collected.

10.1 Independent and dependent variables

You've undoubtedly seen countless studies that claim to reveal important truths about the world, such as that smoking can cause lung cancer, greenhouse gas emissions can cause higher global temperatures, or orgasms can cure hiccups. Much of the time, scientists try to find a **causal** factor that links one variable to another: they suspect that the value of a variable A (often called the **independent variable**, or "**i.v.**" for short) is a *reason*, or **cause**, of a certain value in another variable B (the **dependent variable**, or "**d.v.**").

Just to avoid misunderstandings, when we claim that A **causes** B, we don't normally mean that it *exclusively* causes it, or even that it *reliably* causes it. There are lots of contributing factors to lung cancer besides smoking, after all; and tons of smokers never develop cancer. We simply mean that A is a contributing factor to B, and that the value of the A variable exerts some, but not total, influence over the value of the B variable.

Importantly, we're using the word **variable** here in a different, but

related way than we used it in chapters 3, 8, and 9. As we did in chapter 6 (see p. 43 footnote), we use "variable" here to mean a specific aspect of the **objects of a study** that can differ, or "vary." The objects in our study (often people, but sometimes companies, organizations, environments, nations, *etc.*) *each* have a value for the variable. Thus if you think of a "per-capita income" variable, you might think of an entire *array* of floats, each of which represented the average income-per-resident of a single nation.

The variables in question can be from any of the scales of measure from chapter 6. Take the smoking example, with patients as the object of study. We might say that independent variable A is categorical, with values `SMOKER` and `NON-SMOKER`. The dependent variable B is also categorical: `CANCER` and `NO-CANCER`. The key question is: do people with A = `SMOKER` also have B = `CANCER` *more often* (a higher percentage of the time) than people with A = `NON-SMOKER` do?

In the greenhouse gas emissions example, our objects of study might be *years*. Our variables are both numeric, with A (a measure of yearly greenhouse gas emissions, measured in gigatonnes CO_2) on the ratio scale, and B (average worldwide temperature increase/decrease) on an interval scale. Here, the question would be: do years in which A is relatively high typically also have B relatively high? Put another way: do years in which earthlings have released more gas into the atmosphere tend to correspond with years in which the global temperature increased?

And of course, we might have one categorical variable and one numeric. Perhaps our objects of study are American adults, and while our categorical A variable has values `DEMOCRAT`, `REPUBLICAN`, `OTHER`, and `INDEPENDENT`, our numerical B is yearly income. Our question would be: do adherents of one political party tend to be more wealthy than those of another?

Or, flipping sides, the independent variable A could be numeric while the dependent variable B is categorical. Our objects of study might be high school seniors applying to UMW. Let A be the number of different colleges a student applied to, and B a categorical variable with values `ADMITTED-TO-UMW` and `NOT-ADMITTED-TO-UMW`.

The question of interest is here is: do students who apply to more colleges tend to get in to UMW more often?

10.2 Association and causality

All of the above questions can be answered with data. In future chapters, we'll learn the exact Python commands to ask them, and how to interpret the answers.

For now, I merely want to draw your attention to the fact that these are all questions of **association**, not causation. An association between variables merely means that they are **correlated** in some way statistically.[1] If A = `SMOKER` goes with B = `CANCER` more often than A = `NON-SMOKER` does, then there *is* an association between the two, period. If yearly income B is on average higher for A = `REPUBLICAN` than for A = `DEMOCRAT`, then there *is* an association between the two, period.

(By the way, a key nuance will turn out to be: *how much* more often does A = `SMOKER` need to go with B = `CANCER` in order for us to be confident that there is a true association? Or *how much* more wealthy do the A = `REPUBLICAN`s need to be on average for us to have confidence we've identified a real link to political party? That one's a little tricky, and we'll postpone addressing it for now.)

So anyway, the question of association turns out to be pretty straightforward to answer. Python will simply tell us if variables are associated or not. More difficult, however, is determining **causality** (a.k.a. **causation**). Does a person's political affiliation influence how much wealth they have? Or is it the other way around: does a person's wealth cause them to vote a certain way? Or is it neither of these, with some third factor (perhaps values, or life philosophy) helping determine *both* variables?

If the first of these three is the case, we would write "$A \rightarrow B$," pronounced "A causes B". If the second, we'd write, "$B \rightarrow A$," and

[1] Another way to put this is to say that the variables are **dependent** on each other, although this is confusing because we're already using the word "dependent" to refer to one of the variables.

for the third, we'd write "$C \to A, B$" for some other (possibly yet to be determined) variable C. Determining which (if any) of these is true calls for some careful thinking, intuition, and additional kinds of statistical tests.

In fact, just to blow your mind, Figure 10.1 gives a partial list of the various types of causation that *could* be the true explanation, once we find out that A and B have an association. As you can see, there are a lot of ways to go wrong. Only *one* of the possibilities is that "A actually causes B," which is what we suspected in the first place. The others are all ways of producing that same association we picked up in the data.

10.3 Confounding factors

Let me speak to two of the items in the Figure 10.1 table in particular. The third one on the list, **external causation**, is a case where a third variable (call it C) comes into play. We refer to this as a **confounding factor** (or **confounding variable**) because it "confounds" us: causes us to interpret the meaning behind the data in an incorrect way. The example in the table is a famous and funny one: clearly sharks don't react to Ben & Jerry's daily net profits, and people (probably) don't run out and buy ice cream to cope with their anxiety about shark attacks. Neither $A \to B$ nor $B \to A$, but a third variable – hot days – influence both of them.

Now of course it's not always this obvious. Here's an example I ran across recently. A magazine article reported on a new health scare: scientists have discovered that *eating barbecue can increase your risk of cancer.* Pictorially, this claim is illustrated in the **causal diagram** in Figure 10.2 (flip to p. 94), which shows our i.v. and our d.v. ; the arrow means exactly what it meant earlier.

Unlike sharks and ice cream, this one seems plausible. And I'm not claiming to have read enough about their study to tell whether the researchers' claim is bogus. But I couldn't help thinking that there are a great many possibly confounding factors that could be blurring the results. For one, choosing to eat barbecue a lot is probably often associated with a less healthy, higher-fat diet in general (I can speak from experience on that). If that's true, and if high-fat diets

Symbology	Name	Example
$A \to B$	causation	Regular exercise does indeed normally lead to a lower resting heart rate.
$B \to A$	reverse causation	Smoking doesn't cause depression; depression causes smoking.
$C \to A, B$	external causation (confounding factor)	Ice cream sales don't cause shark attacks; high temperatures boost both ice cream sales and ocean swimming.
$A \to B$ & $C \to B$	multiple causation	A liberal arts education does improve critical thinking skills, but lots of other things do too.
$A, C \to B$	joint causation	Just being tall doesn't necessarily make you a good basketball player, but if your height is accompanied by another factor as well (athleticism), then you will be.
$A \to C \to B$	indirect causation	People who use antiperspirant tend to get more dates, but it's not because of the antiperspirant *per se*; it's because they don't have an unpleasant odor.
$A \not\to B$	spurious association	Although for many years the outcome of the Washington football game immediately preceding a Presidential election predicted the election's outcome, that was by coincidence.

Figure 10.1: Various types of causality that could be the underlying reason why an association between A and B exists.

– whether featuring lots of barbecue or not – are associated with these same poor health outcomes, then we'd have the picture on the left-side of Figure 10.3 (also on p. 94). The red bubble represents the confounding factor, which is influencing both i.v. and d.v. If this picture were the correct one of the underlying phenomenon, then the correlation we thought were picking up between barbecue and cancer was actually due to fat content.

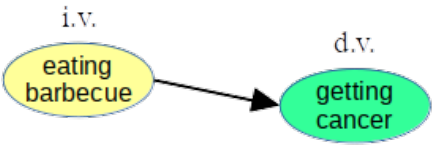


Figure 10.2: A hypothesis as to causality: eating barbecued foods increases one's risk for certain types of cancer.

Another example is the right-hand side of Figure 10.3, below. Perhaps barbecue is more popular culturally in some areas of the country (say, the South, where I certainly see it eaten a lot), and perhaps those areas have other environmental factors that can lead to cancer. In this case, the "South" confounder indirectly affects the d.v. (via another variable, the environment) but it still affects it.

It's not hard to think of others. These were just the first two that came to mind. The point is that it's really hard to be sure you've thought of all of them!

Paranoia and overparanoia

All this should lead you to be somewhat paranoid, but not *over*-paranoid. Confounding variables can definitely lead us to make mistakes in our reasoning, but perhaps they're not *quite* as common as you think. Understand that a confounding factor is *not* simply any other factor that affects the dependent variable. Instead, for a variable to be confounding **it must affect both the independent *and* the dependent variable.**

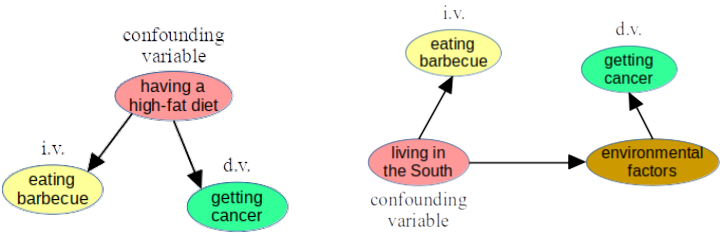


Figure 10.3: Other hypotheses as to causality, each resulting in the same associations in the data, yet involving confounding factors.

Let me illustrate with an example. I suspect that on average, men are taller than women. And I further suspect that there's causality here, and that it goes from A (sex) to B (height), not the other way around. (Clearly people don't spontaneously "turn male" because they reach a certain height.) So my thinking on the subject is summed up in Figure 10.4.

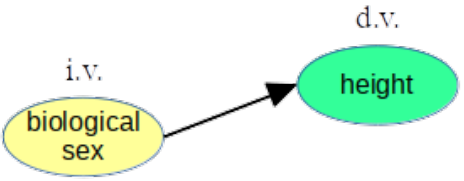


Figure 10.4: Stephen's hypothesis: a person's biological sex (male or female) plays a causal role in determining their height.

Now let me show you what I mean by "overparanoia." What if someone said, "but wait, Stephen, not so fast! You've got potential confounding variables out the wazoo! Why, surely heredity plays some role in a person's height – tall parents are more likely to have tall offspring, just due to genetics. And nutrition, too, is a factor: it's been demonstrated that impoverished communities suffering from malnutrition will have children with stunted growth. And heck, if you're born at a high elevation (like Nepal), there's less gravitational pull dragging your body down to earth, so it stands to reason that you'll probably grow taller. And on and on!" Figure 10.5 depicts this (supposed) scientific nightmare.

But plausible as some of those theories are, they are *not* confounding variables! These are simply *other factors that may affect the d.v.* Sure, they may also play a causal role in determining a person's height, but they do *not* invalidate our finding about sex and height.

For them to truly be confounders, they would have to affect the yellow *and* the green variable, and I'm pretty sure they do not. Do tall parents tend to bear more sons, and short parents more daughters? If not, this isn't a confounder. Do boys have more nutritious diets than girls? (In some parts of the world, that may

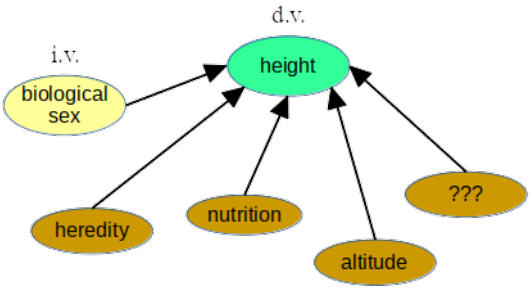


Figure 10.5: Oh geez – confounding variables galore? ***No!***

unfortunately be true, but I don't believe it is in our country.) So that one isn't a confounder either. Having additional causes of an effect does not nullify a genuine effect. Only a lurking variable that pulls the marionette strings of both i.v. and d.v. can do that.

10.4 Dealing with confounding factors

Confounding factors are evil, and we must deal with them seriously. There are essentially two ways to do that: one that requires us to be smart, and one that requires us to have money.

Controlling for a confounding factor

If we anticipate that a certain variable may be a confounding factor, we can **control** for it. There are several techniques for this, some of which you'll learn in your statistics class, but the simplest one to understand involves **stratification**.

Let's make a silly example this time. We'll go back to the earlier pinterest theme. I think I've noticed over the past few years that the heavy pinterest users I know seem to almost always have long hair. I've developed a hypothesis about this, which involves theories about how protein filaments in follicles with longer protrusions lead to certain chemical changes in the brain. These mind alterations, if unchecked, lead to increased creativity, craftiness, and a desire to share artistic creations with other like-minded individuals. Further,

these aesthetic desires manifest themselves in increased usage of the `pinterest.com` website, as measured in number of logins per day.

My theory is thus reflected in the causal diagram in Figure 10.6. Study it carefully.

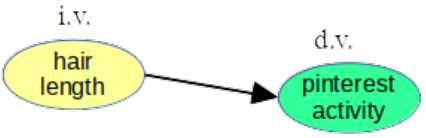


Figure 10.6: A theory about how hair length impacts the number of times a person logs on to pinterest each day.

Now of course this follicle stuff is bogus. I'm using an extreme example to make a point. Quick, can you come up with a possible confounding factor? Yeah, drr: *gender*. It's undoubtedly true that women tend to (but don't always) have longer hair than men, and it's undoubtedly true that `pinterest.com` is a website that tends to appeal to (but not exclusively to) women. And causality-wise, the arrows obviously flow from gender, not to it: the pinterest login screen doesn't change your gender, and a man won't turn into a woman simply by growing his hair long (although a transgender woman might grow her hair long as a signal of her underlying gender change.)

Put that all together, and you get the much more plausible causal diagram in Figure 10.7.

Now then. Controlling for a confounding variable through stratification is done by considering the objects of the study in *groups*, comparing *only those who have the same (or similar) value for the confounding variable.*

In this case, we would separate the men from the women in our study. Looking at *just the men*, we would ask, "is longer hair associated with frequency of pinterest logins?" Then we would do the same, looking at *just the women.* Only if Python reported that both of these separate groups illustrated such a trend would we

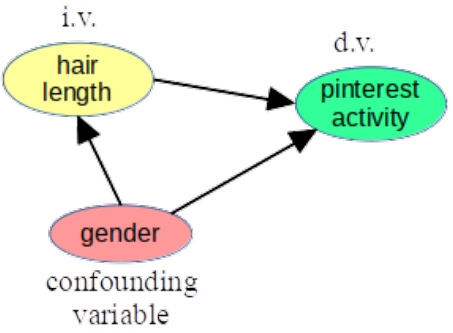


Figure 10.7: An alternative theory that holds that a person's gender influences both their long-haired-ness and their pinterest-ness.

(tentatively) conclude, "hair length itself does play a role in causing pinterest activity, *even when controlling for gender*."

Do the thought experiment to see if you agree. I know the whole follicle theory struck you as dumb (and hopefully, a little funny) to begin with. "Of *course*," you said to yourself, "it's gender, not hair length, that's drawing users to pinterest, dummy!" But suppose we *did* perform that stratification technique, and discovered that the association actually *did* hold in both cases. Would that give it more credence in your mind? It ought to. By stratifying, we've eliminated gender from the picture entirely, and now we're faced with the facts that those with longer hair – regardless of gender – log on to pinterest more often.

Now I wrote the word "tentatively" a couple paragraphs ago, because there are still some caveats. For one, we don't actually know that the causality goes in the stated direction. Removing the gender confounder, we confirmed that there is still an association between hair length and pinterest, but that association might translate into a $B \rightarrow A$ phenomenon. Perhaps users who log on to pinterest a lot see a lot of long-haired users, and (consciously or not) decide to grow their own hair out as a result? That actually sounds more plausible than the original silly theory. Either way, we can't confirm the direction just by stratifying.

The other caveat is even more important, because it's more pervasive: just because we got rid of one confounding variable doesn't mean there aren't others. The whole "control for a variable" approach requires us *to anticipate in advance* what the possible confounding factors would be. This is why I said back on p. 96 that this approach requires the experimenter to be smart.

Running a controlled experiment

The other way to deal with confounding variables is to run a **controlled experiment** instead of an **observational study**. I jokingly said on p. 96 that this option requires the experimenter to have money. Let me explain.

An observational study is one in which data is produced by naturally occurring processes (we'll call them **data-generating processes**, or **DGPs**) and then collected by the researcher. Crucially, the experimenter plays no role in influencing what any of the variable values are, whether that be the i.v. , the d.v. , other related variables, or even possible confounders. Everything just is what it is, and the researcher is simply observing.

Now at first this sounds like the best of all possible worlds. Scientists are supposed to be objective, and to do everything they can to avoid biasing the results, right? True, but the sad fact is that *every observational study has potential confounding factors* and there's simply no foolproof way to account for them all. If you *knew* them all, you could potentially account for them. But in general we don't know. It all hinges on our cleverness, which is a bit like rolling the dice.

A controlled experiment, on the other hand, is one in which *the researcher decides what the value of the i.v. will be for each object of study.* She normally does this randomly, which is why this technique is called **randomization**.

Now controlled experiments bear some good news and some bad news. First, the good news, which is incredibly good, actually: *a controlled experiment automatically eliminates every possible confounding factor, whether you thought of it or not.* Wow: magic!

We get this boon because of how the i.v. works. The researcher's coin flip is the *sole* determinant of who gets which i.v. value. That means that no other factor can be "upstream" of the coin flip and influence it in any way. And this in turn nullifies all possible confounding factors, since as you recall, a confounder must affect both the i.v. and the d.v.

The catch is that controlled experiments can be very expensive to run, and in many cases can't be run at *all*. Consider the barbecue example from p. 92. To carry out a controlled experiment, we would have to:

1. Recruit participants to our study, and get their informed consent.
2. Pay them some $$ for their trouble.
3. For each participant, flip a coin. If it comes up heads, *that person must eat barbecue three times per week for the next ten years.* If it's tails, *that person must never eat barbecue for the next ten years.*
4. At the end of the ten years, measure how many barbecuers and non-barbecuers have cancer.

There's a question of this even being ethical: if we suspect that eating barbecue can cause cancer, is it okay to "force" participants to eat it? Even past that point, however, there's the expense. Ask yourself: if you were a potential participant in this experiment, how much money would you demand in step 2 to change your lifestyle to this degree? You might love barbecue, or you might hate it, but either way, it's a coin flip that makes your decision for you. That's a costly and intrusive change to make.

Other scenarios are even worse, because they're downright impossible. We can't flip coins and make (at random) half of our experimental subjects male and the other half female. We can't (or at least, shouldn't) randomly decide our participants' political affiliations, making one random half be Democrats and the others Republicans. And we certainly can't dictate to the nations of the world to emit large quantities of greenhouse gases in some years and small quantities in others, depending on our coin flip for that year.

Bottom line: if you can afford to gather data from a controlled experiment rather than an observational study, always choose to do so. Unfortunately, it won't always be possible, and we'll have to rest on the uneasy assumption that we successfully predicted in advance all the important confounding variables and controlled for them.

10.5 Spurious associations

Okay, back to Figure 10.1 on p. 93. The other item I'd like to point out in that table is the last one, which is called a **spurious association**. This is written as "$A \not\rightarrow B$," with the arrow crossed out. And it simply means "nope, none of the above: these variables actually aren't associated at all."

You might be scratching your head at that one. Didn't I tell you (p. 91) that Python is smart enough to tell us definitively whether or not two variables are associated? That was supposed to be the easy part; the hard part was only in figuring out what *causes* that association. But now I'm saying that associations might not be associations, and Python is powerless to know the difference!

The root cause of this state of affairs is obvious once you see it, and it has to do with the "how much more?" questions from p. 91. Clearly, when we collect data, there's a "luck of the draw" component ever-present. I might have data that suggests Republican voters have higher income than Democratic voters...but it's of course possible that I just happened to poll some richer Republicans and some poorer Democrats. Suppose I told you I thought women were on average smarter than men, and in my random sample the average men's IQ was 102.7 and the average woman's was 103.5. The women's was indeed greater...but is that *enough* greater? Is the difference explainable simply by the randomness of my poll?

The true answer is that we can *never* know for absolute certainty, unless we can poll the entire population. (Only if we measured the IQ of every man and every woman on planet Earth, and took the means of both groups, could we say which one truly had the higher mean.) But what we have to do is essentially "set a bar" somewhere,

and then determine whether we got over it. We could say "only if the average IQ difference is greater than 5 points will we conclude that there's really a difference."

Setting α

Now the procedure for determining how high to put the "bar" is more complicated and more principled than that. We don't just pick a number that seems good to us. Instead, Python will put the bar at exactly the right height, given the level of certainty we decide to require. Some things that influence the placement of the bar include the sample size and how variable the data is. The thing *we* specify in the bar equation, however, is *how often we're willing to draw a false conclusion.*

That quantity is called "$\boldsymbol{\alpha}$" (pronounced "**alpha**") and is a small number between 0 and 1. Normally we'll set $\alpha = .05$, which means: "Python, please tell me whether the average male and female IQs were *different enough* for me to be confident that the difference was truly a male-vs-female thing, not just an idiosyncrasy of the people I chose for my poll. And by the way, *I'm willing to be wrong 5% of the time about that.*"

It seems weird at first – why would we accept drawing a faulty conclusion 5% of the time? Why not 0%? But you see, we have to put the bar somewhere. If we said, "I never want to think there's an association when there's not one," Python would respond, "well fine, if you're so worried about it then I'll never tell you there is one." There has to be some kind of criterion for judging whether a difference is "enough," and $\alpha = .05$, which is "being suckered only 1 in 20 times" is the most common value for social sciences. ($\alpha = .01$ is commonly used in the physical sciences.)

So, the last entry in the Figure 10.1 table means "even though the A and B variables aren't *really* associated at all – if we gathered some more As and some more Bs, we'd probably detect *no* association – you were fooled into thinking there was one because our random sample was a bit weird." There's really no way around this other than being aware it can happen, and possibly repeating our study with a different data set to be sure of our conclusions.

Chapter 11

Associative arrays in Python (1 of 3)

Our next trick is to represent associative arrays (review section 7.1 on p. 55 if you need to) in Python. To do so, we will use another package, which goes by the adorable name "Pandas":

```
import pandas as pd
```

This code should go at the top of your first notebook cell, right under your "`import numpy as np`" line. The two go hand in hand.

By the way, just as there were other choices besides NumPy `ndarrays` to represent ordinary arrays, there are other choices in Python for associative arrays. The native Python `dict` ("dictionary") is an obvious candidate. Because this won't work well when the data gets huge, however, and because using Pandas now will set up our usage of tables nicely in the next few chapters, we're going to use the Pandas **Series** data type for our associative arrays.

11.1 The Pandas `Series`

A `Series` is conceptually a set of key-value pairs. The keys are normally homogeneous, and so are the values, although the keys might be of a different type than the values. Any of the three atomic types are permissible for either.

Somewhat confusing is that the Pandas package calls the keys "the **index**," which is an overlap with the term we used for ordinary arrays (see p. 7.1). It's not a total loss, though, since if you think hard about it, you'll realize that in some sense, *a regular array is really just an associative array with consecutive integer keys.* Oooo, deep. If you study the two halves of Figure 11.1, I think you'll agree.

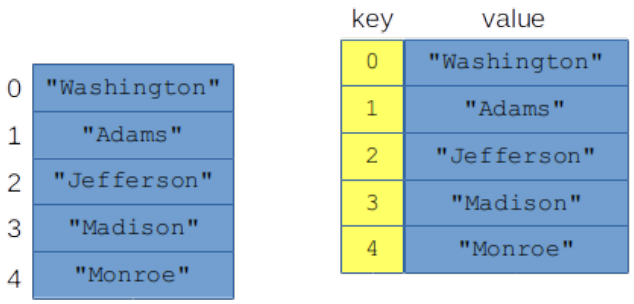


Figure 11.1: An ordinary array, and an associative array, that represent the same information.

Creating `Serieses`

Here are a few common ways of creating a Pandas `Series` object in memory.

Way 1: create an empty `Series`

Perhaps this first one sounds dumb, but we will indeed have occasion to start off with an empty `Series` and then add key/value pairs to it from there. The code is simple:

```
my_new_series = pd.Series()
```

Voilà.

Way 2: `pd.Series([], index=[])`

As with NumPy `ndarrays`, we can explicitly list the values we want in a new `Series`. We also have to list the **index** values (the keys). The syntax for doing so is:

```
alter_egos = pd.Series(['Hulk','Spidey','Iron Man','Thor'],
    index=['Bruce','Peter','Tony','Thor'])
```

This creates the `Series` shown in Figure 11.2.

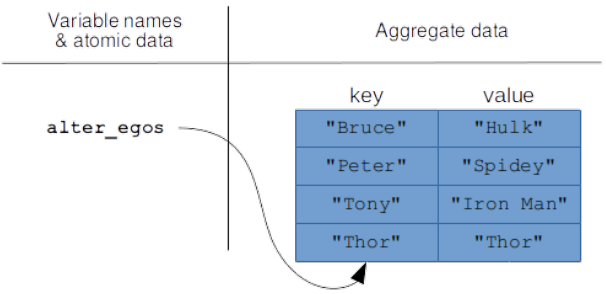


Figure 11.2: A Pandas `Series` in memory.

Be careful to keep all your boxies and bananas straight. Note that both the keys *and* the values are in their own sets of boxies.

We can print (smallish) `Series`es to the screen to inspect their contents:

```
print(alter_egos)
```

```
Bruce        Hulk
Peter      Spidey
Tony     Iron Man
Thor         Thor
dtype: object
```

Also, as we did on p. 63, we can inquire as to both the overarching type of `alter_egos` and also to the kind of underlying data it contains:

```
print(type(alter_egos))
print(alter_egos.dtype)
```

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

Just as it did on p. 71, the "`object`" here is just a confusing way of saying "`str`". Don't read anything more into it than that.

Way 3: "wrapping" an array

Associative arrays, and the Pandas `Series`es we've been using to implement them, are inherently *one*-dimensional data structures. This is just like the NumPy arrays we used before. Pandas `Series`es also provide a bunch of features for manipulating, querying, computing, and even graphing aspects of their content. It's a lot of rich stuff on top of plain-old NumPy.

For this reason, it's common to want to create a `Series` that just "wraps" (or encloses) an underlying NumPy `ndarray`, and provides all that rich stuff.

The way to do this is simple:

```
my_numpy_array = np.array(['Ghost','Pumpkin','Vampire','Witch'])
my_pandas_enhanced_thang = pd.Series(my_numpy_array)
```

You can then treat `my_pandas_enhanced_thang` as an ordinary aggregate variable which has the more sophisticated operations of next chapter automatically glommed on to it. The keys (index values) of this thang will simply be the integers 0 through 3.

Way 4: `pd.read_csv()`

Finally, there's reading data from a text flie, which as I mentioned back in section 8.2 (p.68) is actually the most common. Data typically resides in sources and files external to our programming environment, and we want to do everything we can to play ball with this open universe.

One common data format is called **CSV**, which stands for **comma-separated values**. Files in this format are normally named with a "`.csv`" extension. As the name suggests, the lines in such a file consist of values separated by commas. For example, suppose there's a file called `disney_rides.csv` whose contents looked like this:

```
Pirates of the Carribean,25
Small World,20
Peter Pan,29
```

These are the current expected wait time (in minutes) for each of these Disney World rides at some point of the day.

To read this into Python, we use the `pd.read_csv()` function. It's a bit awkward since it has several mandatory arguments if you want to deal with `Series`es. Here's how it works:

```
wait_times = pd.read_csv('disney_rides.csv', index_col=0,
    squeeze=True, header=None)
```

Most of that junk is just to memorize for now, not to fully understand. If you're curious, `index_col=0` tells Pandas that the first (0th) column – namely, the ride names – should be treated as the `index` for the `Series`. The `header=None` means "there is no separate header row at the top of the file, Pandas, so don't try to treat it like one." If our `.csv` file *did* have a summary row at the top, containing labels for the two columns, then we'd skip the `header=None` part. Finally, "`squeeze=True`" tells Pandas, "since this is so skinny

anyway – just two columns – let's have `pd.read_csv()` return us a `Series`, rather than a more complex `DataFrame` object (which is the subject of Chapter 16)."

Chapter 12

Associative arrays in Python (2 of 3)

K, now we can create `Series`es; let's figure out what we can do with them.

12.1 Accessing individual elements

We can use the `len()` function, which we've already learned two uses for, in yet a third way: to ascertain the number of key/value pairs in a series. Using the Figure 11.2 example (p. 105):

```
print(len(alter_egos))
```

```
4
```

Accessing the value for a given key uses exactly the same syntax that NumPy arrays used (boxies), except with the key in place of the numeric index:

```
superhero = alter_egos['Peter']
print("Pssst...Peter is really {}.".format(superhero))
```

```
Pssst...Peter is really Spidey.
```

This is why it's important that the *keys* of an associative array be unique. If we type "`alter_egos['Peter']`," we need to get back one well-defined answer, not an ambiguous set of alternatives.[1] The values, on the other hand, may very well not be unique.

To overwrite the value for a key with a new value, just treat it as a variable and go:

```
alter_egos['Bruce'] = 'Batman'
print(alter_egos)
```

```
Bruce      Batman
Peter      Spidey
Tony     Iron Man
Thor         Thor
dtype: object
```

This same syntax works for adding an entirely *new* key/value pair as well:

```
alter_egos['Diana'] = 'Wonder Woman'
print(alter_egos)
```

```
Bruce          Batman
Peter          Spidey
Tony         Iron Man
Thor             Thor
Diana    Wonder Woman
dtype: object
```

[1] Pandas, which tries to be All Things To All People™, will actually let you have duplicate index values in a `Series`. What does it do if you ask for "the" value of `Peter`, then, if there's more than one? It gives you back another ***`Series`*** of the different `Peter` superheroes. This is a major pain, because now when you look up a value in the `Series`, you don't know whether you'll get back a single item or another `Series`, which means you have to check to see which one it is, and then write different code to handle the two cases...yick. Just stay far, far away. Make all your keys unique.

It's just like with ordinary variables, if you think about it. Saying "`x=5`" overwrites the current value of `x` if there already *is* an `x`, otherwise it creates a new variable `x` with that value.

Finally, to outright remove a key/value pair, you use the `del` operator:

```
del alter_egos['Tony']
print(alter_egos)
```

```
Bruce           Batman
Peter           Spidey
Thor              Thor
Diana    Wonder Woman
dtype: object
```

Bye bye, Iron Man.

Don't get mad when I tell you that all of the above operations work **in place** on the `Series`, which is very different than some of the "return a modified copy" style we've seen recently. Hence all of these attempts are *wrong*:

```
alter_egos = del alter_egos['Tony']                 <--- WRONG!
alter_egos = alter_egos['Bruce'] = 'Batman'         <--- WRONG!
alter_egos = alter_egos['Diana'] = 'Wonder Woman'   <--- WRONG!
```

You don't "change a value and get a new `Series`"; you just "change it."

Accessing by position

One slightly weird thing you can do with a Pandas `Series` is ignore the key (index) altogether and instead use *the number of the key/value pair* to specify what value you want. This gives me the heebie-jeebies, because as I explained back on p. 57, there really

isn't any meaningful "order" to the key/value pairs of an associative array. In true All Things To All People™ fashion, however, Pandas lets you do this.

Accessing a value by position

You can ask for the value of (say) "the second" superhero. To do so, you use the bizarrely-named **.iloc syntax**:

```
a_hero = alter_egos.iloc[1]
print(a_hero)
```

```
Spidey
```

This is occasionally useful, so I mention it for completeness. The `.iloc` numbers start with 0 (not 1) as is true throughout Python.

Accessing a key by position

Similarly, you can get the *key* (as opposed to the value) of the key/value pair at a particular position. To ask for the key of "the second" superhero, you use the **.index syntax**:

```
a_secret_hero = alter_egos.index[1]
print(a_secret_hero)
```

```
Peter
```

12.2 Vectorized arithmetic operators

As with NumPy `ndarrays`, you can apply arithmetic operators like `+` and `*` to entire `Series`es at a time, which is not only easy code to write but also runs blazing fast. But the Pandas `Series` is even smarter than that.

Consider the memory picture in Figure 12.1. Here we have two `Series`es, one pointed to by a `salaries` variable and the other by

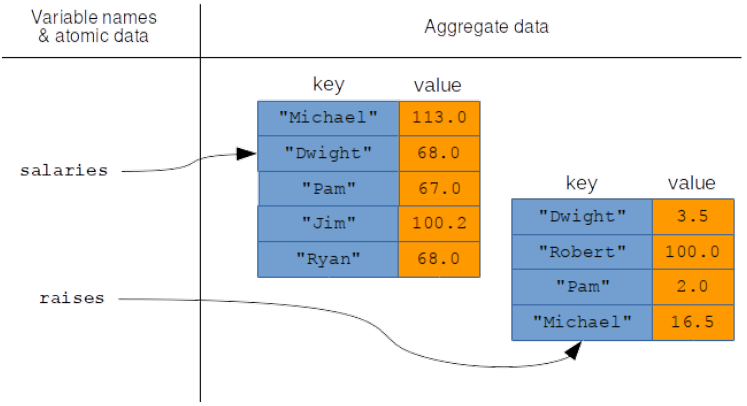


Figure 12.1: Two `Series`es in memory

`raises`, which are of different sizes and which have overlapping, but not identical, sets of keys. What do you suppose Pandas would do if we executed this code?

```
new_salaries = salaries + raises
```

The answer, happily, is the smartest possible thing it could do. Pandas gets neither confused nor stifled by the fact that the keys are in different orders in the two `Series`es, and instead it does what you surely want: add corresponding elements, with matching keys, and produce a new `Series` with all of those sums.

The actual result in this case is in Figure 12.2, and the output is here:

```
new_salaries = salaries + raises
print(new_salaries)
```

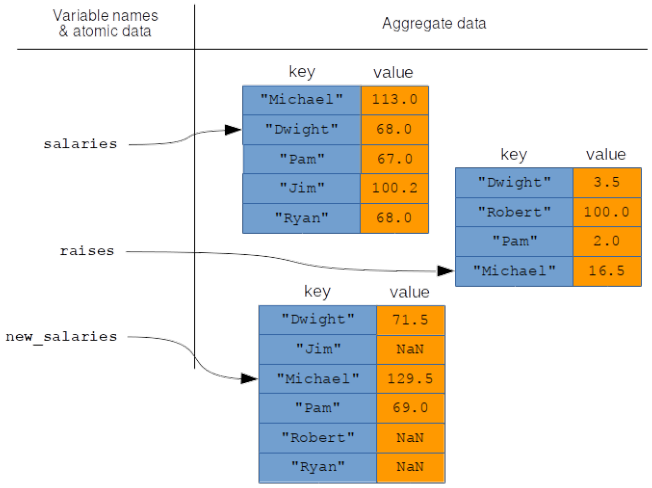


Figure 12.2: The result of +'ing two `Series`es that don't have all the same keys.

```
Dwight      71.5
Jim          NaN
Michael    129.5
Pam         69.0
Robert       NaN
Ryan         NaN
dtype: float64
```

Convince yourself that `Dwight`'s \$68,000 salary got added to his \$3,500 raise, that `Michael`'s \$113,000 salary was added to his \$16,500 raise, *etc.*

Don't get freaked out by those `NaN` entries just yet. The special value "**NaN**" stands for "**not a number**," and basically means that Pandas has to throw up its hands in that case. And with good cause. `Jim` has a current salary of \$100,200 in the first `Series`, but has no value at all in the second one (no raise for Jim this year? Haven't decided what his raise will be yet? Something else?) So Pandas does the safe thing, shrugs, and says "dunno." We say that the `Jim` entry in the `new_salaries Series` is a **missing value**. The same is true for `Robert` and `Ryan`, each of whom was present

in only one of the two operands.

Now I know what you're thinking: "can't Pandas just assume the salary and/or raise is 0 if there's a missing one?" The answer is that yes it can, but it won't do so unless you give the go-ahead. Pandas is being cautious here, and doesn't want to introduce errors into your data stream by false assumptions. (Maybe in your company, for instance, there's a default entry-level salary that every employee receives who's unspecified in the `salary Series`. Or maybe the yearly raise is always assumed to be a flat 2.5% cost-of-living raise unless explicitly specified.)

If we do want Pandas to assume a certain default value, we have to change tactics a bit and go with the `add()` function (or `sub()`, `mul()`, or `div()`):

```
new_salaries = pd.Series.add(salaries, raises, fill_value=0)
print(new_salaries)
```

```
Dwight      71.5
Jim        100.2
Michael    129.5
Pam         69.0
Robert     100.0
Ryan        68.0
dtype: float64
```

The `fill_value` argument is the important one here: it specifies what default value to use if one of the addends is missing a key from the other. Now the result is as in Figure 12.3. You can, of course, choose a `fill_value` other than zero, if you wish.

As with NumPy arrays, we can add (or subtract, or multiply, ...) a single atomic value to a series as well:

```
cost_of_living_increase = salaries * .025
print(cost_of_living_increase)
```

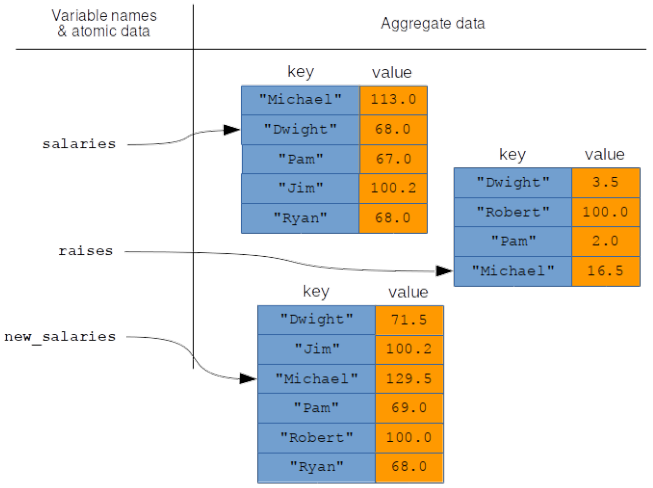


Figure 12.3: Using `add()` instead, and passing a `fill_value`.

```
Michael    2.825
Dwight     1.700
Pam        1.675
Jim        2.505
Ryan       1.700
dtype: float64
```

```
salaries = salaries + cost_of_living_increase
print(salaries)
```

```
Michael    115.825
Dwight      69.700
Pam         68.675
Jim        102.705
Ryan        69.700
dtype: float64
```

It can sometimes be useful to do string concatenation as well, for instance if we had employee first names and last names in two `Series`es with their employee ID as the index:

```
firsts = pd.Series(['Hannibal', 'Clarice', 'Multiple',
    'Buffalo'], index=[666, 1993, 47, 988])
lasts = pd.Series(['Starling', 'Crawford', 'Lecter', 'Bill',
    'Miggs'], index=[1993, 1650, 666, 988, 47])
print(firsts + " " + lasts)
```

```
47         Multiple Miggs
666      Hannibal Lecter
988         Buffalo Bill
1650                 NaN
1993    Clarice Starling
dtype: object
```

12.3 Copying `Serieses`

The rules for copying (or not copying) `Series`es are exactly the same as for NumPy arrays (see Section 9.4 on p. 79). If you merely assign one `Series` object to another variable, the two variables will be pointing to the *same* `Series` in memory, which means that changes to one will be reflected in the other. Calling the `.copy()` method, however, creates an entirely new `Series` in memory.

Make sure you understand the following output to confirm your understanding of this:

```
slayers = pd.Series([120, 72, 200], index=['Buffy','Xander','Willow'])
anti_vamps = slayers
good_guys = slayers.copy()
anti_vamps['Rubert'] = 150
print(slayers)
```

```
Buffy     120
Xander     72
Willow    200
Rubert    150
dtype: int64
```

```
print(anti_vamps)
```

```
Buffy     120
Xander     72
Willow    200
Rubert    150
dtype: int64
```

```
print(good_guys)
```

```
Buffy     120
Xander     72
Willow    200
dtype: int64
```

(The numbers here are approximate IQs; don't mean to be a hater.)

12.4 Sorting `Serieses`

Sorting is slightly more complex than for arrays, since there are two things we might want to sort by: the `Series`' index, or the values themselves. Correspondingly, there are two methods: `.sort_index()` and `.sort_values()`:

```
print(anti_vamps.sort_index())
```

```
Buffy     120
Rubert    150
Willow    200
Xander     72
dtype: int64
```

```
print(anti_vamps.sort_values())
```

```
Xander     72
Buffy     120
Rubert    150
Willow    200
dtype: int64
```

Like NumPy's `np.sort()` function (but unlike its `.sort()` method; refer back to Section 9.5 on p. 81 for details), neither of these methods actually sort the `Series` in place; instead, they return sorted copies. However, they can be made to work in place, by including "`inplace=True`" as an argument:

```
heroes_dumb_to_smart = anti_vamps.sort_values()
print(heroes_dumb_to_smart)
```

```
Xander     72
Buffy     120
Rubert    150
Willow    200
dtype: int64
```

```
print(anti_vamps)
```

```
Buffy     120
Xander     72
Willow    200
Rubert    150
dtype: int64
```

```
anti_vamps.sort_values(inplace=True)
print(anti_vamps)
```

```
Xander     72
Buffy     120
Rubert    150
Willow    200
dtype: int64
```

Another useful feature of both `.sort_X` methods is the ability to *reverse* sort. By adding "`ascending=False`" as an argument (with or without also including the "`inplace=True`" argument; they are combinable with a comma) you produce the reverse order:

```
heroes_smart_to_dumb = anti_vamps.sort_values(ascending=False)
print(heroes_smart_to_dumb)
```

```
Willow    200
Rubert    150
Buffy     120
Xander     72
dtype: int64
```

```
anti_vamps.sort_index(inplace=True, ascending=False)
print(anti_vamps)
```

```
Xander     72
Willow    200
Rubert    150
Buffy     120
dtype: int64
```

12.5 Concatenating and combining

Finally, it is sometimes convenient to be able to combine two or more `Series`es into a single one. But there's a catch. Remember that in order for a `Series` to "work properly," its keys must be unique. Combining two `Series` which share at least one of the same keys is a recipe for disaster!

The syntax for doing so, when the coast is clear, uses the `.append()` method:

```
crazy_example = salaries.append(slayers)
print(crazy_example)
```

```
Michael    113.0
Dwight      68.0
Pam         67.0
Jim        100.2
Ryan        68.0
Xander      72.0
Willow     200.0
Rubert     150.0
Buffy      120.0
dtype: float64
```

Nothing untoward happened here because *The Office* and *Buffy* don't have any overlapping character names. Note that the values all got converted to `float` (instead of `int`), to enforce homogeneity. Note also that `salaries` itself did *not* change as a result of this `.append()` call; instead, a new `Series` was returned that contains all the items.

12.6 Summary

All the functions from this chapter are summarized in Figure 12.4.

Function	Description
`len(`*ser*`)`	Get the number of key/value pairs in the `Series` *ser*.
ser`['Five Guys']`	Get the value of a specific key from the `Series` *ser*.
ser`.iloc[73]`	Treating the key/values pairs in the `Series` *ser* as ordered, get a specific numbered (from 0) value.
ser`.index[73]`	Treating the key/values pairs in the `Series` *ser* as ordered, get a specific numbered (from 0) key.
ser`['Firehouse'] = ...`	Set the value for a key of the `Series` *ser*.
ser`['New Rest'] = ...`	Add an additional key/value pair to the `Series` *ser*. (Same syntax as the previous.)
ser `+ 13`	Add a quantity to each value of *ser*, yielding a new *Series*. (Also works with `-`, `*`, `/`, *etc.*)
ser1 `+` *ser2*	Add pairs of values that have matching keys in two `Series`es, yielding a new `Series`. Use `NaN` for the value of any key that doesn't appear in both *ser1* and *ser2*. (Also works with `-`, `*`, `/`, *etc.*)
`pd.Series.add(`*ser1*`,` *ser2*`, fill_value=`*x*`)`	Add pairs of values that have matching keys in two `Series`es, yielding a new `Series`. Use `x` for any missing values. (Also works with `sub()`, `mul()`, `div()`, *etc.*)
ser1 `=` *ser2*	Make *ser1* point to the same data that *ser2* points to. (*Not* a copy!)
ser1 `=` *ser2*`.copy()`	Make *ser1* point to a new, independent copy of *ser2*.
ser`.sort_index()`	Return a copy of the `Series` *ser* which is sorted by the keys. Can also pass "`inplace=True`" to change *ser* itself, and/or pass "`ascending=False`" to get reverse order.
ser`.sort_values()`	Same as above, except that sorting is done with respect to values, not keys.
ser1`.append(`*ser2*`)`	Return a new `Series` with *ser1*'s and *ser2*'s key/value pairs smooshed together. (Bad things may happen if *ser1* and *ser2* share some of the same keys.)

Figure 12.4: Handy functions, methods, and operators for Pandas `Series`es.

Chapter 13

Associative arrays in Python (3 of 3)

But wait, there's more! We can also use methods like `.min()`, `.max()`, `.idxmin()`, and `.idxmax()` to get the "extremes" of a `Series` – *i.e.* the lowest and highest values in a `Series`, or their keys (indexes). Note that `.idxmin()` does *not* give you the lowest key in the `Series`! Instead, it gives you *the key of the lowest value.* Study this code snippet and its output to test your understanding of this:

```
understanding = pd.Series([15,4,13,3,7], index=[4,10,2,12,9])
print(understanding)
print("The min is {}.".format(understanding.min()))
print("The max is {}.".format(understanding.max()))
print("The idxmin is {}.".format(understanding.idxmin()))
print("The idxmax is {}.".format(understanding.idxmax()))
```

```
4     15
10     4
2     13
12     3
9      7
dtype: int64

The min is 3.
The max is 15.
The idxmin is 12.
The idxmax is 4.
```

The `idxmin` and `idxmax` are `12` and `4`, respectively, since the smallest value in the series (the `3`) has a key of `12`, and the largest value (the `15`) has a key of `4`.

If we did actually want the lowest (or highest) key, we could use the `.index` syntax (see p. 112) to achieve that:

```
print("The lowest key: {}.".format(understanding.index.min()))
print("The highest key: {}.".format(understanding.index.max()))
```

```
The lowest key: 2.
The highest key: 12.
```

And remember that "lowest"/"highest" for string data means alphabetical order.

13.1 Queries

One of the most powerful things we'll do with a data set is to **query** it. This means that instead of specifying (say) a particular key, or something like "the minimum" or "the maximum," we provide our own custom criteria and ask Pandas to give us all values that **match**. This kind of operation is also sometimes called **filtering**, because we're taking a long list of items and sifting out only the ones we want.

The syntax is interesting: you still use the boxies (like you do when giving a specific key) but inside the boxies you put a **condition** that will be used to select elements. It's best seen with an example. Re-using the `understanding` variable from above, we can query it and ask for all the elements greater than 5:

```
more_than_five = understanding[understanding > 5]
print(more_than_five)
```

```
4    15
2    13
9     7
dtype: int64
```

The new thing here is the "`understanding > 5`" thing inside the boxies. The result of this query is itself a `Series`, but one in which everything that doesn't match the condition is filtered out. Thus we only have three elements instead of five. Notice the keys didn't change, and they also had nothing to do with the query: our query was about *values*.

We could change this, if we were interested in putting a restriction on *keys* instead, using the `.index` syntax:

```
index_more_than_five = understanding[understanding.index > 5]
print(index_more_than_five)
```

```
10    4
12    3
9     7
dtype: int64
```

See how tacking on "`.index`" in the query made all the difference.

Query operators

Now I have a surprise for you. It makes perfect sense to use the character "`>`" (called "greater-than," "right-angle-bracket," or simply "wakka") to mean "greater than." And the character "`<`" makes sense as "less than." Unfortunately, the others don't make quite as much sense. See the top table in Figure 13.1.

"Greater/less than or equal to" isn't hard to remember, and it's a good thing Python doesn't require symbols like "≤" or "≥" since those are hard to find on your keyboard. You just type both symbols back-to-back, with no space. More problematic are the last two

entries in the top table. The "`!=`" operator (pronounced "bang-equals") is used as a stand-in for "≠" which also isn't keyboard friendly. And that one doesn't have a good mnemonic; you just have to memorize it.

By far the most error-prone of this set is the "`==`" (**double-equals**) operator, which simply means "equals." **Yes, you do have to use double-equals instead of single-equals in your queries, and yes it matters.** As additional incentive, let me inform you that if you use single-equals when you needed to use double-equals, *it will seem to work at first*, but you will silently get the wrong answer.

Memorize this fact! Failing to use double-equals is quite possibly the single most common programming error for beginners.

Simple query operators:

Symbol	Meaning
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`!=`	*not* equal to
`==`	equal to

Compound query operators:

Symbol	Meaning
`&`	and
`|`	or
`~`	not

Figure 13.1: Query operators: simple and compound

Here are some more examples to test your understanding. Make sure you understand why each output is what it is.

```
understanding = pd.Series([15,4,13,3,7], index=[4,10,2,12,9])
print(understanding[understanding <= 7])
```

```
10    4
12    3
9     7
dtype: int64
```

```
print(understanding[understanding != 13])
```

```
4     15
10     4
12     3
9      7
dtype: int64
```

```
print(understanding[understanding == 3])
```

```
12    3
dtype: int64
```

```
print(understanding[understanding.index >= 9])
```

```
10    4
12    3
9     7
dtype: int64
dtype: int64
```

Compound queries

Often, your query will involve more than one criterion. This is called a **compound condition**. It's not as common with `Series`es as it will be with `DataFrame`s in a couple chapters, but there are still uses for it here.

Suppose I want all the key/value pairs of `understanding` where the value is *between 5 and 14*. This is really two conditions masquerading as one: we want all pairs where (1) the value is greater than 5, **and** also (2) the value is less than 14. I put the word "**and**" in boldface in the previous sentence because that's the operator called for here. We only want elements in our results where *both* things are true, and therefore, we "*and* together the two conditions." ("And" is being used as a verb here.)

The way to achieve this is as follows. The syntax is nutty, so pay close attention:

```
x = understanding[(understanding > 5) & (understanding < 14)]
print(x)
```

```
2    13
9     7
dtype: int64
```

First, notice that we put each of our two conditions *in bananas* here. This is *not* optional, as it turns out: you'll get a non-obvious error message if you omit them. Second, see how we combined the two with the "`&`" operator from the bottom half of Figure 13.1. The result, then, was only the elements that satisfied *both* conditions.

It can be tricky to figure out whether you want an **and** or an **or**. Unfortunately they don't always correspond to their colloquial English usage. Let's see what happens if we switch the "`&`" symbol to a "`|`" (pronounced "pipe"):

```
y = understanding[(understanding > 5) | (understanding < 14)]
print(y)
```

```
4     15
10     4
2     13
12     3
9      7
dtype: int64
```

You can see that we got everything back. That's because **or** means "only give me the elements where *either one* of the conditions, *or both*, are true." In this case, this is guaranteed to match everything, because if you think about it, *every* number is either greater than five, or less than fourteen, or both. (Think deeply.)

Even though in this example it didn't do anything exciting, an "**or**" does sometimes return a useful result. Consider this example:

```
z = understanding[(understanding.index > 10) | (understanding > 5)]
print(z)
```

```
4     15
2     13
12     3
dtype: int64
```

Here we're asking for all key/value pairs in which *either* the key is greater than ten, *or* the value is greater than ten, or both. This reeled in exactly three fish as shown above. If we changed this "`|`" to an "`&`", we'd have caught *no* fish. (Take a moment to convince yourself of that.)

The last entry in Figure 13.1 is the "`~`" sign, which is pronounced "tilde," "twiddle," or "squiggle." It corresponds to the English word **not**, although in an unusual place in the sentence. Here's an example:

```
a = understanding[~(understanding.index > 10) | (understanding > 10)]
print(a)
```

```
4     15
10     4
2     13
9      7
dtype: int64
```

Search for and stare at the squiggle in that line of code. In English, what we said was "give me elements where either the key is *not* greater than ten, or the value is greater than ten, or both." The four matching elements are shown above.

Changing the "or" back to an "and" here gives us this output instead:

```
b = understanding[~(understanding.index > 10) & (understanding > 10)]
print(b)
```

```
4     15
2     13
dtype: int64
```

These are the only two rows where *both* conditions are true (and remember that the first one is "not-ted.")

It can be tricky to get compound queries right. As with most things, it just takes some practice.

Queries on strings

So far our examples have involved only numbers. Pandas also lets us perform queries on text data, specifying constraints on such things as the length of strings, letters in certain positions, and case (upper/lower).

Let's return to the Marvel-themed series from section 11.1:

```
alter_egos = pd.Series(['Hulk','Spidey','Iron Man','Thor'],
    index=['Bruce','Peter','Tony','Thor'])
```

By appending "`.str`" to the end of the variable name, we can get access to most of the string-based methods we'd like to use. For instance, find all the values with exactly four letters:

```
four_letter_names = alter_egos[alter_egos.str.len() == 4]
print(four_letter_names)
```

```
Bruce    Hulk
Thor     Thor
dtype: object
```

or all the values that contain a space:

```
spaced_out = alter_egos[alter_egos.str.contains(' ')]
print(spaced_out)
```

```
Tony    Iron Man
dtype: object
```

or all the keys whose first character is a `T`:

```
to_a_tee = alter_egos[alter_egos.index.str.startswith('T')]
print(to_a_tee)
```

```
Tony    Iron Man
Thor        Thor
dtype: object
```

or all entries where either the value is greater than five letters long or the key is the same as the value:

```
huh = alter_egos[(alter_egos.str.len() > 5) |
    (alter_egos.index == alter_egos)]
print(huh)
```

```
Peter    Spidey
Thor       Thor
dtype: object
```

The possibilities are endless. Some of the more common functions are summarized in Figure 13.2.

Function	Description
ser`.str.len()`	Set a condition on the length of a string.
ser`.str.startswith(`*str*`)`	Request only strings that begin with certain letter(s).
ser`.str.endswith(`*str*`)`	Request only strings that end with certain letter(s).
ser`.str.contains(`*str*`)`	Request only strings that contain certain letter(s) somewhere in them.
ser`.str.isupper()`	Request only strings that are in all upper-case.
ser`.str.islower()`	Request only strings that are in all lower-case.

Figure 13.2: Common query methods for string data.

Last word

A couple things before we move on. You've noticed that in all the above examples, it was necessary to type the `Series` variable name several times:

```
understanding[(understanding < 12) | (understanding > 18)]
alter_egos[(alter_egos.str.isupper()) & (alter_egos.str.len() < 10)]
```

There's really no way around that, sorry; you just have to get used to it. A very common beginner error is to try and write this:

```
understanding[understanding < 12 | > 18]
```

This seems to make perfect sense, especially since it mimics the natural English sentence: "give me all values where `understanding` is *less than 12 or greater than 18*." Unfortunately, it doesn't work like that in Python. The rule is: each side of an *and* or an *or* must be a complete sentence. The phrase "`understanding` is greater than 18" counts as a complete sentence, but "is greater than 18" does not.

Also, whenever I see a line of code that specifies a key to a `Series` (or array), I mentally pronounce the opening boxie (" [") as the word "of". So when I read:

```
print(x[5])
```

I say to myself "print x **of** five."

However, whenever I see a *query*, I mentally pronounce the boxie as the word "where". So when I read:

```
print(x[x > 12])
```

I say to myself "print x **where** x is greater than 12." I've found this helpful in making sense of the meaning of queries, since they're complicated enough as it is!

Chapter 14

Loops

It's time for our first look at a **non-linear** program. Up to now, all of our Python programs have executed step-by-step, start to finish, like a metronome, with each line of code getting executed exactly once. That's about to change. In this chapter, we introduce the concept of a **loop**, which is a programming construct that directs lines of code to be executed *repeatedly*, and out of strict sequence.

14.1 The two species of loops

Although some programming languages try to dress them up further, there are really only two fundamental kinds of loops in the world: **fixed-iteration loops** and **variable-iteration loops**.[1] The first kind is simpler to understand and less error-prone; in most languages (Python included) it is implemented as a "`for` **loop**." The tricker, second kind is available to programmers as a "`while` **loop**."

Happily for us, it turns out that `while` loops don't come up much in Data Science, at least in the beginning. There are some more advanced techniques that use them (for instance, optimization methods and threshold detection) but for us it's going to be `for` loops that dominate the landscape. So let's figure out how they work.

[1] Sometimes these are called **counter-controlled** and **condition-controlled** loops, respectively.

14.2 A word of caution

But before we embark, a cautionary note. Some of the things that loops can do – especially the early examples – can also be done using the queries of the last chapter. For instance, we could use a query to find all the strings in a `Series` that begin with the letter `T`, or we could use a loop to do the same thing.

Here's the rule: **if you can do it without a loop, that is always preferred.** There are two reasons for this. First, it's less code to write, and less error-prone, to use Pandas' built-in features rather than crafting a loop yourself. That's why they created those features (like queries) after all.

Second, and ultimately even more important, using a Pandas function is *much faster* to execute than a loop. The reason has to do with how a loop is eventually broken down into the little instructions a machine can understand: when Python runs a loop, it plods through the steps methodically, whereas the Pandas functions are all pre-baked into a water-cooled rocket engine that can jet out of the gate.

You don't need to know any of those nitty-gritty details. All you have to remember is: don't ever resort to using a loop unless you can't figure out how to do what you want without one. (And unfortunately, there are indeed those times.)

14.3 Iterating through an array

Most often, we'll use a `for` loop to "loop through," or "**iterate** through," the contents of an aggregate data variable. This means that instead of executing a snippet of code *once*, we'll execute it *once per element of the variable*. This "once per element" thing is what makes the code non-linear.

Let's start with the first aggregate data type we learned, a NumPy array.

```
1: villains = np.array(['Jafar','Ursula','Scar','Gaston'])
2: print("Here we go!")
3: for v in villains:
4:     print("Oooo, {} is scary!".format(v))
5:     print("({} has {} letters.)".format(v, len(v)))
6: print("Whew!")
```

(I've numbered the lines in this example so I can refer to them in the text below, but the numbers and colons aren't part of Python.)

Immediately after creating our `villains` array, and printing an introductory message, we encounter our first loop. A loop consists of two parts: the **loop header** and the **loop body**. Here are the rules:

- The loop header consists of the line that begins with "`for`".
- The loop body consists of *all of the consecutive following lines that are **indented** (tabbed-over) one tab.*[2]

That second rule turns out to be more important than it seems at first. A very (*very!*) common error among beginners is to "mis-indent" their code such that their loop body includes more, or less, than they mean it to. So heads up.

Before we continue, stare at that code above and convince yourself of these two facts:

☞ The loop header is line **3**.
☞ The loop body is lines **4 and 5**. (*Not* line 4 only! *Not* lines 4, 5, and 6!)

Now the reason this is important is that a `for` loop works as follows:

[2]Other programming languages – every other one I know besides Python, in fact – uses some other way to designate the loop body than indentation. Many (R and Java, for instance) use curly braces before and after the loop body so that the computer knows where it begins and ends. I personally like this feature of Python's, but there are haters, and the bottom line is you just have to get used to it.

1. First, create a new variable (on the left-hand side of the memory picture) named whatever comes immediately after the word "`for`". (In this example, the name of this new variable will be `v`.)
2. Then, for *each* element of the array, in succession:
   a) Set that variable's value to the next element of the array.
   b) *Execute the entire loop body.* (In this example, lines 4–5.)

In the `villains` example, therefore, the lines in order of execution are:

1, 2, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5, 6.

(Do you agree?)

The memory picture changes constantly throughout any program, including those that contain loops. Let's take a snapshot of memory as it appears immediately after executing line 3 the *second* time. In other words, we'll run the program this far before hitting the pause button:

1, 2, 3, 4, 5, 3, *Freeze!!*

Memory at this instant is depicted in Figure 14.1. The second time we executed line 3, we set `v` (sometimes called the **loop variable**, by the way) to the second element of the array, `"Ursula"`. We're just about to execute line 4 for the second time. Note that the `villains` array is unaffected by this entire loop process: only our temporary, made-up loop variable (`v`) gets a new value each time.

The complete output of the program, as you can easily deduce, is thus:

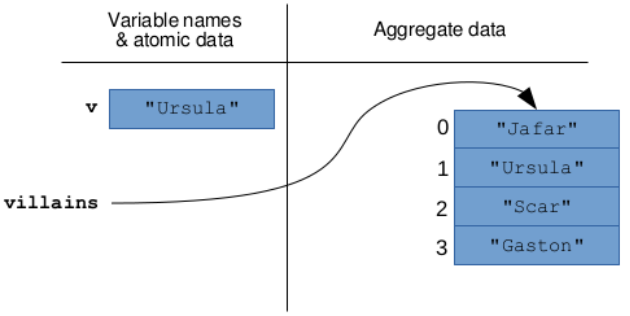


Figure 14.1: A snapshot of memory immediately after the *second* execution of line *3* of the `villains` program.

```
Here we go!
Oooo, Jafar is scary!
(Jafar has 5 letters.)
Oooo, Ursula is scary!
(Ursula has 6 letters.)
Oooo, Scar is scary!
(Scar has 4 letters.)
Oooo, Gaston is scary!
(Gaston has 6 letters.)
Whew!
```

Don't miss the fact that the "`scary!`" and "`has` n `letters`" messages were printed four times each, whereas "`Whew!`" only appeared once. That has everything to do with the indentation: it told Python that lines 4 and 5 *were* part of the loop body, whereas line 6 was just "business as usual," taking place only after all the loop hoopla was over and done with.

14.4 Iterating through the values of a `Series`

Great news: if you mastered the previous section, this one and the next will be a snap. That's because Python, NumPy, and Pandas work together to make iterating through a `Series` pretty much *exactly the same* as iterating through an array. In fact, sometimes you're not even sure which type you've got!

Here's the Marvel `Series` regurgitated yet again:

```
alter_egos = pd.Series(['Hulk','Spidey','Iron Man','Thor'],
    index=['Bruce','Peter','Tony','Thor'])
```

Let's say I want to go through and greet all our heroes. It's a snap! (no pun intended):

```
print("Welcome to the Marvel Cinematic Universe(tm).")
for hero in alter_egos:
    print("Greetings, {}!".format(hero))
print("Go team!")
```

```
Welcome to the Marvel Cinematic Universe(tm).
Greetings, Hulk!
Greetings, Spidey!
Greetings, Iron Man!
Greetings, Thor!
Go team!
```

What could be easier?

Notice that "looping through the `Series`" effectively means "looping through the *values* of the `Series`," not the keys. What if we want to loop through the keys instead?

14.5 Iterating through the keys of a `Series`

I'm glad you asked. But in fact, you already know the answer: just use the `.index` syntax from p. 112!

```
print("Let's iterate through the keys instead:")
for secret_identity in alter_egos.index:
    print("Nice to meet you, {}.".format(secret_identity))
print("Carry on...")
```

```
Let's iterate through the keys instead:
Nice to meet you, Bruce.
Nice to meet you, Peter.
Nice to meet you, Tony.
Nice to meet you, Thor.
Carry on...
```

By the way, you can see that the name of the loop variable is completely at your discretion. I called the previous one "`hero`" and this one "`secret_identity`" just because those names were reflective of their contents. But it's really up to you: it has nothing to do with the name of the `Series` itself. (Yeah, I know the Marvel identities aren't secret anymore, but I'm old school.)

14.6 Iterating through the keys *and* values of a `Series`

Finally, it's common to need access to both halves of each key/value pair as you iterate through a `Series`. The way to accomplish this is to call the `.items()` method of the `Series`. But it's tricky, because when you use `.items()` you assign *two* variables in your loop instead of just one.

Before showing the complete loop, let's focus on just the loop header needed for this technique:

```
for secret_identity, hero in alter_egos.items():
```

I named two loop variables, separated by a comma. The reason I put `secret_identity` first is that in this `Series`, we used `Bruce`, `Peter`, *etc.* as the *keys*, with the superhero names as the values. And with `.items()`, the variable name you want to use for the key is listed first.

The rest of the loop follows logically from this, with both variables available inside the loop body:

```
print("We're now going to recognize some outstanding citizens.")
for secret_identity, hero in alter_egos.items():
    print("{}, known to his friends as {}.".format(hero,
        secret_identity))
    print("The crowd screams: 'YAY {}!'".format(hero.upper()))
print("Thanks, everyone, for your service.")
```

If we freeze the program just after the *third* execution of the loop header this time, we get the picture in Figure 14.2. And the output, of course, is:

```
We're now going to recognize some outstanding citizens.
Hulk, known to his friends as Bruce.
The crowd screams: 'YAY HULK!'
Spidey, known to his friends as Peter.
The crowd screams: 'YAY SPIDEY!'
Iron Man, known to his friends as Tony.
The crowd screams: 'YAY IRON MAN!'
Thor, known to his friends as Thor.
The crowd screams: 'YAY THOR!'
Thanks, everyone, for your service.
```

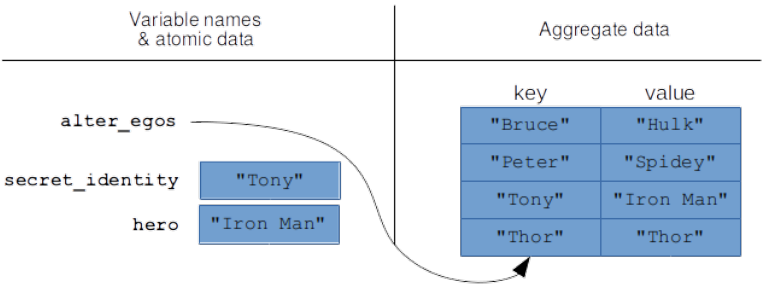


Figure 14.2: A snapshot of memory immediately after the *third* execution of the loop header in the `alter_egos` program.

14.7 Wrapping up

We can, of course, do much more inside loops than just print things. We can perform computations galore. The examples in this chapter were simply to illustrate the structure and behavior of `for` loops, so that you have a framework for understanding how more complex parts fit into them later.

Onward!

Chapter 15

Exploratory Data Analysis: univariate

The fancy term "**Exploratory Data Analysis**" (EDA) basically just means getting acquainted with your data. After importing a new data set into Python, the first thing you normally do is poke around to get an idea of what it contains. You may not even know what questions you eventually want to ask – let alone what the answers are – but sizing up the data is a necessary precursor to those activities.

In this chapter, we'll learn some basic EDA techniques for **univariate data**, which is really all we've studied so far. "Univariate" means to consider just one variable at a time, rather than possible relationships between variables. A single (one-dimensional) NumPy array or Pandas `Series` is a univariate data set, if you treat it in isolation. As it turns out, there's quite a few interesting things you can do with even something that simple.

First, we'll look at **summary statistics**, which are a way to capture the general features of a data set so you can see the forest instead of just a bunch of trees. Which type of summary information is appropriate depends on whether you're dealing with categorical or numeric data.

15.1 Categorical data: counts of occurrences

Let's say you had access to a poll on people's favorite pop stars. You import this into a big ol' Pandas `Series` called `faves`:

```
print(faves)
```

```
0          Katy Perry
1             Rihanna
2       Justin Bieber
3               Drake
4             Rihanna
5        Taylor Swift
6               Adele
7               Adele
8        Taylor Swift
9       Justin Bieber
...
1395       Katy Perry
dtype: object
```

That's great, but it's also kinda TMI. You probably don't care who the *first* person's idol is, nor the fifteenth, nor the last. Much more interesting is simply *how many times* each value appears in the `Series`. This information is available from the Pandas `.value_counts()` method:

```
counts = faves.value_counts()
print(counts)
```

```
Taylor Swift     388
Katy Perry       265
Drake            261
Adele            212
Rihanna          136
Justin Bieber    134
dtype: int64
```

The `.value_counts()` method returns another `Series`, but the *values* of the original `Series` become the *keys* of the new one. This tells us at a glance how popular each answer is relative to the others.

To get percentages instead of totals, just divide by the total and multiply by 100, of course:

```
print(counts / len(counts) * 100)
```

```
Taylor Swift     27.7937
Katy Perry       18.9828
Drake            18.6963
Adele            15.1862
Rihanna          09.7421
Justin Bieber    09.5989
dtype: float64
```

Recall (p. 45) that the **mode** is the only measure of central tendency that makes sense for categorical data. And all you have to do is call `.value_counts()` and look at the top result. (In this case, `Taylor Swift`.)

Note that `.value_counts()` is a Pandas `Series` method, not a NumPy method. If you find yourself with a NumPy array instead, you can just **wrap** it in a `Series` as we did in Section 11.1 (p. 106):

```
my_array = np.array(['red','blue','red','green','green',
    'green','blue'])
print(pd.Series(my_array).value_counts())
```

```
green    3
red      2
blue     2
dtype: int64
```

15.2 Numerical data: quantiles

A **quantile** is a real number between 0 and 1 that represents a "**cut point**" of a numerical data set: roughly speaking, it's the number for which a certain fraction of the values are *less than* that number. So the ".2-quantile" (pronounced "point two quantile") of a variable containing the heights of third-graders might be 50 inches. If that's the case, it would indicate that *20% of the third-graders are less than 50 inches tall.*

Quantiles are very revealing, but underappreciated. Most people don't seem to know how to interpret them. But once you figure it out, you'll realize that quantiles tell you almost everything possible to know about a numeric variable: by dialing the quantile between 0 and 1, you can tell exactly how common values in certain ranges are.

In Python, you simply call the `.quantile()` method on a `Series`, passing a number between 0 and 1 as an argument, and it tells you exactly where that cut point is.

Now there's a little bit of weirdness around the edges, depending on the exact definition used to calculate the quantiles. Let's say I collected some salary data, and got these responses ("k" means "thousand dollars per year," and "M" means "million dollars per year"):

35k 22k 67k 45k 35k 8M 94k 51k 53k 64k 54k

How would I calculate, say, the .7-quantile? First, sort the numbers:

22k 35k 35k 45k 51k 53k 54k 64k 67k 94k 8M

(yes, we *do* include the 35k value twice; don't eliminate duplicates) and then spread out the quantiles "evenly" from 0 to 1:

value:	22k	35k	35k	45k	51k	53k	54k	64k	67k	94k	8M
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
quantile:	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	1.0

Don't get picky on me. If you were picky, you could quibble at saying "the .3-quantile is 45k" since it's technically *not* true that 30% of the values are less than 45k: in truth, 3 out of 11 (27.3%) of them are. Whatever, whatever. The point is that 45k is at the "cut point" that's $\frac{3}{10}$ths of the way through the values from min to max. Quantiles aren't about laser precision anyway: they're about understanding the general pattern of the data.

"Special" quantiles

You'll realize as an immediate consequence of the above that the **median** is just another name for the **.5-quantile**. It's the value for which half the data points are below it, and half above. Also, the **0-quantile** is just the minimum of the data set, and the **1-quantile** the maximum.

Other kinds of "-tiles"

This whole idea may ring bells with other names for you: qua**r**tiles, qu**i**ntiles, deciles, or (most probably) percentiles. All of those are basically special cases of quantiles. They split the data into evenly-sized groups:

- "quartiles": split the data into four groups, with the split points being the .25-, .5-, and .75-quantiles.
- "quintiles": split the data into five groups, with the split points being the .2-, .4-, .6, and .8-quantiles.
- "deciles": split the data into ten groups, with the split points being the .1-, .2-, .3-, .4-, .5-, .6-, .7-, .8-, and .9-quantiles.
- "percentiles": split the data into 100 groups, with the split points being the .01-, .02-, .03-, ..., .98-, and .99-quantiles.

Be careful to understand that "evenly-sized groups" does not mean "groups with the same-sized range," but rather "groups with *the same number of data points in them.*" Normally, in fact, the ranges will *not* be the same size. The lowest quintile for a data set of IQs might range from 47 (the lowest IQ in the data set) all the way up to 83, whereas the IQs in the middle quintile might all be in the narrow range 96 to 104.

The IQR (interquartile range)

Speaking of quartiles, you'll commonly hear data scientists cite the **IQR**, or **interquartile range**, as a measure of how widely varying a univariate data set is. It's simply the distance between the .25-quantile and the .75-quantile; or in quartile terms, the difference between the "upper" and "lower" quartiles.

Because of how quantiles work, exactly 50% of the data points are between the .25- and .75-quantiles. This means that the more spread out the data points are, the larger the IQR, and vice versa. In this sense, it's akin to the standard deviation (see p. 153) which you may be familiar with.

A quantile example

Let's nail this down with an example. I have a (fictitious) data set containing the number of YouTube plays for each of a selection of videos. It's called `num_plays`. Here are the first few values:

```
0     791
1    3133
2       0
3    1789
4     297
5     219
6    1688
7     209
8     422
9   91454
dtype: int64
```

That's great, but it's both too much information and too little: we can pore through the plays for every single video, but it's hard to get our head around what the overall contents are. So let's run some quantiles. We'll start with the .1-quantile:

```
print(num_plays.quantile(.1))
```

```
0.0
```

Whoa. The .1-quantile is *zero*. Think about what that means. Pictorially, sorting the data would give this:

value:	0	0	0	0	0	0	0	0	···	0	0	···
	↑										↑	
quantile:	.0										.1	

Put another way, that means that (at least) *10% of our videos have no plays at all.*

Let's try the .2-quantile:

```
print(num_plays.quantile(.2))
```

```
15.0
```

Okay, now at least we have a pulse. But in case we thought this was data set was packed with big hits, think again: a full 20% of these videos have fewer than 15 plays.

The median is:

```
print(num_plays.quantile(.5))
```

```
263.0
```

That's quite a bit higher. How about the 90% mark?

```
print(num_plays.quantile(.9))
```

```
1378.0
```

All right, so the upper end of these videos are in the thousands. Finally, let's look at the max:

```
print(num_plays.quantile(1))
```

```
982221.0
```

!!

Believe it or not, this sort of thing isn't unusual, especially with data from social phenomena. The tiny fraction of the data at the upper end of the range is *vastly* higher than everything else is. Get your head around that: the median number of plays was a couple hundred, but the maximum number of plays was nearly a *million*.

Computing the IQR of this data set is as simple as finding the difference between the .25 and .75 quantiles:

```
print(num_plays.quantile(.75) - num_plays.quantile(.25))
```

```
399.75
```

15.3 Numerical data: other summary statistics

That YouTube data set is a good segue to talking about that most overused of all statistics: the **mean**. Nearly everyone, if you ask them "what's the typical number of plays for these videos?" will use the mean, or average, to get at the answer. After all, isn't that what we mean by "the average number of plays?"

The answer is: not really, and not usually. Look what happens if we compute the mean (using the `.mean()` `Series` method) in this case:

```
print(num_plays.mean())
```

```
14018.888235294118
```

Consider just how misleading that really is. The "average" number of plays is over 14,000. Yet the *.9*-quantile was less than $\frac{1}{10}$th of that! In fact, even the .97-quantile is only:

```
print(num_plays.quantile(.97))
```

```
3836.0
```

So *over 97% of the videos have less than the mean of 14,000 plays.* I think you'll agree that it is nonsensical to claim that "the typical number of plays is 14,018," no matter how you slice it.

We'll see in the next section why the mean is hopelessly skewed here. Basically, unless the data is symmetrical and "bell-curvy," it gives a meaningless number. It is almost *always* safer and more illuminating to look at the median (or other quantiles).

For completeness, one other commonly cited summary statistic is the **standard deviation**, which can be computed with the `.std()` method:

```
print(num_plays.std())
```

```
93031.835
```

The standard deviation, like the IQR, is a measure of the "spread" of a data set – a high number means (in this example) higher variability in the number of plays from video to video. As with the mean, it's essentially meaningless (no pun intended) unless the data is nice and bell-curve shaped.

Speaking of which, we'll never be able to judge the "shape" of anything unless we get some graphical plots involved. So let's turn our focus to that.

15.4 Plotting univariate data

There are basically two useful ways of plotting a `Series` with univariate data. In one, you care about the specific labels (*i.e.* keys, or "index") of the values in the `Series`, and you want them to be prominent in the plot. In the other, you don't; you just want to show the values themselves, so you can visualize how they are distributed irrespective of what label they might have.

Let's do the first one first.

Bar charts of labeled data

Let's read a data set on the world countries with the highest GDP (Gross Domestic Product). Here's a CSV file called `gdp.csv`[1]:

```
Nation,Trillions
Italy,2.26
Germany,4.42
Brazil,2.26
United States,21.41
France,3.06
Canada,1.91
Japan,5.36
China,15.54
India,3.16
United Kingdom,3.02
```

We'll read that into a `Series` using our technique from p. 107:

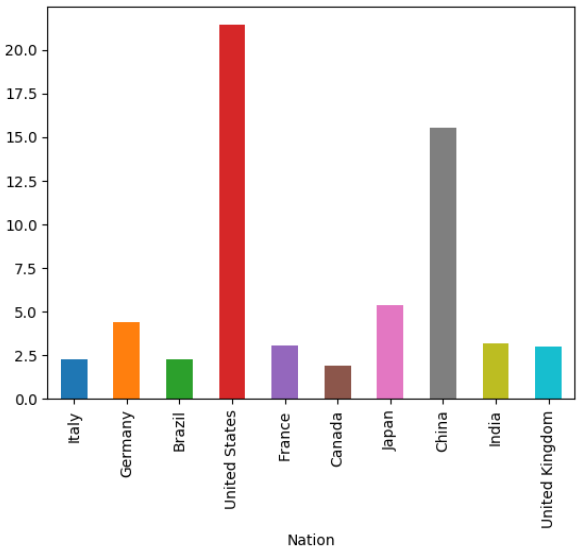
```
gdp = pd.read_csv('gdp.csv', squeeze=True, index_col=0,
    header=None)
print(gdp)
```

[1] Recall the caveat about filename extensions in the p. 69 footnote.

```
0
Nation            Trillions
Italy                  2.26
Germany                4.42
Brazil                 2.26
United States         21.41
France                 3.06
Canada                 1.91
Japan                  5.36
China                 15.54
India                  3.16
United Kingdom         3.02
Name: 1, dtype: object
```

and now, we can visualize the relative sizes of these economies with the `.plot()` method. The `.plot()` method takes, among other things, a "`kind`" argument which specifies what kind of plot you want. In this case, a **bar chart** is the correct thing:

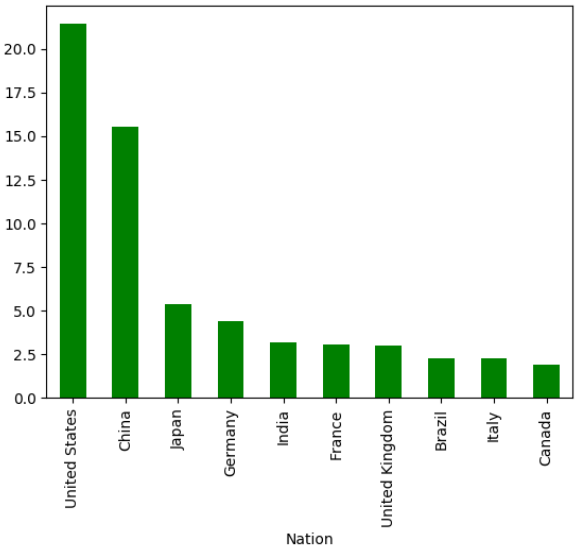
```
gdp.plot(kind='bar')
```



There are a zillion ways to customize these plots, and I'll only mention a very, very few. A more complete list of options is available by Googling, or going to `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html`

For instance, to make all the bars the same color, we can pass "`color="blue"`". Sorting the values is something we already know how to do, with `.sort_values()`:

```
gdp.sort_values(ascending=False).plot(kind='bar')
```



You see what I mean about "caring about the labels/keys/index" for this sort of plot: if we hadn't labeled the bars, the plot would tell us nothing useful.

I'm sure you've seen lots of bar charts in your life, so this is nothing new. But consider how much information is embedded in this infographic. Not only can we tell that the U.S. and China are the two biggest economies, we can tell that they are *far and away* the two biggest, with Japan and Germany (the next two highest) only a fraction.

Bar charts of occurrence counts

A very common special case of a bar chart is one where we combine it with the `.value_counts()` method. Let's go back to Taylor vs. Katy:

```
print(faves)
```

```
0          Katy Perry
1             Rihanna
2       Justin Bieber
3              Drake
4             Rihanna
5        Taylor Swift
6              Adele
7              Adele
8        Taylor Swift
9       Justin Bieber
...
1395       Katy Perry
dtype: object
```

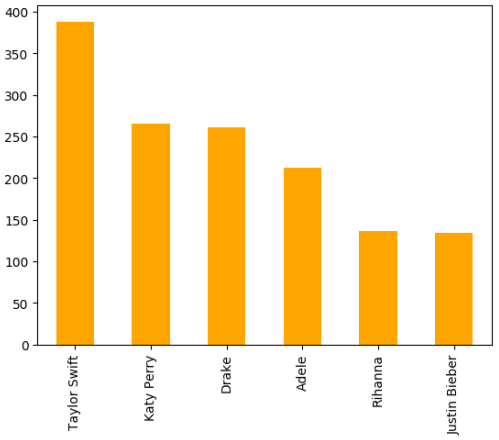
It would be useful to see an infographic on how popular each celebrity is, and combining `.value_counts()` and `.plot()` makes it a snap:

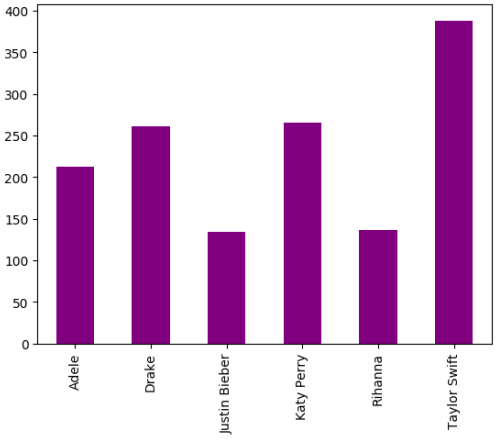
```
faves.value_counts().plot(kind='bar',color="orange")
```

The `.sort_values()` method wasn't needed here, because our friend `.value_counts()` already returns its answer in decreasing numerical order. If we wanted the bars in alphabetical order instead, we'd just sort the `Series` by index before plotting:

```
faves.value_counts().sort_index().plot(kind='bar',
    color="purple")
```

These long lines with lots of strung-together methods are concise, but can also be confusing. It's always an option to use temporary variables to store the intermediate results instead:






```
counts = faves.value_counts()
alphbetical_counts = counts.sort_index()
alphbetical_counts.plot(kind='bar',color="purple")
```

Just a matter of preference.

15.5 Numerical data: histograms

As I mentioned on p. 154, sometimes we don't actually care about the labels in a `Series`, only the values. This is when we're trying to size up how *often* values of various magnitudes appear, irrespective of which specific objects of study those values go with.

My favorite plot is the **histogram**. It's super powerful if you know how to read it, but underused because few people seem to know how. The idea is that we take a numeric, univariate data set, and divide it up into **bins**. Bins are sort of the reverse of quantiles: all bins have the *same* size range, but a *different* number of data points fall into each one.

Suppose we had data on the entire history of a particular NCAA football conference. A `Series` called "`pts`" has the number of points scored by each team in all that conference's games. It looks like this:

```
print(pts)
```

```
0        7
1       35
2       40
3       17
4       10
...
399     14
dtype: int64
```

Some basic summary statistics of interest include:

```
print("min: {}".format(pts.quantile(0)))
print(".25-quantile: {}".format(pts.quantile(.25)))
print(".5-quantile: {}".format(pts.quantile(.5)))
print(".75-quantile: {}".format(pts.quantile(.75)))
print("max: {}".format(pts.quantile(1)))
print("mean: {}".format(pts.mean()))
```

```
min: 0.0
.25-quantile: 17.0
.5-quantile: 25.0
.75-quantile: 32.0
max: 55.0
mean: 23.755
```

Looks like a typical score is in the 20's, with the conference record being a whopping 55 points in one game. The IQR is 32 – 17, or 15 points.

We can plot a histogram of this `Series` with this code:

```
pts.plot(kind='hist')
```

The result is in Figure 15.1. Stare hard at it. Python has divided up the points into ranges: 0 through 5 points, 6 through 11 points, 12 through 17, *etc.* Each of these ranges is a bin. The height of each blue bar on the plot is simply the number of games in which a team scored in that range.

Now what do we learn from this? Lots, if we know how to read it. For one thing, it looks like the vast majority of games have teams scoring between 12 and 38 points. A few teams have managed to eke out 40 or more, and there have been a modest number of single-digit scores or shutouts. Moreover, it appears that scores between 24 and 38 are considerably more common than those between 12 and 24. Finally, this data shows some evidence of being "bell-curvy" in the sense that values in the middle of the range are more common than values at either end, and it is (very roughly) symmetrical on both sides of the median.

This is even more precise information than the quantiles gave us. We get an entire birds-eye view of the data set. Whenever I'm looking at a numerical, univariate data set, pretty much the first thing I do is throw a histogram up on the screen and spend at least a couple minutes staring at it. It's almost the best diagnostic tool available.

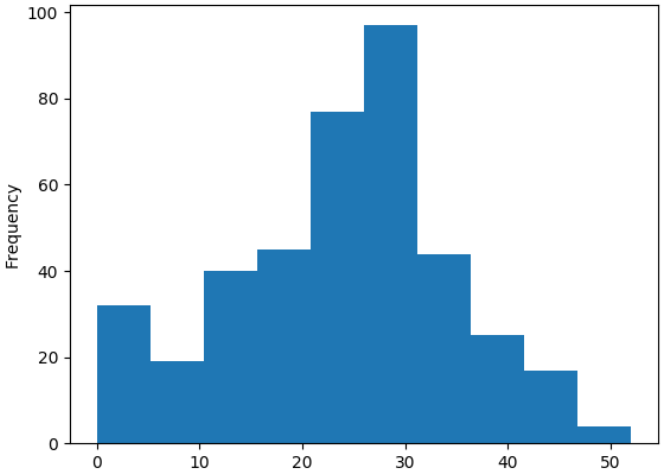


Figure 15.1: A histogram of the historical points-per-game for teams in a certain NCAA football conference.

Bin size

Now one idiosyncrasy with histograms is that a lot depends on the bin size and placement. Python made its best guess at a decent bin size here by choosing ranges of 6 points each. But we can control this by passing a second parameter to the `.plot()` function, called "`bins`":

```
pts.plot(kind='hist', bins=30)
```

Here we specifically asked for *thirty* bins in total, and we get the result in Figure 15.2. Now each bin is only two points wide, and as you can see there's a lot more detail in the plot.

Whether that amount of detail is a good thing or not takes some practice to decide. Make your bins too large and you don't get much precision in your histogram. Make them too small and the trees can overwhelm the forest. In this case, I'd say that Figure 15.2

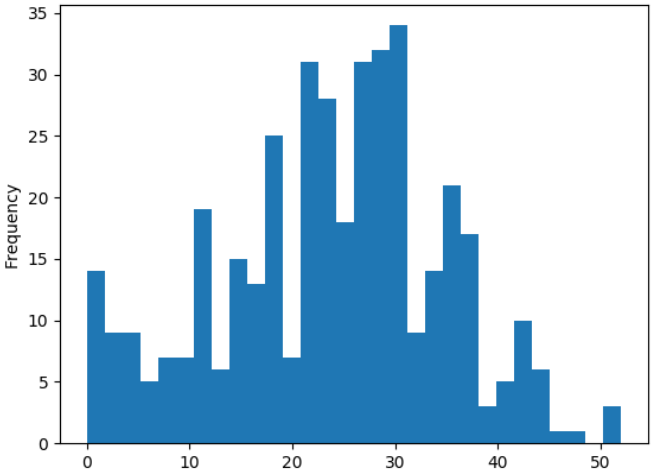


Figure 15.2: The same data set as in Figure 15.1, but with more (and smaller) bins.

is good in that it tells us something not apparent from Figure 15.1: there are quite a few shutouts (zero-point performances), not merely games with six-points-or-less. Whether the trough between 22 and 24 points is meaningful is another matter, and my guess is that part is obscuring the more general features apparent in the first plot.

The rule is: whenever you create a histogram, *take a few minutes to experiment with different bin sizes.* Often you'll find a "sweet spot" where the amount of detail is just right, and you'll get great insight into the data. But you do have to work at it a little bit.

Non-bell-curvy data

Let's return again to the YouTube example. We had some surprises when we looked at the quantiles and saw that the 1-quantile (max) was astronomically higher than the .9-quantile was. Let's see what happens when we plot a histogram (show in Figure 15.3):

```
num_plays.plot(kind='hist', color="red")
```

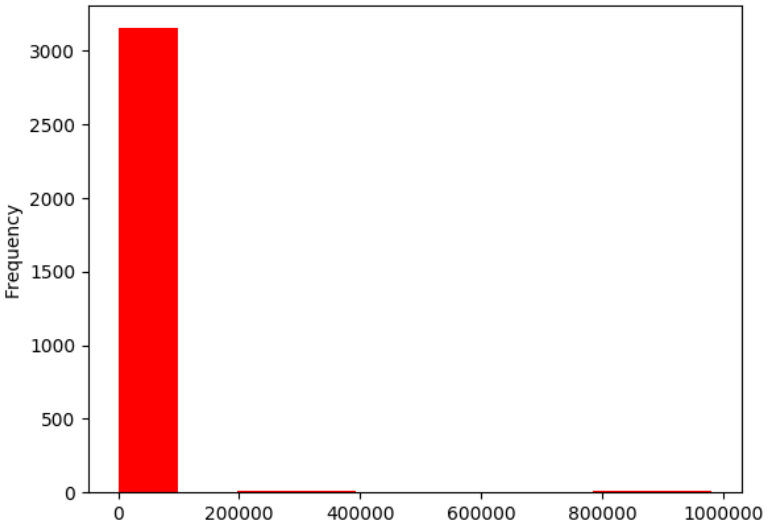


Figure 15.3: A first attempt at plotting the YouTube `num_plays` data set.

Huh?? Wait, where are all the bars of varying heights? We seem to have got only a single one.

But they're there! They're just so small you can't see them. If you stare at the x-axis – and your eyesight is good – you might see tiny signs of life at higher values. But the overall picture is clear: the vast, vast majority of videos in this set have between 0 and 100,000 plays.

Let's see if we can get more detail by increasing the number of bins (say, to 1000):

```
num_plays.plot(kind='hist', bins=1000, color="red")
```

We now get the left-hand side of Figure 15.4. It didn't really help much. Turns out the masses aren't merely crammed below a hundred thousand plays; they're crammed below *one* thousand. We need another approach if we're going to see any detail on the low-play videos.

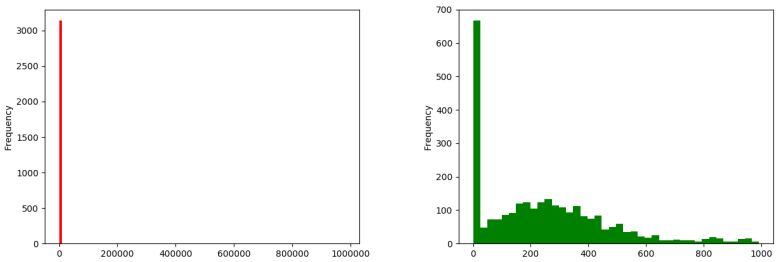


Figure 15.4: Further attempts at plotting the YouTube `num_plays` data set. On the left side, we decreased the bin size to no avail. On the right side, we gave up on plotting the popular videos and concentrated only on the unpopular ones, which does illuminate the lower end somewhat. (Don't miss the x-axis ranges!)

The only way to really see the distribution on the low end is to *only* plot the low end. Let's use a query (recall section 13.1 from p. 124) to filter out *only* the videos with 1000 plays or fewer, and then plot a histogram of that:

```
unpopular_video_plays = num_plays[num_plays <= 1000]
unpopular_video_plays.plot(kind='hist', color="green")
```

This gives the right-hand side of Figure 15.4. Now we can at least see what's going on. Looks like our `Series` has a crap-ton of videos that have never been viewed at all (recall our .1-quantile epiphany for this data set on p.151) plus a chunk that are in the 500-views-or-fewer range.

The takeaway here is that not all data sets (by a long shot!) are bell-curvy. Statistics courses often present nice, symmetric data sets on physical phenomena like bridge lengths or actor heights or

free throw percentages, which have nice bell curves and are nicely summarized by means and standard deviations. But for many social phenomena (like salaries, numbers of likes/followers/plays, lengths of Broadway show runs, *etc.*) the data looks more like this YouTube example. A few extremely large values dominate everything else by their sheer magnitude, which makes it more difficult to wrap your head around.

It also makes it more challenging to answer the question, "what's the *typical* value for this variable?" It ain't the mean, that's for sure. If you asked me for the "typical" number of plays of one of these YouTube videos, I'd probably say "zero" since that's an extremely common value. Another reasonable answer would be "somewhere in the low hundreds," since there are quite a few videos in that range, as illustrated by the right-hand-side of Figure 15.4. But you'd be hard-pressed to try and sum up the entire data set with a single typical value. There just isn't one for stuff like this.

15.6 Numerical data: box plots

Let's talk about one more type of plot in this chapter, even though it's really most useful when dealing with bivariate data, as we'll address in chapter 20. It's called the **box plot** (also known as a "**box-and-whisker" plot**). We can create one by passing "`kind="box"`" to the `.plot()` method (here for the NCAA football data):

```
pts.plot(kind="box")
```

The result is shown in Figure 15.5, along with some annotations in red so you can figure out what's going on.

For now, don't worry about the mysterious word "`None`" at the bottom. (This indicates which "group" the box represents, and will feature prominently in our bivariate data chapter.) For a univariate data set like this one, the x-axis has no meaning. The y-axis, on the other hand, is easy to understand: it's the number of points per football game.

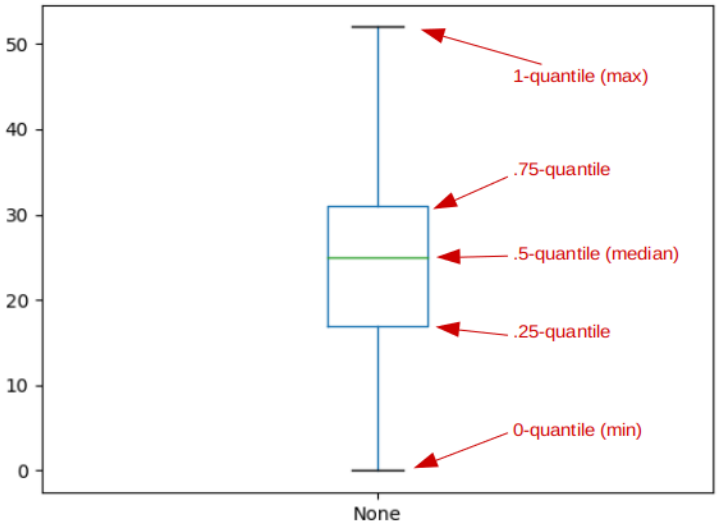


Figure 15.5: A box plot of the NCAA points data.

Now the thing to realize about box plots is that they're essentially just a graphical way of showing quartiles; or, put another way, a graphical way of showing these five quantiles:

- The 0-quantile (the minimum value) is the y-value of the bottom "whisker."
- The .25-quantile is the y-value of the bottom of the "box."
- The .5-quantile (the median) is the y-value of the horizontal line within the box.
- The .75-quantile is the y-value of the top of the "box."
- The 1-quantile (the maximum value) is the y-value of the top "whisker."

Using your quantile knowledge from section 15.2, you'll realize the following fact: *the box alone contains exactly half the data points.* This is a key insight. While the whiskers show the entire range of the data, the box shows the middle 50% of it. (And the height of the box is precisely the IQR.) This makes it very easy to grasp where the bulk of the data lies, and it reinforces the lesson we learned from the histogram on this data set (Figure 15.1 on page 161): a

big chunk of the time, teams score in the 20's.

You might object to showing an entire plot for this, since I've just revealed that it's merely a fancy way to show five numbers. And you're right, in a way. However, when we show multiple *groups* of data side-by-side, each with their own box, it becomes a particularly powerful tool. Stay tuned for that.

Outliers

What happens if we show our head-scratching YouTube data set as a box plot? You get the monstrosity in Figure 15.6.

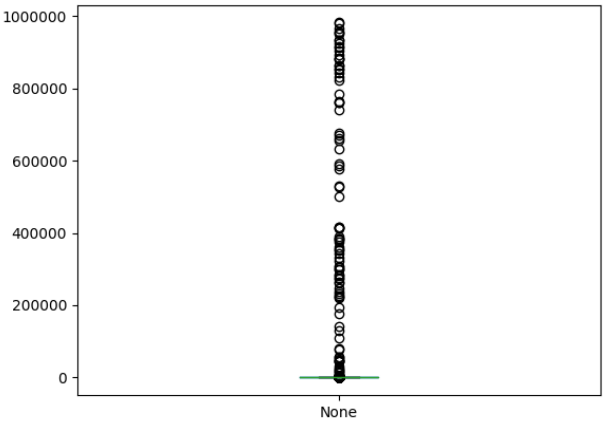


Figure 15.6: A box plot of a non-bell-curvy data set.

Geez Louise, does that look wacky. The little circles (which to me always looked like bubbles from fish breath) represent **outliers**, an important concept in data science. An outlier is basically any data point that's so far out of the normal range that it seems strange. Python is essentially flagging it for us, so we can judge for ourselves whether it was a data entry error or just a strange data point. In this case, these aren't errors – there's just a handful of videos that have been played a ton of times. And this makes the whole box plot look weird.

Notice from Figure 15.6 that *the entire box and both whiskers* have gotten smooshed at the bottom of the figure, as if crushed by the gravity of a black hole. You'll see that the top whisker doesn't *really* mean "maximum," since it's way down there in thousand-land despite the fact that we have videos with almost a million views. The top whisker truly means "the maximum *reasonable-looking* data point in the `Series`," where "reasonable-looking" is something Pandas is trying to make an educated guess about. There are ways to tweak what counts as an outlier, but my purpose here is just to get you to realize that when you have a highly skewed data set (like YouTube), prepare to see lots of things that are considered "outliers," and prepare to comb through all the mess on your box plots to try and discern the true meaning it's trying to convey.

Chapter 16

Tables in Python (1 of 3)

The third of our three aggregate data types from waaaay back in Chapter 7 was the **table**. Don't worry: we haven't forgotten about him. In this chapter, we'll implement him by means of the Pandas `DataFrame`, the most important data type in this entire book.

16.1 Reading a `DataFrame` from a `.csv` file

Unlike NumPy arrays and Pandas `Series`es, which we learned several different ways to create, we're only going to learn one way to create a `DataFrame`. That's because `DataFrame`s are normally big enough that it's just too tedious to ever type them in manually. Instead, we'll read them from an external source; a `.csv` file.

We'll actually use the same `read_csv()` function that we used in section 11.1 (p. 107), although oddly, this time we won't need to specify as many arguments. Let's say we have a "`davieses.csv`" file with these contents:

```
person,age,gender,height,instrument
Dad,50,M,73,piano
Mom,49,F,66,flute
Lizzy,21,F,63,guitar
TJ,20,M,71,trombone
Johnny,17,M,72,euphonium
```

We can read it into a `DataFrame` with this code:

```
my_first_df = pd.read_csv("davieses.csv").set_index('person')
print(my_first_df)
```

```
         age gender  height instrument
person
Dad       51      M      73      piano
Mom       49      F      66      flute
Lizzy     21      F      63     guitar
TJ        20      M      71   trombone
Johnny    17      M      72  euphonium
```

A couple things. First, you may have noticed that the `davieses.csv` file had a "header" row. This means that the first line of the file is not like the others: instead of containing information on a specific family member, it contains the *kind* of information for *every* family member. It looked like this:

```
person,age,gender,height,instrument
```

and you'll notice that these words (except for the first one; more on that in a moment) became the *column names* when we imported the data. This sort of information, by the way, is called "**metadata**," a geeky-sounding word that basically means "data about data." If "Lizzy plays the guitar" is a piece of data, then "family members play instruments" is a piece of *meta*data.

Second, don't miss the ending I tacked on to the `read_csv()` line, where I called the `.set_index()` method on the `DataFrame`. This tells Pandas that one of the columns in the `DataFrame` should be designated as the **index** (or the **keys**).

Back on p. 57 I asserted that unlike associative arrays, tables didn't have keys. And that's true of the general "table" concept. But Pandas designed their `DataFrame`s to behave in the same way as their `Series`es: one uniquely-valued column will be used to identify each row.

This choice is usually easy; if you glance back to Figure 7.3 (p. 57), we'd probably want to choose the screenname as the index (although a case could be made for the real name column instead). For the table in Figure 7.4 (p. 59), it would be the item column. In the `DataFrame` we just created above, obviously `person` is the correct choice – it's the only one sure to be unique.[1]

Anyway, designating a column as the index in this way sort of removes it from the other, "ordinary" columns. In the output, above, you may notice that the word "`person`" is printed somewhat lower than the other column names are. It turns out that if we want to talk about the index column specifically, we'll need to use a slightly different technique than we do for the other columns. More on that next chapter.

Finally, note that calling `.set_index()` is optional. It's perfectly fine to just call `pd.read_csv()` and leave it at that. In that case, Pandas will use integers (starting with 0, of course) as the index/keys.

16.2 Missing values

Let's change the example to a different family, and a slightly bigger `DataFrame`. The "`simpsons.csv`" file is reproduced below. Do you notice anything odd about it?

```
name,species,age,gender,fave,IQ,hair,salary
Homer,human,36,M,beer,74,,52000
Marge,human,34,F,helping others,120,stacked tall,
Bart,human,10,M,skateboard,90,buzz,
Lisa,human,8,F,saxophone,200,curly,
Maggie,human,1,F,pacifier,100,curly,
SLH,dog,4,M,,,shaggy,
```

What I mean is the positioning of some of the commas. The sharp-eyed reader will see a "double comma" in Homer's row. Even a

[1] With apologies to boxing legend George Foreman, who named all four of his sons "George."

dull-eyed reader will notice several commas in a row in SLH's[2] row. And nearly *every* row (the exception being Homer's) *ends* with a comma, which just looks messed up.

This weird punctuation implies the existence of **missing values**, which means just what it sounds like: there's simply no data for certain columns of certain rows. Homer doesn't have a "`hair`" value, no one *but* Homer has a "`salary`" value, and SLH is missing all kinds of stuff.

When we read this into a Pandas `DataFrame` a la:

```
simpsons = pd.read_csv("simpsons.csv").set_index('name')
```

the result looks like this:

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0           NaN  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      NaN
Bart      human   10      M      skateboard   90.0          buzz      NaN
Lisa      human    8      F       saxophone  200.0         curly      NaN
Maggie    human    1      F        pacifier  100.0         curly      NaN
SLH         dog    4      M             NaN    NaN        shaggy      NaN
```

The missing values come up as `NaN`'s, the same value you may remember from p. 114. The monker "not a number" makes sense for the `salary` case, although I think it's a bit weird for Homer's `hair` (not a number? is hair *supposed* to be a number?...) At any rate, we can expect that this will be the case for many real-world data sets.

"Missing" can mean quite a few subtly different things, actually. Maybe it means that the value for that object of study was collected, but lost. Maybe it means it was never collected at all. Maybe it means that variable doesn't really make *sense* for that object, as in the case of a dog's IQ. Ultimately, if we want to use the other values in that row, we'll have to come to terms with what the missing

[2]The Simpson's dog was named "Santa's Little Helper."

values *mean*. For now, let's just learn a couple of coarse ways of dealing with them.

One (sometimes) handy method is `.dropna()`. If you call it, it will return a modified copy of the `DataFrame` in which any row with an `NaN` is removed. This turns out to be overkill in the Simpson's case, though:

```
print(simpsons.dropna())
```

```
Empty DataFrame
Columns: [species, age, gender, fave, IQ, hair, salary]
Index: []
```

In other words, nothing's left. (Every row had at least one `NaN` in it, so nothing survived.)

We could pass an optional argument to `.dropna()` called "`how`", and set it equal to `"all"`: in this case only rows with *all* `NaN` values are removed. Sometimes that's "underkill," as in our Simpson's example: after all, none of the rows are *entirely* `NaN`'s, so calling `.dropna(how="all")` would leave everything intact.

Another option is the `.fillna()` method, which takes a "default value" argument: any `NaN` value is replaced with the default in the modified copy returned. Let's try it with the string `"none"` as the default value:

```
print(simpsons.fillna("none"))
```

```
        species  age gender            fave    IQ          hair salary
name
Homer     human   36      M            beer    74          none  52000
Marge     human   34      F  helping others   120  stacked tall   none
Bart      human   10      M      skateboard    90          buzz   none
Lisa      human    8      F       saxophone   200         curly   none
Maggie    human    1      F        pacifier   100         curly   none
SLH         dog    4      M            none  none        shaggy   none
```

This is possibly useful, but in this case it's not a perfect fit because different columns call for different defaults. The `fave` and `hair` columns could well have "`none`" (indicating no favorite thing, and no hair, respectively) but we might want the default `salary` to be 0. The way to accomplish that is to change the individual columns of the `DataFrame`. Here goes:

```
simpsons['salary'] = simpsons['salary'].fillna(0)
simpsons['IQ'] = simpsons['IQ'].fillna(100)
simpsons['hair'] = simpsons['hair'].fillna("none")
print(simpsons)
```

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0          none  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
SLH         dog    4      M             NaN  100.0        shaggy      0.0
```

Here we've assumed that the default IQ, for someone who hasn't taken the test, is 100 (the average). I left the `NaN` in `fave` as is, since that seemed appropriate.

By the way, that code is actually more than it may appear at first. When we execute a line like:

```
simpsons['salary'] = simpsons['salary'].fillna(0)
```

we're really saying "please *replace* the `salary` column of the `simpsons DataFrame` with a new column. That new column should be – wait for it – the existing `salary` column but with zeros replacing the `NaN`'s."

We'll see many more cases of changing `DataFrame` columns wholesale in the following chapters.

16.3 Removing rows/columns

Finally, after reading a `.csv` file into a `DataFrame`, there are times when you want to manually delete certain rows and/or columns that are not going to be of interest.

The easiest syntax for deleting a row (say, Santa's Little Helper) is:

```
simpsons = simpsons.drop('SLH')
```

The `.drop()` method takes the index of the undesired row as an argument, Like most of the methods we've seen so far, it returns a modified copy of the `DataFrame` it's called on, so you have to reassign this to the original variable (or use the `inplace=True` argument).

You can even delete multiple rows at the same time by enclosing the undesired indices in boxies:

```
simpsons = simpsons.drop(['Homer','Marge','SLH'])
```

Deleting a column is even more common, since many tables "in the wild" have many, many columns, only a few of which you may care about in your analysis. You can whack one entirely with the `del` operator, just like we did for `Series`es (p. 111):

```
del simpsons['IQ']
```

Chapter 17

Tables in Python (2 of 3)

It's easy to get tripped up on Pandas' syntax for accessing the individual bits of `DataFrame`s. First, let's talk about rows and columns, and then we'll talk about the atomic elements ("cells") themselves.

17.1 Accessing individual rows and columns

Suppose you have a `DataFrame` called `df`. Here's how you can extract particular rows and columns:

- `df.loc[`*i*`]` – access the **row** with **index** *i*
- `df.iloc[`*n*`]` – access **row** number *n*
- `df[`*c*`]` – access **column** *c*

The second of these is reminiscent of the `.iloc` syntax we learned for `Series`es on p. 112. With it, we specify the *number* we want, rather than the index/key/label. That's not super common to do, but it happens. More common is the first form: we specify the row we want by its index.

The last one is tricky, because everyone (including me, several times a week, it seems) assumes that just typing (say) "`df['Bart']`" would give you `Bart`'s row. This is probably how it *ought* to work, since `Series`es worked that way. Alas, no: if you specify neither `.loc` nor `.iloc`, you're asking for a *column*, not a row.

Yet another odd thing is how a single row is presented on the screen. Let's go back to the `simpsons` data set (bottom of p. 174), and access the `Bart` row the proper way (with `.loc`):

```
print(simpsons.loc['Bart'])
```

```
species          human
age                 10
gender               M
fave        skateboard
IQ                   90
hair               buzz
salary                0
Name: Bart, dtype: object
```

This bugs the heck out of me. Bart, like all other Simpsons, was a *row* in the original `DataFrame`, but here, it presents Bart's information vertically instead of horizontally. I find it visually jarring. The reason Pandas does it this way is that *each row of a* ***`DataFrame`*** *is a* ***`Series`***, and the way Pandas displays `Series`es is vertically. We'll deal somehow.

Btw, for any of the three options, you can provide a *list* with multiple things you want, instead of just one thing. You do so by using *double* boxies:

- `df.loc[[`*i1,i2,i3,...*`]]` – access the row**s** with indices *i1, i2, i3, etc.*
- `df.iloc[[`*n1,n2,n3,...*`]]` – access the row**s** numbered *n1, n2, n3, etc.*
- `df[[`*c1,c2,c3,...*`]]` – access the column**s** names *c1, c2, c3, etc.*

Examples

To test your understanding of all of the above, confirm that you understand the following examples:

```
print(simpsons.iloc[3])
```

```
species        human
age                8
gender             F
fave       saxophone
IQ               200
hair           curly
salary             0
Name: Lisa, dtype: object
```

```
print(simpsons['age'])
```

```
name
Homer     36
Marge     34
Bart      10
Lisa       8
Maggie     1
SLH        4
Name: age, dtype: int64
```

```
print(simpsons.loc[['Lisa','Maggie','Bart']])
```

```
        species  age gender        fave     IQ   hair  salary
name
Lisa     human    8      F   saxophone  200.0  curly     0.0
Maggie   human    1      F    pacifier  100.0  curly     0.0
Bart     human   10      M  skateboard   90.0   buzz     0.0
```

```
print(simpsons.iloc[[1,3,4]])
```

```
        species  age gender            fave     IQ          hair  salary
name
Marge    human   34      F  helping others  120.0  stacked tall     0.0
Lisa     human    8      F       saxophone  200.0         curly     0.0
Maggie   human    1      F        pacifier  100.0         curly     0.0
```

```
print(simpsons[['age','fave','IQ']])
```

```
         age            fave     IQ
name
Homer     36            beer   74.0
Marge     34  helping others  120.0
Bart      10      skateboard   90.0
Lisa       8       saxophone  200.0
Maggie     1        pacifier  100.0
SLH        4             NaN   30.0
```

Incidentally, you'll notice how the `name` values are treated differently from all the other columns, since `name` is the `DataFrame`'s index. For one thing, `name` *always* appears, even though it's not included among the columns we asked for. For another, it's listed at the bottom of the single-row `Series` listings rather than up with the other values in that row.

17.2 Accessing individual elements

I mentioned above the eternal truth that *each row of a `DataFrame` is a `Series`*. Once you grasp this, you'll realize that you can access an individual "cell" of a `DataFrame` simply by getting the row you want, and then getting the specific value from that. A two-step process for doing this would be:

```
lisas_row = simpsons.loc['Lisa']
lisas_iq = lisas_row['IQ']
print(lisas_iq)
```

```
200.0
```

But a shorter, one-stepper just combines these two operations on the same line:

```
lisas_iq = simpsons.loc['Lisa']['IQ']
print(lisas_iq)
```

```
200.0
```

17.3 Accessing a DataFrame's metadata

We can get some meta-information about a `DataFrame` without even looking at individual rows. If we want to know what the index values themselves are, we use `.index`:

```
print(simpsons.index)
```

```
Index(['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie', 'SLH'],
    dtype='object', name='name')
```

That weird-looking output tells us several things. First, the index of this `DataFrame` consists of `string`s (remember from p. 71 that's what "`dtype='object'`" means). Second, the *name* of the index column is, ironically, "`name`". (It could be named anything at all, of course.) Third, the actual index values are `Homer`, `Marge`, and all the rest.

That's the index, or the "row names," if you will. To get the column names, we use `.columns`:

```
print(simpsons.columns)
```

```
Index(['species', 'age', 'gender', 'fave', 'IQ', 'hair',
    'salary'], dtype='object')
```

Interestingly, this too is an "`Index`" beast, also comprised of `strings`. Pandas treats both "axes" of a `DataFrame` similarly, in that both of them are the same type of thing (an "`Index`"). Notice that `name` is not present in the column names list, because as the `DataFrame`'s index it's a different sort of thing.

How many rows does a `DataFrame` have? This is answerable by using the `len()` function again:

```
print(len(simpsons))
```

```
6
```

This is our third use of the word `len()`: it can be used to find the number of characters in a `string`, the number of key/value pairs of a `Series`, and (here) the number of rows of a `DataFrame`.

Finally, we often want to get a quick sense of how large a `DataFrame` is, both in terms of rows and columns. The `.shape` syntax is handy here:

```
print(simpsons.shape)
```

```
(6, 7)
```

This tells us that `simpsons` has six rows and seven columns. As I mentioned previously (p. 56) this is definitely not the typical case: most `DataFrames` will have many more rows (thousands or even millions) than columns (at most, dozens).

17.4 Sorting `DataFrames`

Sorting a `DataFrame` is largely like sorting a `Series`, except we have more choices: instead of just the keys and the values, we have the index and potentially *many* different columns.

The `.sort_index()` method works just like it did for `Series`es:

```
print(simpsons.sort_index())
```

```
        species  age gender            fave     IQ          hair   salary
name
Bart      human   10      M      skateboard   90.0          buzz      0.0
Homer     human   36      M            beer   74.0          none  52000.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
SLH         dog    4      M             NaN   30.0        shaggy      0.0
```

The result is rows sorted alphabetically by name. And I hate to keep repeating myself, but remember that `.sort_index()` returns a modified copy, unless you pass the `inplace=True` argument. The `ascending=False` argument is also allowed, and will sort by the index highest-to-lowest instead of lowest-to-highest.

To sort by one of the columns, we call `.sort_values()` and pass it the column name:

```
print(simpsons.sort_values('IQ')
```

```
        species  age gender            fave     IQ          hair   salary
name
SLH         dog    4      M             NaN   30.0        shaggy      0.0
Homer     human   36      M            beer   74.0          none  52000.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
```

Sometimes we want to include more than one column in the sort. Why? As a tie-breaker. Consider sorting a roster for a student club, which has `first_name` and `last_name` columns, among other things. We might want to sort the list alphabetically by last name, but for students with the same last name, we should go to the first name as a tie-breaker (so that `Angela Smith` shows up after `Velma Patterson` but before `Brad Smith`).

To do this, we pass a list of columns, instead of a single column:

```
print(simpsons.sort_values(['gender','hair','IQ']))
```

```
        species  age gender            fave     IQ          hair   salary
name
Maggie   human    1      F        pacifier  100.0         curly      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
Bart     human   10      M      skateboard   90.0          buzz      0.0
Homer    human   36      M            beer   74.0          none  52000.0
SLH        dog    4      M             NaN   30.0        shaggy      0.0
```

Here, we said "sort the rows alphabetically by `gender`. For rows with the same `gender`, use `hair` as a tie-breaker. And for rows with the same `gender` *and* the same `hair`, use `IQ` as a second tie-breaker." Glance at that output and convince yourself that it's correct.

We control the "ascendingness" of the multi-column sort by specifying a list of *each* `ascending` value, one for each column we're sorting by. Consider this:

```
print(simpsons.sort_values(['gender','hair','IQ'],
    ascending=[False,True,False]))
```

```
        species  age gender            fave     IQ          hair   salary
name
Bart     human   10      M      skateboard   90.0          buzz      0.0
Homer    human   36      M            beer   74.0          none  52000.0
SLH        dog    4      M             NaN   30.0        shaggy      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
Maggie   human    1      F        pacifier  100.0         curly      0.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
```

Now we're saying "sort *reverse* alphabetically by `gender`, breaking ties by comparing `hair` alphabetically, and breaking further ties by `reverse` sorted order by IQ."

Oh, and the `inplace=True` argument works for all these examples as well.

17.5 Summary statistics for DataFrames

Summary statistics like the mean, median, minimum/maximum, and the like, can of course all be computed on individual columns of a `DataFrame`, because each column is a `Series`:

```
print(simpsons['IQ'].median())
```

```
95.0
```

```
print(simpsons['salary'].sum())
```

```
52000.0
```

You can also, believe it or not, compute the sum/mean/max/etc on the entire `DataFrame`. This computes it on every column individually:

```
print(simpsons.mean())
```

```
age         15.500000
IQ         102.333333
salary    8666.666667
dtype: float64
```

Pandas left out the non-numeric columns (`species`, `gender`, *etc.*) and computed the mean of each of the others, giving us a `Series` containing their values.

Finally, I often find the `.describe()` method useful:

```
print(simpsons.describe())
```

```
       count   6.000000    6.000000      6.000000
       mean   15.500000  102.333333   8666.666667
       std    15.436969   56.645094  21228.911104
       min     1.000000   30.000000      0.000000
       25%     5.000000   78.000000      0.000000
       50%     9.000000   95.000000      0.000000
       75%    28.000000  115.000000      0.000000
       max    36.000000  200.000000  52000.000000
```

Neat! We get the number of values, the mean, the standard deviation, and all the quartiles for each of the numeric columns. Lots of dashboard information at a glance!

Chapter 18

Tables in Python (3 of 3)

18.1 Queries

Back in section 13.1 (p. 124), we learned how to write simple **queries** to selectively match only certain elements of a `Series`. The same technique is available to us with `DataFrame`s, only it's more powerful since there are more columns to work with at a time.

Let's return to the Simpsons example from p. 174, which is reproduced here:

```
        species  age gender            fave     IQ          hair   salary
name
Homer     human   36      M            beer   74.0          none  52000.0
Marge     human   34      F  helping others  120.0  stacked tall      0.0
Bart      human   10      M      skateboard   90.0          buzz      0.0
Lisa      human    8      F       saxophone  200.0         curly      0.0
Maggie    human    1      F        pacifier  100.0         curly      0.0
SLH         dog    4      M             NaN  100.0        shaggy      0.0
```

We can filter this on certain rows by including a query in boxies:

```
adults = simpsons[simpsons.age > 18]
```

```
       species  age gender            fave     IQ          hair   salary
name
Homer    human   36      M            beer   74.0          none  52000.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
```

As with Seriesses, we can't forget to repeat the name of the variable ("simpsons") before giving the query criteria ("> 18"). Unlike with Seriesses, we also specify a column name (".age") that we want to query.

We can also provide compound conditions in just the same way as before (section 13.1, p. 128). If we want only human children, we say:

```
kids = simpsons[(simpsons.age <= 18) & (simpsons.species == "human")]
```

```
        species  age gender        fave     IQ   hair  salary
name
Bart      human   10      M  skateboard   90.0   buzz     0.0
Lisa      human    8      F   saxophone  200.0  curly     0.0
Maggie    human    1      F    pacifier  100.0  curly     0.0
```

whereas if we want everybody who's smart and/or old, we say:

```
old_andor_wise = simpsons[(simpsons.IQ > 100) | (simpsons.age > 30)]
```

```
       species  age gender            fave     IQ          hair   salary
name
Homer    human   36      M            beer   74.0          none  52000.0
Marge    human   34      F  helping others  120.0  stacked tall      0.0
Lisa     human    8      F       saxophone  200.0         curly      0.0
```

To narrow it down to only specific columns, we can combine our query with the syntax from section 17.1 (p. 177). You see, our query gave us another (shorter) DataFrame as a result, which has the same rights and privileges as any other DataFrame. So tacking on another pair of boxies gives us just a column:

```
print(simpsons[simpsons.age > 18]['fave'])
```

```
name
Homer               beer
Marge    helping others
Name: fave, dtype: object
```

while tacking on *double* boxies gives us columns:

```
print(simpsons[simpsons.age > 18][['fave','gender','IQ']])
```

```
                  fave gender     IQ
name
Homer             beer      M   74.0
Marge  helping others      F  120.0
```

Note that in the first of these cases, we got a *Series* back, whereas in the second (with the double boxies) we got a DataFrame with multiple columns.

Combining all these operations takes practice, but lets you slice and dice a DataFrame up in innumerable different ways.

18.2 The .groupby() method

One of the most useful methods in the whole DataFrame repertoire is .groupby(). It applies when you want summary statistics (mean, quantile, max/min, *etc.*) not for the *whole* data set, but for each *subset* of the data set, where the subsets split on the values of one of the variables.

Here's an example in action. It's old news that we could find, say, the median IQ of the Simpsons family overall:

```
print(simpsons['IQ'].median())
```

```
95.0
```

But it's new news that we can do this for each gender separately, via:

```
print(simpsons.groupby('gender')['IQ'].median())
```

```
gender
F    120.0
M     74.0
Name: IQ, dtype: float64
```

We give a *categorical* variable as the argument to `.groupby()`, and specify a *numeric* variable as the column we wish to analyze. Finally, we choose the summary statistic we want (`.median()` in the above case).

All this produces a resulting ***`Series`***. Think hard: the *keys* of the resulting `Series` are the *values* of the categorical variable (in the original `Series`) that we grouped by; and the *values* of the resulting `Series`, are the results of applying the summary statistic function to each of the subsets separately.

So now, in addition to knowing that the overall Simpson family median IQ is 95, we also know that among Simpson boys and men, it's only 74, whereas among girls and women, it's an impressive 120.

Another example: let's find the maximum age for each hair style:

```
print(simpsons.groupby('hair')['age'].max())
```

```
hair
buzz            10
curly            8
none            36
shaggy           4
stacked tall    34
Name: age, dtype: int64
```

(Since there's so many different hairstyles present, Maggie turns out to be the only one whose age is not represented here.)

18.3 Looping with `DataFrames`

Just as we wrote `for` loops to iterate through the elements of a NumPy array (section 14.3) and a Pandas `Series` (section 14.4), so we can iterate through the rows of a `DataFrame`. We'll do so using the weirdly-named `.itertuples()` method.

By using `.itertuples()` in the loop header, we have available to us in the loop body a `Series` representing *the current row.* ("Current row" just means "the row we're on as we successively go through all the rows in sequence.") We can access individual elements of it by using column names and the dot (or boxie) syntax, as follows:

```
for row in simpsons.itertuples():
    print("A certain {}-year old has {} hair.".format(row.age,
        row.hair))
```

```
A certain 36-year old has none hair.
A certain 34-year old has stacked tall hair.
A certain 10-year old has buzz hair.
A certain 8-year old has curly hair.
A certain 1-year old has curly hair.
A certain 4-year old has shaggy hair.
```

I called the loop variable "`row`" because it represents a row of the `DataFrame` (duh), although you can call it anything you want to. (I could have named it "`family_member`" or "`Simpson`" instead.)

This is very intuitive. Slightly less intuitive is that if we want the index column (in `simpsons`, it's called "`name`," remember) we can't use the name of the index column. Instead, we have to literally say "`.Index`," and yes that's a *capital* '`I`'.

To illustrate:

```
for family_member in simpsons.itertuples():
    print("{} Simpson, {} years of age, has {} hair.".format(
        family_member.Index, family_member.age, family_member.hair))
```

```
Homer Simpson, 36 years of age, has none hair.
Marge Simpson, 34 years of age, has stacked tall hair.
Bart Simpson, 10 years of age, has buzz hair.
Lisa Simpson, 8 years of age, has curly hair.
Maggie Simpson, 1 years of age, has curly hair.
SLH Simpson, 4 years of age, has shaggy hair.
```

Chapter 19

Exploratory Data Analysis: bivariate (1 of 2)

In this chapter, we'll extend our EDA repertoire to cover **bivariate data**, which means studying the relationships between *pairs* of variables, rather than focusing only on one variable at a time. This is where most of the action is: you'll be awed and impressed by how much more we can dig out of a data set in this chapter.

Bivariate data analysis is especially suited to the **table**s (in Python, `DataFrame`s) from section 7.1 and chapters 16–18. This is because each column of a table is a variable that matches one-for-one with every *other* column in the table.

In the Simpsons example (p. 174), the fourth `species` value corresponds to Lisa, as does the fourth `age` value, the fourth `fave` value, the fourth `gender` value, the fourth `fave` value, the fourth `IQ` value, the fourth `hair` value, and the fourth `salary` value. This means that if we examine any two columns, we know that matching indices go together (*i.e.*, represent the same person). This implicit connection is what allows us to meaningfully examine a pair of variables.

19.1 The concept of statistical significance

Before we get to the details, we need to face head-on what is probably the single most important concept in statistics, that of **statistical significance** (or "stat sig," for short). It is so immensely important that I'm going to ask you to put down whatever snack you're eating right now, fold your hands, and pay very close attention.

All forms of bivariate analysis are variations on a single theme, namely: discovering whether or not an *association* exists between two variables. Recall from section 10.2 (p. 91) that an association means that two variables are correlated in some way: that certain values of one tend to go more often with certain values of the other. To make it concrete, let's say one of our variables is `sex` (at birth, male or female) and the other is `height` (in inches, say). We want to know: "are taller people more often male, and shorter people more often female, or is there no connection between `sex` and `height`?"

Now the first thing you think of doing, of course, is getting a **sample** (recruiting volunteers, say) of both males and females, measuring their heights, and taking the average (mean). Let's say you do that, and you come up with the following numbers:

Females – average height: 65.5 inches
Males – average height: 69.3 inches

Clearly, in your *sample*, males were on average somewhat taller – 3.8 inches taller, in fact. A careless thinker would immediately conclude: "aha! My hypothesis is confirmed. I scientifically carried out my study, and mathematically computed the results, and now here is some hard data proving the conclusion that generally speaking, men tend to be taller than women."

Are you convinced by that reasoning?

I hope you're not. Here's why. Let's change the example, and suppose that instead of height, we measured our volunteers' IQ. Taking the averages as before, we come up with these numbers:

Females – average IQ: 102.4
Males – average IQ: 98.6

In this case, the average of the females in the sample was higher than the males was. Shall we conclude that in general, women tend to be smarter than men?

Confirmation bias

If you're like most people, you'll accept that first finding as confirmation of men's tallness, and you'll reject the second finding as just a fluke of the sample. Undoubtedly, this is because you went into the question already having an opinion about the matter. You just *know in your heart* that men *do* tend to be taller than women (you've observed thousands of both sexes, in fact, and have in fact noticed that trend) whereas you know in your heart that neither sex has an advantage in intelligence (ditto). This leads you to reason as follows:

1. "Well of course my male volunteers were taller than the female ones. I've known all along that males tend to be taller in general, and this just confirms it!"
2. "Aw, c'mon, we only sampled a few people and measured their IQs. Sure, these particular women might have been a bit smarter than these particular men, but if I ran the experiment again on different volunteers, it might just as easily go the other way. It'd be silly to draw a grand conclusion from that."

Psychologists call this fallacy of reasoning "**confirmation bias**." We have a natural tendency to interpret information in a way that affirms our prior beliefs. Data that seems to contradict it, we simply talk our way out of.

Confirmation bias is one of the most insidious enemies of humankind. It leads to wrong reasoning, the entrenchment of beliefs, dangerous overconfidence, polarization, and in the worst cases, **groupthink**. When a group of people succumbs to groupthink, "orthodox" viewpoints are encouraged, while alternative viewpoints are dismissed and suppressed. Every piece of evidence that conforms

to the group's consensus belief is hailed as evidence confirming it, and evidence that contradicts it is chalked up to mere statistical anomaly.

One of many examples of this phenomenon was the CIA circa 2002: from the top to the bottom, nearly every member of the organization was *certain* that Sadaam Hussein's terrible regime in Iraq possessed weapons of mass destruction (WMDs). Later, when it was inexplicably discovered that this "fact" wasn't true after all (*after* we had made irreversible decisions based on it), analysts pored over the CIA's decision-making process to try and make sense of it. Confirmation bias was perhaps the key ingredient.

Be aware of it in your own thinking, and at all costs steer yourself away from it!

The perils of eyeballing it

Okay. Let's suppose that we've freed ourselves from confirmation bias, and we're actually looking at the numbers objectively. The first question still remains: *does* an association exist between the two variables?

Men were an average of 3.8 inches taller in our sample. That's a difference...but is it *enough* of a difference? Women were smarter by 3.8 IQ points. That's a difference...but is it *enough*?

To clarify, when we say "enough," what we mean is: "*enough of a difference to generalize our findings to the population at large.*" Here's a paradox: what we have is a sample; yet oddly, it's never the sample we actually care about. We care about what the sample tells us *about the* ***population***.

Think about it. Nobody cares whether the 14 females we surveyed are smarter on average than the 17 males we surveyed. But if we make the claim: "*in general* women are smarter than men," suddenly everybody cares. To use another example, nobody cares that 58% of the 2,000 people in our phone sample said they intend to vote for Elizabeth Warren, and only 39% said Donald Trump. But if we say "*across the country*, we predict more Warren votes than Trump votes by such-and-such margin," this is big news.

Now the first law to beat into your head is that *you absolutely cannot reliably eyeball it.* This is what everyone who hasn't taken a Data Science or Statistics class tries to do. They squint at the difference (3.8 IQ points, *e.g.*) and bite their lip and mutter, "well, that sure *seems* (or *doesn't seem*) like a pretty big difference. I'll bet this says (or doesn't say) something about intelligence among the sexes in general."

Stop. You cannot. People are demonstrably very bad at judging whether or not a difference between groups is "enough." Part of the problem is that the answer to the question turns on three separate things: how big the difference is, how large your sample size is, and – importantly – how variable the data is (meaning, how widely the points you sample differ from each other). All three of these factors need to be mixed into a soup in just the right way in order to properly judge, and human intuition is just flat terrible at doing that.

So eyeballing is a non-starter. But happily, it turns out that statistics provides us an iron-clad, dependable, quantitative, take-it-to-the-bank method for determining whether the pattern in a data set is "enough" to justifiably claim an association between variables. And that is the concept of statistical significance.

A "statistically significant difference"

The correct way to determine whether a difference is "enough" is to use the appropriate **statistical test**. A statistical test is a standard procedure for incorporating all three factors I previously mentioned (the degree of difference, the sample size, and the amount of variance among data points) to come up with a principled, defensible answer to this question: is the pattern I see in my sample a *statistically significant* one? Can we be reasonably confident that it will also be true of the population as a whole?

All the statistical tests we'll learn (in the next chapter) have a common output: a ***p*-value**. That's nice because you don't have to memorize a lot of different rules for interpreting a lot of different tests.

So what the heck is a "p-value?" There have been controversies galore[1], and even entire books[2] devoted to the subject, which means that whatever I choose to write here can be nitpicked by statisticians a dozen different ways.

That's okay. I'm going to write it anyway, and this will serve you very well. Here goes:

A p-value is a number between 0 and 1. If you run a statistical test on your bivariate data, and the p-value is ***less than*** your α ("alpha") value (normally, .05), then there ***is*** a statistically significant association between the variables. Otherwise, there isn't. End of story.

Recall from section 10.5 (p. 102) that α is "where to set the bar" to detect a meaningful association. It's essentially how often we're willing to draw a false conclusion. For social science data (that is, data involving humans), you should always choose .05 to avoid controversy. For physical science data, you should always choose .01.

The bottom line is this: if you spot a possible relationship between two of your variables (like gender and IQ), run the appropriate statistical test (see next chapter) and look at the p-value. If it's less than α, then the difference you thought you saw officially *is* "enough." You can therefore declare "yep, these two variables *are* associated, to a confidence level of α." If it's not less than α, then even though you thought you saw a meaningful tendency in the data, you can officially say, "nope, it's not a stat sig diff. This is very likely just an artifact of the particular data points I collected in my sample. Pay of no mind."

[1] For instance:

- Colquhoun D (2017). "p-values." *Royal Society Open Science.* **4**(12): 171085.
- Murtaugh, Paul A. (2014). "In defense of p-values." *Ecology.* **95**(3): 611–617
- Wasserstein, Ronald L.; Lazar, Nicole A. (2016). "The ASA's Statement on p-Values: Context, Process, and Purpose." *The American Statistician*, **70**(2): 129–133

[2] Vickers, A. (2009). *What is a p-value anyway?* Boston: Pearson.

19.2 Moving on

Which statistical test is appropriate depends on your two variables' scales of measure: in particular, whether they are categorical or numeric. There are three scenarios for bivariate analysis: two categorical variables, two numeric variables, or one of each. In the next chapter, in addition to learning how to meaningfully plot all three cases, we'll learn how to run and interpret the statistical test applicable to each case, in order to determine once and for all whether "enough" is *enough*.

Chapter 20

Exploratory Data Analysis: bivariate (2 of 2)

20.1 Three bivariate scenarios

As we saw with univariate data in chapter 15, different kinds of plots and statistics are appropriate depending on the variable's scale of measure – categorical or numeric. There are thus three different cases for bivariate analysis:

- Two categorical variables
- One categorical variable and one numeric variable
- Two numeric variables

We'll consider each case in turn. Throughout all the remaining sections, we'll use this fictitious data set, called `people`:

```
   gender  salary   color  followers
0    male   54.94  purple         26
1  female   72.48  purple         22
2    male    9.47    blue         27
3   other   60.08     red         22
4    male   37.62     red         13
                .
                .
```

Each row represents one fictional person we interviewed, and in-

cludes their `gender`, their `salary` (in thousands of dollars per year), their favorite `color`, and the number of `followers` they have on some unspecified social media website.

The `DataFrame` has 5000 rows, and no special "index" variable: none of the columns that we collected are unique, so we just let Pandas default to indexing the rows by number, 0 through 4,999.

20.2 Importing `scipy.stats`

All of the statistical tests we'll demonstrate in this chapter come from the **SciPy** Python package (pronounced "sigh pie.") SciPy is huge, and has several different parts; for the time being, we'll only be using the "`stats`" component. Therefore, we need one additional import statement:

```
import scipy.stats
```

You can include this in a cell at the top of your Jupyter Notebook just like your `numpy` and `pandas` imports.

20.3 Two categorical variables

Okay. Let's return our attention to the `people` `DataFrame`, and begin with a bivariate analysis of the `gender` and `color` columns. The first thing we should do, of course, is inspect each one individually, using `.value_counts()` and perhaps a bar chart from sections 15.1 and 15.4. Let's say we've done that.

The next obvious question: is there an *association* between the two variables? In other words, are there particular values of one that tend to go with particular values of the other? In still other words, do people of different genders tend to have different favorite colors?

Contingency tables

The first tool to get at this question is called a **contingency table**. This is very much like `.value_counts()`, but for two variables instead of one. Our function is `crosstab()` from the Pandas package: if we give it two columns as arguments, it computes the complete set of counts from all possible combinations of variables. Here's what it looks like:

```
pd.crosstab(people.gender, people.color)
```

```
color   blue  green  pink  purple  red  yellow
gender
female   240    402   665     644  289     378
male    1403      0     0     248  463     258
other      1      2     2       2    1       2
```

Interpreting this is straightforward. Every cell in the matrix tells us how many people had a particular gender and a particular favorite color. For instance, there were 378 females who named `yellow` as their favorite color, and no males at all chose `green`.

Plotting two categorical variables

So now we have a table of counts – how to turn this into a pretty and informative plot?

Unfortunately, there doesn't seem to be any great way to do this. There's something called a "mosaic plot" which attempts it, but they're not very easy to visually interpret. Another option is a "heat map," which essentially reproduces the above table as squares in a grid, with each square color coded on a continuum by its height (for instance, low numbers might be dark blue and high numbers bright yellow, with a rainbow spectrum of number in between). That's sort of okay, but to be honest I prefer to just look at the numbers.

The χ^2 test

The statistical test to use for two categorical variables is called the χ^2 **test** (pronounced "kai-squared," not "chai-squared," by the way). To run it, it's convenient to first store the contingency table itself as a variable. I'll call it `gender_color` since it's a table of the genders of people and their favorite colors:

```
gender_color = pd.crosstab(people.gender, people.color)
```

Now, we run the test by calling the `chi2_contingency()` function from SciPy:

```
scipy.stats.chi2_contingency(gender_color)
```

```
(2125.8933435, 0.0, 10, array([[8.60798e+02, 2.11534e+02,
    3.49241e+02, 4.68098e+02, 3.94270e+02, 3.34056e+02],
   [7.79913e+02, 1.91657e+02, 3.16424e+02, 4.24113e+02,
    3.57223e+02, 3.02667e+02],
   [3.28800e+00, 8.08000e-01, 1.33400e+00, 1.78800e+00,
    1.50600e+00, 1.27600e+00]]))
```

I know, I know: that output is downright hideous. Here's the deal, though: all you have to do is look at *the second number* in that long, banana-and-boxie-laden thing. **The second number is the p-value.** It is 0.0. This is obviously lower than .05 (our α), and therefore, *we can conclude that `gender` and `color` are associated.*

All the other stuff in that output are fine-grained details that statisticians like to pore over. For us, the only thing we need to see from a χ^2 (or any other) test is the p-value.

20.4 One categorical and one numeric variable

Next let's consider the case where we want to test for an association between one categorical and one numeric variable. This is the "`gender` vs. `IQ`" scenario from the last chapter. In the `people` example, we might look at `gender` vs. `salary` to whether different genders earn different amounts of money on average.

Grouped box plots

The best plot for this scenario (IMHO) is the **grouped** box plot. It's the same as the chapter 15 box plots, except that we draw a different box (and pair of "whiskers") for each group.

Here's the command in Pandas:

```
people.boxplot('salary',by='gender')
```

This produces the plot on the left-hand side of Figure 20.1. Refer back to section 15.6 (p. 165) for instructions on how to interpret each part of the box and whiskers. From the plot, it doesn't look like there's much difference between the `male`s and `female`s, although those identifying with neither gender look perhaps to be somewhat of a salary disadvantage.

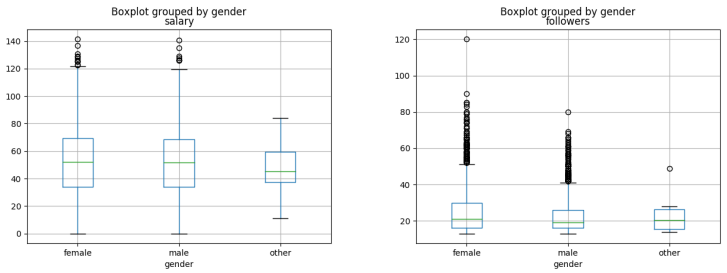


Figure 20.1: Grouped box plots of salary (left) and number of social media followers (right), grouped by gender.

Similarly, we get the plot on the right-hand side with this code:

```
people.boxplot('salary',by='followers')
```

This looks more skewed (`female`s appear to perhaps have more followers on average than `male`s), but of course we won't know for sure until we run the right statistical test.

The *t*-test

The test we'll use for significance here is called the ***t*-test** (sometimes "Student's *t*-test") and is used to determine whether the *means* of two groups are significantly different.[1] Remember, we can get the mean salary for each of the groups by using the `.groupby()` method:

```
people.groupby('gender')['salary'].mean()
```

```
gender
female    52.031283
male      51.659983
other     48.757000
```

Females have the edge over males, 52.03 to 51.66. Our question is: is this "enough" of a difference to justify generalizing to the population?

To run the *t*-test, we first need a `Series` with just the `male` salaries, and a different `Series` with just the `female` salaries. These two `Series`es are *not* usually the same size. Let's use a query to get those:

[1] Strictly speaking, the *t*-test assumes that the data sets you're comparing are "bell curvy" (or "normally distributed," to be precise) and we haven't checked for that here. However, since we're doing *exploratory* data analysis (not drawing up and documenting final conclusions) it's common to use a *t*-test as a quick-and-dirty just to see what's worth investigating.

```
female_salaries = people[people.gender=="female"]['salary']
male_salaries = people[people.gender=="male"]['salary']
```

and then we can feed those as arguments to the `ttest_ind()` function:

```
scipy.stats.ttest_ind(female_salaries, male_salaries)
```

```
Ttest_indResult(statistic=0.52411385896, pvalue=0.60022263724)
```

This output is a bit more readable than the χ^2 was. The second number in that output is labeled "`pvalue`", which is over .05, and therefore we conclude that *there is no evidence that average salary differs between males and females.*

Just to complete the thought, let's run this on the `followers` variable instead:

```
female_followers = people[people.gender=="female"]['followers']
male_followers = people[people.gender=="male"]['followers']
scipy.stats.ttest_ind(female_salaries, male_salaries)
```

```
Ttest_indResult(statistic=9.8614730266, pvalue=9.8573024317e-23)
```

Warning! When you first look at that p-value, you may be tempted to say "9.857 is *waaay* greater than .05, so I guess this is a 'no evidence' result as well." Not so fast! If you look at the entire number – including the ending – you see:

`9.857302431746571e-23`

that sneaky little "`e-23`" at the end is the kicker. This is how Python displays numbers in **scientific notation** The "`e`" means "times-ten-to-the." In mathematics, we'd write that number as:

$$9.857302431746571 \times 10^{-23}$$

which is:

.00000000000000000000000009857302431746571

Wow! That's clearly waaay *less than* .05, and so we can say *the average number of followers does depend significantly on the gender.*

Be careful with this. It's an easy mistake to make, and can lead to embarrassingly wrong slides in presentations. ☺

More than two groups: ANOVA

By the way, the t-test is only appropriate when your categorical variable has *two* values (male vs. female, for example, or vaccinated vs. non-vaccinated). If there are more than two, the appropriate test to run is called an "ANOVA" (ANalysis Of VAriance). It's beyond the scope of this text, but it's described in any intro stats textbook and is eminently Googleable.

20.5 Two numeric variables

Finally, we have the case where both variables are numeric. The `salary` and `followers` columns are this case. Are they associated?

Scatter plots

The correct plot to visualize this is the **scatter plot**. It has an axis for each numeric variable, and plots one dot (or other marker) for each object of study: its x/y coordinates depend on that object's value for each variable.

The Pandas code is as follows:

```
people.plot.scatter(x='followers',y='salary')
```

which produces Figure 20.2. Interestingly, there appear to be a lot of people pegged at zero salary, and also at 10-ish followers. (These observations would have shown up in a univariate analysis as well.) There's no super obvious connection between the two variables, but

if you squint at the plot it (maybe) looks like there's a slight up-and-to-the-right trend, which would indicate that having more followers is modestly associated with earning more money.

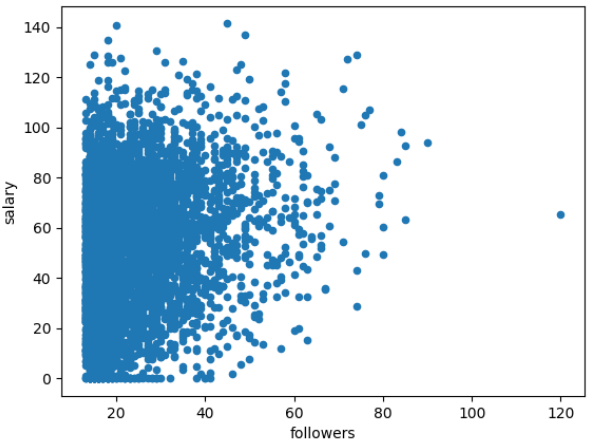


Figure 20.2: A scatter plot of `followers` vs. `salary`. Each point in the plot represents one person, with the x and y coordinates corresponding to his/her/their number of followers and salary.

20.6 Pearson correlation coefficient

The test we'll use to see whether this pattern is significant is the **Pearson correlation coefficient** (also called "Pearson's r"). To run it, we call SciPy's `pearsonr()` function and pass it the two columns:

```
scipy.stats.pearsonr(people.salary, people.followers)
```

```
(0.2007815176819964, 1.2285885030618397e-46)
```

We're given two numbers as output. The *second* of these is the p-value, and remembering our pitfall from above, we're savvy enough to notice the `e-46` at the end and declare it significant. So we can

say *we have high confidence that a person's salary is associated with their number of social media followers.*

Now for the first number, which is the actual "correlation coefficient." *If* the second number is below α and therefore significant (as it was here), you then look at the first number and see whether it's positive or negative. Positive numbers indicate **positive correlations**: an increase in one of the variables corresponds to an *increase* in the other. Negative numbers indicate **negative correlations**: an increase in one of the variables corresponds to a *decrease* in the other. Here, we have a positive number, which means that *having more followers tends to go with a higher salary.*

As an example of the second (negative) case, suppose two of our variables in a data set of sailboat races were `length` (the length of the sailboat, from bow to stern) and `finish_time` (the number of minutes the boat took to complete the race). We're likely to see a negative correlation in this case, because physics tells us that *longer* boats can travel through the water *faster* (and therefore have *lower* finish times). These two variables would thus be correlated, but in a negative way: a high value for one would typically indicate a *low* value for the other.

Chapter 21

Branching

In this chapter, we'll learn our next programming trick: how to execute code *conditionally*. This is called **branching**. It's another variant of non-linear programming, like the loops from chapter 14: it enables something other than the strict, start-to-finish, line-by-line execution of a program. In particular, branching allow us to designate certain lines of code to be executed "only sometimes."

21.1 The `if` statement

The main branching statement in Python and most languages is the `if` **statement**. Here's a couple of them in action:

```
 1: name = "Horace"
 2: cash_on_hand = 100000
 3: IQ = 90
 4: print("Nice to meet you, {}!".format(name))
 5: if cash_on_hand > 5000:
 6:     print("Wow, you're rich! Gimme a fiver.")
 7:     cash_on_hand = cash_on_hand - 5
 8: if IQ > 100:
 9:     print("Wow, you're smart! Read a book.")
10:     IQ = IQ + 5
11: print("{}'s IQ is {} and he has ${}.".format(name,
12:     IQ, cash_on_hand))
```

Even without any explanation, you might be able to figure out that the output of the code snippet above is:

```
Nice to meet you, Horace!
Wow, you're rich! Gimme a fiver.
Horace's IQ is 90 and he has $99995.
```

If not, stay tuned.

Just like a `for` loop, every `if` statement has a header and a body. And just like a `for` loop, the determining factor of which lines constitute the body depends on the indentation:

- ☞ The first `if` statement's header is line **5**.
- ☞ The first `if` statement's body is lines **6 and 7**.
- ☞ The second `if` statement's header is line **8**.
- ☞ The second `if` statement's body is lines **9 and 10**.

When an `if` statement is reached, its **condition** is evaluated; in the first case, the condition "`cash_on_hand > 500`" is evaluated to `True`, and in the second case, "`IQ > 100`" is determined to be `False`. Then, *only if* the condition is true will the body of the `if` statement execute. Otherwise, it'll be skipped over.

Thus, the lines of the above program execute in this order: 1, 2, 3, 4, 5, 6, 7, 8, 11/12. Lines 9 and 10 are skipped entirely, since Horace's IQ wasn't above average. Observe that the `cash_on_hand` variable was updated inside the body of the first `if` statement, but that `IQ` was not.

Compound conditions

Conditions can be more complicated than the ones above; just as with queries (p. 128) they can contain more than one component:

```
if cash_on_hand > 10000 and IQ < 50:
    print("Wow, some dumb people are sure rich!")
```

You might have been surprised to see the word "`and`" in that `if` statement instead of the character "`&`". I feel you. It's totally inconsistent, but nevertheless true: although in a query, you must use the symbols `&`, `|`, and `~`, in an `if` condition, you must use the words `and`, `or`, and `not`. (In other news, the bananas around the components of an `if` condition aren't necessary, but you can include them if you want.)

For your convenience, the `if` condition operators are listed in Figure 21.1. (**Remember** the double-equals!!)

21.2 The `if/else` statement

The above examples execute the `if` body as long as the condition is true, and do nothing otherwise. It's common to want to do something else in the "otherwise" case instead, and for that, we have the `if/else` statement.

```
name = "Gladys"
cash_on_hand = 2000
IQ = 120
print("Nice to meet you, {}!".format(name))
if cash_on_hand > 5000:
    print("Wow, you're rich! Gimme a fiver.")
    cash_on_hand = cash_on_hand - 5
else:
    print("I wish you well!")
if IQ > 100:
    print("Wow, you're smart! Read a book.")
    IQ = IQ + 5
else:
    print("You're currently not that smart. Read a book!")
    IQ = IQ + 10
print("{}'s IQ is {} and she has ${}.".format(name, IQ,
    cash_on_hand))
```

```
Nice to meet you, Gladys!
I wish you well!
Wow, you're smart! Read a book.
Gladys's IQ is 125 and she has $2000.
```

You can see that "`I wish you well!`" was printed. This is because `cash_on_hand` was *not* greater than 5000 (as required by the `if` condition). Also, the "...`you're smart!`..." message was printed but not the "...`not that smart`..." one. Both the `if` part and the `else` part have an indented body, although only the `if` part has a condition.

And that brings up another point. Although it hardly seems worth mentioning, let me nevertheless emphasize this oft-overlooked truth:

Whenever an `if`/`else` statement is reached, ***either* the `if` body *or* the `else` body will always be executed. It's never both, and it's never neither one.**

This is always, always true, because of the nature of things. The reason the `else` header doesn't have a condition is because its condition is implicitly *the exact opposite* of the `if` condition. Period. In *any* case where the `if` condition isn't true – and *only* in such a case – will the **else** condition be executed.

To test whether you fully understand this point, see if you can predict the output of the following program, which 99% of beginning programmers get wrong:

```
name = "Javier"
lang = "French"

if lang == "Spanish":
    print("Hola, {}!".format(name))
if lang == "French":
    print("Bonjour, {}!".format(name))
if lang == "Chinese":
    print("Ni hao, {}!".format(name))
else:
    print("Hello, {}!".format(name))
```

Seriously, don't feel bad if you miss this one. The answer (*drum roll please*) is:

```
Bonjour, Javier!
Hello, Javier!
```

Wait...wut? Why did it print two messages? Surely if `Javier`'s preferred language is `French`, it ought to say "`Bonjour`" and skip all the other options?

To understand this behavior, you have to realize that an `if`/`else` statement is a *single* entity. This multi-lingual greeting program has three components:

1. an `if` statement
2. an `if` statement
3. an `if`/`else` statement

And you must remember our golden rule in the shaded box: ***either* the `if` body *or* the `else` body will always be executed: never both, and never neither one.** Therefore, the above program does this:

1. If the language is Spanish, print an "Hola" message. (Otherwise, do nothing.)
2. If the language is French, print a "Bonjour" message. (Otherwise, do nothing.)
3. If the language is Chinese print a "Ni hao" message. Otherwise, print a "Hello" message.

Once you recognize that structure, you'll realize that when step 3 is encountered, the program *must* print either "Ni hao" or "Hello." It can't print both, and it can't print neither. An `if`/`else` just doesn't work any other way.

21.3 The `if`/`elif`/`else` statement

The problem with the previous example is that we really want our four languages to be **mutually exclusive** options. If `lang` is `"Spanish"`, we want it to print "`Hola`" and skip all the rest. The easiest way to get this behavior is to use "`elif`" (a horrid abbreviation of the phrase "else if").

Operator	Meaning
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`!=`	*not* equal to
`==`	equal to
`and`	and
`or`	or
`not`	not

Figure 21.1: Operators in `if` statements: simple and compound.

Squint hard at this program until you see the differences between it and the previous one:

```
name = "Javier"
lang = "French"

if lang == "Spanish":
    print("Hola, {}!".format(name))
elif lang == "French":
    print("Bonjour, {}!".format(name))
elif lang == "Chinese":
    print("Ni hao, {}!".format(name))
else:
    print("Hello, {}!".format(name))
```

It's identical except that we replaced the second two `if`'s with `elif`'s. This tells Python: only if the language is *not* Spanish should you then consider whether or not it's French. And only if it's *not* French (and *not* Spanish) should you consider whether or

not it's Chinese. And only if it's not Chinese (and not French (and not Spanish)) should you print "Hello."

Realize, too, that an entire `if`/`elif`/`elif`/.../`elif`/`else` chain is a *single* statement, no matter how many conditions it has. You can't just have an "`elif`" (or an "`else`," for that matter) floating out in the void without an initial `if` to anchor it. This may help you to understand how the `elif` structure acts, and why it will only ever execute *one* of the bodies. It's because:

Whenever an `if`/`elif`/.../`elif`/`else` statement is reached, **exactly *one* of the bodies will be executed. It's never more than one, and it's never none.**

Whether to use sequential `if`s or a chain of `elif`s isn't always an easy question to answer. Neither choice is always right: you have to think rigorously logically about how the program should act. Ask yourself: "do I want Python to consider *all* of these conditions – and execute the appropriate `if` bodies – no matter what? Or do I want it to bail out as soon as it finds one that's true?" Like most things, it takes practice to get right.

21.4 Nesting

Now as if this weren't complex enough, let me inform you that the body of an `if` (or `else`, or `elif`) statement can *itself* contain other `if` statements! This is actually quite common. Consider this example:

```
first_name = "Emma"
last_name = "Watson"
gender = "female"
marital_status = "single"
degree = "BA"

if degree == "PhD" or degree == "MD":
    print("Why hello, Dr. {}!".format(last_name))
elif gender == "male":
    print("Why hello, Mr. {}!".format(last_name))
elif gender == "female":
    if marital_status == "married":
        print("Why hello, Mrs. {}!".format(last_name))
    elif marital_status == "single":
        print("Why hello, Miss {}!".format(last_name))
    else:
        print("Why hello, Ms. {}!".format(last_name))
else:
    print("Why hello, Mx. {}!".format(last_name))
```

```
Why hello, Miss Watson!
```

This program implements the quite convoluted social norms for salutations in Western culture. Consider it carefully. If a person is either a Ph.D. or a Medical Doctor, that trumps everything, and we use "Dr." as their form of address. This is to indicate just how studly these people are.

If they don't have such a college degree – and *only* if they don't (notice the `elif`) – will we then consider their gender. For men, it's simple: a plain old "Mr." will do. For women, it's complicated: their marital status now comes into play. This is the *nested* part of the structure: we have another `if` statement (a whole `if`/`elif`/`else` chain, actually) *inside* the "`gender == "female"`" condition. If the person in question identifies as neither `male` nor `female`, all that Mrs./Miss/Ms. stuff will be skipped, and we'll drop straight to the `Mx.` message.

Pay careful attention to the indentation in these examples, since it's the key to discerning the structure of the program.

21.5 Combining branching with loops

And now for the really important application of branching in data science programs: combining it with *loops*.

Just as we can have an `if` statement (or `if`/`else`, or `if`/`elif`/`else`) inside an `if` body, so we can have an `if` statement (and friends) inside a *loop* body. This is where we're going to get a lot of mileage.

Let's return to Springfield. Our `simpsons DataFrame` from p. 174 looked like this:

```
         species  age gender            fave     IQ          hair   salary
name
Homer      human   36      M            beer   74.0          none  52000.0
Marge      human   34      F  helping others  120.0  stacked tall      0.0
Bart       human   10      M      skateboard   90.0          buzz      0.0
Lisa       human    8      F       saxophone  200.0         curly      0.0
Maggie     human    1      F        pacifier  100.0         curly      0.0
SLH          dog    4      M             NaN  100.0        shaggy      0.0
```

Now an ordinary loop could print (say) the name and favorite things of all the characters:

```
for row in simpsons.itertuples():
    print("{} likes {}.".format(row.Index, row.fave))
```

```
Homer likes beer.
Marge likes helping others.
Bart likes skateboard.
Lisa likes saxophone.
Maggie likes pacifier.
SLH likes nan.
```

But combining this with branching techniques gives us more power. We could, for example, print only the females:

```
for row in simpsons.itertuples():
    if row.gender == "F":
        print("{} likes {}.".format(row.Index, row.fave))
```

```
Marge likes helping others.
Lisa likes saxophone.
Maggie likes pacifier.
```

or give different messages for different age ranges:

```
for row in simpsons.itertuples():
    if row.age >= 18:
        print("{} earns ${} outside the home.".format(row.Index,
            row.salary))
    else:
        print("Aw, {}'s just a kid.".format(row.Index))
```

```
Homer earns $52000.0 outside the home.
Marge earns $0.0 outside the home.
Aw, Bart's just a kid.
Aw, Lisa's just a kid.
Aw, Maggie's just a kid.
Aw, SLH's just a kid.
```

or combine these things in innumerable ways:

```
for row in simpsons.itertuples():
   if row.species == "human":
      if row.IQ > 115:
         print("We'd like to nominate {} for a Nobel prize.".format(
            row.Index))
      elif row.IQ >= 90:
         print("You can trust {} with a {}, or even a knife.".format(
            row.Index, row.fave))
      else:
         print("Hmm. No comment.")
      if row.salary > 0:
         print("The {}-year-old {} is gainfully employed.".format(
            row.age, row.Index))
   else:
      print("Hey...{} is some kind of animal!".format(row.Index))
```

```
Hmm. No comment.
The 36-year-old Homer is gainfully employed.
We'd like to nominate Marge for a Nobel prize.
You can trust Bart with a skateboard, or even a knife.
We'd like to nominate Lisa for a Nobel prize.
You can trust Maggie with a pacifier, or even a knife.
Hey...SLH is some kind of animal!
```

You get the idea. Using a loop, we can successively consider each element of an array/`Series` or the rows of a `DataFrame`. Using `if` and friends, we can treat each one differently depending on its characteristics. The possibilities are endless!

Chapter 22

Functions (1 of 2)

And now for the very last "pure programming" lesson of the book: writing **function**s. This is more or less the final tool in the programmer's toolkit, and as I've learned over my years of teaching, it often causes the most trouble.

Now you might be thinking, "hey waitaminit, we've known about functions since all the way back on p. 20. This is something new?" Yes it is. Previously in this book, we've done a lot of *calling* functions – from simple ones like `len()` and `np.append()` to complex ones like `pd.read_csv()` to `scipy.stats.chi2_contingency()` – that someone else has written for us. By contrast, in this chapter, we look behind the curtain and join the production staff: we write our *own* functions.

22.1 Why do all this

It turns out there's a lot of syntactic nonsense involved to get all the wiring right when you do this. It can cause students to pull their hair out. So it's worth asking at the outset: what do we get for all this pain?

The answer is subtle, and can seem underwhelming at first, but it's crucial. It essentially boils down to this lesson: any complex creative work (including a computer program) should be **modular** in its design. This means that it should be composed of smaller

building blocks, which are in turn composed of still smaller building blocks. The entire thing should comprise an organized whole.

Any other way of doing it leads to madness.

Think of a car engine. When a mechanic opens up the hood, he or she doesn't see just one big monolithic thing called "The Engine," but rather piston assemblies, spark plugs, water pumps, drive shafts, and lots of other subsystems. It's what allows piece by piece investigation of problems, and piece by piece replacement of bad parts.

Or think of a rock 'n' roll tune. It's not just an impenetrable mass of sound. Instead, it's a collection of recognizable bass lines, drum sequences, vocal patterns, and variations on common guitar riffs. I don't mean to minimize the creativity involved in its orchestration; in fact, the novel combination of the myriad possible building blocks *is* the creativity. If the song were just an impermeable wall of sound, it would be noise, not music.

In the same way, once your data analysis code approaches a certain size, it really must be written in a modular way or it will become a hopelessly tangled mess, what programmers refer to as "spaghetti code." And the way to achieve this is by writing it in terms of functions that you then call at the appropriate time.

One other advantage to this approach, by the way, is that functions are **reusable**. Think of how many programmers all over the world have had reason to call `np.sort()`, or `scipy.stats.pearsonr()`! The same function becomes applicable in a variety of different contexts, so that nobody has to reinvent the wheel.

22.2 The `def` statement

Okay, down to brass tacks. The way to create (not call) a function in Python is to use the `def` statement. For our first example, let's write a function to compute an (American) football team's score in a game:

```
def football_score(num_tds, num_fgs):
    return num_tds * 7 + num_fgs * 3
```

For those not familiar with football scoring, each "touchdown" (or TD for short) a team scores is worth seven points, and each "field goal" (or FG) is worth three. (For those who *are* familiar with football scoring, please forgive the simplifications here – extra point kicks, safeties, *etc.* It's a first example.)

As you can see from the code snippet, above, the word `def` (which stands for "define," since we're "defining" – a.k.a. writing – a function) is followed by the *name* of our function, which like a variable name can be any name we choose. After the name is the list of **argument**s to the function, in bananas.

All that is the **header** of the function. The **body**, like other "bodies" we've seen (p. 137, p. 212) is *indented* underneath. The `football_score` body is just one line long, but it can be as many lines as necessary.

Finally, we see the word "`return`" on that last line. This is how we control the **return value** which is given back to the code that called our function (review section 5.3 on p. 38 if you need a refresher on this). Whenever a `return` statement is encountered during the execution of a function, the function immediately *stops* executing, and the specified value is handed back to the calling code. More on that in a minute.

22.3 Writing vs. calling

Now here's one of the most perplexing things for beginners. Consider this code:

```
team_name = "Broncos"
num_tds = 3
num_fgs = 2

def football_score(num_tds, num_fgs):
    return num_tds * 7 + num_fgs * 3
```

It surprises many to learn that this code snippet *does not compute anything*, football score or otherwise. The reason? We only *wrote* a function; we didn't actually *call* it.

This is sort of like building an impressive machine but then never pushing the "On" button. The above code says to do four things:

1. Create a `team_name` variable and set its value to the string `"Broncos"`.
2. Create a `num_tds` variable and set its value to the integer 3.
3. Create a `num_fgs` variable and set its value to the integer 2.
4. Create a function called `football_score` which, *if it is ever called in the future*, will compute and return the score of a football game.

In other words, that last step is just preparatory. It tells Python: "by the way, in case you see any code later on that calls a function named '`football_score`,' here's the code you should run in response."

To actually call your function, you have to use the same syntax we learned on p. 20, namely:

```
team_name = "Broncos"
num_tds = 3
num_fgs = 2

the_score = football_score(num_tds, num_fgs)
print("The {} scored {} points.".format(team_name,
    the_score))
```

```
The Broncos scored 27 points.
```

Follow the thread of execution closely here. First, the three variables are created, in what I'll often call "the main program." By "main," I really just mean the stuff that's all the way flush-left, and thus not inside any "`def`." It's the main program in the sense that when you execute the cell, it's what immediately happens without needing to be explicitly called.

Then, after those three variables are created, the `footbal_score()` function is called, at which point *the flow of execution is transferred to the inside of the function.* Since this simple function has only one line of code in its body (the `return` statement), executing it is really quick; but it's still important to realize that for a moment, Python isn't "in" that Broncos cell at all. Instead it jumps to the function, carries out the code inside it, and then `return`s the value...

...right back into the waiting arms of the main program, which stores that returned value (an integer 27, as it turns out) in a new variable named `the_score`. Then the flow continues, and the `print()` statement executes as normal.

Bottom line: every time you want to run your function's code – whether that's a hundred times, once, or not at all – you need to call it by typing the name of the function (with no "`def`") followed by a banana-separated list of arguments.

22.4 Naming arguments

Speaking of arguments, here's the next thing many students have trouble with. The names of the variables that your main program passes to the function are normally *not* the same as the arguments defined by the function itself.

What? Yeah.

Consider this example:

```
jets_touchdowns = 1
jets_field_goals = 3
jets_total = football_score(jets_touchdowns, jets_field_goals)

colts_tds = 1
colts_fgs = 0
colts_total = football_score(colts_tds, colts_fgs)

print("The Jets won {} to {}.".format(jets_total, colts_total))
```

```
The Jets won 16 to 7.
```

This code contains two calls to the `football_score()` function. In the first call, the variables `jets_touchdowns` (with value 2) and `jets_field_goals` (0) were passed. In the second, `colts_tds` (1) and `colts_fgs` (3) were. In *neither* case were the arguments literally named "`num_tds`" and "`num_fgs`", which were the function's own argument names.

To be crystal clear: whenever `football_score()` is called, two arguments are passed to it. The function chooses to name the first one it receives "`num_tds`" and the second one it receives "`num_fgs`". But these are *its own personal names.* They normally have nothing to do with what the calling code chooses to name them.

Why does it work this way? Perhaps this example makes clear the reason. If the main program had to name its variables exactly the same as the (indented) function did, then the function would not be reusable. In order for it to be called with different values in different contexts, there needs to be this flexible decoupling of variable names.

To reinforce the lesson, note too that you can call a function without even having variables in the main program at all:

```
x = football_score(5, 2)
print("Some mythical team scored {} points today.".format(x))
```

```
Some mythical team scored 41 points today.
```

Here we literally passed the values 5 and 2 to the function, instead of creating variables to hold them. The function doesn't mind: it just says, "hey man, whatever's given to me in slot #1, I'm going to name '`num_tds`,' and whatever's delivered through slot #2, I'm going to call '`num_fgs`.' I don't care what the outside world's variable names are...or even whether they have names at all. I just work here."

22.5 Passing aggregate data to functions

Even though the previous example involved passing atomic data to a function, you can totally pass aggregate data as well. Suppose we'd like to be able to easily compute the IQR (recall p. 150) of a univariate data set. Writing a function to do that is a snap:

```
def IQR(some_data):
    return some_data.quantile(.75) - some_data.quantile(.25)
```

We can now call it on anything we like, like our examples from p. 150 and p. 159:

```
print("The IQR of the YouTube plays data is {}".format(IQR(num_plays)))
print("The IQR of the NCAA scoring data is {}".format(IQR(pts)))
```

```
The IQR of the YouTube plays data is 412.
The IQR of the NCAA scoring data is 15.
```

Again, we named the function's own argument (`some_data`) something different than the variables it was called with (`num_plays` first, and then `pts`). This is a happy and healthy thing.

22.6 Returning text

So far, our functions have returned numeric answers. But they can certainly return text as well. Here's a function which assembles a person's full name out of his or her constituent components:

```
def full_name(last_name, first_name, middle_initial):
    return first_name + " " + middle_initial + ". " + last_name

my_full_name = full_name("Davies", "Stephen", "C")
her_full_name = full_name("Clinton", "Hillary", "R")

print("Your author's full name is: {}".format(my_full_name))
print("Another person's full name is: {}".format(her_full_name))
```

```
Your author's full name is: Stephen C. Davies
Another person's full name is: Hillary R. Clinton
```

(Recall from p. 34 that the "+" operator is used for the concatenation of strings.)

22.7 Returning `True` or `False`

It's common for a programmer to want a function which, instead of returning a number or text, tells her *whether or not something is true.* This lets her use the return value of such a function as the condition of an `if` statement.

Here's a trivial example:

```
def is_old_enough_to_vote(age):
    if number >= 18:
        return True
    else:
        return False
```

```
x = is_old_enough_to_vote(13)
if x:
    print("Yes, a 13-year-old can vote!")
else:
    print("Alas, a 13-year-old must wait.")

if is_old_enough_to_vote(19):
    print("Yes, a 19-year-old can vote!")
else:
    print("Alas, a 19-year-old must wait.")
```

```
Alas, a 13-year-old must wait.
Yes, a 19-year-old can vote!
```

The values `True` and `False` are called **boolean value**s, after the 19th-century mathematician George Boole. Note that in Python they must begin with *capital* letters.

The somewhat odd-looking "`if x:`" line is possible because `x` was set to the return value of a call to `is_old_enough_to_vote()`, and that function returned a boolean value.

22.8 Multiple return statements

Another thing the "old enough to vote" example illustrates is the presence of more than one `return` statement in a function. You might have thought this was useless, since on p. 225 I mentioned that as soon as a `return` is encountered during execution, the function is immediately completed. Why, then, would one ever have more than one – the second one could never be reached, right? Wrong. The branching nature of the `if`/`else` statement (above) means that the first `return` will be skipped over in some situations (negative arguments, *etc.*) so it's perfectly sensible to have more than one.

A more complicated example would be the "salutation" algorithm from section 21.4 (p. 217), this time embodied in a function:

```
def salutation(gender, marital_status, degree):
    if degree == "PhD" or degree == "MD":
        return "Dr."
    elif gender == "male":
        return "Mr."
    elif gender == "female":
        if marital_status == "married":
            return "Mrs."
        elif marital_status == "single":
            return "Miss"
        else:
            return "Ms."
    else:
        return "Mx."

my_salutation = salutation("male", "married", "PhD")
print("Why hello, {} Davies.".format(my_salutation))
print("And hello, {} Davies.".format(salutation("female",
    "married", "BS")))
```

```
Why hello, Dr. Davies.
And hello, Mrs. Davies.
```

Wow, all that code is in *one* function? Yeah. That's not unusual at all, although you should strive to make functions as compact as they can be. (The `salutation()` function *is* as compact as it can be, actually: there's no way to shorten it without changing what it does.)

The `salutation()` function, as you can see, has a veritable crapton of `return` statements – six in fact. But only one will ever be reached, because as soon as one *is* reached, the function is officially finished, and returns that value.

22.9 Returning nothing at all

Does it ever make sense for a function to return *nothing*? Oddly, yes: if it has a useful side effect. One side effect is printing:

```
def cheer_for(team):
    if team != "Christopher Newport":
        print("Go {} go!!".format(team))
    else:
        print("Uhh...no.")

cheer_for("Mary Washington")
cheer_for("Lady Eagles")
cheer_for("Christopher Newport")
```

```
Go Mary Washington go!!
Go Lady Eagles go!!
Uhh...no.
```

Here, instead of "`something = cheer_for(...)`" we type plain-old "`cheer_for(...)`". That's because there's no return value to capture, so there's no point in setting it equal to anything. We call the function just to enjoy its messages.

22.10 Calling a function from another function

Finally, note that we can put any code we desire inside a function's body, including one or more calls to other functions! Check out this bad boy:

```
def greet(first_n, last_n, middle_i, gender, status, deg, lang):
    sal = salutation(gender, status, deg)
    if lang == "Swedish":
        greeting = "Hej"
    elif lang == "Russian":
        greeting = "Privet"
    elif lang == "Hindi":
        greeting = "Namaste"
    else:
        greeting = "Yo"
    full = full_name(last_n, first_n, middle_i)
    print("{}, {} {}!".format(greeting, sal, full))

greet("Greta","Thunberg","F","female","single","none","Swedish")
greet("Maria","Sharapova","S","female","single","none","Russian")
greet("Garry","Kasparov","K","male","married","BA","Russian")
greet("Angela","Merkel","D","female","married","PhD","German")
```

```
Hej, Miss Greta F. Thunberg!
Privet, Miss Maria S. Sharapova!
Privet, Mr. Garry K. Kasparov!
Yo, Dr. Angela D. Merkel!
```

In the course of its duties, `greet()`'s function body calls both `salutation()` and `full_name()` for help. They each produce components of its complete solution. Good teamwork!

Chapter 23

Functions (2 of 2)

Like many things in life, writing functions is best learned by example. This chapter will feature several more of them that you can learn from and imitate.

Basketball scoring: `bb_pts()`

Continuing the sports theme, the total points a basketball player scores is related to the number of shots she makes of various kinds. Typically, the "box score" of a game (see example in Figure 23.1) reports three scoring stats: (1) the *total* number of "field goals"[1] a player made and attempted, (2) the number of these field goals, if any, that were for three points[2], and (3) the number of free throws ("easy" penalty shots) the player attempted and made.

Confusingly, (1) *includes* (2). In other words, if the first number is 4 and the second is 1, the player didn't score 4 regular two-point baskets and 1 three-pointer, but rather *3* two-point baskets and 1 three-pointer.

In Figure 23.1, the FGM-A column gives the first of these three categories, 3PM-A the second, and FTM-A the third. The PTS

[1] A "field goal" in basketball just means "a regular basket" – *i.e.*, not a free throw penalty shot.

[2] In most leagues, a basket is worth 2 points unless the shooter was farther than a certain distance from the hoop when she shot it, in which case it's worth 3.

Mary Washington

PLAYER	MIN	FGM-A	3PM-A	FTM-A	PTS
STARTERS					
50 - Tory Martin - f	25	6-10	0-0	1-4	13
10 - Faith St. Clair - g	23	1-2	1-1	0-0	3
14 - Maddie Shifflett - g	29	5-9	0-2	0-0	10
22 - Molly Sharman - g	25	5-8	1-2	2-2	13
23 - Emily Thompson - g	33	6-9	5-7	5-6	22
RESERVES					
05 - Karissa Highlander	7	1-4	0-0	0-0	2
20 - Hannah Stockman	19	2-8	0-5	0-0	4
21 - Emily Shively	10	0-2	0-1	0-0	0
32 - Bri Harper	3	1-4	1-3	0-0	3
34 - Ashley Martin	16	0-2	0-1	3-4	3
40 - Thora Gibbs	10	1-1	0-0	0-0	2
TM - TEAM					
TOTALS		28-59	8-22	11-16	75
		47.5%	36.4%	68.8%	

Figure 23.1: A basketball box score.

column gives the total number of points that player scored. (For example, Molly Sharman made 5 of her 8 attempted field goals, one of which was for three points, and she also converted both free throw attempts.)

All that took a lot longer to explain than the corresponding Python function:

```
def bb_pts(fgm, threep_fgm, ftm):
    return ((fgm - threep_fgm) * 2) + (threep_fgm * 3) + ftm

torys_pts = bb_pts(6, 0, 1)
print("Tory scored {} points.".format(torys_pts))
print("Emily scored {} points.".format(bb_pts(6,5,5)))
print("Lady Eagles scored {} points!".format(bb_pts(28,8,11)))
```

```
Tory scored 13 points.
Emily scored 22 points.
Lady Eagles scored 75 points!
```

Strictly speaking you don't need all those bananas (regular PEMDAS order-of-operations applies) but I think it's a good idea to include them for clarity and grouping.

"Exceptions": `mean_no_outliers()` and `quiz_avg()`

Sometimes we want to take the straight average of a data set, but other times we may want to filter out any strange or exceptional cases. Let's say we're computing the average age of a classroom of college students, but we want to remove any adult learners over 30 since that would skew the result. We could do this sort of thing with a function like this:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    return a[(a >= low_cutoff) & (a <= high_cutoff)].mean()

our_class = np.array([20,18,19,18,22,21,76,20,22,22,21,18])
print("The average age (excluding outliers) is {}.".format(
    mean_no_outliers(our_class, 0, 30)))
```

```
The average age (excluding outliers) is 20.09090909090909.
```

We've provided two arguments to the function besides the data set itself: a lower and upper bound. Anything falling outside that range will be filtered out. In the example function call, we passed 0 for the `low_cutoff` since we didn't desire to filter anything at the low end. (If we wanted to, say, also remove children from the data set, we could have set that to 16 or so.)

By the way, you might find the number of decimal places printed to be unsightly. If so, we could enhance our function by rounding the result to (say) two decimals with NumPy's `round()` function:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    return np.round(a[
        (a >= low_cutoff) & (a <= high_cutoff)].mean(),2)

print("The average age (excluding outliers) is {}.".format(
    mean_no_outliers(our_class, 0, 30)))
```

```
The average age (excluding outliers) is 20.09.
```

At this point you might think this function is getting pretty big for a one-liner. I agree. Let's split it up and use some temporary variables to make it more readable:

```
def mean_no_outliers(a, low_cutoff, high_cutoff):
    filtered_data = a[(a >= low_cutoff) & (a <= high_cutoff)]
    filtered_average = np.round(filtered_data)
    return np.round(filtered_average,2)
```

Much clearer!

A related but different example would be to remove a fixed *number* of data points from the end, instead of data points outside a specified range. For instance, in my classes, I often give students (say) eight quizzes during a semester, and drop the lowest two scores. That could be done with:

```
def quiz_avg(quizzes):
    dropped_lowest_two = np.sort(quizzes)[2:]
    return dropped_lowest_two.mean()

filberts_quizzes = np.array([7,9,10,7,0,8,4,10])
print("Filbert's avg score was {}.".format(quiz_avg(
    filberts_quizzes)))
```

```
Filbert's avg score was 8.5.
```

Filbert's 0 and 4 were dropped, leaving him with a pretty good semester score.

The trick to this implementation is *sorting* the quiz scores. Once you do that, it's easy to pick out the top six to take the average, since the lowest two scores will be at the beginning of the (sorted) array. Two notes here:

- We use the `np.sort()` function, not the `.sort()` method, since we don't want to permanently change the order of `quizzes`. We only need a temporarily sorted copy so we can omit the lowest two entries.
- That business in the boxies ("`[2:]`") is a **slice** (recall section 9.2 on p. 76) which says "only give me entries number 2 through the end of the array." And that's exactly what the doctor ordered to omit the first two.

Searching for values: `any_zeros()`

I'll end this chapter with an example which, like the "preferred language" example on p. 214, flummoxes nearly every beginning student.

Suppose students in a DATA 101 course are given labs to complete, each one worth up to 20 points. (This is purely hypothetical, as you can see.) At the midway point of the semester, the instructor would like a quick list of any students who failed to turn in one of the labs, so he can harass them for their own good.

Here's the `gradebook` `DataFrame` this professor is using:

student	Q1	Q2	Q3	Q4	lab1	lab2	lab3	lab4	lab5
Filbert	7	9	10	7	15	19	14	20	20
Jezebel	8	7	0	6	12	12	16	0	20
Betty Lou	10	10	10	10	20	20	20	20	20
Biff	3	2	6	5	10	12	0	0	16
Melvin	0	0	10	10	0	18	20	14	20

Let's write a function called `print_harass_list()` whose job is to tell this professor which students he should check up on. We'll write it as follows:

```
def print_harass_list(gradebook):
    for row in gradebook.itertuples():
        if any_zeros(np.array([row.lab1, row.lab2, row.lab3,
            row.lab4, row.lab5])):
            print("Better check up on {}.".format(row.Index))
```

Note that we've pushed some of the work on to a new function, `any_zeros()`, that we haven't written yet. This is good organizational style. Now `print_harass_list()` can do the job of iterating through the `DataFrame` rows, extracting the lab scores, and printing a message if necessary, whereas it defers to `any_zeros()` to inspect the lab scores and determine the presence of any zeros.

It doesn't work until we actually write the second function, of course, so here goes. Heads up, since this is the part that perplexes students. The following implementation of `any_zeros()` looks perfectly reasonable, yet is dead WRONG:

```
def any_zeros_WRONG(labs):
    for lab in labs:
        if lab == 0:
            return True
        else:
            return False
```

It looks so correct! And yet it is not. Check out the result:

```
print_harass_list(gradebook)
```

```
Better check up on Melvin.
```

Clearly we need to check up on `Jezebel` and `Biff` as well (look at their scores for labs 3 and 4), yet they inexplicably didn't get printed.

Here's what's WRONG with that `any_zeros()` attempt. Stare carefully at that loop and realize that the *body* of the loop is comprised of a single `if`/`else` statement. And remember our cardinal rule from the grey box on p. 214: either the `if` body or the `else` body will *always* be executed.

That means that this loop is destined to only execute exactly once! It doesn't matter how long the `labs` array is. It effectively looks

only at the *first* element, and decides based solely on that whether or not the entire array has any zeros in it!

The correct version of `any_zeros()` would look like this:

```
def any_zeros(labs):
    for lab in labs:
        if lab == 0:
            return True
    return False
```

At first glance, it may appear unchanged, but look again. First of all, there's no `else` anymore. Second of all, the "`return False`" line is indented *evenly with the word* `for`. This means that "`return False`" is *not* part of the loop at all: it will only run *after* the entire loop has executed.

That turns out to make all the difference. The function will dutifully go through *each* element of the `labs` array, inspecting each one to see whether it's zero. As soon as it finds a zero, it returns `True`, since then its job is done. Only after inspecting the *entire* array, and coming up empty on its zero quest, does this function then have the audacity to return `False`, meaning "nope! Clean as a whistle." The result:

```
Better check up on Jezebel.
Better check up on Biff.
Better check up on Melvin.
```

Postlude: thinking algorithmically

Getting tripped up on that last example is, I believe, usually a case of *thinking holistically* rather than *thinking algorithmically*. Math classes have trained people to think holistically, by which I mean looking at (say) a bunch of equations and viewing them as "all equally true, all at once." And this is the correct way to think mathematically. If I give you five simultaneous equations

that state relationships among variables, they aren't really in any order. They're just "five true things."

But programming requires you to think algorithmically. You have to execute the code in your head, step by step, and realize the consequences. The appealing symmetry of the WRONG `any_zeros()` function is appealing because you're looking at it as a whole: "it's looping (seemingly) through all the elements, with zeros being an indicator of `True`ness and non-zeros being an indicator of `False`ness. What's not to like?" The error, as you saw, is that when running through the data step-by-step, there are immense ramifications of returning early. That's only apparent if you think of the code executing sequentially as it goes. You have to pretend you're the computer, not a mathematician.

Chapter 24

Recoding and transforming

It's often the case that although a `DataFrame` contains the raw information you need, it's not exactly in the form you need for your analysis. Perhaps the data is in different units than you need – meters instead of feet; dollars instead of yen. Or perhaps you need some *combination* of available quantities – miles per gallon instead of just miles and gallons separately. Or perhaps you need to reframe a variable by binning it into meaningful subdivisions – categorizing a raw column of salaries into "high," "medium," and "low" wage earners, for instance.

In data science, these activities are known as **recoding** and/or **transforming**. There's not a sharp division between the two; usually I think of recoding as converting a single variable to one with different units (as in the dollars-to-yen and high/medium/low earners examples) and transforming as creating a new variable entirely out of a combination of columns (like miles per gallon). In both cases, though, we'll be creating and adding new columns to a `DataFrame`. These columns are sometimes called **derived columns** since they're based on (derived from) existing columns rather than containing independent information.

24.1 Recoding with simple operations

Consider the following soccer data set called `worldcup2019.csv`. Each row of this data set represents one player's performance in a particular 2019 World Cup game. Notice that we have a couple of players with more than one row (Megan Rapinoe and Rose Lavelle), and several rows for the same game (the first four rows are all from the June 28th game, for instance):

```
last,first,date,inmins,insecs,outmins,outsecs,gls,asst,tkls,shots
Morgan,Alex,28-Jun-2019,0.0,0.0,90.0,0,0,0,0,2,1
Rapinoe,Megan,28-Jun-2019,0.0,0.0,74.0,27.0,2,0,2,3
Press,Christen,28-Jun-2019,74.0,27.0,90.0,0.0,0,0,1,0
Lavelle,Rose,28-Jun-2019,0.0,0.0,90.0,0.0,0,1,3,0
Lavelle,Rose,7-Jul-2019,0.0,0.0,90.0,0.0,1,0,4,1
Rapinoe,Megan,7-Jul-2019,0.0,0.0,83.0,16.0,1,1,3,2
Lloyd,Carli,7-Jul-2019,83.0,16.0,90.0,0.0,0,0,1,0
Dunn,Crystal,23-Jun-2019,42.0,37.0,81.0,5.0,0,1,1,2
```

The data set doesn't really have a meaningful index column, since none of the columns are expected to be unique. So we'll leave off the "`.set_index()`" method call when we read it in to Python:

```
wc = pd.read_csv('worldcup2019.csv')
print(wc)
```

```
     last first   date inmins insecs outmins outsecs gls asst tkls shots
   Morgan  Alex 28-Jun      0      0      90       0   0    0    2     1
  Rapinoe Megan 28-Jun      0      0      74      27   2    0    2     3
    Press Chris 28-Jun     74     27      90       0   0    0    1     0
  Lavelle  Rose 28-Jun      0      0      90       0   0    1    3     0
  Lavelle  Rose  7-Jul      0      0      90       0   1    0    4     1
  Rapinoe Megan  7-Jul      0      0      83      16   1    1    3     2
    Lloyd Carli  7-Jul     83     16      90       0   0    0    1     0
     Dunn Cryst 23-Jun     42     37      81       5   0    1    1     2
```

Let's zero in on the columns with `mins` and `secs` in the names. These columns show us the minute and second that the player went

`in` to the game, and the minute and second that they came `out`. For example, Alex Morgan played the entire 90-minute match on June 28th. Rapinoe started that game, but came out for a substitute at the 74:27 mark. Who replaced her? Looks like Christen Press did, since she *entered* the game at exactly the same time. In most rows, the player either started the game, or ended the game or both, but the last row (Crystal Dunn's June 23rd performance) has her entering at 42:37 and exiting at 81:05.

Now the reason I bring this up is because one aspect of our analysis might be computing statistics *per minute* that each athlete played. If one player scored 3 goals in 200 minutes, for example, and another scored 3 goals in just 150 minutes, we could reasonably say that the second player was a more prolific scorer in that World Cup.

This is hard to do with the data in the form that it stands. So we'll **recode** a few of the columns. Let's collapse the minutes and seconds for each of the two clock times into a single value, in minutes. For readability, we'll also round this number to two decimal places using the `round()` function we met on p. 237:

```
wc['intime'] = np.round(wc['inmins'] + (wc['insecs']/60),2)
wc['outtime'] = np.round(wc['outmins'] + (wc['outsecs']/60),2)
```

We're taking advantage of vectorized operations here. For each row, we need to divide the `insecs` value by 60 (to convert it to minutes) and add it to the `inmins` value. Pandas makes this super easy here, since we can just write out those operations once, and it will compute it for every single row!

Let's delete the old, superfluous columns now and looksie:

```
del wc['inmins']
del wc['insecs']
del wc['outmins']
del wc['outsecs']
print(wc)
```

```
      last first   date gls asst tkls shots intime outtime
0  Morgan  Alex 28-Jun   0    0    2     1   0.00   90.00
1 Rapinoe Megan 28-Jun   2    0    2     3   0.00   74.45
2   Press Chris 28-Jun   0    0    1     0  74.45   90.00
3 Lavelle  Rose 28-Jun   0    1    3     0   0.00   90.00
4 Lavelle  Rose  7-Jul   1    0    4     1   0.00   90.00
5 Rapinoe Megan  7-Jul   1    1    3     2   0.00   83.27
6   Lloyd Carli  7-Jul   0    0    1     0  83.27   90.00
7    Dunn Cryst 23-Jun   0    1    1     2  42.62   81.08
```

This is much less unwieldy (more wieldy?) than dealing with minutes and seconds separately.

(Incidentally, notice that the technique presented here creates *new* columns (with new names) and then deletes the old columns. I strongly recommend doing it this way. If you try to change the values of an *existing* `DataFrame` column, Pandas will often give you a strange-looking message informing you of a "`SettingWithCopyWarning`". The meaning is a bit esoteric, but in layman's terms it means "your operation may not have actually worked." Avoid this problem by creating new columns instead.)

24.2 Transforming with simple operations

Now that we've converted the awkward minutes-and-seconds columns to just "`time`" columns, all we need to do to complete our analysis is **transform** this data by computing a new quantity entirely: the *total number of minutes played* for each player in each game. Again, Pandas makes this easy:

```
wc['minsplayed'] = wc.outtime - wc.intime
print(wc)
```

```
      last first   date gls asst tkls shots intime outtime minsplayed
0  Morgan  Alex 28-Jun   0    0    2     1   0.00   90.00      90.00
1 Rapinoe Megan 28-Jun   2    0    2     3   0.00   74.45      74.45
2   Press Chris 28-Jun   0    0    1     0  74.45   90.00      15.55
3 Lavelle  Rose 28-Jun   0    1    3     0   0.00   90.00      90.00
4 Lavelle  Rose  7-Jul   1    0    4     1   0.00   90.00      90.00
5 Rapinoe Megan  7-Jul   1    1    3     2   0.00   83.27      83.27
6   Lloyd Carli  7-Jul   0    0    1     0  83.27   90.00       6.73
7    Dunn Cryst 23-Jun   0    1    1     2  42.62   81.08      38.46
```

Voilà. We now have the time-on-field for each player, which gives us a whole new avenue of exploration. For example, any of the counting stats (goals, assists, *etc.*) can be converted into a "per-minute" version, showing us how productive a player was while on the field. Let's do that for `tkls` ("tackles"), and multiply by 90 to obtain a "tackles-per-90-minutes" statistic[1]:

```
wc['minsplayed'] = wc['outtime'] - wc['intime']
wc['tkl_per_90'] = np.round(wc['tkls'] /
    wc['minsplayed'] * 90,2)
del wc['tkls']
```

```
      last first   date gls asst shots intime outtime minsplayed tkl_90
0   Morgan  Alex 28-Jun   0    0     1   0.00   90.00      90.00   2.00
1 Rapinoe Megan 28-Jun   2    0     3   0.00   74.45      74.45   2.42
2    Press Chris 28-Jun   0    0     0  74.45   90.00      15.55   5.79
3 Lavelle  Rose 28-Jun   0    1     0   0.00   90.00      90.00   3.00
4 Lavelle  Rose  7-Jul   1    0     1   0.00   90.00      90.00   4.00
5 Rapinoe Megan  7-Jul   1    1     2   0.00   83.27      83.27   3.24
6    Lloyd Carli  7-Jul   0    0     0  83.27   90.00       6.73  13.37
7     Dunn Cryst 23-Jun   0    1     2  42.62   81.08      38.46   2.34
```

Transforming grouped data

The above example computed tackles-per-game all right, but it still left us with one row for every player-performance. (In other words, the results had *two* rows for Rose Lavelle, one giving her `tkl_per_90` for the June 28th game, and one giving it for the July 7th game.)

We might instead be interested in a player-by-player analysis: overall in the entire month-long World Cup, which players had the most tackles-per-game? This is easy to do with the `.groupby()` method that we first encountered in section 18.2 (p. 189). First, we group the rows by the first *two* columns (since first-and-last-names-together are needed to uniquely identify a single player):

[1] I'm choosing 90 minutes here because that's how long a regulation-length soccer match is. Therefore, our new `tkl_per_90` column gives us "number-of-tackles-per-complete-game," which is easier to interpret than "tackles-per-*minute*," which would be a miniscule number for any player.

```
grouped_wc = wc.groupby(['last','first'])
```

We then take our new, temporary `grouped_wc` variable and extract the `gls`, `asst`, `shots`, `tkls`, and `minsplayed` columns from it, **summing** each of them to produce the per-player values in the result:

```
by_player = grouped_wc[['gls','asst','shots','tkls',
    'minsplayed']].sum()
```

This yields:

		gls	asst	shots	tkls	minsplayed
last	first					
Dunn	Cryst	0	1	2	1	38.46
Lavelle	Rose	1	1	1	7	180.00
Lloyd	Carli	0	0	0	1	6.73
Morgan	Alex	0	0	1	2	90.00
Press	Chris	0	0	0	1	15.55
Rapinoe	Megan	3	1	5	5	157.72

Now, we're ready to compute a per-game analysis as before, but this time for each player's entire World Cup games:

```
by_player['tkl_per_90'] = (np.round(by_player['tkls'] /
    by_player['minsplayed'] * 90,2))
del by_player['tkls']
```

		gls	asst	shots	minsplayed	tkl_per_90
last	first					
Dunn	Cryst	0	1	2	38.46	2.34
Lavelle	Rose	1	1	1	180.00	3.50
Lloyd	Carli	0	0	0	6.73	13.37
Morgan	Alex	0	0	1	90.00	2.00
Press	Chris	0	0	0	15.55	5.79
Rapinoe	Megan	3	1	5	157.72	2.85

24.3 More complex transformations

In all the above examples, we took advantage of Pandas vectorized operations. With just a single line of code like "`wc['minsplayed'] = wc.outtime - wc.intime`", we could compute our entire new transformed column in one fell swoop.

Sometimes, we're not so lucky. In particular, if the computation of the transformed column is more complicated than just numeric operations – like, if it involves branches, loops, or calling other functions – we normally can't compute it all at once. Instead, we have to resort to a loop.

Pandas makes this procedure a bit awkward in my opinion. But once you learn the pattern, it's not hard to imitate. Here's the pattern for creating a transformed/recoded column that requires more complex operations:

1. Create a **function** that will compute the transformed value for a *single* row. Its arguments should be whatever column values are necessary to derive the new value, and its return value should be the desired transformation.
2. Create an *empty* NumPy array to hold the row-by-row results, and make sure it's the right type.
3. Write a loop that will iterate through all the rows of the original `DataFrame`. For each row, pass the appropriate values to the function, and then *append* the return value to the ever-growing NumPy array.
4. Finally, slap that NumPy array on to the `DataFrame` as a new column.

Here's a couple examples. First, suppose we want to compute a shooting percentage for each player; in other words, how many goals they scored per shot they took. Now you might think we could simply use vectorized operations:

```
wc['shots_per_goal'] = wc.gls / wc.shots
```

The problem is, for players who never attempted a shot in the game, this would result in dividing by zero, a cardinal sin. Sports convention says that if a player makes 0 goals in 0 attempts, their shooting percentage is 0.00, even though mathematically-speaking this is undefined.

Very well, following our procedure from above, we'll first define a function `shooting_perc()`:

```
def shooting_perc(gls, shots):
    if shots == 0:
        return 0.0
    else:
        return np.round(gls / shots * 100, 1)
```

Then, we create an empty NumPy array. Here's how:

```
s_perc = np.array([])
```

Looks weird, I know. But remember, the `array()` function (review p. 63) takes a boxie-enclosed list of elements. If we enclose *nothing* inside the boxies, that effectively makes it an empty list.

And why would we want to do that? Because we need to continually add to this array, one value for each row in the `DataFrame`. At the end, there must be *exactly* as many elements in `s_perc` as there are rows in `wc`, otherwise we won't be able to add it as a new column.

Here's the loop (step 3 from the shaded box):

```
for row in wc.itertuples():
    new_s_perc = shooting_perc(row.gls, row.shots)
    s_perc = np.append(s_perc, new_s_perc)
```

I've chosen to create a temporary variable here (`new_s_perc`) for readability. The first line of the loop body says to take the current `row`'s `gls` and `shots` values, and send them as arguments to the `shooting_perc()` function. That function, which we defined above, will return us a single number which is the shooting percentage for *that row*. The second line then appends that single `new_s_perc` value to the end of the ever-growing `s_perc` array.

Finally, we add this new column to the `wc DataFrame` proper:

```
wc['s_perc'] = s_perc
```

which gives us:

```
      last first   date gls asst shots intime outtime minsplayed s_perc
0  Morgan  Alex 28-Jun   0    0     1   0.00   90.00      90.00     0.0
1 Rapinoe Megan 28-Jun   2    0     3   0.00   74.45      74.45    66.7
2   Press Chris 28-Jun   0    0     0  74.45   90.00      15.55     0.0
3 Lavelle  Rose 28-Jun   0    1     0   0.00   90.00      90.00     0.0
4 Lavelle  Rose  7-Jul   1    0     1   0.00   90.00      90.00   100.0
5 Rapinoe Megan  7-Jul   1    1     2   0.00   83.27      83.27    50.0
6   Lloyd Carli  7-Jul   0    0     0  83.27   90.00       6.73     0.0
7    Dunn Cryst 23-Jun   0    1     2  42.62   81.08      38.46     0.0
```

Rose Lavelle's July 7th game was the only perfect shooting performance in this data set – who knew?

We'll complete this chapter with a slightly more complex example, but which still follows the shaded box pattern.

Say we're also interested in which players *started* which games (as opposed to being a mid-game substitute). Obviously, a starter is someone who entered the game at time 0. To create a new column

for this, we'll need our function to return the boolean value `True` if the player's `intime` value was zero, and `False` otherwise. Here's the complete code snippet for this transformation:

```
def starter_func(intime):
    if intime == 0:
        return True
    else:
        return False

starter = np.array([]).astype(bool)

for row in wc.itertuples():
    starter = np.append(starter, starter_func(row.intime))

wc['starter'] = starter
```

```
      last first   date gls asst tkls shots minsplayed s_perc starter
  0  Morgan  Alex 28-Jun   0    0    2     1       90.00    0.0    True
  1 Rapinoe Megan 28-Jun   2    0    2     3       74.45   66.7    True
  2   Press Chris 28-Jun   0    0    1     0       15.55    0.0   False
  3 Lavelle  Rose 28-Jun   0    1    3     0       90.00    0.0    True
  4 Lavelle  Rose  7-Jul   1    0    4     1       90.00  100.0    True
  5 Rapinoe Megan  7-Jul   1    1    3     2       83.27   50.0    True
  6   Lloyd Carli  7-Jul   0    0    1     0        6.73    0.0   False
  7    Dunn Cryst 23-Jun   0    1    1     2       38.46    0.0   False
```

One subtle point that is easy to miss: when we first created the empty `starter` array, we typed "`.astype(bool)`" at the end. This is because by default, the values of a new empty array will be **float**s. This worked fine for the shooting percentage example, because that's actually what we wanted, but here we want `True`/`False` values instead (for "starter" and "non-starter.")

Pretty cool, huh? The original `DataFrame` had the information we wanted, but not in the form we really needed it. What we wanted was not the entry time and exit time of each player (both in minutes and seconds) but rather the total time that player was on the pitch, and whether or not they started the game. We also wanted to convert several of the raw statistics into per-complete game numbers,

and to compute meaningful ratios like shooting percentage or fouls per assist.

Recoding and transforming turn out to be common tasks for a simple reason: *whoever collects a data set can rarely predict how an analyst will eventually use it.* We're very grateful to the author of the `.csv` file, since it contains the raw material we need to evaluate our team's performances; but how were they to know that length-of-time-on-the-field and who-started-which-game was going to be important to us? They couldn't. But thanks to recoding and transformation skills, we can cope.

Chapter 25

Machine Learning: concepts

When ordinary people hear the words "Data Science," I'll bet the first images that come to mind are of the closely-related fields of **data mining** and **machine learning (ML)**, even if they don't know those terms. After all, this is where all the sexy tech is, and the success stories too: Netflix magically knowing which movies you'll like, grocery chains using data from loyalty cards to optimally place products; the Oakland A's scouring minor league stats to build a champion team with chump change (see: *Moneyball*). There are also creepier applications of this technology: Google placing personalized eye-catching ads in front of you using data they mined from your email text, or Cambridge Analytica projecting from voter personalities to the best ways to micro-target them.

All these examples have one thing in common: they actually *make* the discoveries and predictions from the data. They're the coup de grâce. They take place after we've already acquired our data, imported it to an analysis environment (like Python), stored it in the appropriate data structures (like associative arrays or tables), recoded/transformed/pre-processed it as necessary, and explored it enough to know what we want to ask. All that stuff was mere prep work. This chapter is where we begin to really rock-and-roll.

25.1 Data mining vs. machine learning

The terms "data mining" and "ML" have a lot of sloppy overlap, but one distinction we can pick out is this. If someone says they're doing data mining, their goal is normally **inference**: deriving high-level strategic insights based on patterns in the data. Discovering that amateur pitching performances translate more reliably to the major leagues than amateur batting performances do, generally speaking, is an inference, and a potentially valuable find.

If someone says they're doing ML, on the other hand, their goal is normally **prediction**: making an educated guess about how a specific case will turn out. When we forecast how many home runs we think a college prospect will hit in his first two years in the majors, we're making a specific prediction rather than inferring a general truth – this, too, is potentially quite valuable, as it may lead us to decide to sign the player or look at different options.

25.2 Deductive vs. inductive reasoning

This chapter contains a lot of vocabulary terms. Before we dive in to the ML-specific ones, I think it's important to take a step back and make a more general point about the kind of "learning" we'll be doing. There are at least two different ways that human beings reach conclusions: **deductively** and **inductively**. Deductive reasoning is associated most prominently with Sherlock Holmes in the public mind. Through sheer application of irrefutable logic, Holmes and his companion Watson deduced new facts from known facts in their quest to catch the criminal. Their logic was seemingly air-tight, since everything they deduced followed directly and irresistibly from what came before.

There's a subdiscipline of Philosophy called Logic which covers exactly such matters. Syllogisms, *modus ponens*, first-order predicate calculus: these are all concepts you'll learn if you take an introductory course in Logic. And the nice thing about deduction is that as long as you follow the rules, *your conclusions will always be dependably correct.*

Inductive reasoning, on the other hand, does *not* always lead to 100% reliably correct conclusions. This may give you pause, and wonder why anyone would ever use it. The reason is that in the vast majority of cases, deductive reasoning simply isn't applicable to your situation, and induction is the only case.

Induction is about *reasoning from examples.* Lots of examples. Living in the world as we do, we observe plenty of examples of how people and things behave, and we start to identify certain general patterns in what we've observed. One thing I noticed long ago is that when I smile and say hi to a person, they normally smile and say hi back. But when I smile and say hi to a dog, or a bush, or a vending machine, I'm normally met with stony silence.

From this, I've **induced** the general rule that people respond to greetings but other objects don't. Now this is *not* 100% reliably true. Even in my own experience, there have been times when I've greeted someone walking down the hallway and been outright ignored. And for all I know, there may be some vending machines out there who might respond if someone talks to them – with technological advancements in voice recognition and synthesis, it's probably just a matter of time before they do. But the point is that *learning this general principle about greetings has served me very well in life.* I don't normally talk to inanimate objects, but I do to people, and this has helped me function in society. Even if a rule *isn't* accurate in absolutely every situation, it can still be very, very important.

If you do a quick scan of your brain, I believe you'll find that the vast majority of the things that you "know" about life were arrived at inductively, rather than deductively. If you ask a friend for money, he'll probably say yes; if you ask a stranger, he'll probably say no. If your friend does say yes, he'll probably expect the favor to be returned at a later point; if the stranger says yes, he probably won't. If you don't study for a test, you'll probably do poorly, and likewise if you wait until the last day to start your 5-page paper. None of these conclusions can be proven deductively, and in fact all of them have exceptions; but not to know these things is to be at a serious disadvantage in trying to make decisions.

I say all this because *everything in ML is about induction, not de-*

duction. As we'll see, the name of the game in ML is looking at lots and lots of past examples, and making future predictions based on them. It's true that "past performance is no guarantee of future success," but past performance *does* tell you *something* valuable about future possibilities, else there'd be no point in trying to learn from it. And the fact that we apply our past lessons in altering our future behavior is undeniable.

25.3 Supervised vs. unsupervised learning

Now let's dive further down to some more technical and fine-grained distinctions. There are two main categories of machine "learning": **supervised** and **unsupervised**. I think these terms are ridiculous and misleading, by the way, but they're what we're stuck with so let's learn what they mean.

In a supervised learning setting, our goal is to predict the value of some **target attribute** of some object of study. As an example, let's say we want to predict someone's *mood* based only on their facial expression and body language. "Mood" might be a categorical variable with values "happy," "angry," "bored," *etc.*

To do this prediction, we'll use a bunch of previously observed examples. These past examples, *for which the person's true mood is known*, are collectively called our **training data**. We remember seeing one person with a smile on their face and their eyes slightly squinted, and later discovered that they were happy. We remember another person with a smile but with wide open eyes, and learned that they, too, were happy. We also remember someone with clenched fists and raised eyebrows, and they turned out to be frightened. A different person with clenched fists but squinted eyes was later revealed to be angry. And so on.

Supervised machine learning is about how to extrapolate from past examples in a principled way, in order to make predictions about future examples whose true value (mood, say) is *not* known. The task is to say, "okay, there's a person down the hallway whose face is slightly flushed and whose arms are tightly crossed. Are they likely to be happy, defensive, angry, embarrassed, or something else?

Let's apply what we've learned from past examples to guess at the answer."

It's called "supervised" precisely because the "true answer" for the target attribute is known for the training data.

Now suppose we *didn't* know the true answer for our training examples. Say we've observed and recorded the eyebrow position, the mouth configuration, whether the face was flushed or pale or in between, *etc.*, for a bunch of people we've encountered in the past, but we actually never learned what their mood was. What then?

This is an unsupervised learning setting. Predicting a person's mood based on this kind of information turns out to be nearly hopeless. If we don't know what anyone else's mood was, how can we predict what this new person's mood is? But all is not lost – we may still be able to form some conclusions about what *types* of moods there are. For example, we might notice that generally speaking, raised eyebrows tend to be accompanied by certain other indicators. In many past examples, they've appeared together with an open mouth and a rigid posture. In other examples, raised eyebrows instead appeared with lips tightly-pressed together and the forehead slightly tilted forward. We don't know which moods these collections of features might correspond to, since our training data didn't have any information about moods. But we might still infer the presence of a couple of distinct raised-eyebrow moods, since they are so commonly accompanied by either one of two groups of other features.

Classification, regression, and clustering

In the supervised setting, our most common machine learning activities will be **classification** and **regression**. In each one, our job is to predict the value of the target attribute for a new object, based on the previous example objects we've seen. The only difference is the scale of measure of the target variable: if it's categorical, we're performing classification, and our goal is to build a **classifier**: an algorithm (basically, a Python function) that can **classify** future examples by guessing their target value. If the target variable is numeric, then we have regression, and our goal is to make the closest

guess we can to the true target value.

For example, if we have some census and earnings data for a region, and our goal is to predict whether or not someone in that region will be a homeowner or a renter, we're performing classification. If our goal instead is to predict their annual salary, we're doing regression.

By the way, let me make clear that the types of the *other* variables we're considering (*i.e.*, other than the target) don't play in to whether we're doing classification or regression: only the target does. If I'm using race (categorical), gender (categorical), age (numeric), and college degree (categorical) to predict *salary* (numeric), then this is a *regression* problem, even though the majority of the variables are categorical. If I were to use the same four variables to predict *political affiliation* (categorical), then it would be a classification problem, even though we had a numeric variable as a predictor.

In the unsupervised setting, the most common task is **clustering**: finding groups of related objects, with similar attribute values, in order to discern how many basic types of objects there are, and what their typical value ranges are. That's exactly what we did with the mood data, above, in the absence of information about past moods. Another example would be to look at the attributes of various movies on IMDB and discern "what basic types of films there are." We may discover that movies naturally break down into blockbuster action films, period dramas, romantic comedies, and a few other common genres. There will always be objects that defy categorization, and exist on the boundaries of defined clusters, but it's still profoundly insightful to discover the presence of common patterns that bring structure to the data.

Machine learning is a big field, and each aspect has its own techniques and deserves its own treatment. For the rest of this book, we're going to concentrate only on **supervised** learning, specifically the task of **classification**.

Chapter 26

Classification: concepts

26.1 Labeled and unlabeled examples

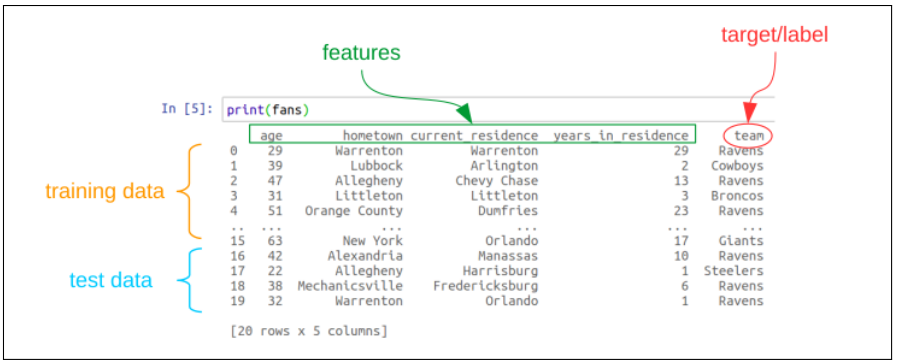
In the activity of classification, the **target** variable we aim to predict is categorical. We sometimes also call this variable the **label**. Since this is a *supervised* process, we are provided with example objects of study that have known "true answers." These are called **labeled examples**. The goal of the activity is to produce good predictions of the labels for other, **unlabeled examples**. A program that can make such predictions, after having studied the labeled examples, is called a **classifier**.

The predicted label can be seen as the "output" from our classifier. All of the other variables are essentially the inputs to our process, which we use to make our predictions. These variables are called **features** (or sometimes, **attributes**).

In terms of Pandas data structures, all these labeled examples will normally come packaged in a `DataFrame`. Each row of the `DataFrame` will be one labeled example, with its features as columns and its target/label as a column (traditionally, the rightmost one).

This is illustrated in Figure 26.1. Here we have some labeled examples for a data set on NFL fans. Each row represents one fan, and shows various features of their existence – how old they are, where they were born, where they live now, and how many years they've

lived in their current residence. The rightmost column gives the target: the team to whom this fan has sworn their allegiance. Our aim would be to predict which team a fan might root for, based on what we know about them. Lest you think this example is frivolous, consider that a sporting goods company might want to send catalogs (paper or electronic) to potential customers, and it would probably boost sales if the cover image of the catalog featured a model wearing apparel from the customer's favorite team, rather than their rival.



	age	hometown	current_residence	years_in_residence	team
0	29	Warrenton	Warrenton	29	Ravens
1	39	Lubbock	Arlington	2	Cowboys
2	47	Allegheny	Chevy Chase	13	Ravens
3	31	Littleton	Littleton	3	Broncos
4	51	Orange County	Dumfries	23	Ravens
..	...	...	...	...	...
15	63	New York	Orlando	17	Giants
16	42	Alexandria	Manassas	10	Ravens
17	22	Allegheny	Harrisburg	1	Steelers
18	38	Mechanicsville	Fredericksburg	6	Ravens
19	32	Warrenton	Orlando	1	Ravens

Figure 26.1: Some labeled examples, divided into training and test sets.

You'll also see in the figure that I've split the rows up into two groups. The first group is called the **training data**, and the second, the **test data**. (Normally we'll shuffle all the rows before assigning them, so that we don't put all the top rows of the `DataFrame` in the training set and all the bottom ones in the test set. But that's harder to show in a picture.)

26.2 Three kinds of examples

Now here's the deal. There are *three* kinds of example rows we're going to deal with:

1. **training data** – *labeled* examples which we will show to our classifier, and from which it will try to find useful patterns to make future predictions.

2. **test data** – *labeled* examples which we will *not* show to our classifier, but which we will use to measure how well it performs.
3. **new data** – *unlabeled* examples that we will get in the future, after we've deployed our classifier in the field, and which we will feed to our classifier to make predictions.

The purpose of the first group is to give the classifier useful information so it can intelligently classify.

The purpose of the second group is *to assess how good the classifier's predictions are.* Since the test set consists of labeled examples, we know the "true answer" for each one. So we can feed each of these test points to the classifier, look at its prediction, compare it to the true answer, and judge whether or not the classifier got it right. Assessing its accuracy is usually just a matter of computing the percentage of how many test points it got right.

The third group exists because after we've built and evaluated our classifier, we actually want to put it into action! These are new data points (new sporting goods customers, say) for which we don't know the "true answer" but want to predict it so we can send catalogs likely to be well-received.

Thou shalt not reuse

Now one common question – which leads to a *super* important point – is this: why can't we use *all* the labeled examples as training data? After all, if we have 1000 labeled examples we've had to work hard (or pay $$) to get, it seems silly to only use some of them to train our classifier. Shouldn't we want to give it all the labeled data possible, so it can learn the maximum amount before predicting?

The first reply is: "but then we wouldn't have any test data, and so we wouldn't know how good our classifier *was* before putting it out in the field." Clearly, before we base major business decisions on the results of our automated predictor, we need to have some idea of how accurate its predictions are.

It's then often countered: "sure, but why not then re-use that data for testing? Instead of splitting the 1000 examples into training

points and test points, why not just use all 1000 for training, and then test the classifier on all 1000 points? What's not to like?"

This is where the super important point comes in, and it's so important that I'll put it all in boldface. It turns out that **you absolutely *cannot* test your classifier on data points that you gave it to train on, because you will get an overly optimistic estimate of how good your classifier actually is.**

Here's an analogy to make this point more clear. Suppose there's a final exam coming up in your class, and your professor distributes a "sample exam" a week before exam day for you to study from. This is a reasonable thing to do. As long as the questions on the sample exam are of the same type and difficulty as the ones that will appear on the actual final, you'll learn lots about what the professor expects you to know from taking the sample exam. And you'll probably increase your actual exam score, since this will help you master exactly the right material.

But suppose the professor uses the *exact same* exam for both the sample exam and the actual final exam? Sure, the students would be ecstatic, but that's not the point. The point is: *in this case, students wouldn't even have to learn the material.* They could simply memorize the answers! And after they all get their A's back, they might be tempted to think they're really great at chemistry...but they probably aren't. They're probably just really great at memorizing and regurgitating.

Going from "the kinds of questions you may be asked" to "*exactly* the questions you *will* be asked" makes all the difference. And if you just studied the sample exam by memorization, and were then asked (surprise!) to demonstrate your understanding of the material on a *new* exam, you'd probably suck it up.

And so, the absolute iron-clad rule is this: **any data that is given to the classifier to learn from must *not* be used to test it.** The test data must be comprised of representative, but different, examples. It's the only way to assess how well the classifier **generalizes** to new data that it hasn't yet seen (which, of course, is the whole point).

Splitting the difference

Okay, so given that we have to split our precious labeled examples into two sets, one for training and one for testing, how much do we devote to each? It turns out that there are some sophisticated techniques (beyond the scope of this book, but stay tuned for Volume Two) in which we can cleverly re-use portions of the data for different purposes, and effectively make use of nearly all of it for training.

But for our introductory approach here, we'll just use a **rule of thumb: 70% for training data, and the other 30% for test data**.

As I mentioned earlier, we'll normally **shuffle** the rows randomly before dividing them into these two groups, just in case there's any pattern to the order in which they appear. For example, in our NFL fan data set, it might turn out that the data came to us sequenced in a way such that people living on the east coast were at the beginning of the `DataFrame` and those living out west were at the end. Any arrangement like this would spell doom for our classification endeavor. For one thing, we wouldn't be training on any west coast people, and so our classifier would be oblivious to what those data points looked like. For another thing, we'd only be using west coasters to *test* our classifier, meaning that whatever accuracy measure we computed is likely to be way off. **Randomizing** the data is the sure way around this.

Here's some code to create training and test sets. The `.sample()` method of a `DataFrame` lets you choose some percentage of its rows randomly. Its `frac` argument is a number between 0 and 1 and specifies what fraction of the rows you want. Using the above rule of thumb, let's choose 70% of them for our training data:

```
training = fans.sample(frac=.7)
print(training)
```

```
    age        hometown current_residence  years_in_residence      team
8    52       Arlington    Fredericksburg                  17    Ravens
15   63        New York           Orlando                  17    Giants
11   29  Fredericksburg           Seattle                  21  Seahawks
19   32       Warrenton           Orlando                   1    Ravens
2    47       Allegheny       Chevy Chase                  13    Ravens
7    31       Warrenton   Charlottesville                   6      Jets
0    29       Warrenton         Warrenton                  29    Ravens
5    32       Warrenton        Winchester                  11    Ravens
4    51   Orange County          Dumfries                  23    Ravens
9    60  Tyson's Corner      Falls Church                   4   Cowboys
10   17  Fredericksburg    Fredericksburg                  17    Ravens
1    39         Lubbock         Arlington                   2   Cowboys
13   35  Mechanicsville           Orlando                   8    Ravens
6    39        Dumfries             Miami                   5    Ravens
```

Notice that the numeric index values (far left) are in no particular order, since that's the point of taking a random sample. Also notice that there are only 14 rows in this `DataFrame` instead of the full 20 that were in `fans`.

Now, we want our test set. The trick here is to say: "give me all the rows of `fans` that were *not* selected for the `training` set." By building a query with the squiggle operator ("`~`", meaning "not") in conjunction with the "`.isin()`" method, we can create a new `DataFrame` called "`test`" that has exactly these rows:

```
test = fans[~fans.index.isin(training.index)]
print(test)
```

```
    age        hometown current_residence  years_in_residence      team
3    31       Littleton         Littleton                   3   Broncos
12   37        Richmond          Richmond                  37    Ravens
14   19        New York          New York                  19    Giants
16   42      Alexandria          Manassas                  10    Ravens
17   22       Allegheny        Harrisburg                   1  Steelers
18   38  Mechanicsville    Fredericksburg                   6    Ravens
```

That code says, in English: "create a new variable `test` that contains only those rows of `fans` whose index is *not* present in any of the `training` `DataFrame`'s indices." As you can verify through visual inspection, the result does have exactly the 6 rows that were missing from `training`.

26.3 "The prior"

One more piece of lingo before we dive into a particular classification technique next chapter. And that's known as "the **prior**" of a data set.

The term comes from something called **Bayesian reasoning**, which is a whole subject (and a super cool one!) in its own right. All you need to know here is the concept of two different quantities: the **prior**, and the **posterior**.

In common usage, the word "prior" means "beforehand," and so it does here: the prior is your best judgment about what the target value of a new example might be *before you actually look at the feature values in that example.* "Posterior," on the other hand, means "afterwards," and means your best judgment about the target value after duly taking into consideration all the feature values.

For example, you may have noticed that in my made-up data set, above, I had a lot of Ravens fans. This is because I live in the D.C. area, and happen to know a lot of Ravens fans. Out of my 20 labeled examples, a whopping twelve of them, in fact, had `Ravens` as their value in the `team` column.

Thus, consider the following question. Suppose you knew nothing about a person except that they were one of Stephen's friends. Which NFL team do you think they'd support? Assuming this data set is representative of Stephen's friends, you'd say: "I'd predict they'd be a Ravens fan, and I'd estimate that I'd have about a 60% chance of being right ($\frac{12}{20}$)." This is the *prior.* You're not taking into account anything about their age, where they were born, *etc.*; in fact, you weren't even told those things. Instead, you're just "using the prior" and treating everyone the same.

It would be a different story if I told you that this person was born in New York City. Then you might squint your eyes at my data set and realize that there are only two New Yorkers in it, and neither one is a Ravens fan: they're both Giants fans! Now you might very well move away from your prior assumption. "Sure, most of Stephen's friends are Ravens fans, so 'Ravens' is a reasonable guess,

but now that you've told me they're from NY, that very well might change my mind. Now, my guess is 'Giants'."

I keep saying "might" and "may" because different kinds of classifiers work in different ways. Some of them may choose to take advantage of some features but not others; some may just stick with the prior in certain situations. The notion of "the prior" is mainly useful as a baseline for comparison: it's the best you can do given no other possibly correlating information. The name of the game in classification, of course, is to intelligently *use* that other information to make more informed guesses, and to beat the prior. One of many ways to approach this is the decision tree classification algorithm, which we'll look at in detail next.

Chapter 27

Decision trees for classification (1 of 2)

So far our classification picture has been very general. We haven't said anything about how our classifier might actually *work*; we've just said that given values for each of the features, it will render a prediction about what the label will be.

In chapters 27 and 28, we'll study one particular algorithm for classification in machine learning: the **decision tree** algorithm. Not only does it make a good introductory technique because of its intuitive appeal, and not only can it classify pretty well in its own right, but it also serves as the basis for a more sophisticated, state-of-the-art classification method called "**random forest**" which we'll explore in Volume Two of this series.

27.1 A working example

Here's a (fictitious) domain problem that we'll use to demonstrate the principles in this chapter and the next. Say we own a videogame business, and we want to send full-color product catalogs to unsuspecting college students, so that they will buy our games and keep us in business (while meanwhile failing out of school due to playing games all the time).

Now full-color catalogs are expensive to print and ship, so we want

to be smart about this. We definitely don't want to send a bunch of catalogs to students who aren't likely buyers; that would run our business into the ground. Instead, we'd like to identify the subset of students who probably gamers, and send catalogs to only *those* students.

Suppose that through nefarious means, we have acquired the following data set:

	Major	Age	Gender	VG
0	PSYC	22	F	No
1	MATH	20	F	No
2	PSYC	19	F	No
3	CPSC	20	M	Yes
4	MATH	18	M	Yes
5	CPSC	20	F	No
6	CPSC	19	O	No
7	CPSC	17	M	Yes
8	PSYC	18	F	No
9	CPSC	20	F	No
10	MATH	18	F	No
11	CPSC	22	F	Yes
12	MATH	21	M	No
13	CPSC	23	M	Yes
14	PSYC	17	M	Yes
15	CPSC	18	F	No
16	PSYC	19	F	Yes

Each row represents one college student, with three features. The first is their major – `PSYC` (Psychology), `MATH` (Mathematics), or `CPSC` (Computer Science). (For simplicity, we'll say these are the only three possibilities, since your author happens to like them the best.) The second is their age (numeric), and the third is their gender: male, female, or other. The last column is our target: *whether or not this student is a videogamer.* Glance over this `DataFrame` for a moment.

Eyeing the prior

As you remember from section 26.3, before we even think about features, we might take a minute to just look at the target variable itself. We ask ourselves "given no other information about a student, what would be our gut feel about their videogame status?" Our pal the `.value_counts()` method is perfect to compute this:

```
print(students.VG.value_counts())
```

```
N    10
Y     7
Name: VG, dtype: int64
```

So if we're smart, we'd guess "no" for such mysterious persons, but we could only expect to be right about $\frac{10}{17}^{\text{ths}}$, or 59%, of the time. Not great, although better than a coin flip.

Sticking with categorical features

Now it turns out that decision trees work best with all categorical features, not a mix of categorical and numeric. So for now, we're going to simply classify each of our students into three buckets: "`young`" (18 or younger), "`middle`" (19-21), and "`old`" (22+).[1] For the moment, don't ask why we chose three age categories instead of two or four, and don't ask why we chose those particular split points. We just did. More on that later.

Our training data now looks like this:

[1] Believe it or not, a time will come in your life when 22 years of age does not remotely seem "old." For undergrads, though, I can see why 22 would seem on the grey side, the Taylor Swift song notwithstanding.

	Major	Age	Gender	VG
0	PSYC	old	F	No
1	MATH	middle	F	No
2	PSYC	middle	F	No
3	CPSC	middle	M	Yes
4	MATH	young	M	Yes
5	CPSC	middle	F	No
6	CPSC	middle	O	No
7	CPSC	young	M	Yes
8	PSYC	young	F	No
9	CPSC	middle	F	No
10	MATH	young	F	No
11	CPSC	old	F	Yes
12	MATH	middle	M	No
13	CPSC	old	M	Yes
14	PSYC	young	M	Yes
15	CPSC	young	F	No
16	PSYC	middle	F	Yes

and we're now officially ready to consider decision trees.

27.2 Decision Trees

First, let's get our head around what a decision tree *is*. Our inaugural example is shown in Figure 27.1. The first thing you'll notice is that it has a branching structure that branches...down. I'm not sure why Data Scientists draw trees growing *down* while the rest of the world (including trees themselves: look outside if you don't believe me) has them growing *up*, but this is the convention so we'll just deal with it. To make it even more comical, the oval at the top of the tree is called the **root** of the tree. Really.

Continuing full bore with the botany analogy, the lines connecting the various shapes are, as you might suspect, called **branches**, and the darker rectangles are called **leaves**. One non-botanic bit of lingo is the name for the other ovals: they're called **nodes**.

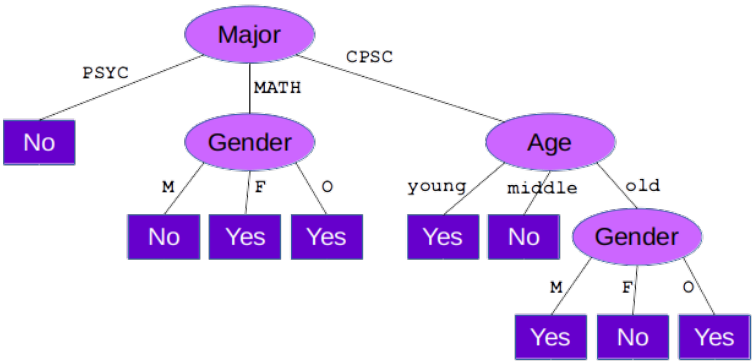


Figure 27.1: A decision tree (not a particularly good one, as it'll turn out) for the videogame data set.

Classifying with a decision tree

Okay. Now what does a decision tree "mean?" Geekily-put, it's the pictorial codification of an algorithm for classification. Not so geekily, it's a map that tells your classifier what rules to follow as it forms its prediction for an example data point.

You simply start at the root, considering feature values at each node, and following the matching branch down the tree. When you reach a leaf, the prediction you give is written on the leaf node. It's that simple.

☞ First example: suppose we have a 24-year-old male Psychology major. We want to know whether he's likely to play videogames. The decision tree in Figure 27.1 tells us to first consider his Major, since that's the root. Now because this guy's major is `PSYC`, we take the left branch and are immediately done: we've already reached a leaf. Our prediction for this guy will be `No`, he probably doesn't play videogames.

☞ Second example: we have an 18-year-old Math major who doesn't identify with either of the binary genders. Starting again at the root, we now follow the middle branch for `MATH`. Now, we look at the person's Gender. Since it is `O`, we follow

the right branch, and give a prediction of `Yes`: we predict they *do* play videogames.

☞ Third example: we now have a 22-year-old female Computer Science major. Do we think she would play videogames? The root tells us to look at her Major first, which means we go right; then we look at her Age, and since she's positively ancient we go right again; and finally, her Gender tells us to predict `No`, she's probably not a gamer.

Most students find this process very straightforward. In the next chapter, we'll look at two key questions: first, how to turn a diagram like Figure 27.1 into Python code? And second, what's the best way to make a *good* tree – *i.e.*, one that makes as many successful predictions as possible?

Chapter 28

Decision trees for classification (2 of 2)

28.1 Decision Trees in Python

Our decision tree pictures from chapter 27 were quite illustrative, but of course to actually automate something, we have to write code rather than draw pictures. What would Figure 27.1 (p. 273) look like in Python code? It's actually pretty simple, although there's a lot of nested indentation. See if you can follow the flow in Figure 28.1.

Here we're defining a function called `predict()` that takes three arguments, one for each feature value. The eye-popping set of `if`/`elif`/`else` statements looks daunting at first, but when you scrutinize it you'll realize it perfectly reflects the structure of the purple diagram. Each time we go down one level of the tree, we indent one tab to the right. The body of the "`if major == 'PSYC':`" statement is very short because the left-most branch of the tree (for Psychology) is very simple. The "`elif major == 'CPSC':`" body, by contrast, has lots of nested internal structure precisely because the right-most branch of the tree (for Computer Science) is complex. *Etc.*

```
def predict(major, age, gender):
    if major == 'PSYC':
        return 'No'
    elif major == 'MATH':
        if gender == 'M':
            return 'No'
        elif gender == 'F' or gender == 'O':
            return 'Yes'
    elif major == 'CPSC':
        if age == 'young':
            return 'Yes'
        elif age == 'middle':
            return 'No'
        elif age == 'old':
            if gender == 'M' or gender == 'O':
                return 'Yes'
            elif gender == 'F':
                return 'No'
```

Figure 28.1: A Python implementation of the decision tree in Figure 27.1.

If we call this function, it will give us exactly the same predictions we calculated by hand on p. 273:

```
print(predict('PSYC','M','old'))
print(predict('MATH','O','young'))
print(predict('CPSC','F','old'))
```

```
No
Yes
No
```

28.2 Decision Tree induction

Okay, so now we understand what a decision tree is, and even how to code one up in Python. The key question that remains is: how do we figure out what tree to build?

There are lots of different choices, even for our little videogame example. We could put any of the three features at the root. For

each branch from the root, we could put either of the other features, or we could stop with a leaf. And the leaf could be a `Yes` leaf or a `No` leaf. That's a lot of "coulds." How can we know what a *good* tree might be – *i.e.*, a tree that classifies new points more or less correctly?

The answer, of course, is to take advantage of the training data. It consists of labeled examples that are supposed to be our guide. Using the training data to "learn" a good tree is called **inducing** a decision tree. Let's see how.

"Greedy" algorithms

Our decision tree induction algorithm is going to be a **greedy** one. This means that instead of looking ahead and strategizing about future nodes far down on the tree, we're just going to grab the immediate best-looking feature at every individual step and use that. This won't by any means guarantee us the best possible tree, but it will be quick to learn one.

An illustration to help you understand greedy algorithms is to think about a strategy game like chess. If you've ever played chess, you know that the only way to play well is to think ahead several moves, and anticipate your opponent's probable responses. You can't just look at the board naïvely and say, "why look at that: if I move my rook up four squares, I'll capture my opponent's pawn! Let's do it!" Without considering the broader implications of your move, you're likely to discover that as soon as you take her pawn, she turns around and takes your rook because she's lured you into a trap.

A *greedy* algorithm for chess would do exactly that, however. It would just grab whatever morsel was in front of it without considering the fuller consequences. That may seem really dumb – and it is, for chess – but for certain other problems it turns out to be a decent approach. And decision tree induction is one of those.

The reason we resort to a greedy algorithm is that for any real-sized data set, *the number of possible trees to consider is absolutely overwhelming.* There's simply not enough time left in the universe

to look at them all – and that's not an exaggeration. So you have to find *some* way of picking a tree without actually contemplating every one, and it turns out that grabbing the immediately best-looking feature at each level is a pretty good way to do that.

Choosing "the immediate best" feature

Now what does that mean, anyway: "choosing the immediate best feature?" We're going to define it as follows: the best feature to put at any given node is *the one which, if we did no further branching from that node but instead put only leaves below it, would classify the most training points correctly.* Let's see how this works for the videogame example.

Our left-most feature in the `DataFrame` is Major, so let's consider that one first. Suppose we put Major at the root of the tree, and then made each of its branches lead to leaves. What value should we predict for each of the majors? Well, we can answer that with another clever use of `.value_counts()`, this time conjoining it with a call to `.groupby()`. Check out this primo line of code:

```
students.groupby('Major').VG.value_counts()
```

Stare hard at that code. You'll realize that all these pieces are things you already know: we're just combining them in new ways. That line of code says "take the entire `students DataFrame`, but treat each of the majors as a separate group. And what should we do with each group? Well, we count up the values of the `VG` column for the rows in that group." The result is as follows:

```
Major  VG
PSYC   No     3
       Yes    2
MATH   No     3
       Yes    1
CPSC   No     4
       Yes    4
Name: VG, dtype: int64
```

We can answer "how many would we get right?" by reading right off that chart. For the PSYC majors, there are two who play videogames and three who do not. Clearly, then, if we presented a Psychology major to this decision tree, it ought to predict 'No', and that prediction would be correct for 3 out of the 5 Psychology majors on record. For the MATH majors, we would again predict 'No', and we'd be correct 3 out of *4* times. Finally, for the CPSC majors, we have 4 Yeses and 4 Nos, so that's not much help. We essentially have to pick randomly since the training data doesn't guide us to one answer or the other. Let's choose 'Yes' for our Computer Science answer, just so it's different than the others. The best one-level decision tree that would result from putting Major at the top is therefore depicted in Figure 28.2. It gets **ten** out of the seventeen training points correct (59%). Your reaction is probably "Big whoop – we got that good a score just using the prior, and ignoring all the features!" Truth. Don't lose hope, though: Major was only one of our three choices.

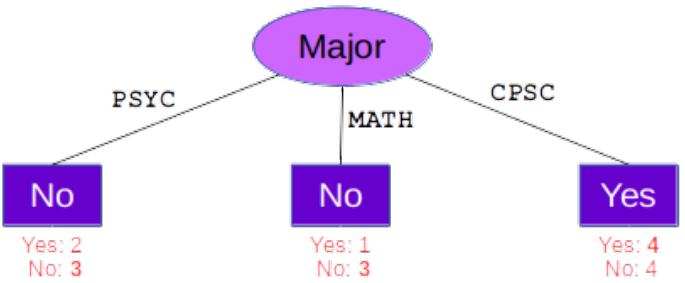


Figure 28.2: A one-level decision tree if we put the Major feature at the root – it would classify **ten** of the seventeen training points correctly.

Let's repeat this analysis for the other two features and see if either one fares any better. Here's the query for Age:

```
students.groupby('Age').VG.value_counts()
```

This yields:

```
Age      VG
middle   No     6
         Yes    2
old      Yes    2
         No     1
young    No     3
         Yes    3
Name: VG, dtype: int64
```

Making the sensible predictions at the leaves based on these values gives the tree in Figure 28.3. It gets **eleven** points right (65%) – a bit of an improvement.

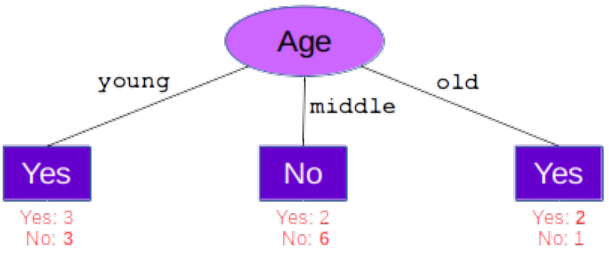


Figure 28.3: A one-level decision tree if we chose the Age feature for the root – it would classify **eleven** of the seventeen training points correctly.

Finally, we could put Gender at the root. Here's the query for it:

```
students.groupby('Gender').VG.value_counts()
```

```
Gender   VG
F        No     8
         Yes    2
M        Yes    5
         No     1
O        No     1
Name: VG, dtype: int64
```

Paydirt! Splitting on the Gender feature first, as shown in Figure 28.4, gets us a whopping **fourteen** points correct, or over 82%.

This is clearly the winner of the three. And since we're being greedy and not bothering to look further downstream anyway, we hereby elect to put Gender at the root of our tree.

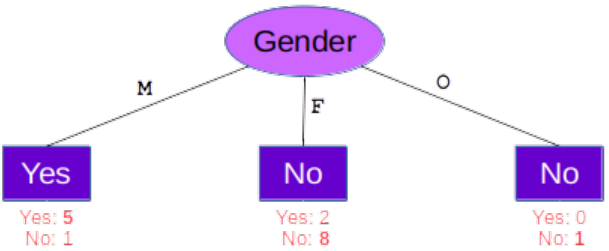


Figure 28.4: A one-level decision tree if we chose the Gender feature for the root. It would classify **fourteen** of the seventeen training points correctly – easily the best of the three choices.

It's worth taking a moment to look at those `.value_counts()` outputs and see if you can develop some intuition about why Gender worked so much better at the root than the other two features did. The reason is that for this data set, Gender split the data into groups that were more **homogeneous** than the other splits gave. "Homogeneous" here means that each group was more "pure," or put another way, more lopsided towards one of the labels. Gender gave us a 5-to-1 lopsided ratio on the `M` branch, and an even more lopsided 2-to-8 ratio on the `F` branch. Intuitively, this means that Gender really is correlated with videogame use, and this shows up in purer splits. Contrast this with the situation when we split on Major first, and we ended up with a yucky 4-to-4 ratio on the `CPSC` branch. An even split is the worst of all possible worlds: here, it means that learning someone's a Computer Science major doesn't tell you jack about their videogame use. That in turn means it's pretty useless to split on.

Lather, rinse, repeat

So far, we've done all that work just to figure out which feature to put at the root of our tree. Now, we progress down each of the branches and do the exact same thing: figure out what to put at

each branch. We'll continue on and on like this for the entire tree. It's turtles all the way down.

Let's consider the *left* branch of Figure 28.4. What do we do with males? There are now only two remaining features to split on. (It wouldn't make sense to split on Gender again, since the only people who will reach the left branch are males anyway: there'd be nothing to split on.)

Thus we could put either Major or Age at that left branch. To figure out which one is better, we'll do the same thing we did before, only with one slight change: now, we need to consider *only males* in our analysis.

We augment our primo line of code from above with a query at the beginning, so that our counts include only males:

```
students[students.Gender=="M"].groupby('Major').VG.value_counts()
```

```
Major  VG
CPSC   Yes    3
MATH   No     1
       Yes    1
PSYC   Yes    1
Name: VG, dtype: int64
```

Wow, cool: the CPSC and PSYC folks are perfectly homogeneous now. If we end up deciding to split on Major here, we can put permanent dark purple squares for each of those majors simply declaring "Yes." In all, splitting here gives us 5 out of 6 correct. The tree-in-progress we'd end up with is in Figure 28.5.

Our other choice, of course, is to split on Age instead:

```
students[students.Gender=="M"].groupby('Age').VG.value_counts()
```

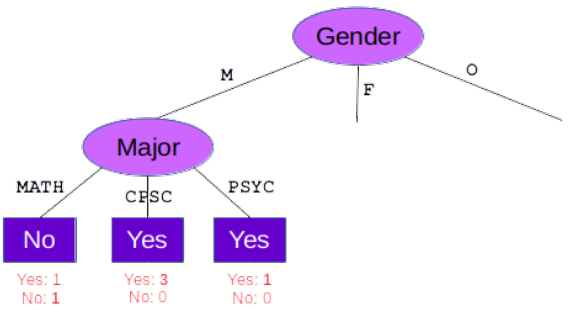


Figure 28.5: The tree-in-progress if we choose to split on Major in the male branch.

```
Age      VG
middle   No     1
         Yes    1
old      Yes    1
young    Yes    3
Name: VG, dtype: int64
```

Again, 5 out of 6 correct. Here, `middle`-aged students are the only heterogeneous group; the `old` folks and `young`-uns are clean splits. With this choice, our tree would appear as in Figure 28.6.

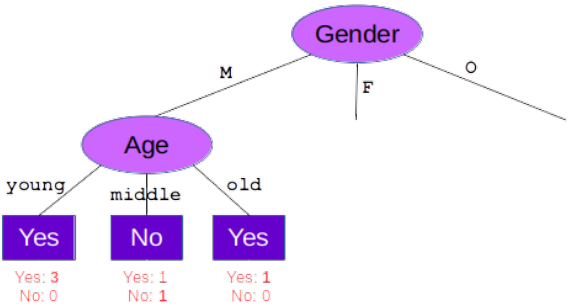


Figure 28.6: On the other hand, the tree-in-progress if we choose to split on Age in the male branch instead.

So at this point, since splitting on either feature and then stopping would give us exactly 5 out of 6 points correct, we just flip a coin. I

just flipped one, and it came out tails (for Age) – hope that's okay with you.

Finishing up the left branch

The two "Yes" leaves in Figure 28.6 are now set in stone, since every single `young` male in our training set was indeed a videogamer, as was every `old` male. Now we just have to deal with the `middle` branch.

Only one feature now remains to split on – Major – so we'll do that, and produce the result in Figure 28.7. There's exactly one `middle`-aged male `MATH` major in the original `DataFrame` (line 12, p. 272), and he's labeled "No," so we'll guess "No" in the `MATH` branch. Similarly, we have one data point to guide us for `CPSC` majors (line 3), so we'll predict "Yes" in this case. The `PSYC` branch presents a conundrum, though: our data set doesn't have *any* `middle`-aged male Psychology majors, so how do we know what to guess in this case?

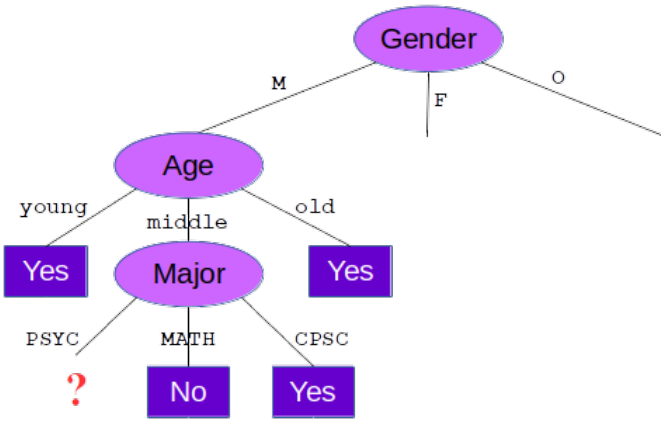


Figure 28.7: Going one level further down after splitting on Age for males. We have data for `middle`-aged `CPSC` and `MATH` males...but what to do with `middle`-aged `PSYC` males?

The best way to handle this is to fall back to a more general case where you *do* have examples. It's true that we have no training

points for `middle`-aged male Psychology majors, but we *do* have points for `middle`-aged males-in-general, and we discovered that 5 out of 6 of them were gamers. So it makes sense to default to "Yes" in the `PSYC` branch of this part of the tree, even though we don't have any data points exactly like that. So that's what we'll do. The finished left branch is depicted in Figure 28.8.

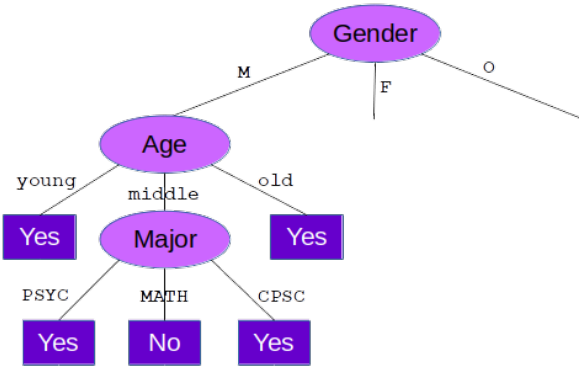


Figure 28.8: The decision tree we're in the process of inducing, with the left branch entirely completed.

Finishing up the rest of the tree

The rest of the process is just the same stuff done over and over.[1] At each branch of the tree, we take the subset of the training points that remain (*i.e.*, the training points that match the path from the root thus far, and are therefore applicable), and decide what to branch on next. When we get to a completely homogeneous group, we stop and put a leaf there. The end result of all these efforts is

[1] If you're wondering whether there's a way to automate this, the answer is a resounding **yes**! There are many packages in Python and other languages which will automatically build a decision tree from a training set; the `DecisionTreeClassifier` from the `scikit-learn` package is one of them. This exercise of learning how to build a decision tree manually is so you can understand the concepts of what's going on under the hood – kind of like you learn how to add numbers in grade school even though you'll normally use a calculator later on in life.

the final decision tree for the videogame data set, in Figure 28.9, and its Python equivalent in Figure 28.10.

One interesting aspect of our final tree is the female→`PSYC`→`middle`-aged branch. You'll see that this leaf is labeled "Yes(?)" in the diagram. Why the question mark? Because this is the one case where we have a **contradiction** in our training data. Check out lines 2 and 16 back on p. 272. They each reflect a `middle`-aged female Psychology major, but with *different* labels: the first one is not a videogame player, but the second one is.

I always thought the term "contradiction" was amusing here. Two similar people don't have exactly the same hobbies – so what? Is that really so surprising? Do all middle-aged female Psychology majors have to be identical?

Of course not. But you can also see things from the decision tree's point of view. The only things it knows about people are those three attributes, and so as far as the decision tree is concerned, the people on lines 2 and 16 really *are* indistinguishable. When contradictions occur, we have no choice but to fall back on some sort of majority-rules strategy: if out of seven otherwise-identical people, two play videogames and five do not, we'd predict "No" in that branch. In the present case, we can't even do *that* much, because we have exactly one of each. So I'll just flip a coin again. (`*flip*`) It came up heads, so we'll go with "Yes."

Notice that in this situation, the resulting tree will actually misclassify one or more *training* points. If we called our function in Figure 28.10 and passed it our person from line 2 (`'PSYC'`, `'middle'`, `'F'`), it would return `"Yes"` even though line 2 is not a gamer. Furthermore, contradictions are the *only* situation in which this will ever happen; if the data is contradiction-free, then every training point will be classified correctly by the decision tree.

Paradoxically, it turns out that's not necessarily a good thing, as we'll discover in Volume Two of this series. For now, though, we'll simply declare victory.

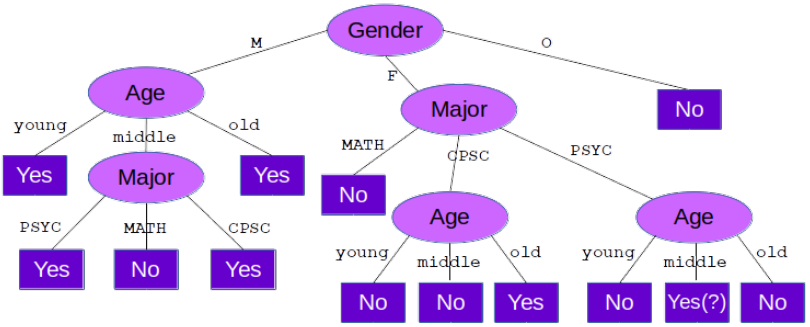


Figure 28.9: The final decision tree for the videogame data set.

```
def predict(major, age, gender):
    if gender == 'M':
        if age == 'young':
            return 'Yes'
        elif age == 'middle':
            if major == 'PSYC' or major == 'CPSC':
                return 'Yes'
            elif major == 'MATH':
                return 'No'
        elif age == 'old':
            return 'Yes'
    elif gender == 'F':
        if major == 'MATH':
            return 'No'
        elif major == 'CPSC':
            if age == 'young' or age == 'middle':
                return 'No'
            elif age == 'old':
                return 'Yes'
        elif major == 'PSYC':
            if age == 'young' or age == 'old':
                return 'No'
            elif age == 'middle':
                return 'Yes'  # Here's our "contradiction"
    elif gender == 'O':
        return 'No'
```

Figure 28.10: The final decision tree for the videogame data set, as a Python function.

Chapter 29

Evaluating a classifier

Once we've built a classifier – whether it's a decision tree or any other kind – the next step is to evaluate it to see how well it performs. This is sometimes called the classifier's **performance**. It will determine whether we deem it accurate enough to set it loose in the field, and if so, how accurate we can expect its predictions to be.

At the risk of repeating myself (recall my stern lecture from section 26.2) you must evaluate your classifier by testing it on data that was *not* used to train it. On p. 265 we learned how to randomly divide a data set into separate training and test sets.

Suppose we've done that. Suppose the `students` `DataFrame` from chapters 27 and 28 was the result of randomly choosing 70% of the labeled examples from an original data set, and that we have preserved the remaining 30% of the rows in a `DataFrame` called `students_test`, which contains:

	Major	Age	Gender	VG
0	CPSC	young	F	No
1	CPSC	old	O	No
2	MATH	old	F	No
3	CPSC	middle	M	No
4	PSYC	middle	F	Yes
5	PSYC	young	F	No
6	MATH	old	M	Yes
7	CPSC	middle	M	Yes

Our question is: "how well does our classifier do on this test data?"

29.1 What "doing well" means

The most common (and simplest) way to measure a classifier's performance is to simply count how many of the test points it correctly classifies, and divide by the total number of test points. This gives us the classification **accuracy** as a fraction between 0 and 1 (or, if we want to multiply by 100, a "percentage accuracy" from 0% to 100%.) It's possible to do this because, as you'll remember, our test data is comprised of *labeled* examples, just like our training data is. Therefore, we know the "right answer" for each test point, and we can simply compare it to our classifier's prediction.

Even though this is the most common approach, it's worth taking a moment to consider alternatives. The key assumption of this accuracy measure is that *all kinds of prediction errors are equal.* In the videogame case, we're saying that mistakenly labeling a videogamer as a non-videogamer is "just as bad" as mistakenly labeling a non-videogamer as a videogamer. And that might be just the right thing for our gaming company to do.

But consider other settings. Suppose that our classifier's inputs are features from an MRI image, and our prediction is "cancer" or "no cancer." Now, it's a much different story. Mistakenly predicting that a certain patient has cancer when they actually don't might throw a needless scare into them. That's bad. But it's far worse

in the other direction: mistakenly giving a clean bill of health to a patient who actually has early stage cancer risks losing a life. In cases like this, we would need to penalize our classifier more harshly for false negatives than for false positives.

It's also a different story when the labels aren't equally represented. Recall the NFL fan prediction problem from Figure 26.1 (p. 262). Consider if we performed fan prediction in a city like Dallas, which is comprised of (say) 99% Cowboys fans and only 1% Ravens fans. If we were to penalize a classifier equally for mistaken-Cowboy-predictions and mistaken-Ravens-predictions, a one-line classifier could earn a pretty good score:

```
def predict(age, hometown, current_residence, yrs_in_residence):
    return "Cowboys"
```

It's not even worth trying hard to ferret out the few Ravens fans if we're going to be docked a full point every time we dare to predict one. They're just too rare. The only way to get a classifier to be bold and try to identify the tiny population of Ravens fans is to penalize it more heavily for missing them than for falsely identifying them.

Anyway, for the rest of this chapter, we'll use the vanilla "count all prediction mistakes equally" approach, but it's worth remembering that this doesn't make sense in all situations.

29.2 Calculating accuracy in Python

Calculating your classifier's accuracy is actually a snap. Once your classifier's code is in a function, you just need a loop.

Return to the videogame example from last chapter, and the decision tree classifier we wrote on p. 287. We'll use a counter variable, initialized to zero, that will keep track of our number of correct predictions. We'll then loop through each row of the test set, feeding that row's features to the classifier function. If the return value

from the classifier matches the value of that row's target, ka-ching! We increment our counter to increase our score. If it doesn't, we don't. At the end, we divide by the number of test points to get our percentage. Simple!

```
count = 0
for row in students_test.itertuples():
    if predict(row.Major, row.Age, row.Gender) == row.VG:
        count += 1

accuracy = count / len(students_test) * 100
print("Our accuracy on the test set was {}%.".format(accuracy,
    count, len(students_test)))
```

```
Our accuracy on the test set was 87.5%.
```

If we want more detail, we could print a message for each prediction, and flag the incorrect ones for easy identification:

```
count = 0
for row in students_test.itertuples():
    if predict(row.Major, row.Age, row.Gender) == row.VG:
        print("  Predicted {}/{}/{} right!".format(row.Major,
            row.Age, row.Gender))
        count += 1
    else:
        print("X Predicted {}/{}/{} wrong. :(".format(row.Major,
            row.Age, row.Gender))

accuracy = count / len(students_test) * 100
print("Our accuracy on the test set was {}% ({}/{}).".format(
    accuracy, count, len(students_test)))
```

```
  Predicted CPSC/young/F right!
  Predicted CPSC/old/O right!
  Predicted MATH/old/F right!
X Predicted CPSC/middle/M wrong. :(
  Predicted PSYC/middle/F right!
  Predicted PSYC/young/F right!
  Predicted MATH/old/M right!
  Predicted CPSC/middle/M right!
Our accuracy on the test set was 87.5% (7/8).
```

Not too shabby. As you can see, the only test point we missed was the male `middle`-aged `CPSC` major, which our classifier figured would be a videogamer. Live and learn.

The data size here is laughably small so that I can fit everything on the page. But it's worth considering these three quantities anyway:

Classifier's performance on training set	94.1% (16/17)
Classifier's performance on test set	82.5% (7/8)
Just using the prior on test set	62.5% (5/8)

These three quantities will nearly always be in this order from top to bottom. When we test our classifier on the very data it was trained on, we get an inflated view of its accuracy – for decision trees, recall, it will always be 100% less any contradictions. Testing it on the data it has not yet seen gives the truer (more realistic) picture. Finally, your classifier had better outperform just using the prior (here, choosing "No" because the majority of training points were "No") or this whole thing is a pretty useless enterprise!

Index