Stephen Davies, Ph.D.
University of Mary Washington

THE CRYSTAL BALL INSTRUCTION MANUAL

VOLUME ONE: INTRODUCTION TO DATA SCIENCE

version 1.1

The Crystal Ball Instruction Manual

Volume One: Introduction to Data Science

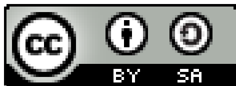
version 1.2

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

Copyright © 2025 Stephen Davies.

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

Permission is granted to copy, distribute, transmit and adapt this work under a Creative Commons Attribution-ShareAlike 4.0 International License:



http://creativecommons.org/licenses/by-sa/4.0/

If you are interested in distributing a commercial version of this work, please contact the author at stephen@umw.edu.

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

Cover art copyright © 2020 Elizabeth M. Davies.

Contents

Contents i

1 Introduction 1

2 A trip to Jupyter 9

3 Three kinds of atomic data 13

4 Memory pictures 25

5 Calculations 31

6 Scales of measure 43

7 Three kinds of aggregate data 53

8 Arrays in Python (1 of 2) 61

9 Arrays in Python (2 of 2) 73

10 Interpreting Data 89

11 Assoc. arrays in Python (1 of 3) 103

12 Assoc. arrays in Python (2 of 3) 109

13 Assoc. arrays in Python (3 of 3) 123

14 Loops 135

15 EDA: univariate **145**

16 Tables in Python (1 of 3) **169**

17 Tables in Python (2 of 3) **177**

18 Tables in Python (3 of 3) **187**

19 EDA: bivariate (1 of 2) **193**

20 EDA: bivariate (2 of 2) **201**

21 Branching **211**

22 Functions (1 of 2) **223**

23 Functions (2 of 2) **235**

24 Recoding and transforming **243**

25 Machine Learning: concepts **255**

26 Classification: concepts **261**

27 Decision trees (1 of 2) **269**

28 Decision trees (2 of 2) **275**

29 Evaluating a classifier **289**

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.

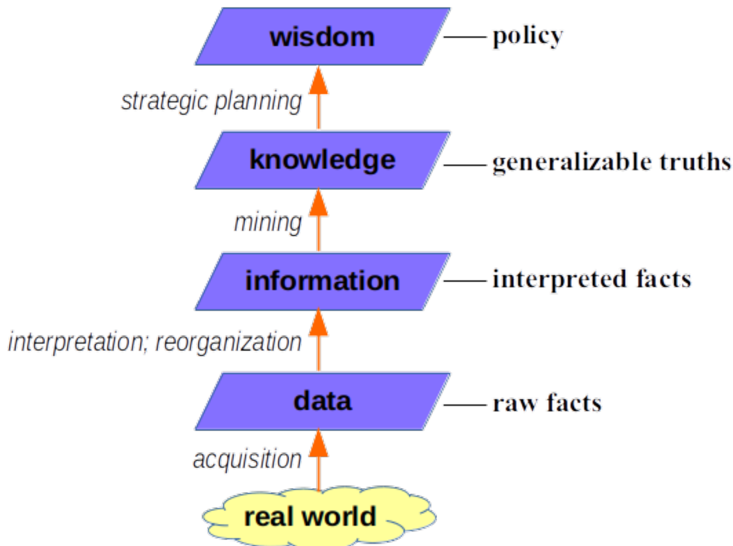


Figure 1.1: The data-to-wisdom hierarchy.

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`.

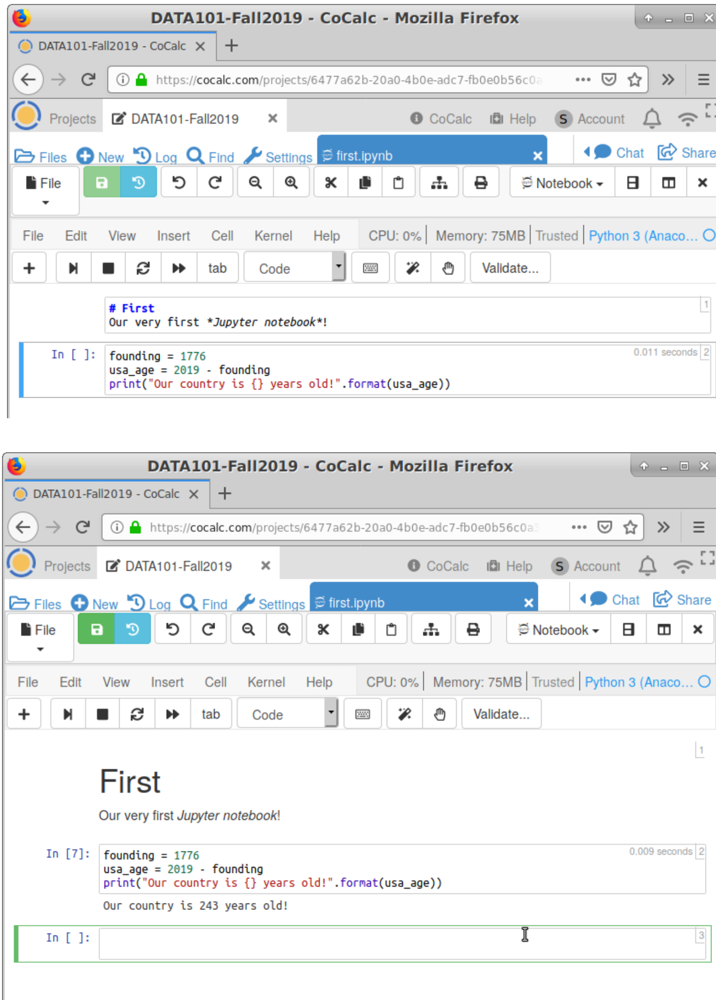


Figure 2.1: A Jupyter Notebook with one Markdown cell and one Code cell. In the top image, the two cells have been edited but not yet "run" – hence the Markdown formatting is unrendered and the code has not been executed. The bottom pane shows both cells after the use has chosen "Run All" from the "Cell"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

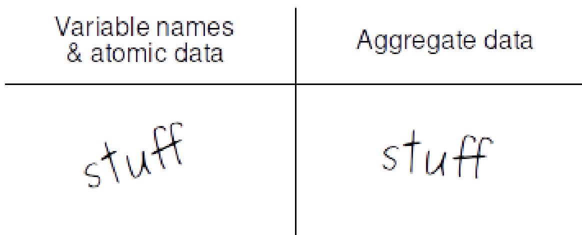
> **The code itself is only a means to an end. The purpose of the code is to read or write what's in memory.**

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.

Variable names & atomic data | Aggregate data
pin_count 844
username "Bekka Palmer"

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"
```

Variable names & atomic data | Aggregate data
board_name "Things to Make"
pin_count 844
username "Bekka Palmer"
avg_impressions 1739.3

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!'
```

	Variable names & atomic data	Aggregate data
board_name	"Things to Make"	
pin_count	845	
username	"Bekka Palmer"	
avg_impressions	2000.97	
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`.

	Variable names & atomic data	Aggregate data
board_name	"Things to Make"	
pin_count	999.635	
username	11	
avg_impressions	2000.97	
another_board	"Pink!"	

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!
```

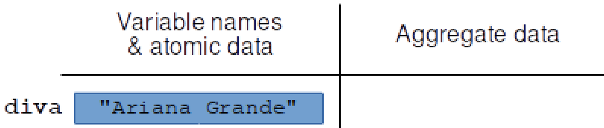
Wait...did the "`diva.upper()`" part just not work? Did it get skipped? Did we do it wrong somehow?

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?
```

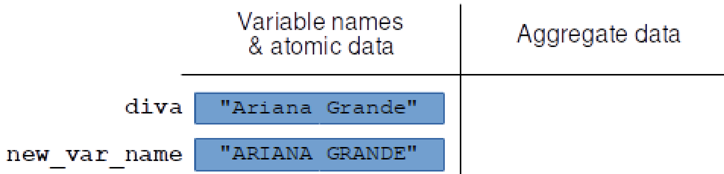
The root cause of this and practically all perplexing Python printing can be discovered by consulting the memory picture. Here's how it starts out when we first define `diva`:



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"`).

key	value
"Philadelphia"	"Eagles"
"Baltimore"	"Ravens"
"Dallas"	"Cowboys"
"New England"	"Patriots"

key	value
"Sheryl Swoops"	22
"Michael Jordan"	23
"Mia Hamm"	9
"John Elway"	7
"LeBron James"	23
"Alex Morgan"	13
"Dan Marino"	13

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 53$^{\text{rd}}$ 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-

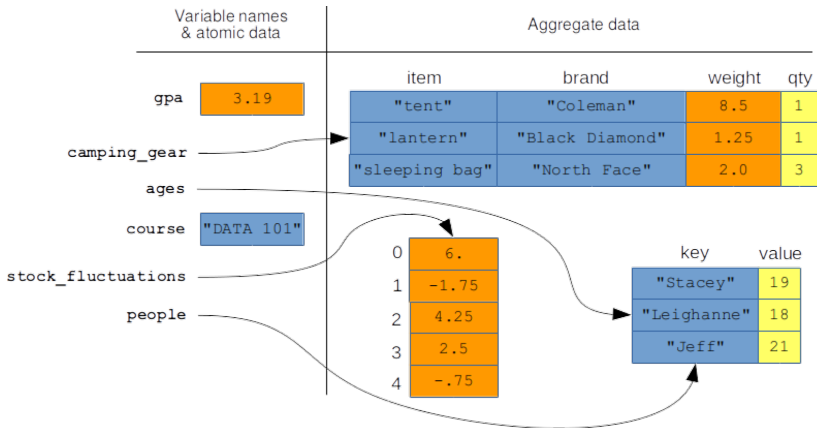


Figure 7.4: Where aggregate data variables – and their variable names – live in memory.

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, `list`s 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.

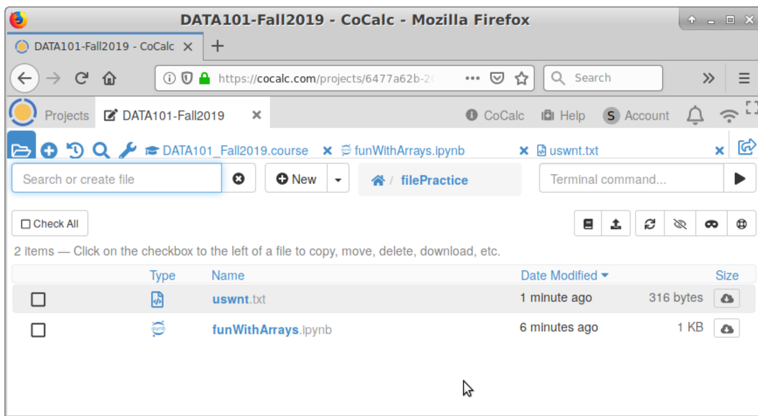


Figure 8.1: A directory (folder) on CoCalc, which contains two files: a plain-text file (called `uswnt.txt`) and a Jupyter Notebook which will read from it (`funWithArrays.ipynb`).

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.

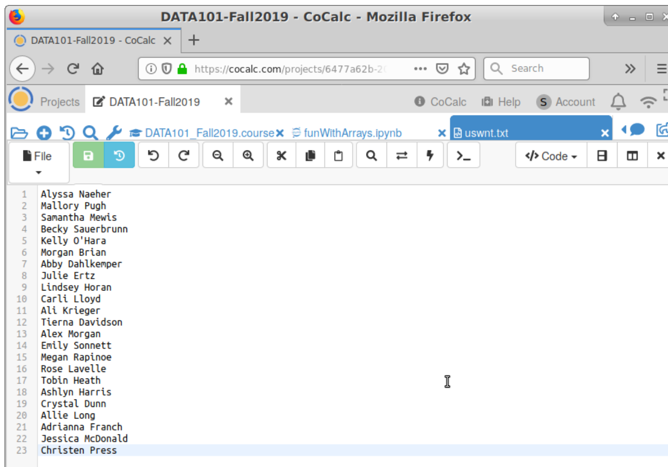


Figure 8.2: The contents of a plain-text file, as rendered by CoCalc.

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:
`ints`	`int`
`floats`	`float`
`strs`	`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 `ints` or `floats`. If you're reading `strs`, however, you need to set the **delimiter** to something *that doesn't appear in any of the `strs`.* 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.

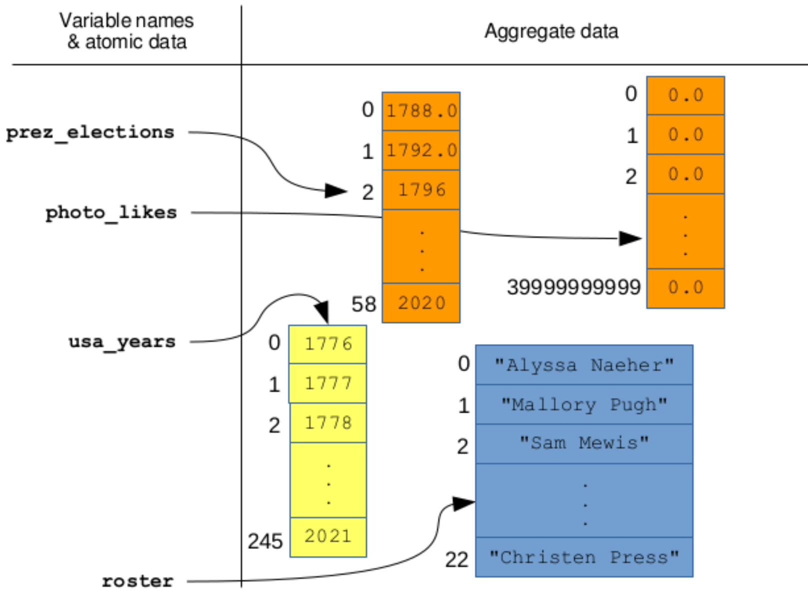


Figure 8.3: The memory picture of the four arrays we created in section 8.2.

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.

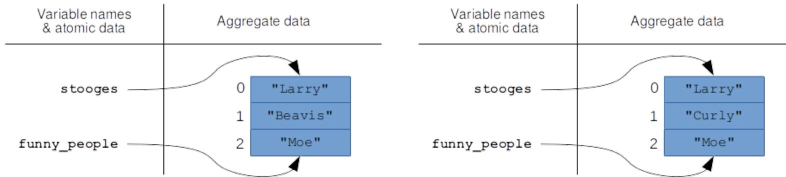


Figure 9.1: The code on p. 79 immediately before (left side) and after (right side) the line "`stooges[1] = 'Curly'`" is reached.

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'].
```

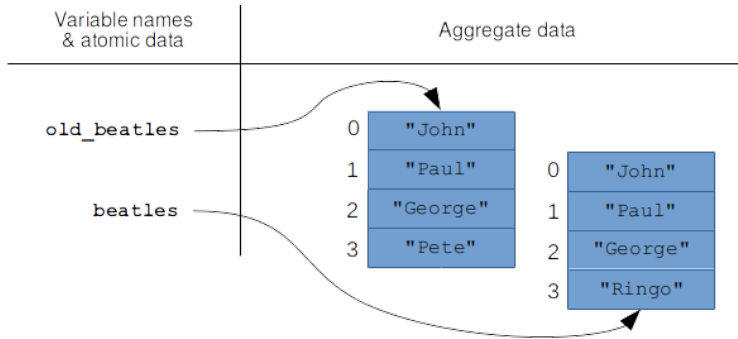


Figure 9.2: The memory picture after calling the `.copy()` method, instead of simply assigning to a new variable.

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.

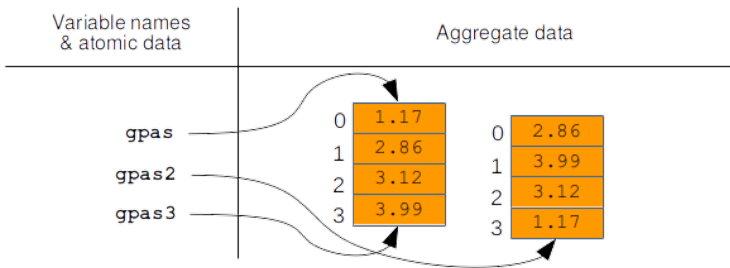


Figure 9.3: The state of affairs after `.sort()`ing the `gpas` array in place.

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.

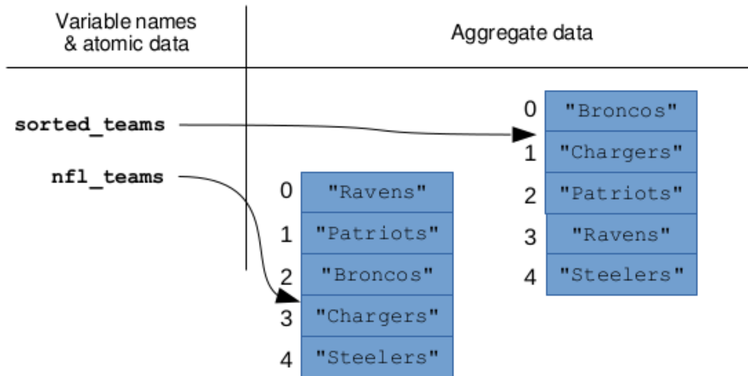


Figure 9.4: Calling the `np.sort()` function (as opposed to calling the `.sort()` method **on** the array) returns a sorted copy.

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 non-sense 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 "$C \rightarrow A, B$" 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 $A \rightarrow B$ nor $B \rightarrow A$, 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
$A \rightarrow B$	causation	Regular exercise does indeed normally lead to a lower resting heart rate.
$B \rightarrow A$	reverse causation	Smoking doesn't cause depression; depression causes smoking.
$C \rightarrow A, B$	external causation (confounding factor)	Ice cream sales don't cause shark attacks; high temperatures boost both ice cream sales and ocean swimming.
$A \rightarrow B$ & $C \rightarrow B$	multiple causation	A liberal arts education does improve critical thinking skills, but lots of other things do too.
$A, C \rightarrow B$	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.
$A \rightarrow C \rightarrow B$	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.
$A \not\rightarrow B$	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.

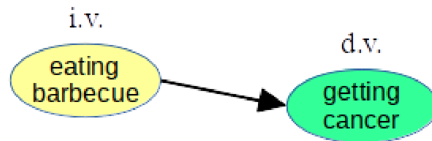


Figure 10.2: A hypothesis as to causality: eating barbecued foods increases one's risk for certain types of cancer.

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.**

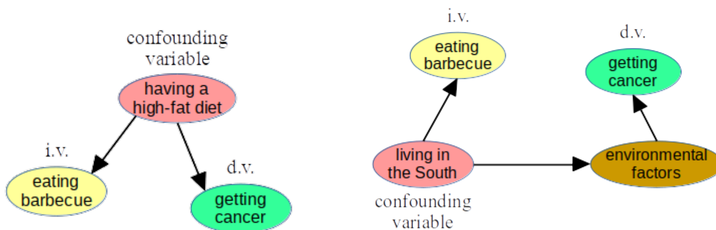


Figure 10.3: Other hypotheses as to causality, each resulting in the same associations in the data, yet involving confounding factors.

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.

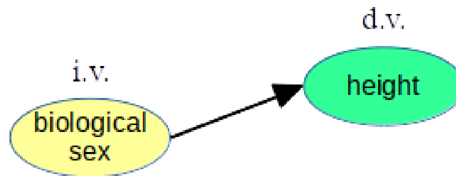


Figure 10.4: Stephen's hypothesis: a person's biological sex (male or female) plays a causal role in determining their height.

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

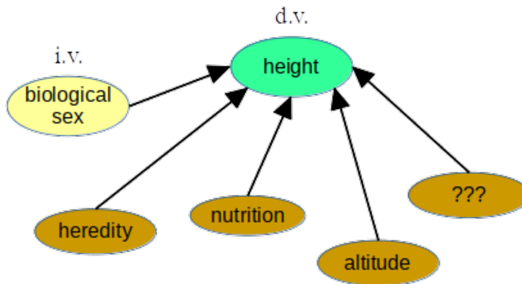


Figure 10.5: Oh geez – confounding variables galore? ***No!***

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.

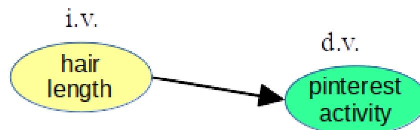


Figure 10.6: A theory about how hair length impacts the number of times a person logs on to pinterest each day.

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

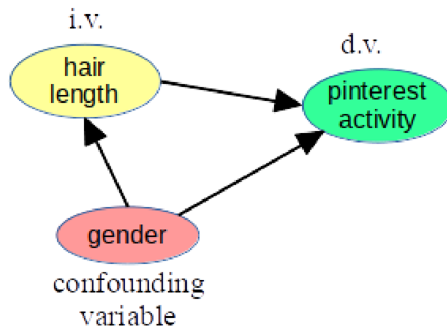


Figure 10.7: An alternative theory that holds that a person's gender influences both their long-haired-ness and their pinterest-ness.

(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 "$A \not\to B$," 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.

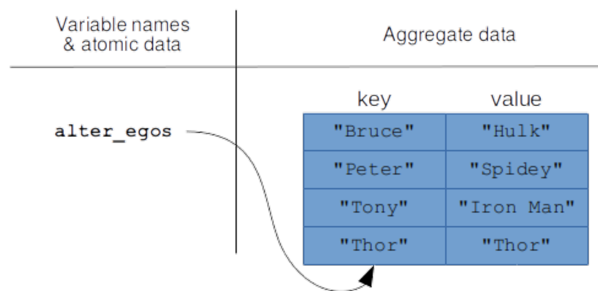


Figure 11.2: A Pandas `Series` in memory.

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

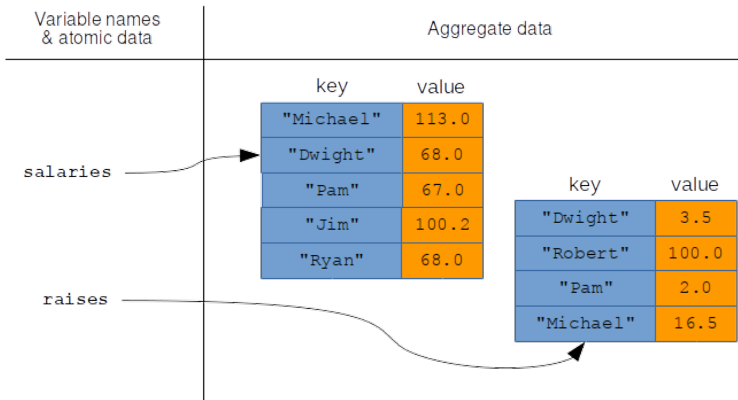


Figure 12.1: Two `Serieses` in memory

`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)
```

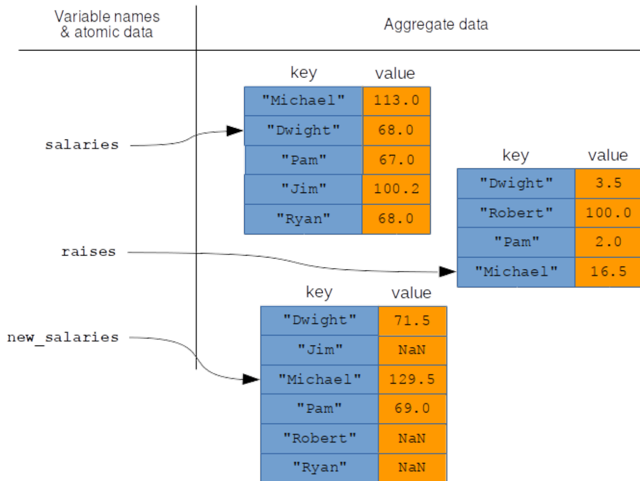


Figure 12.2: The result of +'ing two `Series`es that don't have all the same keys.

```
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)
```

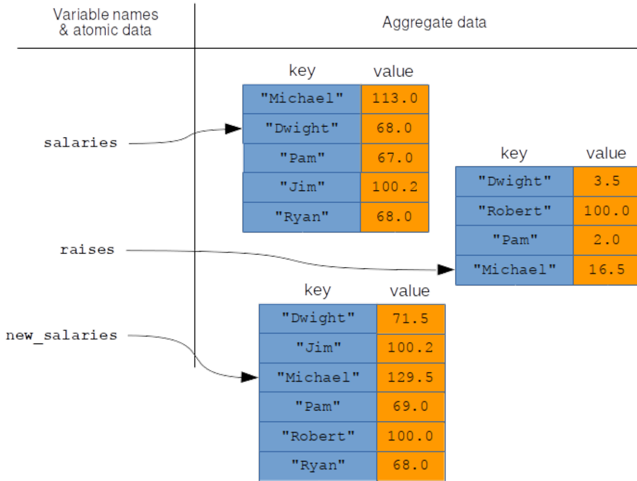


Figure 12.3: Using `add()` instead, and passing a `fill_value`.

```
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) Serieses 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:

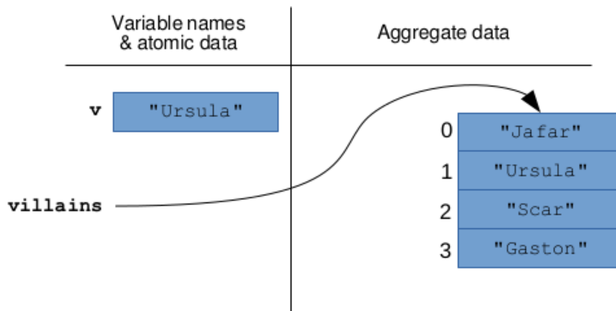


Figure 14.1: A snapshot of memory immediately after the *second* execution of line *3* of the `villains` program.

```
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.
```

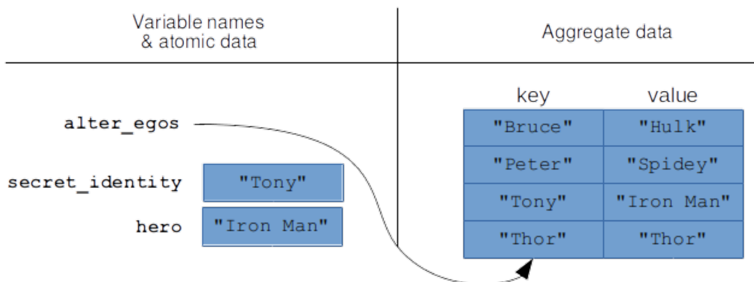


Figure 14.2: A snapshot of memory immediately after the *third* execution of the loop header in the `alter_egos` program.

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.