

The Datatrain R Bootcamp

Contents

1	Welcome!	7
What this is, and what it isn't	7	
Who this is for	7	
What you will learn	7	
Who are we?	8	
I	Core theory	9
2	What is data science?	11
What is the data life cycle?	15	
Data science 'in the wild'	16	
3	The reproducibility crisis	19
The crisis	19	
The 'reproducibility' movement	23	
Why R?	24	
4	Data ethics	25
A few principles	25	
Warm up scenarios	26	
Case studies	26	
More scenarios	30	
5	Visualizing data	33
The value of <i>data viz</i>	33	
Plot gallery	35	
Chart junk	56	
Final thoughts	57	
6	Setting up RStudio	61
7	Running R code	67
Running code in the <i>Console</i>	68	
Use R like a calculator	68	
Getting along with R	69	
8	Using RStudio & R scripts	73
R and RStudio: what's the difference?	73	
Two-minute tour of RStudio	73	
Scripts	74	
Your working directory	78	
Typical workflows	79	

9 Variables	81
Introducing variables	81
Types of data in R	82
10 Vectors	85
Declaring and using vectors	85
Math with two vectors	86
Functions for handling vectors	87
Subsetting vectors	89
11 Dataframes	93
Subsetting & exploring dataframes	94
Creating dataframes	98
Modifying dataframes	99
12 Packages	103
Packages you already have	103
Installing a new package	105
Loading an installed package	106
Calling functions from a package	106
13 Importing data	113
Reading in data	113
.csv files	115
Prepping your data for R	118
Managing files & folders	119
14 ggplot	121
What is ggplot2?	121
Scatter plot	123
Bar plot	129
15 Dataframe wrangling	139
The dplyr package	139
The %>% pipe	139
dplyr verbs	140
II Reproducible research	147
16 Markdown documentation	149
What is Markdown?	149
Raw Markdown example	150
Exercises	152
17 R Markdown	155
Do first, explain later	155
Add an R chunk	156
In-line R code	157
Tables	158
Bibliographies	158
18 Interactive maps	161
Mapping with leaflet	161
Geocoding	170
Final thoughts	172

CONTENTS	5
III Review exercises	175
19 A mystery dataset	179
20 A dplyr survey	181
21 Global health & ggplot	183

Chapter 1

Welcome!

What this is, and what it isn't

This is not a textbook or an encyclopedia. This is not a reference manual. It is not exhaustive or comprehensive. We won't mention statistical tests at all. There is almost no theory. In fact, this curriculum is biased, non-representative, and incomplete – *by design*.

So what is this? This guide is an *accelerator*, an *incubator* designed to guide you along the most direct path from your first line of code to becoming a capable data scientist. Our goal is to help you through the most dangerous period in your data science education: your very first steps. The first three weeks. That is when 99% percent of people give up on learning to code.

But it doesn't need to be this way. We made this book to reach more than just the 1%.

We have based our approach on three core premises:

Premise 1: We learn best by doing. Our goal is to get you *doing* data science. We will keep theory and detail to a minimum. We will give you the absolute basics, then offer you exercises and puzzles that motivate you to learn the rest. Then, once you've been *doing* data science for a bit, you soon begin *thinking* like a data scientist. By that, we mean tackling ambiguous problems with persistence, independence, and creative problem solving.

Premise 2: We learn best with purpose. Once you gain comfort with the basic skills, you will be able to start working on real data, for real projects, with real impact. You will start to *care about what you are coding*. And that is when the learning curve *skyrockets* – because you are motivated, and because you are learning *reactively*, instead of passively. Our goal is to get you to the point of take-off as quickly as possible.

Premise 3: A simple toolbox is all you need to build a house. Once you become comfortable with a few basic coding tools, you can build pretty much anything. The toolbox doesn't need to be that big; if you know how to use your tools well, and if you have enough building supplies (i.e., data), the possibilities are limitless.

Who this is for

The target audience for these tutorials is the *rookie*: the student who *wants* to work with data but has *zero* formal training in programming, computer science, or statistics.

What you will learn

- The **Core theory** unit establishes the conceptual foundations and motivations for this work: what data science is, why it matters, and ethical issues surrounding it: the good, the bad, and the ugly. Don't slog through this all at once. Sprinkle it in here and there. The most important thing, at first, is to start writing code.

The next several units comprise a *core* curriculum for tackling data science problems:

- The **Getting started** unit teaches you how to use R (in RStudio). Here you will add the first and most important tools to your toolbox: working with variables, vectors, dataframes, scripts, and file directories.
- The **Basic R workflow** unit teaches you how to bring in your own data and work with it in R. You will learn how to produce beautiful plots and how to reorganize, filter, and summarize your datasets.

For these first two units, we encourage you to take on these modules one at a time, in the exact order they are presented: we put a lot of thought into what we included in these modules (and what we did not).

- The **Review exercises** unit provides various puzzles that allow you to apply the basic R skills from the previous unit to fun questions and scenarios. In each of these exercises, questions are arranged in increasing order of difficulty, so that beginners will not feel stuck right out of the gate, nor will experienced coders become bored. This is where you really begin to cut your teeth on real-world data puzzles: figuring out how to use the R tools in your toolbag to tackle an ambiguous problem and deliver an excellent data product.
- The **Reproducible research** unit equips you with basic tools needed for truly reproducible data science: documenting your research and code with **Markdown**; weaving together your code and your reporting with **RMarkdown**; allowing users to explore the data themselves with an interactive **Shiny** dashboard or web app; and sharing your code and tracking versions of your code using **Git**.

Who are we?

www.datatrain.global.

Part I

Core theory

Chapter 2

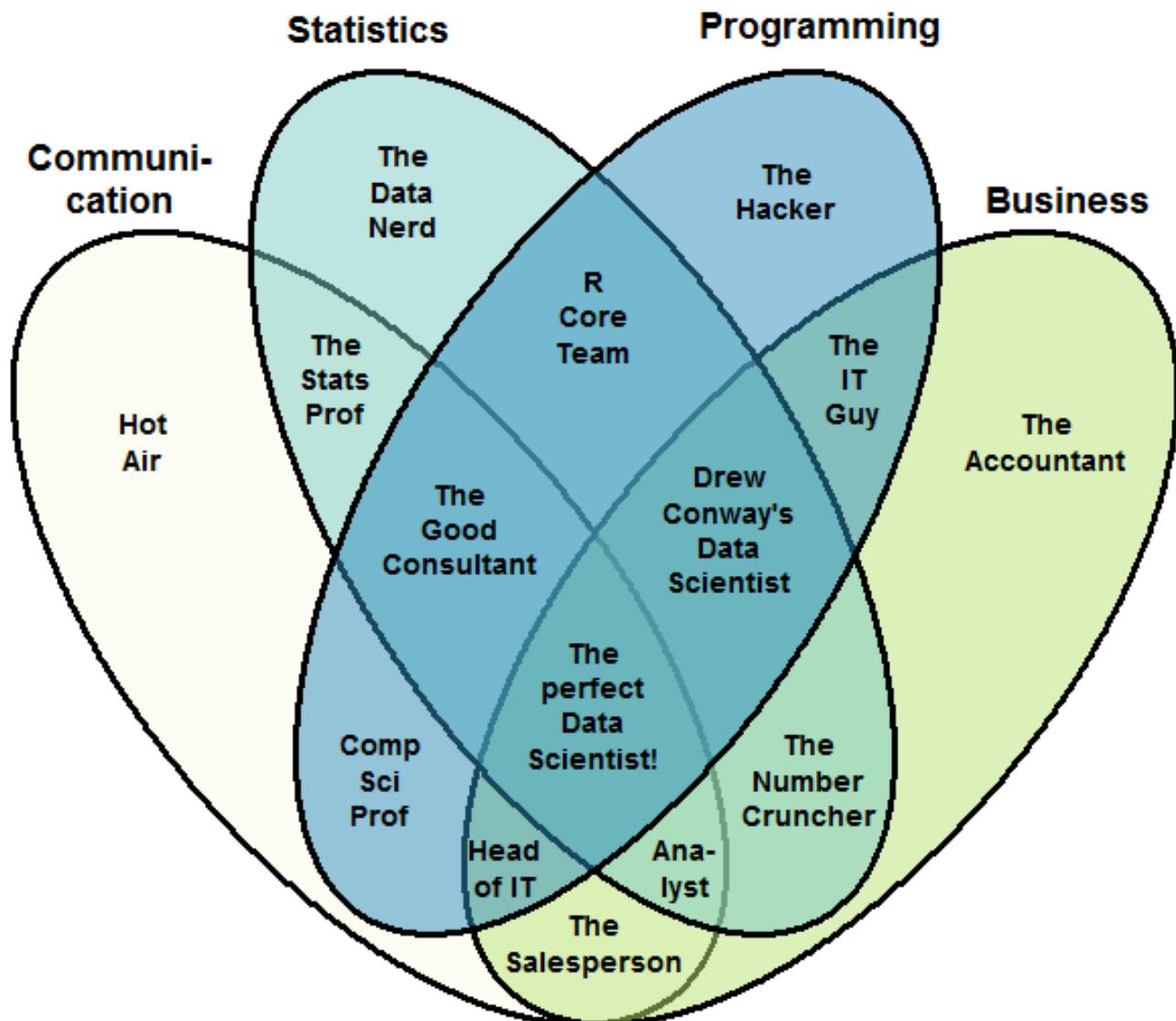
What is data science?

The definition of data science is a moving target. Thirty years ago (1990), ‘data science’ was an uncommon term that essentially just meant statistics. Twenty years ago (2000), the phrase mainly referred to querying SQL databases. Fifteen years ago (2005), it was “dashboards” and “predictive analytics”. Ten years ago (2010), it was ‘big data’ and ‘data mining’. Nowadays folks think of A.I. and machine learning.

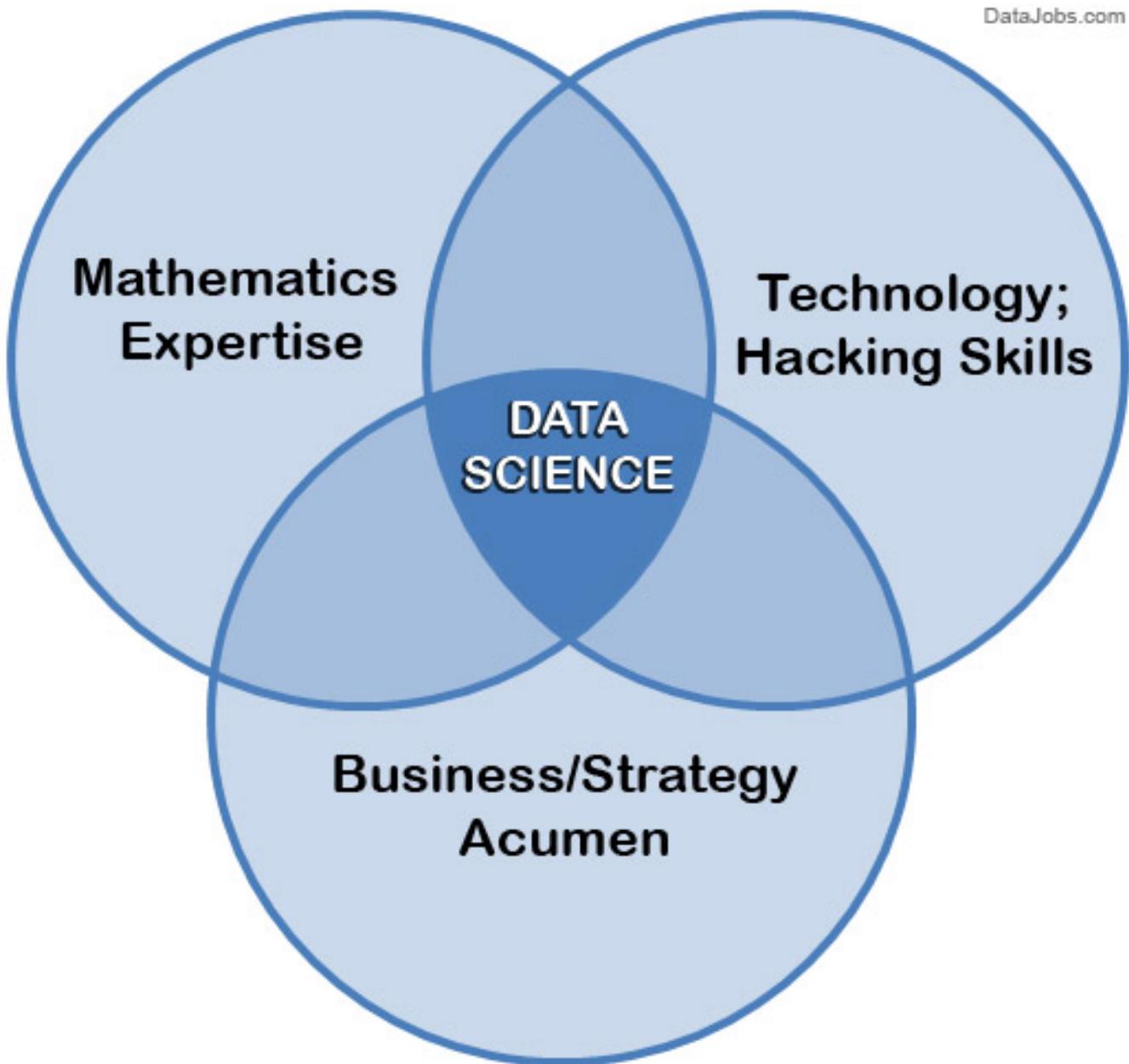
In 10 years? Who knows.

So what is it? There are many definitions out there. Search the internet for the answer and you will find complex diagrams, such as this one, suggesting that a data scientist is someone who has the right blend of programming skills, statistical knowledge, communication ability, and business acumen:

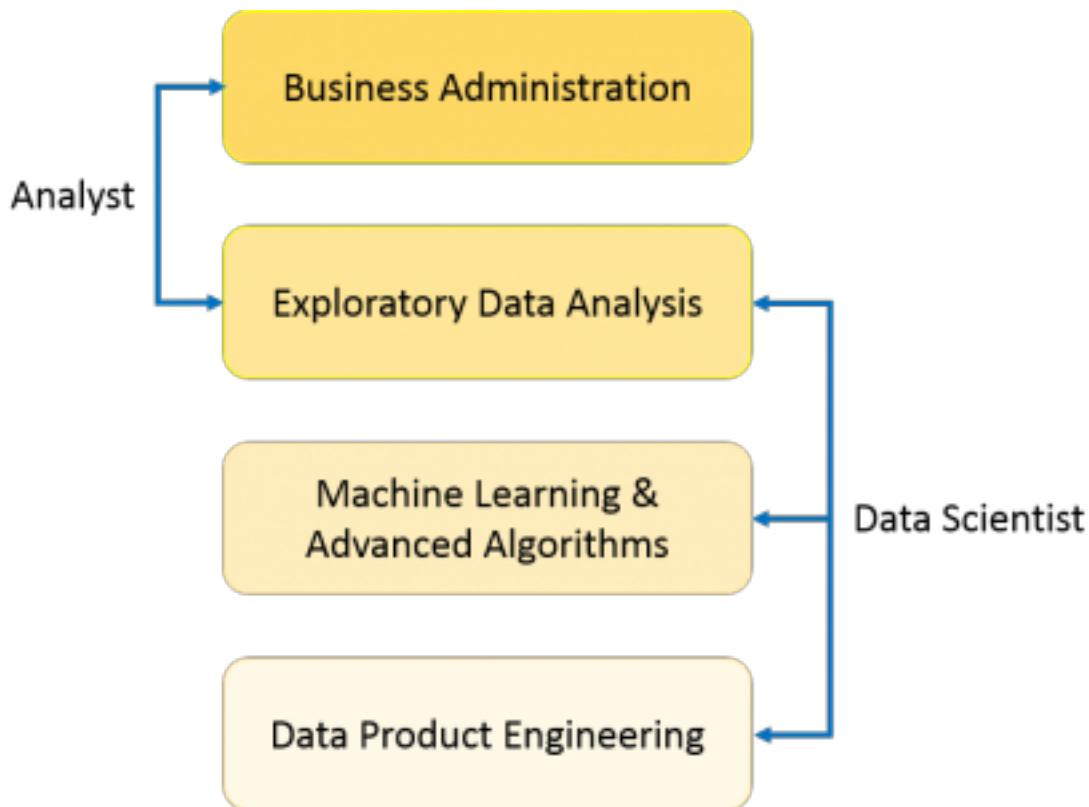
The Data Scientist Venn Diagram



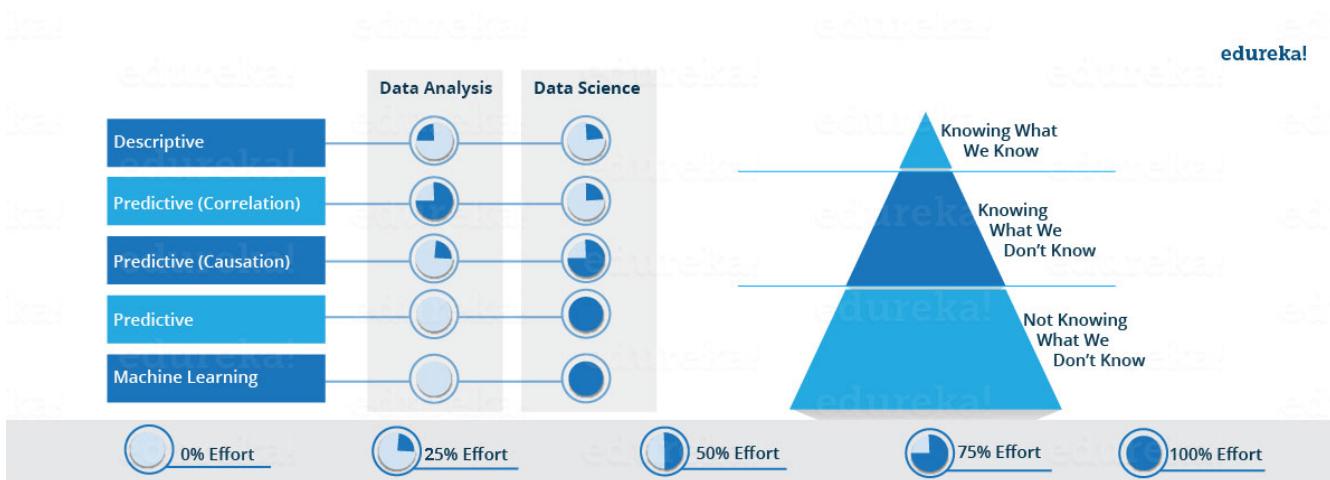
... or here is a more popular, simpler version of the same diagram:



Some argue that data science is simply an extension of statistics. You will also find attempts to distinguish between categories of data science, or to draw lines around what data science is and what it is not. A classic example is the bizarre delineation corporations draw between a data *scientist* and a data *analyst*:



... or ...



Our take? Those definitions are useful, interesting, and to some degree accurate. *But* data science is too new, and too fluid, to be fixed into some static definition. So, to keep our definition accurate, we'll keep it broad:

Data science is simply “doing science with data”. And for our purposes, the only difference between our definition and the definition of science itself is not in the word “data” – since nowadays all scientists are, to some degree, “data scientists” – but rather in the word “doing”. Data science is about *doing* stuff with data – about *making a difference with data*. And that’s what this course is going to be about. *DOING*.

But we’ll go one step outward. Data science is not just the combination of academic disciplines like stats and business strategy. Good data science also needs to involve (1) **domain knowledge** (i.e., familiarity with the problem being solved), (2) **a bias to real-world effects** rather than theoretical frameworks, and (3) **a desire to work in the real world**. To do so, data scientists generally need to be effective communicators and have an iterative mentality: they try something, evaluate its effects, try something else, and repeat.

Our definition is very broad, we know. We consider the “analyst” working in business intelligence to be a data scientist; and so too do we think that a data scientist could be an engineer who is processing large amounts of data to extract basic trends. Again, data scientists are those who *do science with data*. That’s a lot of people.

In our experience, the best data scientists aren’t simply the best programmers or best statisticians; the best ones are the people who consider themselves to be *something else first*. They are the journalists, artists, epidemiologists, psychologists, historians, environmentalists, sports analysts, and political commentators who *also* know how to work with data. In other words, the best data scientists are the ones already out there, on the ground, already embedded in the system they want to improve, positioned perfectly to get the right data, to ask the right questions, and to actually *do* something with insights from the data. Again, data science is about *DOING*.

To summarize, data science is about applying data to problems. It is impact-driven, transdisciplinary, and suited to well-rounded, multi-dimensional professionals.

What is the data life cycle?

There is a misperception about data science work that it is largely or even exclusively interpretative: that is, a data scientist looks at a big set of data and builds a fancy statistical model, then a light bulb goes off in her head, she has some insight, and then acts on that insight.

The reality is data science is much more than that. And most of data science is a combination of *(a)* getting data ready for analysis, *(b)* hypothesis testing, and *(c)* figuring out what to do with the results of *a* and *b*. That is, data science in practice is generally not some artesenal genius staring at a table of numbers until “insight” magically occurs. Rather, it is a lot of work, a lot of structured theories which can be confirmed or falsified, and a lot of *imagination* applied to the task of implementation.

In other words, data goes through a whole *lifecycle* of which analysis is just a small part.

What is the data lifecycle? Here’s how we conceptualize it:

- 0. Observation**
- 1. Problem identification & definition**
- 2. Question formation**
- 3. Hypothesis generation**
- 4. Data collection**
- 5. Data processing**

This step is usually the most intensive. Half the battle is wrangling raw data and making it ready for a visualization or a hypothesis test. Note that this step has *nothing to do with statistical tests* – data science is not the same as statistics!

6. Model building / hypothesis testing

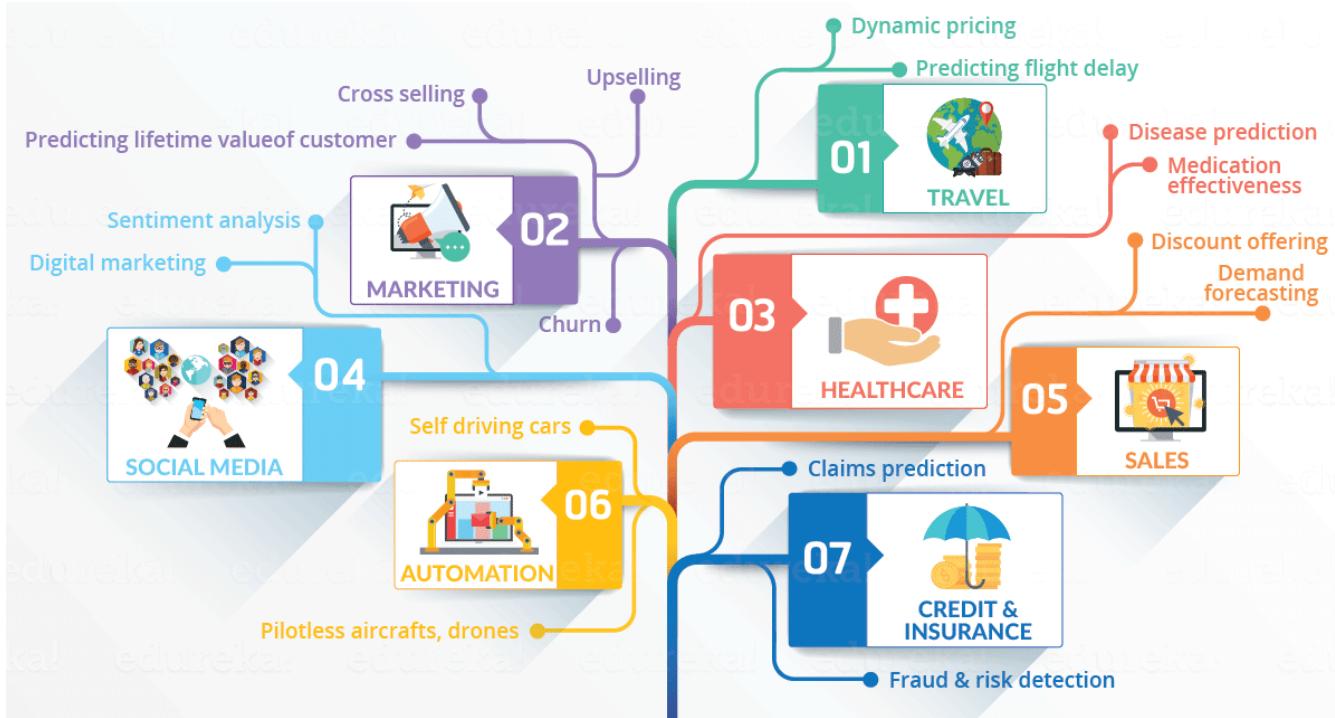
Note that this step is usually where the scientific method stops. In science, once you analyze your test, you interpret your results and loop back to the beginning of the data cycle. But in *applied data science*, there are a few more steps:

- 7. Operationalization:** This means determining how best to incorporate the data insights into operations.
- 8. Communication / dissemination**
- 9. Action:** This means actually implementing the change.
- 10. Observation:** back to the beginning of the cycle.

Again, the above should look a lot like the scientific method. The main differences are (a) “data processing”, which in reality takes up most of any data scientist’s time, (b) the bias towards action, and (c) the iterative / looped nature of the lifecycle.

Data science ‘in the wild’

Enough theory. What do data scientists actually do? Again, you can search for an answer online and find complex diagrams like this one:



But to capture every problem that data scientists are working on, this diagram would have to be even more convoluted and complex. Data scientists are working on a *ton* of problems.

The most stereotypical data science problems tend to involve advertising, social media, and corporate profiteering:

- Targeted advertising
- Social media feed optimization (getting you to scroll just a little further)
- Facial recognition (automated tagging at Facebook)
- Voice recognition ('Hey, Siri!', 'Alexa!')
- Making video games more fun / addictive
- Dynamic airline pricing
- Search autocomplete
- Autocorrect
- Virtual assistants

These are the kinds of problems that the best-paid data scientists in the world are working to solve. Right now there are thousands of programmers in Mountain View, Cupertino, and elsewhere in the Bay Area (and New York, and London, and Beijing) trying to solve the problem of you not spending enough time on social media.

Maybe you care about these problems, maybe you don't. Maybe they make you indignant or angry. Maybe you find it *problematic* that these things are even considered problems at all. As far as we're concerned, it is deeply unfortunate that our highest-paid data scientists are focusing on problems like these.

But take heart – there are plenty of other data scientists out there working on *actual* problems that are actually *important*:

- Identifying disease through imagery
- Automating identification of credit card fraud
- Filtering spam with malware or viruses.
- Preventive maintenance at nuclear facilities
- Improving chemotherapy dosage
- Increasing voter turnout
- Improve matchmaking systems (liver transplants, love, etc.)
- Measuring deforestation with satellite imagery.
- Efficient and equitable vaccine distribution
- Identifying tax evaders
- Predictive policing
- Storm surge forecasting
- Identifying and removing child pornography from the internet
- Surveilling emergency rooms to predict disease outbreaks
- Detecting fake news
- Increasing accountability and legitimacy of carbon markets
- Quantifying the likelihood of recidivism to prevent over-incarceration

The list goes on. The number of worthwhile problems waiting for data scientists is limitless, there are data scientists working on problems like these right now, and the demand for civic-minded data scientists is immense.

All of this matters for a lot of reasons. The first is that data science is not always a good thing; it can be weaponized by corporations and governments in spite of the public interest, and for that we need to be very careful about how we use it and how we teach it.

But the second reason this matters is that data science can be an *equally powerful force for social good*. We can use data science to make progress on the most urgent and injurious social and environmental problems of our time.

However – and this is the third reason all this matters – data science can only achieve social good *if* we recruit students to its ranks who are values-driven, civic-minded, and committed to using data science for good.

Fourth, and finally, this matters because the Facebook data scientists are using the exact same principles and basic tools as the non-profit data scientists. At their core, the foundational skillsets are the same.

And that's what this book is all about.

Chapter 3

The reproducibility crisis

The crisis

There is a crisis in the sciences: the reproducibility crisis. It is also known as the replication crisis. This refers to the fact that many scientific studies have been impossible to reproduce, calling into question the validity of those studies' findings.

This crisis began in the mid-2000's, when psychologists realized they could not reproduce most of their colleagues' results. They tried to repeat the experiments, following the methods step-by-step, but failed to get the same results. This was enormously unsettling for psychologists, and it cast major doubts upon the validity of psychological theory.

The realization that much of published research is not actually reproducible soon spread to medical research...

The screenshot shows the PLOS MEDICINE website. At the top, the journal logo 'PLOS MEDICINE' is displayed in a dark purple font. To the right of the logo is a search bar with the placeholder text 'advanced search'. Below the header, there are several status indicators: 'OPEN ACCESS' with a lock icon, 'ESSAY', '75,226 Save', '5,909 Citation', '2,720,175 View', and '7,825 Share'. The main title of the article is 'Why Most Published Research Findings Are False' by John P. A. Ioannidis. Below the title, it says 'Published: August 30, 2005 • <https://doi.org/10.1371/journal.pmed.0020124>'.

...then it sprung up in marketing...

The Desperate Need for Replications

John E. Hunter

Journal of Consumer Research, Volume 28, Issue 1, June 2001, Pages 149–158,
<https://doi.org/10.1086/321953>

Published: 01 June 2001

...and economics...

The screenshot shows the header of the Science magazine website with navigation links for COMMENTARY, JOURNALS, COVID-19, and Science. Below the header, there are links for News Home, All News, ScienceInsider, and News Features. The main content area features a dark background with white text. It includes a category link 'NEWS | BRAIN & BEHAVIOR'. The main title is 'About 40% of economics experiments fail replication survey' in large, bold, white font. A subtitle below it reads 'Compared with psychology, the replication rate "is rather good," researchers say'. At the bottom of the article preview, it says '3 MAR 2016 • BY JOHN BOHANNON'.

...and the sports sciences...

FiveThirtyEight



Politics Sports Science Podcasts Video

How Shoddy Statistics Found A Home In Sports Research

By [Christie Aschwanden](#) and [Mai Nguyen](#)
 Graphics by [Ella Koeze](#)
 Filed under [Meta-Science](#)
 Published May 16, 2018

...and the life sciences too:

For a complete history of the crisis, check out this article from Wikipedia.

Why is this happening? There are many reasons. Many studies, particularly those in psychology and the social sciences, involve small cohorts of participants. When sample sizes are low, results may not be representative of underlying truths.

On rare occasions it is intentional and fraudulent: scientists face pressure to publish interesting results, so much so that they might fabricate or filter their data to make their results significant.

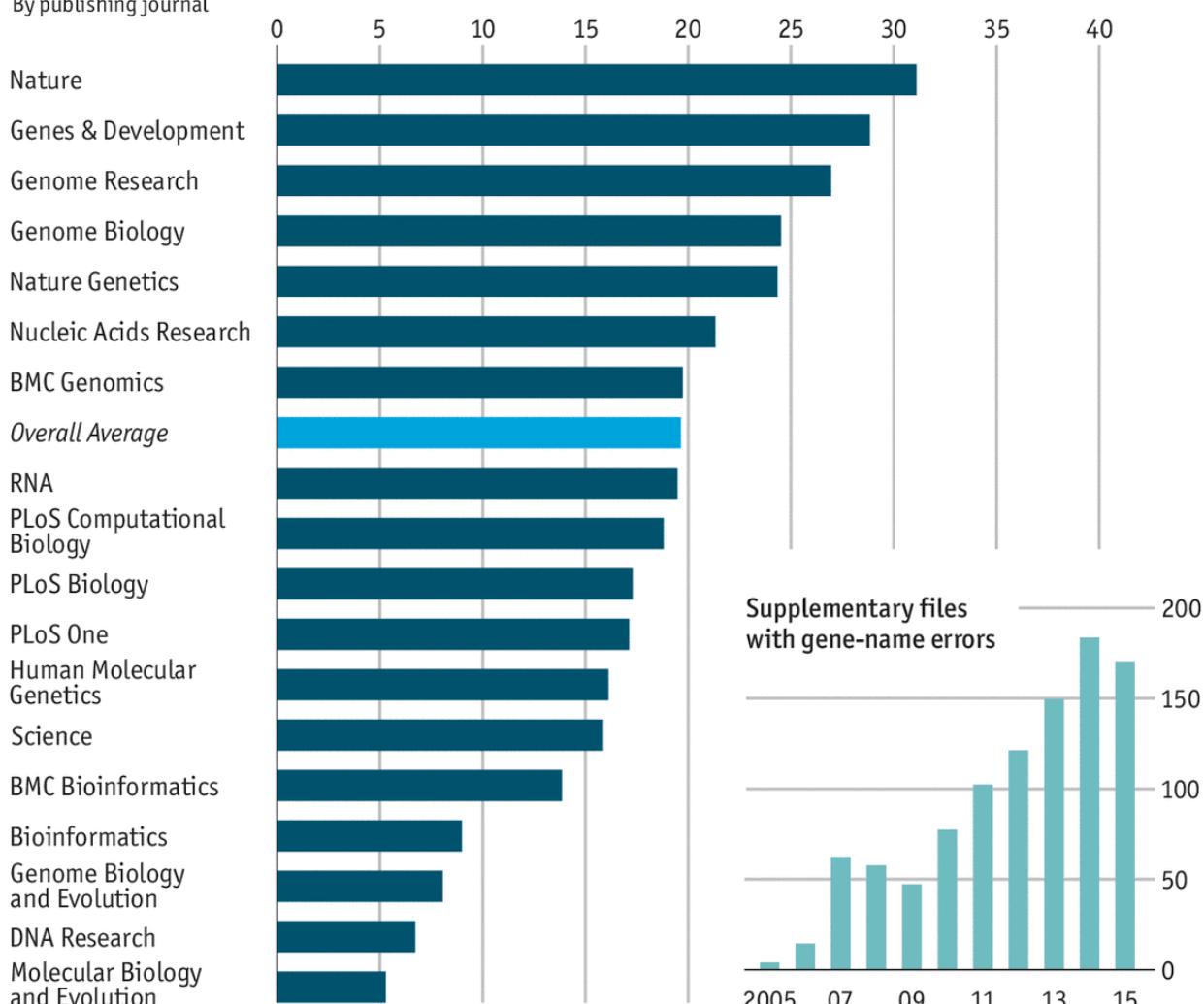
But the most common causes of reproducibility failure are, by far, (1) **poorly documented steps in data processing** – if you don't know how exactly the authors of a paper formatted their raw data to prepare them for analysis, you simply can't reproduce the analysis – and (2) simple, **honest mistakes**, such as typos in spreadsheets.

Consider this summary of the reproducibility crisis from *The Economist*. A scary percentage of genomics studies have simple spreadsheet errors:

#VALUE! error

Genomics papers with spreadsheet errors in supplementary files, 2005–15, %

By publishing journal



Source: "Gene name errors are now widespread in the scientific literature", Ziemann, Eren and El-Osta, 2016

Economist.com

This is a big deal: if a significant part of science is *wrong*, then what do we know? How can we be sure what we know is right? How can we build off of previous research? How can we distinguish valid science from the rest? If science can't be trusted, what value does it have for society? What kind of *damage* is it doing to society?

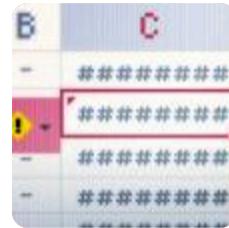
This crisis is ongoing, and it is impacting our handling of the COVID-19 pandemic. On October 5, 2020, the world learned that 16,000 COVID-19 cases disappeared from the UK's public health database due to a simple glitch in *Microsoft Excel*.

 Slate

An Outdated Version of Excel Led the U.K. to Undercount COVID-19 Cases

According to the BBC, the error was caused by the fact that Public Health England developers stored the test results in the file format known as ...

Oct 7, 2020

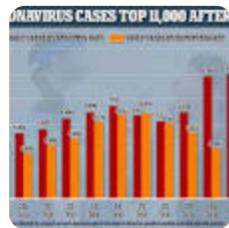


 Daily Mail

Furious blame game after 16,000 Covid cases are missed due to Excel glitch

The extraordinary meltdown was caused by an Excel spreadsheet ... rate of new Covid-19 infections has soared in dozens of areas of England ...

Oct 5, 2020



That event demonstrated that the reproducibility crisis is not just an academic concern. It can have serious and potentially deadly consequences for the public.

But there are **silly examples** of the replication crisis, too. Perhaps our favorite is this: in August 2021, when we Googled “reproducibility crisis”, one of the top search results is this video from *Science*, the world’s most prestigious scientific journal:

www.sciencemag.org › custom-publishing › webinars › re...

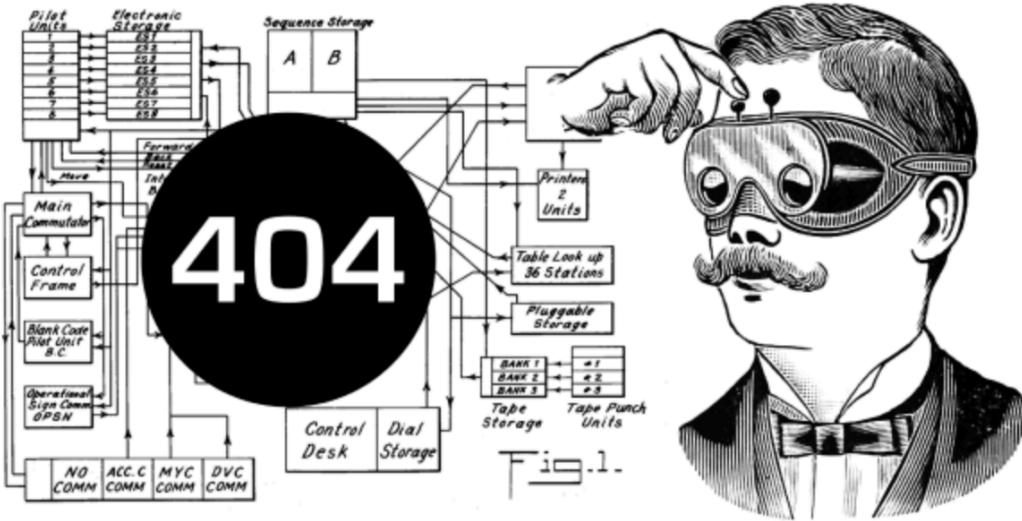
Reproducibility in crisis: Sample quality and the importance of ...



There is a reproducibility crisis occurring in the life sciences that impacts all researchers, influencing the ...

Feb 21, 2018

But when we click on this link, here’s what we see:



Hmmm ... this doesn't look like science.

It seems you're in search of a page that doesn't exist, or may have moved. You can use the Back button in your browser to return to the page that brought you here, or [search for your missing page](#).

If you'd like to visit a page that has plenty of science on it, please visit our homepage.

[BACK TO HOME](#)

The ‘reproducibility’ movement

Because of this crisis, there has emerged a much needed move to make all science “reproducible”. This means making sure that someone else can copy what you did, and get the same results. This is important for identifying scientific fraud, of course, but also for helping us to overcome human bias, mistakes, wishful thinking, etc. Reproducibility is not just a “nice-to-have”; in modern science (and data science), it’s a “must”.

Good data science must be reproducible. The idea is that work done by scientist A is “reproducible” by scientist B. In other words, if the findings of the research are of any generalizable value, then the results of two scientists working on the same problem should be identical (or very high in agreement). In practice, this means using data and code in a structured, well-documented, accessible, clear way, and ensuring that others can do the same.

Reproducible research also means using tools that others can easily use, and methods that others can easily copy. Programming languages like R and Python are ideal for this.

Reproducible research matters for lots of reasons:

1. Because making your work reproducible means that *you* will have less problems returning to that work at a later time.
2. Because making your work reproducible means that *others* can collaborate with you, help you, error-check you, and build on your work.
3. Because making your work reproducible means you are fighting the plague of irreproducible results which have characterized the replication crisis.

Making your work reproducible is going to be a bit more work, but it's not optional. And there are tools and best practices in place to make it as painless as possible. Basically, reproducible research involves the following:

- Using code to format and manage your data instead of spreadsheet software such as *Excel* or *GoogleSheets*, since those products will not keep a step-by-step record of each thing that you do. When you code, each command line is both an action you take to process your data *and* a record of what that action is.
- Coding with free, open-source tools, such as *R*.
- For any specific niche task in your analysis, such as processing a batch of images, using other open source tools (e.g., *ImageJ*) that can be used free-of-charge by anyone with an internet connection anywhere.
- Documenting *everything* you do with the data, by commenting your code thoroughly and by creating “Wiki” pages for your projects.
- Making your code open source and freely available online.
- Making your data open source (while protecting privacy and confidentiality of participants).
- Providing tools, such as *Shiny* in *R*, that allow others to explore your data themselves, rather than trusting your own narratives about the data.
- Using tools for generating reports, such as *Rmarkdown*, that remove the ‘middle-man’ and avoid potential typo’s and fabrications.
- Collaborating openly with others.

You will be learning how to do all of these things in this course. We are going to focus on *reproducible research*, *literate programming*, *documentation*, and other components of data science (and research in general) which ensure that (a) our methods and findings can be easily sanity-tested by others, and (b) we set ourselves and our projects' up for future collaborations, hand-offs, and expansion.

Why R?

This course is largely about learning to *do*, and will largely use *R*. *R* is not the only tool in the data scientists' toolbox (there are many), but it's a good one, is extremely popular, there is almost nothing you cannot do with it, it can be applied to many fields, and – most importantly – it is a free, open-source tool with an active open-source coding community. The millions of *R* users worldwide emulate the spirit of reproducible research we are trying to advocate for here.

A final thought

A research article about *results* is advertising, not scholarship.

Scholarship is an article with transparent, reproducible methods.

Chapter 4

Data ethics

A few principles

This orientation to the principles of data ethics is not going to be adequate or sufficient. We just need to provide enough context for you (1) to appreciate the limitless complexity and uncertainty of many ethical issues in data science, and (2) to start exploring the complex ethical scenarios below on your own or in dialogue with others.

In most frameworks for data ethics, three foundational principles are used to help us think about whether certain research actions are ethical. Those three principles are:

1. **Respect** for persons and their autonomy: participation must be based upon informed consent, and privacy must be honored at all times. Immature or incapacitated persons must be protected as they mature or heal.
2. **Beneficence**: in our work with data, potential risks are minimized while potential benefits are maximized.
3. **Justice**: Benefits and risks are distributed equally across groups of people. A classic way of asking whether something is ‘just’ is asking using John Rawls’ concept of the ‘**veil of ignorance**’: pretend you have no information at all about your circumstances or your place in the social order: You don’t know your place of birth, year of birth, sex, skin color, language, religion, immigration status, health conditions, or anything else. In other words, you have no information whatsoever that might introduce bias into the way you think about the world. Free of circumstantial bias, what arrangements would you choose to put in place to maximize fairness and fortune for all, and to minimize the chances that you would get screwed by the system?

These principles can guide us as we navigate ethically ambiguous scenarios. When we ask whether something is ethical, we are asking whether all of these principles are upheld. We could also be asking whether the violation of one of these principles might be justified by upholding another in an impactful way.

The question, ‘Is something ethical?’ is usually not easy to answer, particularly when it comes to the use of data in tackling social problems. It is important to note that reasonable people regularly disagree on these ideas; that is why we have committees and drawn-out processes for obtaining permission to use data in research and commerce.

So why do these principles matter? Because without them, we would not be able to have conversations about the ethics of difficult situations. We need articulated principles that we can point to and debate together. Principles like these allow you to have an account for why you feel the way you do about a certain issue. Without that account, we can’t learn from each other’s perspectives.

Note also that these principles were designed with **individual human subjects** in mind. It is an open question of active debate how exactly these principles can be applied to **communities of individuals** all at once – what exactly does it mean for a group to consent to something? Does every single individual need to consent? The majority? – or how they translate to our treatment of **non-human communities**: animals, plants, and places.

Let’s stop there and explore some concrete scenarios. For a better orientation to ethical precepts underlying issues of data ethics, this chapter by Shannon Vallor is the best open-source resource that we have been able to find. Many of the case studies and scenarios presented below are adapted from that chapter.

Warm up scenarios

Practice applying the above principles to these scenario questions. For each scenario, describe your **opinion vector** (the *direction* of your opinion – yes or no – and the *strength* of your opinion).

Location tracking

Is it ethical for Google to track and store your location information in order to monitor traffic and operating hours of local businesses? Such traffic information is known to help direct emergency service vehicles along the safest and fastest route.

Targeted advertising

Is it ethical for internet search engines to tailor advertisements according to your search history?

Dynamic pricing

Is it ethical for airfare search engines to adjust ticket prices according to your recent search history?

Social media scrolling

Is it ethical for Instagram to count how many milliseconds you spend on each post, then use that info to develop a strategy for getting you to spend more time on its app?

Controversial content

You are a data scientist at Facebook. Based on your analyses of user data, you have discovered that when you show readers sensational or hyperbolic content, such as someone ranting that a new vaccine is an attempt at government-subsidized mind control, the readers stay on Facebook longer and scroll through more content. Since that translates to profits, is it OK for your team to increase the amount of sensational content in users' feeds?

Case studies

Use these case studies to reflect upon and discuss the ambiguity, complexity, and dangers of data ethics issues.

The Facebook ‘Social Contagion’ Study

In 2014, data scientists from Facebook published an article in a prestigious academic journal. In this article, they demonstrated that the emotions and moods of users could be manipulated by toggling the amount of positive or negative content in their feeds. They found that these emotional effects would then be passed to other users in the social network; in other words, emotions and moods could be seeded and were ‘contagious’. To carry out this research, they manipulated the Facebook feeds of 689,000 users.

The image shows the Proceedings of the National Academy of Sciences of the United States of America (PNAS) website. The header features the PNAS logo in large white letters on a dark blue background. To the right of the logo is the journal title "Proceedings of the National Academy of Sciences of the United States of America". Below the logo is a search bar with the placeholder "Keyword, Author,". A horizontal navigation bar below the header includes links for "Home", "Articles" (which is highlighted in white), "Front Matter", "News", "Podcasts", and "Authors".

RESEARCH ARTICLE



Experimental evidence of massive-scale emotional contagion through social networks

Adam D. I. Kramer, Jamie E. Guillory, and Jeffrey T. Hancock

+ See all authors and affiliations

PNAS June 17, 2014 111 (24) 8788-8790; first published June 2, 2014; <https://doi.org/10.1073/pnas.1320040111>

Edited by Susan T. Fiske, Princeton University, Princeton, NJ, and approved March 25, 2014 (received for review October 23, 2013)

This article has Corrections. Please see:

[Editorial Expression of Concern: Experimental evidence of massivescale emotional contagion through social networks](#) - July 03, 2014

[Correction for Kramer et al., Experimental evidence of massive-scale emotional contagion through social networks](#) - July 03, 2014

Article

Figures & SI

Info & Metrics

PDF

Significance

We show, via a massive ($N = 689,003$) experiment on Facebook, that emotional states can be transferred to others via emotional contagion, leading people to experience the same emotions without their awareness. We provide experimental evidence that emotional contagion occurs without direct interaction between people (exposure to a friend expressing an emotion is sufficient), and in the complete absence of nonverbal cues.

Facebook did not receive specific consent for this study from its users. Instead, the company argued that the purpose of the study was consistent with the user agreement already in place: to give Facebook knowledge it needs to provide users with a positive experience on the platform.

Discuss:

1. In what ways, specifically, did Facebook violate basic principles of data ethics with this study? Enumerate each violation individually.
2. Can a convincing argument be made that justifies this study? What are the strongest arguments in its favor?
3. What are some things that Facebook could have done differently to handle this situation more ethically?
4. Who exactly should be held morally accountable for any harms caused by this study? The data scientists employed to analyze and publish the data? Mark Zuckerberg? All Facebook employees? Facebook stock holders? Are Facebook users accountable at all?

Machine bias: Beauty contests & recidivism

Machine learning (ML) algorithms are developed by ‘training’ models on known datasets. The models are then used to predict values in other datasets. For example, if you label cars in a batch of photos then use them to train a ML model on that labeled dataset, you can then use that model to identify cars in thousands of other photos.

Sounds neat, but this means that ML models are only as good as the data they are trained upon, and often those training datasets are created by human labelers who carry unknown or unspoken biases. A classic example is the Beauty.AI beauty contest that occurred in 2016. A ML algorithm was trained on a large set of human-labeled photos of women, then women around the world were invited to submit selfies in a global beauty contest. A key advertising hook for the contest was that the ‘robot jury’ – the ML algorithm – would be fully impartial and fair. But the results revealed that the ML model was racist: 75% of the 6,000 contestants were white, but 98% of the 44 winners were white. How did this happen? The people who labeled the training set of photos carried implicit bias.

Artificial intelligence (AI)

This article is more than **4 years old**

A beauty contest was judged by AI and the robots didn't like dark skin

The first international beauty contest decided by an algorithm has sparked controversy after the results revealed one glaring factor linking the winners

Sam Levin in San Francisco

Thu 8 Sep 2016 18.42 EDT

[f](#) [t](#) [e](#) 685



▲ One expert says the results offer 'the perfect illustration of the problem' with machine bias. Photograph: Fabrizio Bensch/Reuters

Debacles like this can have much more serious consequences. Court systems use ML models to estimate the risks that a convicted criminal will commit more crimes once they are released from prison. But retrospective studies have shown that these models consistently and incorrectly label black prisoners as more dangerous and more likely to return to prison at a later date. Most of the risk assessments being used today have not received adequate validation, even though they are spitting out predictions that can destroy lives, families, and communities.



Bernard Parker, left, was rated high risk; Dylan Fugett was rated low risk. (Josh Ritchie for ProPublica)

Machine Bias

There's software used across the country to predict future criminals.
And it's biased against blacks.

by Julia Angwin, Jeff Larson, Surya Mattu and Lauren Kirchner, ProPublica
May 23, 2016

Discuss:

1. How might bias have entered the training datasets for these ML models, if the people labeling the data did not deliberately intend to exhibit prejudice against African Americans?
2. The ML models used to predict recidivism are imperfect and inherently prejudiced, but it is not clear whether it would be better to leave decisions of sentencing, bail amounts, and prisoner support services to individual humans in the court and penal systems. Would it be better to stop all use of ML evaluations, or should they be kept in place until better models are developed?
3. Returning to the beauty contest debacle. The attraction of a 'robot jury' compelled people to seek out a single, simplistic definition of beauty, and to place all contestants on the same spectrum of beauty scores. Other than the racial bias baked into their model, what other problems is there with this endeavor? Articulate and enumerate as many as you can. What do those problems tell you about other ethical and humanistic dangers inherent to data science?

Web-scraping OKCupid

In 2016, Danish researchers used new web scraping and text mining software to inventory the user profiles of 60,000 users on the online dating site OkCupid. Their goal was to use this dataset to test for correlations between 'cognitive ability' and sexual orientation, religious affinity, and other personality traits. They chose these user profiles because they were publicly available online to anyone who wanted to sign up for a free account with OKCupid.

When they published their paper, they included a spreadsheet of the 60,000 user profiles in the supplementary material for their article. They had removed the first and last names of the users, but kept everything else, including username, location, sexual orientation, religious orientation, sexual habits, relationship fidelity, drug use history, political views, and more.

The backlash was immediate. When asked why they did not attempt to deidentify or anonymize the data any further, the researchers responded that the data were already public. In the paper itself, the authors wrote: "Some may object to the ethics of gathering and releasing this data ... However, all the data found in the dataset are or were already publicly available, so releasing this dataset merely represents it in a more useful form."

The screenshot shows the FORTUNE website homepage. At the top, there's a navigation bar with links for 'RANKINGS', 'MAGAZINE', 'NEWSLETTERS', 'PODCASTS', 'COVID-19', and 'MORE'. To the right are 'SEARCH', 'SIGN IN', and a red 'Subscribe Now' button. Below the navigation, there are several news cards. One card on the left is titled 'Most Popular' with the sub-headline 'The real reason everyone is quitting their jobs right now'. Another card features two people and the headline 'CONTENT FROM IBM Embracing hybrid multicloud'. A third card shows a person in a lab coat with the headline 'Israel was a vaccination poster child. Now its COVID surge shows the world what's coming next'. A fourth card shows a person in a suit with the headline 'Bitcoin tumbles 11% after El Salvador's adoption of the crypto as legal tender'.

TECH • OKCUPID

Researchers Caused an Uproar By Publishing Data From 70,000 OkCupid Users

BY ROBERT HACKETT
May 18, 2016 2:41 PM CDT

Discuss:

1. What do you make of the authors' argument, that the data were already public, posted freely by the users themselves, so how can this be an issue of privacy or consent?
2. If you disagree with the authors' argument, explain how the users might reasonably object to the authors' actions.
3. What kind of risks did the authors impose upon the users of OKCupid? Are any of those risks new, or were they all present when the users decided to make a profile that could have been accessed by any other user?
4. Does it make an ethical difference that the authors accessed publicly available data in a novel way (web-scraping software) and to a much greater extent (harvesting 60,000 profiles at once) than individual users are typically able to do?
5. Does the software developer of the web-scraping tool bear any responsibility for this scandal? Does he have any ethical obligations regarding how his tool is used, and by whom?

More scenarios

Airport screening

A data scientist has come up with a model that prioritizes security screening at airports according to various passenger characteristics. If using 'place of origin' as a predictor in this model improves the model's predictive performance at identifying passengers who are security threats, is it ethical to include that variable in the model and screen certain passengers disproportionately?

Supporting struggling college students

Your college has developed a model that predicts dropouts. It identifies students at high-risk of dropping out and alerts offices that can direct additional support and resources to these students. Your college has found that this model performs better when it includes the student's place of origin, sex, and race as predictors. Is it ethical to implement this model and divert resources accordingly?

Robot cashiers

Self-checkout stations in grocery stores are convenient, but they take jobs away from workers who may not have many other employable skill-sets. When you are checking out, is it more ethical to use the checkout aisles with human cashiers, even if the line is longer and going more slowly?

Electric cars

Cars running on fossil fuels are bad for the environment, but at least they can be serviced by car mechanics who don't necessarily need a college degree.

Electric cars reduce carbon emissions, but they are replacing car mechanics with computer scientists and software engineers, all of which require extensive undergraduate and post-graduate education. Are electric cars a net social good?

Smartphone app for monitoring cough

A tech start-up has developed an app that can track the prevalence of cough in a network of smartphones. Cough is an important indicator of disease, and cough also helps to spread certain diseases more quickly, such as TB and COVID-19. This app has great potential to help public health officers in the fight against some of the deadliest respiratory diseases. The app works thanks to sophisticated machine learning algorithm for detecting coughs within continuous recordings. That algorithm is currently private and proprietary. Do you agree that this cough monitoring app is a good idea, and that public health officials should promote its use?

Automated suicide prevention system

A large internet search engine has developed a model that can predict whether someone is likely to inflict self-harm or attempt suicide based upon their recent search history. This model is 75% accurate. This company would like to set up an automatic emergency alert system, in which local social service providers are notified about at-risk users in their area. They want to automatically enroll users in this service. Is this an ethical feature to add to their product?

Malaria medicine distribution

Your company is trying to distribute a new malaria medicine in a remote region of Africa without primary care clinics, where tens of thousands people die from malaria each year. This medicine is highly effective, but it is also known to cause birth complications. You need to ensure that it is not administered to pregnant women. Your team's plan is to go door to door and distribute the medicine to women who say they aren't pregnant.

But in this region, cultural attitudes to pregnancy, and the notion of sharing your pregnancy status with a stranger, are very sensitive. Daughters and wives may not feel safe to answer such questions truthfully.

Your team has to choose between (1) taking women's responses at their word, (2) avoiding the pregnancy issue by only distributing it to men, (3) not distributing the medicine at all, (4) some as-yet-unknown solution. What do you do?

User accountability

Let's say that you have disagreed with one or more of the claims about social media in the 'Warm-Up Scenarios' section above. Is it ethical for you to continue to use Google, Instagram, or Facebook?

Chapter 5

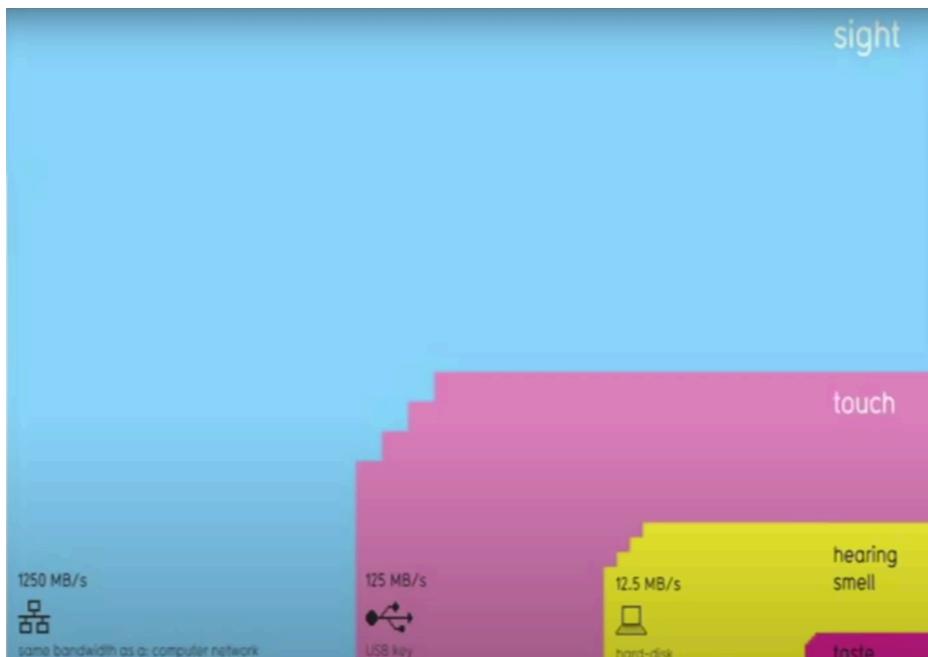
Visualizing data

This course will focus heavily on visualizing data in plots, maps, and dashboards. If there is anything you take from this course, it will be this: you will be able to take data and make some pretty pictures. *And that's not trivial!*

Why? Because **humans are wired to process information through *pictures*.** We can translate images into meaning with amazing speed.

The value of *data viz*

This screenshot, from David McCandless's TED talk about the beauty of data visualization, depicts how quickly each of our senses can process information.



This plot's punchline: When we process data with our eyes – with *pictures* – we can take in a lot of information all at once.

Let's try this out with an example. Below this paragraph is another paragraph describing a painting. Try this: scroll down quickly, look at this paragraph for just *one second*, then keep scrolling until it is out of view.

Go!

One-second paragraph:

Fog rises from the evergreen forest of a distance mountain range. A whitewater creek cascades down a streambed with large, rounded boulders, arriving at a broad flatwater pool where ducks are milling. There is one group of four and another group of two. On the shore near the ducks, a wooden dinghy is tied up to a small dock with eight pilings. The dock lead to a path through more round peddles and tall grass, past a chair and a fire ring, and continues uphill to a small cabin. The evening sun and sparse fairweather clouds are reflected in the cabin's large, multi-paned windows under the small front porch. A cobblestone chimney on the side of the cabin has a whisp of smoke rising from it. The steep roof implies that this cabin is designed to withstand heavy snow. Tall evergreens tower over the diminutive cabin; the cabin seems to be placed up against a forested hillside. There are only a few deciduous trees in view, and their leaf colors – combined with the lack of snow in the distant mountains – imply that the time of year is early fall.

End of paragraph.

OK. Now try to answer these simple questions:

- What was this a painting of, in general? Can you describe the scene?
- What details do you recall?

OK. Try this next: the actual painting is at the very end of this chapter. Scroll down to it quickly, look at this painting for just *one second*, then scroll back up to this spot in the module.

Ready? Go!

< *Return to this line!* >

Now try to answer those same questions above. What was this a painting of? Did you catch any more details? Was there anything in the water? Was there smoke coming out of the chimney? What time of day was it?

Which type of visual information was easier for you to process quickly? Text, or a picture?

Think about the profound *differences* in these two forms of visual communication:

When we read text, we are working outward, from individual details to the big picture: we process each individual word, understand their individual meanings, understand their meanings in the context of each individual sentence, then use all of the information to step back and imagine the scene based on the details.

In contrast, when we look at a picture, we are working inward, from the big picture down to the details. We understand the scene first, then we start exploring the finer points. And, since each finer point is interpreted from within the context of the bigger picture, we can make sense of the details much more efficiently.

Pictures communicate data. **This is why data science and data visualization nearly always go hand in hand.**

Data scientists use visualizations both to communicate their insights externally, e.g., to the public in a *Twitter* post, but also internally: when they are working with the data themselves. A data scientist's workflow is peppered with data visualization, because – again – visualizing your data is the most effective way of making sense of it: Download the data, then visualize it. Do something to the data, then visualize what you've done. Repeat, then visualize, then repeat again.

The point here is that great data visualizations are not simply pretty. Much more importantly, they are *effective* too. They are the best means you have of conveying insights from your data to someone else.

A final thought: Keep in mind that plots can be effective and misleading at the same time. There is a politics to plots and maps; they can have agendas, and they can manipulate viewers into interpreting the data in certain ways. So, it is incomplete for us to say simply that a good plot is an effective plot. Here's a better definition: *a good plot is one that is both effective and fair.*

So, when you are viewing other people's plots and making plots of your own, keep these five rules in mind:

1. A bad plot is an ineffective one, even if it is beautiful.

2. A good plot is an effective plot.
3. A great plot is one that is both effective *and* beautiful.
4. If you ever have to make a trade-off between effectiveness and beauty, sacrifice beauty.
5. Any plot that misleads or manipulates the viewer is bad, no matter how effective or beautiful it is.

Before you begin evaluating the plots in the gallery below, enjoy this excellent talk by the Egyptian data scientist, **David McCandless**, about the **beauty of data visualization** ([link here](#)).

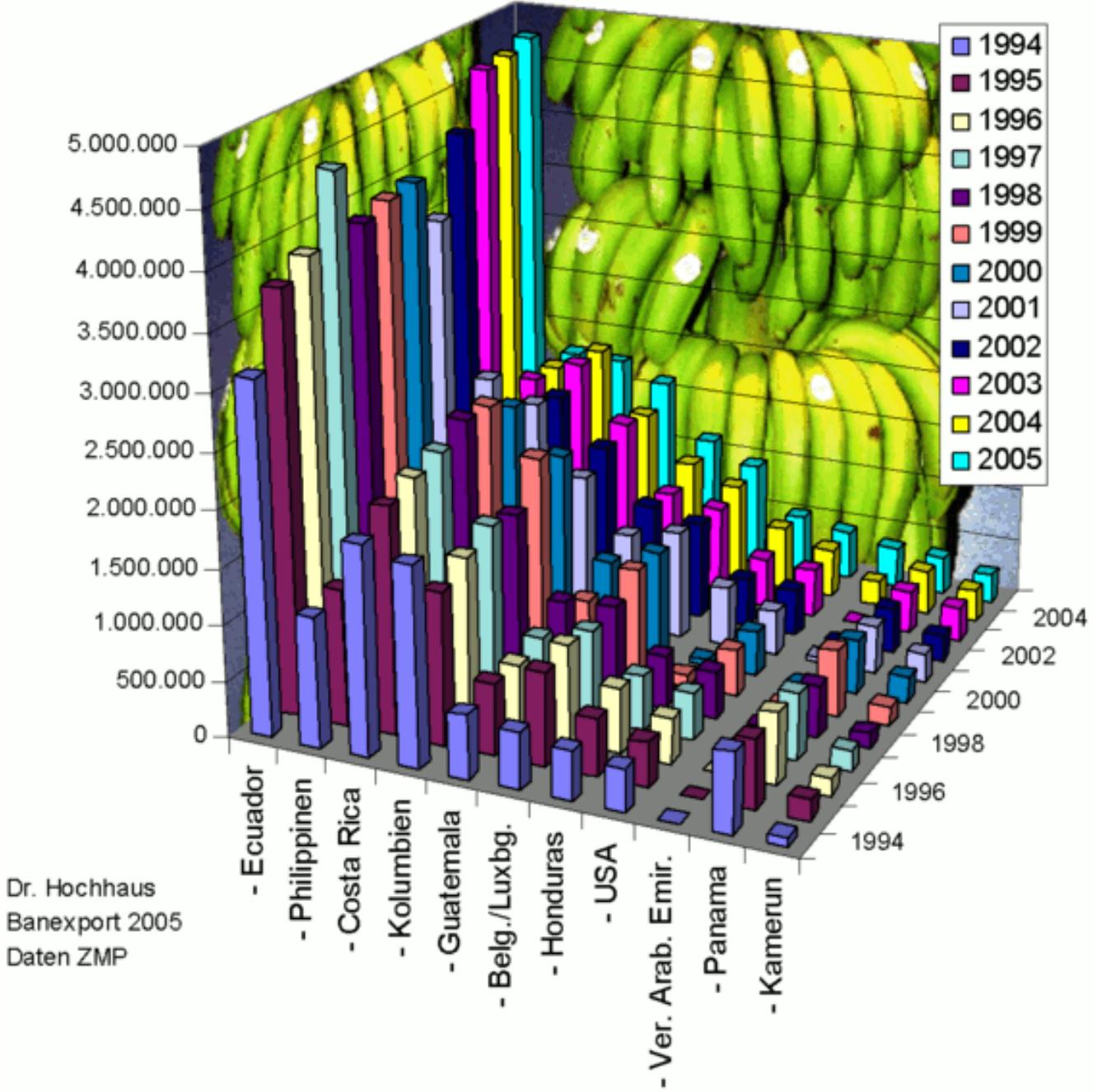
Plot gallery

What follows is a gallery of plots: some good, some bad, and some ugly too. Let's use these to explore what works and what doesn't. For each plot, ask yourself three questions:

- What makes this plot good (as in, effective and not misleading)?
- What makes this plot bad (ineffective and/or misleading)? How could the plot be improved?
- What might make this plot prettier?

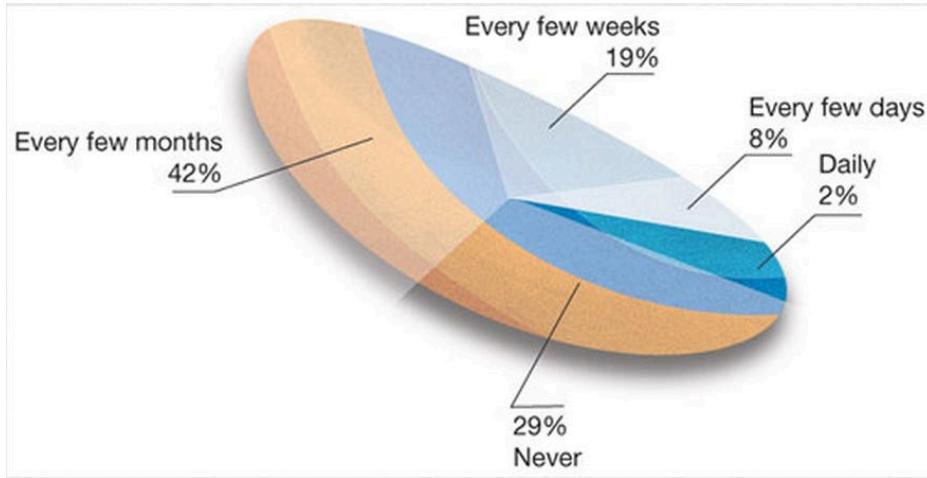
The point of this is not to make fun of others for their plots. The point is to learn from their choices. Because plot technique matters. Data science is about communication, action, and impact. You will spend so much time working on an analysis, and you are gonna go through all the work of making a plot. What a shame if the end product undermines all of that hard work!

Export von Bananen in Tonnen von 1994-2005



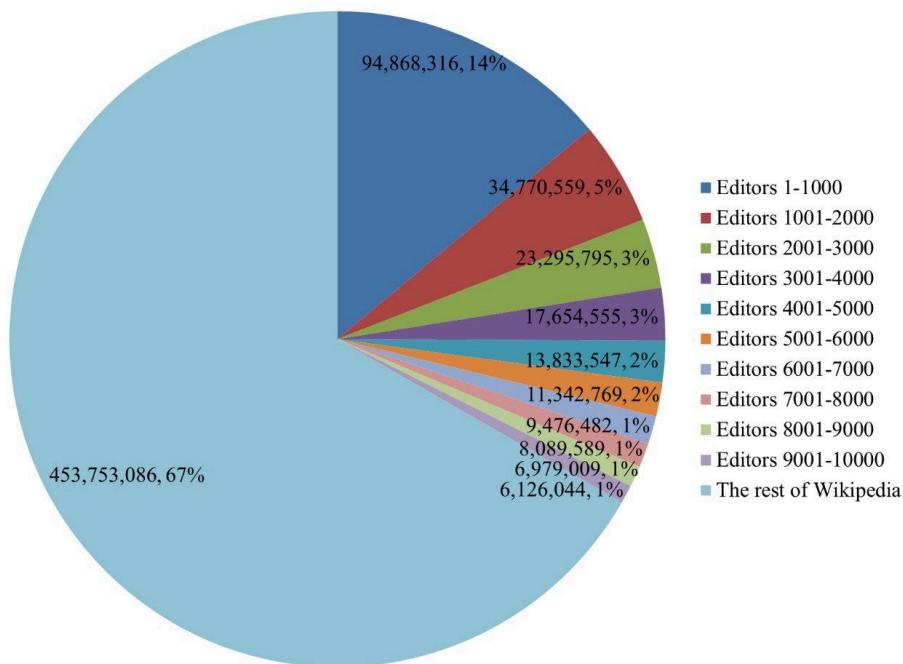
- **Good:** Frankly, there is not much good about this plot. Yes, it has a lot of information, but this crosses the line into information overload. It is so convoluted and difficult to interpret that we quickly lose interest in spending time exploring its details.
- **Bad:** The 3D perspective (1) makes it almost impossible to compare bar heights, (2) causes a lot of the bars in the back to be hidden, and (3) adds needless complexity.
- **Bad:** The 3D perspective makes it almost impossible to compare bar heights.
- **Bad:** The colors representing each year do not follow a logical sequential flow; years are sequential, and colors can be too (think the ROYGBIV rainbow sequence).

- **Ugly:** The bananas! Sure, this plot has to do with banana exports, but those banana pictures don't represent anything at all about the data and they make everything else convoluted. Plus, it's cheesy.



We don't think this plot is good or pretty.

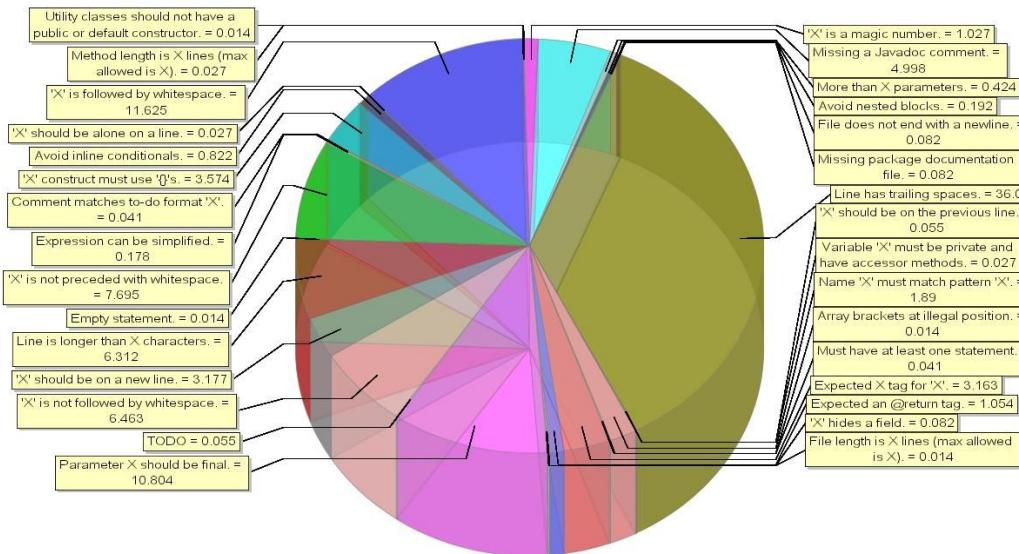
- **Bad:** This is an unfamiliar plot format; is it a pie chart? A blood platelet? A pickle?
- **Bad / Ugly:** Why is it lopsided or rotated? That has nothing to do with the data.
- **Bad:** What is this plot even about? There are no context clues whatsoever. Titles and labels, in moderation, can be really helpful.
- **Bad / Ugly:** What are the colors representing? They seem to have no relation at all to the pie slices. Very confusing.
- **Bad:** The slices do not seem to represent the percentages accurately. The 8% slice does not look four times larger than the 2% slice.



Here is another pie chart that isn't very effective.

- **Bad:** Lots of significant digits in these numbers. Instead of forcing viewers to read numbers like 453,753,086, why not display 454 M?
- **Bad:** The percentages next to the other numbers make it even harder to read.
- **Bad:** Superimposing text on top of the pie slices makes it impossible to use the slices for their intended purpose: visually comparing the size of subgroups in the data.
- **Bad:** There are so many color-coded slices that it takes far too long to understand the details.
- **Bad:** One reason it takes so long is that the text is redundant: don't put a lot of text on the pie *and* put a lot of text in your legend. Figure out a way to point to each pie slice with a line, then have all the info for that slice in the same spot.
- **Bad / Ugly:** The dark text on top of dark colors is hard to read.

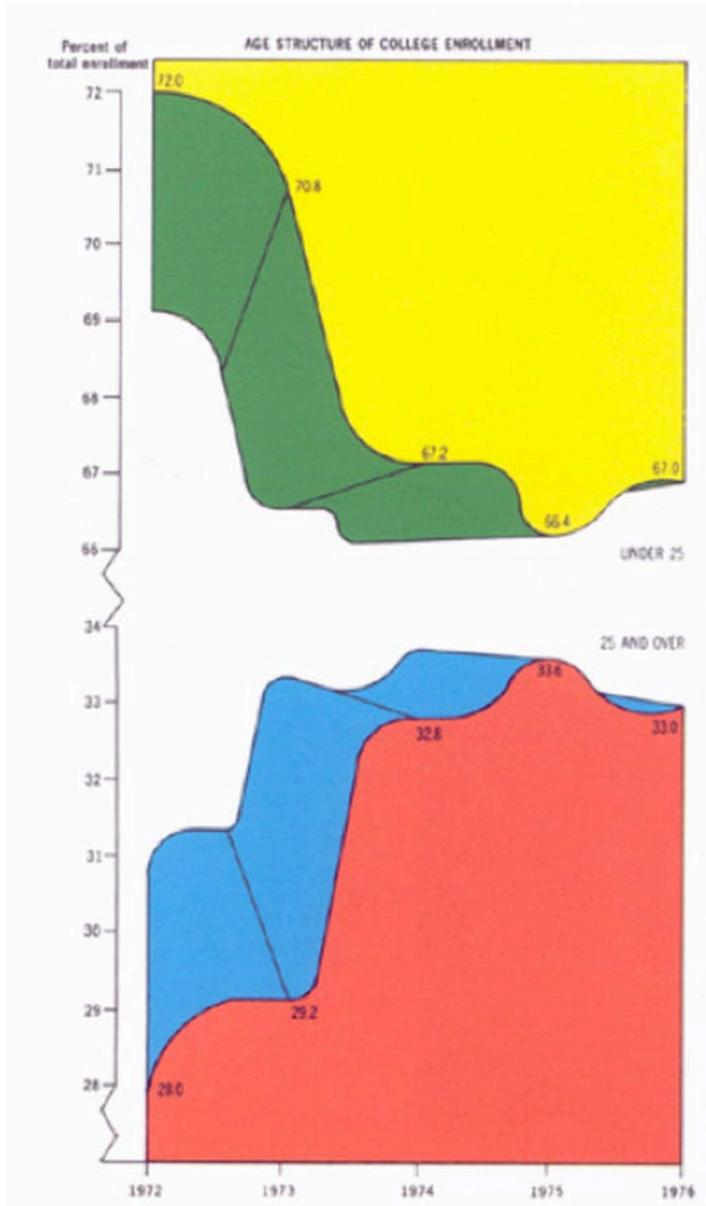
Pie charts are *super* common in media, but they are actually an infamously bad form of data visualization. That's because the human eye is much worse at comparing *areas*, such as the size of a pie slice, than they are at comparing *heights*. To make matters worse, we are worse at comparing areas for non-rectangular shapes, like a pie slice, than for squares or rectangles. So: avoid pie charts.



- **Bad:** Pie chart.
- **Bad:** There is no reason for this to be 3D. The third dimension has nothing to do with the data.
- **Bad:** There is no reason for this to be semi-transparent. It just makes everything even more convoluted.
- **Bad:** The text is way too small to read.
- **Bad:** On a related note, there is way too much text.
- **Bad / Ugly:** The yellow text boxes and dark lines around them add uninformative junk to this plot. If all lines unrelated to data or labels were removed, this chart would be more intelligible.



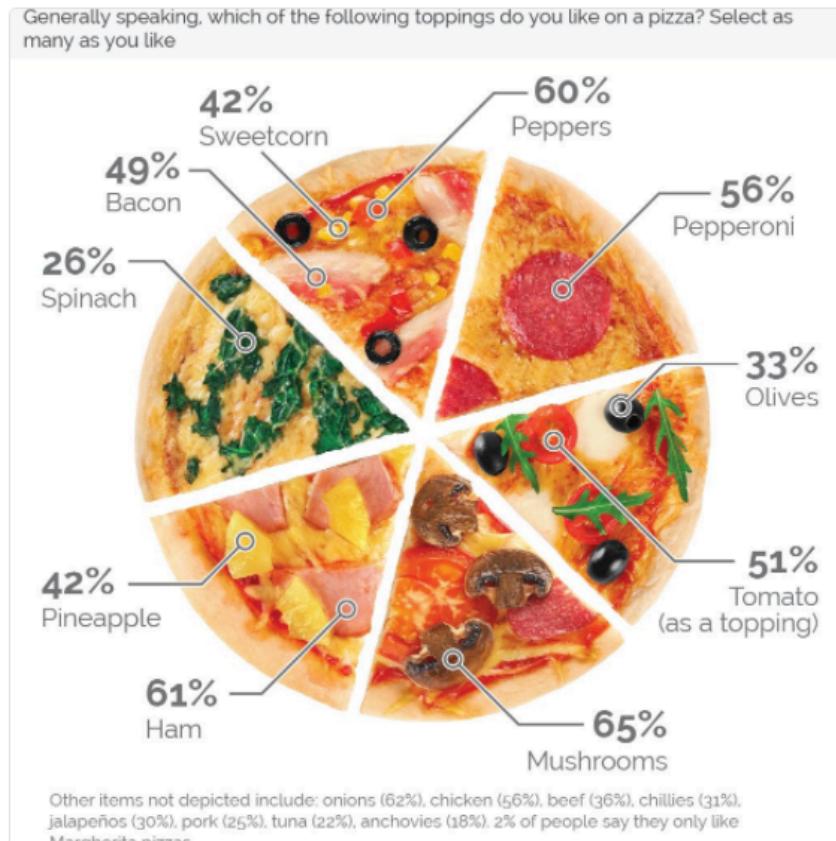
- **Bad:** Information overload.
- **Bad:** It is not clear whether the repeat use of colors in each row conveys any meaning, or if it is just a random recycling of colors.
- **Bad:** The text is too small to read.
- **Bad:** Abbreviations are not explained.



- **Bad:** The text is very small.
- **Bad / Ugly:** The third dimension, with a weird perspective effect added, has nothing to do with the data and makes this plot difficult to understand. Should you pay attention to the edge in the distance or the line in the foreground? Are they the same?
- **Bad:** The y axis is crazy! (1) The scale break is confusing. (2) The attempt to plot these two subgroups as separate trends belies the fact that one is just the remainder of the other: the two curves sum to 100%. (3) By attempting to place the two trends on the same proportional scale, this plot gives the visual impression that the changes over time are really extreme.
- **Bad:** The trend lines are unnecessarily curved. The plot's authors probably only have data for each semester, but smoothed lines give the impression that they have more data.


[Follow](#)

Forget pepperoni - mushroom is Britain's most liked pizza topping (65%), followed by onion (62%) and then ham (61%)
yougov.co.uk/news/2017/03/0 ...



4:00 AM - 6 Mar 2017

364 Retweets 549 Likes



179

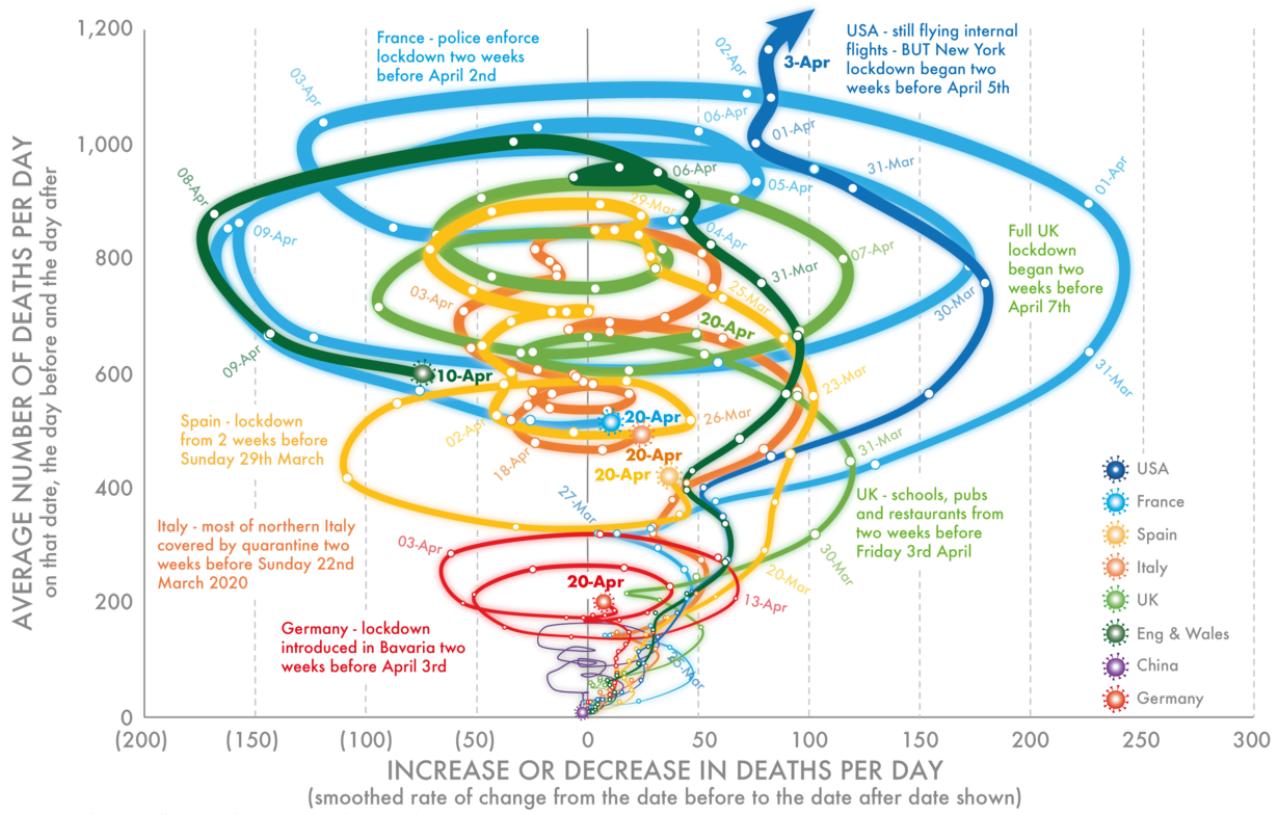
364



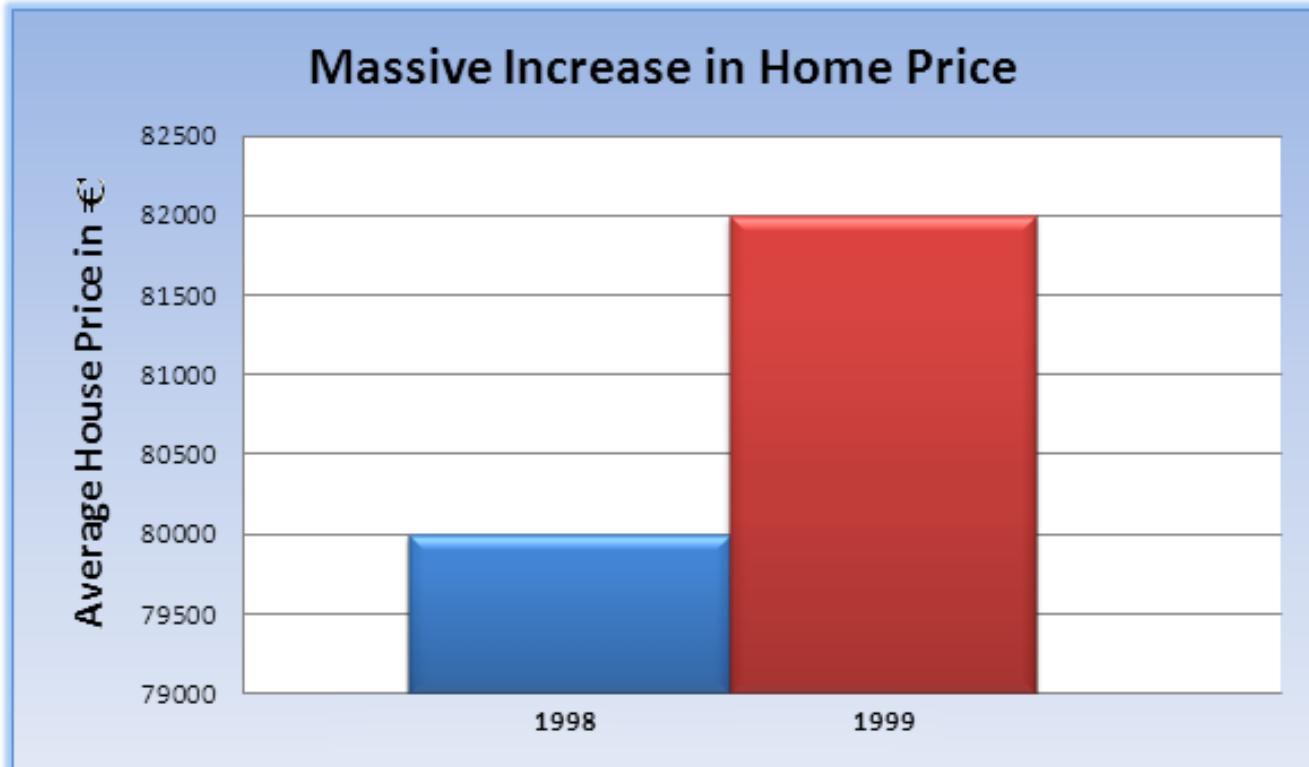
549



- **Bad:** These percentages sum to more than 100%. That needs to be explained, or avoided.
- **Bad:** The use of a pizza pie, though creative, implies that the size of the pie slices will have something to do with the data. They don't.
- **Bad:** Check out the fine print on the bottom. Several toppings were left off this chart, and some of them were quite popular; for example, onions had 62% popularity and chicken was 56% popular. Why are they not on this chart but spinach (26%) is? The arbitrary exclusion of categories should immediately make viewers suspicious: how are these decisions being made?

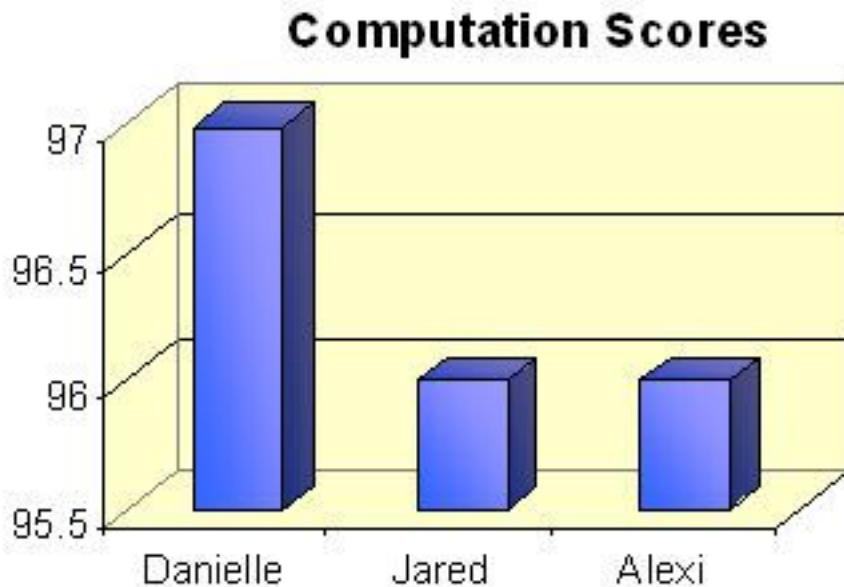


This is another case of information overload, and another case of a creative visualization that doesn't really help us make sense of what's going on. If we wanted to understand the take-a-way message or punchline from this plot, we're not sure we'd be able to. We could stare at this for minutes and still not understand what we are looking at, and it is so complicated that we would rather ignore this plot than go through that effort.



- **Bad:** The y axis range makes the difference in home prices sound a lot larger than it actually is. Based on the height of these two bars, you would expect the 1999 price to be 3x the 1998 price when, in actuality, it is larger by 2.5%.
- **Bad:** The use of qualifiers such as “Massive” is generally discouraged. It’s a little too controlling. Let the viewers make up their own minds about which differences are substantial and which are trivial.

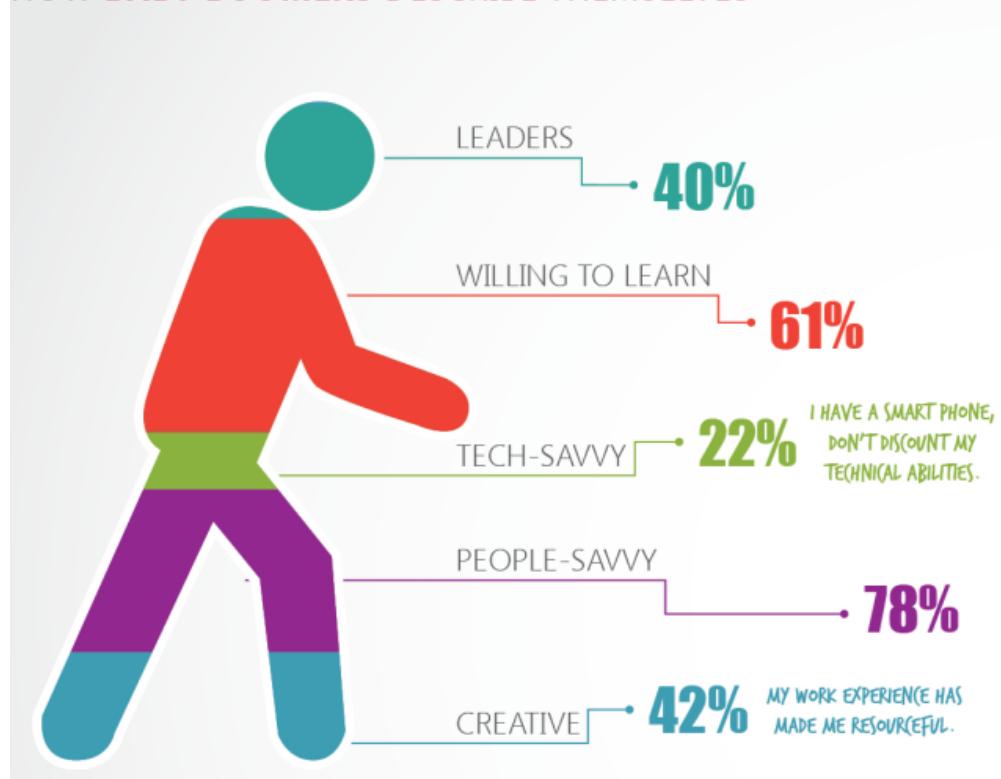
These two notes are classic indicators of misleading or agenda-driven data visualization.



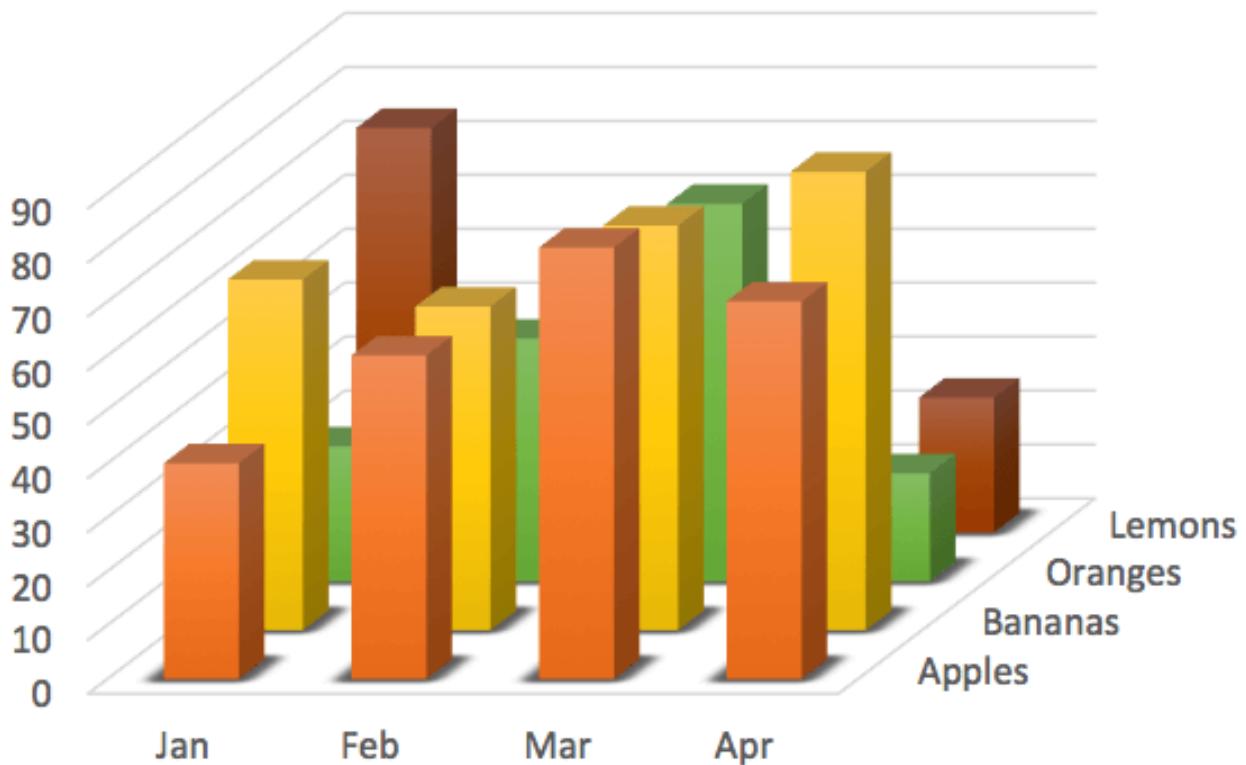
This is another example of manipulating perception with a sneaky y-axis range.

Also, like the examples above, this plot doesn't have to be 3D. The height has a meaning, but what does the depth mean? It is junk, in the sense that it does not add meaning.

HOW BABY BOOMERS DESCRIBE THEMSELVES



- **Bad:** These percentages sum to more than 100%. That needs to be explained, or avoided.
- **Bad:** The rationale for ordering of the categories is not clear. They are not ranked by percentage. Are they supposed to correspond to the body part being pointed to in the figure? Should crotch equal ‘tech-savvy’?
- **Bad:** The geometric shape itself (the person figure) does not help at all in visualizing the percentages. Can you compare the area of the feet and the area of the head? The fact that they provide the actual numbers in large bold font suggests that they knew the figure would not be useful as a picture.



- **Bad:** Some data are completely obscured. What is happening to lemons in February and March?
- **Bad:** Like the infamous banana plot above, the 3D perspective here makes it almost impossible to compare bar heights and adds needless complexity.

Anatomy of a Winning TED Talk

1%

Sophisticated Visual Aids

We're not sure who puts the D in TED—most of the best presentations favor tepid PowerPoint slide shows (sorry, Brené Brown), Pictionary-quality drawings (really, Simon Sinek?), or no props at all.

5%

Opening Joke

Remember the one about the shoe salesmen who went to Africa in the 1900s? That's how Benjamin Zander opened his talk—which turned out to be about classical music.

5%

Spontaneous Moment

Don't overprepare. Tease the guy in the front row ("You could light up a village with this guy's eyes"). Command the stagehand who handles the human brain you brought.

5%

Statement of Utter Certainty

People come for answers—give 'em what they want, as Shawn Anchor did: "By training your brain ... we can reverse the formula for happiness and success."

12%

Snappy Refrain

The TED equivalent of "I have a dream." Example: "People don't buy what you do; they buy why you do it." Repeat 7x.

23%

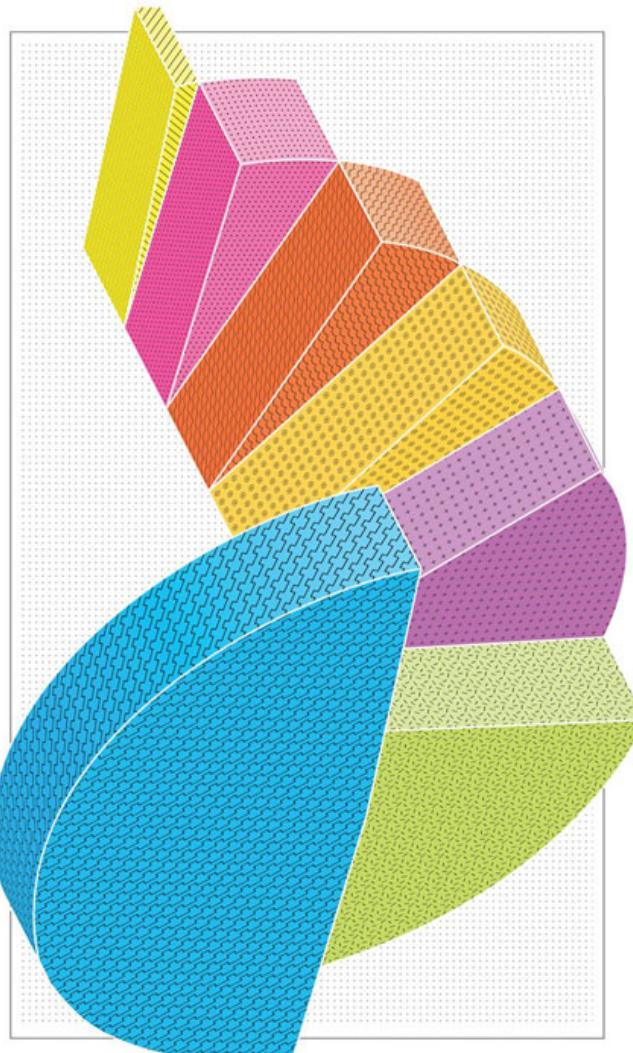
Personal Failure

Be relatable. We want to know about that nervous breakdown. Or at least the time you didn't fit in at summer camp.

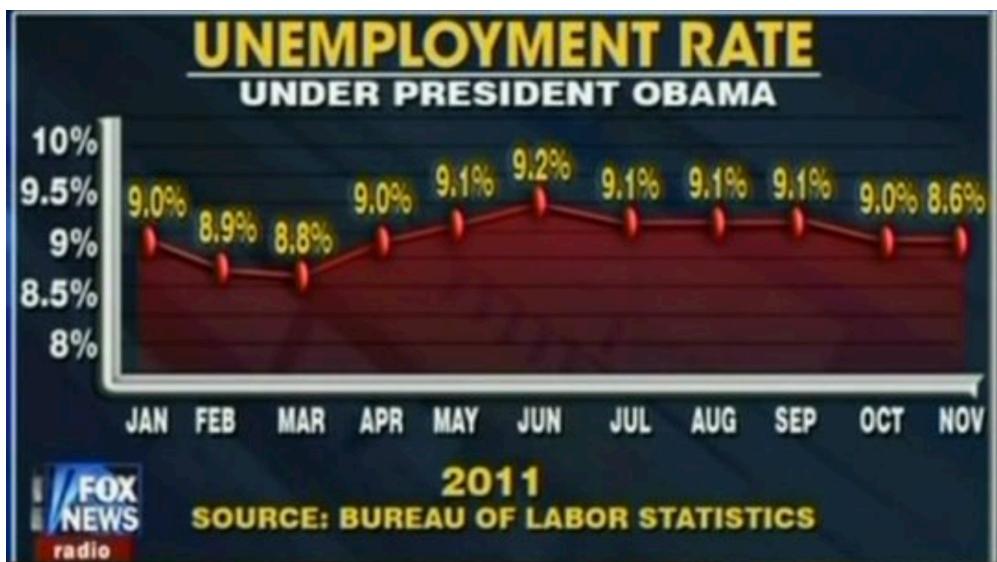
49%

Contrarian Thesis

Wait a sec—we should be playing *more* videogames? The more choices we have, the worse off we are? TED is where conventional wisdom goes to die.

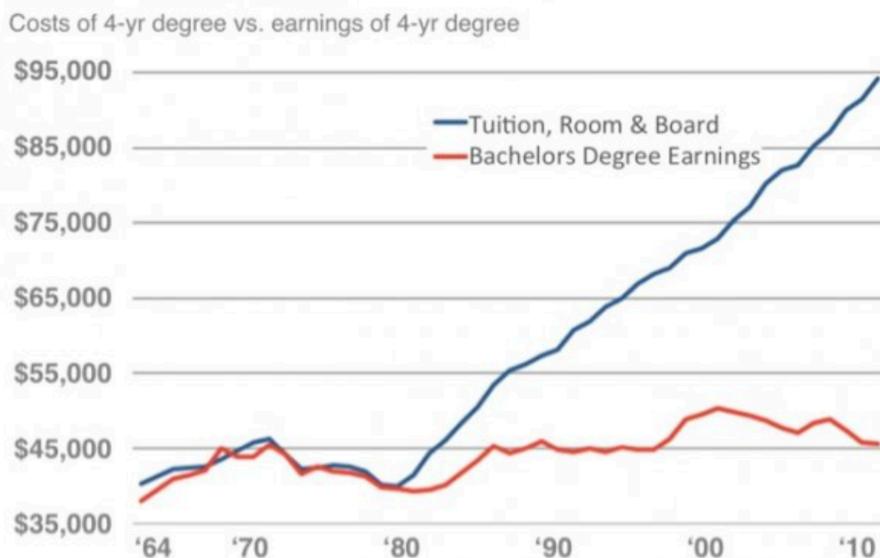


- **Bad / Ugly:** The contortion of this pie chart into a spiraling 3D object is confusing and gratuitous.
- **Bad / Ugly:** Some colors are very similar to each other.
- **Bad:** Pie chart!



- **Bad:** The small y axis range exaggerates changes in the unemployment rate.
- **Bad:** The final data point on this plot is wrong! 8.6% should not be at the same height as 9.0%.
- **Bad:** The title of this plot implies that the data will show unemployment trends throughout the Obama administration, which began in 2009, but this data is for 2011 only.

The diminishing financial return of higher education

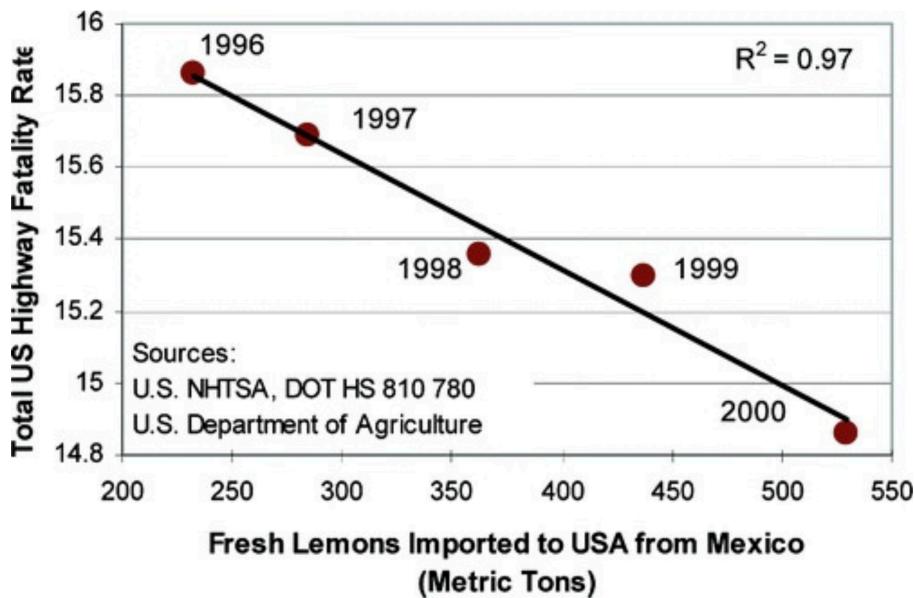


Source: U.S. Census Data & NCES Table 345.

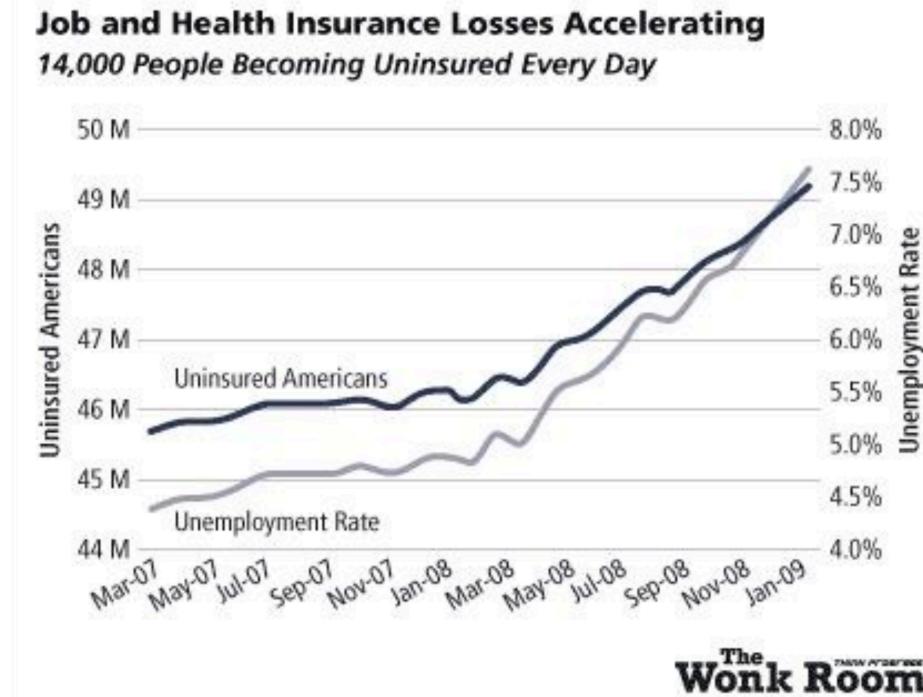
Notes: All figures have been adjusted to 2010 dollars using the Consumer Price Index from the BLS.

- **Good:** All in all, this is a very well made plot. Unnecessary lines and text are kept to a minimum; the title and subtitle are clear. The text size is appropriate. The source of the data is provided.
- **Bad:** This plot is misleading, because it is plotting data with different units on the same y axis. The blue line is the average *four-year* cost of a college degree. The red line is average *annual* salary for someone with a Bachelor's degree.

How should these data be plotted differently in order to correctly explore whether a four-year degree is still worthwhile in terms of its benefits to a 30-year career?



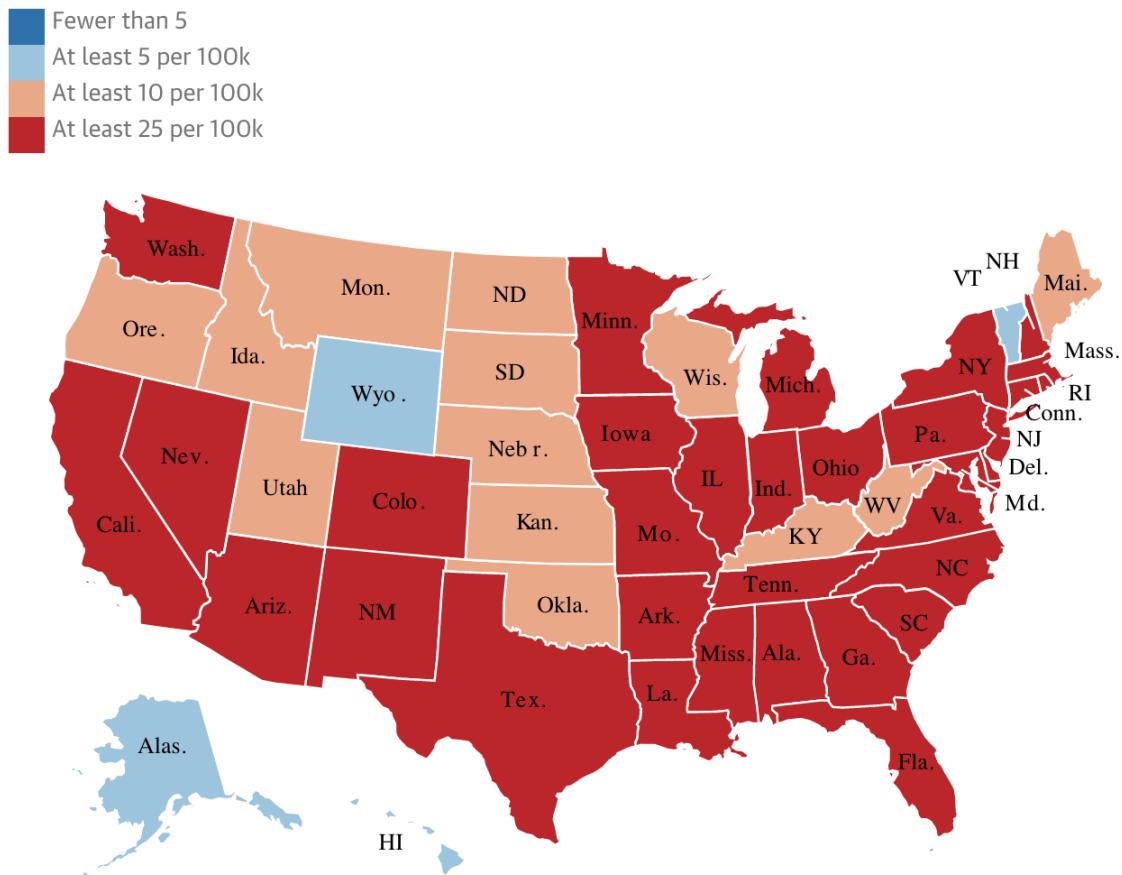
This is a classic example of a *spurious correlation*: two trends that are correlated but have absolutely nothing to do with each other.



This is another example of a beautiful plot, but it is also an example of how a plot's message can be coaxed by

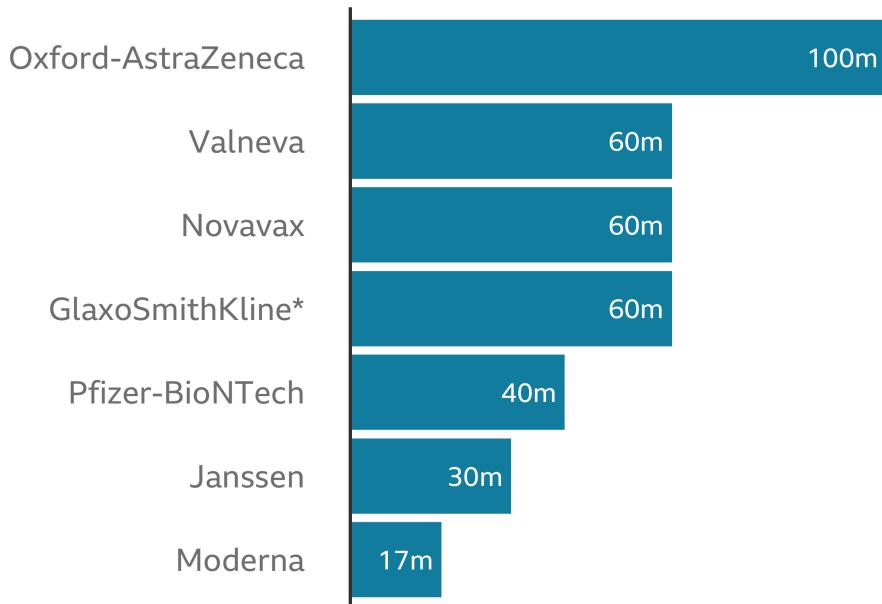
manipulating y axis scales. If the curve for Uninsured Americans were plotted on the same percentage scale as the Unemployment Rate, it would not seem to be accelerating so rapidly.

Number of confirmed Covid-19 deaths per 100,000 Americans



- **Good:** This is a clean and simple plot, more or less, without too much text.
 - **Good:** The color scale relates to only 4 categories: that is simple enough to make sense of quickly.
 - **Good:** The color palette follows an intuitively sequential trend: Blue = low/no bad; Red = high/severe.
 - **Bad:** The abbreviation system for states is inconsistent. Some use initials, some use abbreviations; some use a period, some don't.
 - **Bad:** The lowest color category is not used in this plot, and could be removed to increase simplicity.
 - **Bad:** Showing a map of the U.S. states with this particular color scale can be confusing: if you had to guess what this chart is about, you would probably assume it is an electoral map.

How many millions of doses of vaccine has the UK ordered?

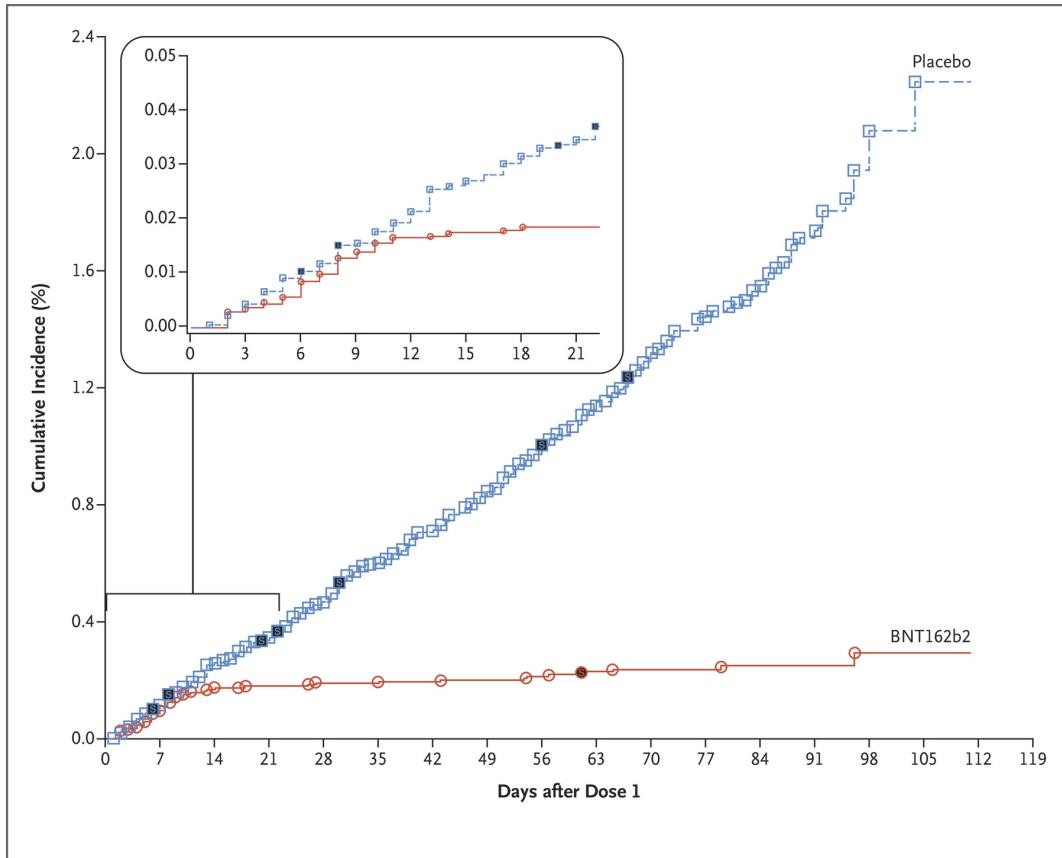


*Joint project with Sanofi Pasteur

Source: UK government, 8 January

BBC

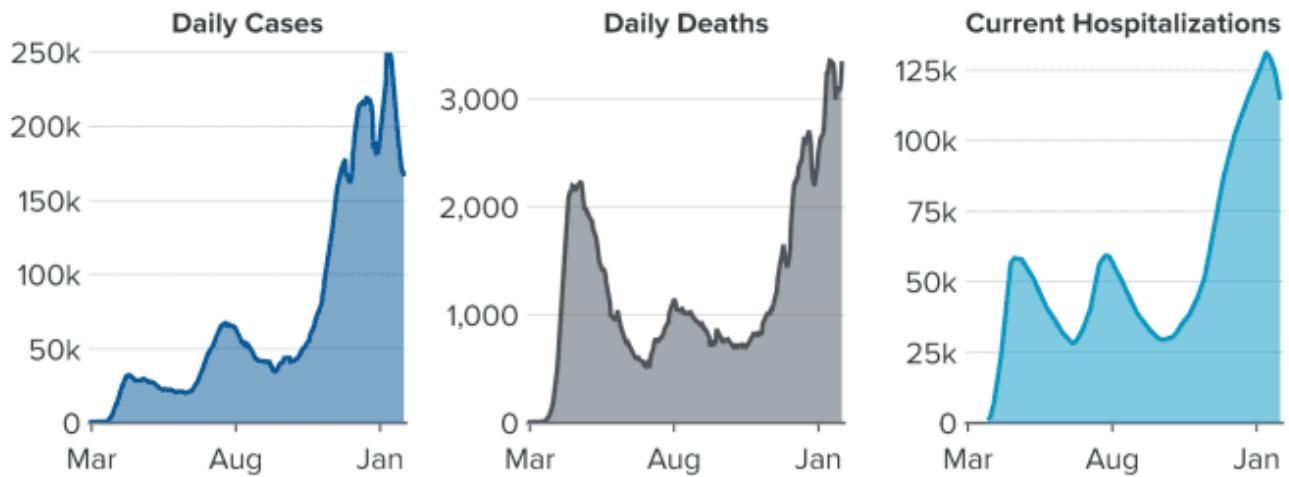
- **Good:** This is a nice plot. It is simple and junk-free. No unnecessary text or axes.
- **Good:** Large font, strong color contrast.
- **Good:** The axis starts at zero, and the bars widths are proportional to the data, e.g., the 30m bar is half the width of the 60m bar.
- **Good:** Since this plot is so simple, the actual numbers for each bar can be included without cluttering the plot.



- **Good:** This is a simple plot that tells a good story: a week or so after receiving a dose, vaccinated participants contracted COVID-19 at a much lower rate than those who received the placebo.
- **Bad:** The labels are not clear to viewers who are not clinical virologists. To every extent possible, data visualizations should be inclusive and inviting. Don't make someone feel stupid by forcing them to look at their plot.
- **Bad:** The overlay that zooms in on the first three weeks makes this plot a bit cluttered. We might suggest plotting those first three weeks in a separate plot, adjacent to this one but not embedded within it.
- **Ugly:** This is an effective plot, but it is not beautiful. What would you do to make this plot more beautiful without compromising its message?

Coronavirus in the U.S.

Seven-day average lines



SOURCE: Johns Hopkins University (cases/deaths), Covid Tracking Project (hospitalizations). As of 1/26.

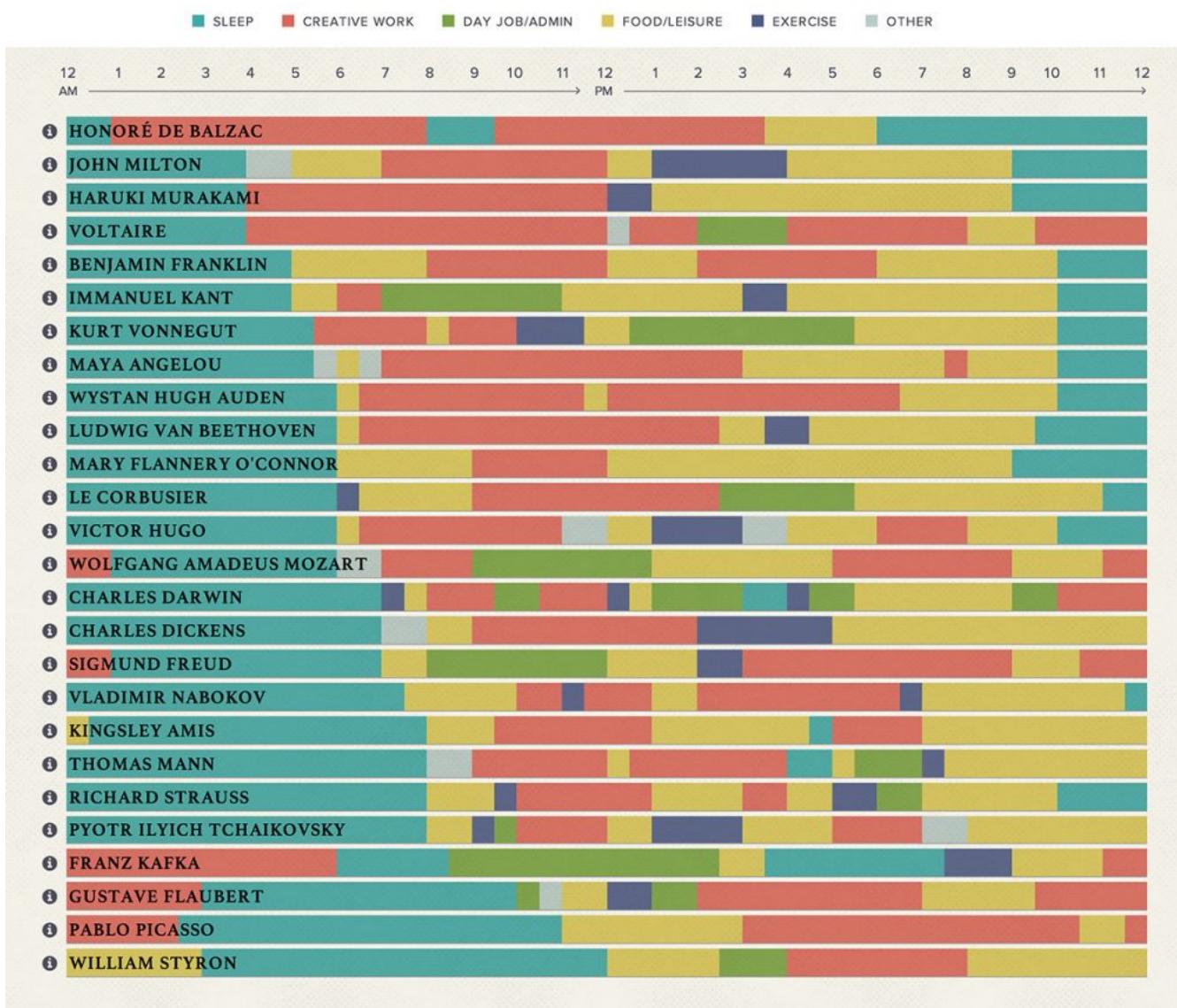


- **Good:** These are clear and simple plots with an obvious take away: daily COVID-19 case counts correspond to deaths and hospitalizations.
- **Good:** No extraneous labels or lines. These plots are chart-junk free.

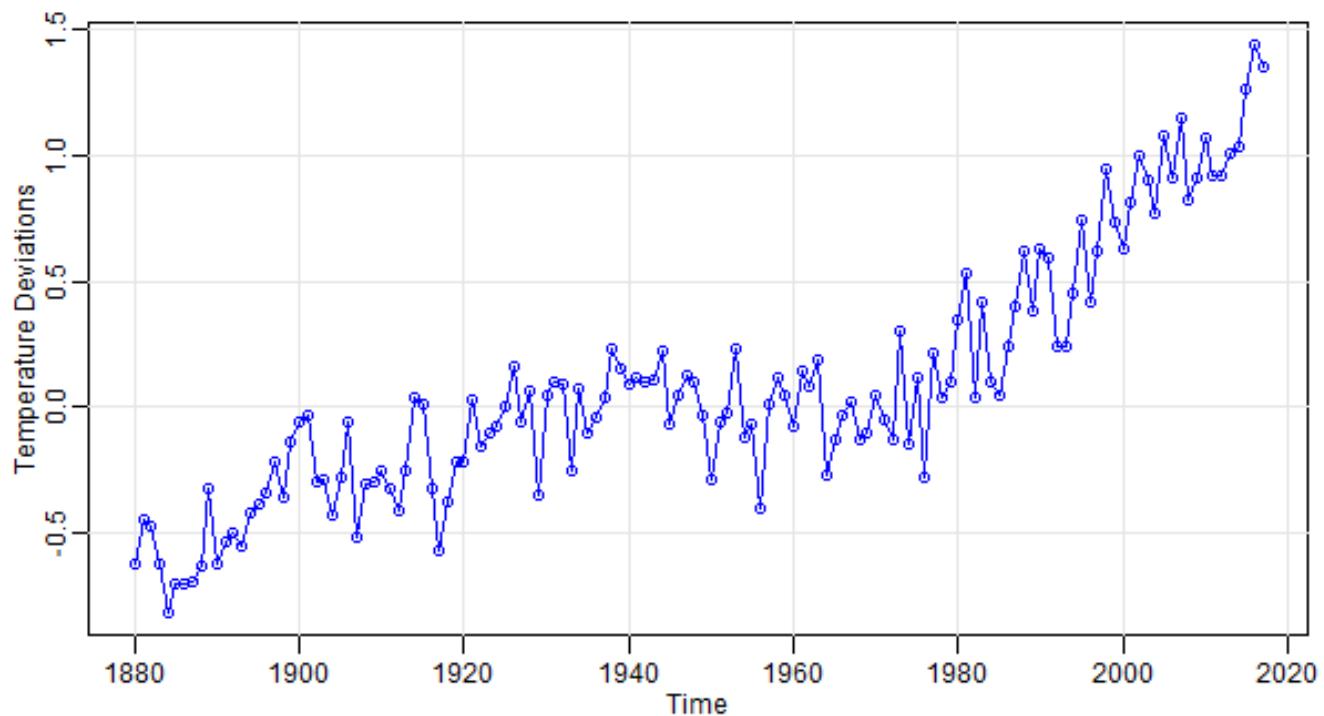
What do you think about the choice to use three different scales for the y axis? That tends to lead to confusion, but do you think that, in this case, it was justified?

THE DAILY ROUTINES OF

FAMOUS CREATIVE PEOPLE

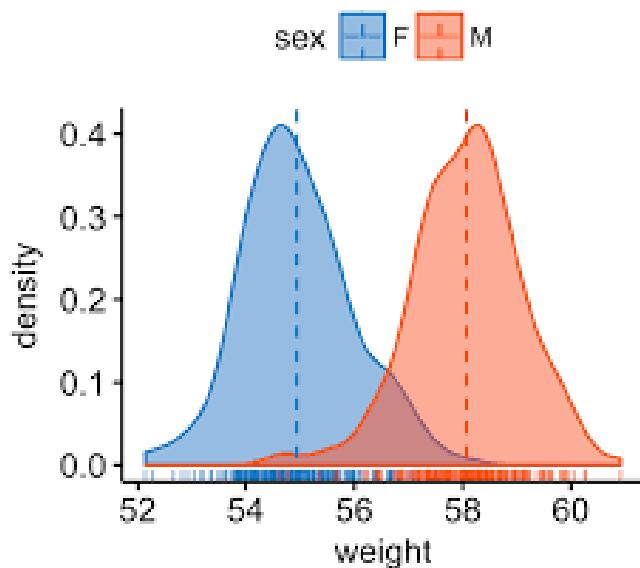


Now this is a data visualization! It is a tad complicated, but it is elegant and fun to explore.



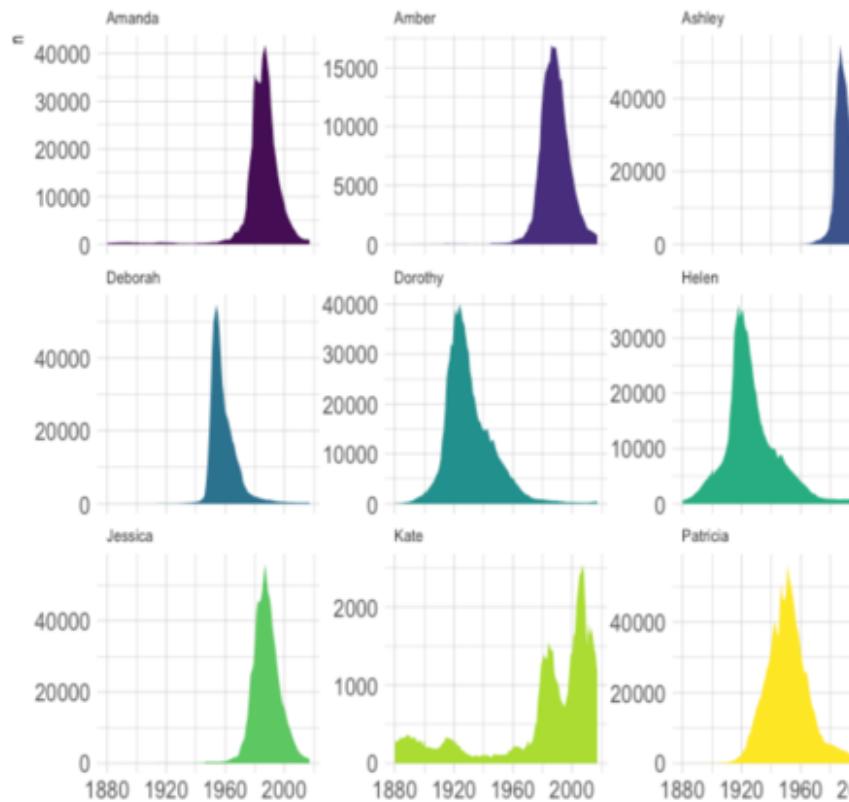
This plot is straightforward and simple.

- What it is showing is fairly self-explanatory, though some viewers might benefit from a more informative y-axis label.
- To help explain the y axis, it may be helpful to include a helper line at 0.0 degrees.
- Are the points really necessary? Since it is fairly clear that this is an annual dataset, those points are just repeating the information contained in the line, aren't they?



Another straightforward and simple plot, whose meaning is pretty obvious even without a title.

- Some more effort could have gone into the x axis label. Are the units kilograms, or pounds?
- This plot conveys a lot of information really intuitively. You can intuit that the dotted lines are probably mean weights; that the distributions show the range of data.
- The semi-transparent colors make it possible to see how the two distributions overlap. Very helpful!
- It is interesting that the male/female color associations are opposite the convention. Do you think that was intentional?
- Note the little tick marks along the x axis. That is a subtle way of indicating sample size; it shows how much data are used to populate each distribution.



Some baby names throughout the last 140 years

- In this plot, chart junk is kept to a minimum, which is nice, but the names are a little too small to read. It is strange that the names are a smaller font size than the axes labels.
- What do you think about the use of color? The colors are pretty, but it is a purely aesthetic choice; the colors don't correspond to anything about the data at all.
- It was an interesting decision to only include x axis labels on the bottom row of plots. In a way this is nice because it (1) reinforces the fact that each plot is using the same x axis range, and (2) removes redundant content from the plot. However, it makes it a bit more laborious to explore the plots in the top row.
- It was also an interesting decision to make the y axis ranges different for each plot. There are trade-offs to this decision. What would be lost if all of these facets were forced to use the same y axis range?

Chart junk

A few times already, we have referred to the concept of chart junk. This refers to the idea that the best plots are the ones that minimize the ink-to-data ratio. In other words, there should be no extraneous or unnecessary ink on your plot.

The chart junk principle applies to both graphical and tabular representations of data. Which of these tables is easier to read?

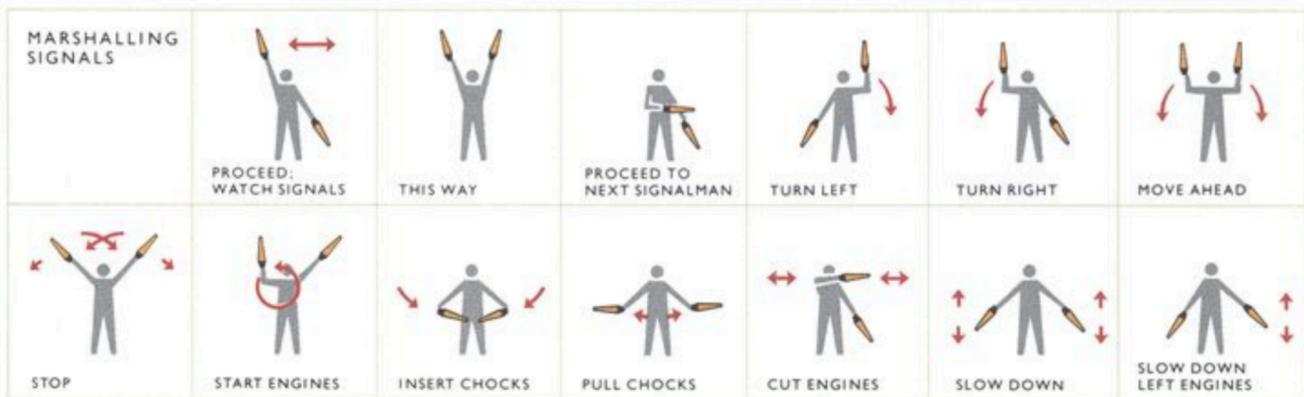
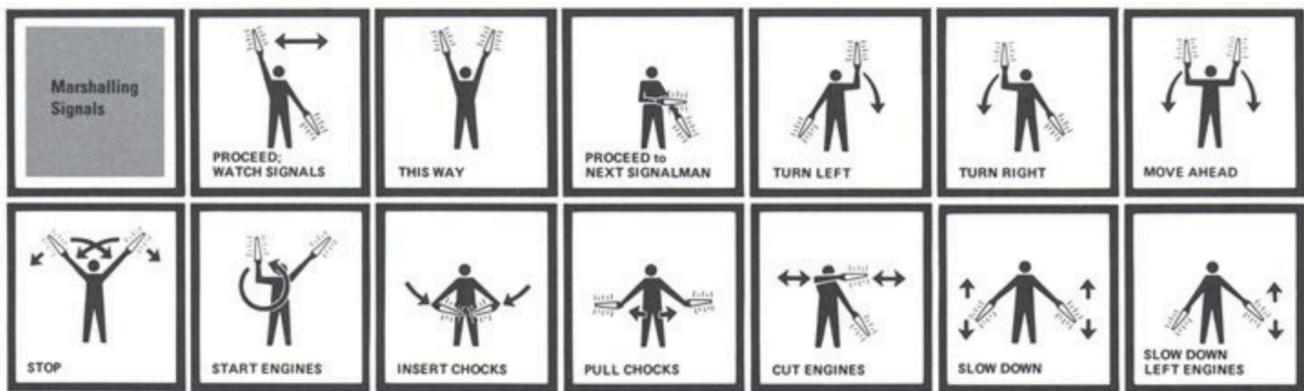
Before

Team	1999	2000	2001	2002	2003	2004	2005	2006
Arizona Diamondbacks	\$61,184,250	\$72,345,275	\$72,095,020	\$77,893,950	\$80,657,500	\$80,521,550	\$88,348,000	\$96,943,475
Atlanta Braves	\$68,134,250	\$70,441,200	\$74,073,950	\$75,379,325	\$96,872,425	\$79,021,000	\$85,148,575	\$79,703,300
Baltimore Orioles	\$73,034,250	\$74,441,200	\$76,424,250	\$76,300,525	\$89,479,250	\$89,479,250	\$84,354,075	\$84,354,075
Boston Red Sox	\$74,142,150	\$64,963,275	\$62,492,900	\$96,300,525	\$89,479,775	\$104,340,450	\$109,714,225	\$109,203,500
Chicago White Sox	\$22,740,725	\$26,839,225	\$57,743,150	\$52,826,750	\$49,048,075	\$62,704,325	\$59,655,550	\$58,915,075
Chicago Cubs	\$51,889,225	\$60,449,450	\$51,553,075	\$67,581,100	\$72,092,250	\$78,630,925	\$77,866,900	\$84,679,625
Cincinnati Reds	\$28,188,575	\$43,391,550	\$43,488,360	\$37,542,000	\$56,874,900	\$58,485,450	\$49,716,225	\$63,111,200
Cleveland Indians	\$60,769,300	\$72,962,275	\$76,446,925	\$66,787,875	\$79,282,925	\$78,807,750	\$76,060,700	\$56,793,875
Colorado Rockies	\$7,374,250	\$8,484,200	\$8,484,200	\$8,484,200	\$8,484,200	\$8,484,200	\$8,484,200	\$8,484,200
Detroit Tigers	\$39,460,000	\$39,540,225	\$44,492,150	\$49,160,000	\$47,772,325	\$41,387,100	\$81,459,525	\$76,201,650
Florida Marlins	\$17,277,775	\$17,303,450	\$29,586,000	\$57,480,075	\$43,185,975	\$38,995,175	\$56,593,675	\$14,421,625
Houston Astros	\$49,443,275	\$47,489,925	\$55,599,075	\$68,741,251	\$67,778,700	\$74,661,350	\$73,825,975	\$88,991,025
Kansas City Royals	\$32,794,225	\$20,922,325	\$30,726,750	\$40,730,000	\$38,659,125	\$39,674,175	\$34,149,075	\$46,770,750
Los Angeles Dodgers	\$70,759,250	\$81,871,200	\$73,209,000	\$70,200,000	\$70,244,250	\$67,624,325	\$57,830,375	\$70,830,375
Los Angeles Angels	\$7,265,275	\$7,388,000	\$154,975,000	\$73,179,475	\$73,179,475	\$73,179,475	\$73,179,475	\$73,179,475
Milwaukee Brewers	\$18,129,400	\$26,519,800	\$59,497,525	\$43,351,575	\$55,023,275	\$27,514,650	\$49,734,825	\$56,790,000
Minnesota Twins	\$18,162,400	\$16,884,125	\$22,548,000	\$38,877,875	\$53,446,350	\$61,524,050	\$52,421,300	\$61,350,825
Montreal/Washington Nationals	\$14,977,325	\$30,006,750	\$88,978,750	\$34,527,225	\$49,950,950	\$35,997,925	\$49,484,575	\$52,722,325
New York Mets	\$37,524,475	\$79,600,775	\$83,191,400	\$50,993,850	\$100,748,800	\$96,758,950	\$97,609,400	\$97,020,725
New York Yankees	\$37,524,475	\$79,600,775	\$83,191,400	\$50,993,850	\$100,748,800	\$96,758,950	\$97,609,400	\$97,020,725
Oakland Athletics	\$34,340,700	\$29,603,075	\$31,326,000	\$36,741,025	\$48,423,875	\$55,393,625	\$53,720,450	\$62,320,075
Philadelphia Phillies	\$26,118,525	\$40,780,750	\$48,061,700	\$65,741,525	\$61,917,250	\$86,334,050	\$91,471,075	\$81,738,625
Pittsburgh Pirates	\$18,498,050	\$27,815,700	\$42,496,650	\$36,485,850	\$48,696,300	\$29,840,675	\$34,047,325	\$41,841,200
San Diego Padres	\$42,703,875	\$45,684,175	\$35,493,025	\$35,711,200	\$37,658,325	\$54,630,500	\$56,150,175	\$62,250,625
Seattle Mariners	\$48,941,925	\$56,640,050	\$87,446,075	\$80,282,675	\$80,726,400	\$72,807,000	\$87,496,350	\$84,924,410
St. Louis Cardinals	\$42,141,250	\$42,141,250	\$42,141,250	\$42,141,250	\$42,141,250	\$42,141,250	\$42,141,250	\$42,141,250
St. Louis Cardinals	\$42,141,250	\$56,901,725	\$66,412,000	\$71,241,325	\$47,857,175	\$75,431,525	\$59,730,425	\$88,073,625
Tampa Bay Rays	\$29,269,400	\$60,817,050	\$56,881,125	\$30,896,425	\$19,630,000	\$27,321,000	\$26,490,675	\$51,621,175
Texas Rangers	\$71,956,670	\$68,071,050	\$71,374,120	\$50,777,750	\$87,195,400	\$47,263,775	\$46,089,375	\$79,790,075
Toronto Blue Jays	\$42,797,425	\$44,459,925	\$87,677,225	\$66,262,350	\$47,480,550	\$48,693,275	\$43,621,625	\$66,587,925
Average	\$43,336,913	\$49,875,634	\$56,243,975	\$59,605,910	\$53,677,748	\$62,107,270	\$56,561,310	\$72,013,975

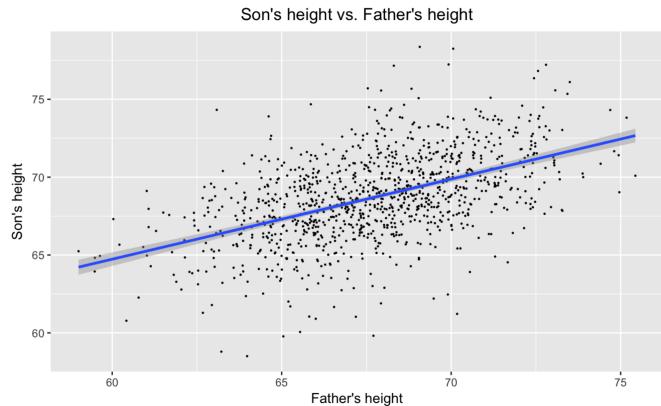
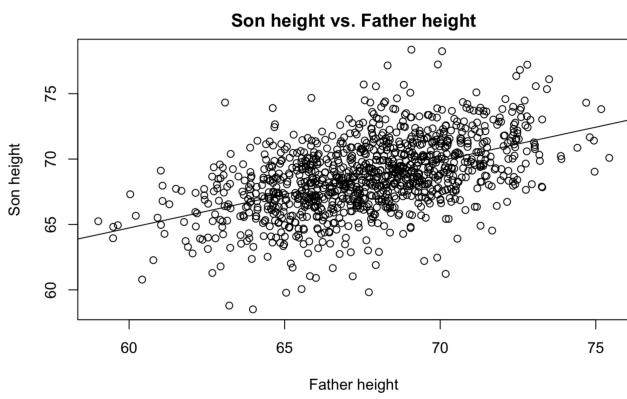
After

Team	Average Salary (\$ millions)							
	1999	2000	2001	2002	2003	2004	2005	2006
Arizona Diamondbacks	61.2	72.3	72.5	77.9	80.7	60.5	58.3	55.9
Atlanta Braves	68.1	70.4	74.1	75.4	96.9	79.0	85.1	85.1
Baltimore Orioles	73.1	70.2	62.4	47.3	59.9	45.7	66.6	64.8
Boston Red Sox	55.1	65.0	85.6	90.3	89.5	104.3	108.3	111.2
Chicago White Sox	22.7	26.8	57.7	52.8	49.0	62.7	69.7	98.9
Chicago Cubs	51.9	50.4	61.6	67.6	72.1	79.5	77.9	84.7
Cincinnati Reds	28.6	43.4	43.5	37.5	50.9	39.5	49.7	53.1
Cleveland Indians	60.8	73.0	76.6	65.8	39.4	28.8	36.1	56.8
Colorado Rockies	53.7	54.6	65.8	52.6	55.8	57.7	41.2	34.3
Detroit Tigers	30.5	53.9	44.5	49.2	47.3	41.4	61.6	76.2
Florida Marlins	17.5	17.3	29.6	37.5	43.2	39.0	55.9	14.4
Houston Astros	49.6	47.5	55.9	58.7	67.8	74.7	73.8	89.0
Kansas City Royals	22.8	20.9	30.7	40.7	39.0	39.7	34.1	40.8
Los Angeles Dodgers	70.8	81.6	93.9	91.2	101.8	86.2	67.5	91.8
Anaheim/Los Angeles Angels	39.3	42.9	37.6	55.1	73.2	93.6	81.9	103.6
Milwaukee Brewers	30.3	28.5	39.9	43.4	36.0	27.5	40.2	56.8
Minnesota Twins	18.5	15.9	22.5	38.7	53.5	51.5	52.4	61.4
Montreal/Washington Nationals	15.0	30.0	29.0	34.5	50.0	36.0	40.5	52.7
New York Mets	57.8	79.5	83.2	91.0	100.7	96.8	97.0	97.0
New York Yankees	75.9	79.8	88.5	108.6	133.7	157.6	199.0	177.4
Oakland Athletics	22.3	29.6	31.3	36.7	48.4	55.4	53.7	62.3
Philadelphia Phillies	26.1	40.8	40.1	51.7	61.0	86.3	91.7	81.7
Pittsburgh Pirates	18.5	27.8	42.5	36.5	48.7	29.8	34.0	41.8
San Diego Padres	42.7	45.7	35.5	35.7	37.9	54.6	56.2	62.3
Seattle Mariners	48.0	56.6	67.5	80.3	80.7	72.8	67.1	84.9
San Francisco Giants	44.9	51.7	58.6	72.5	79.2	66.1	86.0	99.9
St. Louis Cardinals	42.3	56.9	66.6	71.2	67.1	75.6	89.7	85.0
Tampa Bay Rays	29.3	50.6	50.9	30.7	19.6	27.3	26.7	31.6
Texas Rangers	72.0	68.1	71.4	90.8	87.1	47.3	46.1	52.8
Toronto Blue Jays	42.8	44.5	67.7	66.3	47.5	48.1	43.6	66.6
Average Salary	43.3	49.9	56.2	59.6	63.9	62.1	66.4	72.1

Which of these diagrams is easier to read?



And what about these basic scatterplots? Which is more effective and elegant?



Final thoughts

To get a sense of what can be done with data visualization – and just how enthusiastic data scientists can get about data viz – enjoy this video by the Swedish epidemiologist **Hans Rosling**, the pioneer of **interactive data visualization** ([link here](#)).

As you go down the rabbit hole of data visualization, it will be important to become familiar with the work of **Edward Tufte**, the grandfather of thinking about data viz as an art form. This video showcases Tufte and other data scientists who have been inspired by his work ([link here](#)).

Finally, this video offers a nice and concise summary of Edward Tufte's principles of data visualization ([link here](#)).



(PART) Getting started

Chapter 6

Setting up RStudio

First, let's get the right programs installed on your computer. Then we will explain what they are and why you need them.

First, download and install R:

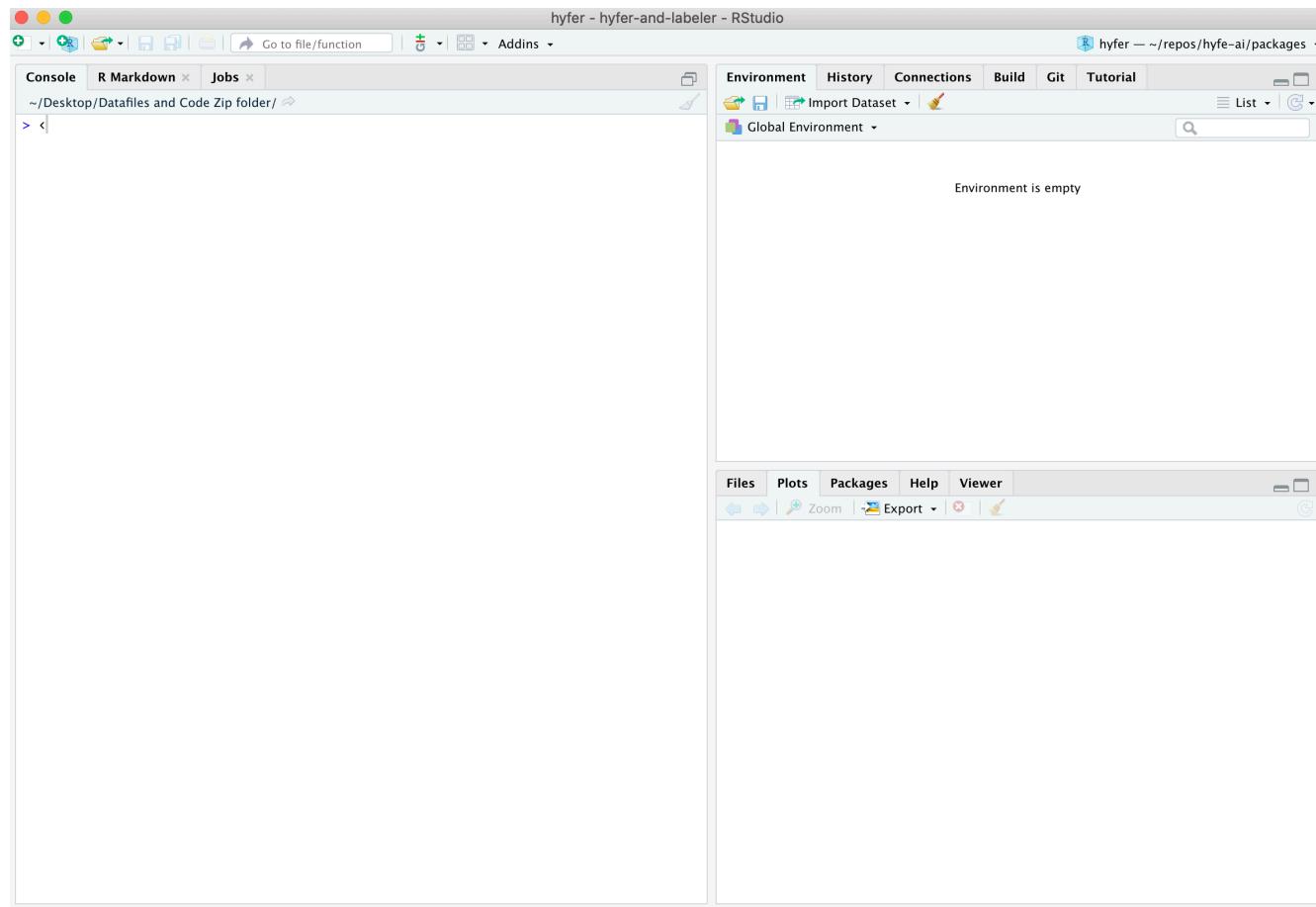
Go to the following website, click the *Download* button, and follow the website's instructions from there. <https://mirrors.nics.utk.edu/cran/>

Second, download and install RStudio:

Go to the following website and choose the free Desktop version: <https://rstudio.com/products/rstudio/download/>

Third, make sure RStudio opens successfully:

Open the RStudio app. A window should appear that looks like this:

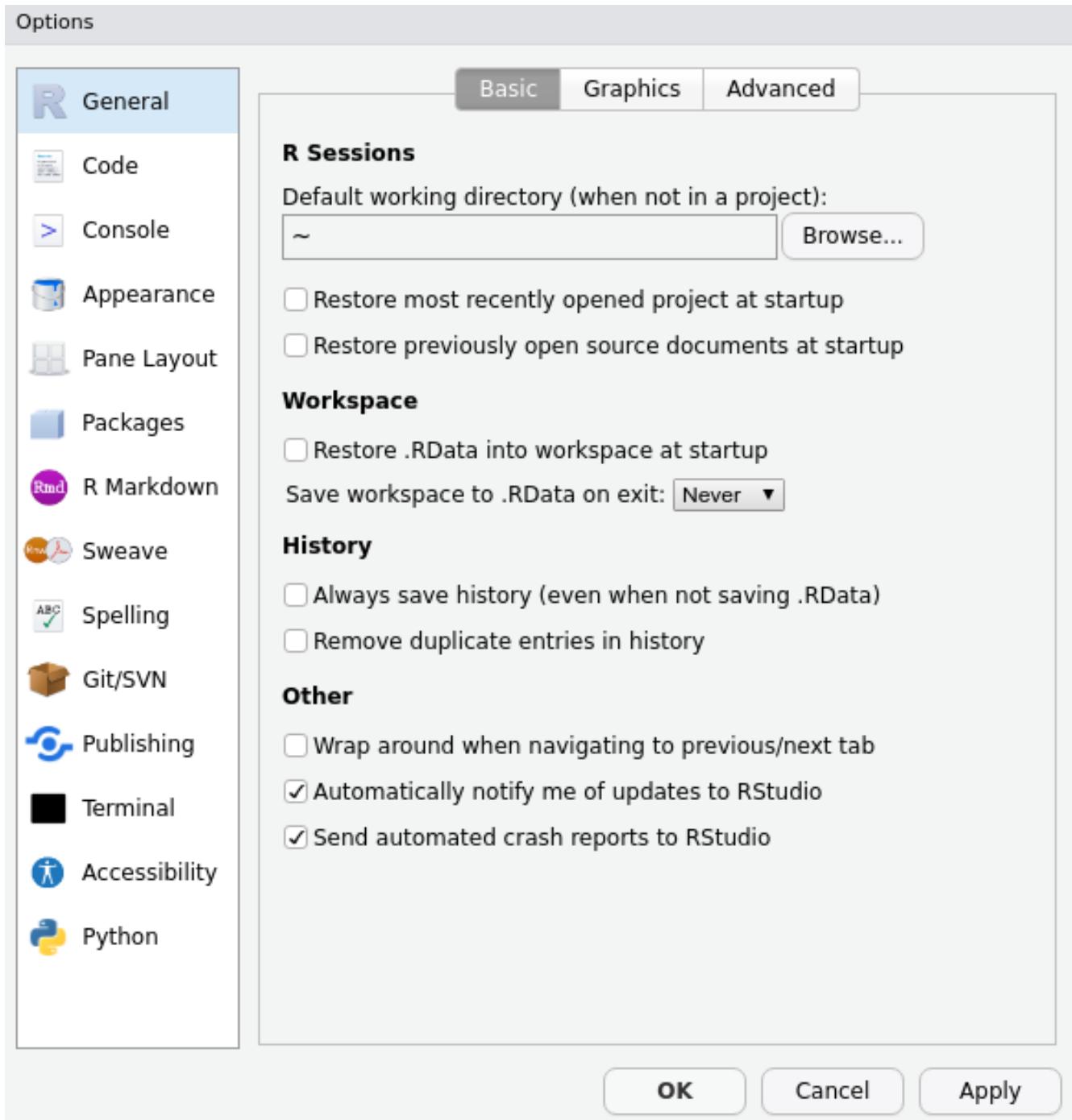


Fourth, make sure R is running correctly in the background:

In RStudio, in the pane on the left (the “Console”), type `2+2` and hit Enter.
If R is working properly, the number “4” will be printed in the next line down.

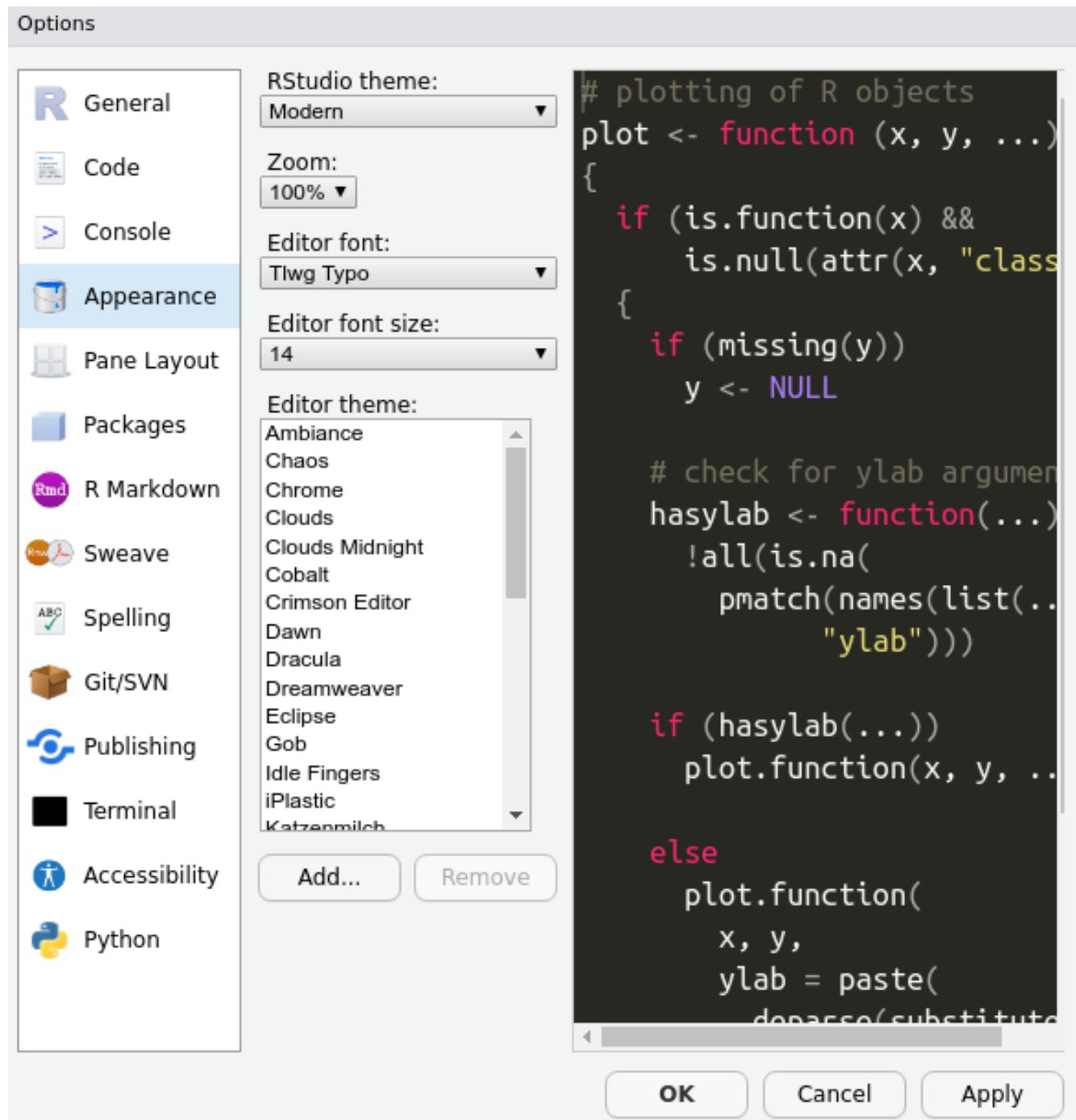
Finally, some minor adjustments to make RStudio run smoother (and look cooler):

Go to Tools > Global Options and make sure your General settings match these exactly:



Specifically, **uncheck** the option under *Workspace* to ‘Restore .RData into workspace at startup.’

Now go to the **Appearance** settings and choose a cool theme!



Boom!

Appendix, install supplementary software:

In order to install certain packages, you'll need to take one more step:

- On Windows, download <http://cran.r-project.org/bin/windows/Rtools/> and run the installer
- On Mac, you need Xcode Command Line Tools. You might already have this. Check by running `devtools::has_devel()`. If you don't have it, open shell/terminal and run

```
xcode-select --install
```

- Alternatively, on Mac, you can download Xcode from the Mac App Store directly: <https://apps.apple.com/ca/app/xcode/id497799835?mt=12>

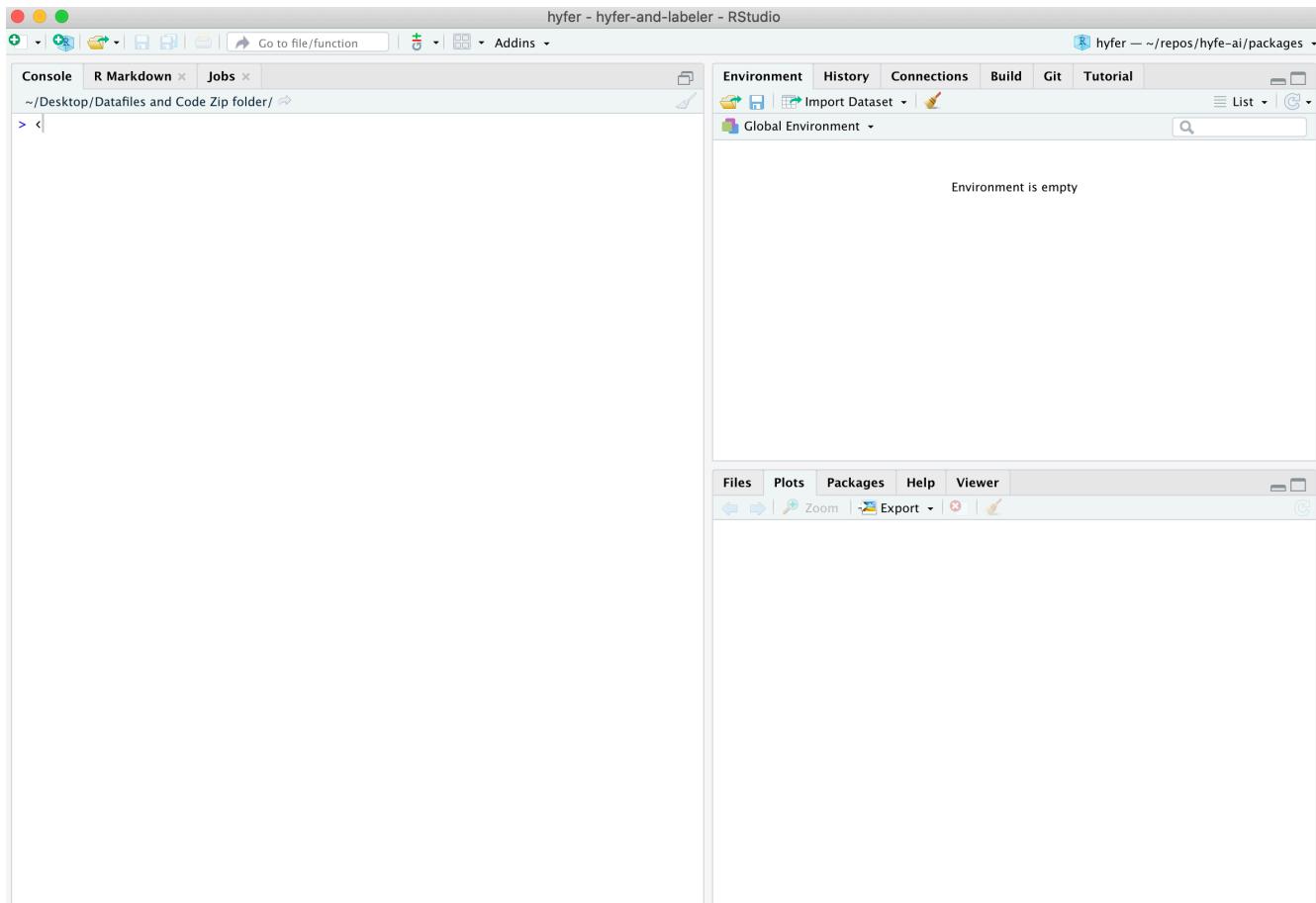
Chapter 7

Running R code

Learning goals

- Learn how to run code in R
- Learn how to use R as a calculator
- Learn how to use mathematical and logical operators in R

When you open RStudio, you see several different panes within the program's window. You will get a tour of RStudio in the next module. For now, look at the left half of the screen. You should see a large pane entitled the *Console*.



RStudio's *Console* is your window into R, the engine under the hood. The *Console* is where you type commands for

R to run, and where R prints back the results of what you have told it to do. Think of the *Console* as a chatroom, where you and R talk back and forth.

Running code in the *Console*

Type your first command into the *Console*, then press Enter:

```
1 + 1
[1] 2
```

When you press Enter, you send your line of code to R; you post it for R to see. Then R takes it, does some processing, and posts a result (2) just below your command.

Note that spaces don't matter. Both of the following two commands are legible to R and return the same thing:

```
4+4
[1] 8
```

```
4      +      4
[1] 8
```

However, it is better to make your code as easy to read as possible, which usually means using a single space between numbers:

```
4 + 4
[1] 8
```

Try typing in other basic calculations:

Use R like a calculator

As you can tell from those commands you just ran, R is, at heart, a fancy calculator.

Some calculations are straightforward, like addition and subtraction:

```
490 + 1000
[1] 1490
```

```
490 - 1000
[1] -510
```

Division is pretty straightforward too:

```
24 / 2
[1] 12
```

For multiplication, use an asterisk (*):

```
24 * 2
[1] 48
```

You denote exponents like this:

```
2 ^2
[1] 4
```

```
2 ^3
[1] 8
```

```
2 ^4
[1] 16
```

Finally, note that R is fine with negative numbers:

```
9 + -100
[1] -91
```

Getting along with R

Re-running code in the *Console*

If you want to re-run the code you just ran, or if you want to recall the code so that you can adjust it slightly, click anywhere in the *Console* then press your keyboard's Up arrow.

If you keep pressing your Up arrow, R will present you with sequentially older commands. R keeps a history of everything you have said to it since you opened this window.

If you accidentally recalled an old command without meaning to, you can reset the *Console*'s command line by pressing **Escape**.

Incomplete commands

R gets confused when you enter an incomplete command, and will wait for you to write the remainder of your command on the next line in the *Console* before doing anything.

For example, try running this code in your *Console*:

```
45 +
```

You will find that R gives you a little + sign on the line under your command, which means it is waiting for you to complete your command.

If you want to complete your command, add a number (e.g., 3) and hit **Enter**. You should now be given an answer (e.g., 48).

Or, if instead you want R to stop waiting and stop running, just press the **Escape** key.

Semicolons

Semicolons can be used to put two separate commands on the same line of code. For example, these two lines of commands ...

```
4 + 5
[1] 9
6 + 10
[1] 16
```

.. will return the same results as this single line of commands:

```
4 + 5 ; 6 + 10
[1] 9
[1] 16
```

This will become a useful trick in a few modules downstream.

Getting errors

R only understands your commands if they follow the rules of the R language (often referred to as its *syntax*). If R does not understand your code, it will throw an error and give up on trying to execute that line of code.

For example, try running this code in your *Console*:

```
4 + y
```

You probably received a message in red font stating: **Error: object 'y' not found**. That is because R did know how to interpret the symbol y in this case, so it just gave up.

Get used to errors! They happen all the time, even (especially?) to professionals, and it is essential that you get used to reading your own code to find and fix its errors.

Here's another piece of code that will produce an error:

```
dfjkltr9fitwt985ut9e3
```

Using parentheses

R is usually great about following classic rules for Order of Operations, and you can use parentheses to exert control over that order. For example, these two commands produce different results:

```
2*7 - 2*5 / 2
[1] 9
```

```
(2*7 - 2*5) / 2
[1] 2
```

Note that parentheses need to come in pairs: whenever you type an open parenthesis, (, eventually you need to provide a corresponding closed parenthesis,).

The following line of code will return a plus sign, +, since R is waiting for you to close the parenthetical before it processes your command:

```
4 + (5
```

Remember: **parentheses come in pairs!** The same goes for other types of brackets: {} and [...].

Using operators in R

You can ask R basic questions using *operators*.

For example, you can ask whether two values are equal to each other.

```
96 == 95
[1] FALSE
```

```
95 + 2 == 95 + 2
[1] TRUE
```

R is telling you that the first statement is FALSE (96 is not, in fact, equal to 95) and that the second statement is TRUE (95 + 2 is, in fact, equal to itself).

Note the use of double equal signs here. You must use two of them in order for R to understand that you are asking for this logical test.

You can also ask if two values are *NOT* equal to each other:

```
96 != 95
[1] TRUE
```

```
95 + 2 != 95 + 2
[1] FALSE
```

This test is a bit more difficult to understand: In the first statement, R is telling you that it is TRUE that 96 is different from 95. In the second statement, R is saying that it is FALSE that 95 + 2 is not the same as itself.

Note that R lets you write these tests another, even more confusing way:

```
! 96 == 95
[1] TRUE
```

```
! 95 + 2 == 95 + 2
[1] FALSE
```

The first line of code is asking R whether it is not true that 96 and 95 are equal to each other, which is TRUE. The second line of code is asking R whether it is not true that 95 + 2 is the same as itself, which is of course FALSE.

Other commonly used operators in R include greater than / less than symbols ($>$ and $<$, also known as the *left-facing alligator* and *right-facing alligator*), and greater/less than or equal to (\geq and \leq).

```
100 > 100
[1] FALSE
```

```
100 >= 100
[1] TRUE
```

```
(100 != 100) == FALSE
[1] TRUE
```

Use built-in R functions

R has some built-in “functions” for common calculations, such as finding square roots and logarithms. Functions are packages of code that take a given value, transform it according to some internal code instructions, and provide an output. You will learn more about functions in a few modules.

To find the square-root of a number, use the ‘squirt’ command, `sqrt()`:

```
sqrt(16)
[1] 4
```

Note the use of parentheses here. When you are calling a function, when you see parentheses, think of the word ‘of’. You are taking the `sqrt` of the number inside the parenthetical.

To get the log of a value:

```
log(4)
[1] 1.386294
```

Note that the function `log()` is the *natural log* function (i.e., the value that e must be raised to in order to equal 4). To calculate a base-10 logarithm, use `log10()`.

```
log(10)
[1] 2.302585
```

```
log10(10)
[1] 1
```

Another handy function is `round()`, for rounding numbers to a specific number of decimal places.

```
100/3
[1] 33.33333
```

```
round(100/3)
[1] 33
```

```
round(100/3,digits=1)
[1] 33.3
```

```
round(100/3,digits=2)
[1] 33.33
```

```
round(100/3,digits=3)
[1] 33.333
```

Finally, R also comes with some built-in *values*, such as `pi`:

```
pi
[1] 3.141593
```

Exercises

Use R like a calculator

1. Type a command in the *Console* to determine the sum of 596 and 198.
2. Re-run the sum of 596 and 198 without re-typing it.
3. Recall the command again, but this time adjust the code to find the sum of 596 and 298.
4. Practice escaping an accidentally called command: recall your most recent command, then press the right key to clear the *Console*'s command line.

Recalling commands

5. Find the sum of the ages of everyone in your immediate family.
6. Now recall that command and adjust it to determine the *average* age of the members of your family.
7. Find the square root of *pi* and round the answer to the 2 decimal places.

Finding errors

8. This line of code won't run; instead, R will wait for more with a + symbol. Find the problem and re-write the code so that it works.

```
5 * 6 +
```

9. The same goes for this line of code. Fix it, too.

```
sqrt(16
```

10. This line of code will trigger an error. Find the problem and re-write the code so that it works.

```
round(100/3,digits+3)
```

11. Type a command of your own into R that throws an error, then recall the command and revise so that R can understand it.

Show that the following statements are TRUE:

12. *pi* is greater than the square root of 9
13. It is FALSE that the square root of 9 is greater than *pi*
14. *pi* rounded to the nearest whole number equals the square root of 9

Asking TRUE / FALSE questions

15. Write and run a line of code that asks whether these two calculations return the same result:

```
2*7 - 2*5 / 2
```

```
(2*7 - 2*5) / 2  
[1] 2
```

16. Now write and run a line of code that asks whether the first calculation is larger than the second:

Other Resources

Hobbes Primer, Table 1 (Math Operators, pg. 18) and Table 2 (Logical operators, pg. 22)

Chapter 8

Using RStudio & R scripts

Learning goals

- Understand the difference between `R` and `RStudio`
- Understand the `RStudio` working environment and window panes
- Understand what `R` scripts are, and how to create and save them
- Understand how to add comments to your code, and why doing so is important
- Understand what a *working directory* is, and how to use it
- Learn basic project work flow

R and RStudio: what's the difference?

These two entities are similar, but it is important to understand how they are different.

In short, `R` is an open-source (i.e., free) coding language: a powerful programming engine that can be used to do really cool things with data.

`R Studio`, in contrast, is a free *user interface* that helps you interact with `R`. If you think of `R` as an engine, then it helps to think of `RStudio` as the car that contains it. Like a car, `RStudio` makes it easier and more comfortable to use the engine to get where you want to go.

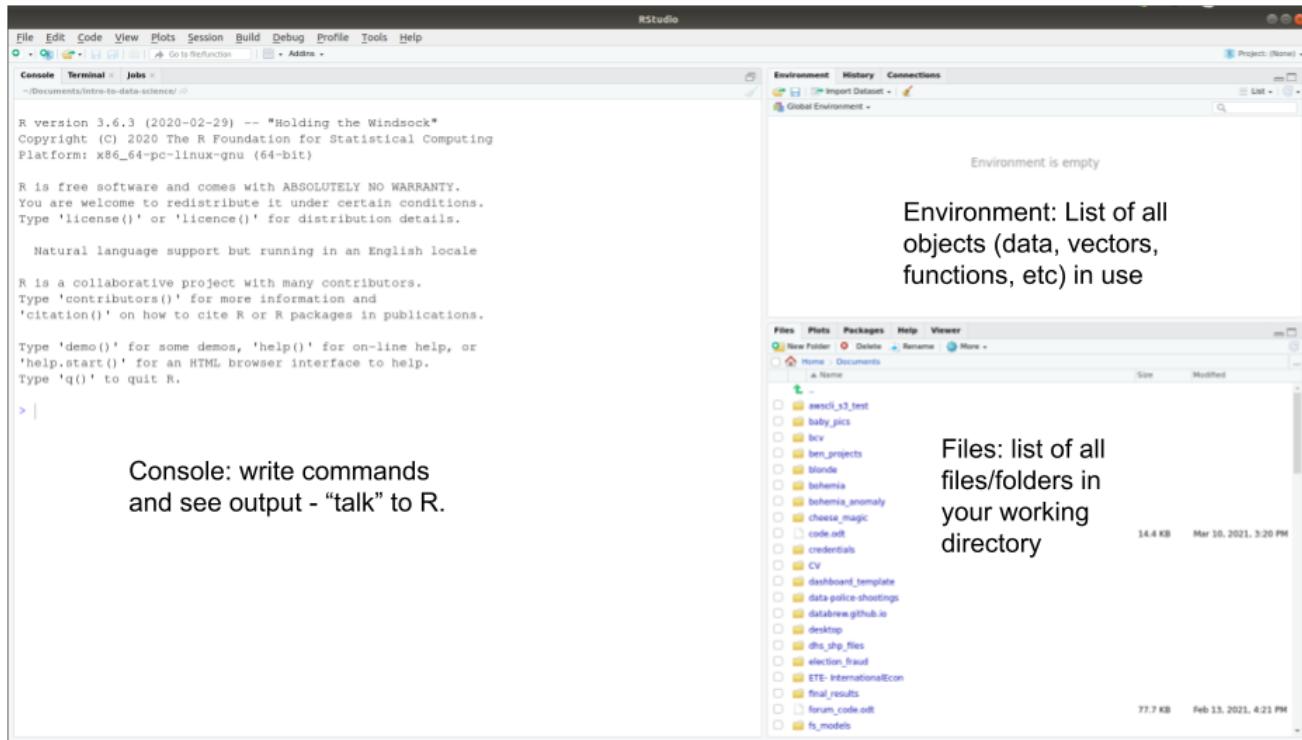
`R Studio` needs `R` in order to function, but `R` can technically be used on its own outside of `RStudio` if you want. However, just as a good car mechanic can get an engine to run without being installed within a car, using `R` on its own is a bit clunky and requires some expertise. For beginners (and everyone else, really), `R` is just so much more pleasant to use when you are operating it from within `RStudio`.

`RStudio` also has increasingly powerful *extensions* that make `R` even more useful and versatile in data science. These extensions allow you to use `R` to make interactive data dashboards, beautiful and reproducible data reports, presentations, websites, and even books. And new features like these are regularly being added to `RStudio` by its all-star team of data scientists.

That is why this book *always* uses `RStudio` when working with `R`.

Two-minute tour of RStudio

When you open `RStudio` for the first time, you will see a window that looks like the screenshot below.



Console

You are already acquainted with RStudio's *Console*, the window pane on the left that you use to "talk" to R. (See the previous module.)

Environment

In the top right pane, the *Environment*, RStudio will maintain a list of all the datasets, variables, and functions that you are using as you work. The next modules will explain what variables and functions are.

Files, Plots, Packages, & Help

You will use the bottom right pane very often.

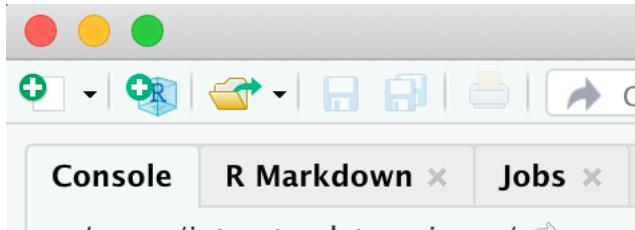
- The **Files** tab lets you see all the files within your **working directory**, which will be explained in the section below.
- The **Plots** tab lets you see the plots you are producing with your code.
- The **Packages** tab lets you see the *packages* you currently have installed on your computer. Packages are bundles of R functions downloaded from the internet; they will be explained in detail a few modules down the road.
- The **Help** tab is very important! It lets you see *documentation* (i.e., user's guides) for the functions you use in your code. Functions will also be explained in detail a few modules down the road.

These three panes are useful, but the most useful window pane of all is actually *missing* when you first open RStudio. This important pane is where you work with **scripts**.

Scripts

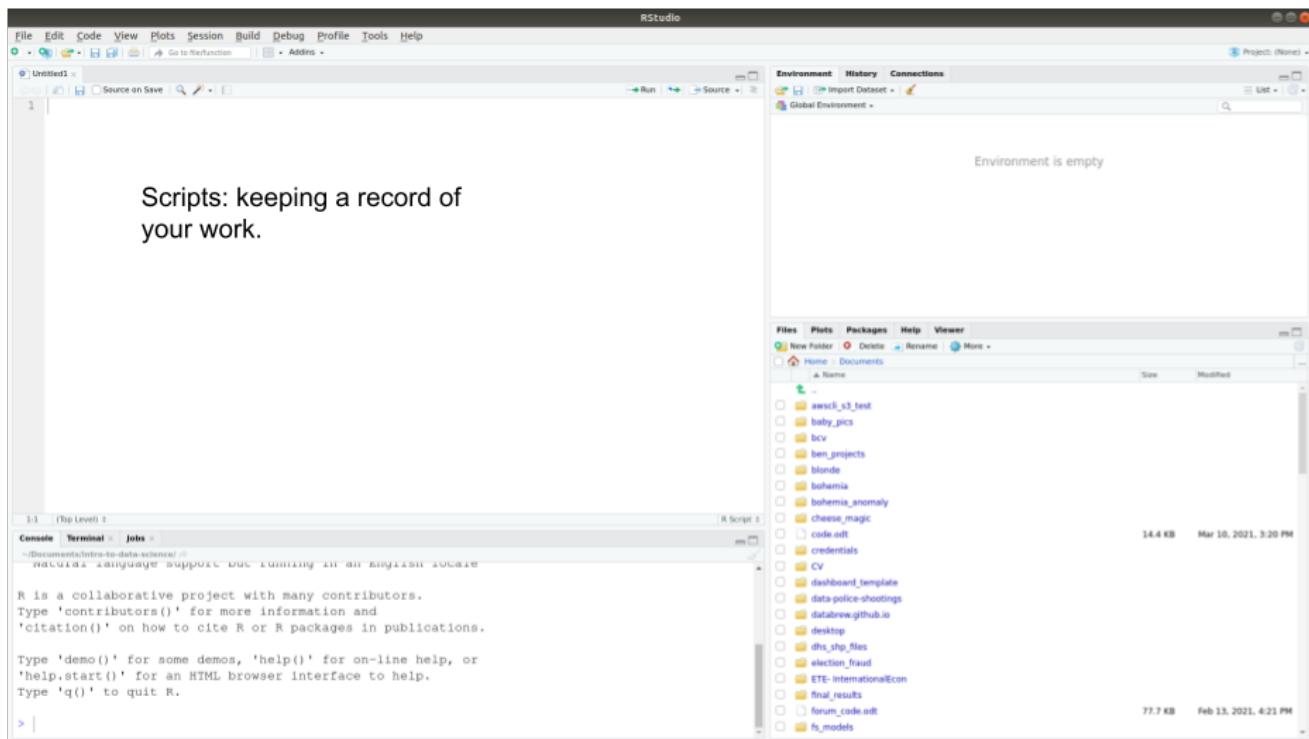
Before explaining what scripts are and why they are awesome, let's start a new script.

To start a new script, go to the top left icon in the RStudio window, and click on the green plus sign with a blank page behind it:



A dropdown window will appear. Select “R Script”.

A new window pane will then appear in the top left quadrant of your RStudio window:



You now have a blank script to work in!

Now type some simple commands into your script:

```
2 + 10
16 * 32
```

Notice that when you press **Enter** after each line of code, nothing happens in the *Console*. In order to send this code to the Console, press **Enter + Command** at the same time (or **Enter + Control**, if you are on Windows) for each line of code.

To send both lines of code to the *Console* at once, select both lines of code and hit **Enter + Command**.

(To select multiple lines of code, you can (1) click and drag with your mouse or (2) hold down your **Shift** key while clicking your down arrow key. To select *all* lines of code, press **Command + A**.)

So let's build up this script. Add a few more lines to your script, such that your script now looks like this.

```
2 + 10
16 * 32
1080 / 360
500 - 600
```

Run all of these lines of code at once.

Now add 10 to the first number in each row, and re-run all of the code.

Think about how much more efficient part (B) was thanks to your script! If you had typed all of that directly into your *Console*, you would have to recall or retype each line individually. That difference builds up when your number of commands grows into the hundreds.

What is an R script, and why are scripts so awesome?

An R script is a file where you can keep a record of your code. Just as a script tells actors exactly what to say and when to say it, an R script tells R exactly what code to run, and in what order to run it.

When working with R, you will almost always type your code into a script first, *then* send it to the *Console*. You can run your code immediately using **Enter + Command**, but you also have a script of what you have done so that you can run the exact same code at a later time

To understand why R scripts are so awesome, consider a typical workflow in *Excel* or *GoogleSheets*. You open a big complicated spreadsheet, spend hours making changes, and save your changes frequently throughout your work session.

The main disadvantages of this workflow are that:

1. There is no detailed record of the changes you have made. You cannot prove that you have made changes correctly. You cannot pass the original dataset to someone else and ask them to revise it in the same way you have. (Nor would you want to, since making all those changes was so time-consuming!) Nor could you take a different dataset and guarantee that you are able to apply the exact same changes that you applied to the first. In other words, your work is not reproducible.
2. Making those changes is labor-intensive! Rather than spend time manually making changes to a single spreadsheet, it would be better to devote that energy to writing R code that makes those changes for you. That code could be run in this one case, but it could also be run at any later time, or easily modified to make similar changes to other spreadsheets.
3. Unless you are an advanced *Excel* programmer, you are probably modifying your original dataset, which is always dangerous and a big No-No in data science. Each time you save your work in *Excel* or *GoogleSheets* (which automatically saves each change you make), the original spreadsheet file gets replaced by the updated version. But if you brought your dataset into R instead, and modified it using an R script, then you leave the raw data alone and keep it safe. (Sure, you can always save different versions of your Excel file, but then you run the risk of mixing up versions and getting confused.)

Working with R scripts allows you to avoid all of these pitfalls. When you write an R script, you are making your work

- **Efficient.** Once you get comfortable writing R code, you will be able to write scripts in a few minutes. Those scripts can modify datasets within seconds (or less) in ways that would take hours (or years) to carry out manually in *Excel* or *GoogleSheets*.
- **Reproducible.** Once you have written an R script, you can reproduce your own work whenever you want to. You can send your script to a colleague so that they can reproduce your work as well. Reproducible work is defensible work.
- **Low-risk.** Since your R script does not make any changes to the original data, you are keeping your data safe. It is *essential* to preserve the sanctity of raw data!

Note that there is nothing fancy or special about an R script. An R script is a simple text file; that is, it only accepts basic text; you can't add images or change font style or font size in an R script; just letters, numbers, and your other keyboard keys. The file's extension, .R tells your computer to interpret that text as R code.

Commenting your code

Another advantage of scripts is that you can include *comments* throughout your code to explain what you are doing and why. A *comment* is just a part of your script that is useful to you but that is ignored by R.

To add comments to your code, use the hashtag symbol (#). Any text following a # will be ignored by R.

Here is the script above, now with comments added:

```
# Define variable x
x <- 2
x

# Make a new variable, y, based on x
y <- x*56

z <- y / 23 # Make a third variable, z, based on y

x + y + z # Now get the sum of all three variables
```

Adding comments can be more work, but in the end it saves you time and makes your code more effective. Comments might not seem necessary in the moment, but it is amazing how helpful they are when you come back to your code the next day. Frequent and helpful comments make the difference between good and great code. Comment early, comment often!

You can also use lines of hashtags to visually organize your code. For example:

```
#####
# Setup
#####

# Define variable x
x <- 2
x

# Make a new variable, y, based on x
y <- x*56

z <- y / 23 # Make a third variable, z, based on y

#####
# Get result
#####

x + y + z # Now get the sum of all three variables
```

This might not seem necessary with a 5-line script, but adding visual breaks to your code becomes immensely helpful when your code grows to be hundreds of lines long.

Saving your work

R scripts are only useful if you save them! Unlike working with *GoogleDocs* or *GoogleSheets*, R will not automatically save your changes; you have to do that yourself. (This is inconvenient, but it is also safer; most of coding is trial and error, and sometimes you want to be careful about what is saved.)

Step 1: Decide where to save your work. The folder in which you save your R script will be referred to as your *working directory* (see the next section). For the sake of these tutorials, it will be most convenient to save all of your scripts in a single folder that is in an easily accessed location.

Step 2: In that location, make a new folder named `datalab`: We suggest making a new folder on your Desktop and naming it `datalab`, but you can name it whatever you want and place it wherever you want.

Step 3: Save your script in that folder To save the script you have opened and typed a few lines of code into, press `Command + S` (or `Control + S`). Alternatively, go to File > Save. Navigate to the folder you just created and

type in a file name that is simple but descriptive. We suggest making a new R script for each module, and naming those scripts according to each module's name. In this case, we recommend naming your script `intro_to_scripts`.

(It is good practice to avoid spaces in your file names; it will be essential later on, so good to begin the correct habit now. Start using an underscore, `_`, instead of a space.)

Your working directory

When you work with data in R, R will need to know where in your computer to look for that data. The folder it looks in is known as your **working directory**.

To find out which folder R is currently using as your working directory, use the function `getwd()`:

```
getwd()
[1] "/Users/mattrudd/datatrain/rbootcamp"
```

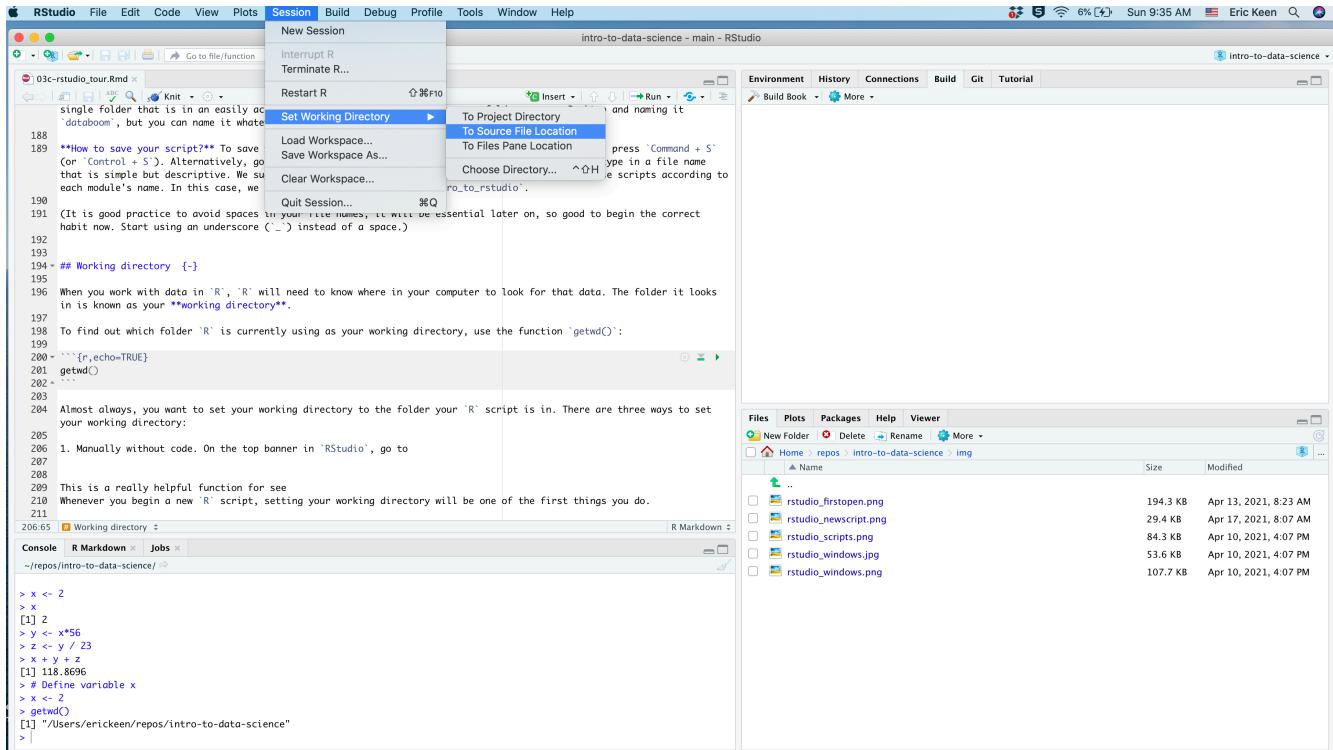
Almost always, you want to set your working directory to the folder your R script is in.

How to set your working directory

Whenever you begin a new R script, setting your working directory will be one of the first things you do.

There are three ways to set your working directory:

1. **Manually without code.** On the top banner in RStudio, go to *Session > Set Working Directory > To Source File Location*:



This action sets your working directory to the same folder that your R script is in. When you do this, you will see that a command has been entered into your *Console*:

```
setwd("~/Desktop/datalab")
```

(Note that the filepath may be different on your machine.) This code is using the function `setwd()`, which is also used in the next option. Go ahead and copy this `setwd(...)` code and paste it into your script, so it will be easy to use next time.

2. **Manually with code, using `setwd()`:** You can manually provide the filepath you want to set as your working directory. This option allows you to set your `wd` to whatever folder you want. The character string within the `setwd()` command is the path to a folder. The formatting of this string must be exact, otherwise R will throw an error. Use option 1 at first to get a sense of how your computer formats its folder paths. Copy, paste, and modify the output from option 1 in order to type your path correctly.
3. **Automatically with code:** There is a command you can run that automatically sets your working directory to the folder that your R script is in. This is the most efficient and useful method, in our experience.

To use this command, you must first install a new package. Run this code:

```
install.packages("rstudioapi")
library(rstudioapi)
```

For now, you do not need to understand what this code is doing. We will explain packages and the `library()` function in a later module.

You can now copy, paste, and run this code to set your working directory automatically:

```
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
```

This is a complicated line of code that you need not understand. As long as it works, it works! Confirm that R is using the correct working directory with the command `getwd()`.

Typical workflows

Now that you know how to create a script and set your working directory, you are prepared to work on data projects in RStudio.

The workflow for beginning a new data project typically goes like this:

In your file explorer...

1. **Create a folder for your project** somewhere on your computer. This will become your working directory.
2. **Create subfolders** within your working directory, if you want. We recommend creating a `data` subfolder, for keeping data, and a `z` subfolder, for keeping miscellaneous documents. The goal is to keep your working directory visually simple and organized; ideally, the only files not within subfolders are your R scripts.
3. **Add data** to your working directory, if you have any.

In RStudio ...

4. **Create a new R script.**
5. **Save it** inside your intended working directory.
6. At the top of your script, use comments to **add a title, author info, and brief description**.
7. Add the code to **set your working directory**.
8. **Begin coding!**

Template R script

Here is a template you can use to copy and paste into each new script you create:

```
#####
# < Add title here >
#####
#
# < Add brief description here >
#
# < Author >
# Created on <add date here >
```

```
#  
#####  
# Set working directory  
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))  
  
#####  
#####  
# Code goes here  
  
#####  
#####  
# (end of file)
```

Exercises

1 (*if not already complete*). Create a working directory for this course. Call it whatever you like, but `datalab` could work great. Place it somewhere convenient on your computer, such as your Desktop.

2. Within this working directory, create three new folders: (1) a `data` folder, which is where you will store the data files we will be using in subsequent modules; (2) a `modules` folder, which is where you will keep the code you use to work on the material in these modules, and (3) a `project` folder, which is where you will keep all your work associated with your summer project.

3. Now follow the *Typical Workflow* instructions above to create a script. Save it within your `modules` folder. Name it `template.R`. Copy and paste the template R code provided above into this file, and save it. This is now a template that you can use to easily create new scripts for this course.

4. Now make a copy of `template.R` to stage a script that you can use in the next module. To do so, in RStudio go to the top banner and click `File > Save As`. Save this new script as `variables.R` (because the next module is called *Variables in R*).

5. Modify the code in `variables.R` so that you are prepared to begin the next module. Change the title, and look ahead to the next module to fill in a brief description. Don't forget to add your name as the author and specify today's date.

Boom!

Other Resources

A Gentle Introduction to R from the RStudio team

Chapter 9

Variables

Learning goals

- How to define variables and work with them in R
- Learn the various possible classes of data in R

Introducing variables

So far we have strictly been using R as a calculator, with commands such as:

```
3 + 5  
[1] 8
```

Of course, R can do much, much more than these basic computations. Your first step in uncovering the potential of R is learning how to use **variables**.

In R, a variable is a convenient way of referring to an underlying value. That value can be as simple as a single number (e.g., 6), or as complex as a spreadsheet that is many Gigabytes in size. It may be useful to think of a variable as a cup; just as cups make it easy to hold your coffee and carry it from the kitchen to the couch, variables make it easy to contain and work with data.

Declaring variables

To assign numbers or other types of data to a variable, you use the < and - characters to make the arrow symbol <-.

```
x <- 3+5
```

As the direction of the <- arrow suggests, this command stores the result of $3 + 5$ into the variable x.

Unlike before, you did not see 8 printed to the *Console*. That is because the result was stored into x.

Calling variables

If you wanted R to tell you what x is, just type the variable name into the *Console* and run that command:

```
x  
[1] 8
```

Want to create a variable but also see its value at the same time? Here's a handy trick: put your command in parentheses:

```
(x <- 3*12)  
[1] 36
```

When you do that, `x` gets assigned a value, then that value is printed to the console.

You can also update variables.

```
(x <- x * 3)
[1] 108
```

```
(x <- x * 3)
[1] 324
```

You can also add variables together.

```
x <- 8
y <- 4.5
x + y
[1] 12.5
```

Naming variables

Here are a few rules:

1. A variable name has to have at least one letter in it. These examples work:

2. A variable name has to be connected. No spaces! It is usually best to represent a space using a period (.) or an underscore (_). Note that periods and underscores can be used in variable names:

```
my.variable <- 5 # periods can be used
my_variable <- 5 # underscores can be used
```

However, hyphens *cannot* be used, since that symbol is used for subtraction.

3. Variables are case-sensitive. If you misspell a variable name, you will confuse R and get an error. For example, ask R to tell you the value of capital X. The error message will be `Error: object 'X' not found`, which means R looked in its memory for an object (i.e., a variable) named X and could not find one.

4. Variable names can be as complicated or as simple as you want.

5. Some names need to be avoided, since R uses them for special purposes. For example, `data` should be avoided, as should `mean`, since both are functions built-in to R and R is liable to interpret them as such instead of as a variable containing your data.

```
supercalifragilistic.expiālidocious <- 5
supercalifragilistic.expiālidocious # still works
[1] 5
```

So those are the basic rules, but naming variables well is a bit of an art. The trick is using names that are clear but are not so complicated that typing them is tedious or prone to errors.

Note that R uses a feature called ‘Tab complete’ to help you type variable names. Begin typing a variable name, such as `supercalifragilistic.expiālidocious` from the example above, but after the first few letters press the Tab key. R will then give you options for auto-completing your word. Press Tab again, or Enter, to accept the auto-complete. This is a handy way to avoid typos.

Types of data in R

So far we have been working exclusively with numeric data. But there are many different data types in R. We call these “types” of data **classes**:

- Decimal values like 4.5 are called **numeric** data.
- Natural numbers like 4 are called **integers**. Integers are also numerics.
- Boolean values (TRUE or FALSE) are called **logical** data.
- Text (or string) values are called **character** data.

In order to be combined, data have to be the same class.

R is able to compute the following commands ...

```
x <- 6
y <- 4
x + y
[1] 10
```

... but not these:

```
x <- 6
y <- "4"
x + y
```

That's because the quotation marks used in naming y causes R to interpret y as a **character** class.

To see how R is interpreting variables, you can use the **class()** function:

```
x <- 100
class(x)
[1] "numeric"

x <- "100"
class(x)
[1] "character"

x <- 100 == 101
class(x)
[1] "logical"
```

Another data type to be aware of is **factors**, but we will deal with them later.

Exercises

Finding the errors

1. This code will produce an error. Can you find the problem and fix it so that this code will work?

```
# Assign 5 to a variable
my_var < 5
```

2. Same for this one:

```
# Assign 5 to a variable
my_var == 5
```

3. Same for this one:

```
x <- 5
y <- 1
x + y
```

Your Bananas-to-ICS ratio

4. Estimate how many bananas you've eaten in your lifetime and store that value in a variable (choose whatever name you wish). (By the way, what is a good method for estimating this as accurately as you can?)

5. Now estimate how many ice cream sandwiches you've eaten in your lifetime and store that in a different variable.

6. Now use these variables to calculate your Banana-to-ICS ratio. Store your result in a third variable, then call that variable in the Console to see your ratio.

7. Who in the class has the highest ratio? Who has the lowest?

Creating boolean variables

8. Assign a FALSE statement of your choosing to a variable of whatever name you wish.
9. Confirm that the class of this variable is “logical.”
10. Confirm that the variable equals FALSE.

Converting Fahrenheit to Celsius:

11. Assign a variable `fahrenheit` the numerical value of 32.
12. Assign a variable `celsius` to equal the conversion from Fahrenheit to Celsius. Unless you’re a meteorology nerd, you may need to Google the equation for this conversion.
13. Print the value of `celsius` to the *Console*.
14. Now use this code to determine the *Celsius* equivalent of 212 degrees *Fahrenheit*.

Wrapping up

15. Now ensure that your entire script is properly commented, and make sure your script is saved in your `datalab` working directory before closing.

Chapter 10

Vectors

Learning goals

- Learn the various structures of data in R
- How to work with vectors in R

Data belong to different *classes*, as explained in the previous module, and they can be arranged into various **structures**.

So far we have been dealing only with variables that contain a single value, but the real value of R comes from assigning *entire sets* of data to a variable.

The simplest data structure in R is a **vector**. A vector is simply a set of values. A vector can contain only a single value, as we have been working with thus far, or it can contain many millions of values.

Declaring and using vectors

To build up a vector in R, use the function `c()`, which is short for “concatenate”.

```
x <- c(5,6,7,8)
x
[1] 5 6 7 8
```

Whenever you use the `c()` function, you are telling R: ‘Hey, get ready. I’m about to give you more than one value at once.’

You can use the `c()` function to concatenate two vectors together:

```
x <- c(5,6,7,8)
y <- c(9,10,11,12)
z <- c(x,y)
z
[1] 5 6 7 8 9 10 11 12
```

You can also use `c()` to add values to a vector:

```
x <- c(5,6,7,8)
x <- c(x,9)
x
[1] 5 6 7 8 9
```

You can also put vectors through logical tests:

```
x <- c(1,2,3,4,5)
4 == x
[1] FALSE FALSE FALSE TRUE FALSE
```

This command is asking R to tell you whether each element in `x` is equal to 4.

You can create vectors of any data class (i.e., data type).

```
x <- c("Ben", "Joe", "Eric")
x
[1] "Ben" "Joe" "Eric"

y <- c(TRUE, TRUE, FALSE)
y
[1] TRUE TRUE FALSE
```

Note that all values within a vector *must* be of the same class. You can't combine numerics and characters into the same vector. If you did, R would try to convert the numbers to characters. For example:

```
x <- 4
y <- "6"
z <- c(x,y)
z
[1] "4" "6"
```

Math with two vectors

When two vectors are of the same length, you can do arithmetic with them:

```
x <- c(5,6,7,8)
y <- c(9,10,11,12)
x + y
[1] 14 16 18 20

x - y
[1] -4 -4 -4 -4

x * y
[1] 45 60 77 96

x / y
[1] 0.5555556 0.6000000 0.6363636 0.6666667
```

What happens when two vectors are *not* the same length?

Well, it depends. If one vector is length 1 (i.e., a single number), then things usually work out well.

```
x <- 5
y <- c(1,2,3,4,5,6,7,8,10)
x + y
[1] 6 7 8 9 10 11 12 13 15
```

In this command, the single element of `x` gets added to each element of `y`.

Another example, which you already saw above:

```
a <- c(1,2,3,4,5)
b <- 4
a == b
[1] FALSE FALSE FALSE TRUE FALSE
```

In this command, the single element of `b` gets compared to each element of `a`.

However, when both vectors contain multiple values but are not the same length, **be warned**: wonky things can happen. This is because R will start recycling the shorter vector:

```
a <- c(1,2,3,4,5)
b <- c(3,4)
a + b
[1] 4 6 6 8 8
```

As this warning implies, this doesn't make much sense. The command will still run, but do not trust the result.

Functions for handling vectors

We are about to list a bunch of core functions for working with vectors. Think of this like a toolbag. Each tool has a specific purpose and limited value: you can't quite build a house with just a hammer. But when you learn how to use all of the tools in your tool bag *together*, you can build almost anything. But you have to know how to use each tool individually first.

`length()` tells you the number of elements in a vector:

```
x <- c(5,6)
length(x)
[1] 2
```

```
y <- c(9,10,11,12)
length(y)
[1] 4
```

The **colon symbol** : creates a vector with every integer occurring between a min and max:

```
x <- 1:10
x
[1] 1 2 3 4 5 6 7 8 9 10
```

`seq()` allows you to build a vector using evenly spaced *sequence* of values between a min and max:

```
seq(0,100,length=11)
[1] 0 10 20 30 40 50 60 70 80 90 100
```

In this command, you are telling R to give you a sequence of values from 0 to 100, and you want the length of that vector to be 11. R then figures out the spacing required between each value in order to make that happen.

Alternatively, you can prescribe the interval between values instead of the length:

```
seq(0,100,by=7)
[1] 0 7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

`rep()` allows you to repeat a single value a specified number of times:

```
rep("Hey!",times=5)
[1] "Hey!" "Hey!" "Hey!" "Hey!" "Hey!"
```

You can also use `rep()` to repeat each element of a vector a set number of times:

```
rep(c("Hey!","Wohoo!"),each=3)
[1] "Hey!" "Hey!" "Hey!" "Wohoo!" "Wohoo!" "Wohoo!"
```

`head()` and `tail()` can be used to retrieve the first 6 or last 6 elements in a vector, respectively.

```
x <- 1:1000
head(x)
[1] 1 2 3 4 5 6
tail(x)
[1] 995 996 997 998 999 1000
```

You can also adjust how many elements to return:

```
head(x,2)
[1] 1 2
tail(x,10)
[1] 991 992 993 994 995 996 997 998 999 1000
```

`sort()` allows you to order a vector from its smallest value to its largest:

```
x <- c(4,8,1,6,9,2,7,5,3)
sort(x)
[1] 1 2 3 4 5 6 7 8 9
```

`rev()` lets you reverse the order of elements within a vector:

```
x <- c(4,8,1,6,9,2,7,5,3)
rev(x)
[1] 3 5 7 2 9 6 1 8 4

rev(sort(x))
[1] 9 8 7 6 5 4 3 2 1
```

`min()` and `max()` lets you find the smallest and largest value in a vector.

```
min(x)
[1] 1

max(x)
[1] 9
```

`which()` allows you to ask, “For which elements of a vector is the following statement true?”

```
x <- 1:10
which(x==4)
[1] 4
```

If no values within the vector meet the condition, a vector of length zero will be returned:

```
x <- 1:10
which(x == 11)
integer(0)
```

`which.min()` and `which.max()` tells you which element is the smallest and largest in the vector, respectively:

```
which.min(x)
[1] 1

which.max(x)
[1] 10
```

`%in%` is a handy operator that allows you to ask whether a value occurs *within* a vector:

```
x <- 1:10
4 %in% x
[1] TRUE

11 %in% x
[1] FALSE
```

`is.na()` is a way of asking whether a vector contains missing, broken, or erroneous values. In R, such values are referred to using the phrase NA. When you see NA, think of R telling you, ‘Nah ah! Nope! Not Available!’

```
x <- c(3,5,7,NA,9,4)
is.na(x)
[1] FALSE FALSE FALSE TRUE FALSE FALSE
```

This function is stepping through each element in the vector `x` and telling you whether that element is `NA`.

Subsetting vectors

Since you will eventually be working with vectors that contain thousands of data points, it will be useful to have some tools for *subsetting* them – that is, looking at only a few select elements at a time.

You can subset a vector using square brackets `[]`. Whenever you use brackets, you are telling R: ‘Hey, I want some numbers, but *not everything*: just certain ones.’

```
x <- 50:100
x[10]
[1] 59
```

This command is asking R to return the 10th element in the vector `x`.

```
x[10:20]
[1] 59 60 61 62 63 64 65 66 67 68 69
```

This command is asking R to return elements 10:20 in the vector `x`.

Exercises

Creating sequences of numbers

1. Use the colon symbol to create a vector of length 5 between a minimum and a maximum value of your choosing.
2. Create a second vector of length 5 using the `seq()` function. Use code to confirm that the length of this vector is 5.
3. Create a third vector of length 5 using the `rep()` function. Use code to confirm that the length of this vector is 5.
4. Finally, concatenate the three vectors and check that the length equals 15.

Basic vector math

5. Create a variable `x` that is a list of numbers of any size. Create a variable `y` of the same length.
6. Check to see if each values of `x` is greater than each value of `y`.
7. Check to see if the smallest value of `x` is greater than or equal to the average value of `y`.

Vectors and object classes

8. Create a vector with at least one number, then a second vector with at least one character string, then a third vector with at least one logical value. Identify the class of all three vectors.
9. Now concatenate these three vectors into a fourth vector. Identify the class of this fourth vector.

Heads & tails

10. Create a vector with at least 15 values.
11. Show the first six values of that vector using the `head()` function.
12. Figure out how to show the same result without a function, but instead with your new vector subsetting skills. Now replicate the `tail()` function, using those same skills. You may need to call the `length()` function as well.

Shoe sizes

13. Create a vector called `shoes`, which contains the shoe sizes of five people sitting near you. Use comments to keep track of which size is whose.
14. Arrange this set of shoe sizes in ascending order.
15. Arrange this set of shoe sizes in descending order.
16. Use code to find the two largest shoe sizes in your vector. Don't use subsetting; instead, write a line of code that would work even if more shoes were added to your vector.
17. What is the shoe size closest to the mean of these shoe sizes?
18. Use the `which()` function to figure out which of your five neighbors this shoe size belongs to.

Swimming timelines

19. Now create a new vector called `swim_days`, which contains the number of days since those same five people last went swimming (in any body of water; estimating the days since is fine).
20. Use code to ask whether anyone went swimming less than five days ago.
21. Which of your neighbors, if any, went swimming in the last month?
22. Which of your neighbors, if any, have not been swimming the last month?
23. On average, how long has it been since these people have gone swimming?

Dealing with NAs

24. Create a vector named `x` with these values: `c(4, 7, 1, NA, 9, 2, 8)`.
25. Use a function to decide whether or not each element of `x` is `NA`.
26. Use another function to find out which element in `x` is `NA`.
27. Write code that will subset `x` only to those values that are `NA`.
28. Write code that will subset `x` only to those values that are *not* `NA`.

Sleep deficits

29. Now create a vector called `sleep_time` with the number of hours you slept for each day in the last week.
30. Check if you slept more on day 3 than day 7.
31. Get the total number of hours slept in the last week.
32. Get the average number of hours slept in the last week.
33. Check if the total number of hours in the first 3 days is less than the total number of hours in the last 4 days.
34. Now create an object named `over_under`. This should be the difference between how much you slept each night and 8 hours (ie, 1.5 means you slept 9.5 hours and -2 means you slept 8 hours).
35. Write code to use `over_under` to calculate your sleep deficit / surplus this week (ie, the total hours over/under the amount of sleep you would have gotten had you slept 8 hours every night).
36. Write code to get the minimum number of hours you slept this week.
37. Write code to calculate how many hours of sleep you would have gotten had you sleep the minimum number of hours every night.
38. Write code to calculate the average of the hours of sleep you got on the 3rd through 6th days of the week.
39. Write code to calculate how many hours of sleep you would get in a year if you were to sleep the same amount every night as the average amount you slept from the 3rd to 6th days of the week.

40. Write code to calculate how many hours of sleep per year someone who sleeps 8 hours a night gets.
41. How many hours more/less than the 8 hours per night sleeper do you get in a year, assuming you sleep every night the average of the amount you slept on the first and last day of this week?
42. What is your total sleep deficit for the last week?
43. How many more hours per night, on average, do you need to sleep for the rest of the month so that, by the end of the month, you have a sleep deficit of zero?

Chapter 11

Dataframes

Learning goal

- Practice exploring, summarizing, and filtering dataframes

A vector is the most basic data structure in R, and the other structures are built out of vectors. But, as a data scientist, the most common data structure you will be working with – by far – is a **dataframe**.

A dataframe, essentially, is a spreadsheet: a dataset with rows and columns, in which each column represents a vector of the same class of data.

Here is what a dataframe looks like:

```
# Using one of R's built-in datasets
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2   setosa
2          4.9         3.0          1.4         0.2   setosa
3          4.7         3.2          1.3         0.2   setosa
4          4.6         3.1          1.5         0.2   setosa
5          5.0         3.6          1.4         0.2   setosa
6          5.4         3.9          1.7         0.4   setosa
```

In this dataframe, each row pertains to a unique iris plant. The columns contain related information about each individual plant.

Here's another data.frame, built from scratch, which shows that dataframes are just a group of vectors:

```
x <- 25:29
y <- 55:59
df <- data.frame(x,y)
df
  x  y
1 25 55
2 26 56
3 27 57
4 28 58
5 29 59
```

In this command, we used the `data.frame()` function to combine two vectors into a dataframe with two columns named `x` and `y`. R then saved this result in a new variable named `df`. When we call `df`, R shows us the dataframe.

The great thing about dataframes is that they allow you to relate different data types to each other.

```
df <- data.frame(name=c("Ben","Joe","Eric"),
                  height=c(75,73,80))
df
  name height
1 Ben     75
2 Joe     73
3 Eric    80
```

This dataframe has one column of class `character` and another of class `numeric`.

Subsetting & exploring dataframes

To explore dataframes, let's use a dataset on fuel mileage for all cars sold from 1985 to 2014.

```
# need to install first install.packages('fueleconomy')
library(fueleconomy)
data(vehicles)
head(vehicles)

  id make      model year       class      trans
1 13309 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
2 13310 Acura 2.2CL/3.0CL 1997 Subcompact Cars   Manual 5-spd
3 13311 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
4 14038 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
5 14039 Acura 2.3CL/3.0CL 1998 Subcompact Cars   Manual 5-spd
6 14040 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd

  drive cyl displ   fuel hwy cty
1 Front-Wheel Drive 4  2.2 Regular 26  20
2 Front-Wheel Drive 4  2.2 Regular 28  22
3 Front-Wheel Drive 6  3.0 Regular 26  18
4 Front-Wheel Drive 4  2.3 Regular 27  19
5 Front-Wheel Drive 4  2.3 Regular 29  21
6 Front-Wheel Drive 6  3.0 Regular 26  17
```

To look at this dataframe in full, you can display it in a separate tab within RStudio using the `View()` function:

```
View(vehicles)
```

A dataframe has rows of data organized into columns. In this dataframe, each row pertains to a single vehicle make/model – i.e., a single *observation*. Each column pertains to a single *type* of data. Columns are named in the *header* of the dataframe.

All the same useful exploration and subsetting functions that applied to vectors now apply to dataframes. In addition to those functions you already know, let's add some new functions to your inventory of useful functions.

Exploration

`head()` and `tail()` summarize the beginning and end of the object:

```
head(vehicles)

  id make      model year       class      trans
1 13309 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
2 13310 Acura 2.2CL/3.0CL 1997 Subcompact Cars   Manual 5-spd
3 13311 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
4 14038 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
5 14039 Acura 2.3CL/3.0CL 1998 Subcompact Cars   Manual 5-spd
6 14040 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd

  drive cyl displ   fuel hwy cty
1 Front-Wheel Drive 4  2.2 Regular 26  20
```

```

2 Front-Wheel Drive 4 2.2 Regular 28 22
3 Front-Wheel Drive 6 3.0 Regular 26 18
4 Front-Wheel Drive 4 2.3 Regular 27 19
5 Front-Wheel Drive 4 2.3 Regular 29 21
6 Front-Wheel Drive 6 3.0 Regular 26 17

tail(vehicles)
  id make      model year   class trans
33437 28868 Yugo GV Plus/GV/Cabrio 1990 Minicompact Cars Manual 4-spd
33438 6635 Yugo GV Plus/GV/Cabrio 1990 Subcompact Cars Manual 5-spd
33439 3157 Yugo           GV/GVX 1987 Subcompact Cars Manual 4-spd
33440 5497 Yugo           GV/GVX 1989 Subcompact Cars Manual 4-spd
33441 5498 Yugo           GV/GVX 1989 Subcompact Cars Manual 5-spd
33442 1745 Yugo           Gy/yugo GVX 1986 Minicompact Cars Manual 4-spd
  drive cyl displ fuel hwy cty
33437 Front-Wheel Drive 4 1.3 Regular 27 21
33438 Front-Wheel Drive 4 1.3 Regular 28 23
33439 Front-Wheel Drive 4 1.1 Regular 29 24
33440 Front-Wheel Drive 4 1.1 Regular 29 24
33441 Front-Wheel Drive 4 1.3 Regular 28 23
33442 Front-Wheel Drive 4 1.1 Regular 29 22

```

`names()` tells you the column names:

```

names(vehicles)
[1] "id"      "make"    "model"   "year"    "class"   "trans"
[10] "drive"   "cyl"     "displ"   "fuel"    "hwy"    "cty"

```

`nrow()`, `ncol()`, and `dim()` tell you about the dimensions of your dataframe:

```

nrow(vehicles)
[1] 33442

```

```

ncol(vehicles)
[1] 12

```

```

dim(vehicles)
[1] 33442   12

```

Note that `length()` does not work the same on dataframes as it does with vectors. In dataframes, `length()` is the equivalent of `ncol()`; it will *not* give you the number of rows in a dataset.

Importantly, you can use `is.na()` to ask whether columns or rows contain NAs:

```

# Check for NAs

# Which rows in the `hwy` column have NA's?
which(is.na(vehicles$hwy))
integer(0)

# (No NAs in that column!)

# What about rows in the `cyl` column?
which(is.na(vehicles$cyl))
[1] 1232 1233 2347 3246 3247 3248 6115 6116 6533 7783 7784 8472
[13] 10613 10614 11696 11697 12411 12412 12413 12928 12929 12934 12935 12944
[25] 16429 16430 21070 23472 23473 23474 24485 24486 24487 24488 24489 26150
[37] 28628 28704 28705 28706 28707 28708 28709 29314 29315 30023 30024 30025
[49] 30026 30027 30028 31063 31064 31065 31066 31067 31068 31069

```

```
# (lots of NAs in that column!)
```

Subsetting

Recall that dataframes are filtered by row and/or column using this format: `dataframe[rows,columns]`. To get the third element of the second column, for example, you type `dataframe[3,2]`.

```
vehicles[3,2]
[1] "Acura"
```

Note that the comma is necessary even if you do not want to specify columns. If you try to type this ...

```
vehicles[3]
```

...R will assume you are asking for the third column, not the third row.

To filter a dataframe to multiple values, you can specify vectors for the `row` and `column`

```
vehicles[1:3,11:12] # can use colons
  hwy cty
1 26 20
2 28 22
3 26 18
vehicles[1:3,c(1,11:12)] # can use c()
  id hwy cty
1 13309 26 20
2 13310 28 22
3 13311 26 18
```

Columns can also be called according to their names. Use the `$` sign to specify a column.

```
vehicles$hwy[1:5]
[1] 26 28 26 27 29
```

Note that when you use a `$`, you will not need to use a comma within your brackets. If you try to run this ...

```
vehicles$hwy[1:5,]
```

...R will throw a fit.

Also recall that you can use logical tests, which return boolean values `TRUE` or `FALSE`, to filter dataframes to rows that meet certain conditions. For example, to filter to only the rows for cars with better than 100 mpg, you can use this syntax:

```
# Build your logical test
verdicts <- vehicles$hwy > 100

# Subset with booleans
vehicles[verdicts,2:3]
      make     model
6533 Chevrolet Spark EV
10613      Fiat    500e
10614      Fiat    500e
16429      Honda   Fit EV
16430      Honda   Fit EV
24487      Nissan   Leaf
24488      Nissan   Leaf
24489      Nissan   Leaf
28628      Scion    iQ EV
```

Or you can write all this in a single line, to be more efficient:

```
vehicles[ vehicles$hwy > 100 , 2:3]
      make   model
6533 Chevrolet Spark EV
10613    Fiat    500e
10614    Fiat    500e
16429    Honda   Fit EV
16430    Honda   Fit EV
24487    Nissan  Leaf
24488    Nissan  Leaf
24489    Nissan  Leaf
28628    Scion   iQ EV
```

Recall that the logical test is returning a bunch of TRUE's and FALSE's, one for each row of `vehicles`. Only the TRUE rows will be returned.

Summarizing

The same summary functions that you have used for vectors work for the columns in dataframes, since each column is also a vector. Check it out:

```
min(vehicles$hwy)
[1] 9

max(vehicles$hwy)
[1] 109

mean(vehicles$cty)
[1] 17.491

sd(vehicles$cty)
[1] 5.582174

str(vehicles$make)
chr [1:33442] "Acura" "Acura" "Acura" "Acura" "Acura" "Acura" "Acura" ...

class(vehicles$hwy)
[1] "numeric"
```

You can also use the `summary()` function, which provides summary statistics for each column in your dataframe:

```
summary(vehicles)
      id          make         model        year
Min. : 1  Length:33442  Length:33442  Min. :1984
1st Qu.: 8361  Class :character  Class :character  1st Qu.:1991
Median :16724  Mode  :character  Mode  :character  Median :1999
Mean   :17038                               Mean   :1999
3rd Qu.:25265                               3rd Qu.:2008
Max.   :34932                               Max.   :2015

      class        trans        drive        cyl
Length:33442  Length:33442  Length:33442  Min.   : 2.000
Class :character  Class :character  Class :character  1st Qu.: 4.000
Mode  :character  Mode  :character  Mode  :character  Median : 6.000
                                         Mean   : 5.772
                                         3rd Qu.: 6.000
                                         Max.   :16.000
                                         NA's   :58

      displ       fuel        hwy        cty
Min.   :0.000  Length:33442  Min.   : 9.00  Min.   : 6.00
```

```
1st Qu.:2.300  Class :character  1st Qu.: 19.00  1st Qu.: 15.00
Median :3.000  Mode  :character  Median : 23.00  Median : 17.00
Mean   :3.353            Mean   : 23.55  Mean   : 17.49
3rd Qu.:4.300            3rd Qu.: 27.00  3rd Qu.: 20.00
Max.   :8.400            Max.   :109.00  Max.   :138.00
NA's    :57
```

The function `unique()` returns unique values within a column:

```
unique(vehicles$fuel)
[1] "Regular"                  "Premium"
[3] "Diesel"                   "Premium or E85"
[5] "Electricity"              "Gasoline or E85"
[7] "Premium Gas or Electricity" "Gasoline or natural gas"
[9] "CNG"                      "Midgrade"
[11] "Regular Gas and Electricity" "Gasoline or propane"
[13] "Premium and Electricity"
```

Finally, the `order()` function helps you sort a dataframe according to the values in one of its columns.

```
# Sort dataframe by highway mileage
# Only keep certain columns
vehicles_sorted <- vehicles[order(vehicles$hwy),
                           c(2,3,4,10:12)]
head(vehicles_sorted)
      make           model year fuel hwy cty
397  Aston Martin       Lagonda 1985 Regular 9 7
398  Aston Martin       Lagonda 1985 Regular 9 7
406  Aston Martin Saloon/Vantage/Volante 1985 Regular 9 7
408  Aston Martin Saloon/Vantage/Volante 1985 Regular 9 7
27725 Rolls-Royce        Camargue 1987 Regular 9 7
27726 Rolls-Royce        Continental 1987 Regular 9 7
```

Reverse the order by wrapping `rev()` around the `order()` call:

```
vehicles_sorted <- vehicles[rev(order(vehicles$hwy)),
                           c(2,3,4,10:12)]
head(vehicles_sorted)
      make           model year fuel hwy cty
6533 Chevrolet Spark EV 2014 Electricity 109 128
10614     Fiat      500e 2014 Electricity 108 122
10613     Fiat      500e 2013 Electricity 108 122
28628    Scion      iQ EV 2013 Electricity 105 138
16430    Honda     Fit EV 2014 Electricity 105 132
16429    Honda     Fit EV 2013 Electricity 105 132
```

Creating dataframes

As shown above, to create a new dataframe, use the `data.frame()` function.

	car	mpg_hwy	mpg_city
100	Acura Legend	23	15
101	Acura Legend	22	17
102	Acura Legend	23	16
103	Acura Legend	21	16
104	Acura Legend	22	17
105	Acura Legend	23	16
106	Acura Legend	24	16

Note how the columns were named in the `data.frame()` call, and that each column is separated by a comma.

You can also stage an empty dataframe, which sounds useless but will become very useful as you start working with `for` loops and other higher-order R tools.

```
df <- data.frame()
df
data frame with 0 columns and 0 rows
```

To coerce an object into a format that R interprets as a dataframe, use `as.dataframe()`:

```
df <- as.data.frame(vehicles)
df[1:4,1:4]
  id make      model year
1 13309 Acura 2.2CL/3.0CL 1997
2 13310 Acura 2.2CL/3.0CL 1997
3 13311 Acura 2.2CL/3.0CL 1997
4 14038 Acura 2.3CL/3.0CL 1998
```

Modifying dataframes

Combining dataframes

To bind multiple dataframes together by row, use `rbind()`:

```
# Build up a dataframe
df1 <- data.frame(name=c("Ben","Joe","Eric","Isabelle"),
                   instrument=c("Nose harp","Concertina","Ukelele","Drums"))
df1
  name instrument
1 Ben   Nose harp
2 Joe   Concertina
3 Eric   Ukelele
4 Isabelle Drums

# Build up a second dataframe
df2 <- data.frame(name=c("Matthew"),
                   instrument=c("Washboard"))

# Combine those dataframes together
rbind(df1,df2)
  name instrument
1 Ben   Nose harp
2 Joe   Concertina
3 Eric   Ukelele
4 Isabelle Drums
5 Matthew Washboard
```

Note that to be combined, two dataframes have to have the exact same number of columns and the exact same column names.

The only exception to this is adding a dataframe with content an empty dataframe. That can work, and that will be helpful in the Deep R modules ahead.

```
df <- data.frame() # stage empty dataframe

df1 <- data.frame(name=c("Ben","Joe","Eric","Isabelle"),
                   instrument=c("Nose harp","Concertina","Ukelele","Drums"))

df <- rbind(df,df1)
```

```
df
  name instrument
1   Ben   Nose harp
2   Joe Concertina
3  Eric    Ukelele
4 Isabelle   Drums
```

You can also bind multiple dataframes together by column, using `cbind()`:

```
df1 <- data.frame(name=c("Ben", "Joe", "Eric", "Isabelle"),
                   instrument=c("Nose harp", "Concertina", "Ukelele", "Drums"))

df <- data.frame(age=c(33,35,35,20), home=c("Canada", "Spain", "USA", "USA"))

df <- cbind(df,df1)

df
  age   home      name instrument
1 33 Canada      Ben   Nose harp
2 35 Spain       Joe Concertina
3 35 USA        Eric    Ukelele
4 20 USA Isabelle   Drums
```

Note that to be combined, two dataframes have to have the exact same number of rows and the exact same column names.

Adding columns

To create a new column for a dataframe, use the `$` symbol and provide the name of the new column:

```
df$x_factor <- c(3,20,60,40)

df
  age   home      name instrument x_factor
1 33 Canada      Ben   Nose harp      3
2 35 Spain       Joe Concertina    20
3 35 USA        Eric    Ukelele     60
4 20 USA Isabelle   Drums        40
```

Altering values

To alter certain values in the dataframe, you can assign new values to a subset of your dataframe.

Here are four ways to do the same thing: updating Isabelle's X-factor:

Option 1: Subsetting a single column

```
df$x_factor[4] <- 70
```

Option 2: Subsetting both rows and columns

```
df[4,5] <- 70
```

Option 3: Subsetting a column based on a logical test

```
df$x_factor[df$name == 'Isabelle'] <- 70
```

Option 4: Subsetting row and columns using logical tests

```
df[df$name == 'Isabelle', names(df) == 'x_factor'] <- 70
```

```
df
  age   home      name instrument x_factor
1  33 Canada     Ben    Nose harp      3
2  35 Spain      Joe Concertina    20
3  35 USA        Eric   Ukelele     60
4  20 USA Isabelle Drums           70
```

Exercises

Reading for errors

What is wrong with these commands? Why will each of them throw an error if you run them, and how can you fix them?

1. vehicles[1,15,]
2. vecihles[1:5,]
3. vehicles\$hwy[15,]
4. vehicles[1:5,1:13]

Subsetting and filtering

5. **Subset one field according to a logical test:** With no more than two lines of code, get the number of Honda cars in the `vehicles` dataset.
6. **Subset one field according to a logical test for a different field:** In a single line of code, show the mileages of all the Toyotas in the dataset.
7. **Subset a dataframe to a single subgroup:** In a single line of code, determine how many different car makes/models were produced in 1995.
8. **Get the mean value for a subgroup of data:** What is the average city mileage for Subaru cars in the dataset?
9. **Subset a dataframe to only data from between two values:** According to this dataset, how many different car makes/models have been produced with highway mileages between 30 and 40 mpg?
10. **Subset by removing NAs:** Create a new version of the `vehicles` dataframe that does not have any NAs in the `trans` column.

Creating dataframes

11. Create a vector called `people` of 5 peoples names from the class.
12. Show with code how many people are in your vector
13. Create another vector called `height` which is the number of centimeters tall each of those 5 people are.
14. Combine these two vectors into a data frame.

Now let's create a new object named `animals`. This is going to be a dataframe with 4 different columns: `species`, `weight` (in kg), `color`, `veg` (whether or not the animal is a vegetarian / herbivore).

15. Come up with five species to add to your dataframe and list them in a vector named `species`.
16. Make the other vectors with details about those species in the correct order.
17. Combine these vectors into a dataframe named `animals`.

Altering dataframes

18. Add a column to your `animals` dataframe named `rank`, which ranks each animal from your least favorite (0) to your most favorite (5).
19. Now write code to manually switch the ranking for your top two favorite animals.
20. What is the mean weight of the herbivorous animals that you listed, if any?
21. What is the mean weight of the omnivorous/carnivorous animals that you listed?

Chapter 12

Packages

Learning goals

- Learn what R packages are and why they are awesome
- Learn how to find and read about the packages installed on your machine
- Learn how to install R packages from CRAN
- Learn how to install R packages from GitHub

As established in the **Calling functions** module, R comes with hundreds of built-in base functions and datasets ready for use. You can also write your *own* functions, which we will cover in an upcoming module.

You can also access thousands of other functions and datasets through bundles of external code known as **packages**. Packages are developed and shared by R users around the world – a global community working together to increase R’s versatility and impact.

Some packages are designed to be broadly useful for almost any application, such as the packages you will be learning in this course (`ggplot`, `dplyr`, `stringr`, etc.). Such packages make it easier and more efficient to do your work with R.

Others are designed for niche problems that can be made much more doable with specialized functions or datasets. For example, the package `PBSmapping` contains shoreline, seafloor, and oceanographic datasets and custom mapping functions that make it easier for marine scientists at the Pacific Biological Station (PBS) in British Columbia, Canada, to carry out their work.

Packages you already have

In RStudio, look to the pane in the bottom right and click on the *Packages* tab. You should see something like this:

Files	Plots	Packages	Help	Viewer	
		<input type="button" value="Install"/> <input type="button" value="Update"/>		<input type="text"/> <input type="button" value="Search"/> <input type="button" value="Help"/>	
		Name	Description	Version	
System Library					
<input type="checkbox"/>	abind	Combine Multidimensional Arrays	1.4-5		
<input type="checkbox"/>	anytime	Anything to 'POSIXct' or 'Date' Converter	0.3.9		
<input type="checkbox"/>	aqp	Algorithms for Quantitative Pedology	1.25		
<input type="checkbox"/>	askpass	Safe Password Entry for R, Git, and SSH	1.1		
<input type="checkbox"/>	assertthat	Easy Pre and Post Assertions	0.2.1		
<input type="checkbox"/>	assocInd	Implements New and Existing Association Indices for Constructing Animal Social Networks	1.0.1		
<input type="checkbox"/>	audio	Audio Interface for R	0.1-7		
<input type="checkbox"/>	av	Working with Audio and Video in R	0.5.1		
<input type="checkbox"/>	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.10		
<input type="checkbox"/>	bangarang	Bangarang data tools for the Kitimat Fjord System	1.0.0		
<input checked="" type="checkbox"/>	base	The R Base Package	4.0.2		
<input type="checkbox"/>	base64enc	Tools for base64 encoding	0.1-3		
<input type="checkbox"/>	beepR	Easily Play Notification Sounds on any Platform	1.3		
<input type="checkbox"/>	BH	Boost C++ Header Files	1.75.0-0		
<input type="checkbox"/>	bioacoustics	Analyse Audio Recordings and Automatically Extract Animal Vocalizations	0.2.5		
<input type="checkbox"/>	bitops	Bitwise Operations	1.0-6		
<input type="checkbox"/>	bmp	Read Windows Bitmap (BMP) Images	0.3		
<input type="checkbox"/>	bookdown	Authoring Books and Technical Documents with R Markdown	0.21		
<input type="checkbox"/>	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-25		
<input type="checkbox"/>	brew	Templating Framework for Report Generation	1.0-6		
<input type="checkbox"/>	brio	Basic R Input Output	1.1.0		
<input type="checkbox"/>	bslib	Custom 'Bootstrap' 'Sass' Themes for 'shiny' and 'rmarkdown'	0.2.4		
<input type="checkbox"/>	cachem	Cache R Objects with Automatic Pruning	1.0.4		

This is displaying all the packages already installed in your system.

If you click on one of these packages (try the `base` package, for example), you will be taken to a list of all the functions and datasets contained within it.

The R Base Package 

Documentation for package ‘base’ version 4.0.2

- [DESCRIPTION file](#).
- [Code demos](#). Use `demo()` to run them.

Help Pages

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Z](#) [misc](#)

base-package	The R Base Package
------------------------------	--------------------

-- A --

abbreviate	Abbreviate Strings
abs	Miscellaneous Mathematical Functions
acos	Trigonometric Functions
asosh	Hyperbolic Functions

When you click on one of these functions, you will be taken to the help page for that function. This is the equivalent of typing `? <function_name>` into the *Console*.

Installing a new package

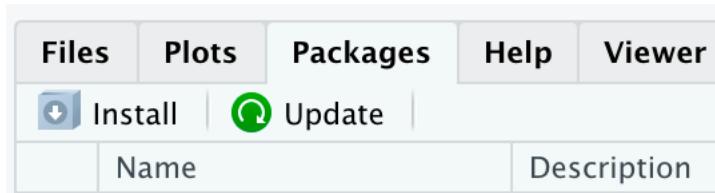
There are a couple ways to download and install a new R package on your computer. Most packages are available from an open-source repository known as CRAN (which stands for Comprehensive R Archive Network). However, an increasingly common practice is to release packages on a public repository such as GitHub.

Installing from CRAN

You can install CRAN packages one of two ways:

Through clicks:

In RStudio, in the bottom-right pane, return to the *Packages* tab. Click on the “Install” button.



You can then search for the package you wish to install then click **Install**.

Through code:

You can download packages from the *Console* using the `install.packages()` function.

```
install.packages('fun')
```

Note that the package name must be in quotation marks.

Installing from GitHub

To install packages from GitHub, you must first download a CRAN package that makes it easy to do so:

```
install.packages("devtools")
```

Most packages on GitHub include instructions for downloading it on its GitHub page.

For example, visit this GitHub page to see the documentation for the package **wesanderson**, which provides color palette themes based upon Wes Anderson's films. On this site, scroll down and you will find instructions for downloading the package. These instructions show you how to install this package from your R *Console*:

```
devtools::install_github("karthik/wesanderson")
```

Now go to your *Packages* tab in the bottom-right pane of RStudio, scroll down to find the **wesanderson** package, and click on it to check out its functions.

Loading an installed package

There is a difference between *installed* and *loaded* packages. Go back to your *Packages* tab. Notice that some of the packages have a checked box next to their names, while others don't.

These checked boxes indicate which packages are currently *loaded*. All packages in the list are *installed* on your computer, but only the checked packages are *loaded*, i.e., ready for use.

To load a package, use the `library()` function.

```
library(fun)
library(wesanderson)
```

Now that your new packages are loaded, you can actually use their functions.

To emphasize: a package is installed *only once*, but you `library()` the package in *each and every* script that uses it. Think of a package as camping gear. Like an R package, camping gear helps you do cool things that you can't really do with the regular stuff in your closet. And, like an R package, you only need to install (i.e., purchase) your gear once; but it is useless unless you pack it in your car (i.e., `library()` it) *every time* you go on a trip.

Calling functions from a package

Most functions from external packages can be used by simply typing the name of the function. For example, the package **fun** contains a function for generating a random password:

```
random_password(length=24)
[1] "};<Nv3b\\1a9ISF+*prXZ`g5"
```

Sometimes, however, R can get confused if a new package contains a function that has the same name of some function from a different package. If R seems confused about a function you are calling, it can help to specify which package the function can be found in. This is done using the syntax <package_name>::<function_name>. For example, the following command is a fine alternative to the command above:

```
fun::random_password(length=24)
[1] "9Vg#t]x!1N2Jh1WqK|YsHZ4z"
```

Note that this was done in the example above using the `devtools` package.

Side notes

Package dependencies

Most packages contain functions that are built using functions built from other packages. Those new functions depend on the functions from those other packages, and that's why those other packages are known as *dependencies*. When you install one function, you will notice that R typically has to install several other packages at the same time; these are the dependencies that allow the package of interest to function.

Package versions

Packages are updated regularly, and sometimes new versions can break the functions that use it as a dependency. Sometimes you may have to install a new version (*or sometimes an older version!*) of a dependency in order to get your package of interest to work as desired.

Review: the workflow for using a package

To review how to use functions from a non-base package in R, follow these steps (examples provided:)

1. Install the package *once*.

```
# Example from CRAN
install.packages("wesanderson")

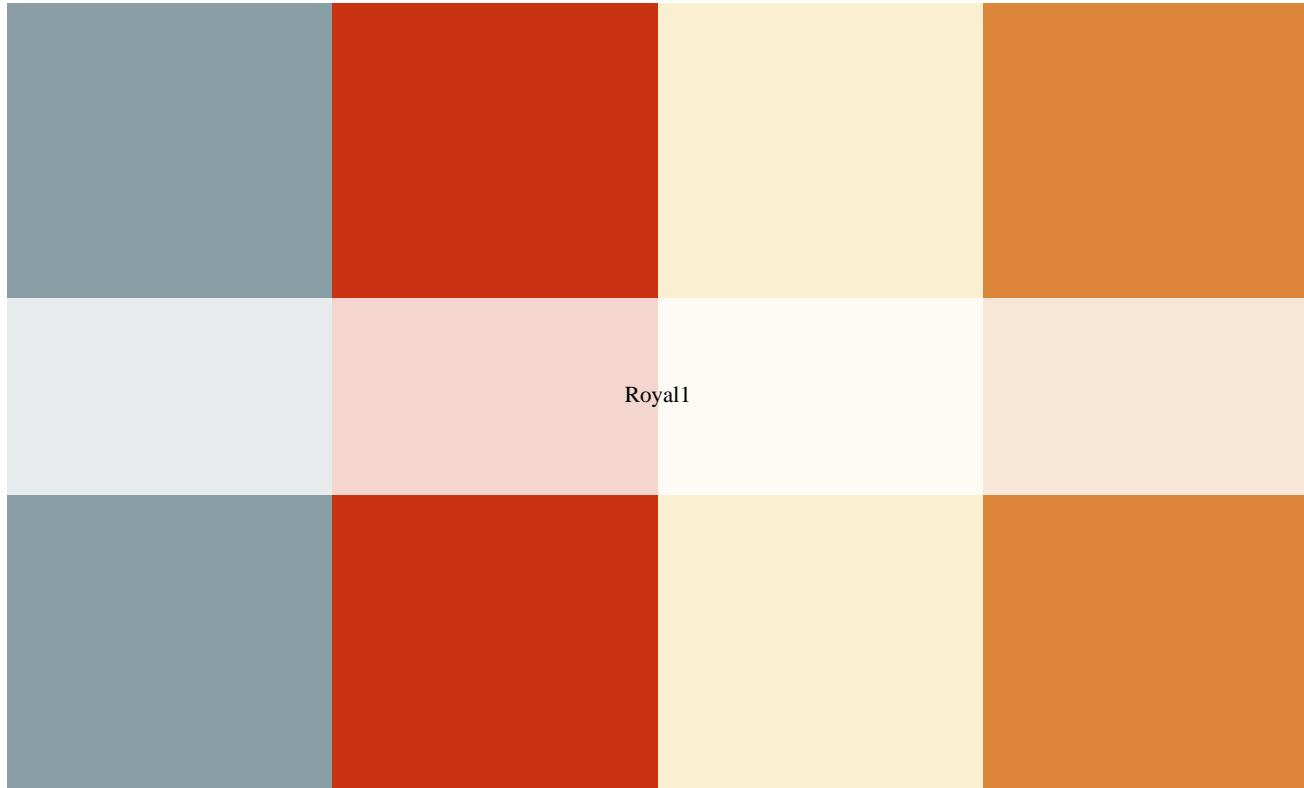
# Example from GitHub
devtools::install_github("karthik/wesanderson")
```

2. Load the package *in each script*.

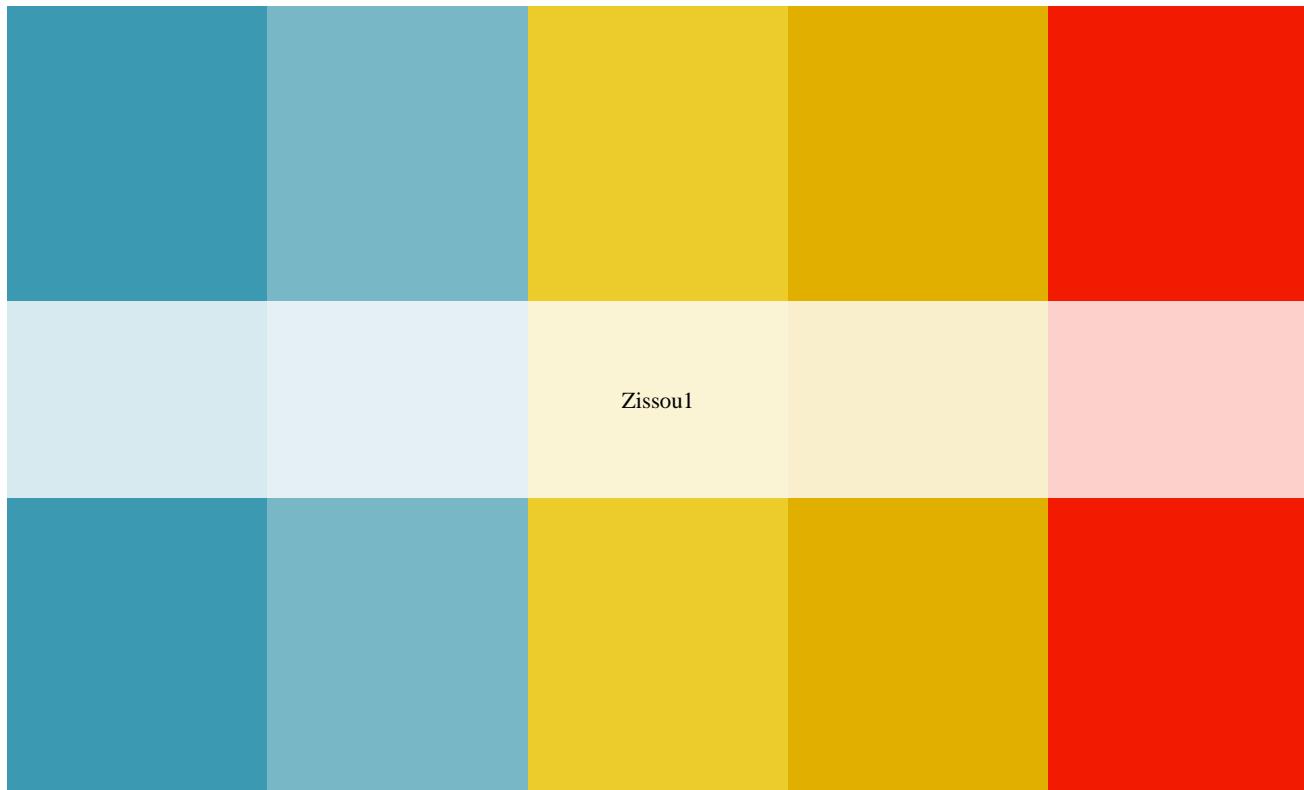
```
# Example
library(wesanderson)
```

3. Call the function.

```
wes_palette("Royal1")
```



```
wesanderson::wes_palette("Zissou1")
```



(This function creates a plot displaying the different colors contained within the specified palette.)

4. Get help with the question mark: ?

```
?wes_palette
```

Exercises

Let's install some packages:

1. Install the `babynames` package.
2. Install `ggplot2`.
3. Install `dplyr`.
4. Install `RColorBrewer`.
5. Install `tidyverse`.
6. Install `gapminder`.
7. Install `readr`.
8. Install `gsheet`.
9. Install `readxl`.
10. Write 7 lines of code which *load* the above packages.

(PART) Basic R workflow

Chapter 13

Importing data

Learning goals

- How to load, or “read”, your data into R
- How to format your data for easily importing data in R
- Understand what a .csv file is, and why they are important in data science
- How to set up your project directory and read data from other folders

To work with your own data in R, you need to load your data in R’s memory. This is called **reading in** your data.

Reading in data

The general workflow for reading in data is as follows:

1. In RStudio, set your working directory.
2. Place your data file in your working directory. (See the section below if you want to keep your data somewhere else.)
3. In your R script, read in your data file with one of the core functions below.

You can use this simple data file, , to practice.

Core functions for reading data

To become agile in reading various types of data into R, here are five key functions you should know:

`readr::read_csv()`

Reading in data is simple and easy if your data are saved as a .csv, a comma-separated file. You can find functions for reading all sorts of file types into R, but the quickest way to begin working with your own data in R is to maintain that data in .csv’s.

The function `read_csv()`, from a package named `readr`, becomes useful when you begin working with (1) data from the internet, (2) data within the `tidyverse`, which you will be introduced to in the next module, and/or (2) very large dataset, since it reads data much more quickly and provides progress updates along the way.

Here’s an example of reading a file directly from the internet ...

```
library(readr)
df <- read_csv('https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/deaths.csv')
df
# A tibble: 891 x 12
  PassengerId Survived Pclass Name     Sex      Age SibSp Parch Ticket   Fare Cabin
      <dbl>     <dbl> <dbl> <chr>   <chr>   <dbl> <dbl> <dbl> <chr>   <dbl> <chr>
```

```

1      1      0      3 Braun~ male    22      1      0 A/5 2~  7.25 <NA>
2      2      1      1 Cumin~ fema~   38      1      0 PC 17~ 71.3  C85
3      3      1      3 Heikk~ fema~   26      0      0 STON/~  7.92 <NA>
4      4      1      1 Futre~ fema~   35      1      0 113803 53.1  C123
5      5      0      3 Allen~ male    35      0      0 373450  8.05 <NA>
6      6      0      3 Moran~ male    NA      0      0 330877  8.46 <NA>
7      7      0      1 McCar~ male    54      0      0 17463   51.9  E46
8      8      0      3 Palss~ male    2       3      1 349909 21.1  <NA>
9      9      1      3 Johns~ fema~   27      0      2 347742 11.1  <NA>
10     10     1      2 Nasse~ fema~   14      1      0 237736 30.1  <NA>
# ... with 881 more rows, and 1 more variable: Embarked <chr>

```

Note that when you use `read_csv()` instead of `read.csv()`, your data are read in as a `tibble` instead of a datafram. You will be introduced to `tibbles` in the next modules on dataframes; for the time being, think of a `tibble` as a fancy version of a datafram that can be treated exactly as a regular datafram.

`read.csv()`

This function, `read.csv()`, is the base function for reading in a `.csv`. It is strictly used for reading in local files (not from the internet).

This function reads in your data file as a datafram. Save your dataset into R's memory using a variable (in this case, `df`).

```

df <- read.csv("super_data.csv")
df

```

	patient_id	height_in	weight_lb	comment
1	1	74	135	not very nice
2	2	56	112	kinda cute
3	3	59	156	kinda cute but had a ring
4	4	43	102	so small!

The `read.csv()` function has plenty of other inputs in the event that your data file is unable to follow the formatting rules outlined above (see `?read.csv()`). The three most common inputs you may want to use are `header`, `skip`, and `stringsAsFactors`.

- Use the `header` input when your data does not contain column names. *For example*, `header=FALSE` indicates that your datafile does not have any column names.
- Use the `skip` input when you want to skip some lines of metadata at the top of your file. This is handy if you really don't want to get rid of your metadata in your header. *For example*, `skip=2` skips the first two rows of the datafile before R begins reading data.
- Use the `stringsAsFactors` input when you want to make absolutely sure that R interprets any non-numeric fields as characters rather than factors. We have not focused on factors yet, but it can be frustrating when R mistakes a column of character strings as a column factors. To avoid any possible confusion, use `stringsAsFactors=TRUE` as an input.

For example, here is how to read in this data without column names:

```

df <- read.csv("super_data.csv",skip=1,header=FALSE)
df
  V1 V2  V3          V4
1  1 74 135      not very nice
2  2 56 112      kinda cute
3  3 59 156 kinda cute but had a ring
4  4 43 102      so small!

```

If you do this without setting header to FALSE, your first row of data gets used as column names and it becomes a big ole mess:

```
df <- read.csv("super_data.csv", skip=1)
df
  X1 X74 X135           not.very.nice
1  2  56  112           kinda cute
2  3  59  156 kinda cute but had a ring
3  4  43  102           so small!
```

`readxl()`

To read in an *Excel* spreadsheet, use the `read_xlsx()` function from the package `readxl`.

If `super_data` were an `.xlsx` file, the command would look like this:

```
library(readxl)
df <- read_xlsx("super_data.xlsx", sheet=1)
```

`gsheet()`

To read in an *GoogleSheets* spreadsheet, use the `gsheet2tbl()` function from the package `gsheet`.

Make sure link sharing is turned on for the *GoogleSheet* you are trying to access:

```
library(gsheet)
df <- gsheet2tbl("https://docs.google.com/spreadsheets/d/1uQ5PfGITnjHngK41FWHeedioca30wpyEsB0pc_B6T3E/edit")
df
# A tibble: 4 x 4
  patient_id height_in weight_lb comment
    <dbl>      <dbl>     <dbl> <chr>
1         1        74      135 not very nice
2         2        56      112 kinda cute
3         3        59      156 kinda cute but had a ring
4         4        43      102 so small!
```

`readRDS()`

Another niche function for reading data is `readRDS()`. This function allows you to read in *R data objects*, which have the file extension `.rds`. These data objects need not be in the same format as a `.csv` or even a dataframe, and that is what makes them so handy. A colleague could send you an `.rds` object of a vector, a list, a plotting function, or any other kind of R object, and you can read it in with `readRDS()`.

For example, contains a `tibble` version of the dataframe above. When you read in that `.rds` file, it is already formatted as a `tibble`:

```
df <- readRDS("super_data.rds")
df
# A tibble: 4 x 4
  patient_id height_in weight_lb comment
    <dbl>      <dbl>     <dbl> <chr>
1         1        74      135 not very nice
2         2        56      112 kinda cute
3         3        59      156 kinda cute but had a ring
4         4        43      102 so small!
```

.csv files

The `.csv` is such an important file type when you work with data that it is worth introducing it with more detail.

When you preview a `.csv`, it looks something like this:

station	year	doy	sit	x	y	n	bhvr	id	i
Boat	2004	159	2004060701	-129.1875	53.101	NA	TR	BCY0470	
Boat	2004	189	2004070701	-129.1875	53.101	NA	BNF	BCX0361	
Boat	2004	196	2004071401	-129.1875	53.101	NA	BNF	BCX0361	
Boat	2004	196	2004071401	NA	NA	NA	BNF	BCX0711	
Boat	2004	200	2004071801	-129.1875	53.101	NA	BNF	BCX0275	
Boat	2004	209	2004072701	-129.194	53.1018	NA	MI-FE	BCY0321	
Boat	2004	216	2004080301	-129.1875	53.101	NA	BNF	BCY0049	
Boat	2004	216	2004080301	NA	NA	NA	BNF	BCX0255	
Boat	2004	217	2004080401	-129.1916	53.0696	NA	MI-FE	BCY0117	
Boat	2004	217	2004080401	NA	NA	NA	MI-FE	BCY0189	
Boat	2004	218	2004080501	-129.18067	53.0899	NA	BNF	BCX0083	
Boat	2004	218	2004080501	NA	NA	NA	BNF	BCX0231	
Boat	2004	219	2004080601	-129.1875	53.101	NA	BNF	BCY0013	
Boat	2004	221	2004080801	-129.1875	53.101	NA	TR	BCX0231	
Boat	2004	221	2004080801	NA	NA	NA	TR	BCX0121	
Boat	2004	223	2004081001	-129.1875	53.101	NA	TR	BCX0121	
Boat	2004	227	2004081401	-129.1875	53.101	NA	TR	BCX0171	
Boat	2004	228	2004081501	NA	NA	NA	TR	BCX0022	
Boat	2004	228	2004081502	NA	NA	NA	TR	BCX0121	
Boat	2004	229	2004081601	-129.1875	53.101	NA	TR	BCX0022	
Boat	2004	230	2004081701	-129.1875	53.101	NA	RE-TR	BCY0013	
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0171	
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCY0013	
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0694	
Boat	2004	230	2004081703	NA	NA	NA	RE-TR	BCX0239	
Boat	2004	230	2004081704	NA	NA	NA	TR	BCY0117	
Boat	2004	231	2004081801	-129.194	53.1018	NA	MI-FE	BCX0567	
Boat	2004	231	2004081802	-129.237	53.068	NA	BNF	BCX0083	
Boat	2004	233	2004082001	NA	NA	NA	TR	BCX0174	
Boat	2004	235	2004082201	-129.1875	53.101	NA	TR	BCX0380	
Boat	2004	235	2004082201	NA	NA	NA	TR	BCX0380	

A neat spreadsheet of rows and columns.

When you open up this same dataset in a simple text editor, it looks like this:

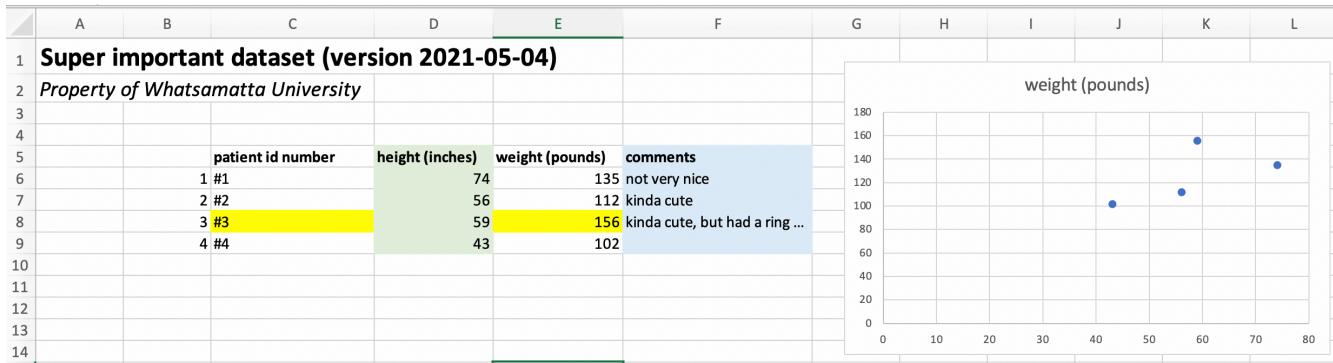
station	year	doy	sit	x	y	n	bhvr	id	name	stage	bhvr2	bhvr3	loc.conf	type	conf	month	dom	date	time
Boat	2004	159	2004060701	-129.1875	53.101	NA	TR	BCY0470	,A,TR,TR	,0,1,6,7,6/7/04	16:35								
Boat	2004	189	2004070701	-129.1875	53.101	NA	BNF	BCX0361	,,F,BNF,BNF	,0,1,7,7,7/7/04	13:33								
Boat	2004	196	2004071401	-129.1875	53.101	NA	BNF	BCX0361	,,F,BNF,BNF	,0,1,7,14,7/14/04	14:55								
Boat	2004	196	2004071401	NA	NA	NA	BNF	BCX0711	,,A,BNF,BNF	,0,1,7,14,7/14/04	14:55								
Boat	2004	200	2004071801	-129.1875	53.101	NA	BNF	BCX0275	,,A,BNF,BNF	,0,1,7,18,7/18/04	16:52								
Boat	2004	209	2004072701	-129.194	53.1018	NA	MI-FE	BCY0321	,,A,MI-FE,MI-FE	,0,1,7,27,7/27/04	7:35								
Boat	2004	216	2004080301	-129.1875	53.101	NA	BNF	BCY0049	,,F,BNF,BNF	,0,1,8,3,8/3/04	13:41								
Boat	2004	216	2004080301	NA	NA	NA	BNF	BCX0255	,,A,BNF,BNF	,0,1,8,3,8/3/04	13:41								
Boat	2004	217	2004080401	-129.1916	53.0696	NA	MI-FE	BCY0117	,,A,MI-FE,MI-FE	,0,1,8,4,8/4/04	12:50								
Boat	2004	217	2004080401	NA	NA	NA	MI-FE	BCY0189	,,F,MI-FE,MI-FE	,0,1,8,4,8/4/04	12:50								
Boat	2004	218	2004080501	-129.18067	53.0899	NA	BNF	BCX0083	,,A,BNF,BNF	,0,1,8,5,8/5/04	14:23								
Boat	2004	218	2004080501	NA	NA	NA	BNF	BCX0231	,,A,BNF,BNF	,0,1,8,5,8/5/04	14:23								
Boat	2004	219	2004080601	-129.1875	53.101	NA	BNF	BCY0013	,,A,BNF,BNF	,0,1,8,6,8/6/04	8:19								
Boat	2004	221	2004080801	-129.1875	53.101	NA	TR	BCX0231	,,A,TR,TR	,0,1,8,8,8/8/04	12:28								
Boat	2004	221	2004080801	NA	NA	NA	TR	BCX0121	,,F,TR,TR	,0,1,8,8,8/8/04	8:19								
Boat	2004	223	2004081001	-129.1875	53.101	NA	TR	BCX0121	,,F,TR,TR	,0,1,8,10,8/10/04	13:13								
Boat	2004	227	2004081401	-129.1875	53.101	NA	TR	BCX0171	,,F,TR,TR	,0,1,8,14,8/14/04	11:44								
Boat	2004	228	2004081501	NA	NA	NA	TR	BCX0022	,,A,TR,TR	,0,1,8,15,8/15/04	11:44								
Boat	2004	228	2004081502	NA	NA	NA	TR	BCX0121	,,F,TR,TR	,0,1,8,15,8/15/04	18:30								
Boat	2004	229	2004081601	-129.1875	53.101	NA	TR	BCX0022	,,A,TR,TR	,0,1,8,16,8/16/04	9:00								
Boat	2004	230	2004081701	-129.1875	53.101	NA	RE-TR	BCY0013	,,A,RE-TR,RE-TR	,0,1,8,17,8/17/04	7:10								
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0171	,,F,BNF,BNF	,0,1,8,17,8/17/04	11:50								
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCY0013	,,A,BNF,BNF	,0,1,8,17,8/17/04	11:50								
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0694	,,A,BNF,BNF	,0,1,8,17,8/17/04	11:50								
Boat	2004	230	2004081703	NA	NA	NA	RE-TR	BCX0239	,,M,RE-TR,RE-TR	,0,1,8,17,8/17/04	14:00								
Boat	2004	230	2004081704	NA	NA	NA	TR	BCY0117	,,A,TR,TR	,0,1,8,17,8/17/04	15:30								
Boat	2004	231	2004081801	-129.194	53.1018	NA	MI-FE	BCX0567	,,A,MI-FE,MI-FE	,0,1,8,18,8/18/04	14:36								
Boat	2004	231	2004081802	-129.237	53.068	NA	BNF	BCX0083	,,A,BNF,BNF	,0,1,8,18,8/18/04	14:06								
Boat	2004	233	2004082001	NA	NA	NA	TR	BCX0174	,,F,TR,TR	,0,1,8,20,8/20/04	11:15								
Boat	2004	235	2004082201	-129.1875	53.101	NA	TR	BCX0380	,,A,TR,TR	,0,1,8,22,8/22/04	16:09								
Boat	2004	236	2004082301	-129.1933	53.0527	NA	RE-TR	BCX0049	,,F,RE-TR,RE-TR	,0,1,8,23,8/23/04	10:30								
Boat	2004	236	2004082303	-129.18067	53.0899	NA	MI-FE	BCX0380	,,A,MI-FE,MI-FE	,0,1,8,23,8/23/04	13:41								
Boat	2004	239	2004082603	-129.237	53.068	NA	RE-TR	BCY0189	,,F,RE-TR,RE-TR	,0,1,8,26,8/26/04	8:05								
Boat	2004	239	2004082605	-129.2113	53.077	NA	TR	BCZ0053	,,F,TR,TR	,0,1,8,26,8/26/04	9:00								
Boat	2004	239	2004082605	-129.1875	53.101	NA	RE-TR	BCZ0053	,,F,RE-TR,RE-TR	,0,1,8,26,8/26/04	19:12								
Boat	2004	239	2004082605	NA	NA	NA	RE-TR	BCY0135	,,A,RE-TR,RE-TR	,0,1,8,26,8/26/04	19:12								
Boat	2004	246	2004090201	-129.1933	53.0527	NA	TR	BCY0013	,,A,TR,TR	,0,1,9,2,9/2/04	10:40								
Boat	2004	246	2004090202	-129.1933	53.0527	NA	RE-TR	BCX0386	,,A,RE-TR,RE-TR	,0,1,9,2,9/2/04	10:45								
Boat	2004	246	2004090203	NA	NA	NA	RE-TR	BCX0144	,,A,RE-TR,RE-TR	,0,1,9,2,9/2/04	11:00								
Boat	2004	246	2004090203	NA	NA	NA	RE-TR	BCX0171	,,F,RE-TR,RE-TR	,0,1,9,2,9/2/04	11:00								
Boat	2004	249	2004090501	-129.1875	53.101	NA	TR	BCY0347	,,A,TR,TR	,0,1,9,5,9/5/04	9:30								
Boat	2004	251	2004090701	NA	NA	NA	TR	BCX0750	,,A,TR,TR	,0,1,9,7,9/7/04	11:00								
Boat	2004	251	2004090701	NA	NA	NA	TR	BCY0383	,,A,TR,TR	,0,1,9,7,9/7/04	11:00								
Boat	2004	253	2004090902	-129.3198	53.0977	NA	BNF	BCY0013	,,A,BNF,BNF	,0,1,9,9,9/9/04	13:00								
Boat	2004	253	2004090902	NA	NA	NA	BNF	BCX0239	,,M,BNF,BNF	,0,1,9,9,9/9/04	13:00								
Boat	2004	253	2004090902	NA	NA	NA	MI-FE	BCX0239	,,C,MI,MI	,0,1,9,9,9/9/04	13:00								
Boat	2004	253	2004090903	-129.3032	53.0711	NA	TR	BCY0013	,,A,TR,TR	,0,1,9,9,9/9/04	14:20								
Boat	2004	253	2004090903	NA	NA	NA	TR	BCX0171	,,F,TR,TR	,0,1,9,9,9/9/04	14:20								
Boat	2004	253	2004090903	NA	NA	NA	TR	BCX0144	,,F,TR,TR	,0,1,9,9,9/9/04	14:20								
Boat	2004	253	2004090904	-129.2002	53.0472	NA	BNF	BCX0711	,,A,BNF,BNF	,0,1,9,9,9/9/04	15:30								
Boat	2004	254	2004091005	NA	NA	NA	TR	BCX0022	,,A,TR,TR	,0,1,9,10,9/10/04	15:30								
Boat	2004	258	2004091401	-129.2002	53.0472	NA	TR	BCX0567	,,A,TR,TR	,0,1,9,14,9/14/04	12:12								
Boat	2004	258	2004091402	-129.2002	53.0472	NA	BNF	BCY0277	,,A,BNF,BNF	,0,1,9,14,9/14/04	13:30								
Boat	2004	258	2004091402	NA	NA	NA	BNF	BCY0013	,,A,BNF,BNF	,0,1,9,14,9/14/04	13:30								

This looks scary, but it is actually really simple. A .csv is a simple text file in which each row is on a new line and columns of data are separated by commas. As a simple text file, there is no fancy formatting. There are no “Sheets” or “Tabs”, as you would find in GoogleSheets or Excel; it is a simple ‘flat’ file.

One of the major advantages of working with .csv’s is that the format is cross-platform and non-proprietary. That is, they work on Windows, Mac, Linux, and any other common type of computer, and they do not require special software to open.

Prepping your data for R

For those of us used to working in *Excel*, *GoogleSheets*, or *Numbers*, it will take some adjustment to get into the habit of formatting your data for R. We are used to seeing spreadsheets that look something like this:



To read a `.csv` into R without issues or fancy code, this spreadsheet will need to be simplified to look like this:

	A	B	C	D
1	patient_id	height_in	weight_lb	comments
2	1	74	135	not very nice
3	2	56	112	kinda cute
4	3	59	156	kinda cute but had a ring
5	4	43	102	
c				

Workflow for formatting your data

Below is the general workflow for preparing your data for R is the following:

- Get your data into `.csv` format.** In *Excel* and *Numbers*, you can use ‘Save As’ to change the file format. In *GoogleSheets*, you can ‘Download As’ a `.csv`. This will remove any colors, thick lines, special fonts, bold or italicized font styles, and any other special formatting. All that will be left is your data, and that’s the way R likes it.
- Remove blank columns** before and in the middle of your data.
- Remove fancy elements such as graphs.**
- Simplify your ‘header’.** The space above your data is your spreadsheet’s header. It includes column names and metadata like title, author, measurement units, etc. It is possible to read data with complex headers into R, but again we are going for simplicity here, so we suggest (1) simplifying your header to contain column names only, and (2) moving metadata to a `README.txt` file that lives in the same folder as your data.
- Simplify column names.** Remove spaces, or replace them with `,`, `-` or `_`. Make your column names as simple and brief as possible while still being informative. Include units in the column names, as in the screenshot above. Be sure that each column has a name.
- Remove all commas and hashtags from your dataset.** You can do this with the ‘Find & Replace’ feature built-into in most spreadsheet editors.

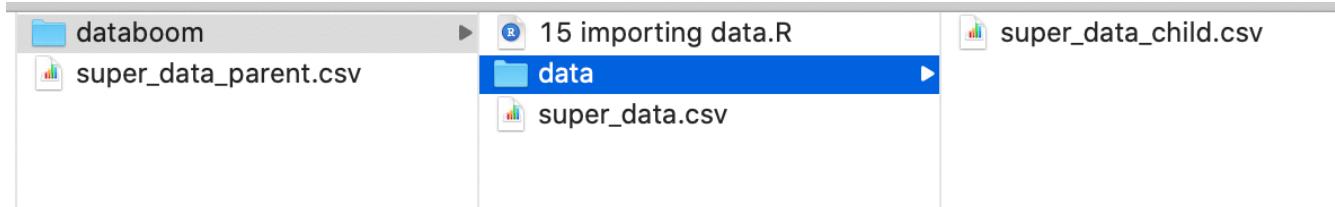
Managing files & folders

Reading data from other folders

The data-reading functions above require only a single input: the *path* to your data file. This *path* is relative to the location of your working directory. When your data file is *inside* your working directory, the path simplifies to be the same as the filename of your data:

```
df <- read.csv("super_data.csv")
```

Sometimes, though, you will want to keep your data somewhere nearby but not necessarily *within* your working directory. Consider the following scenario, in which three versions of the “super_data.csv” dataset occur near a working directory being used for this module:



We have a version within the same directory as our R file (i.e., our working directory), another version within a *child* folder within the directory (i.e., a subfolder), and another version in the *parent* folder of the working directory.

To read a file from a *child* folder, add the prefix, `./<child name>/`, to your command:

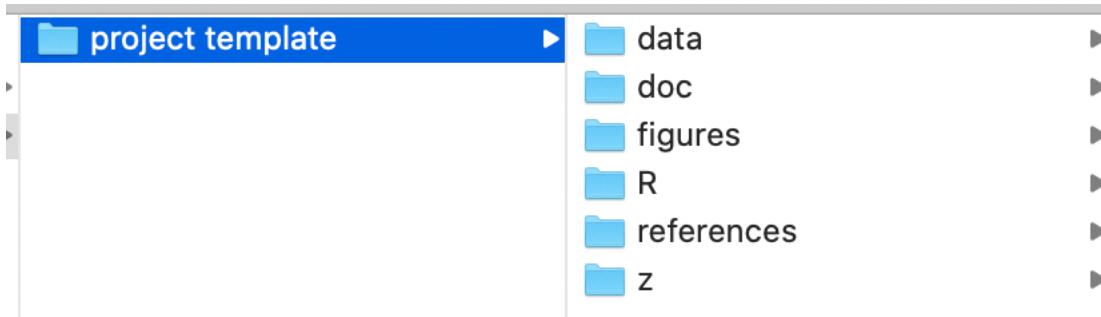
```
df <- read.csv("./data/super_data.csv")
```

To read a file from a *parent* folder, add the prefix, `../`, to your command:

```
df <- read.csv("../data/super_data.csv")
```

Managing files

Now consider the following scenario, in which your project folder structure looks like this:



This structure can be an effective and simple way of organizing your files for a project, and we recommend using it.

Here's what these child folders should contain.

- `./data/` contains data, of course.
- `./doc/` contains documents, such as manuscript drafts.
- `./figures/` contains files for graphs and figures.
- `./R/` contains R scripts, of course.
- `./references/` contains journal articles and other resources you are using in your research.

Since your R code is going into the R child folder, that is what you should set your working directory for those R scripts to. In that case, *how to read data from the `data` folder*, which is a separate child folder of your parent folder?

Here's how:

```
df <- read.csv("../data/super_data.csv")
```

Chapter 14

ggplot

Learning goals

- Understand what `ggplot2` is and why it's used
- Be able to think conceptually in the framework of the “grammar of graphics”
- Learn the basic syntax for creating different plots using `ggplot2`

What is `ggplot2`?

`ggplot2` is an R package. It's one of the most downloaded packages in the R universe, and has become the gold standard for data visualization. It's extremely powerful and flexible, and allows for creating lots of visualizations of different types, ranging from maps to bare-bones academic publications, to complex, paneled charts with labeling, etc.

Because the syntax is so different from “base” R, it can give the impression of having a somewhat steep learning curve. But in reality, because the principles are so conceptually simple, learning is fairly fast. Generally those who choose to learn it stick with it; that is, once you go gg, you don't go back.

Note: we will refer heavily to this [online guide about ggplot](#)

The name & concept

The “gg” in `ggplot` stands for “grammar of graphics”, with “grammar” meaning “the fundamental principles or rules of an art or science” (Wickham, 2010). Just as all languages share common principles of grammar and syntax, so too do the many forms of data visualization. The basic idea is that all graphs can be described using a *layered* grammar: all graphs represent a dataset using the same layers of visual order.

Plots are made of layers. Think of how you draw a plot from scratch:

First, you get a piece of paper – a **canvas**.

Second, you draw the x axis and y axis: each direction on your canvas represents the range of a set of data. This establishes a **landscape of coordinates**.

Third, the data need to be placed somewhere in that landscape. You **map the data** to the coordinates.

Fourth, when you actually draw the data at their prescribed locations on the plot, you have to decide how to do so. You use **geometric objects** – like points, lines, and bars – and other **aesthetic attributes** – like colors, line thicknesses, and dot size.

Fifth, you add **labels** – such as axis titles, an overhead title, or a legend – to help the viewer understand the plot.

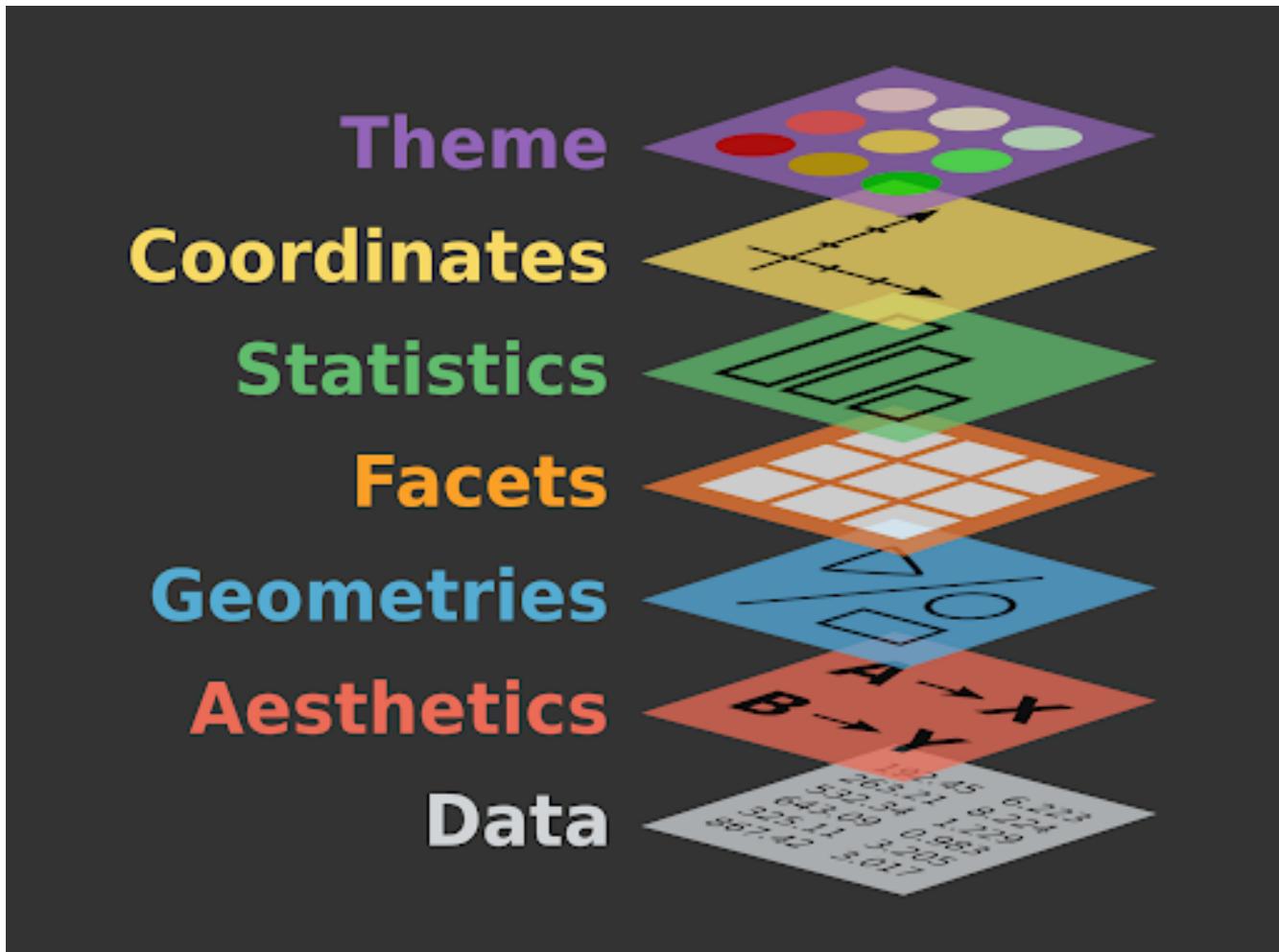
You now have a basic plot. But sometimes you will add additional layers:

Sixth, you may add **statistical summaries** – such as regression lines or standard error bars.

Seventh, you may decide to do an overhaul and split your plot into several **facets**, in which subgroups of the data are plotted separately to produce a multi-panel plot.

Finally, in the final layer, you may decide to stylize the entire plot to fit a **visual theme**, such as the trademark styles of vendors like the *The Economist* or *The New York Times*.

When you produce a plot with `ggplot`, you will mirror this same process step-by-step. This is why you will often see the process underlying `ggplot` described using a graphic like this:



Note: If you want to learn more about the theory, the most well-known “grammar of graphics” was written in 2005 and laid out some abstract principles for describing statistical graphics (Wilkinson, 2005).

Setting up ggplot

Let's learn by doing. First, install and load `ggplot2` and associated packages.

```
library(ggplot2)
library(readr)
library(ggthemes)
library(dplyr)
```

Download the `titanic` dataset, the manifest of *Titanic* passengers with details such as age, passenger class, fare paid, and whether or not they survived.

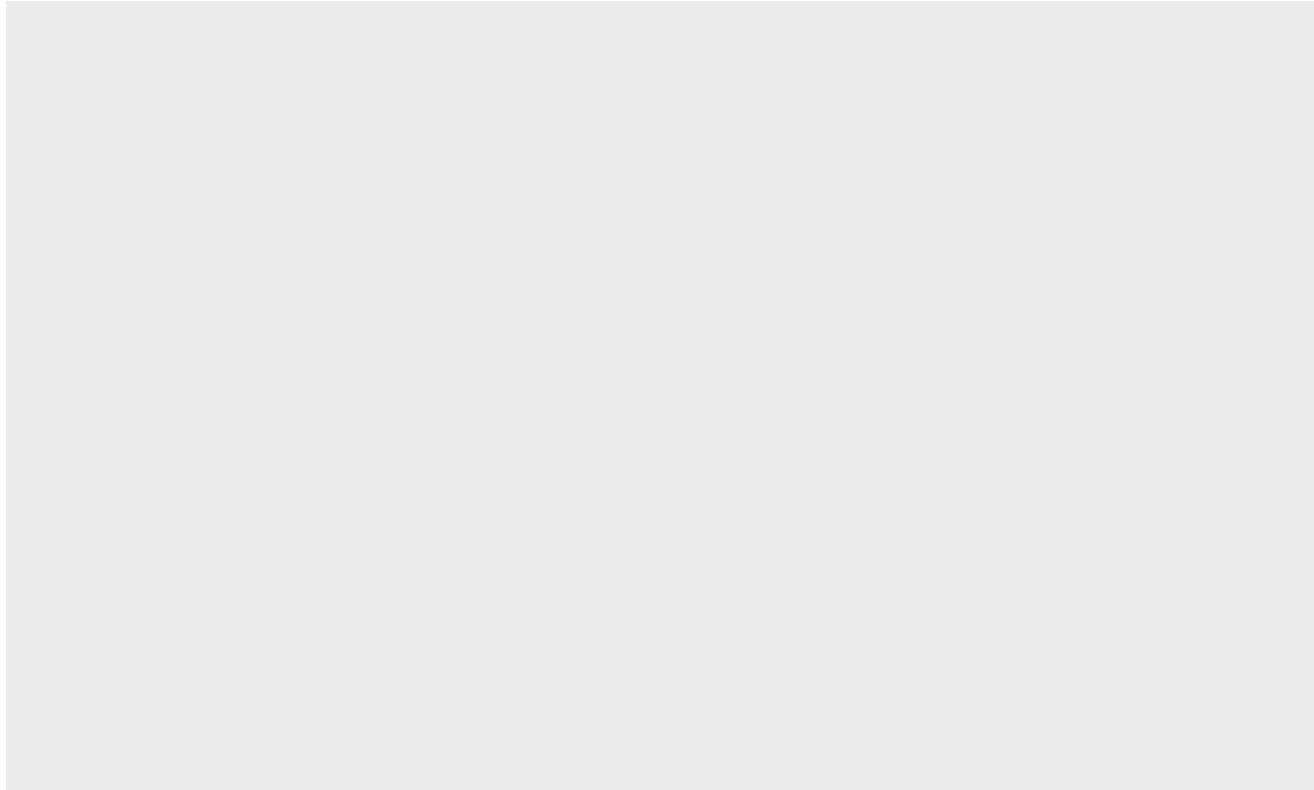
```
titanic<- read_csv("https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/deaths.csv")
```

Scatter plot

Say you want to explore the relationship between passengers' age and the fare they paid to travel aboard the *Titanic*.

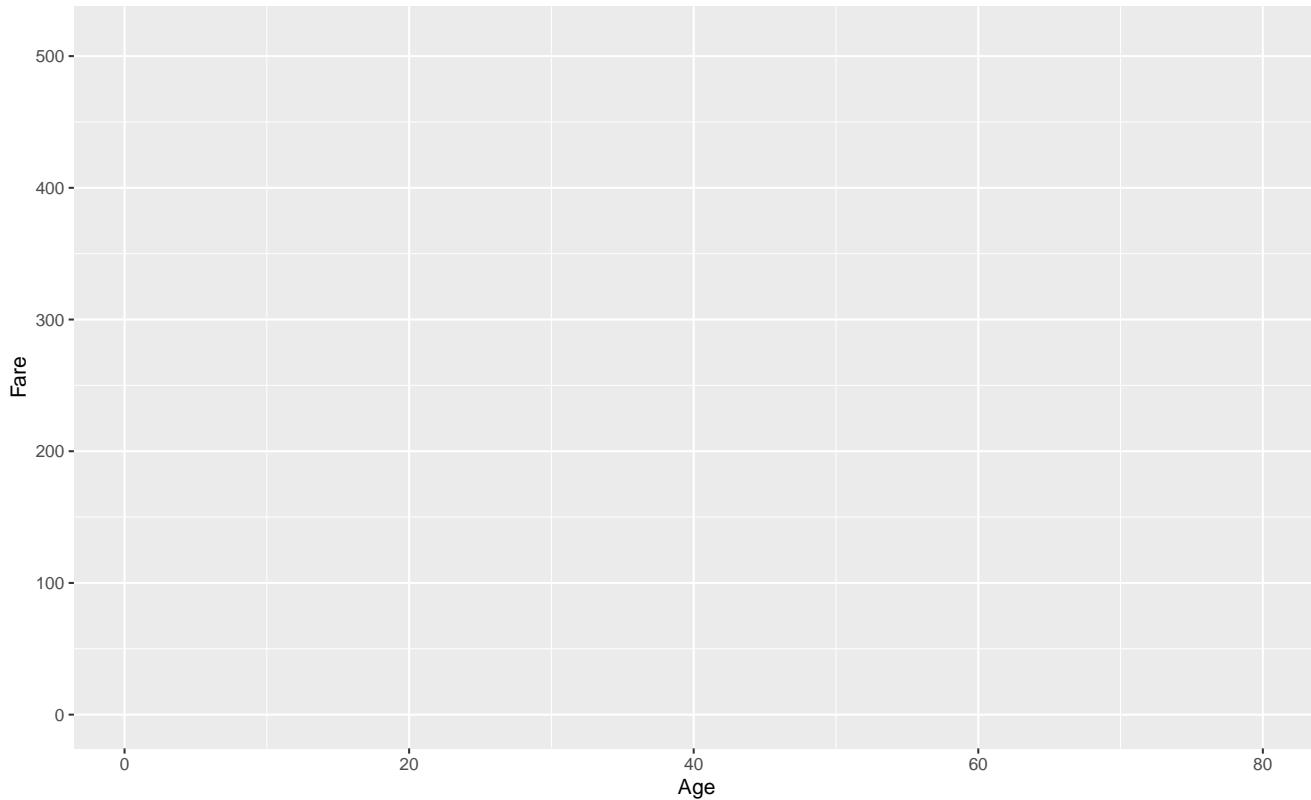
- (1) **Set up our canvas.** If we just type `ggplot()` without anything in the parentheses, the function will just return a blank piece of paper.

```
ggplot()
```



- (2) **Draw the axes** and, 3., setup our **landscape of coordinates**. To do so, we need to feed `ggplot()` some data and tell it which columns should be mapped onto the axes.

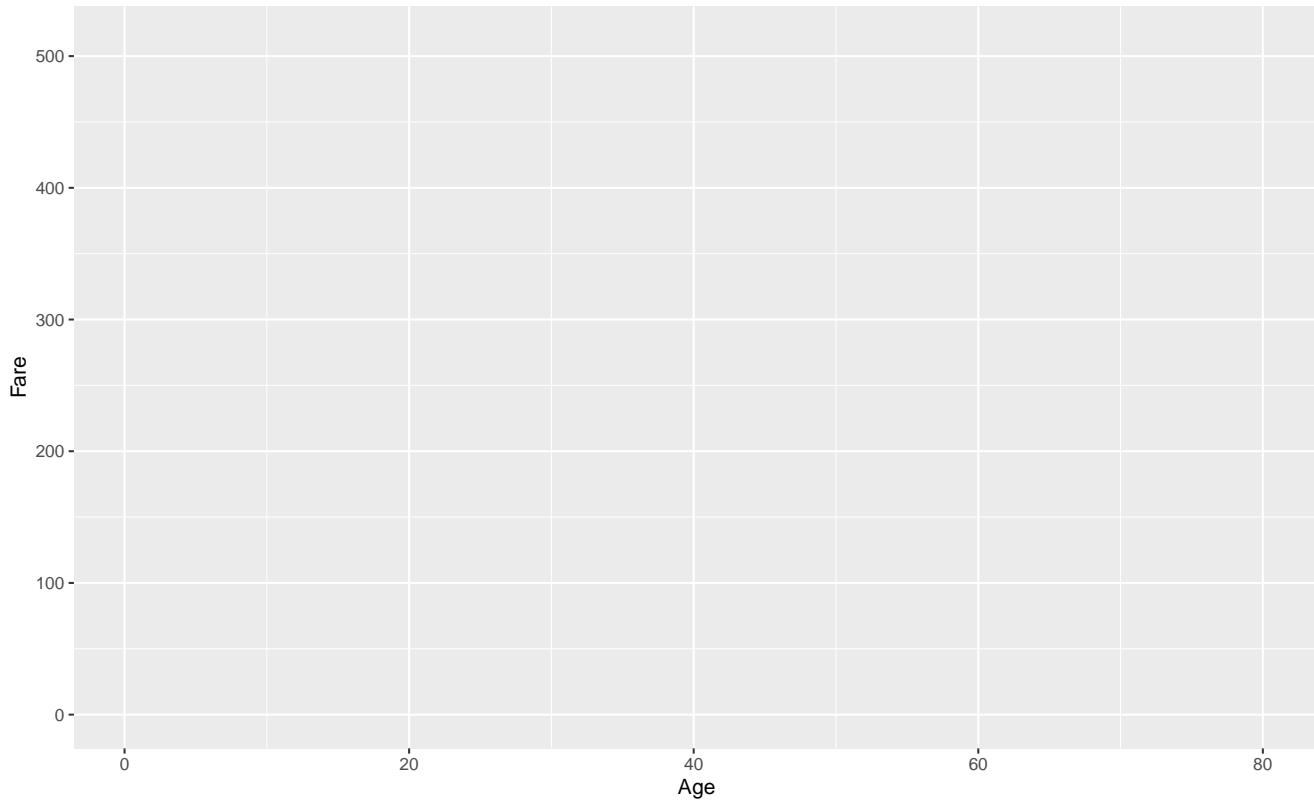
```
ggplot(data = titanic,  
       aes(x = Age, y = Fare))
```



That code looks a bit clunky, we know. The `aes()` input, which is short for `aesthetics`, is actually a function. Everything included in its parentheses will be used to *map your data to the plot's aesthetic attributes*. So far we have simply said that `Age` should be mapped to the *x* axis and that `Fare` should be mapped to *y*.

But let's say we also want to color-code the points on our plot according to male/female. To do so, we will add specifications to this `aes()` function.

```
ggplot(data = titanic,  
       aes(x = Age, y = Fare, color=Sex))
```



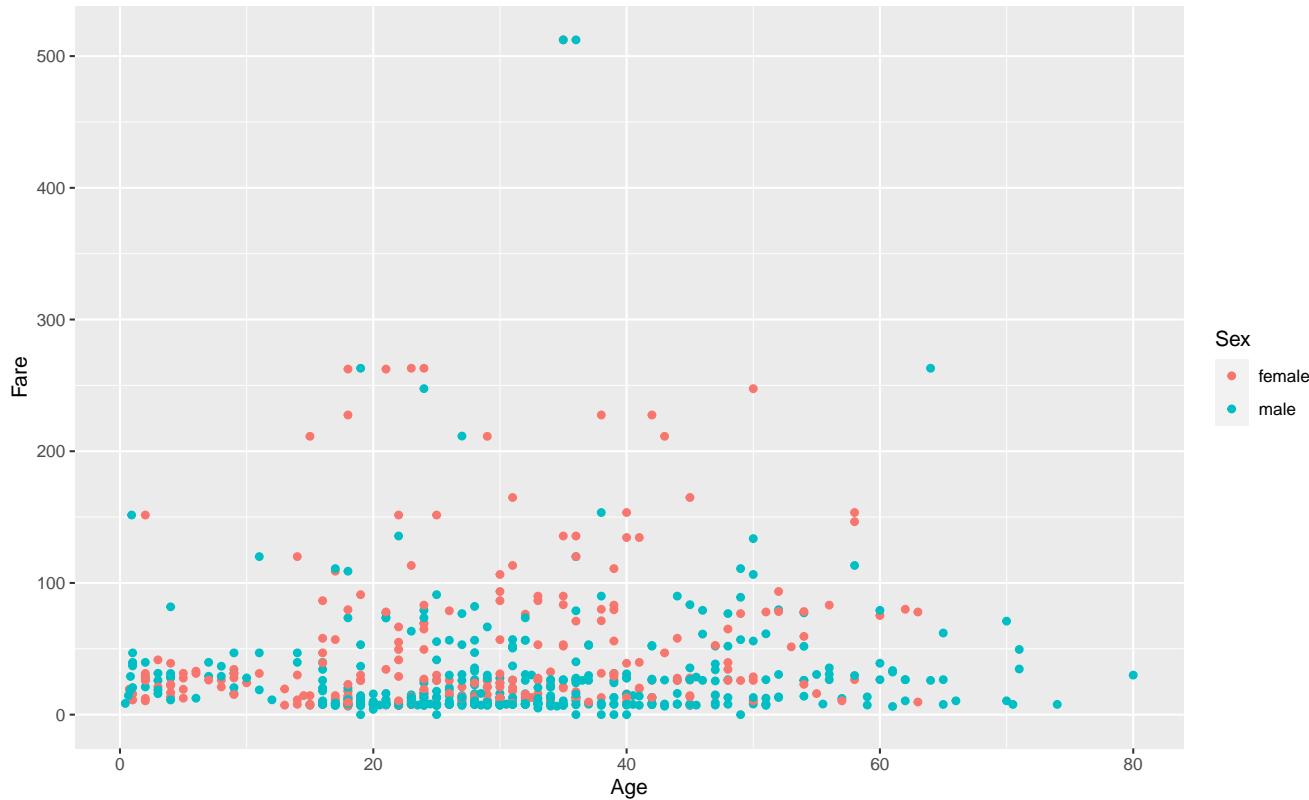
Your plot is still blank, but in the background `ggplot()` is all setup to make your plot. Since this `ggplot()` call is the basis of everything that will happen next – it contains the data and the way you want to map it to attributes of your plot – let's save it to a variable for easy recall. We'll use `p` for “plot”.

```
p <- ggplot(data = titanic,
             aes(x = Age, y = Fare, color=Sex))
```

Note that you don't need to write out `titanic$Age` or `titanic$Fare`. You've told `ggplot` that your data is `titanic`, so it knows to look inside that dataframe for those columns.

(4) **Map our data to geometric shapes.** In this case, a scatter plot of points:

```
p <- p + geom_point()
p
```



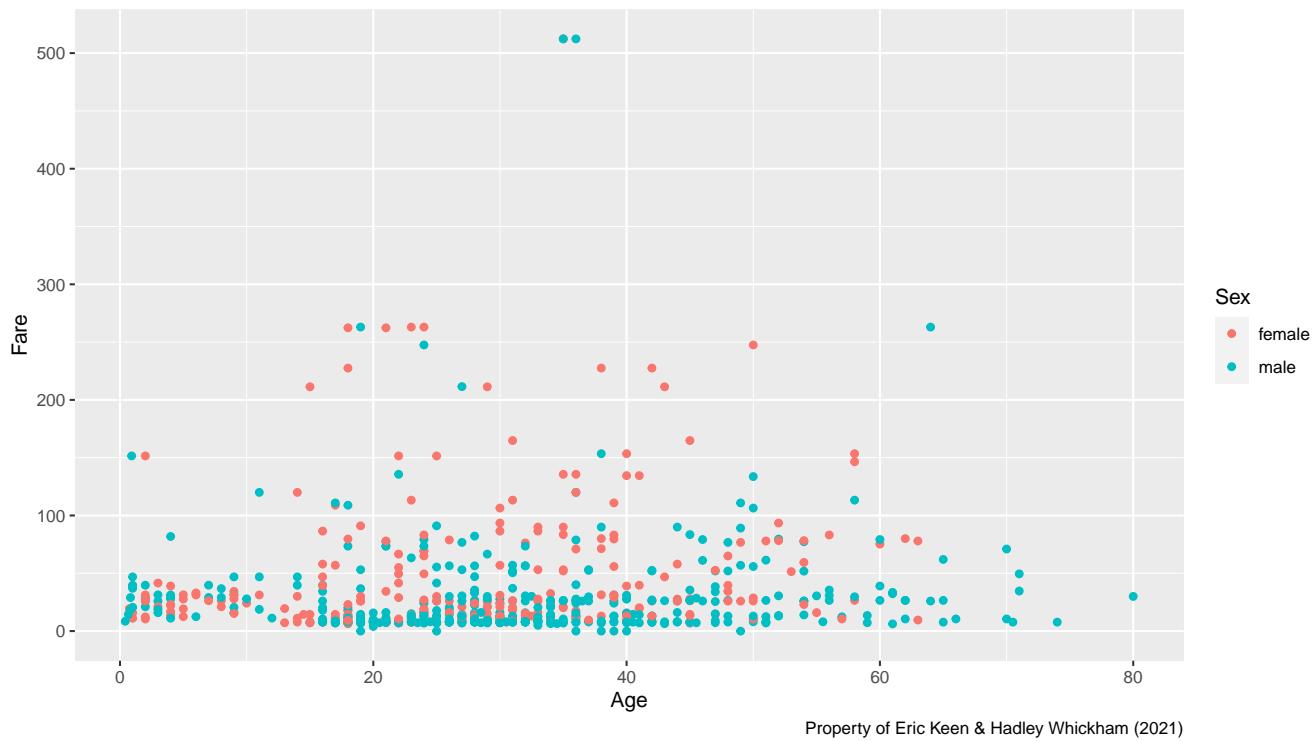
Note the use of a plus sign, `+`. You are *adding* layers to your plot.

- (5) **Add some more labels.** You see that `ggplot()` has automatically added axis titles and a legend, but we can add some more using the `labs()` function. Let's add an overhead title, a sub-title, and a caption.

```
p <- p + labs(title = 'Relationship between Age, Fare, and Sex',
               subtitle = 'Passengers on the RMS Titanic',
               caption = 'Property of Eric Keen & Hadley Wickham (2021)')
```

p

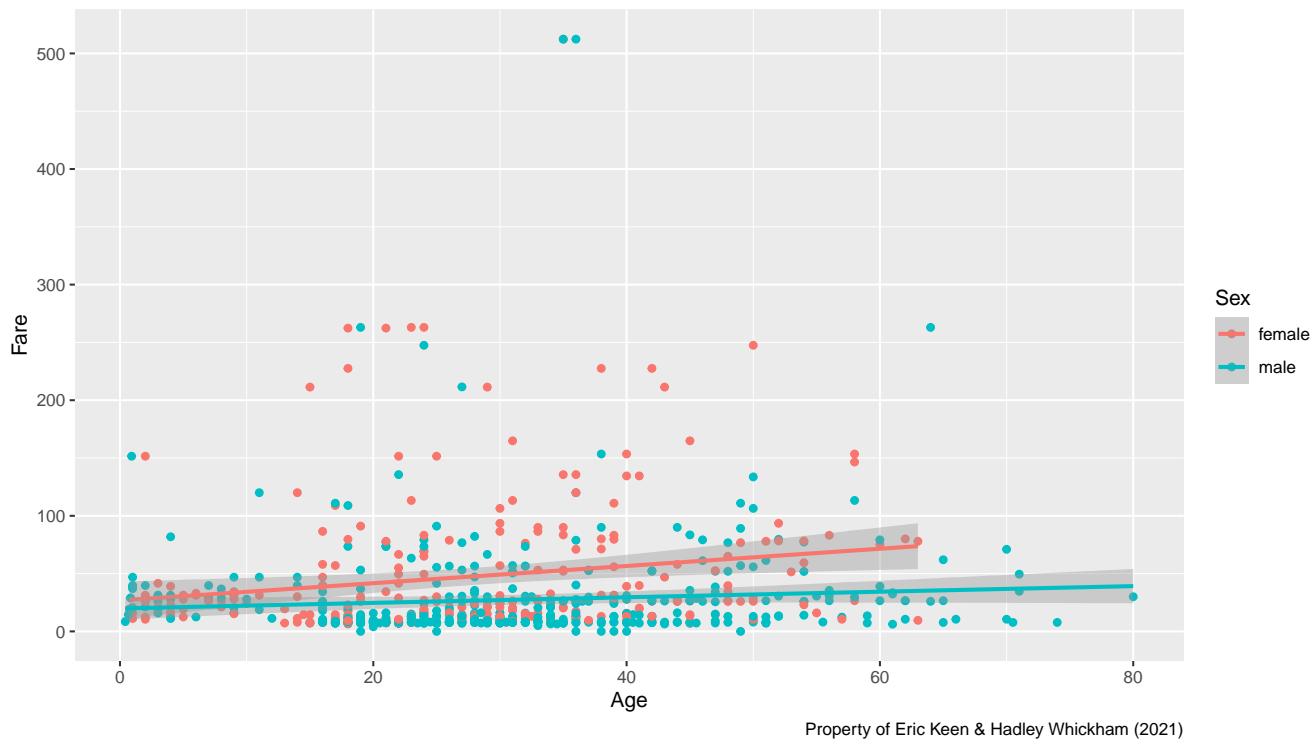
Relationship between Age, Fare, and Sex
Passengers on the RMS Titanic



(6) Add a statistical summary, like a smoothed regression line.

```
p <- p + geom_smooth(method = 'lm')
p
```

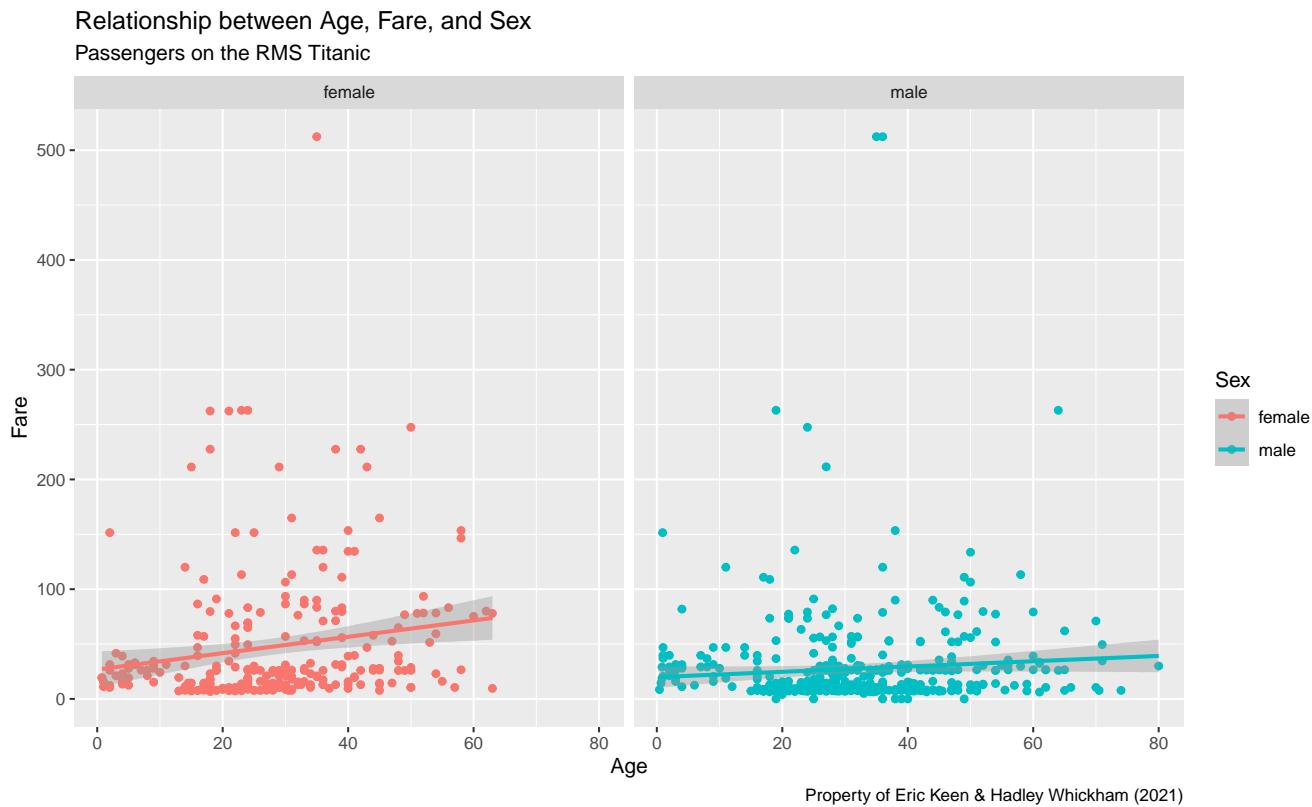
Relationship between Age, Fare, and Sex
Passengers on the RMS Titanic



Note that `ggplot()` automatically produced a different regression line for each sex. That's nice, but now our plot is getting pretty cluttered.

- (7) Clean up the look by using **facets**: a separate plot for each sex.

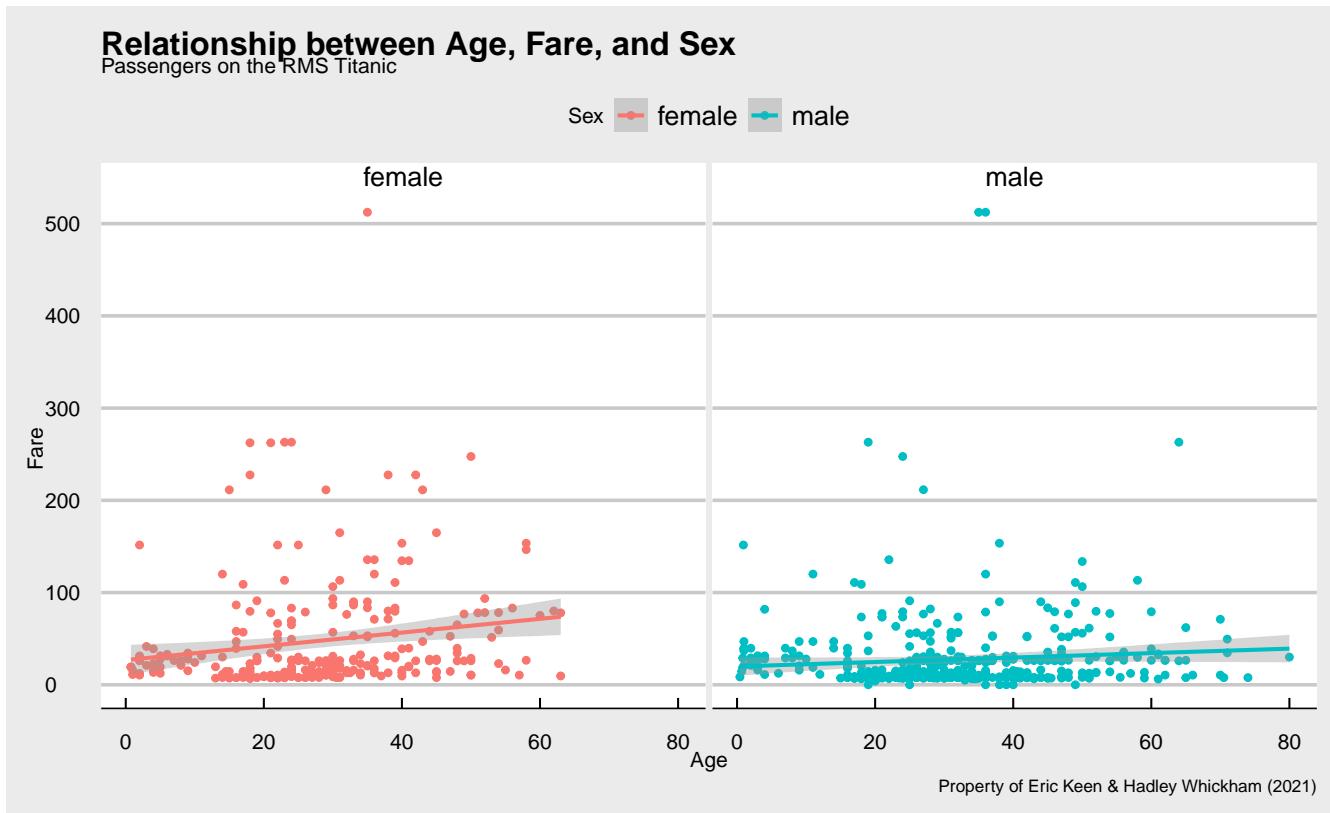
```
p <- p + facet_wrap(~Sex)
p
```



Beautiful!

- (8) Finally, let's **stylize** the entire plot with a different **theme**. You can find theme options in the 'ggthemes' package.

```
p <- p + theme_economist_white()
p
```

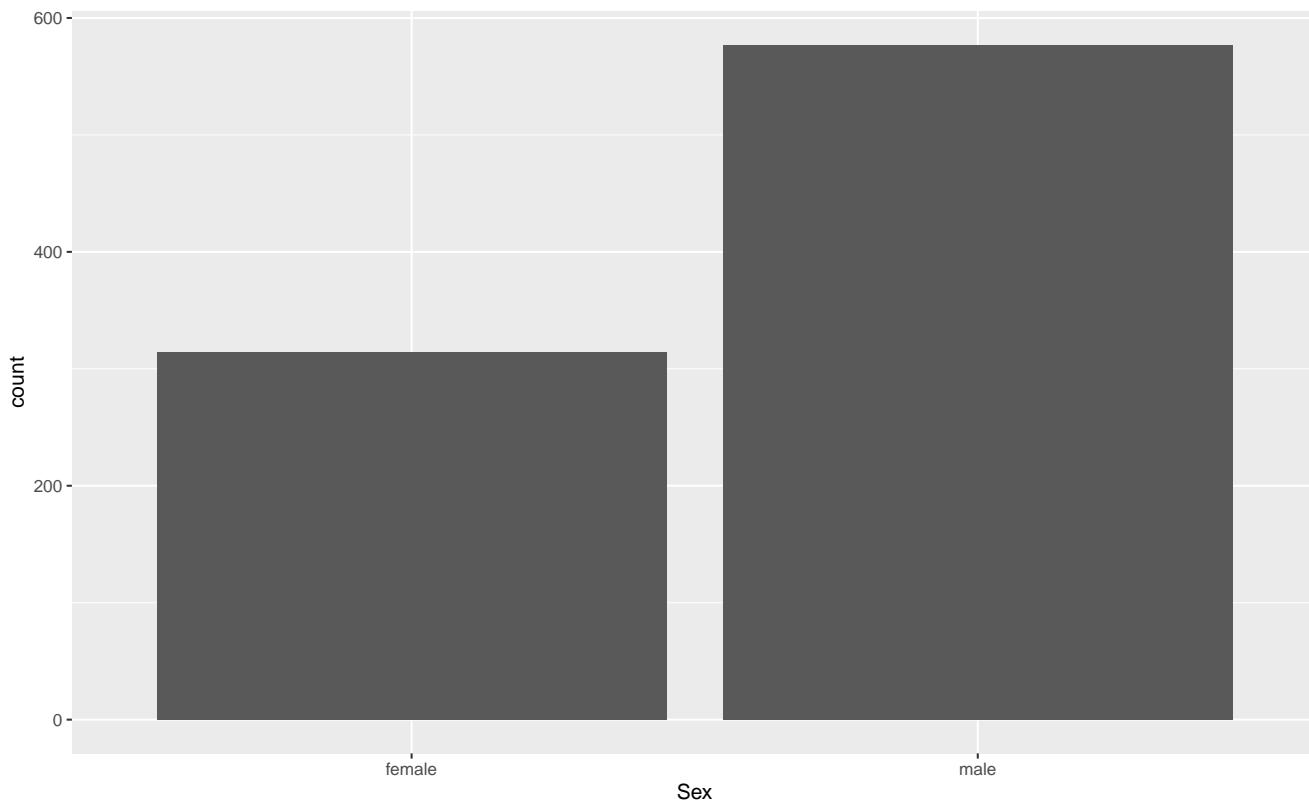


Bar plot

In a bar plot, your data are mapped to bars instead of points. And, instead of showing every data point, you are summarizing the data in some way – i.e., displaying a *statistic*. That statistic is usually just a count of the number of data points in each subgroup.

Let's make a bar plot that compares the number of men and women on the *Titanic*:

```
ggplot(data = titanic,
       aes(x = Sex)) +
       geom_bar(stat='count')
```



Note that, for the `aes()` call, we only provided the *x* axis attribute: `Sex`.

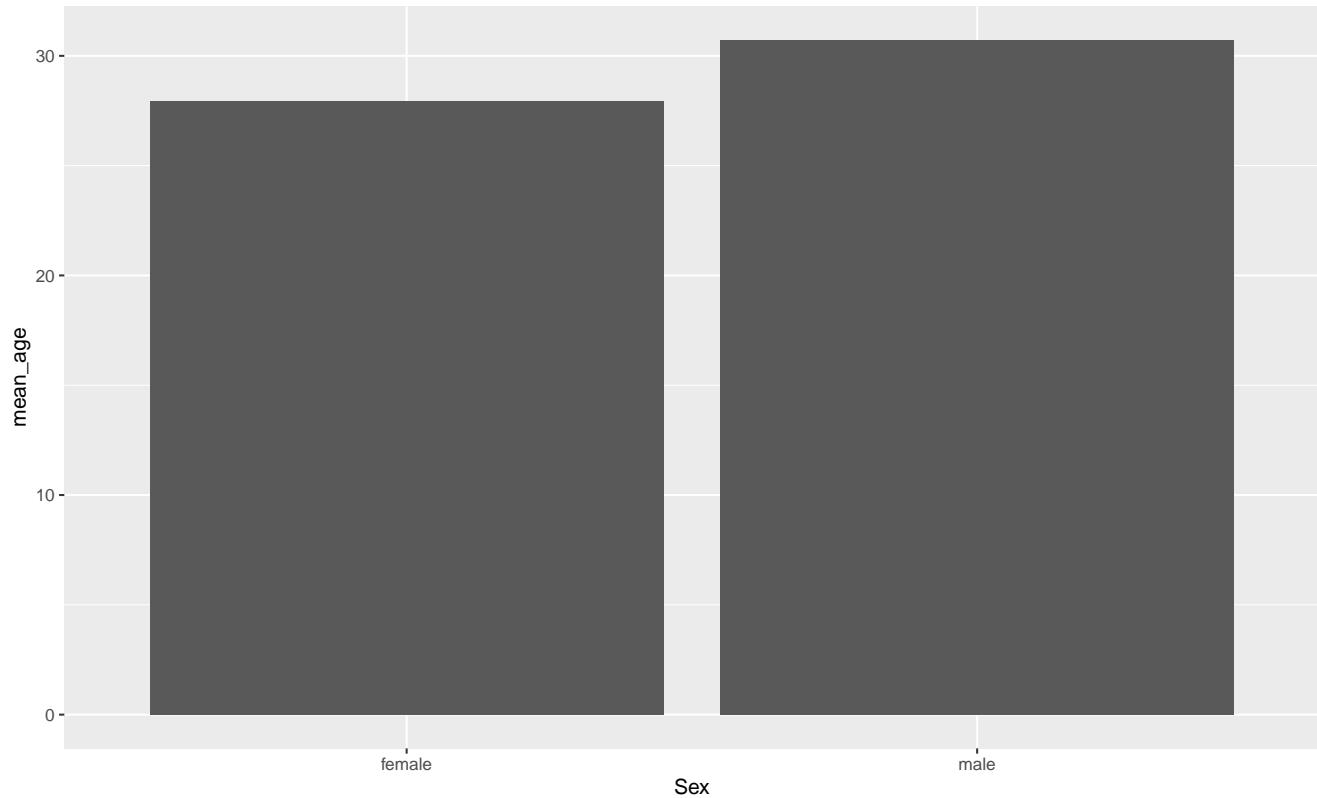
Then, in the `geom_bar()` call, we told R what `statistic` should be represented by that bars: '`count`'.

But you are allowed to explicitly set the bars' heights (i.e., the *y* dimension) to represent a different statistic. Let's say we wanted each bar to represent the mean age of men and women:

```
# First, determine the mean age of each sex
mean_age_males <- mean(titanic$Age[titanic$Sex == 'male'], na.rm = TRUE)
mean_age_females <- mean(titanic$Age[titanic$Sex == 'female'], na.rm = TRUE)

# Make a new dataframe with this summary data
titanic_age <- data.frame(Sex = c('male', 'female'),
                           mean_age = c(mean_age_males, mean_age_females))

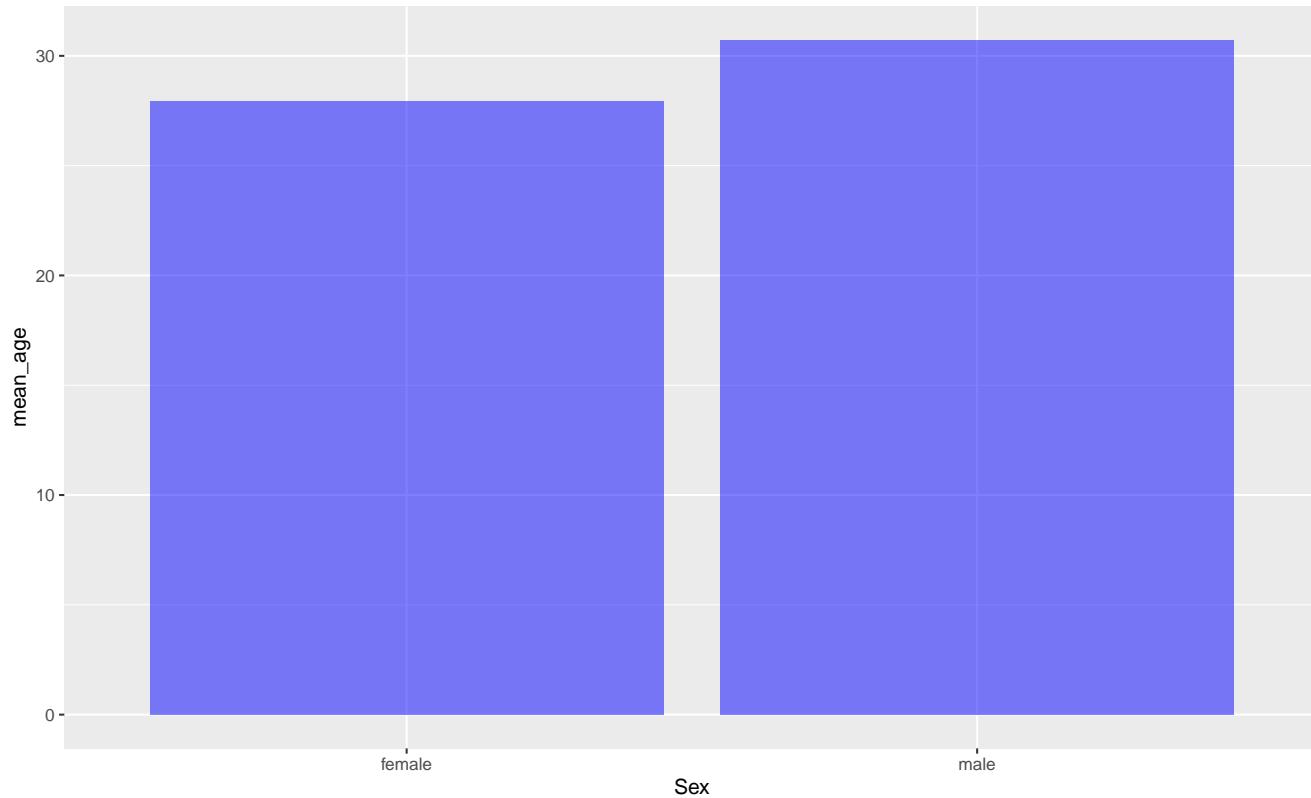
# Plot it
ggplot(data = titanic_age,
       aes(x = Sex, y = mean_age)) +
  geom_bar(stat = 'identity')
```



In this case, we are explicitly defining the y axis in the `aes()` call, and telling `geom_bar()` to just use the values we specified in `aes()` (that's what '`identity`' means; you are telling `ggplot()` to just use what you already gave it.)

You can specify other aesthetic attributes, *unrelated to the data*, within the `geom_bar()` call:

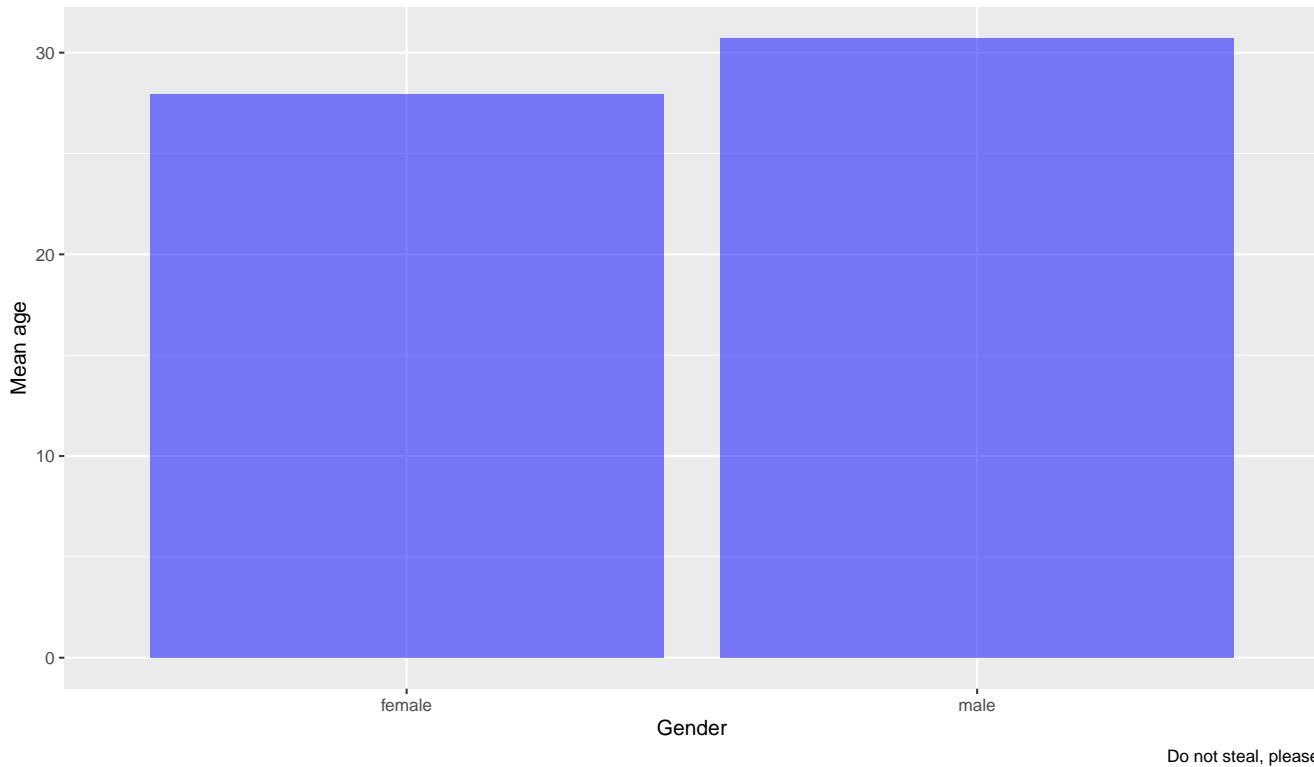
```
ggplot(data = titanic_age,
       aes(x = Sex, y = mean_age)) +
  geom_bar(stat = 'identity', fill = 'blue', alpha = 0.5)
```



Now add better labels:

```
ggplot(data = titanic_age,
        aes(x = Sex, y = mean_age)) +
  geom_bar(stat = 'identity', fill = 'blue', alpha = 0.5) +
  labs(y = 'Mean age',
       x = 'Gender',
       title = 'Titanic passengers: Average age by sex',
       caption = 'Do not steal, please')
```

Titanic passengers: Average age by sex



Do not steal, please

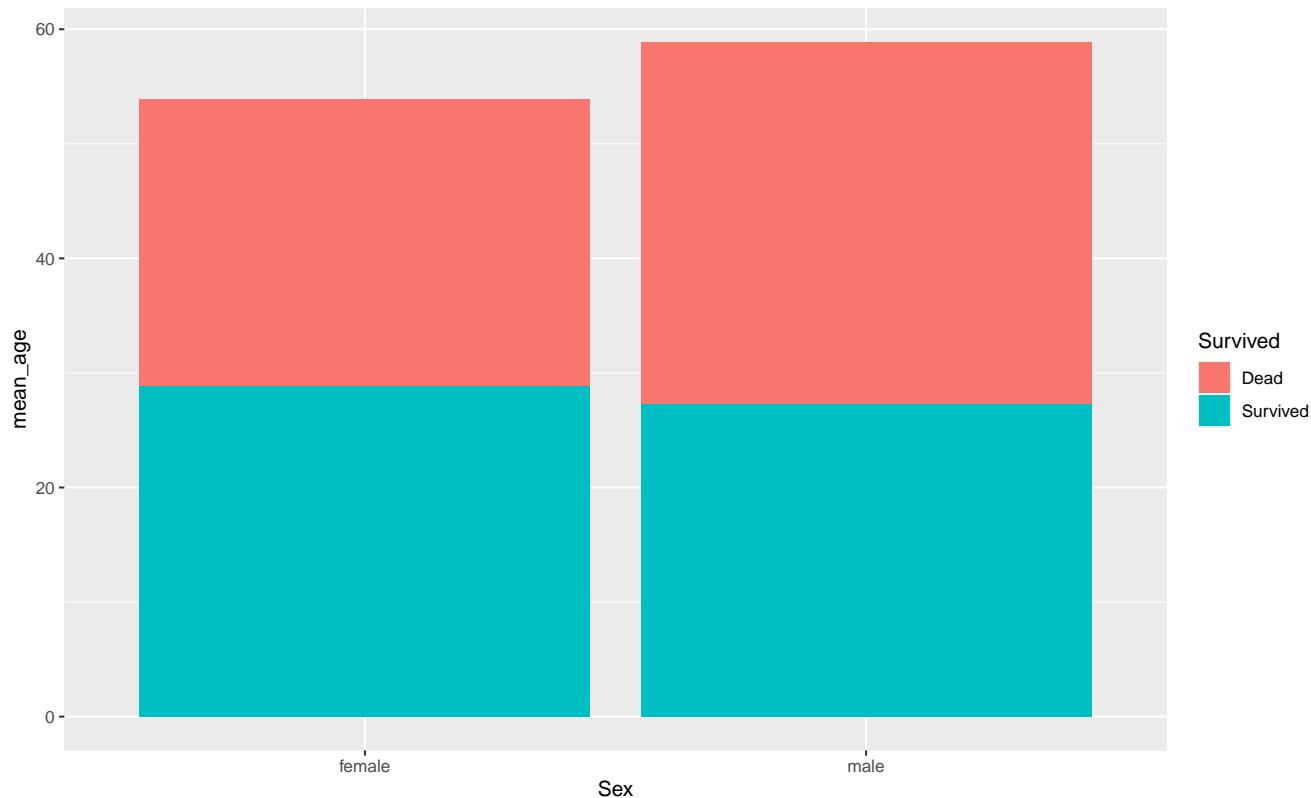
You can add another variable to your bar plot as follows. Let's say you want to see the average age in each sex, grouped by who survived and who didn't:

```
# First, produce your summary dataframe using some dplyr magic:
titanic_em <- titanic %>%
  group_by(Sex, Survived) %>%
  summarise(mean_age= mean(Age, na.rm = TRUE))

titanic_em <- titanic_em %>%
  mutate(Survived = ifelse(Survived == 1, 'Survived', 'Dead'))

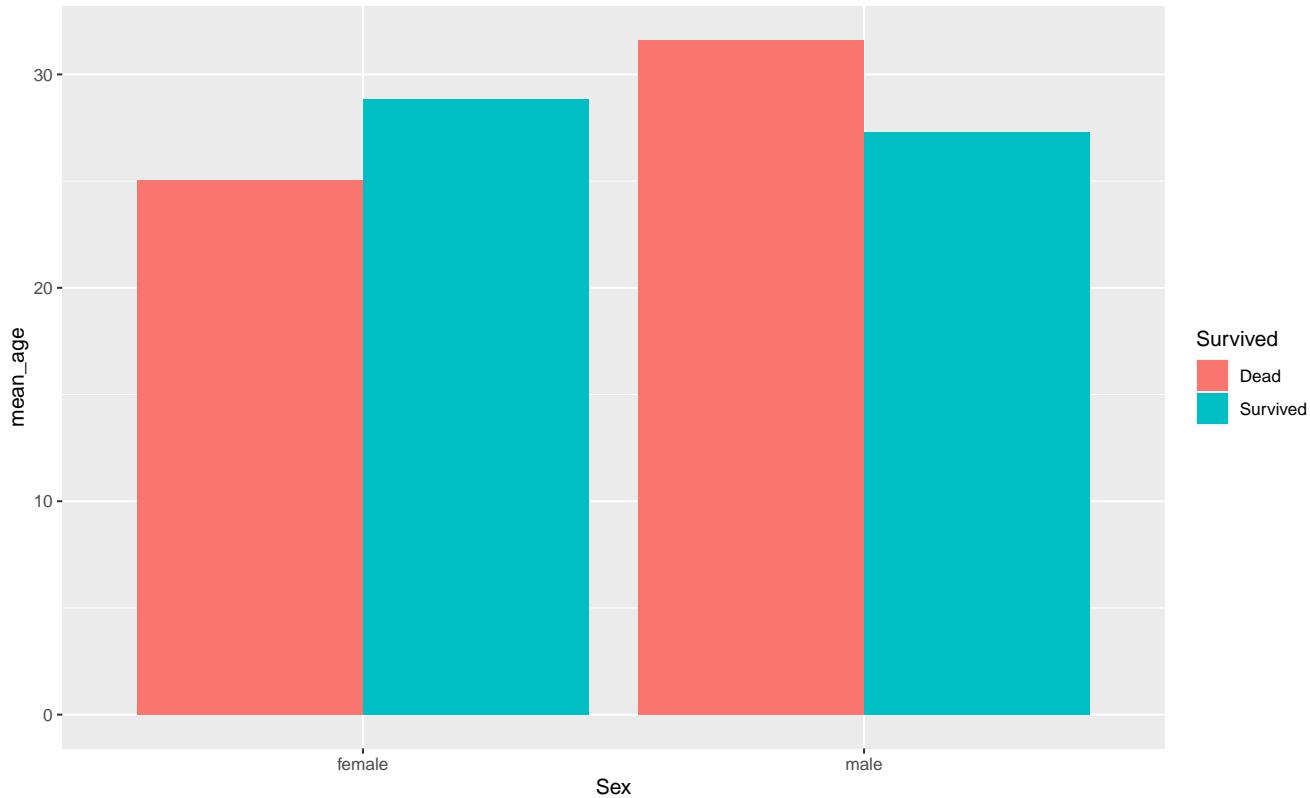
# Check it out
titanic_em
# A tibble: 4 x 3
# Groups:   Sex [2]
  Sex     Survived mean_age
  <chr>   <chr>      <dbl>
1 female  Dead        25.0
2 female  Survived    28.8
3 male    Dead        31.6
4 male    Survived    27.3

# Now plot it
ggplot(data = titanic_em,
       aes(x=Sex, y=mean_age, fill = Survived)) +
  geom_bar(stat='identity')
```



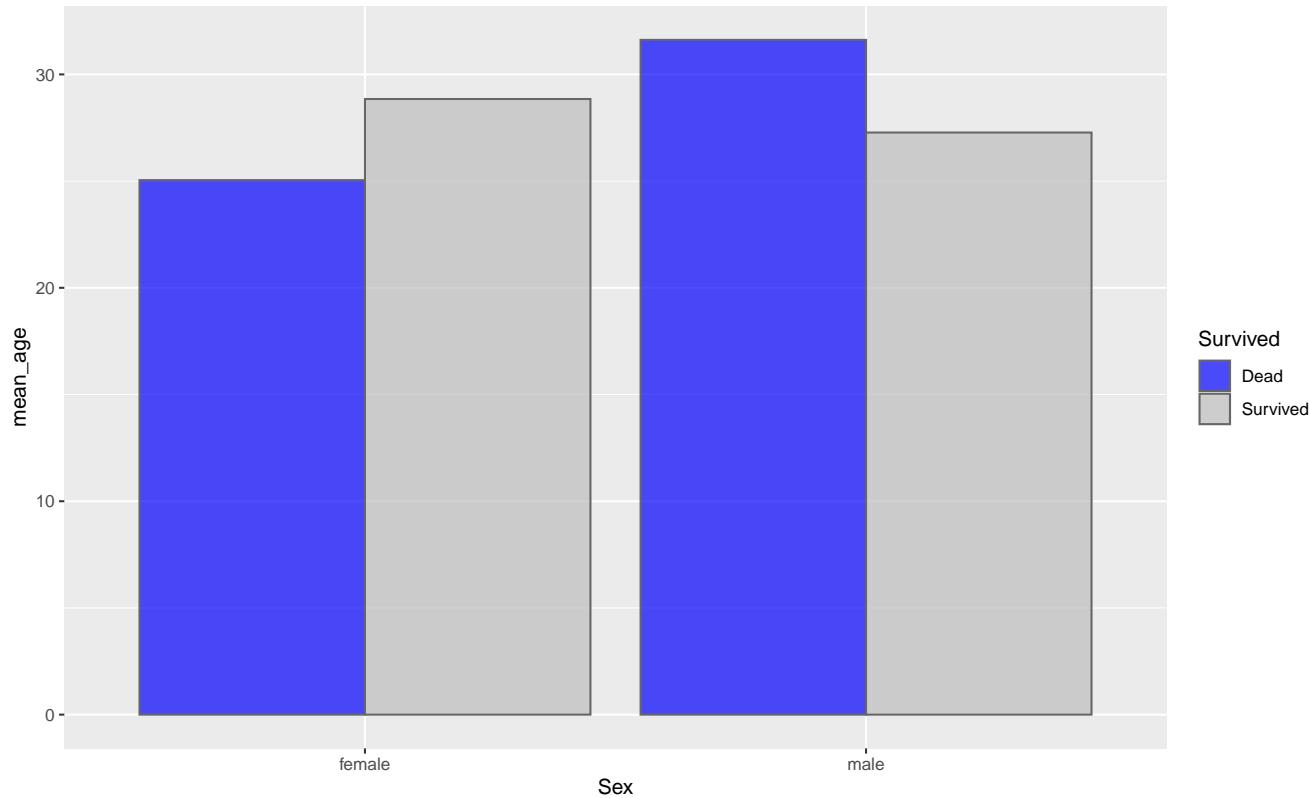
Rather than stack the bars, you can place them side by side:

```
ggplot(data = titanic_em,  
       aes(x=Sex, y=mean_age, fill = Survived)) +  
  geom_bar(stat='identity', position = 'dodge')
```



If you don't love these default colors (even if they are colorblind-friendly), you can manually define the colors for each group of bars:

```
ggplot(data = titanic_em,
       aes(x=Sex, y=mean_age, fill = Survived)) +
  geom_bar(stat='identity',
           position = 'dodge',
           alpha = 0.7,
           color='grey40') + # bar edge
  scale_fill_manual(values = c('blue', 'grey')) # bar fill
```



Exercises

More *Titanic* plots

1. Make a scatterplot similar to what you did above, but this time color-code by class instead of sex.
2. Notice that `ggplot()` automatically uses a continuous color scale for `Pclass`, since it has numeric values. To force `ggplot()` to consider `Pclass` as categories (1st class, 2nd class, 3rd class), replace `Pclass` with `factor(Pclass)`. Did the style of your color scale change?
3. Modify the title, subtitle, and caption to be more descriptive.
4. Produce a bar plot that compares the number of passengers in each class.
5. Make your bar plot as *ugly* as possible!
6. Now make it as *beautiful* as possible, including a concise but informative title, subtitle, and caption.

Baby names

Download the dataset on baby names given to newborns in the USA:

```
library(babynames)
Error in library(babynames): there is no package called 'babynames'
bn <- babynames
Error in eval(expr, envir, enclos): object 'babynames' not found
```

7. Create a line chart showing the number of girls named Mary over time.
8. Change the color of the line to blue.
9. Add a fitting title to the plot.
10. Create a bar chart showing the number of girls named Emma, Olivia, Ava, Sophia, and Emily in 2010.

- 11.** Change the X label to “Names” and the y label to “Total”. (*Hint:* check out the `labs()` help page.)
- 12.** Change the color of the bar to grey and make it more translucent.
- 13.** Create a bar chart showing the number of people named Emma, Olivia, Ava, Sophia, and Emily in 2010, colored by sex.
- 14.** Create a beautiful chart showing your name over time.

Chapter 15

Dataframe wrangling

Learning goals

- Understand the importance of *tidy* dataframes
- Understand what the `tidyverse` is and why it is awesome
- Feel comfortable working with dataframes using `dplyr` functions.

The `dplyr` package

Data scientists largely work in data frames and *do things* to data. This is what the package `dplyr` is optimized for. It consists of a series of “verbs” which cover 95% of what you need to do for most basic data processing tasks.

```
install.packages('dplyr') # if you haven't yet  
library(dplyr)
```

The `dplyr` package contains a set of **verbs**: things you do to dataframes. Those verbs are:

- `filter()`
- `arrange()`
- `select()`
- `rename()`
- `distinct()`
- `mutate()`
- `summarise()`

The `%>%` pipe

`%>%` is a “pipe”. It is a way to write code without so many parentheses. For example, what if I want to find the square root of the sum of the first six elements of a sequence of 10 to 20 by 2?

Here's what that command would look like in base R:

```
sqrt(sum(head(seq(10, 20, 2))))  
[1] 9.486833
```

Pretty overwhelming, and pretty easy to make errors in writing it out.

But the above could also be written a simpler way:

```
seq(10, 20, 2) %>% head %>% sum %>% sqrt
[1] 9.486833
```

When you see the `%>%` pipe symbol, think of the word “**then**”.

The above code could be read aloud like so: “First, get a sequence of every second number between 10 and 20. **Then**, take the first six values. **Then**, sum those samples together. **Then**, take the square root of that sum.”

Using the `%>%` pipe framework, your code turns from a nonlinear series of parentheses and brackets to a linear progression of steps, which is a closer fit to how we tend to think about working with data. Instead of working from the inside of a command outward, we thinking linearly: take the data, **then** do things with it, **then** do more things with it, etc.

Here’s another example:

```
mean(sd(1:100))
[1] 29.01149
```

... could also be written as:

```
1:100 %>% sd %>% mean
[1] 29.01149
```

dplyr verbs

To practice the `dplyr` verbs, let’s make a small dataframe named `people`:

```
people <- data.frame(who = c('Joe', 'Ben', 'Xing', 'Coloma'),
                      sex = c('Male', 'Male', 'Female', 'Female'),
                      age = c(35, 33, 32, 34))
people
#> #>   who     sex age
#> 1 Joe     Male 35
#> 2 Ben     Male 33
#> 3 Xing   Female 32
#> 4 Coloma Female 34
```

`filter()`

The `filter()` function is used to subset a dataframe, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions.

```
people %>% filter(sex == 'Male')
#> #>   who     sex age
#> 1 Joe     Male 35
#> 2 Ben     Male 33

people %>% filter(sex == 'Female')
#> #>   who     sex age
#> 1 Xing   Female 32
#> 2 Coloma Female 34
```

You can also filter according to multiple conditions. Here are three ways to achieve the same thing:

```
people %>% filter(sex == 'Female' & age < 33)
#> #>   who     sex age
#> 1 Xing   Female 32
```

```
people %>% filter(sex == 'Female', age < 33)
  who   sex age
1 Xing Female 32

people %>% filter(sex == 'Female') %>% filter(age < 33)
  who   sex age
1 Xing Female 32
```

Note that when a condition evaluates to NA, its row will be dropped. This differ from the base subsetting works with [...].

arrange()

Arrange means putting things in order. That is, `arrange()` orders the rows of a data frame by the values of selected columns.

```
people %>% arrange(age)
  who   sex age
1 Xing Female 32
2 Ben   Male  33
3 Coloma Female 34
4 Joe   Male  35

people %>% arrange(sex)
  who   sex age
1 Xing Female 32
2 Coloma Female 34
3 Joe   Male  35
4 Ben   Male  33

people %>% arrange(who)
  who   sex age
1 Ben   Male  33
2 Coloma Female 34
3 Joe   Male  35
4 Xing Female 32
```

To reverse the order, use `desc()`:

```
people %>% arrange(desc(age))
  who   sex age
1 Joe   Male  35
2 Coloma Female 34
3 Ben   Male  33
4 Xing Female 32
```

You can also arrange by multiple levels:

```
people %>% arrange(sex, age)
  who   sex age
1 Xing Female 32
2 Coloma Female 34
3 Ben   Male  33
4 Joe   Male  35
```

select()

Select only certain variables in a data frame, making the dataframe skinnier (fewer columns).

```
people %>% select(age)
#> #> age
#> #> 1 35
#> #> 2 33
#> #> 3 32
#> #> 4 34

people %>% select(sex, age)
#> #>   sex age
#> #> 1 Male 35
#> #> 2 Male 33
#> #> 3 Female 32
#> #> 4 Female 34
```

As you select columns, you can rename them like so:

```
people %>% select(sex, years = age)
#> #>   sex years
#> #> 1 Male 35
#> #> 2 Male 33
#> #> 3 Female 32
#> #> 4 Female 34
```

You can also select a set of columns using the : notation:

```
people %>% select(who:sex)
#> #>   who   sex
#> #> 1 Joe   Male
#> #> 2 Ben   Male
#> #> 3 Xing  Female
#> #> 4 Coloma Female
```

rename()

The function `rename()` changes the names of individual variables.

This verb takes the syntax `<new_name> = <old_name>` syntax.

```
people %>% rename(gender = sex, years = age, first_name = who)
#> #>   first_name gender years
#> #> 1 Joe   Male 35
#> #> 2 Ben   Male 33
#> #> 3 Xing  Female 32
#> #> 4 Coloma Female 34
```

mutate()

The function `mutate()` adds new variables and preserves existing ones.

New variables overwrite existing variables of the same name.

```
people %>% mutate(agein2020 = age - 1)
#> #>   who   sex age agein2020
#> #> 1 Joe   Male 35     34
#> #> 2 Ben   Male 33     32
#> #> 3 Xing  Female 32     31
#> #> 4 Coloma Female 34     33

people %>% mutate(is_male = sex == 'Male')
#> #>   who   sex age is_male
#> #> 1 Joe   Male 35     TRUE
```

```

2   Ben   Male  33    TRUE
3  Xing  Female 32   FALSE
4 Coloma Female 34   FALSE

people %>% mutate(average_age = mean(age))
  who      sex age average_age
1  Joe     Male 35      33.5
2  Ben     Male 33      33.5
3 Xing   Female 32      33.5
4 Coloma Female 34      33.5

```

You can call `mutate()` multiple times in the same pipe:

```

people %>% mutate(average_age = mean(age)) %>%
  mutate(diff_from_avg = age - average_age)
  who      sex age average_age diff_from_avg
1  Joe     Male 35      33.5        1.5
2  Ben     Male 33      33.5       -0.5
3 Xing   Female 32      33.5       -1.5
4 Coloma Female 34      33.5        0.5

```

You can also remove variables can be removed by setting their value to `NULL`.

```

people %>% mutate(age = NULL)
  who      sex
1  Joe     Male
2  Ben     Male
3 Xing   Female
4 Coloma Female

```

A similar function, `transmute()`, adds new variables and drops existing ones, kind of like a combination of `select()` and `mutate()`.

```

people %>% transmute(average_age = mean(age))
  average_age
1      33.5
2      33.5
3      33.5
4      33.5

```

group_by()

Most data operations are done on groups defined by variables. The function `group_by()` takes an existing table and converts it into a grouped one where operations are performed “by group”.

```

people %>%
  group_by(sex) %>%
  mutate(average_age_for_sex = mean(age))
# A tibble: 4 x 4
# Groups:   sex [2]
  who      sex     age average_age_for_sex
  <chr> <chr> <dbl>                <dbl>
1  Joe     Male    35                  34
2  Ben     Male    33                  34
3 Xing   Female  32                  33
4 Coloma Female 34                  33

```

```

people %>%
  group_by(sex) %>%
  mutate(average_age_for_sex = mean(age)) %>%

```

```

mutate(diff_from_avg_for_sex = age - average_age_for_sex)
# A tibble: 4 x 5
# Groups:   sex [2]
  who     sex     age average_age_for_sex diff_from_avg_for_sex
  <chr>  <chr> <dbl>             <dbl>                  <dbl>
1 Joe     Male    35            34                   1
2 Ben     Male    33            34                  -1
3 Xing   Female  32            33                  -1
4 Coloma Female  34            33                   1

```

Note that a similar verb, `ungroup()`, removes grouping.

`summarize()`

`summarize()` or `summarize()` creates an entirely new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarizing all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

```

people %>%
  summarize(average_age = mean(age))
  average_age
1      33.5

people %>%
  summarize(average_age = mean(age),
            standard_dev_of_age = sd(age),
            oldest_age = max(age),
            youngest_age = min(age))
  average_age standard_dev_of_age oldest_age youngest_age
1      33.5           1.290994       35          32

people %>%
  group_by(sex) %>%
  summarise(avg_age = mean(age),
            oldest_age = max(age),
            total_years = sum(age))
# A tibble: 2 x 4
  sex     avg_age oldest_age total_years
  <chr>  <dbl>     <dbl>        <dbl>
1 Female    33       34          66
2 Male      34       35          68

people %>%
  group_by(sex) %>%
  summarise(sample_size = n())
# A tibble: 2 x 2
  sex     sample_size
  <chr>      <int>
1 Female        2
2 Male          2

```

Note the use of the function, `n()`. This simple function counts up the number of records in each group.

Exercises

Answer these questions using the new `dplyr` verbs you just learned:

Baby names over time

1. Run the below code to load a dataset about baby names given in the USA since the 1800's.

```
library(dplyr)
library(babynames)
Error in library(babynames): there is no package called 'babynames'
bn <- babynames
Error in eval(expr, envir, enclos): object 'babynames' not found
```

2. Check out the first and last six rows of `bn`.
3. What are the names of the variables in this dataset?
4. How many rows are in this dataset?
5. What is the earliest year in this dataset?
6. Create a dataframe named `turn_of_century`, which contain data only for the year 1900.
7. Create a dataframe named `boys`, containing only boys.
8. Create a dataframe named `moms_gen`. This should be females born in the year of birth of your mom.
9. Order `moms_gen` by `n`, in ascending order (i.e., with the least popular name at top). Look at the result; what is the least popular name among women the year your mom was born?
10. Reverse the order and save the result into an object named `moms_gen_ordered`.
11. Create an object named `boys2k`. This should be all males born in the year 2000.
12. Arrange `boys2k` from most to least popular. What was the most popular boys name in 2000?
13. What percentage of boys were named `Joseph` in 2000?
14. Were there more Jims or Matthews in 2020?
15. Create an object named `tot_names_by_year`, which contains the total counts for boy and girl names assigned in each year of the dataset. You should have four columns: `year`, `boys`, `girls`, and `tot`.
16. How many people were born with *your* name in 2020?
17. Was your name more prevalent in 2020 than it was in the year you were born?
18. What if you account for the changing overall population size? In other words, is the *proportional prevalence* of your name greater in 2020 or your birth year?
19. In which year was your name the most prevalent?
20. Create a basic plot of the proportional prevalence of your name since the earliest year of this dataset.
21. Update this plot with lines for your parent's names and your siblings names, if you have any.
22. Format that plot so that it is gorgeous and well-labelled.
23. Screenshot it and email it to your family.

Part II

Reproducible research

Chapter 16

Markdown documentation

Learning goals

- How to document your projects with **Markdown**, and why doing so is so important.

This is a **meta-tutorial** for using **Markdown**, a “syntax” for writing *beautiful* reports with simple text files. This tutorial, which was written in **Markdown**, has been provided twice: first, as its formatted ‘book-ready’ form, and second as the raw text file used to write it.

What is **Markdown**?

Markdown lets you write documents formatted with fancy **HTML** (the main language used to create websites) without having to know how to write **HTML** code. Instead, you just need to know how to use a few common symbols.

Why?

Markdown was created because most data scientists used to document their code and data projects with simple text files (`.txt`). Text files are designed to be simple, so they have no special formatting: no nice fonts, no headings, no bold, no italics, no bullet lists. And, as a result of their simplicity, text files are (1) *boring* and (2) *difficult to organize*.

Markdown is the perfect solution: it translates a simple text file into a beautifully formatted report.

Recall that, in order for your research to be truly reproducible, you have to **document your work thoroughly**. Doing so requires two things: First, that you provide a detailed description of your work in a simple text file, and second, that your documentation be *well-organized* and *enjoyably legible*. Think about it: if your documentation is incomplete, impossible to navigate, and difficult to read, can you really say that your project is reproducible?

What can you do with **Markdown**?

First of all, **Markdown** ...

1. Lets you organize your report into sections and subsections.
2. Lets you *italicize* and **emphasize** certain sections of your work.
3. Lets you distinguish between normal text and `code` or `filenames`.
4. Lets you write large chunks of code, like the one below.

```
# This is a large chunk of code.  
x <- 1:10  
y <- 15:20  
z <- x + y
```

Also, Markdown ...

- Lets you weave together your documentation with R code that *actually runs* (this is called **RMarkdown**, which is covered in the next module), opening up possibilities for automatic reporting and truly reproducible research publications.
- Lets you produce publication-ready articles and books, since there is a world of open-source formatting templates available to you.
- Lets you add hyperlinks.
- Lets you type HTML code directly into the same document, if you want to.
- Lets you include images, like the one below.



Common Markdown mistakes

- When you try to make a section heading (e.g., `## New section`), you forget to put a space between the hashtags and the name of the section.
- When you try to begin a new paragraph, you don't add *two spaces* at the end of your sentence. (Without those two spaces, Markdown will assume you are still working in the same paragraph.)
- The same thing happens often when making a list. Without *two spaces* at the end of each item, the items might not get placed on new lines.
- When you try to begin a new paragraph, you don't add an empty line between your paragraphs.

Raw Markdown example

Below we provide the raw textfile we used to write the section above.

```
# `Markdown` documentation

##### Learning goals {-}

- How to document your projects with `Markdown`, and why doing so is so important.

&nbs;
```

This is a **meta-tutorial** for using `Markdown`, a "syntax" for writing **beautiful** reports with simple text files. This tutorial, which was written in `Markdown`, has been provided twice: first, as its formatted 'book-ready' form, and second as the raw text file used to write it.

```
## What is `Markdown`? {-}
```

`Markdown` lets you write documents formatted with fancy `HTML` (the main language used to create websites) without having to know how to write `HTML` code. Instead, you just need to know how to use a few common symbols.

```
### Why? {-}
```

`Markdown` was created because most data scientists used to document their code and data projects with simple text files (`.txt`). Text files are designed to be simple, so they have no special formatting: no nice fonts, no headings, no bold, no italics, no bullet lists. And, as a result of their simplicity, text files are (1) *boring* and (2) *difficult to organize*.

`Markdown` is the perfect solution: it translates a simple text file into a beautifully formatted report.

Recall that, in order for your research to be truly reproducible, you have to **document your work thoroughly**. Doing so requires two things: First, that you provide a detailed description of your work in a simple text file, and second, that your documentation be *well-organized* and *enjoyably legible*. Think about it: if your documentation is incomplete, impossible to navigate, and difficult to read, can you really say that your project is reproducible?

```
### What can you do with `Markdown`? {-}
```

First of all, `Markdown` ...

1. Lets you organize your report into sections and subsections.
2. Lets you *italicize* and **emphasize** certain sections of your work.
3. Lets you distinguish between normal text and `code` or `filenames`.
4. Lets you write large chunks of code, like the one below.

```

```
This is a large chunk of code.
x <- 1:10
y <- 15:20
z <- x + y
```
```

Also, `Markdown` ...

- Lets you weave together your documentation with `R` code that *actually runs* (this is called `RMarkdown`, which is covered in the next module), opening up possibilities for automatic reporting and truly reproducible research publications.
- Lets you produce publication-ready articles and books, since there is a world of open-source formatting templates available to you.
- Lets you add [hyperlinks](<https://www.markdownguide.org/getting-started/>).
- Lets you type `HTML` code directly into the same document, `if you want to`.
- Lets you include images, like the one below.

```
<center>
{width=50%}
</center>
```


Common `Markdown` mistakes {-}

- When you try to make a section heading (e.g., `## New section`), you forget to put a space between the hashtags and the name of the section.
- When you try to begin a new paragraph, you don't add *two spaces* at the end of your sentence. (Without those two spaces, `Markdown` will assume you are still working in the same paragraph.)
- The same thing happens often when making a list. Without *two spaces* at the end of each item, the items might not get placed on new lines.
- When you try to begin a new paragraph, you don't add an empty line between your paragraphs.

Exercises

Open a new, blank Markdown document:

1. In RStudio, go to the top left dropdown menu for making a new file (look for the green plus sign.)
2. Select “Text File”
3. Save this text file as `test.md`.

The prefix, `.md`, specifies that this text file should be interpreted as a **Markdown** file. Once you save the file, you should see some new options appear, one of which is preview.

4. At the top of your `test.md` file, write: `# Title`
5. Click the Preview button and see if a **Markdown** file is produced.

Make your own Markdown:

Use the **Markdown** example above, which you can compare to the formatted version at the top of this module, to figure out how to add the following elements to your document.

Use the *Preview* button to test if your entries are doing what you want them to.

6. Write a silly sentence.
7. Make a word in that sentence *italicized*.
8. Make a word in that sentence **boldface**.
9. Add the first section to your document. Name it whatever you want.
10. Add a sentence in this new section, formatting one of its words as a piece of `code`.
11. Start a new paragraph and add another sentence.
12. Start a third paragraph and add vertical space between it and the previous paragraph.
13. Add a multi-line chunk of code.
14. Download a silly meme from the internet and add it to your document. (*Hint*: you will have to place that image file in the same folder as your **Markdown** file.)

- 15.** Add another subsection to the subsection you are working in. Name it “Grocery List”.
- 16.** Type out an enumerated grocery list.
- 17.** Add another subsection, this time at the same hierarchichal level as the one you created in step 9. Name it “Bucket List”.
- 18.** Create a bullet list of things you want to accomplish before you die.
- 19.** Make one of your list items act as a link to another website.

Update your Git repo README with Markdown

This will be a review of both **Markdown** and **Git**!

- 20.** Navigate to your **GitHub** repo online.
- 21.** If you have not yet added a **README** file to your repo, click on the button that generates one for you.
- 22.** Click on the file to edit it.
- 23.** Add **Markdown** to your **README**, showcasing the tricks you haved learned in this exercise thus far. *Preview* your changes as you go and make sure everything looks how you intended.
- 24.** Commit your changes and check out your **README** on the homepage of your repo.

Make a Markdown CV

It is increasingly common for data scientists to have a **Markdown** version of their **CV** posted as its own page connected to their **GitHub** profile. Check out these examples:

- Here
- Also here
- And here!

Maintaining your **CV** this way can be very efficient. It is easy to format and easy to update. And, most importantly, rather than have various versions of your **CV** on your computer or in GoogleDrive, you can just share a *link* to your live, published **CV**. What better way to demonstrate your skillset in reproducible, professional-quality research?

- 25.** Use the instructions provided here to create a new repo for your **CV**, type up your **CV** in **Markdown**, and publish it for free on **GitHub Pages**.

This will also be a review of both **Markdown** and **Git**!

Resources

<https://www.markdownguide.org/basic-syntax/>

Chapter 17

R Markdown

Learning goals

- How to create RMarkdown documents
- Why they are awesome
- Why they are essential to truly *reproducible* reporting.

R Markdown is the combination of R and Markdown: you create a Markdown document that has R built-in to it: the two languages are woven, or “knitted”, together.

Do first, explain later

Let's not talk about it. Let's just do it.

Start an RMarkdown report

1. Open Rstudio.
2. Click File -> New File -> R Markdown
3. Fill out “Title”, “Author”, and click “Set to HTML”
4. Type Ctrl + S to save your file in your working directory.

Compile your report

5. Have a look at the document. What does each section mean?
6. Click the “Knit” button at the top.
7. Cool, right?

But also kind of boring. Let's make some changes to this report.

Modify the template

8. Let's replace the “setup chunk” with the below.
9. Then, let's change the title to some interesting question.
10. Now, let's delete all of the other stuff in the template so far (beginning at “R Markdown”).
11. Put in some section headers using Markdown. Make the following sections:

```
# Introduction
```

```
# Methods
```

```
# Results
# Discussion
# Bibliography
```

- 12.** Write a two-sentence introduction to your made-up research topic.
13. Knit again, to make sure that we did not break anything. (You should `knit` often!)

Good? Great.

Add an R chunk

In your *Methods* section, let's create an R chunk in which we can generate some fake data.

To make an R chunk, type this:

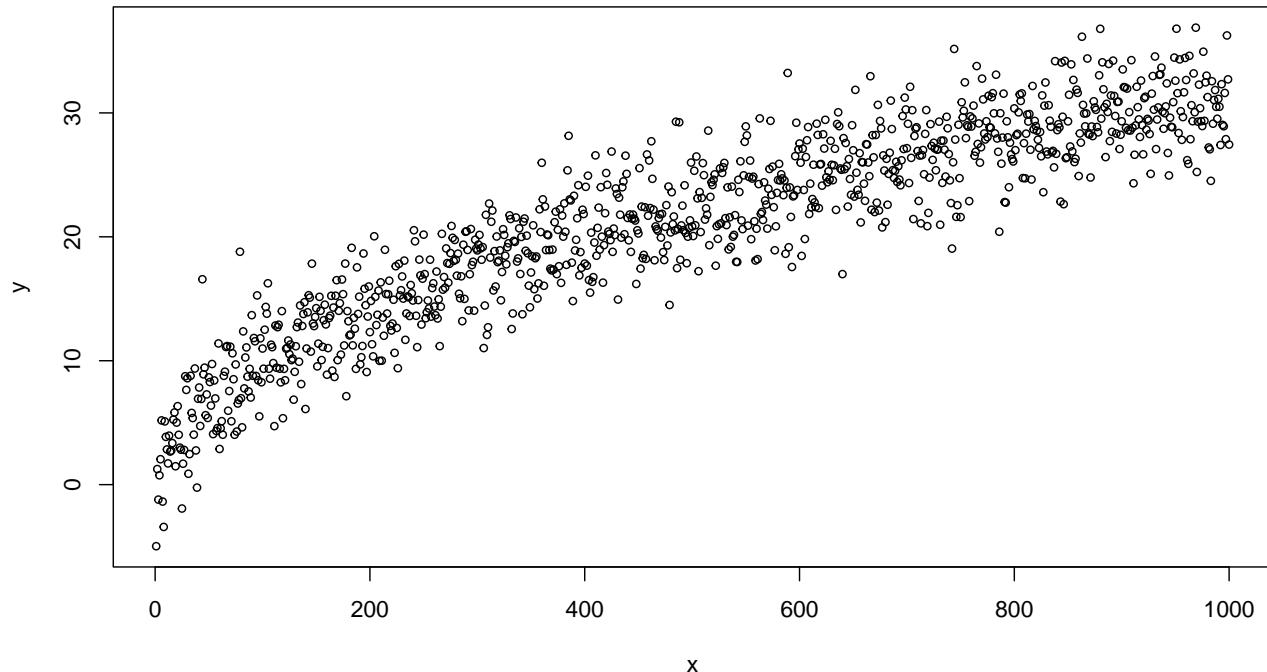
```
```r
code goes here
```

```

Those three marks represent the start and end of a chunk of code.

Now add this code into your chunk:

```
x <- 1:1000
y <- sqrt(x) + rnorm(1000, 0, sd=3)
plot(y ~ x, cex=.7)
```



Know, re-knit your doc. Everything work?

Chunk options

Hide a chunk

To run the code within an R chunk but hide the code in your final output, change the bracketed code at the top of your chunk to read `{r, echo = FALSE}`.

Don't evaluate a chunk

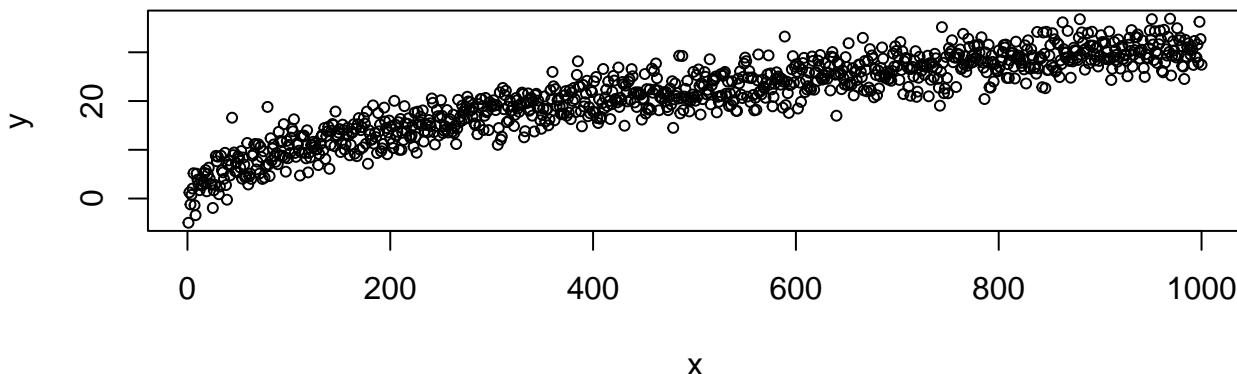
To skip over a chunk without evaluating it, change your brackets to look like this: `{r, eval = FALSE}`.

What if you want to hide a chunk *and* skip over it? `{r, echo = FALSE, eval = FALSE}`. This is a nice way to store code scraps that you don't want to run but aren't ready to get rid of.

Change figure size

To change the dimensions of a plot output, change your brackets to look like this: `{r, fig.height = 3, fig.width = 7}`

Doing so produces a plot like this:



Add figure caption

To add a figure caption, change your brackets like so: `{r, fig.cap = "A squished plot."}`

In-line R code

In your *Results* section, add this line:

Our analysis was based on `r length(x)` observations.

Know, re-knit your doc. What was printed before the word “observations”?

This is why RMarkdown is reproducible. Think of how a typical research report works: you get your results from R or whatever program you are using, then you manually write those results numbers into a *Word* document. What is reproducible about that?

With RMarkdown, you can let the data underlying your analysis *literally speak for themselves*.

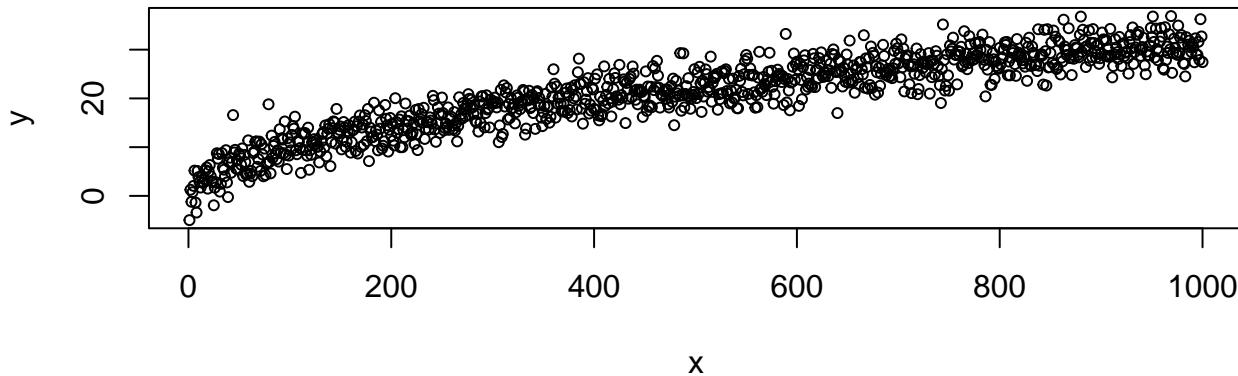


Figure 17.1: A squished plot.

Table 17.1: A pretty sweet table.

| x | y |
|---|------------|
| 1 | -4.9693761 |
| 2 | 1.2577974 |
| 3 | -1.2072758 |
| 4 | 0.7542793 |
| 5 | 2.0509106 |
| 6 | 5.1839579 |

Tables

To make a pretty table out of a data frame, create a chunk that looks like this:

```
library(knitr)
df <- data.frame(x,y)
kable(head(df), caption = "A pretty sweet table.")
```

Bibliographies

- In RStudio, create a new Text File and save it as `bibliography.bib` in the same location as your RMarkdown file.
- Look up this paper
- Copy the DOI.
- Go to <https://doi2bib.org/>
- Copy the `bibtex` format
- Paste into your `.bib` file.

Note that the top line of this `bib` entry has the phrase `Rudd2014`. This is known as the citation's “key” – you can use this phrase to reference this source within your Markdown doc.

- Back in your RMarkdown doc, in your *Discussion* section, write some line about a paper.

- At the end of that sentence, add this text: ([@Rudd2014]).
- Add `bibliography: bibliography.bib` to your “yaml”, the header information at the very top of your RMarkdown file. Change the `yaml` so that it looks like this:

```
---
```

```
output: html_document
bibliography: bibliography.bib
---
```

- Re-knit your doc to see what happens. Was the citation automatically generated?

Exercises

Before getting started, think of a testable hypothesis which can be answered via a survey of your friends and classmates. For example: “woman have more dreams than men”, or “people born abroad have shorter last names than people born in the USA”, or “shoe size is associated with GPA”.

1. Create a repository on github called `report`.
2. Clone that repository to your local machine.
3. `cd` into that repository.
4. In that repo, create a new Rmarkdown document. Name it `paper.Rmd`.
5. Replace the setup chunk with the one used in the in-class exercise (above).
6. Create a *GoogleForms* survey with at least 5 interesting questions on a topic which interests you (one topic, five questions).
7. Get at least 5 people to fill out your survey.
8. In your *GoogleForms* settings, create a *GoogleSheet* of your results.
9. In the sharing settings of your *GoogleSheet*, set to “public”.
10. Copy the URL of your *GoogleSheet*.
11. Create an R chunk in your `paper.Rmd`.
12. Use the `gsheet` package to read your survey results as a dataframe.
13. After reading in the data, create the following sections in your report: `Introduction`, `Methods`, `Results`, `Conclusion`.
14. Spend 5-15 minutes on the internet reading about your topic / hypotheses.
15. Find 2-5 authors/articles of interest related to your topic. Cite them in the *Introduction*.
16. Describe in plain English your approach to testing your hypothesis in the *Methods* section.
17. In your *Methods* section, provide a URL link to your survey.
18. After the *Methods* section, write the R code (in an `echoed` chunk) which tests your hypothesis.
19. In the *Results* section, write a description of both (a) your sample and (b) your results. Include at least two charts. Include at least one table.
20. In the *Conclusion* section, write interesting reflection on your data.
21. Got here? Cool, you’re fast. Now configure your computer to generate pdfs via Rmarkdown: mac / windows

- 22.** Make your paper in pdf.
- 23.** Check out these theme options for Rmarkdown docs.
- 24.** Push your code to git
- 25.** Check out how to make presentations in Rmarkdown. Make one.
- 26.** Check out how to make dashboards in Rmarkdown. Make one.
- 27.** Change your citation style. To do so, use a new csl from this site.

Chapter 18

Interactive maps

Learning goals

- Learn how to create interactive, *beautiful* maps with `leaflet`.
- Understand why `leaflet` is so awesome and valuable in *reproducible* data science.

Mapping with `leaflet`

If you know about the right packages, making really nice maps in R is actually pretty easy. R's spatial visualization tool set is one of the things that elevates it above the other open-source coding tools out there.

Being able to make a map programmatically is a powerful skill. And we can do it quickly and easily in R thanks to the package `leaflet`.

Let's load it up:

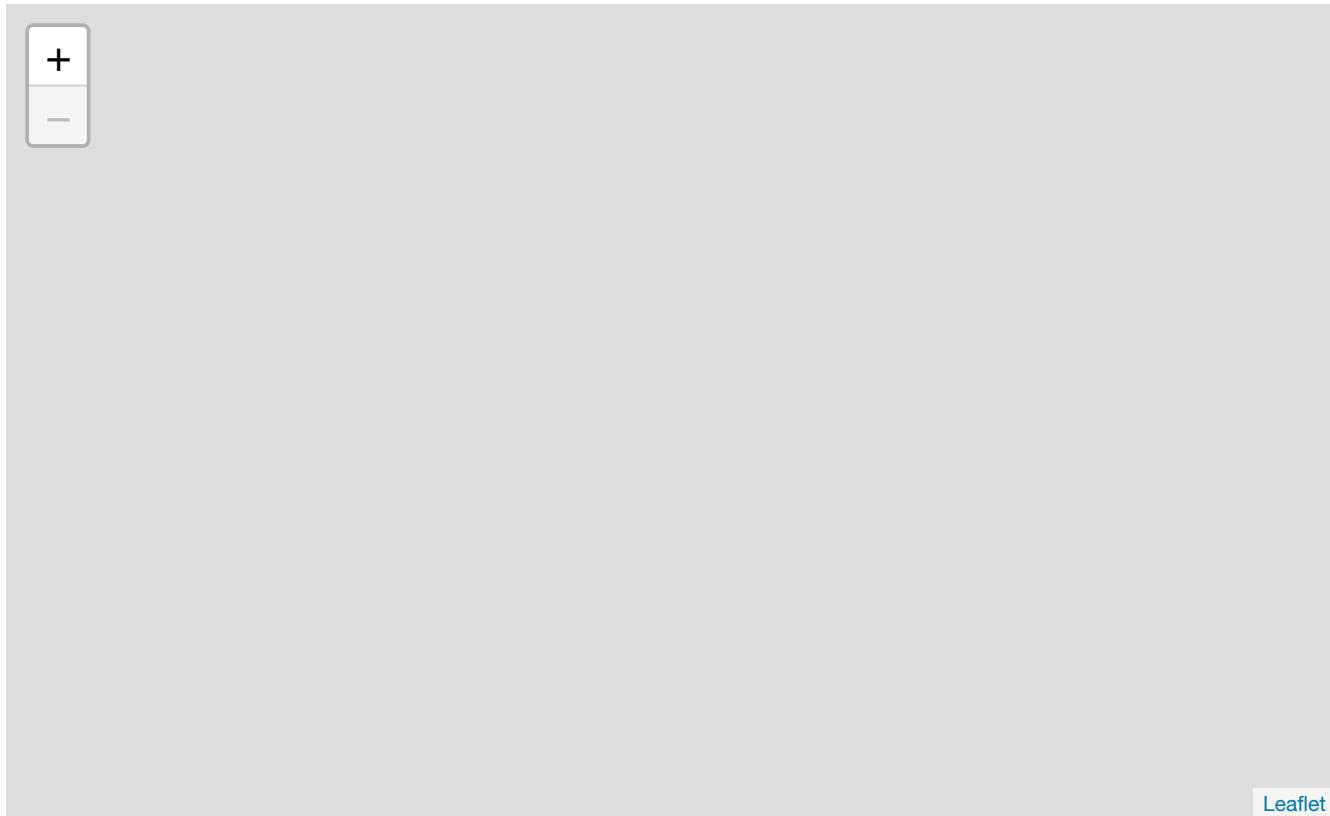
```
install.packages('leaflet')  
  
library(leaflet)  
  
# Some other packages too:  
library(dplyr)  
library(ggplot2)
```

To practice making a map, let's focus in on the location of Sewanee: The University of the South, in Tennessee, USA.

```
sewanee <- data.frame(longitude=-85.9211,  
                      latitude=35.2031)
```

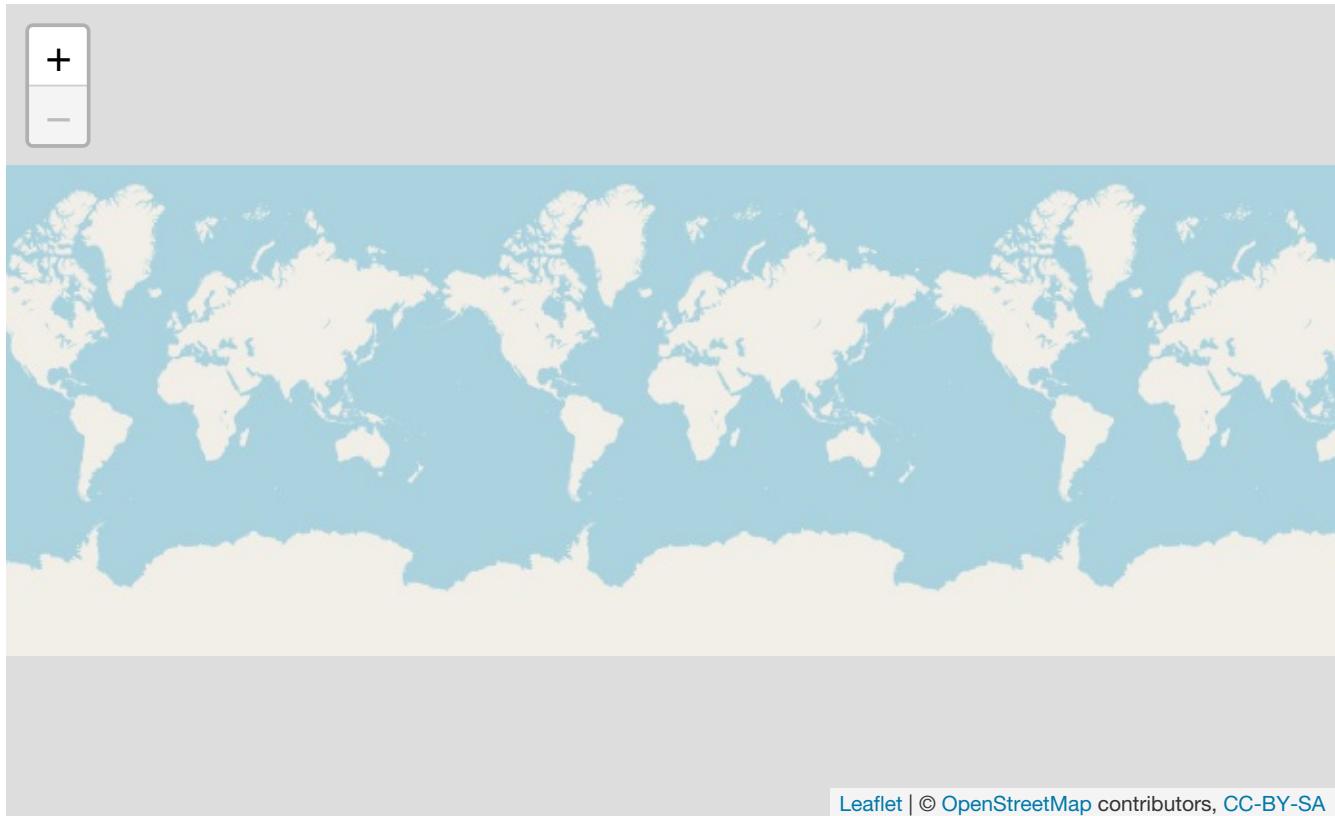
To begin a `leaflet` map, simply type `leaflet()`. This creates a blank canvas.

```
leaflet()
```



Now, add the background map. To do so, `leaflet` uses “tiles”. Tiles are basically images – little pictures stitched together to give you a sense of a continuous map.

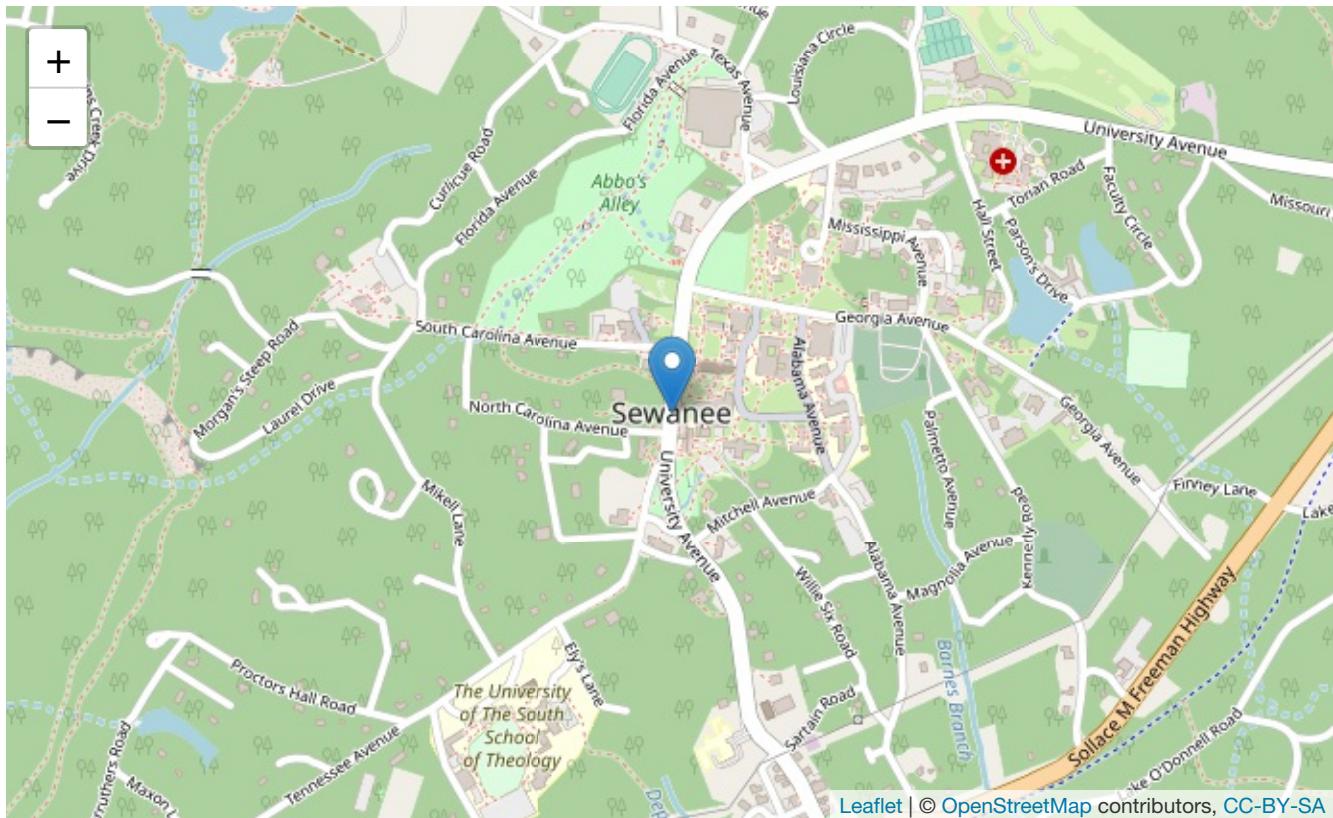
```
leaflet() %>% addTiles()
```



Try zooming into this map to get a sense of how powerful these interactive `leaflet` maps can be.

To get the map focused on a specific area of interest, bring in “markers” based on your data:

```
leaflet() %>%
  addTiles() %>%
  addMarkers(data=sewanee)
```

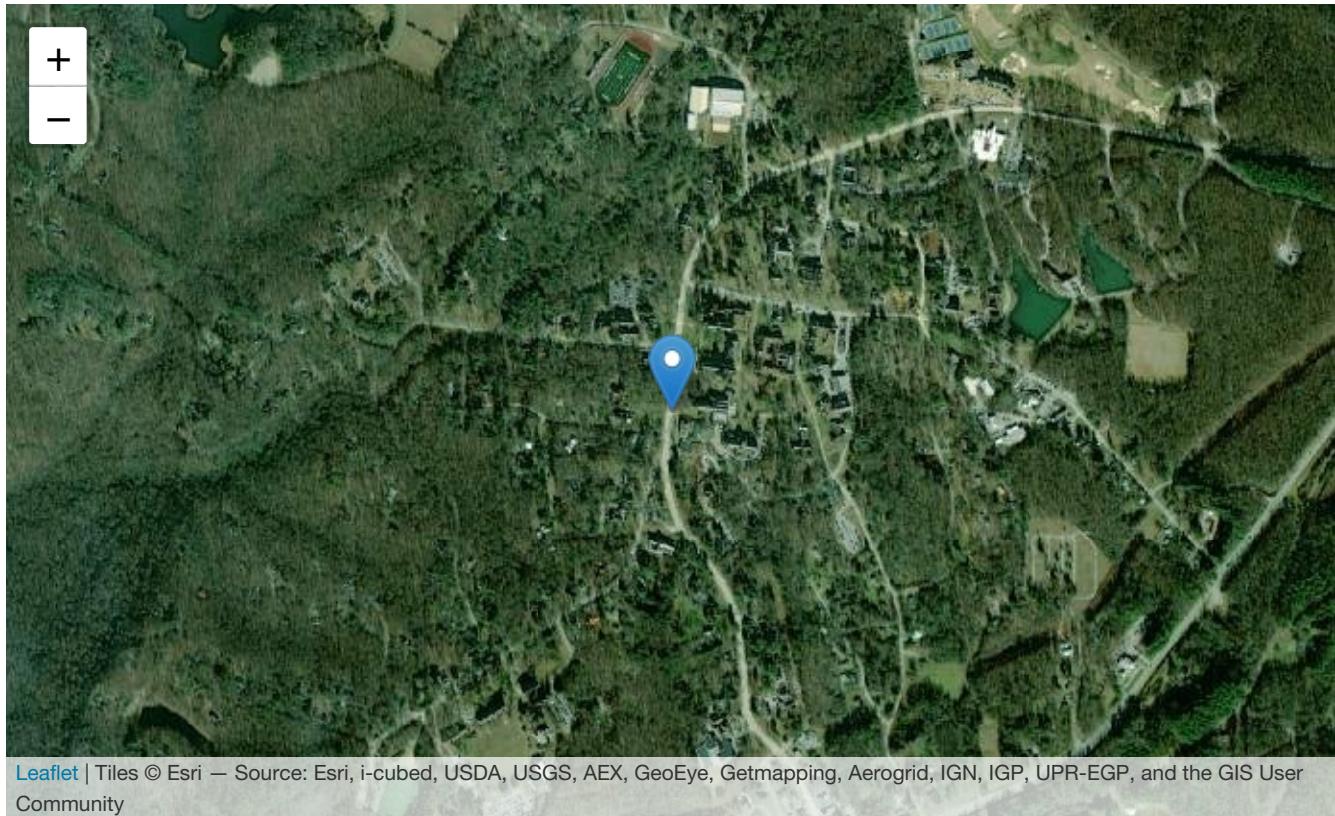


Ta-dah! There's your map. Try zooming in & out again.

Styling your leaflet map

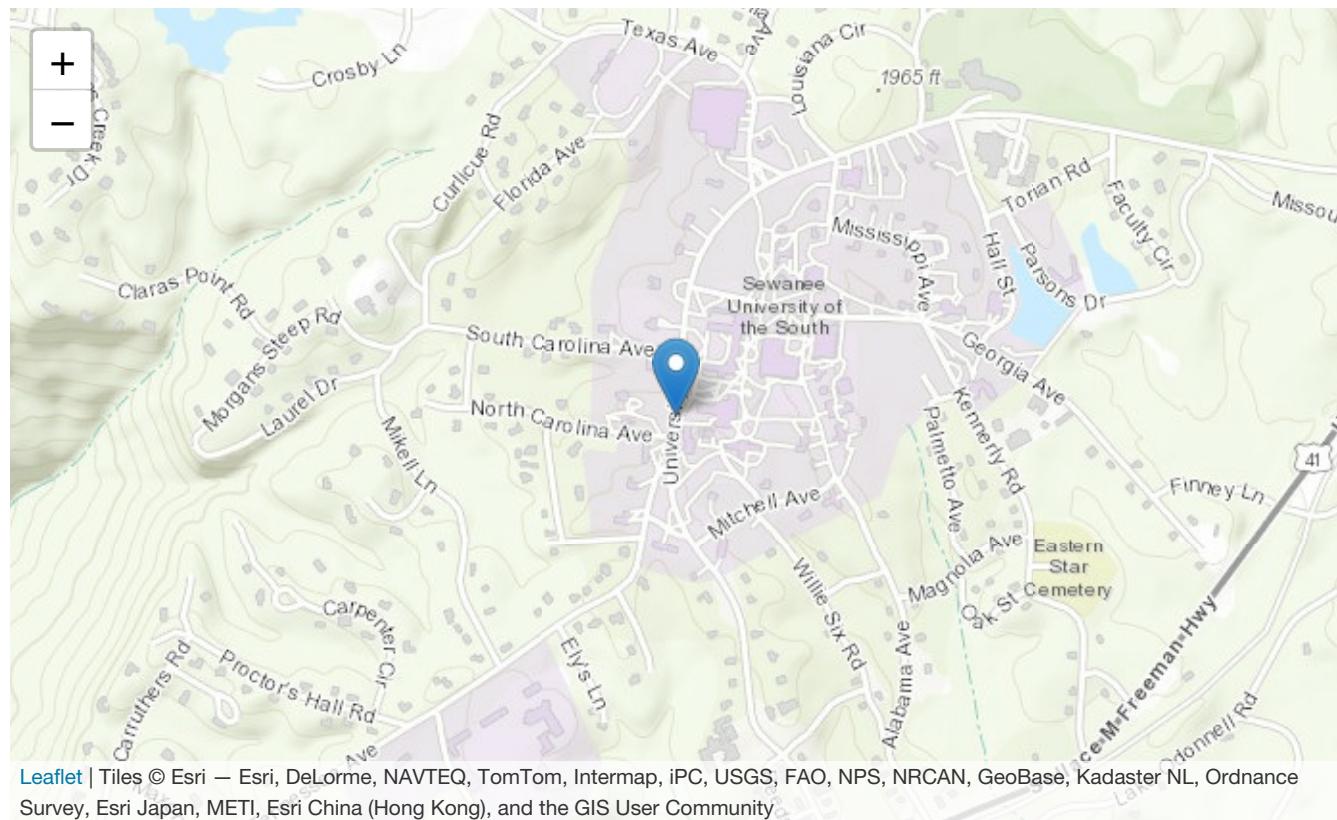
How about satellite imagery instead?

```
leaflet() %>%  
  addProviderTiles(providers$Esri.WorldImagery) %>%  
  addMarkers(data=sewanee)
```



How about topography instead?

```
leaflet() %>%
  addProviderTiles(providers$Esri.WorldTopoMap) %>%
  addMarkers(data=sewanee)
```

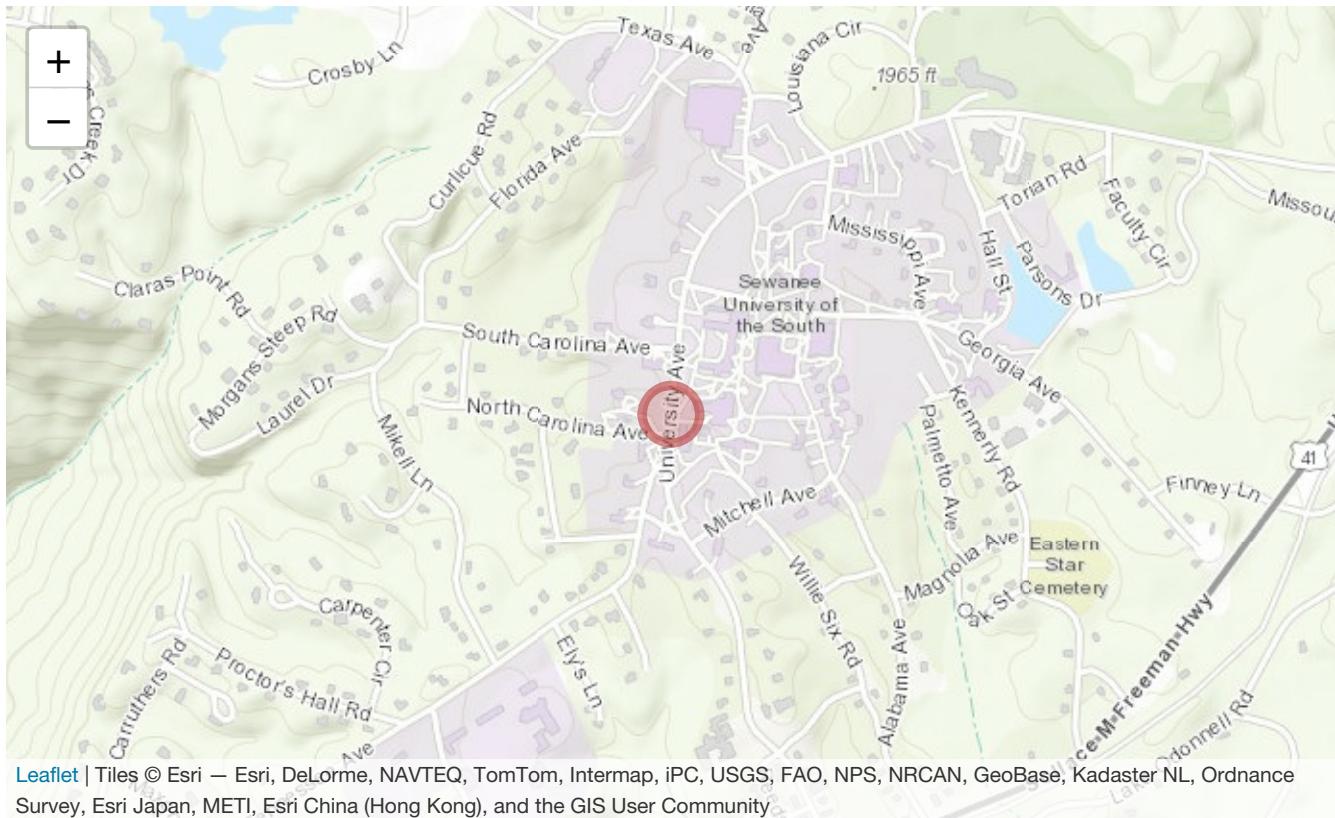


The `leaflet` package has plenty of other tile styles available through the `addProviderTiles()` function. To preview all the options, check out this site.

Working with markers in `leaflet`

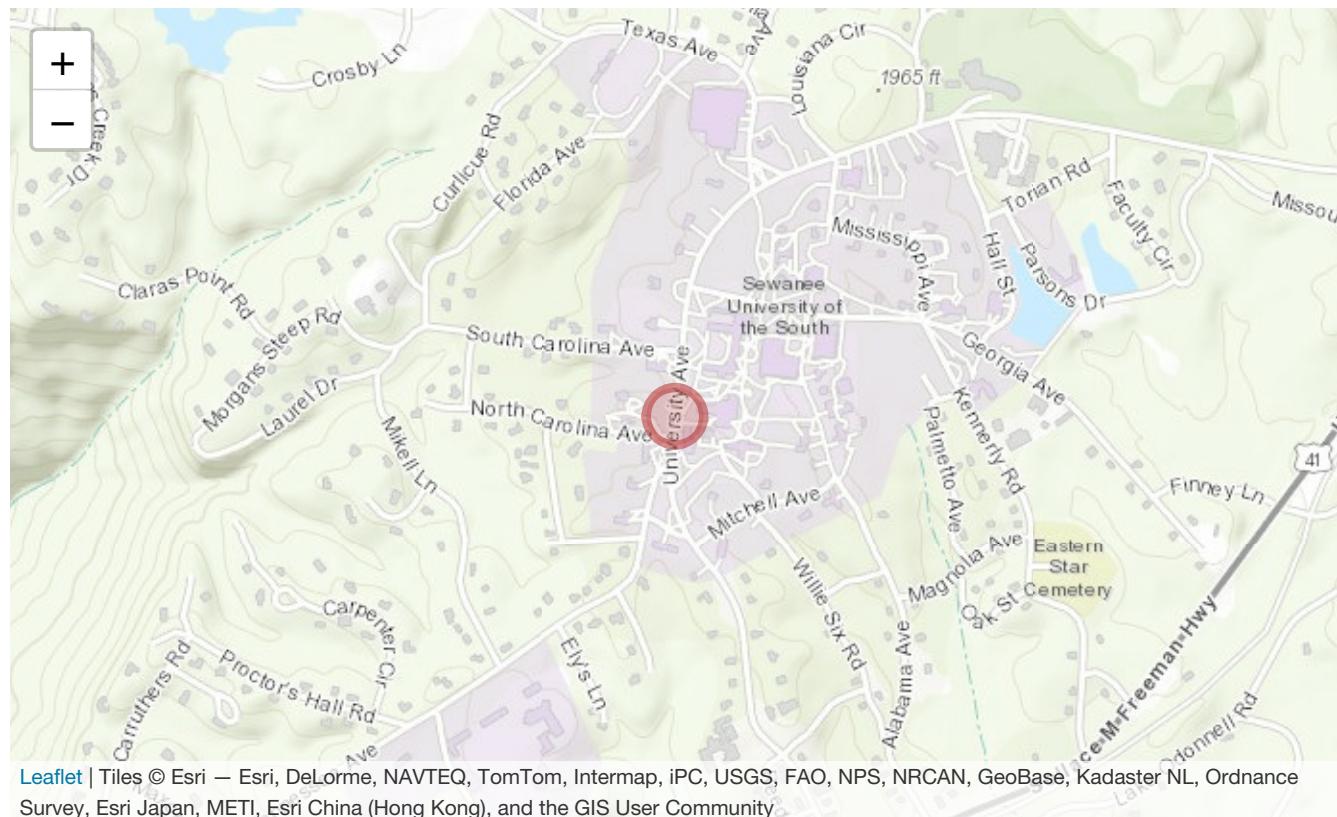
To change the markers from a “dart” to a circle, use `addCircleMarkers()` instead:

```
leaflet() %>%
  addProviderTiles(providers$Esri.WorldTopoMap) %>%
  addCircleMarkers(data=sewanee,
    color = "firebrick",
    radius = 15)
```



To add ‘pop up’ information when you click on your marker, use the `popup` input:

```
leaflet() %>%
  addProviderTiles(providers$Esri.WorldTopoMap) %>%
  addCircleMarkers(data=sewanee,
    color = "firebrick",
    radius = 15,
    popup = paste0('<b>YSR, yall.</b> <br>',
      'Lat: ', sewanee$longitude, '<br>',
      'Lon: ', sewanee$latitude, '\n'))
```



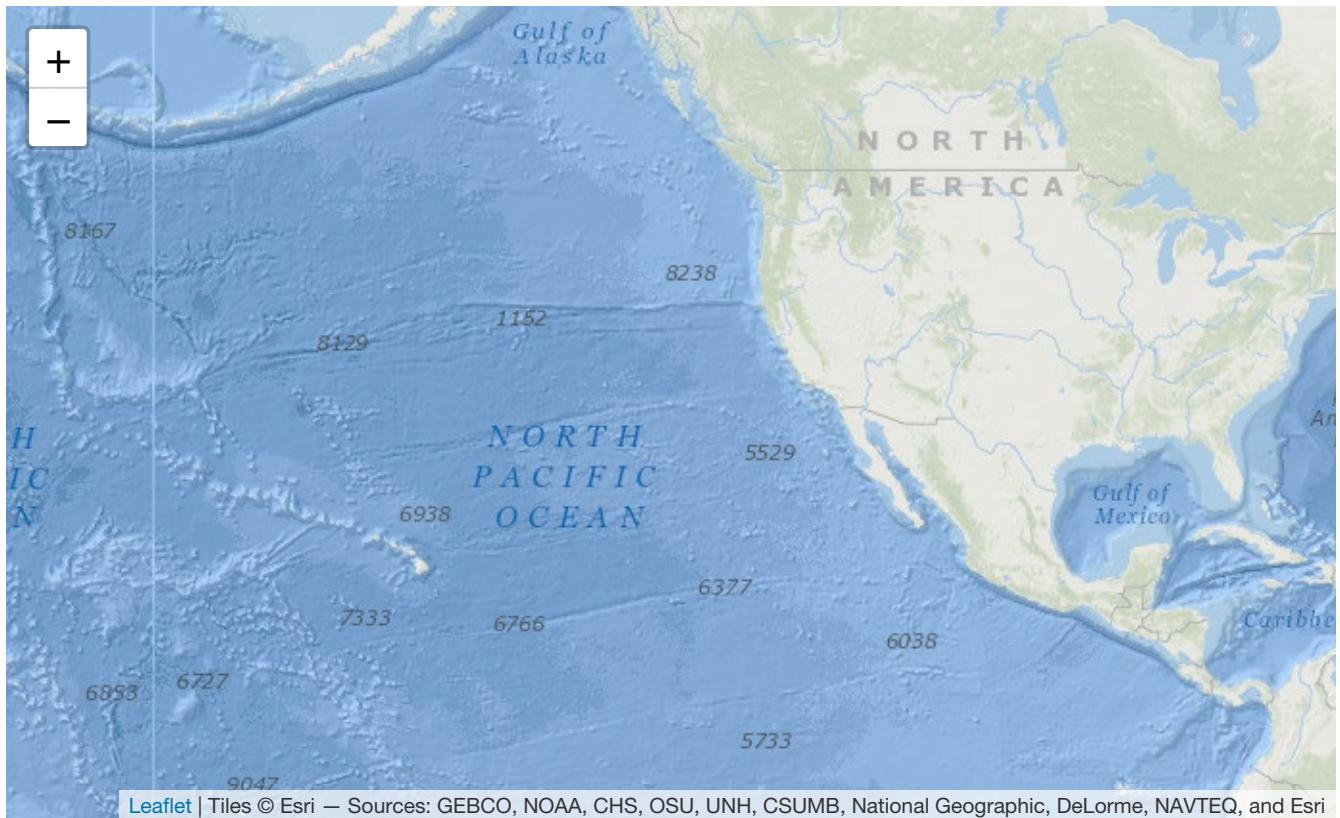
Click on the marker and see how it looks! Note that we used a bit of HTML to make the text look prettier.

Mapping marine areas

Note that `leaflet` is also useful in mapping marine areas.

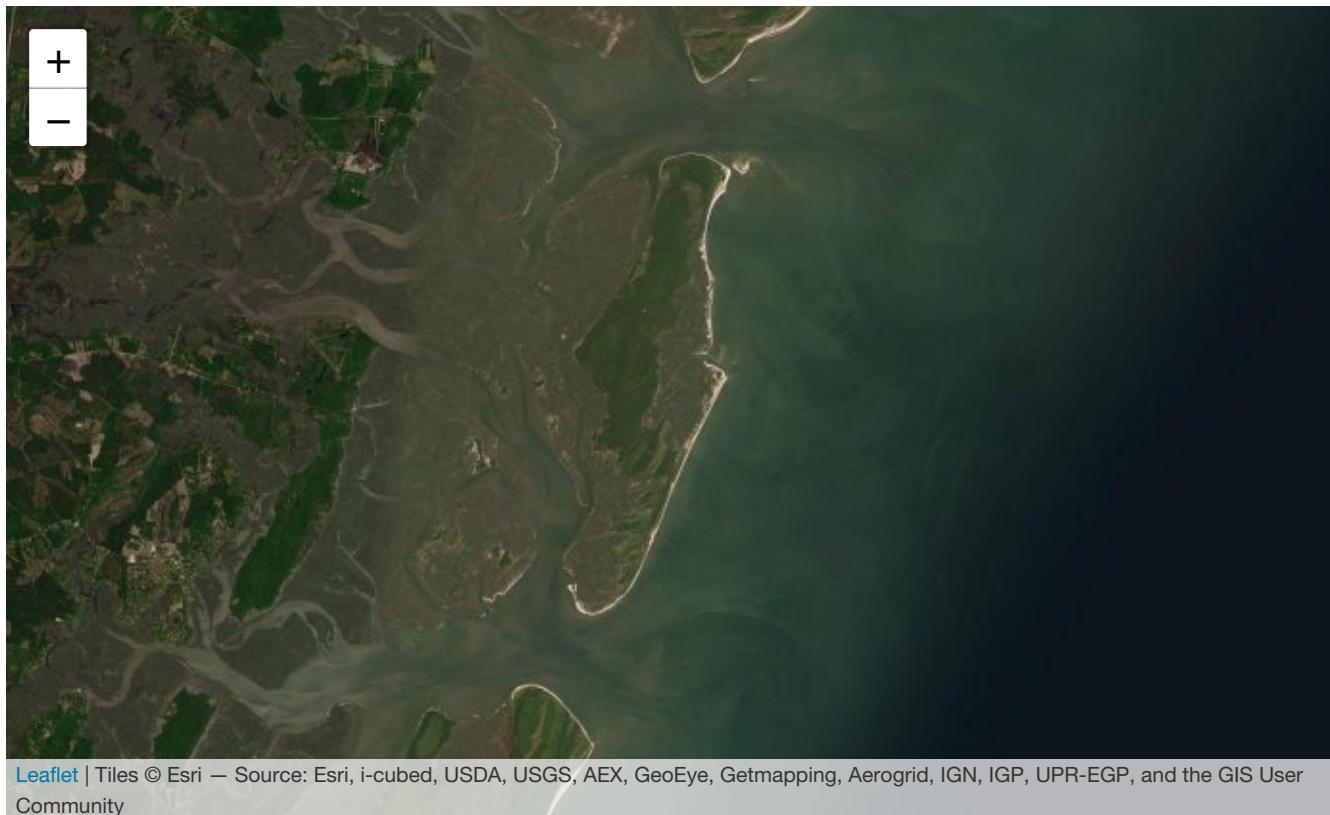
For example, map the bathymetry of the northeast Pacific basin:

```
leaflet() %>%
  addProviderTiles(providers$Esri.OceanBasemap) %>%
  fitBounds(lng1 = -165,
            lng2 = -100,
            lat1 = 10,
            lat2 = 50)
```



Or get satellite imagery for an island off the coast of Georgia, USA:

```
leaflet() %>%
  addProviderTiles(providers$Esri.WorldImagery) %>%
  fitBounds(lng1 = -81.2,
            lng2 = -81.1,
            lat1 = 31.54,
            lat2 = 31.7)
```



Geocoding

To jumpstart your mapping skills in R, the second package you need to know about is `tidygeocoder`. This package helps you “**geocode**” a mailing address. Geocoding means providing the latitude and longitude for an address, which allows you to find it on a map.

First, install & load the package:

```
install.packages('tidygeocoder')

library(tidygeocoder)
```

Second, create a dataframe for the address(es) you wish to map:

```
addresses <- data.frame(name= 'White House',
                         addr = "1600 Pennsylvania Ave NW, Washington, DC")
```

Now, geocode:

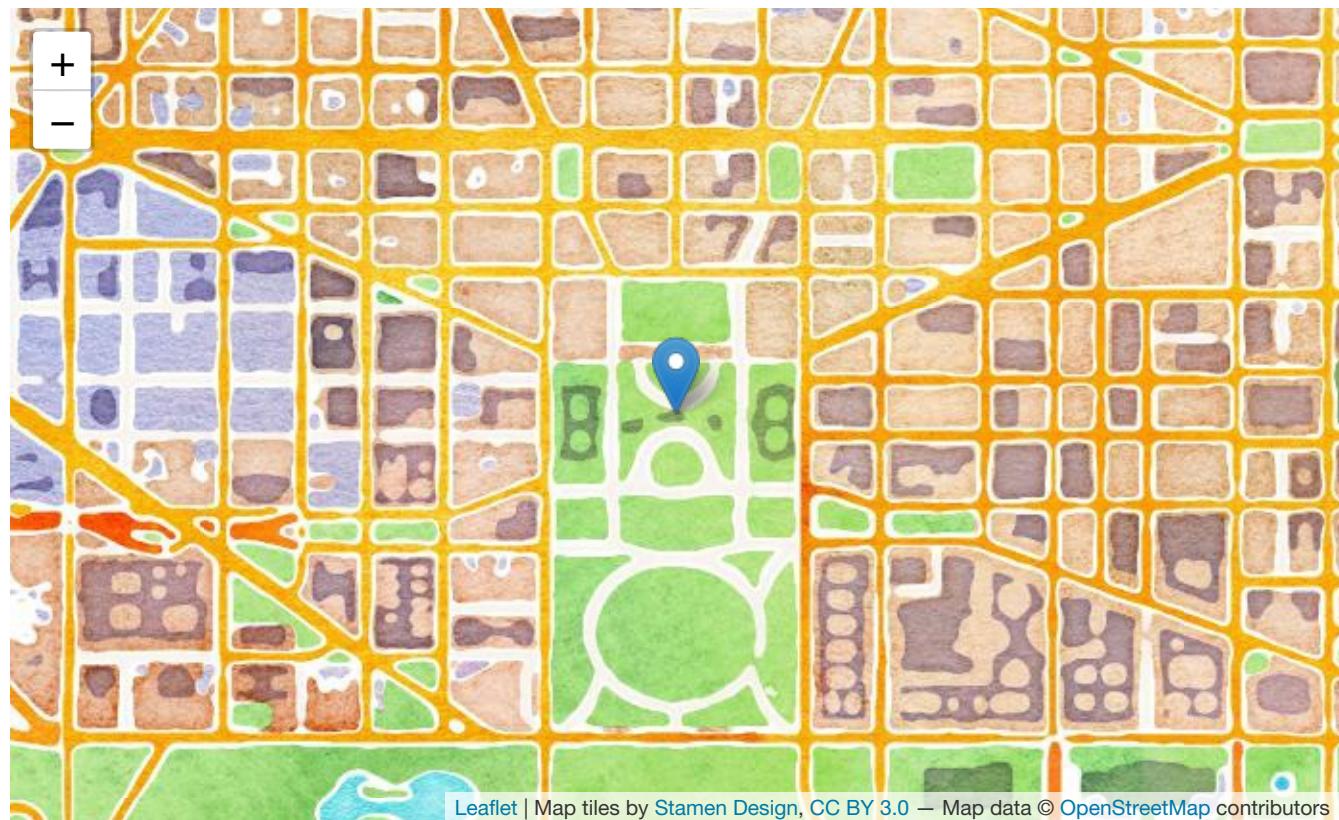
```
lat_longs <- addresses %>%
  geocode(addr, method = 'osm', lat = latitude , long = longitude)
```

Check out the result:

| | | latitude | longitude |
|------|--|----------|-----------|
| name | addr | <dbl> | <dbl> |
| 1 | White House 1600 Pennsylvania Ave NW, Washington, DC | 38.9 | -77.0 |

And check out your map:

```
leaflet(data=lat_longs) %>%
  addProviderTiles(providers$Stamen.Watercolor) %>%
  addMarkers()
```



You can also **reverse geocode**, i.e., get the mailing address for a lat-long coordinate.

Say these are your coordinates of interest:

```
lat_longs <- data.frame(latitude=48.8584,
                         longitude=2.2945)
```

Now, reverse-geocode:

```
reverse <- lat_longs %>%
  reverse_geocode(lat = latitude,
                 long = longitude)
```

Check out the result:

```
reverse
# A tibble: 1 x 3
  latitude longitude address
    <dbl>     <dbl> <chr>
1      48.9       2.29 Tour Eiffel, 5, Avenue Anatole France, Quartier du Gros-Ca~
```

Now, add the mailing address as a popup in your marker:

```
leaflet() %>%
  addProviderTiles(providers$HikeBike.HikeBike) %>%
  addCircleMarkers(data=lat_longs,
                   color = "firebrick",
                   radius = 15,
```

```
popup = gsub(",","<br>", reverse$address))
```

+
e [User:TheDJ]] if
- maintain
or the
IM

Contact [[User:TheDJ]] if
you want to maintain
OSMdb and/or the
wmflabs' OSM
tiles server.

Contact [[User:TheDJ]] if
you want to maintain
OSMdb and/or the
wmflabs' OSM
tiles server.

Con
you
OSM
wmf
tiles



[User:TheDJ]] if
maintain
or the
IM

Contact [[User:TheDJ]] if
you want to maintain
OSMdb and/or the
wmflabs' OSM
tiles server.

Contact [[User:TheDJ]] if
you want to maintain
OSMdb and/or the
wmflabs' OSM
tiles server.

Con
you
OSM
wmf
tiles

Leaflet | © OpenStreetMap contributors

Final thoughts

Maps, even simple ones, are amazingly effective data visualizations. The density and layers of information contained within a map are incredible!

All of those virtues are compounded with an interactive map. At each zoom level, you gain a new perspective. Like shiny apps, leaflet maps put the viewer in control of how they explore and understand the data.

Exercises

1. Create a new Rmarkdown document.
2. Create a code chunk. In this chunk, read in some data on “conflicts”. To read in the data, run the below code.

```
download.file('https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/conflicts.RData'
             destfile = 'conflicts.RData')

load('conflicts.RData')
```

This data comes from <https://ucdp.uu.se/encyclopedia>. Take a minute or two to look at the website.

3. Have a look in the data. Which fields are likely to be geographic?
4. Make a simple x-y plot using geographic fields.
5. Create an object named `conflicts_afg`. This should be a plot of conflicts in Afghanistan.
6. Color the points by year.
7. Instead of year, color the points by `deaths_civilians`.
8. Color the points by date, but make point size reflect `deaths_civilians`.

9. Create a leaflet map of conflicts in a country of your choice.
10. Use `addTiles`.
11. Use `addProviderTiles` to make your map a satellite map.
12. Explore other tiles
13. Add pop-ups to your maps by using the `popup` argument within `addMarkers`.
14. Add `clusterOptions = markerClusterOptions()` to make your points clustered.
15. Replace your markers with “circle markers”.
16. Create a shiny app wherein the user selects a country and time frame, and the app shows both (a) an interactive map and (b) a plot of the number of conflicts by year for that country.

Part III

Review exercises

Learning goals

- This is a review exercise: apply the `dplyr` and `ggplot` skills introduced in the previous modules to explore a mystery dataset.

Chapter 19

A mystery dataset

Use the `dplyr` verbs to answer these questions.

First, download the following dataset by running this code.

```
library(readr)
library(dplyr)
df <- read_csv('https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/deaths.csv')
```

1. Review the dataset and try to figure out which each row represents. “Each row is a _____.”
2. How many people are in the dataset?
3. Use `summarize()` to count the number of men and women.
4. Use `summarize()` to count the number of people in each class.
5. Use `summarize()` to count the number of men and women in each class.
6. What is the average age of men in the dataset?
7. What is the average age of women in the dataset?
8. Use `mutate` to create a variable called `died`. This should be a boolean based on the `Survived` column (in which 1 means the person survived, and 0 means the person died).
9. Use `mutate` to create a variable called `child`. This should be a boolean based on the `Age` column, indicating if someone was less than 18 years old.
10. Create a different dataframe for men vs. women. Name them accordingly.
11. Create a different dataframe for class 1, class 2, and class 3. Name them accordingly.
12. For each of the 5 datasets you’ve just created, what is the death rate?
13. For each of the 5 datasets, how many children died?
14. Now, using the *full* original dataset, calculate the child-specific death rate for each combination of class and sex (ie, “first class females”, “third class males”, etc.).
15. What did you find? What might explain that?
16. What is the average age of men and women, separately, in each class?

Chapter 20

A `dplyr` survey

Learning goals

- This is a review exercise: apply the `dplyr` and `ggplot` skills introduced in the previous modules to analyze the results of a recent survey.

Use the `dplyr` verbs to answer these questions.

First, download the results of a recent survey by running this code.

```
# Load library
library(gsheet)
library(dplyr)

# Read in data
survey <- gsheet::gsheet2tbl('https://docs.google.com/spreadsheets/d/1iVt9FX9J2iv3QFKBM7Gzb9dgva70XrW1lxM')

# remake the names
names(survey) <- c('time', 'sex', 'age','sib', 'dad_mus', 'person_mus', 'joe_mus_is', 'eyesight', 'height')
```

1. Review the `survey` dataset and try to figure out which each row represents. “Each row is a _____.”
2. How many respondents are in the dataset?
3. Create a dataframe called `old_people`. This should include only people older than 20. Write code to calculate the number rows in your new dataframe.
4. Create a dataframe called `captivated`. This should include all those people who find Joe’s moustache to be “deeply captivating”. Write code to calculate the number rows in your new dataframe.
5. Create a dataframe called `special_people`. This should be people who are taller than 175cm, prefer cats over dogs, and consider themselves to be average at rock, paper, scissors.
6. In the full dataset (`survey`), do more people like cats or dogs? What about among “special” people?
7. Create a new variable in `survey` called “`std_shoes`” that standardizes shoe sizes by converting men’s shoe size to women’s (There is an approximate 1.5 size difference between Men’s and Women’s sizing (e.g., a men’s size 7 is roughly equivalent to a women’s size 8.5)
8. Get the avg shoe size by sex (Male, Female, Prefer not to say)
9. Get Average age, height, & number of siblings, by the sex
10. Do people that have ever had a mustache think there will be more pandemics on average than those who have never had a mustache?
11. Do people that prefer cats have smaller feet on average than those who prefer dogs?

12. Is eyesight associated with moustache perception?
13. What percentage of people think they are better than average at rock scissors paper?
14. What percentage of men and women think they are better than average at rock scissors paper?
15. How many people think money matters more than love?
16. Create a dataframe, grouped by whether or not people's dads had moustaches, with variables showing each of the following: the maximum age, maximum height, minimum number of pandemics, and average number of siblings
17. What percentage of men have terrible eyesight?
18. How many women have a shoe size of 9 or more?
19. Create a variable in the `survey` dataset named `days_old`. Use the `bday` variable and subtract it from `Sys.Date()`.
20. What is the standard deviation of age?
21. Create a one-column dataset which contains the name(s) of the person(s) with the most number of siblings (hint: use the following dplyr verbs in this order: `filter`, `select`).
22. Create a bar chart that compares the average age of those who prefer cats vs dogs
23. Create a scatter plot that shows the relationship between age and height (make it look nice!)
24. Create a scatter plot that shows the relationship between height and shoe size
25. Create a bar chart that shows the average number of siblings for dad's mustache status.
26. Color the barchart above and add some degree of transparency in the color. Add a title.
27. Create a bar chart that shows the average shoe size by sex and cat/dog preference.

Chapter 21

Global health & ggplot

This review exercise is useful immediately after you learn `ggplot2` and `dplyr`.

First, let's read in some data on health from the World Bank:

```
library(readr)
library(dplyr)
library(gapminder)
Error in library(gapminder): there is no package called 'gapminder'
gm <- gapminder::gapminder
Error in loadNamespace(x): there is no package called 'gapminder'
```

1. How many rows are in the dataset?
2. How many columns are in the dataset?
3. What are the names of the columns?
4. What is the oldest year in the dataset?
5. What is the country/year with the greatest population in the dataset?
6. Get the average GDP per capita for each continent in 1952.
7. Get the average GDP per capita for each continent for the most recent year in the dataset.
8. Average GDP is a bit misleading, since it does not take into account the relative size (in population) of the different countries (ie, China is a lot bigger than Cambodia). Look up the function `weighted.mean`. Use it to get the average life expectancy by continent for the most recent year in the dataset, weighted by population.
9. Make a barplot of the above table (ie, average life expectancy by continent, weighted by population).
10. Make a point plot in which the x-axis is country, and the y-axis is GDP. Add the line `theme(axis.text.x = element_text(angle = 90))` in order to make the x-axis text vertically aligned. What's the problem with this plot? How many points are there per country?
11. Make a new version of the above, but filter down to just the earliest year in the dataset.
12. Make a scatterplot of life expectancy and GDP per capita, just for 1972.
13. Make the same plot as above, but for the most recent year in the data.
14. Make the same plot as the above, but have the size of the points reflect the population.
15. Make the same plot as the above, but have the color of the points reflect the continent.

16. Filter the data down to just the most recent year in the data, and make a histogram (`geom_histogram`) showing the distribution of GDP per capita.
17. Get the average GDP per capita for each continent/year, weighted by the population of each country.
18. Using the data created above, make a plot in which the x-axis is year, the y-axis is (weighted) average GDP per capita, and the color of the lines reflects the continent.
19. Make the same plot as the above, but facet the plot by continent.
20. Make the same plot as the above, but remove the coloring by continent.
21. Make a plot showing France's population over time.
22. Make a plot showing all European countries' population over time, with color reflecting the name of the country.
23. Create a variable called `status`. If GDP per capita is over 20,000, this should be "rich"; if between 5,000 and 20,000, this should be "middle"; if this is less than 5,000, this should be "poor".
24. Create an object with the number of rich countries per year.
25. Create an object with the percentage of countries that were rich each year.
26. Create a plot showing the percentage of countries which were rich each year.
27. Create an object with the number of people living in poor countries each year.
28. Create a chart showing the number of people living in rich, medium, and poor countries per year (line chart, coloring by `status`).
29. Create a chart showing the life expectancy in Somalia over time.
30. Create a chart showing GDP per capita in Somalia over time.
31. Create a histogram of life expectancy for the most recent year in the data. Facet this chart by continent.
32. Create a barchart showing average continent-level GDP over time, weighted for population, with one bar for each year, stacked bars with the color of the bars indicating continent (`geom_bar(position = 'stack')`).
33. Create the same chart as above, but with bars side-by-side (`geom_bar(position = 'dodge')`)
34. Generate 3-5 more charts / tables that show interesting things about the data.
35. Make the above charts as aesthetically pleasing as possible.

Bibliography

- Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.
- Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.