DataBOOM: the canon for data science

Databrew

2021-06-09

Contents

Chapter 1

Welcome!

Welcome to `DataBOOM`, a curriculum designed to guide you from your very first line of code towards becoming a professional data scientist.

What this is, and what it isn't

This is not a textbook or a reference manual. It is not exhaustive or comprehensive. It is a *training manual* designed to *empower researchers to do impactful data science.* As such, its tutorials and exercises aim to get you, the researcher, to start writing your own code as quickly as possible and – equally of importance – to *start thinking like a data scientist*, by which we mean tackling ambiguous problems with persistence, independence, and creative problem solving.

Furthermore, this is not a fancy interactive tutorial with bells or whistles. It was purposefully designed to be simple and "analog". You will not be typing your code into this website and getting feedback from a robot, or setting up an account to track your progress, or getting pretty merit badges or points when you complete each module.

Instead, you will be doing your work on your own machine, working with real folders and files, downloading data and moving it around, etc. – all the things you will be doing as a data scientist in the real world.

Who this is for

This curriculum covers everything from the absolute basics of writing code in `R` to machine learning with `tensorflow`. As such, it is designed to be useful to everyone in some way. But the target audience for these tutorials is the student

who *wants* to work with data but has *zero* formal training in programming, computer science, or statistics.

This curriculum was originally developed for the **Sewanee Data Institute for Social Good** at Sewanee: The University of the South, TN, USA.

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.

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`) to explore and plot data. 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 format data to simplify analysis and add tools for *data wrangling* (i.e., transforming and re-formatting data to prepare it for plotting and analysis). You will also learn how to conduct basic statistics, from exploratory data analyses (e.g., producing and comparing distributions) to significance testing.
- The **Essential `R` skills** unit equips you with the tools, tricks, and mindset for tackling the most common tasks in data science. 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 next several units provide a suite of skills essential to any data science professional:

- The **Interactive dashboards** unit teaches you how to make dashboards and websites for projects using `shiny` in `RStudio`.
- The **Databases** unit teaches you how to access, create, and work with relational databases online using `SQL` and its alternatives.
- The **Documenting your work** unit teaches you to use `R Markdown` to produce beautiful, reproducible data reports. You will also learn about *version control*, using `Git` and `GitHub` to collaborate on shared projects and work on data science teams.
- The **Sharing research** unit teaches you to produce publishable research articles and compelling presentations.

The final unit, **Advanced skills**, introduces you to a variety of advanced data science techniques, from interactive maps to iterative simulations to machine learning, that can help you begin to specialize your skillset.

Who we are

Joe Brew is a data scientist, epidemiologist, and economist. He has worked with the Florida Department of Health (USA), the Chicago Department of Public Health (USA), the Barcelona Public Health Agency (Spain), the Tigray Regional Health Bureau (Ethiopia) and the Manhiça Health Research Center (Mozambique). He is a co-founder of Hyfe and DataBrew. His research focuses on the economics of malaria and its elimination. He earned his BA at Sewanee: The University of the South (2008), an MA at the Institut Catholique de Paris (2009) and an MPH at the Kobenhavns Universitet (2013). He is passionate about international development, infectious disease surveillance, teaching, running, and pizza.

Ben Brew is a data scientist, economist, and health sciences researcher. In addition to co-founding DataBrew, he has spent most of the last few years working with SickKids Hospital in Ontario on machine learning applications for cancer genomics. He earned his BA at Sewanee: The University of the South (2012), and a Master's in Mathetical Models for Economics from the Paris School of Economics (Paris I) (2015). He is passionate about econometrics, applied machine learning, and cycling.

Eric Keen is a data scientist, marine ecologist, and educator. He is the Science Co-director at BCwhales, a research biologist at Marecotel, a data scientist at Hyfe, and a professor of Environmental Studies at Sewanee: the University of the South. He earned his BA at Sewanee (2008) and his PhD at Scripps Institution of Oceanography (2017). His research focuses on the ecology and conservation of whales in developing coastal habitats. He is passionate about whales, conservation, teaching, small-scale farming, running, and bicycles. And pizza.

Part I

Core theory

Chapter 2

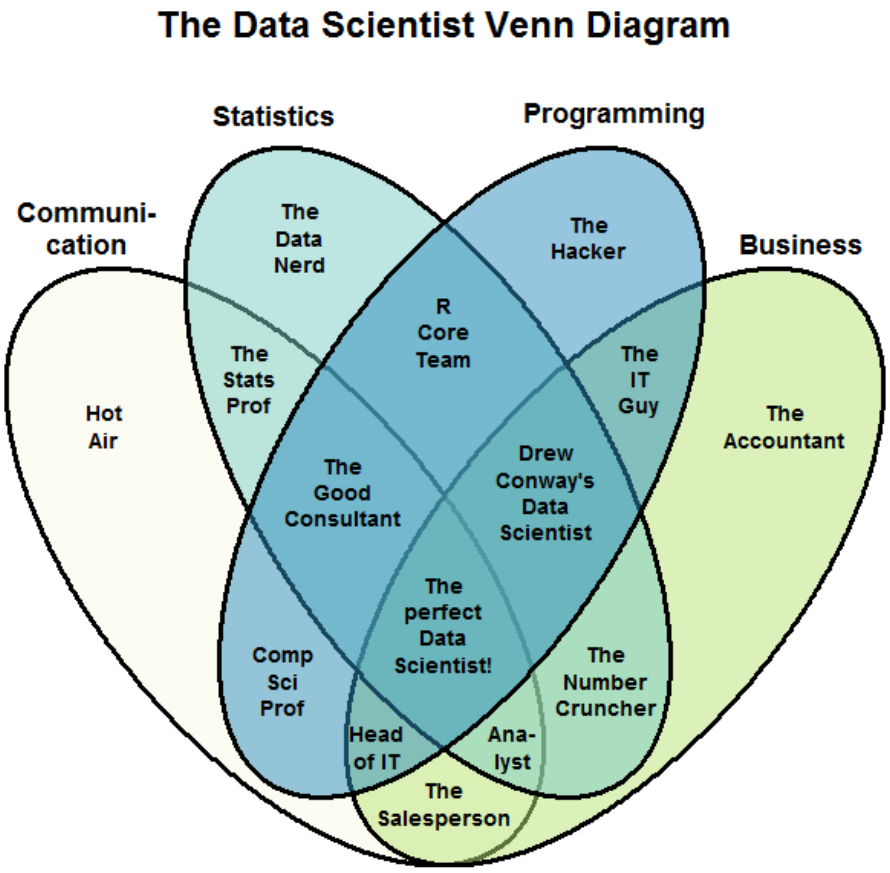
Principles of data science

What is data science?

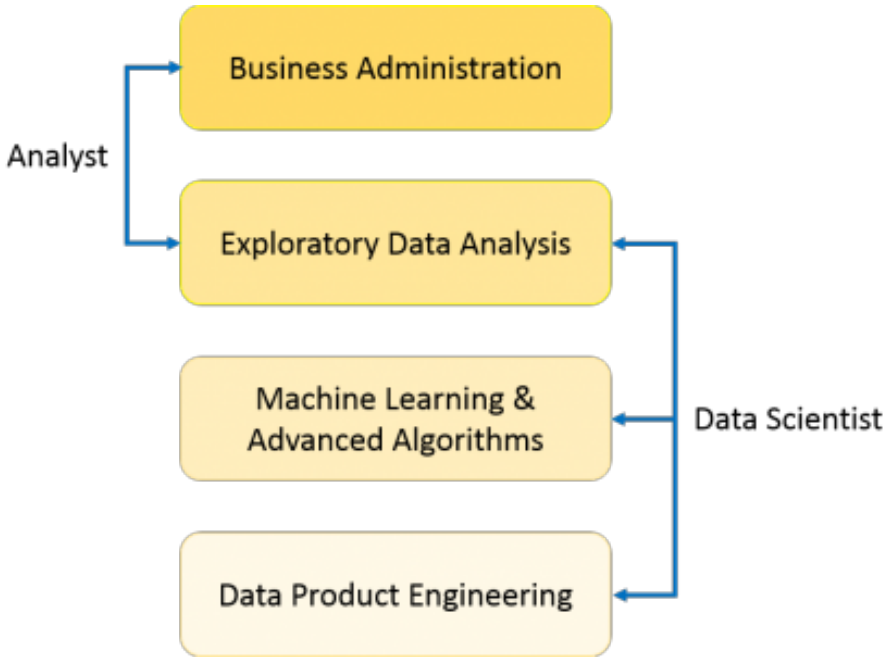
Data science is an interdisciplinary field. Some have argued that it is not a field unto itself, but rather an extension of statistics. In this course, however, we'll take the majority view that data science is its own field: a new field, which combines statistics, mathematics, and computer science.

But we'll go one step outward. Data science is not just the combination of those academic disciplines which form its core; its also something more. Good data science involves domain knowledge (ie, familiarity with the problem being solved), effective communication, an iterative mentality (ie, creating feedback loops for rapid hypothesis testing), a bias to real-world effects rather than theoretical frameworks, and a willingness/desire to work in the real world.

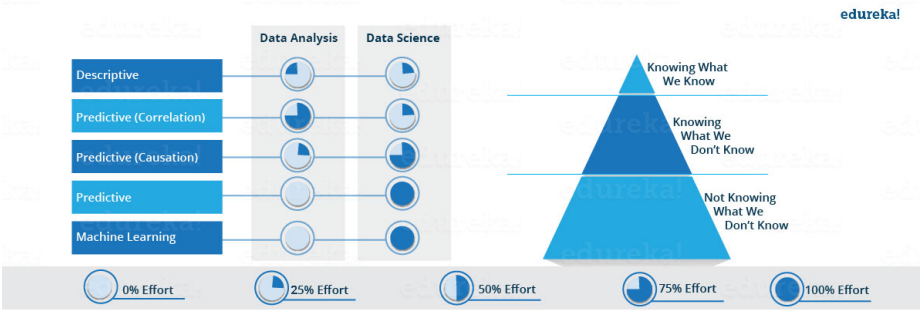
There are a lot of Venn diagrams and figures out there, trying to show what data science is. For example…


The Data Scientist Venn Diagram
Statistics
Programming
Communi-cation
Business
The Data Nerd
The Hacker
R Core Team
The Stats Prof
The IT Guy
Hot Air
The Accountant
The Good Consultant
Drew Conway's Data Scientist
The perfect Data Scientist!
Comp Sci Prof
The Number Cruncher
Head of IT
Ana-lyst
The Salesperson

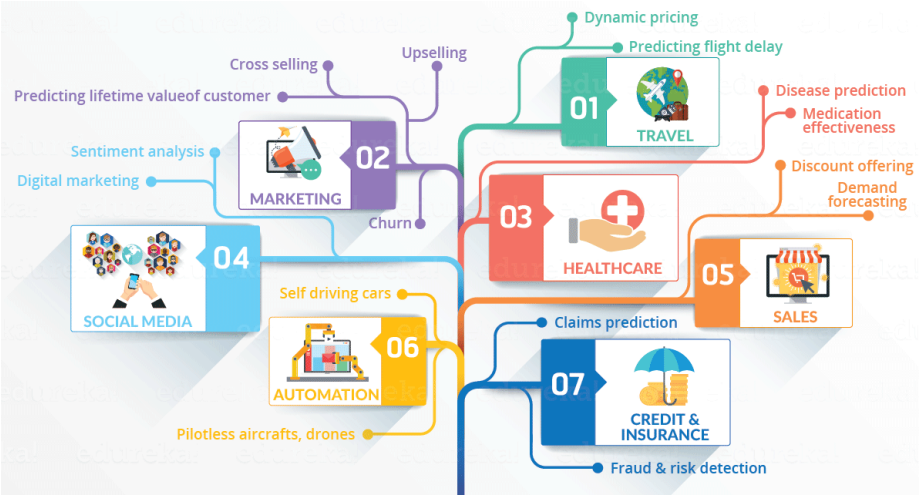
... or ...


Business Administration
Analyst
Exploratory Data Analysis
Machine Learning &
Advanced Algorithms
Data Scientist
Data Product Engineering

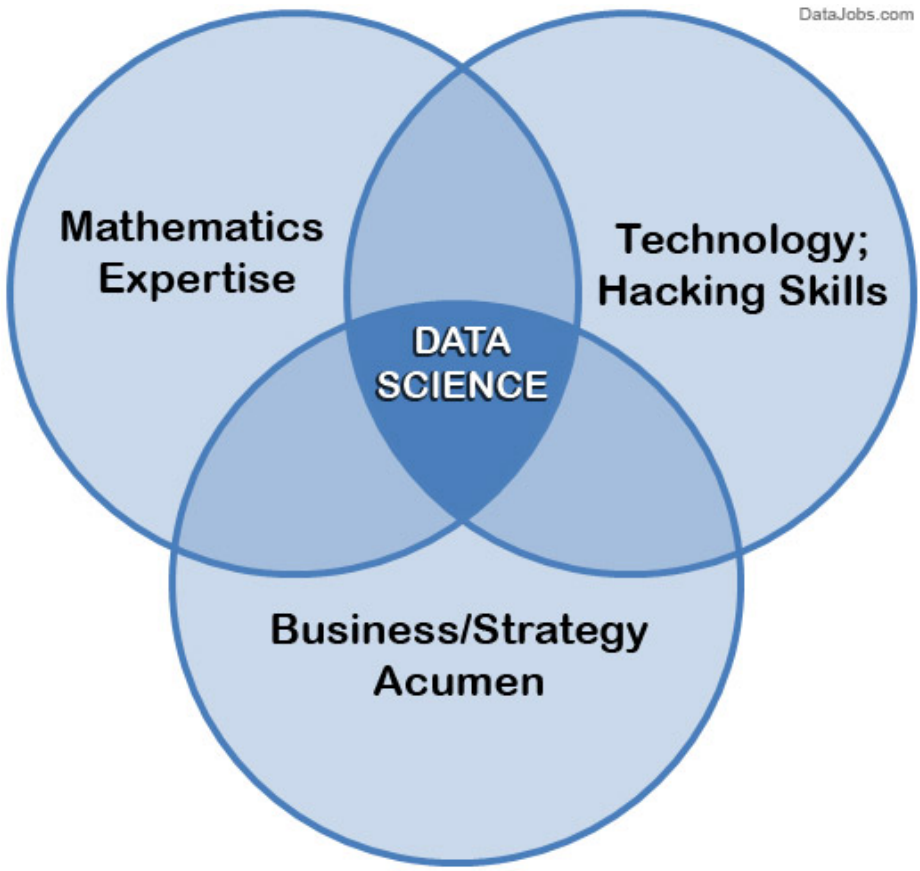
... or ...


edureka!
Data Analysis
Data Science
Descriptive
Predictive (Correlation)
Predictive (Causation)
Predictive
Machine Learning
0% Effort
25% Effort
50% Effort
75% Effort
100% Effort
Knowing What
We Know
Knowing
What We
Don't Know
Not Knowing
What We
Don't Know

... or ...


Dynamic pricing
Predicting flight delay
01
TRAVEL
Disease prediction
Medication effectiveness
Cross selling
Upselling
Predicting lifetime valueof customer
Sentiment analysis
Digital marketing
02
MARKETING
Churn
03
HEALTHCARE
Discount offering
Demand forecasting
04
SOCIAL MEDIA
Self driving cars
05
SALES
Claims prediction
06
AUTOMATION
07
CREDIT & INSURANCE
Pilotless aircrafts, drones
Fraud & risk detection

... or ...


DataJobs.com
Mathematics Expertise
Technology; Hacking Skills
DATA SCIENCE
Business/Strategy Acumen

Our take? These are useful, interesting, and to some degree accurate *but* data science is too new, and too fluid, to be fixed into some static definition. 15 years ago, data science was "dashboards" and "predictive analytics". 10 years ago, data science was "big data" and "data mining". Since then, machine learning and artificial intelligence have taken up an ever-larger chunk of what most people consider data science. And in 5 or 10 years? Who knows.

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. And that's what this course is going to be about. *DOING*.

Data scientist vs data analysist vs data engineer

There is a lot of fuss out there about what constitutes data science and what doesn't. Our definition is broad enough to encompass lots of things. So, we consider an "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.

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, 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, but rather 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. Good, effective data scientists, in practice, are often not those with the most cutting-edge algorithm, but rather those who are able to get the right data and ask the right questions.

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 and definition
2. Question formation
3. Hypothesis generation
4. Data collection
5. Data processing
6. Model building / hypothesis testing
7. Operationalization
8. Communication / dissemination
9. Action
10. Observation

Does the above look a lot like the sceintific method? That's because it basically is. 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 - let's talk about what data scientists are actually doing in the real world.

Data scientists are working on a ton of problems. Some examples include:

- Targeted advertising
- Dynamic airline pricing
- Social media feed optimization
- Making video games more fun / addictive
- Identifying and filtering inappropriate content on the internet

- Search autocomplete
- Automating identification of credit card fraud
- Facial recognition
- Voice recognition
- Filtering spam
- Autocorrect
- Virtual assistants
- Preventive maintenance at nuclear facilities
- Identifying tax evaders
- Identifying disease through imagery
- Improving chemotherapy dosage
- Quantifying the likelihood of recedivism to prevent over-incarceration
- Surveilling emergency rooms to predict disease outbreaks
- Improve matchmaking systems (liver transplants, love, etc.)
- Detecting fake news
- Predictive policing

Why R?

This course if largely about learning to *do*, and will largely use R. R is not the only tool in the data scientists' toolbox, but it's a good one, is extremely popular, has a larger community and user base, and can be applied to many fields.

What are some of the other tools and languages used by data scientists? There are plenty (and we won't cover them here). But the main reason we choose R for this course is the fact that it is open-source and free. This matters because…

The reproducibility crisis

There is a crisis in science (and data science): the reproducitibility crisis (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 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, and how can we separate the wheat from the chaff?

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. And reproducible science 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.

In this course, we'll 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.

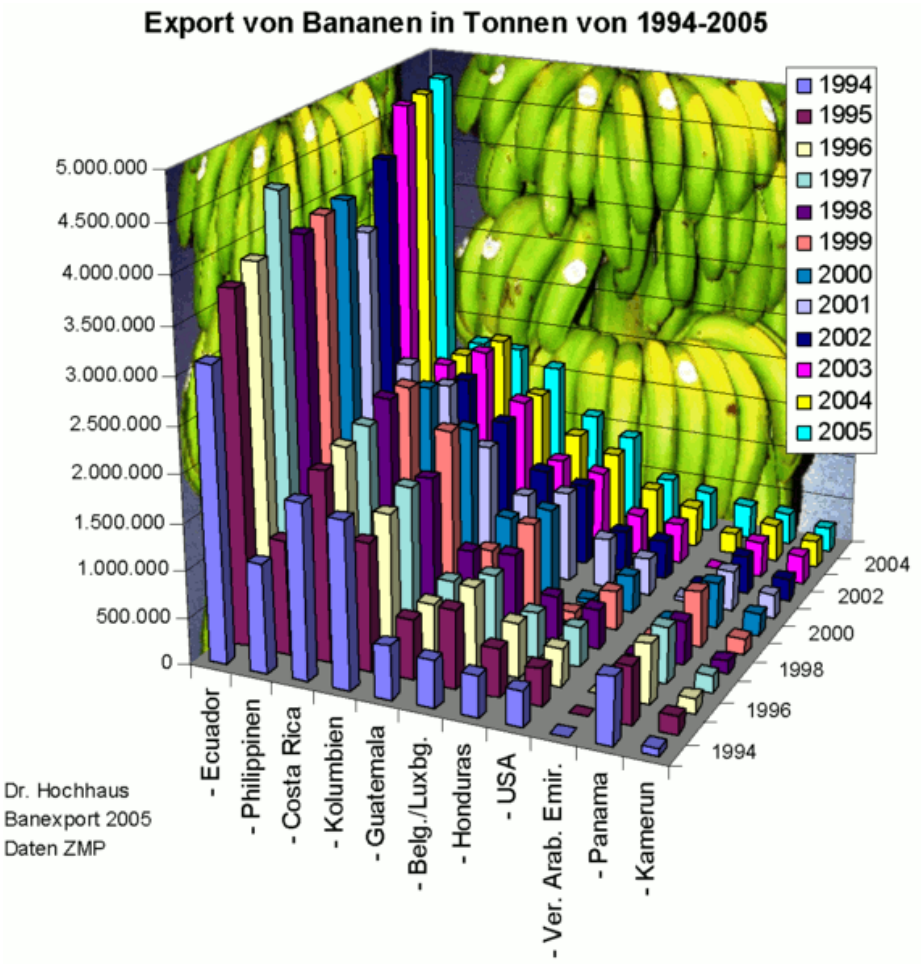
Chapter 3

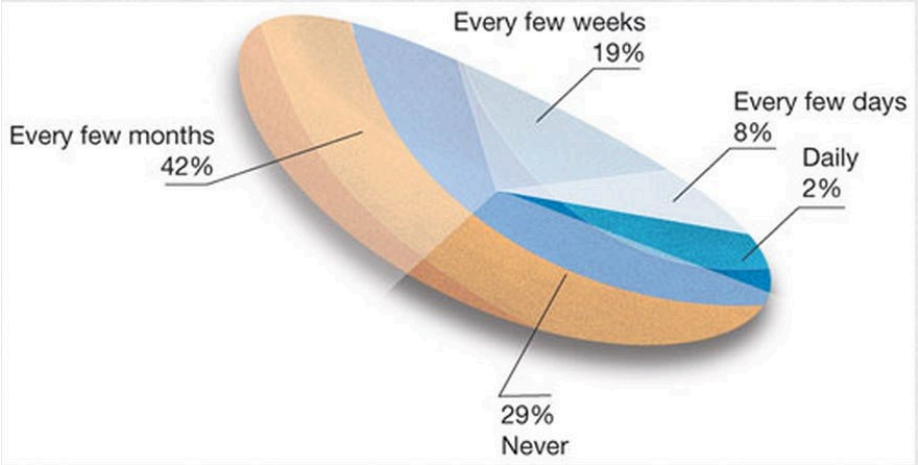
Visualizing data

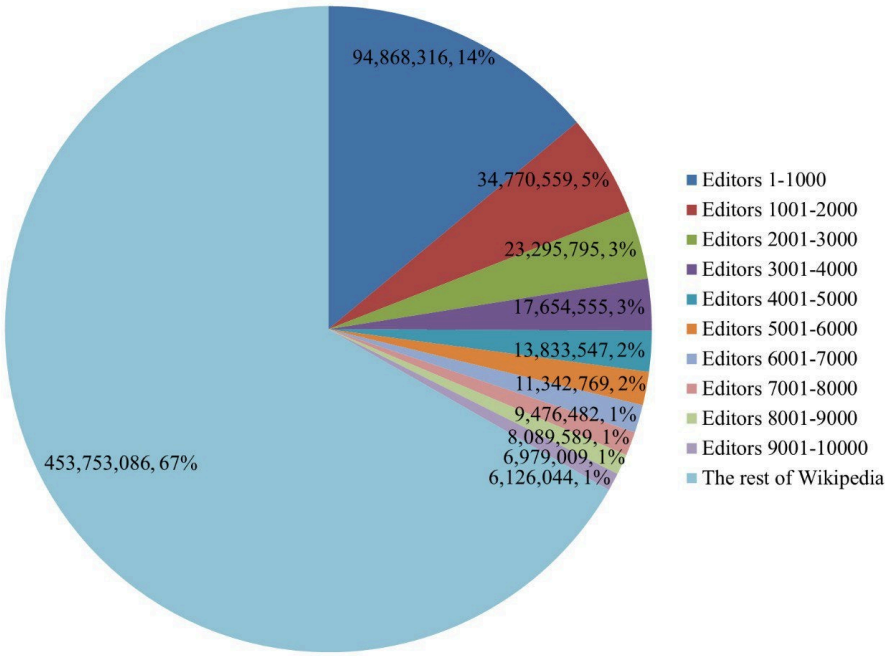
3.1 The importance of visualization

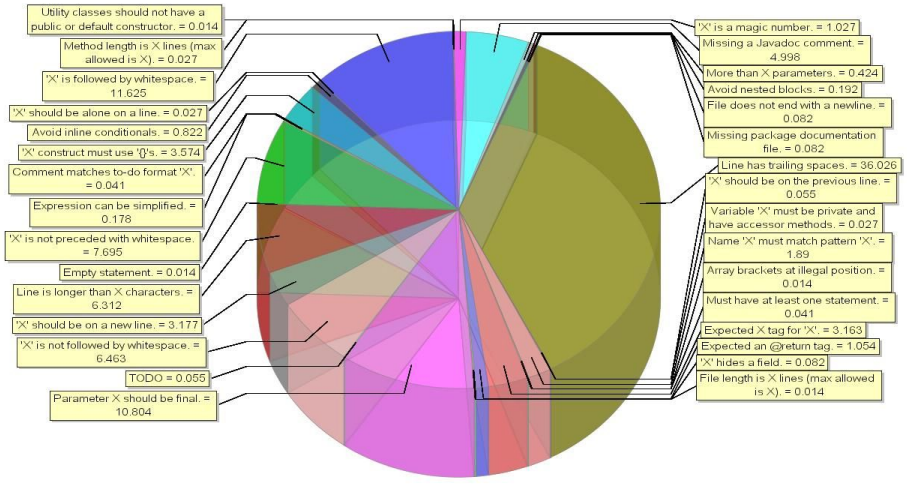
Much of the next few weeks is going to focus on visualization. The reason why is that data visualization is a powerful, quick, and clear tool for communicating data, exploring it, and understanding it. With that in mind, let's get started with some *data viz*.

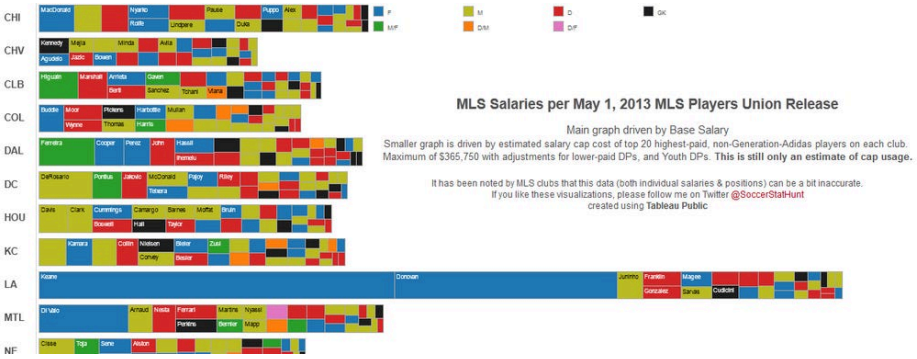
3.2 Bad examples

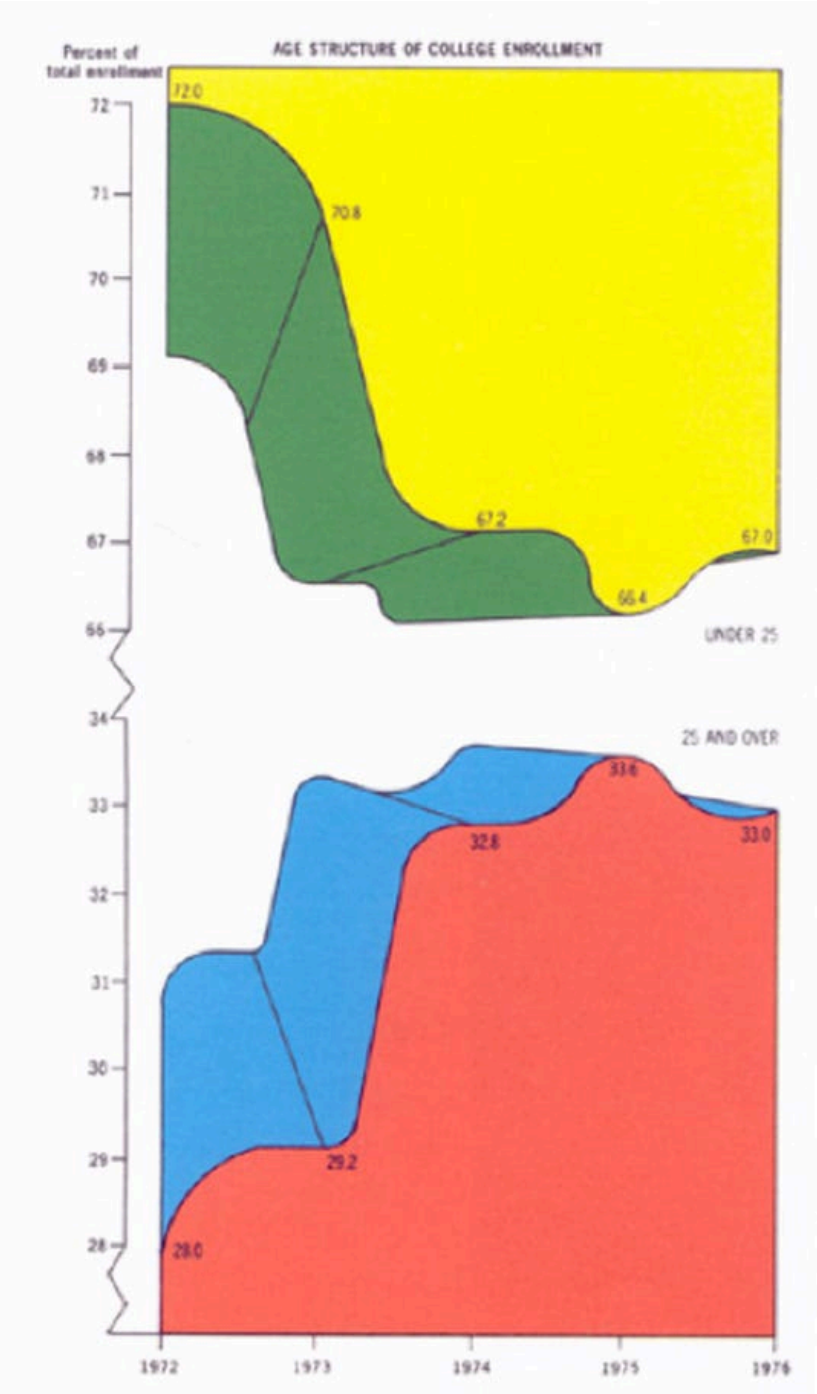


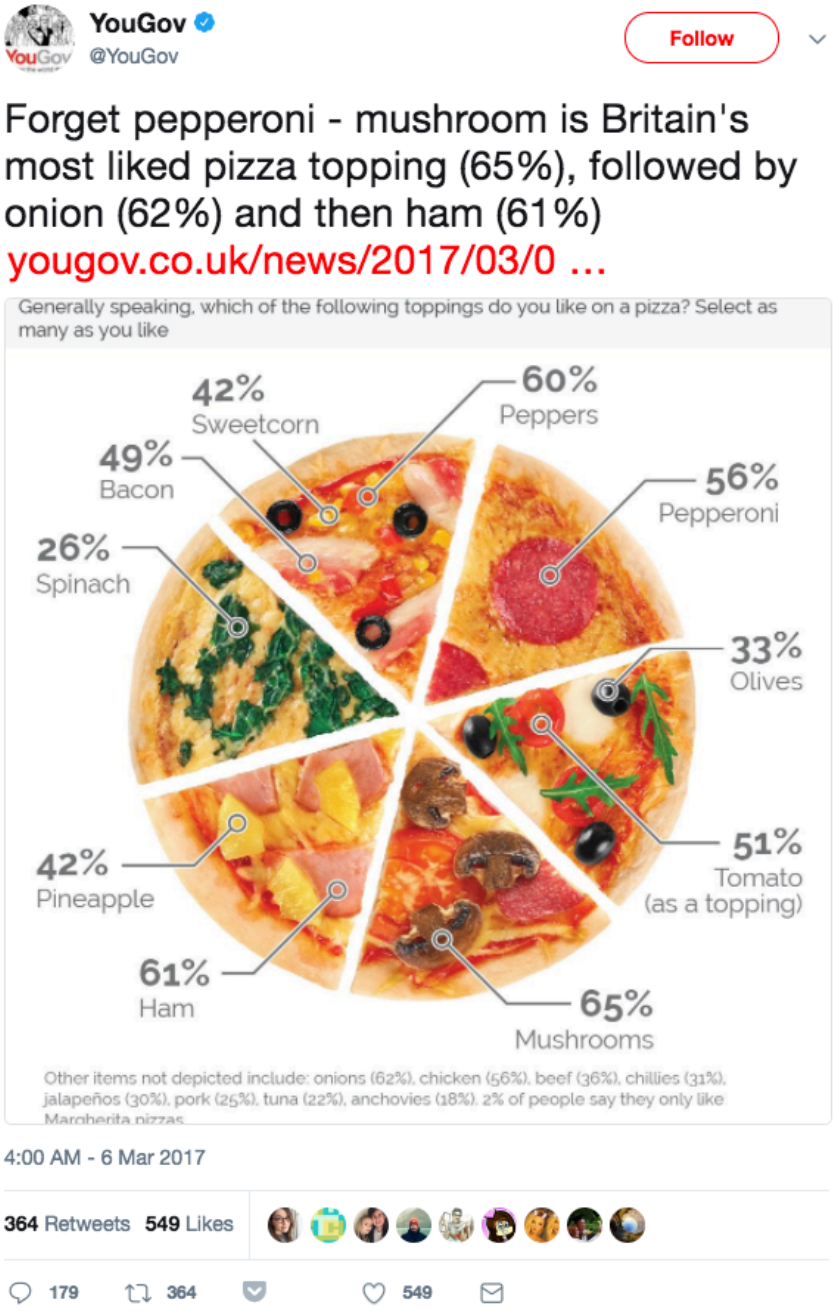

Every few weeks
19%
Every few days
8%
Daily
2%
Every few months
42%
29%
Never

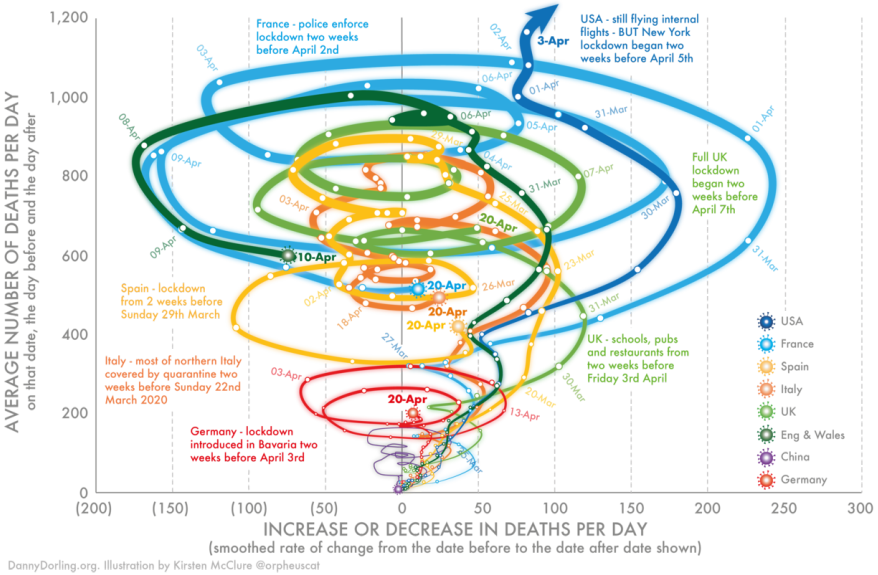

94,868,316, 14%
34,770,559, 5%
23,295,795, 3%
17,654,555, 3%
13,833,547, 2%
11,342,769, 2%
9,476,482, 1%
8,089,589, 1%
6,979,009, 1%
6,126,044, 1%
453,753,086, 67%
Editors 1-1000
Editors 1001-2000
Editors 2001-3000
Editors 3001-4000
Editors 4001-5000
Editors 5001-6000
Editors 6001-7000
Editors 7001-8000
Editors 8001-9000
Editors 9001-10000
The rest of Wikipedia

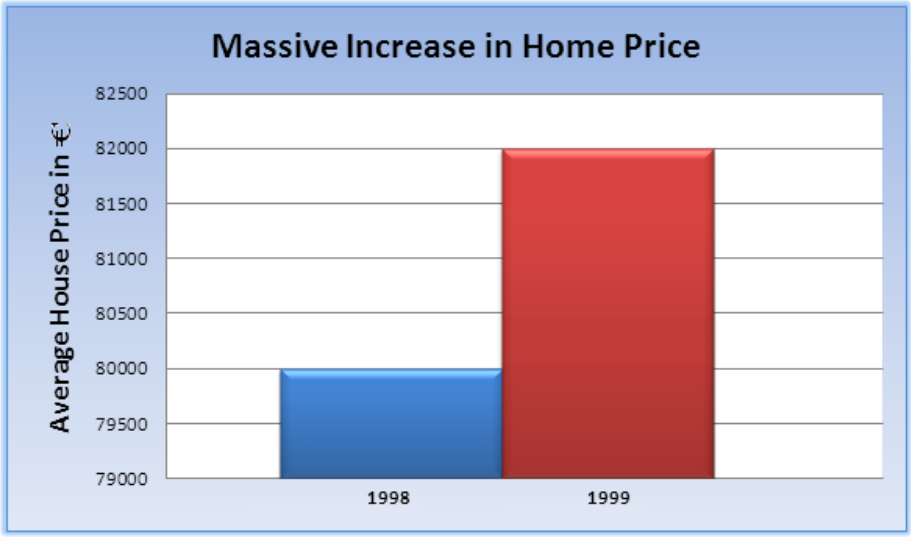

Utility classes should not have a public or default constructor. = 0.014
Method length is X lines (max allowed is X). = 0.027
'X' is followed by whitespace. = 11.625
'X' should be alone on a line. = 0.027
Avoid inline conditionals. = 0.822
'X' construct must use '{}'s. = 3.574
Comment matches to-do format 'X'. = 0.041
Expression can be simplified. = 0.178
'X' is not preceded with whitespace. = 7.695
Empty statement. = 0.014
Line is longer than X characters. = 6.312
'X' should be on a new line. = 3.177
'X' is not followed by whitespace. = 6.463
TODO = 0.055
Parameter X should be final. = 10.804
'X' is a magic number. = 1.027
Missing a Javadoc comment. = 4.998
More than X parameters. = 0.424
Avoid nested blocks. = 0.192
File does not end with a newline. = 0.082
Missing package documentation file. = 0.082
Line has trailing spaces. = 36.026
'X' should be on the previous line. = 0.055
Variable 'X' must be private and have accessor methods. = 0.027
Name 'X' must match pattern 'X'. = 1.89
Array brackets at illegal position. = 0.014
Must have at least one statement. = 0.041
Expected X tag for 'X'. = 3.163
Expected an @return tag. = 1.054
'X' hides a field. = 0.082
File length is X lines (max allowed is X). = 0.014

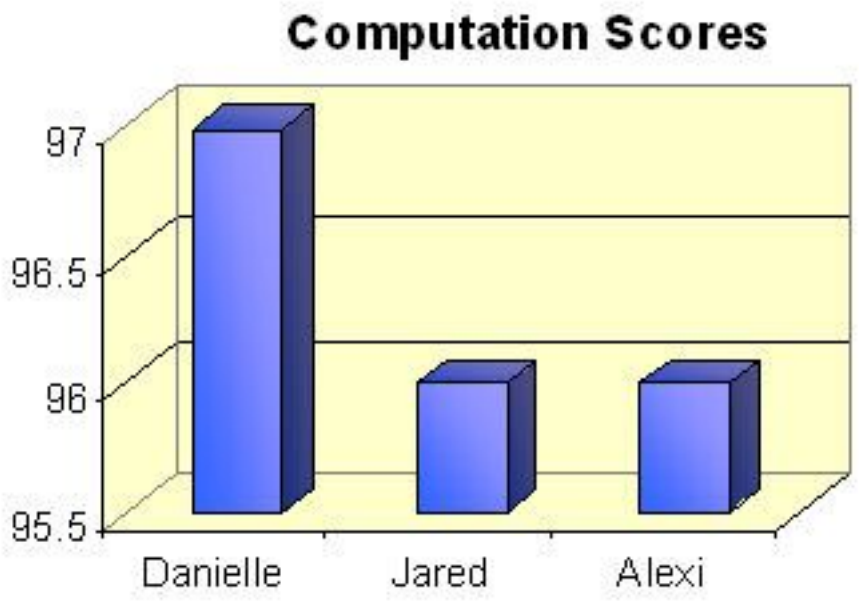

CHI
CHV
CLB
COL
DAL
DC
HOU
KC
LA
MTL
NF
MLS Salaries per May 1, 2013 MLS Players Union Release
Main graph driven by Base Salary
Smaller graph is driven by estimated salary cap cost of top 20 highest-paid, non-Generation-Adidas players on each club.
Maximum of $365,750 with adjustments for lower-paid DPs, and Youth DPs. This is still only an estimate of cap usage.
It has been noted by MLS clubs that this data (both individual salaries & positions) can be a bit inaccurate.
If you like these visualizations, please follow me on Twitter @SoccerStatHunt
created using Tableau Public

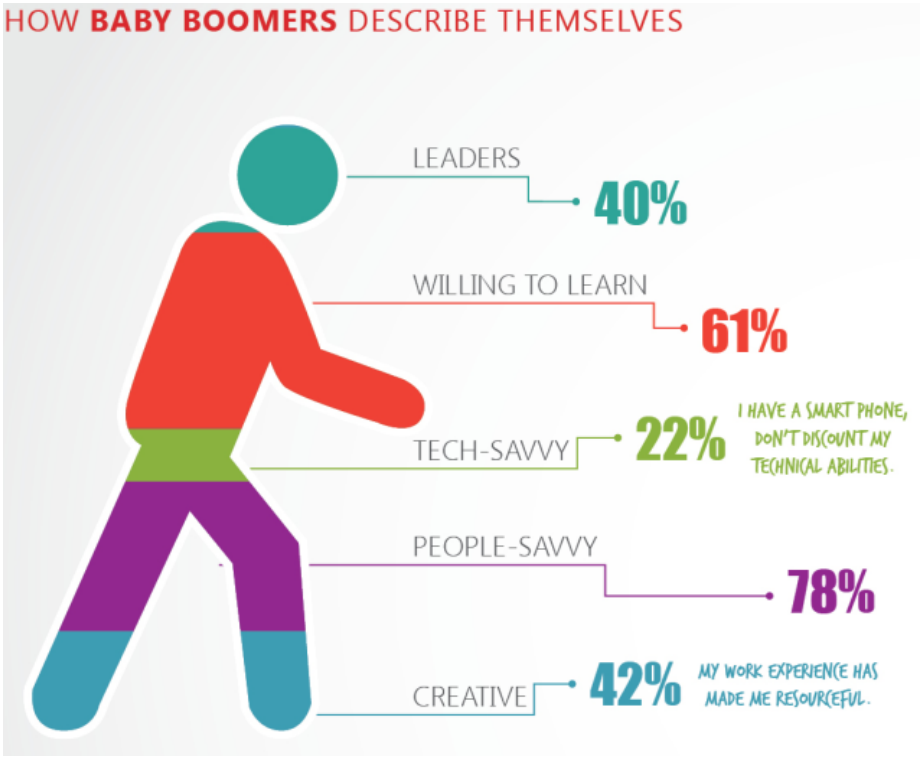

AGE STRUCTURE OF COLLEGE ENROLLMENT
Percent of total enrollment
72
71
70
69
68
67
66
34
33
32
31
30
29
28
72.0
70.8
67.2
66.4
67.0
UNDER 25
25 AND OVER
33.6
32.8
33.0
29.2
28.0
1972
1973
1974
1975
1976

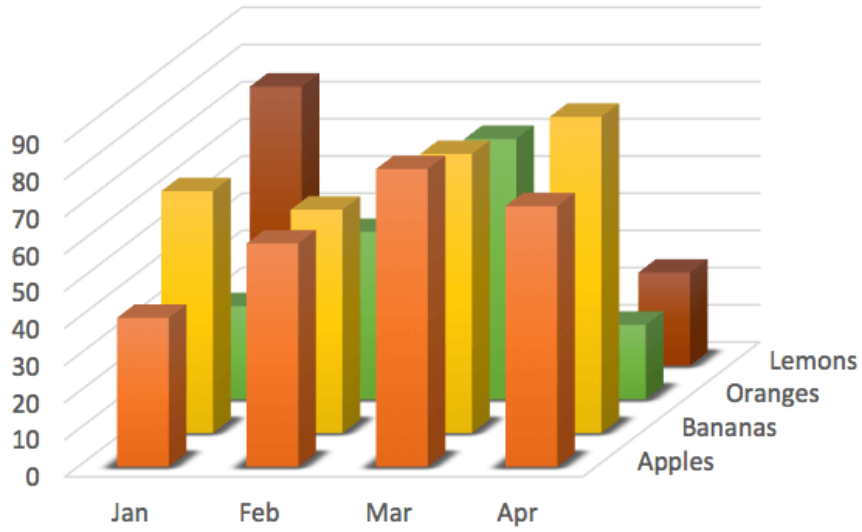

YouGov
@YouGov
Follow
Forget pepperoni - mushroom is Britain's most liked pizza topping (65%), followed by onion (62%) and then ham (61%)
yougov.co.uk/news/2017/03/0 ...
Generally speaking, which of the following toppings do you like on a pizza? Select as many as you like
42% Sweetcorn
60% Peppers
49% Bacon
56% Pepperoni
26% Spinach
33% Olives
51% Tomato (as a topping)
42% Pineapple
61% Ham
65% Mushrooms
Other items not depicted include: onions (62%), chicken (56%), beef (36%), chillies (31%), jalapeños (30%), pork (25%), tuna (22%), anchovies (18%). 2% of people say they only like Margherita pizzas
4:00 AM - 6 Mar 2017
364 Retweets 549 Likes
179
364
549


AVERAGE NUMBER OF DEATHS PER DAY
on that date, the day before and the day after
1,200
1,000
800
600
400
200
0
(200)
(150)
(100)
(50)
0
50
100
150
200
250
300
INCREASE OR DECREASE IN DEATHS PER DAY
(smoothed rate of change from the date before to the date after date shown)
France - police enforce lockdown two weeks before April 2nd
USA - still flying internal flights - BUT New York lockdown began two weeks before April 5th
Full UK lockdown began two weeks before April 7th
Spain - lockdown from 2 weeks before Sunday 29th March
Italy - most of northern Italy covered by quarantine two weeks before Sunday 22nd March 2020
Germany - lockdown introduced in Bavaria two weeks before April 3rd
UK - schools, pubs and restaurants from two weeks before Friday 3rd April
USA
France
Spain
Italy
UK
Eng & Wales
China
Germany
DannyDorling.org. Illustration by Kirsten McClure @orpheuscat


Massive Increase in Home Price
Average House Price in €
82500
82000
81500
81000
80500
80000
79500
79000
1998
1999


Computation Scores
97
96.5
96
95.5
Danielle
Jared
Alexi


HOW BABY BOOMERS DESCRIBE THEMSELVES
LEADERS
40%
WILLING TO LEARN
61%
TECH-SAVVY
22%
I HAVE A SMART PHONE, DON'T DISCOUNT MY TECHNICAL ABILITIES.
PEOPLE-SAVVY
78%
CREATIVE
42%
MY WORK EXPERIENCE HAS MADE ME RESOURCEFUL.


90
80
70
60
50
40
30
20
10
0
Jan
Feb
Mar
Apr
Lemons
Oranges
Bananas
Apples

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"). Commend 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 Achor 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%

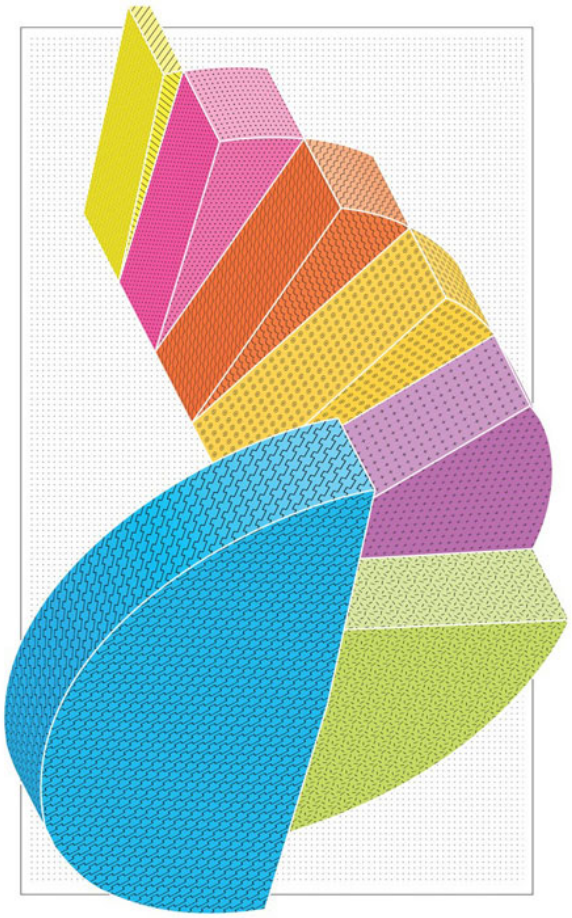
Personal Failure

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

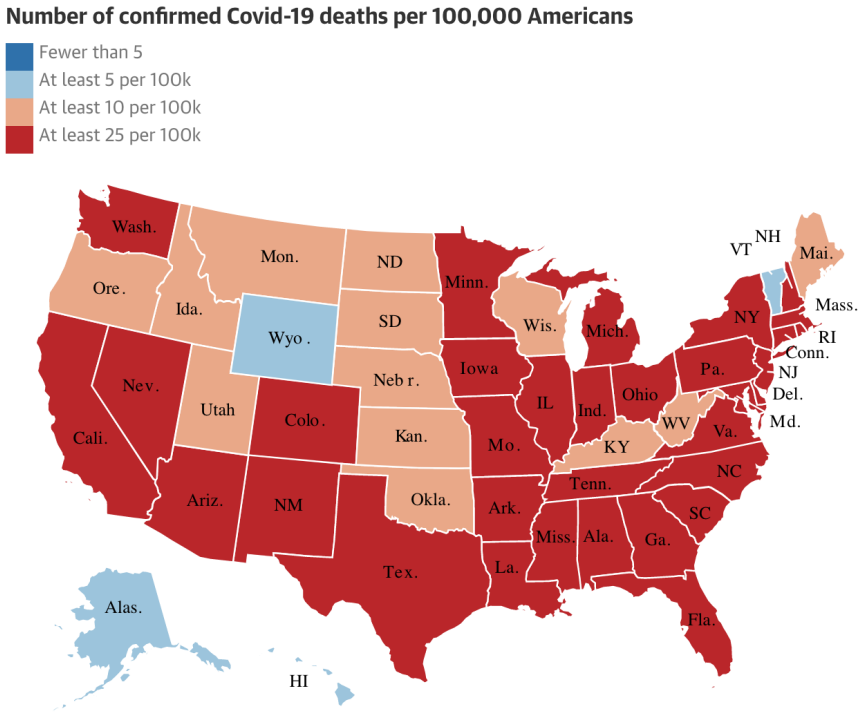
49%

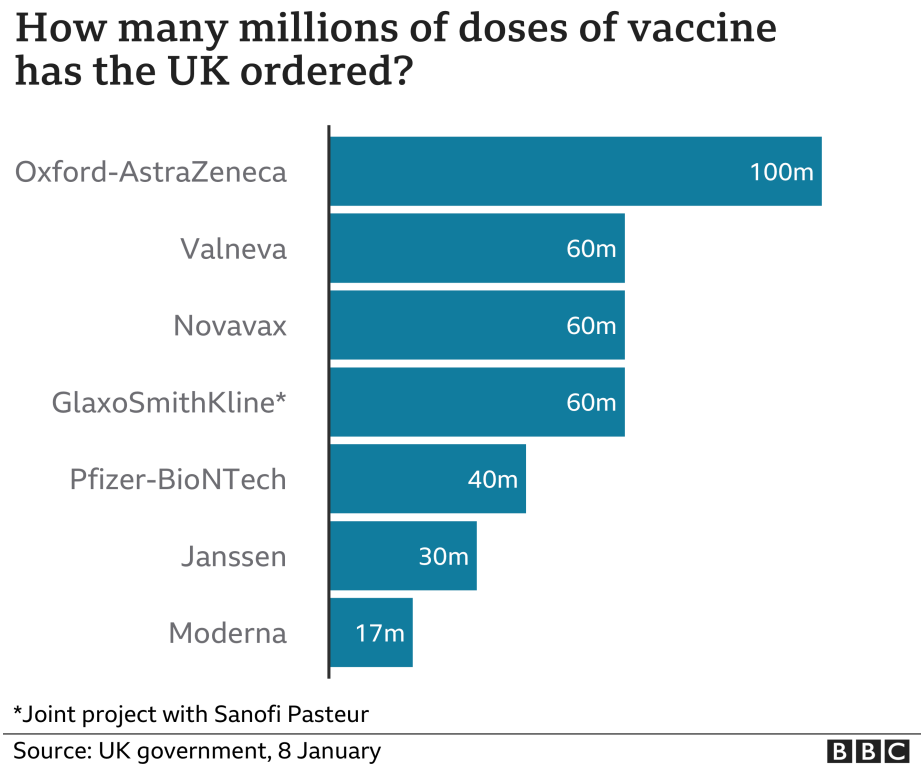
Contrarian Thesis

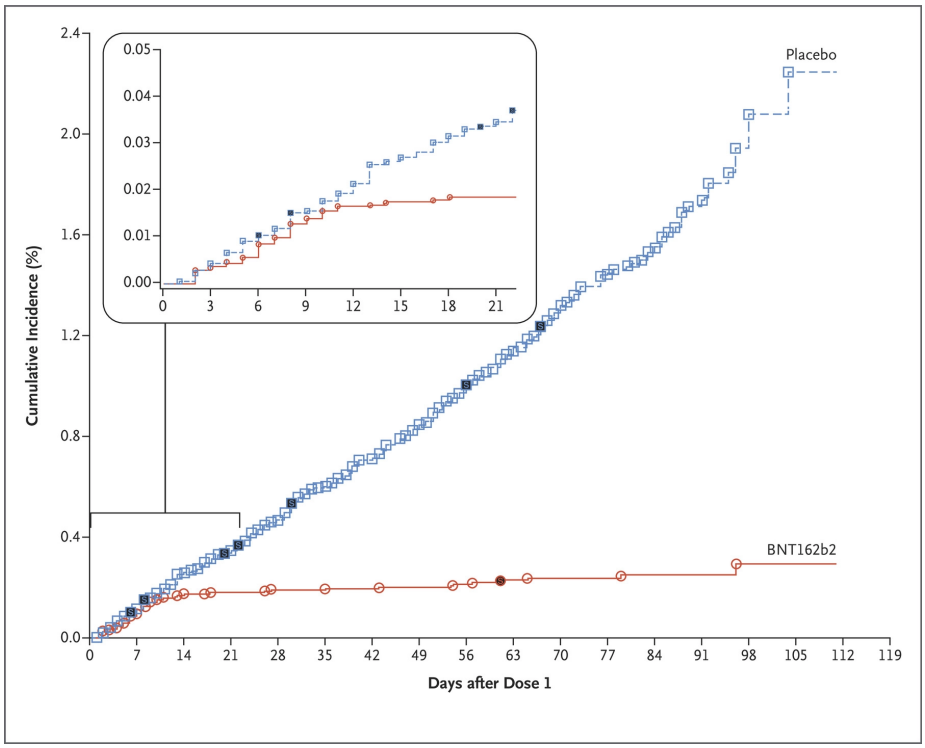
Wait a sec—we should be playing *more* videogames? The more choices we have, the worse off we are? TED is where conventional wisdom goes to die.

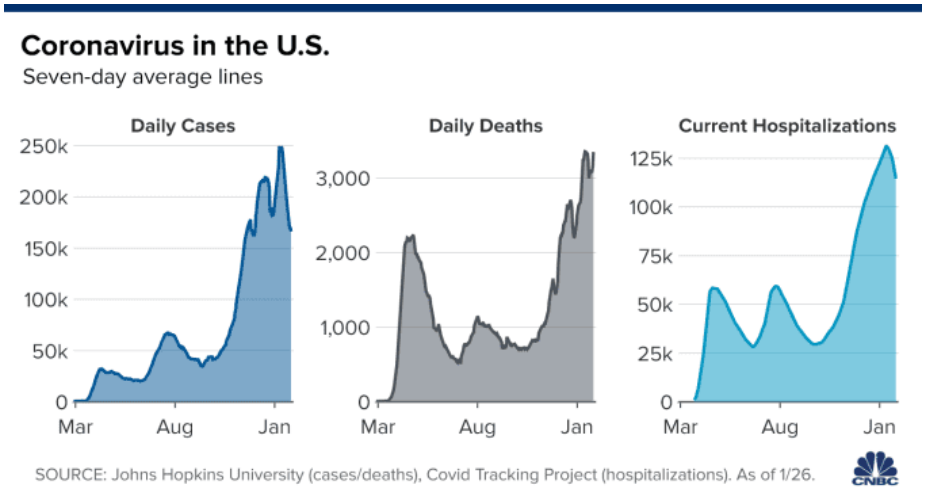


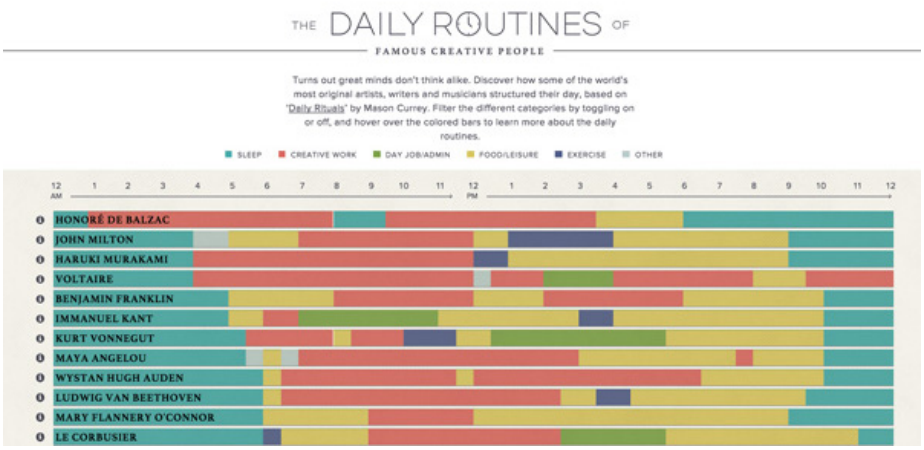
3.3 Good exaples

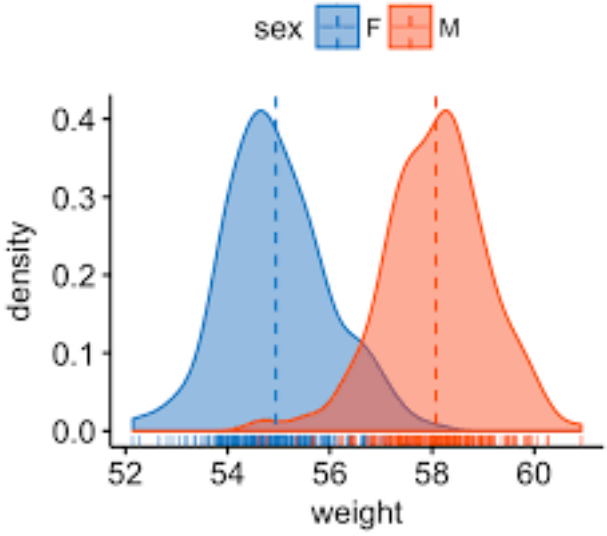


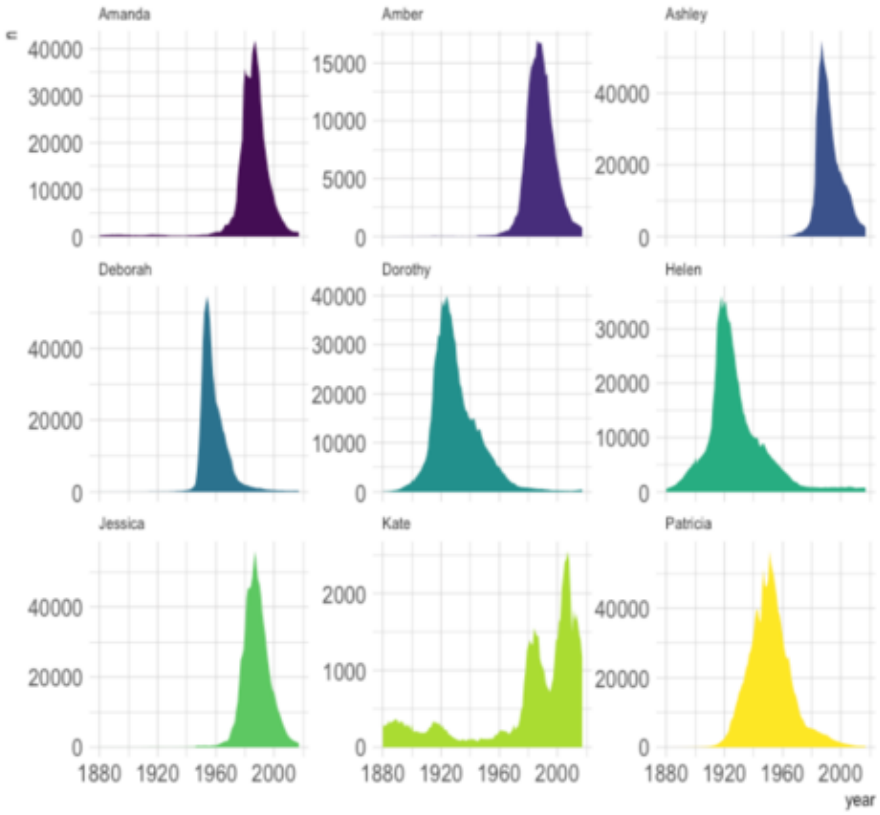

How many millions of doses of vaccine has the UK ordered?
Oxford-AstraZeneca
100m
Valneva
60m
Novavax
60m
GlaxoSmithKline*
60m
Pfizer-BioNTech
40m
Janssen
30m
Moderna
17m
*Joint project with Sanofi Pasteur
Source: UK government, 8 January
BBC

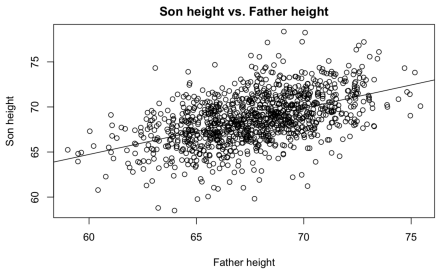

Placebo
BNT162b2
Cumulative Incidence (%)
Days after Dose 1
0.0
0.4
0.8
1.2
1.6
2.0
2.4
0
7
14
21
28
35
42
49
56
63
70
77
84
91
98
105
112
119
0.00
0.01
0.02
0.03
0.04
0.05
0
3
6
9
12
15
18
21

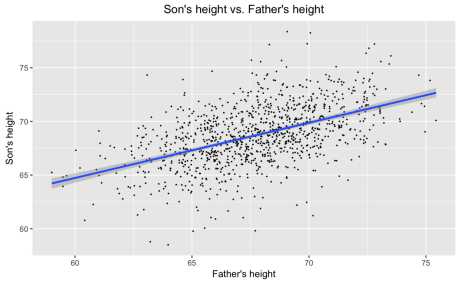

Coronavirus in the U.S.
Seven-day average lines
Daily Cases
250k
200k
150k
100k
50k
0
Mar
Aug
Jan
Daily Deaths
3,000
2,000
1,000
0
Mar
Aug
Jan
Current Hospitalizations
125k
100k
75k
50k
25k
0
Mar
Aug
Jan
SOURCE: Johns Hopkins University (cases/deaths), Covid Tracking Project (hospitalizations). As of 1/26.
CNBC

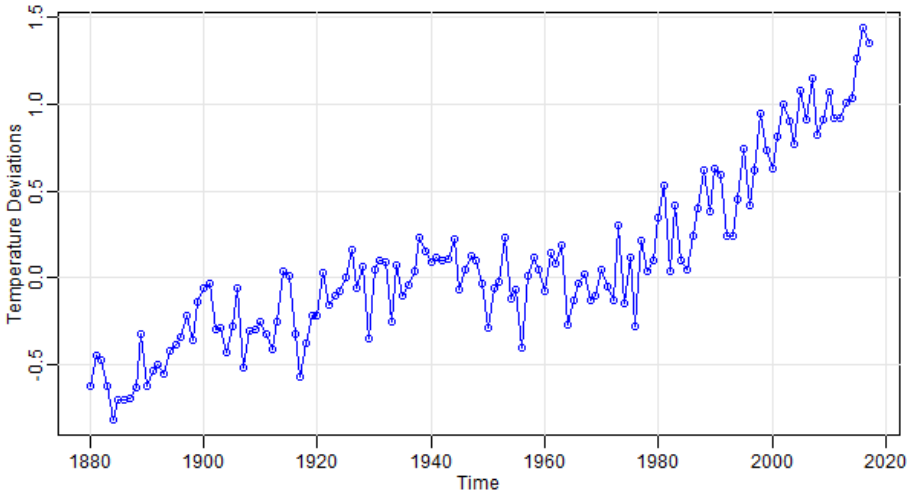

THE DAILY ROUTINES OF
FAMOUS CREATIVE PEOPLE
Turns out great minds don't think alike. Discover how some of the world's most original artists, writers and musicians structured their day, based on "Daily Rituals" by Mason Currey. Filter the different categories by toggling on or off, and hover over the colored bars to learn more about the daily routines.
SLEEP
CREATIVE WORK
DAY JOB/ADMIN
FOOD/LEISURE
EXERCISE
OTHER
12 AM 1 2 3 4 5 6 7 8 9 10 11 12 PM 1 2 3 4 5 6 7 8 9 10 11 12
HONORÉ DE BALZAC
JOHN MILTON
HARUKI MURAKAMI
VOLTAIRE
BENJAMIN FRANKLIN
IMMANUEL KANT
KURT VONNEGUT
MAYA ANGELOU
WYSTAN HUGH AUDEN
LUDWIG VAN BEETHOVEN
MARY FLANNERY O'CONNOR
LE CORBUSIER


sex
F
M
density
0.4
0.3
0.2
0.1
0.0
52
54
56
58
60
weight


Amanda
Amber
Ashley
Deborah
Dorothy
Helen
Jessica
Kate
Patricia
n
year
1880 1920 1960 2000


Son height vs. Father height
Son height
Father height


Son's height vs. Father's height
Son's height
Father's height



3.4 Quick video

Let's watch a quick video on data visualization here.

(PART) Getting started

Chapter 4

Setting up `RStudio`

First, download and install `R`:

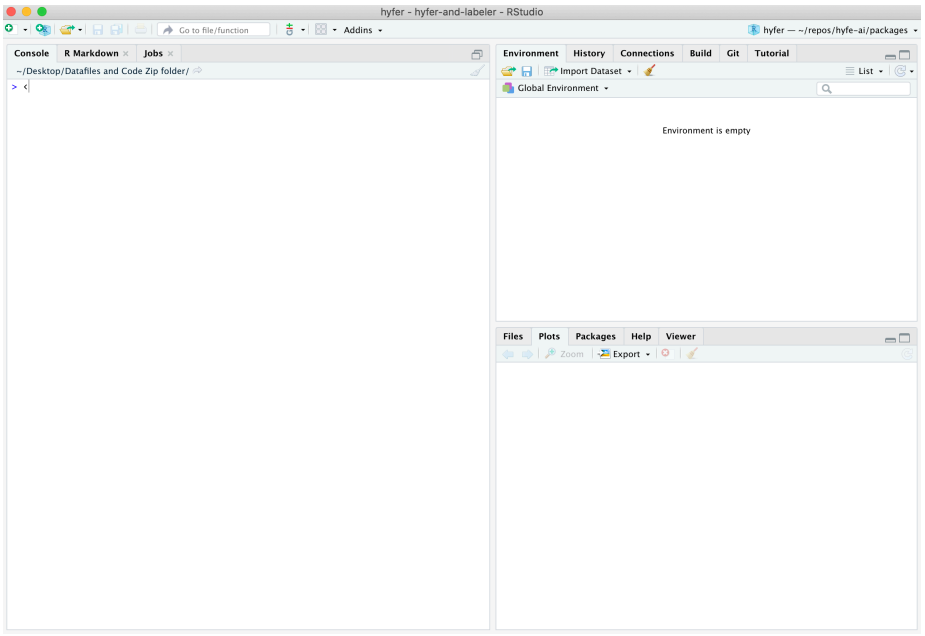
Go to the following website, click the *Download* button, and follow the website's instructions from there. https://mirrors.nics.utk.edu/cran/

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:

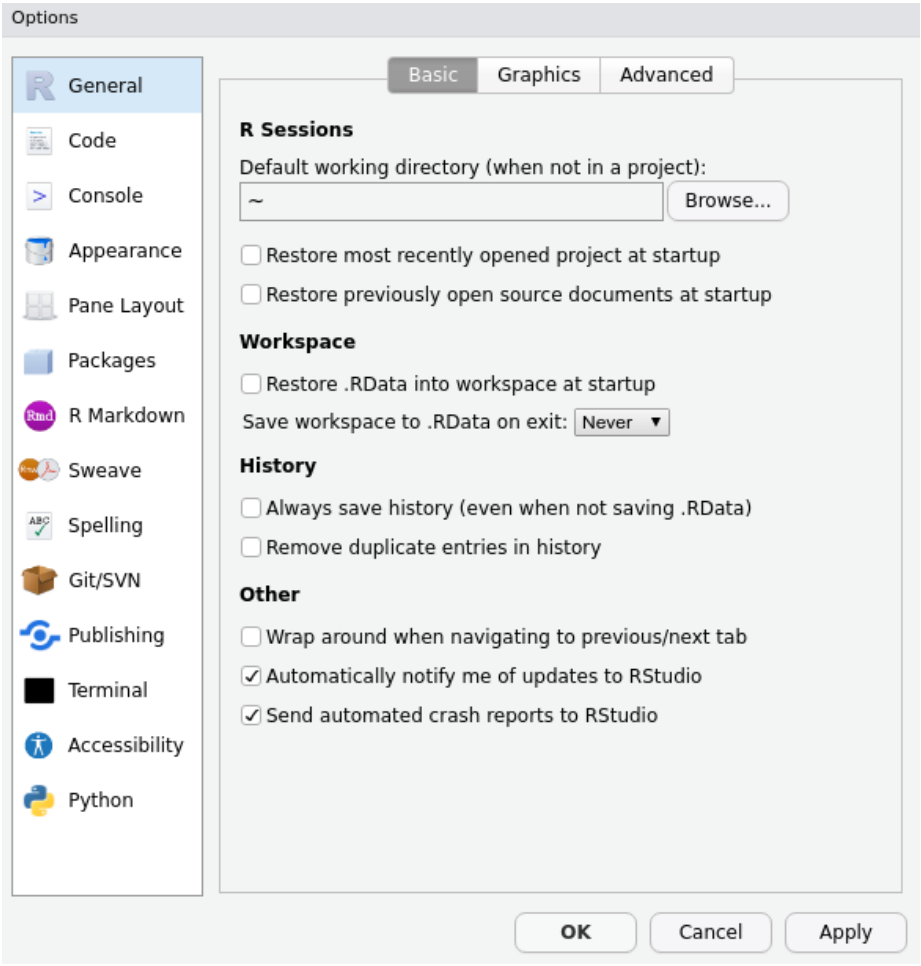


Fourth, make sure `R` is running correctly in the background:

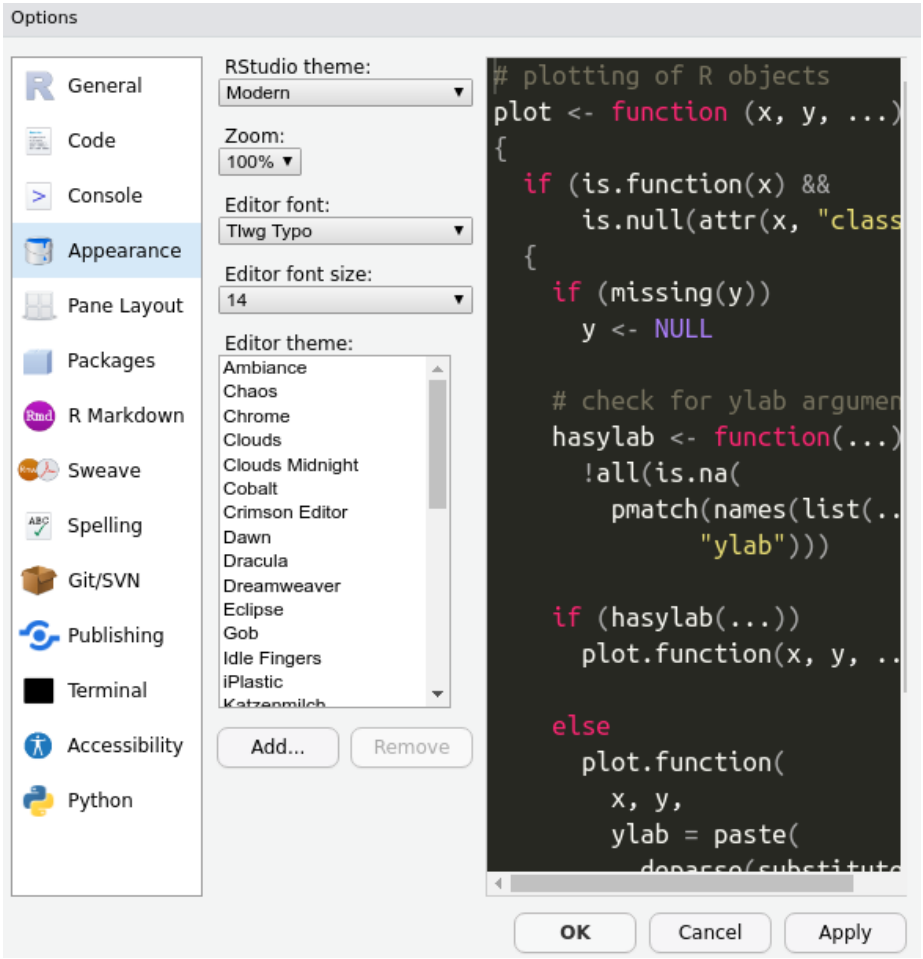
In `RStudio`, in the pane on the left (the "Console"), type 2+2 and hit Enter. If `R` is working properly, the number "4" will be printed in the next line down.

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

Tools > Global Options:



Cooler themes!


Options
General
Code
Console
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
Accessibility
Python
RStudio theme:
Modern
Zoom:
100%
Editor font:
Tlwg Typo
Editor font size:
14
Editor theme:
Ambiance
Chaos
Chrome
Clouds
Clouds Midnight
Cobalt
Crimson Editor
Dawn
Dracula
Dreamweaver
Eclipse
Gob
Idle Fingers
iPlastic
Add...
Remove
# plotting of R objects
plot <- function (x, y, ...)
{
if (is.function(x) &&
is.null(attr(x, "class
{
if (missing(y))
y <- NULL
# check for ylab argumen
hasylab <- function(...)
!all(is.na(
pmatch(names(list(..
"ylab")))
if (hasylab(...))
plot.function(x, y, ..
else
plot.function(
x, y,
ylab = paste(
OK
Cancel
Apply

Boom!

Chapter 5

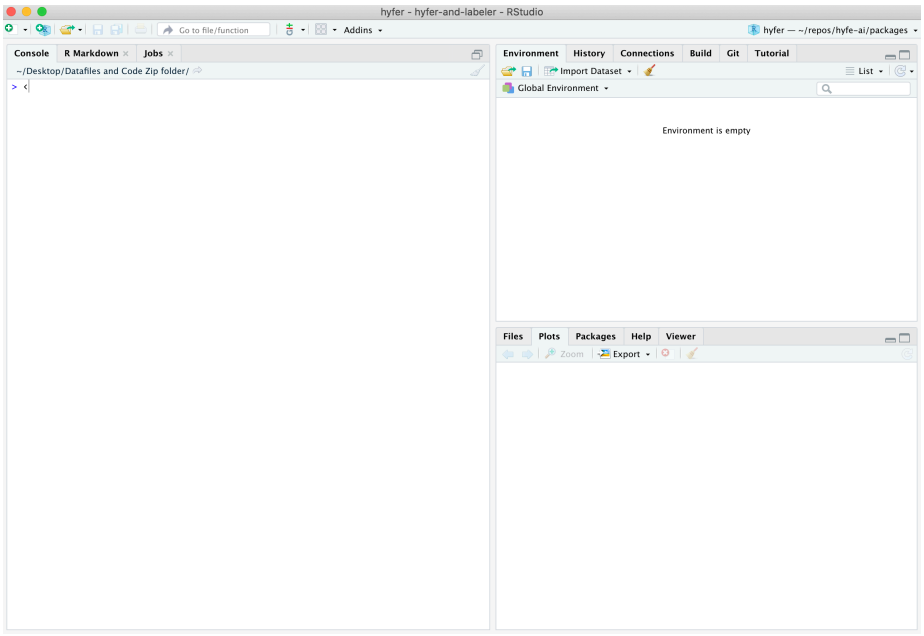
Running R code

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

RStudio's *Console*

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.

Running code in the *Console*

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

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

When you press `Enter`, R processes the command you fed it, then returns its 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 spaces.

Exercise 1

Type a command in the *Console* to determine the sum of 596 and 198.

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.

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

Exercise 2

A. Re-run the sum of 596 and 198 without re-typing it.

B. Recall the command again, but this time adjust the code to find the sum of 596 and 298.

C. Practice escaping an accidentally called command: recall your most recent command, then clear the *Console*'s command line.

Incomplete commands in `R`

`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`).

If instead you want `R` to stop waiting and stop running, hit the `Escape` key.

Getting errors in R

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 + 6p
```

You probably received a message in red font stating `Error: unexpected symbol in "4 + 6p"`. That is because R did know how to interpret the symbol `p` in this case.

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.

Exercise 3

Type a command in R that throws an error, then recall the command and revise so that R can understand it.

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 straightfoward too:

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

For multiplication, use an asterisk (*):

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

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
```

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
```

Exercise 4

A. Find the sum of the ages of everyone in your immediate family.

B. Now recall that command and adjust it to determine the *average* age of the members of your family.

Using operators in R

You can get R to evaluate logical tests 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 (`>` and `<`), and greater/less than or equal to (`>=` and `<=`).

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

Exercise 5

A. 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
```

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

Use built-in functions within R

`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.

```
sqrt(16)
[1] 4

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
```

Exercise 6

Find the square root of *pi* and round the answer to the 2 decimal places.

Review assignment:

Show that the following statements are TRUE:

A. `pi` is greater than the square root of 9

B. It is FALSE that the square root of 9 is greater than `pi`

C. `pi` rounded to the nearest whole number equals the square root of 9

Other Resources

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

Chapter 6

Using `RStudio` and `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 a 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),

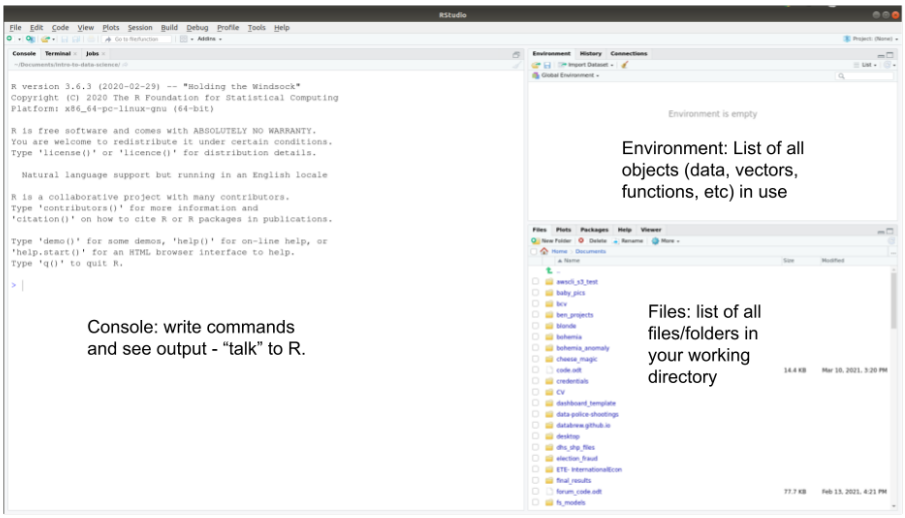
R is just so much more pleasant to use when you are operating it from within RStudio.

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.

This is the pane that is used the least often, and if you wish it can simplify your workspace to minimize it.

Files, Plots, Packages, & Help

You will use the bottom right pane very often.

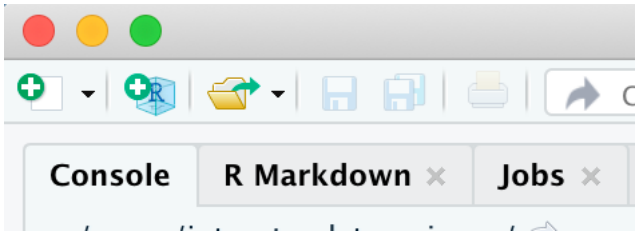
- The **Files** tab lets you see all the files within your **working directory**, which will be explained in the section below.
- The **Plots** tab lets you see the plots you are producing with your code.
- The **Packages** tab lets you see the *packages* you currently have installed on your computer. 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

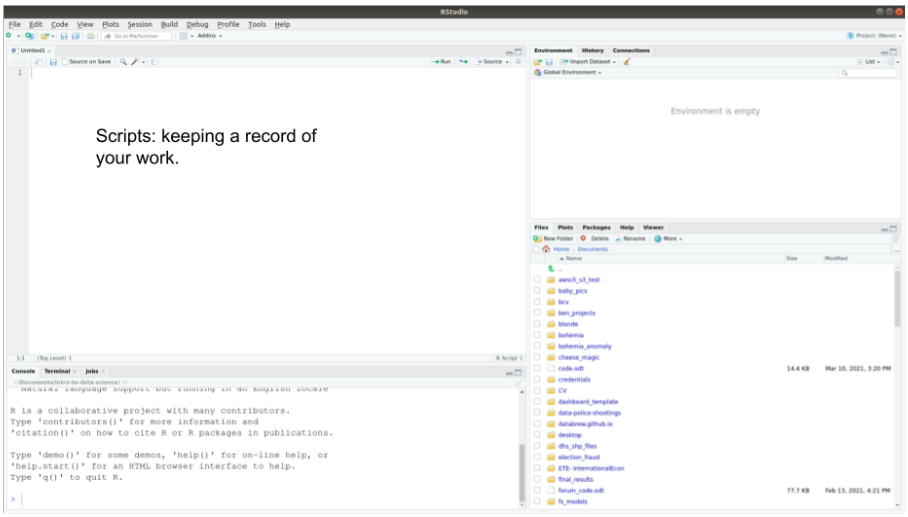
Before explaining what scripts are and why they are awesome, let's start a new script.

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`.)

Exercise 1

Add a few more lines to your script, such that your script now looks like this.

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

(A) Run all of these lines of code at once.

(B) 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* of *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 anR script, you can reproduce your own work whenever you want to. You can send your script to a colleague so that they can reproduct 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.)

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. We suggest making a new folder on your Desktop and naming it `databoom`, but you can name it whatever you want and place it wherever you want.

How to save your script? 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 `09_intro_to_rstudio`.

(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] "/home/benbrew88/Documents/intro-to-data-science"
```

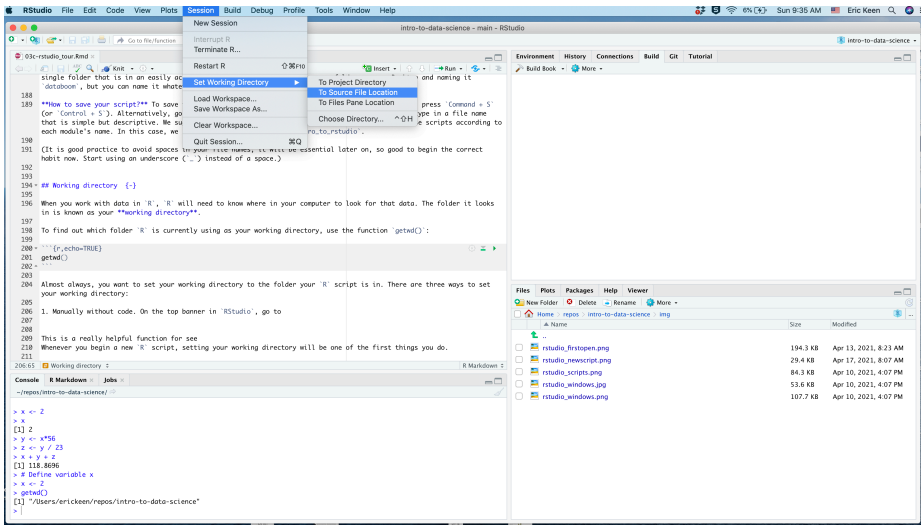
Almost always, you want to set your working directory to the folder your R script is in.

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*:

(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.

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)
```

Review assignment:

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

Part 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.

Part 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.

Part 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 `10_variables.R` (because the next module is called *Module 10: Variables in `R`*).

Part 5. Modify the code in `10-variables.R` so that you are prepared to begin the next module. Change the title, and look ahead to *Module 10* 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 7

Variables in R

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:

```
x <- 3*12 ; x
[1] 36
```

The semicolon simulates hitting `Enter`. It says: first run `x <- 3*12`, then run `x`.

You can also update variables.

```
x <- x * 3 ; x
[1] 108
x <- x * 3 ; x
[1] 324
```

You can also add variables together.

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

Naming variables

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.

You can make variable names as complicated or simple as you want.

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

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.

Also note that variables are case-sensitive! If you name a variable `My_variable`, `R` will not recognize it if you refer to it as `My_Variable`.

Naming theory

Naming variables 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.

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.

Note that `R` uses a feature called 'Tab complete' to help you type variable names. Begin typing a variable name, such as `supercalifragilistic.expialidocious` 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.

Exercise 1

A. Estimate how many bananas you've eaten in your lifetime and store that value in a variable (choose whatever name you wish).

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

C. 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.

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

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.

Exercise 2

A. Assign a `FALSE` statement of your choosing to a variable of whatever name you wish.

B. Confirm that the class of this variable is "logical."

C. Confirm that the variable equals `FALSE`.

Exercise 3

A. Create a variable x that is a list of numbers of any size. Create a variable y with the same criteria.

B. Check to see if each values of x is greater than each value of y

C. Check to see if the smallest value of x is greater than or equal to the average value of y.

Review assignment

Converting Farenheit to Celsius:

A. Assign a variable `farenheit` the numerical value of 32.

B. Assign a variable `celsius` to equal the conversion from Fahrenheit to Celsius. Unless your a meteorology nerd, you may need to Google equation for this conversion.

C. Print the value of `celsius` to the *Console*.

D. Now use this code to determine the *Celsius* equivalent of 212 degrees *Farenheit*.

Chapter 8

Structures for data in R

Learning goals

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

Introducing data structures

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.

Vectors

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
```

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
```

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
```

You can also put vectors through logical tests:

```
x <- 1:5
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"
```

Exercise 1

A. Create a vector with at least one number, character (string) and logical value

B. Identify the class of each element of the vector

Useful functions for handling vectors

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

```
x <- c(5,6)
y <- c(9,10,11,12)

length(x)
[1] 2
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!"
```

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
```

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)
```

%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
```

Exercise 2

A. Use the colon symbol to create a vector of length 5 between a minimum and a maximum value of your choosing.

B. Create a second vector of length 5 using the `seq()` function.

C. Create a third vector of length 5 using the `rep()` function.

D. Finally, concatenate the three vectors and check that the length makes sense.

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 `[ ]`.

```
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`.

Exercise 2

A. Figure out how to replicate the `head()` function using your new vector subsetting skills.

B. Now replicate the `tail()` function, using those same skills as well as the `length()` function you just learned.

Review assignment

A. Create a vector called "sleep_time" with the number of hours you slept for each day in the last week.

B. Check if you slept more on day 3 than day 7

C. Get the total number of hours slept in the last week

D. Get the average number of hours slept in the last week

E. 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

F. 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)

G. 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)

H. Write code to get the minimum number of hours you slept this week.

I. Write code to calculate how many hours of sleep you would have gotten had you sleep the minimum number of hours every night.

J. Write code to calculate the average of the hours of sleep you got on the 3rd through 6th days of the week.

K. 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.

L. Write code to calculate how many hours of sleep per year someone who sleeps 8 hours a night gets.

M. 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?

Chapter 9

Calling functions

Learning goals

- Understand what functions are, and why they are awesome
- Understand how functions work
- Understand how to read function documentation

Introducing `R` functions

You have already worked with many `R` functions; commands like `getwd()`, `length()`, and `unique()` are all functions. You know a command is a function because it has parentheses, `()`, attached at its end.

Just as **variables** are convenient names used for calling *objects* such as vectors or dataframes, **functions** are convenient names for calling *processes* or *actions*. An `R` function is just a batch of code that performs a certain action.

Variables represent data, while functions represent code.

Most functions have three key components: (1) one or more inputs, (2) a process that is applied to those inputs, and (3) an output of the result. When you call a function in `R`, you are saying, "Hey `R`, take this information, do something to it, and return the result to me." You supply the function with the inputs, and the function takes care of the rest.

Take the function `mean()`, for example. `mean()` finds the arithmetic mean (i.e., the average) of a set of values.

```
x <- c(4,6,3,2,6,8,5,3) # create a vector of numbers
mean(x) # find their mean
[1] 4.625
```

In this command, you are feeding the function `mean()` with the input `x`.

Base functions in R

There are hundreds of functions already built-in to R. These functions are called *"base functions"*. Throughout these modules, we have been – and will continue – introducing you to the most commonly used base functions.

You can access other functions through bundles of external code known as *packages*, which we explain in an upcoming module.

You can also write your *own* functions (and you will!). We provide an entire module on how to do this.

Note that not all functions require an input. The function `getwd()`, for example, does not need anything in its parentheses to find and return current your working directory.

Saving function output

You will almost always want to save the result of a function in a new variable. Otherwise the function just prints its result to the *Console* and R forgets about it.

You can store a function result the same way you store any value:

```
x <- c(4,6,3,2,6,8,5,3)
x_mean <- mean(x)
x_mean
[1] 4.625
```

Functions with multiple inputs

Note that `mean()` accepts a second input that is called `na.rm`. This is short for `NA.remove`. When this is set to `TRUE`, R will remove broken or missing values from the vector before calculating the mean.

```
x <- c(4,6,3,2,NA,8,5,3)  # note the NA
mean(x,na.rm=TRUE)
[1] 4.428571
```

If you tried to run these commands with `na.rm` set to `FALSE`, R would throw an error and give up.

Note that you provided the function `mean()` with two inputs, `x` and `na.rm`, and that you separated each input with a comma. This is how you pass multiple inputs to a function.

Function defaults

Note that many functions have default values for their inputs. If you do not specify the input's value yourself, `R` will assume you just want to use the default. In the case of `mean()`, the default value for `na.rm` is `FALSE`. This means that the following code would throw an error …

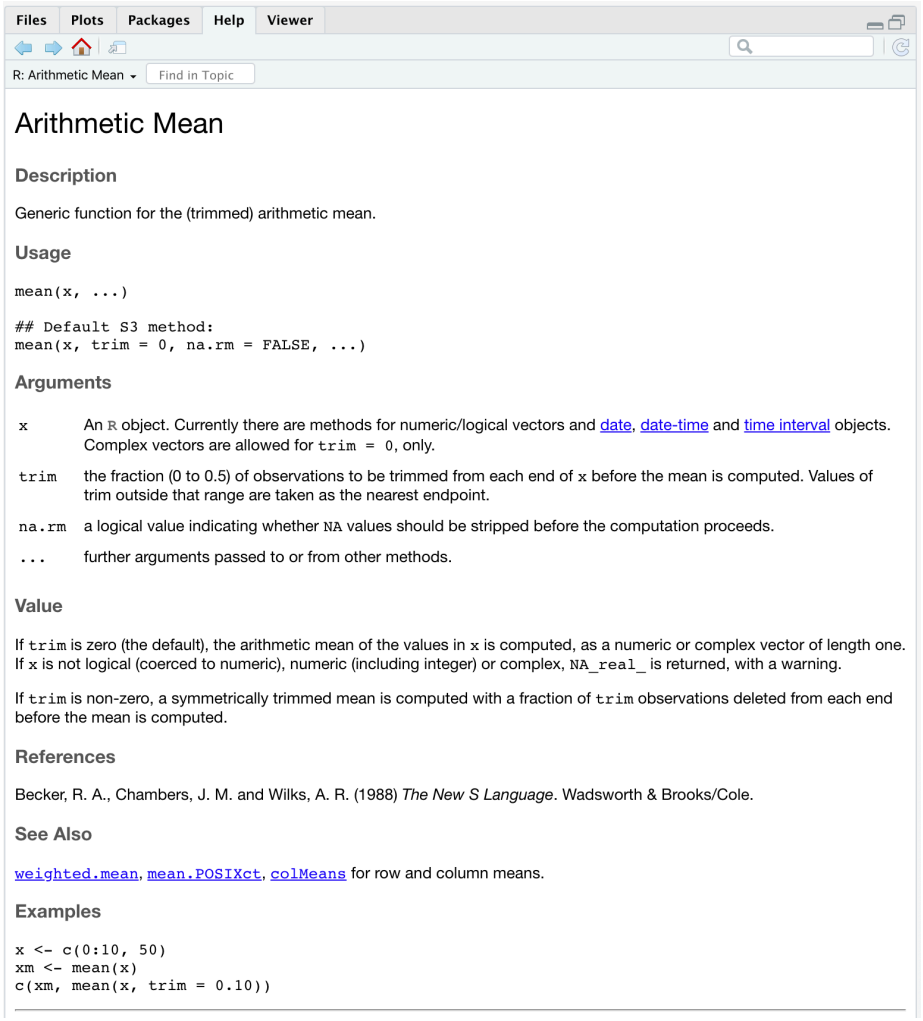
```
x <- c(4,6,3,2,NA,8,5,3)  # note the NA
mean(x)
[1] NA
```

Because `R` will assume you are using the default value for `na.rm`, which is `FALSE`, which means you do not want to remove missing values before trying to calculate the mean.

Function documentation (i.e., getting help)

Functions are designed to accept only a certain number of inputs with only certain names. To figure out what a function expects in terms of inputs, and what you can expect in terms of output, you can call up the function's help page:

When you enter this command, the help documentation for `mean()` will appear in the bottom right pane of your `RStudio` window:



Files Plots Packages Help Viewer

R: Arithmetic Mean — Find in Topic

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

weighted.mean, mean.POSIXct, colMeans for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

Learning how to read this documentation is essential to becoming competent in using R.

Be warned: not all documentation is easy to understand! You will come to really resent poorly written documentation and really appreciate well-written documentation; the few extra minutes taken by the function's author to write good documentation saves users around the world hours of frustration and confusion.

- The `Title` and `Description` help you understand what this function does.
- The `Usage` section shows you how type out the function.
- The `Arguments` section lists out each possible argument (which in R lingo is another word for *input* or *parameter*), explains what that input is asking

for, and details any formatting requirements.

- The `Value` section describes what the function returns as output.
- At the bottom of the help page, example code is provided to show you how the function works. You can copy and paste this code into your own script of *Console* and check out the results.

Note that more complex functions may also include a `Details` section in their documentation, which gives more explanation about what the function does, what kinds of inputs it requires, and what it returns.

Function examples

`R` comes with a set of base functions for descriptive statistics, which provide good examples of how functions work and why they are valuable.

We can use the same vector as the input for all of these functions:

```
x <- c(4,6,3,2,NA,8,9,5,6,1,9,2,6,3,0,3,2,5,3,3)  # note the NA
```

`mean()` has been explained above.

```
result <- mean(x,na.rm=TRUE)
result
[1] 4.210526
```

`median()` returns the median value in the supplied vector:

```
result <- median(x,na.rm=TRUE)
result
[1] 3
```

`sd()` returns the standard deviation of the supplied vector:

```
result <- sd(x,na.rm=TRUE)
result
[1] 2.594416
```

`summary()` returns a vector that describes several aspects of the vector's distribution:

```
result <- summary(x,na.rm=TRUE)
result
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  0.000   2.500   3.000   4.211   6.000   9.000       1
```

Review assignment

A. Create a vector named `years` with the years of birth of everyone in the room.

B. What is the average year of birth?

C. What is the median year of birth?

D. Create a vector called `ages` which is the (approximate) age of each person.

E. What is the minimum age?

F. What is the maximum age?

G. What is the median age?

H. What is the average age?

I. "Summarize" `ages`.

J. What is the range of ages?

K. What is the standard deviation of ages?

L. Look up help on the function `sort`.

M. Created a vector called `sorted_ages`. It should be, well, sorted ages.

N. Look up the `length` function.

O. How many people are the group?

P. Create an object called `old`. Assign to this object a number (such as 36) at which someone becomes "old".

Q. Create an object called `old_people`. This should be a boolean/logical vector indicating if each person is old or not.

R. Is person 7 old?

S. How many years from being old/young is person 12?

Chapter 10

Subsetting and filtering

Learning goals

- Understand how to subset / filter

A quick review of booleans

Recall that a boolean / logical data type is either `TRUE` or `FALSE`. For example:

```
joes_age <- 35
old_age <- 36
joe_is_old <- joes_age >= old_age
joe_is_old
```

```
[1] FALSE
```

Recall also that you can calculate whether a condition is true or false on multiple elements of a vector. For example:

```
ages <- c(10, 20, 30, 40, 50, 60)
old_age <- 36
people_are_old <- ages >= old_age
people_are_old
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Boolean vectors are useful for subsetting, ie keeping those elements of a vector for which a condition is `TRUE`. You have already done this using indexes, for example:

```
ages[3]
```

```
[1] 30
```

But you can also do this using booleans. For example:

```
foo <- 1:4
foo[2:4]
```

```
[1] 2 3 4
```

```
foo[c(2,3,4)]
```

```
[1] 2 3 4
```

```
foo[c(FALSE, TRUE, TRUE, TRUE)]
```

```
[1] 2 3 4
```

```
foo[foo >= 2]
```

```
[1] 2 3 4
```

Review assignment

A. Create a vector named `nummies` of all numbers from 1 to 100

B. Create another vector named `little_nummies` which consists of all those numbers which are less than or equal to 30

C. Create a boolean vector named `these_are_big` which indicates whether each element of `nummies` is greater than or equal to 70

D. Use `these_are_big` to subset `nummies` into a vector named `big_nummies`

E. Create a new vector named `these_are_not_that_big` which indicates whether each element of `nummies` is greater than 30 and less than 70. You'll need to use the `&` symbol.

F. Create a new vector named `meh_nummies` which consists of all `nummies` which are greater than 30 and less than 70.

G. How many numbers are greater than 30 and less than 70?

H. What is the sum of all those numbers in `meh_nummies`

Chapter 11

Base plots

Learning goals

- Make basic plots in R
- Basic adjustments to plot formatting

Introduction

To learn how to plot, let's first create a dataset to work with:

```
country <- c("USA","Tanzania","Japan","Ctr. Africa Rep.","China","Norway","India")
lifespan <- c(79,65,84,53,77,82,69)
gdp <- c(55335,2875,38674,623,13102,84500,6807)
```

These data come from this publicly available database that compares health and economic indices across countries in 2011.

The `lifespan` column presents the average life expectancy for each country.

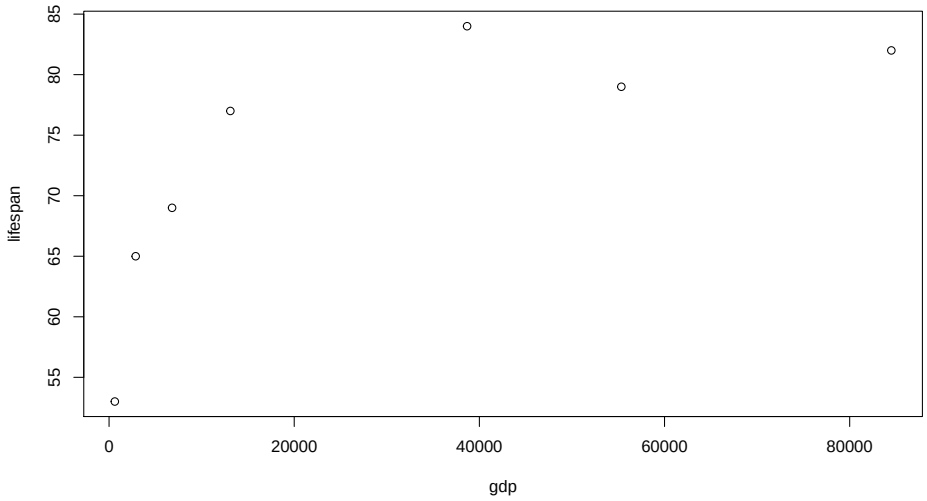
The `gdp` column presents the average GDP per capita within that country, which is a common index for the income and wealth of average citizens.

Let's see if there is a relationship between life expectancy and income.

Create a basic plot

The simplest way to make a basic plot in R is to use its built-in `plot()` function:

```
plot(lifespan ~ gdp)
```



This syntax is saying this: plot column `lifespan` as a function of `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Note that `R` uses the variable names you provided as the x- and y-axes. You can adjust these labels however you wish (see formatting section below).

You can also produce this exact same plot using the following syntax:

```
plot(y=lifespan, x=gdp)
```

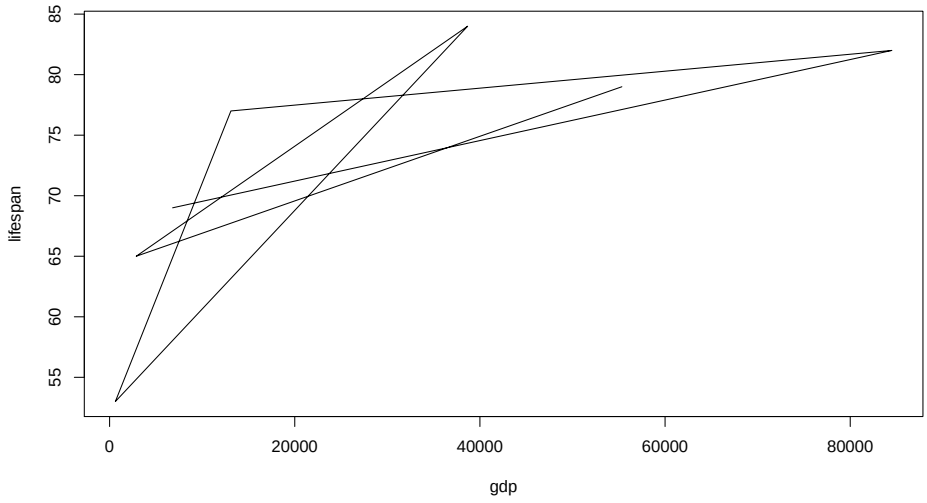
Choose whichever one is most intuitive to you.

Most common types of plots

The plot above is a **scatter plot**, and is one of the most common types of plots in data science.

You can turn this into a **line plot** by adding a parameter to the `plot()` function:

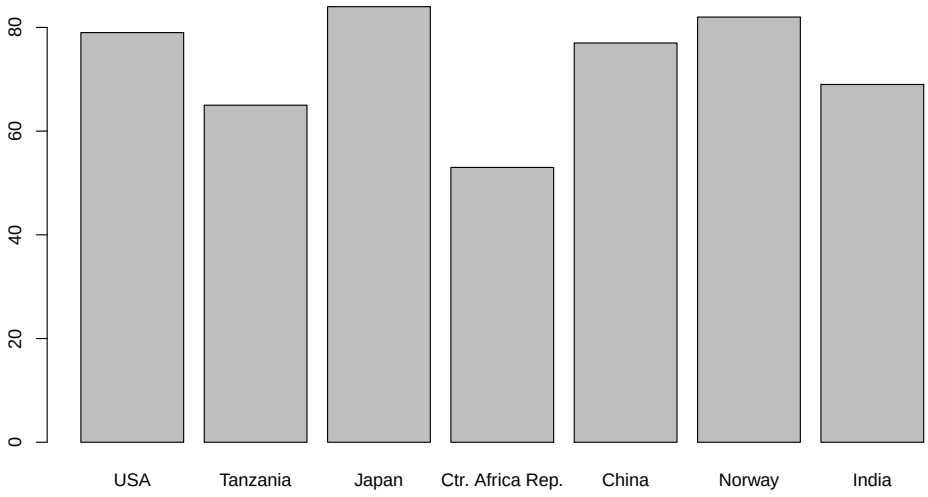
```
plot(lifespan ~ gdp, type="l")
```



What a mess! Rather than connecting these values in the order you might expect, `R` connects them in the order that they are listed in their source vectors. This is why line plots tend to be more useful in scenarios such as time series, which are inherently ordered.

Another common plot is the **bar plot**, which uses a different `R` function:

```
barplot(height=lifespan,names.arg=country)
```



In this command, the parameter `height` determines the height of the bars, and `names.arg` provides the labels to place beneath each bar.

There are many more plot types out there, but let's stop here now.

Exercise 1

Produce a bar plot that shows the GDP for each country.

Basic plot formatting

You can adjust the default formatting of plots by adding other inputs to your `plot()` command. To understand all the parameters you can adjust, bring up the help page for this function:

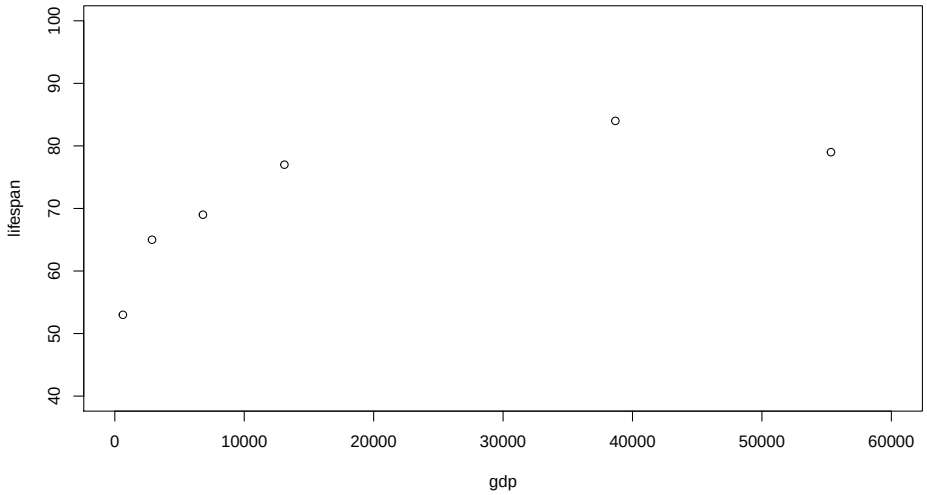
```
?plot
```

If multiple help page options are returned, select the *Generiz X-Y Plotting* page from the `base` package. This is the plot function that comes built-in to `R`.

Here we demonstrate just a few of the most common formatting adjustments you are likely to use:

Set plot range using `xlim` (for the x axis) and `ylim` (for the y axis):

```
plot(lifespan ~ gdp,xlim=c(0,60000),ylim=c(40,100))
```



In this command, you are defining axis limits using a 2-element vector (i.e., `c(min,max)`).

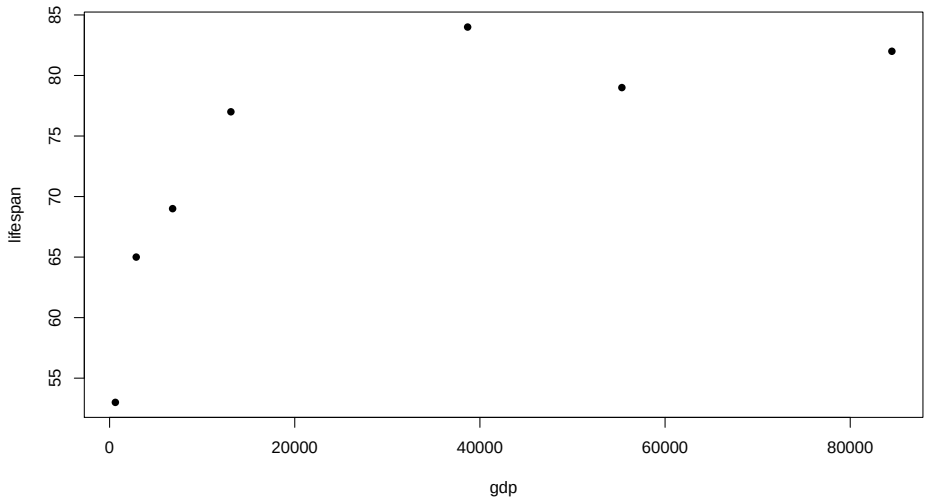
Note that it can be easier to read your code if you put each input on a new line, like this:

```
plot(lifespan ~ gdp,
     xlim=c(0,60000),
     ylim=c(40,100))
```

Make sure each input line within the function ends with a comma, otherwise you R will get confused and throw an error.

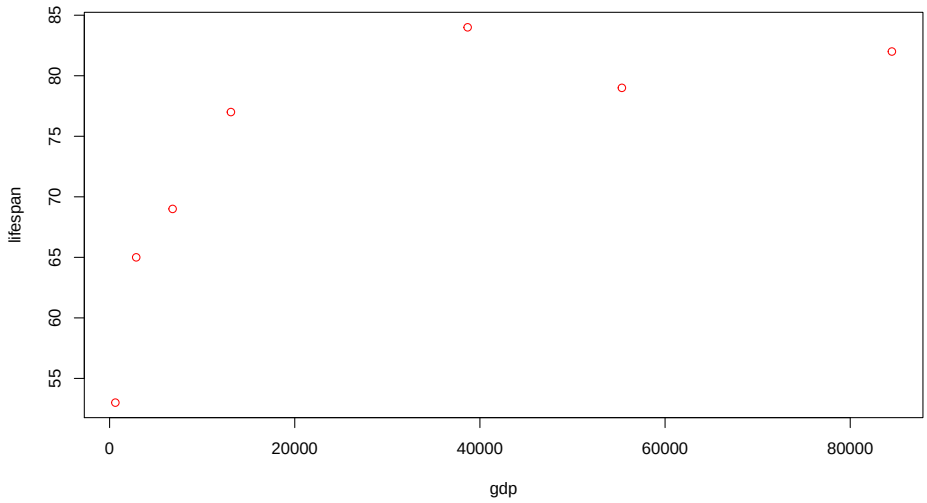
Set dot type using the input `pch`:

```
plot(lifespan ~ gdp,pch=16)
```



Set dot color using the input `col` (the default is `col="black"`)

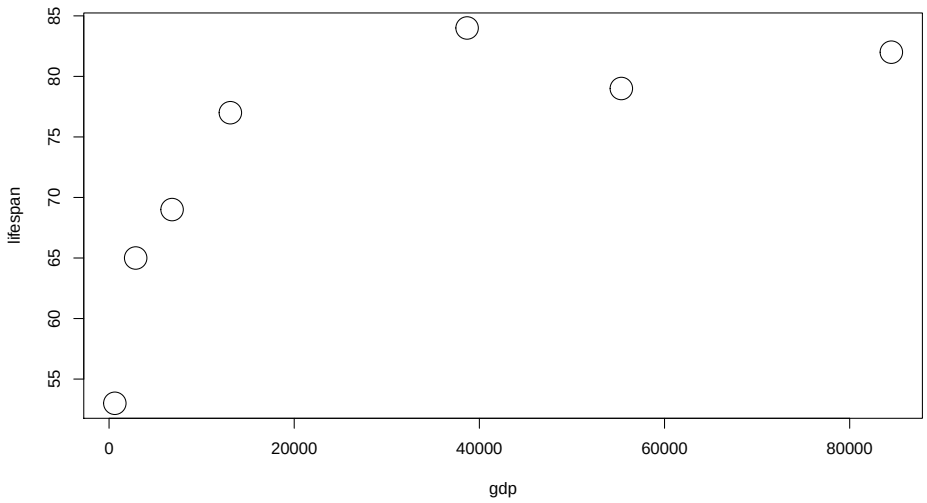
```
plot(lifespan ~ gdp,col="red")
```



Here is a great resource for color names in R.

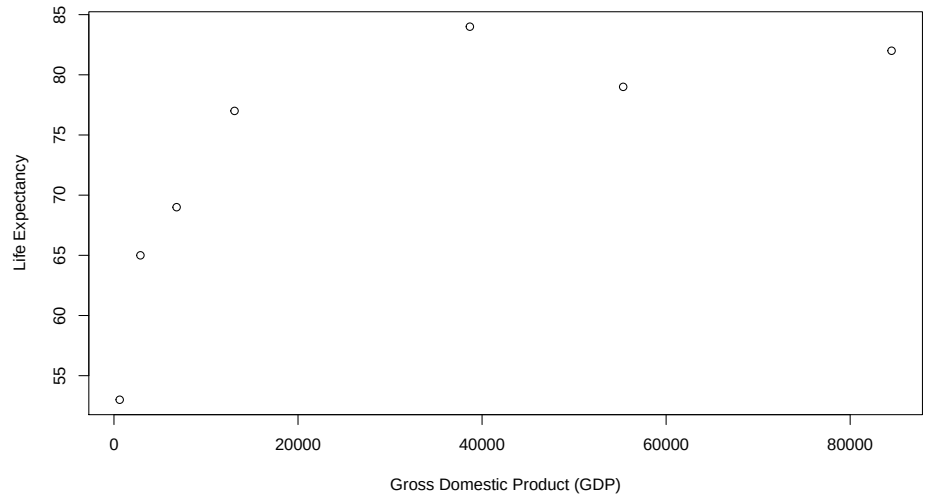
Set dot size using the input `cex` (the default is `cex=1`):

```
plot(lifespan ~ gdp,cex=3)
```



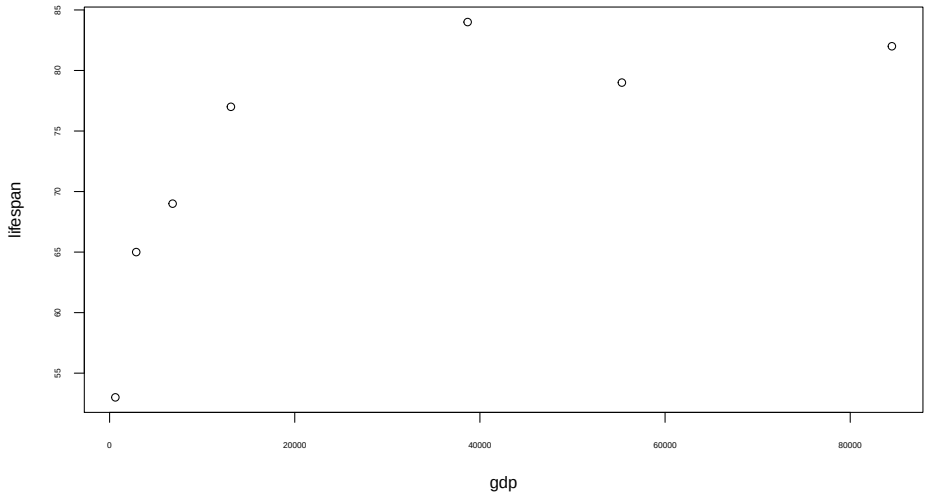
Set axis labels using the inputs `xlab` and `ylab`:

```
plot(lifespan ~ gdp, xlab="Gross Domestic Product (GDP)",ylab="Life Expectancy")
```



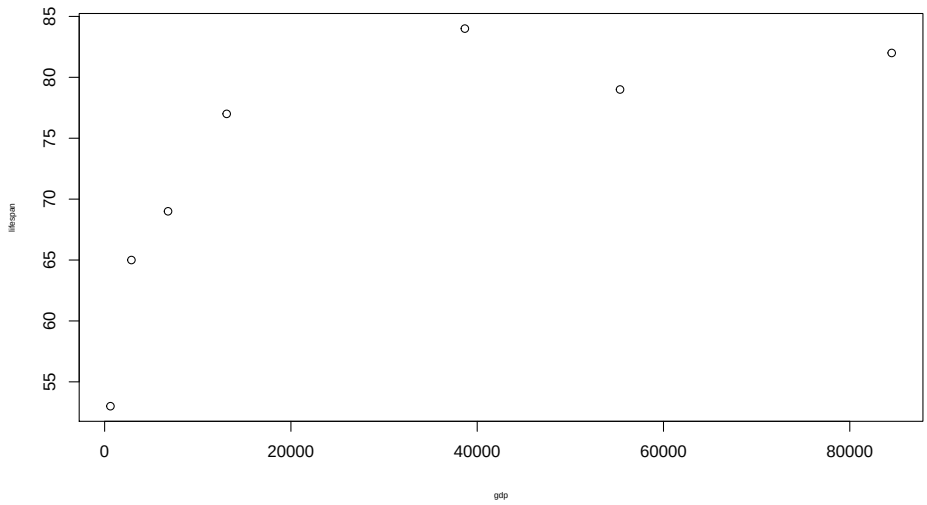
Set axis number size using the input `cex.axis` (the default is `cex.axis=1`):

```
plot(lifespan ~ gdp,cex.axis=.5)
```



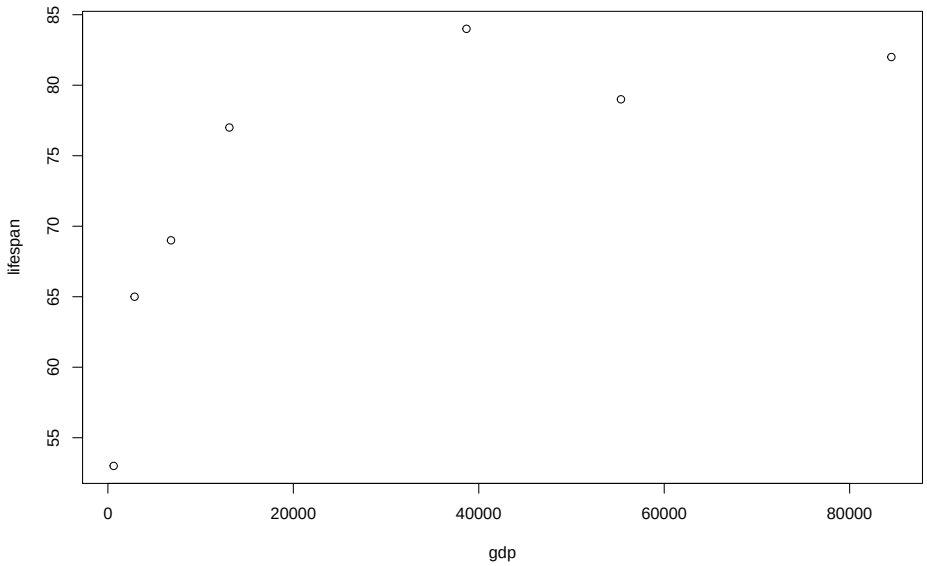
Set axis label size using the input `cex.label` (the default is `cex.lab=1`):

```
plot(lifespan ~ gdp,cex.lab=.5)
```



Set plot margins using the function `par(mar=c())` before you call `plot()`:

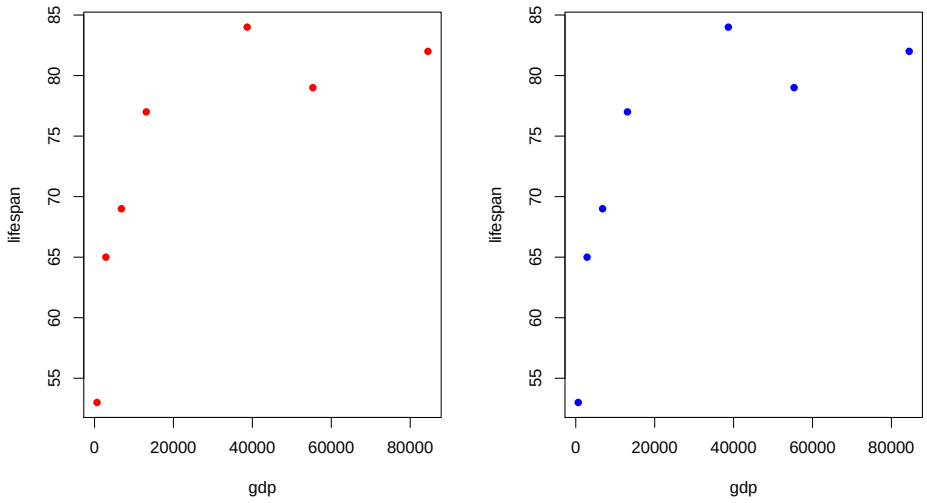
```
par(mar=c(5,5,0.5,0.5))
plot(lifespan ~ gdp)
```



In this command, the four numbers in the vector used to define `mar` correspond to the margin for the bottom, left, top, and right sides of the plot, respectively.

Create a multi-pane plot using the function `par(mfrow=c())` before you call `plot()`:

```
par(mfrow=c(1,2))
plot(lifespan ~ gdp,col="red",pch=16)
plot(lifespan ~ gdp,col="blue",pch=16)
```



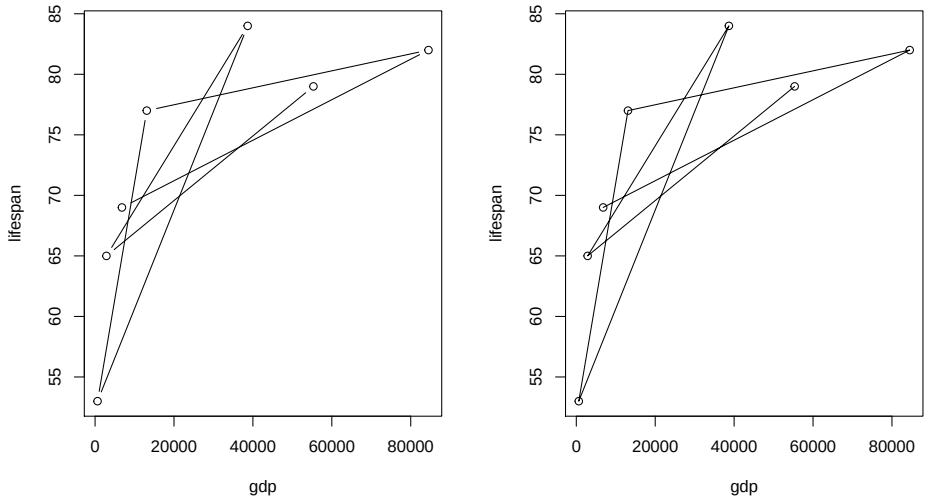
In this command, the two numbers in the vector used to define `mfrow` correspond to the number of rows and columns, respectively, on the entire plot. In this case, you have 1 row of plots with two columns.

Note that you will need to reset the number of panes when you are done with your multi-pane plot!

```
par(mfrow=c(1,1))
```

Plot dots and lines at once using the input `type`:

```
par(mfrow=c(1,2))
plot(lifespan ~ gdp,type="b")
plot(lifespan ~ gdp,type="o")
```

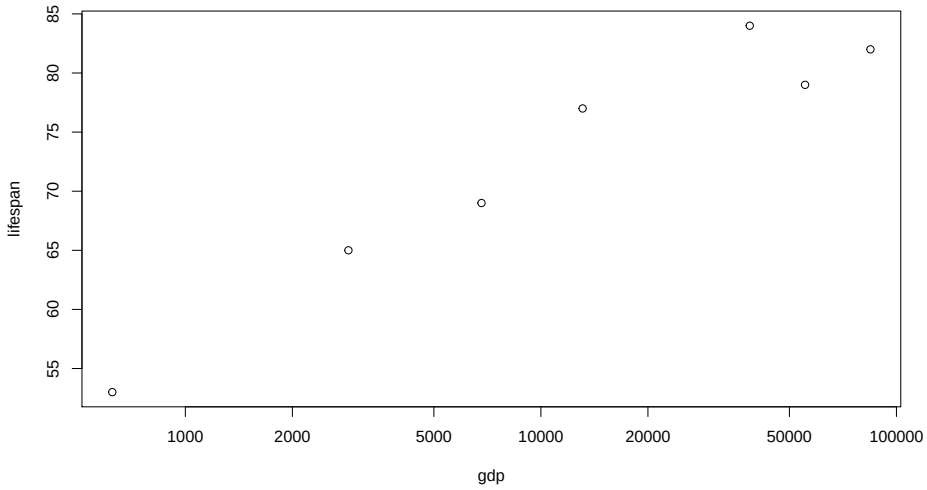



```
par(mfrow=c(1,1))
```

Note the two slightly different formats here.

Use a logarithmic scale for one or of your axes using the input `log`

```
plot(lifespan ~ gdp,log="x")
```



Exercise 2

Produce a *beautifully* formatted plot that incorporates **all** of these customization inputs explained above into a multi-paned plot.

Plotting with data frames

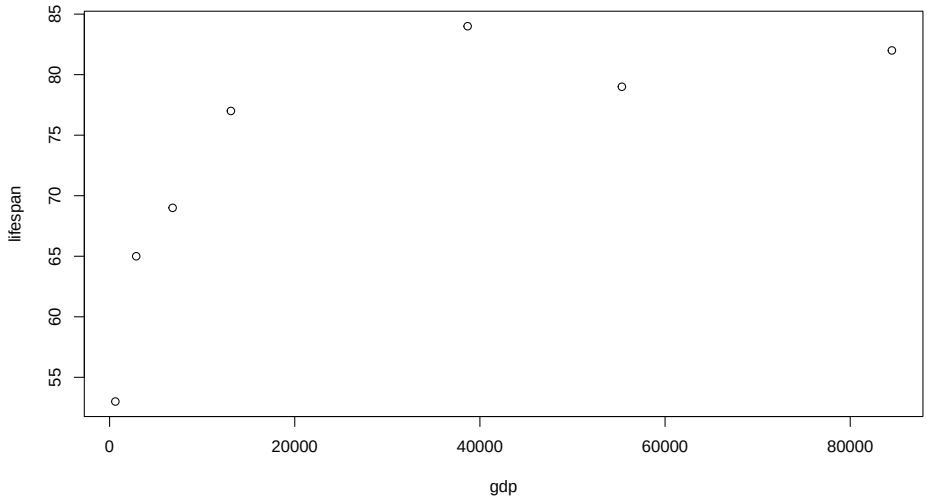
So far in this tutorial we have been using vectors to produce plots. This is nice for learning, but does not represent the real world very well. You will almost always be producing plots using dataframes.

Let's turn these vectors into a dataframe:

```
df <- data.frame(country,lifespan,gdp)
df
           country lifespan   gdp
1              USA       79 55335
2         Tanzania       65  2875
3            Japan       84 38674
4 Ctr. Africa Rep.       53   623
5            China       77 13102
6           Norway       82 84500
7            India       69  6807
```

To plot data within a dataframe, your `plot()` syntax changes slightly:

```
plot(lifespan ~ gdp, data=df)
```



This syntax is saying this: using the dataframe named `df` as a source, plot column `lifespan` as a function of column `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Another way to write this command is as follows:

```
plot(df$lifespan ~ df$gdp)
```

In this command, the `$` symbol is saying, "give me the column in `df` named `lifespan`". It is a handy way of referring to a column within a dataframe by name. You will learn more about working with dataframes in an upcoming module.

Exercise 2

A. Use the `df` dataframe to produce a bar plot that shows life expectancy for each country.

B. Use the `df` dataframe to produce a jumbled line plot of life expectancy as a function of GDP. Reference the `plot()` documentation to figure out how to change the thickness of the line.

Next-level plotting

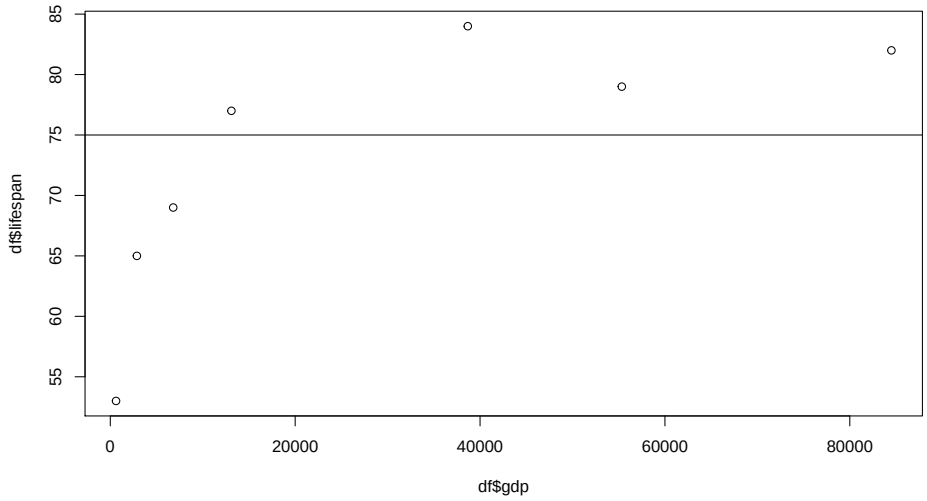
The possibilities for data visualization in `R` are pretty much limitless, and over time you will become fluent in making gorgeous plots. Here are a few common tools that can take your plots to the next level.

Adding lines

In some cases it is useful to add reference lines to your plot. For example, what if we wanted to be able to quickly see which countries had life expectancies below 75 years?

You can add a line at `lifespan = 75` using the function `abline()`.

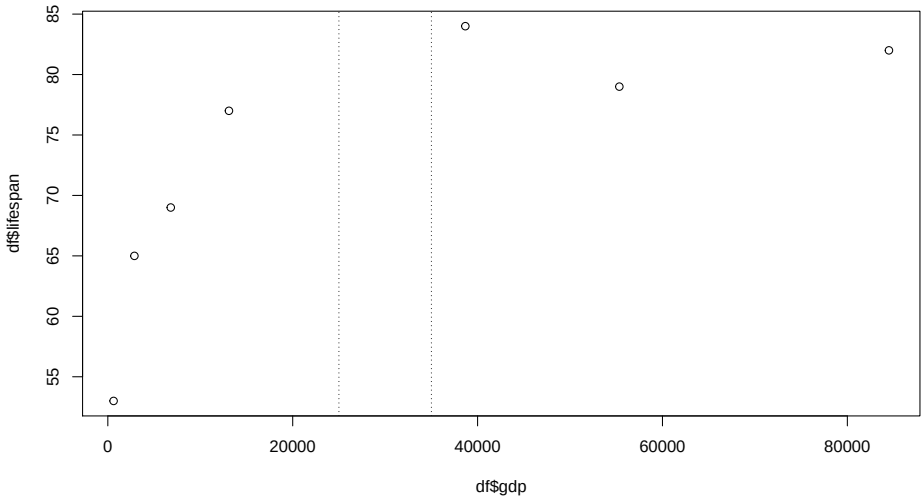
```
plot(df$lifespan ~ df$gdp)
abline(h=75)
```



In this command, the `h` input means "place a horizontal line at this y value.".

Similarly, you can use `v` to specify vertical lines at certain x values.

```
plot(df$lifespan ~ df$gdp)
abline(v=c(25000,35000),lty=3)
```



Note here that another input, `lty`, was used to change the type of line printed. (Refer to `?abline()` for more details).

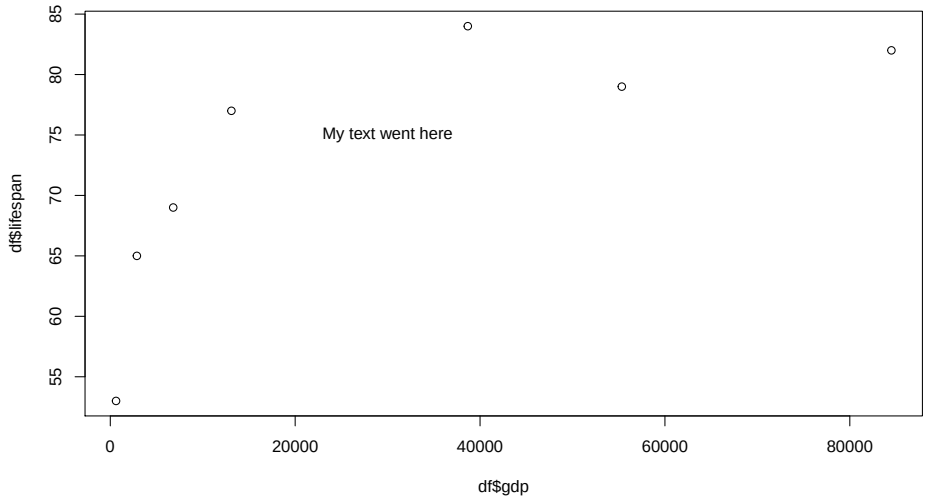
Exercise 4

Produce a plot of life expectancy as a function of GDP per capita. Then add a line to your plot that indicates which countries have per-capita GDPs that fall below (or above) the average per-capita GDP for the whole dataset. Make your line dashed and color it red.

Adding text

Use the `text()` function to add labels to your plot:

```
plot(df$lifespan ~ df$gdp)
text(x=30000,y=75,labels="My text went here")
```



Exercise 5

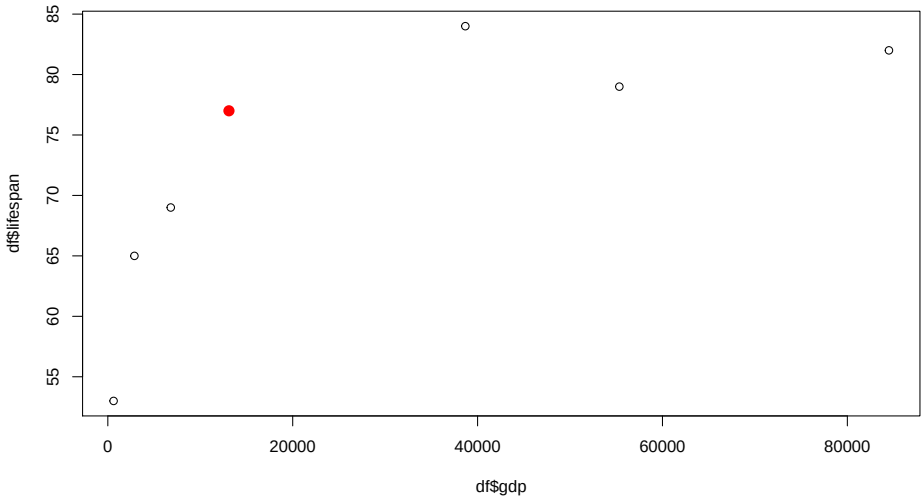
Produce a plot of life expectancy as a function of GDP per capita, then label each point by country. Make the labels small and place them *to the right* of their associated dot (Hint: use `?text` for help).

Highlighting certain data points

It can be helpful to highlight a certain data point (or group of data points) using a different dot size, format, or color.

To highlight a single data point, here is one approach you can take: first, plot all points, *then* re-plot the point of interest using the `points()` function:

```
plot(df$lifespan ~ df$gdp)
points(x=df$gdp[5],y=df$lifespan[5],col="red",pch=16,cex=1.5)
```



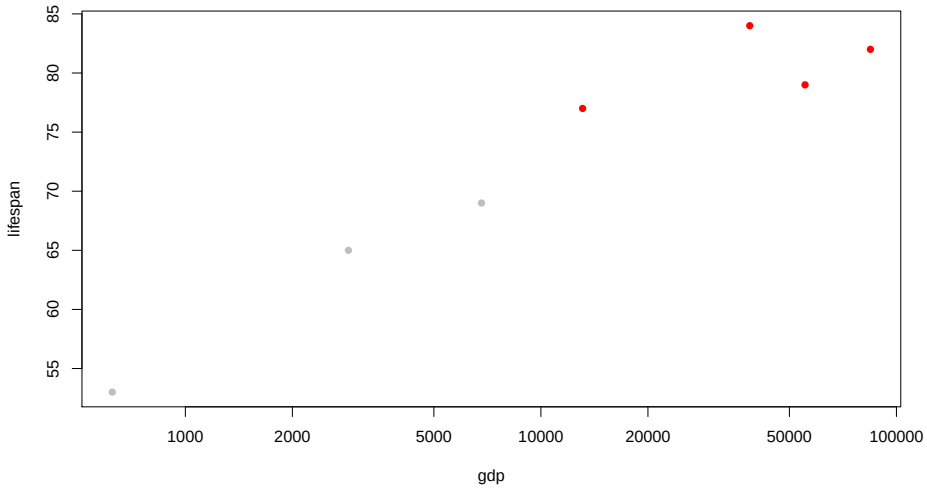
In this example, we re-plotted the data for the fifth row in the dataframe (in this case, China).

To highlight a group of data points, try this approach:

- First, create a vector that will contain the color for each data point.
- Second, determine the color for each data point using a logical test.
- Third, use your vector of colors within your `plot()` command.

For example, let's highlight all countries whose life expectancy is greater than 75.

```
# First
cols <- rep("grey",times=length(lifespan)) # create a vector of colors the length of vector `life
cols
[1] "grey" "grey" "grey" "grey" "grey" "grey" "grey"

# Second
change_these <- which(lifespan > 75)
change_these # these are the elements that we want to highlight
[1] 1 3 5 6
cols[change_these] <- "red"  # change the color for these elements to a highlight color

# Third
plot(lifespan ~ gdp,pch=16,col=cols,log="x")
```



Exercise 6

Produce a plot of life expectancy as a function of GDP per capita, in which all countries with GDPs below $10,000 have larger dots of a different color.

Building a plot from the ground up

In many applications it can be helpful to have complete control over the way your plot is built. To do so, you can build your plot from the very bottom up in multiple steps.

The steps for building up your own plot are as follows:

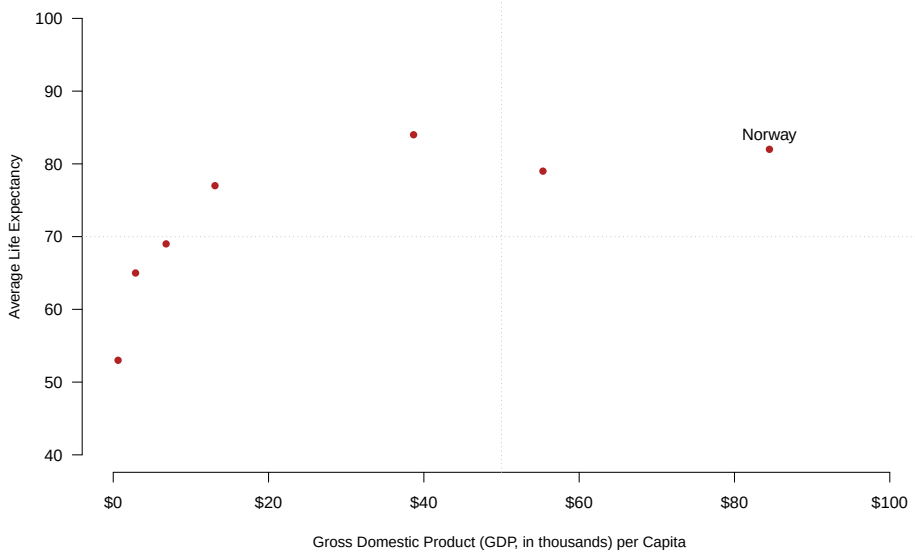
1. **Stage a blank canvas**: A plot begins with a blank canvas that covers a certain range of values for `x` and `y`. To stage a blank canvas, add this parameters to your `plot()` function: `type="n", axes=FALSE, ann=FALSE, xlim=c(__, __), ylim=c(__, __)"`. These commands tell `R` to plot a blank space, not to print axes, not to print annotations like x- or y-axis labels, and to limit your canvas to a certain coordinate range. Be sure to add numbers to the `xlim()` and `ylim()`commands.
2. **Add your axes**, if you want, using the function `axis()`. The command `axis(1)` prints the x-axis, and `axis(2)` prints the y-axis. This function allows you to define where tick marks occur and other details (see `?axis`).
3. **Add axis titles** using the function `title()`.
4. **Add reference lines**, if you want, using `abline()`. Do this before adding data, since it is usually nice for data points to be superimposed *on top of* your reference lines.

5. **Add your data** using either `points()` or `lines()`.

6. **Add text labels**, if you want, using `text()`.

Here is an example of this process:

```
# 1. Stage a blank canvas
par(mar=c(4.5,4.5,1,1))
plot(1,type="n",axes=FALSE,ann=FALSE,xlim=c(0,100000),ylim=c(40,100))

# 2. Add axes
axis(1,at=c(0,20000,40000,60000,80000,100000),labels=c("$0", "$20","$40","$60","$80","$100"))
axis(2,at=seq(40,100,by=10),las=2)

# 3. Add axis titles
title(xlab="Gross Domestic Product (GDP, in thousands) per Capita ",cex.lab=.9)
title(ylab="Average Life Expectancy",cex.lab=.9)

# 4. Add reference lines
abline(h=70,v=50000,lty=3,col="grey")

# 5. Add data
points(x=gdp,y=lifespan,pch=16,col="firebrick")

# 6. Add text
text(x=gdp[6],y=lifespan[6],labels="Norway",pos=3)
```



Review assignment

1. Create a vector of the names of 5 people who are sitting near you. Call it `people`.
2. Create a vector of those same 5 people's shoe sizes (in the same order!). Call it `shoe_size`.
3. Create another vector of those same 5 people's height. Call it `height`.
4. Create another vector of those same 5 people's sex. Call it `sex`.
5. Make a scatterplot of height and shoe size. Is there a correlation?
6. Look up help for `boxplot`. Make a `boxplot`.
7. Make a dataframe named `df`. It should contain columns of all the vectors created so far.
8. Make a side by side `boxplot` with shoe sizes of males vs females.
9. Try to find documentation for making a "histogram" (hint: use text autocomplete).
10. Make a histogram of people's height.

Other Resources

- R color palette
- The R Graph Gallery

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`

Introducing `R` packages

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.

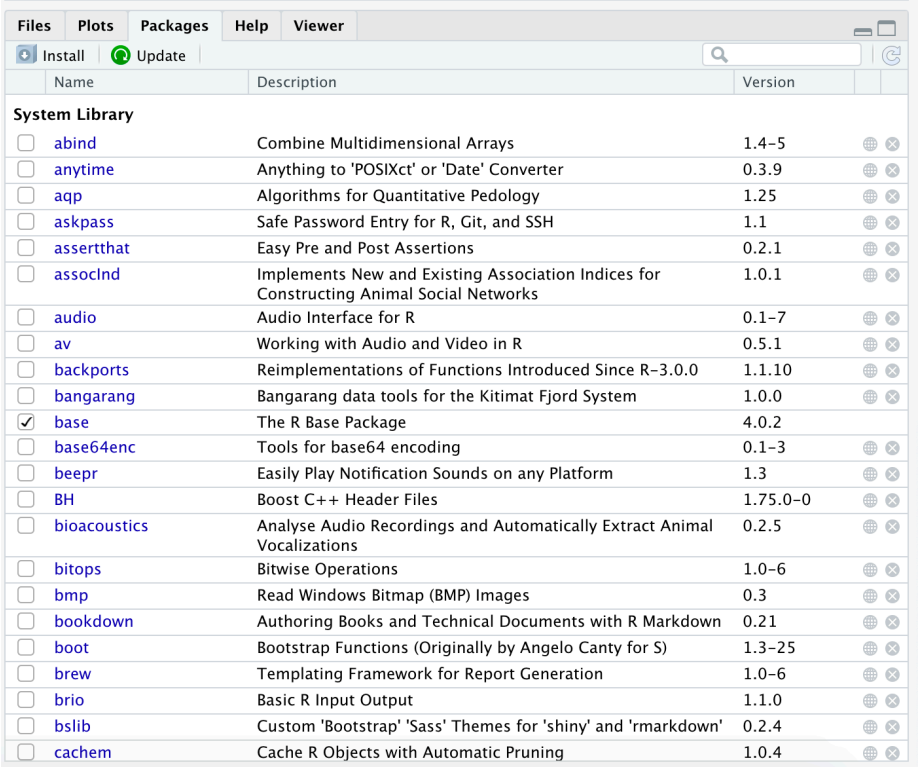
You can also access thousands of other functions and datasets through bundles of external code known as **packages**. Packages are developed and shared by `R` users around the world – a global community working together to increase `R`'s versatility and impact.

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.

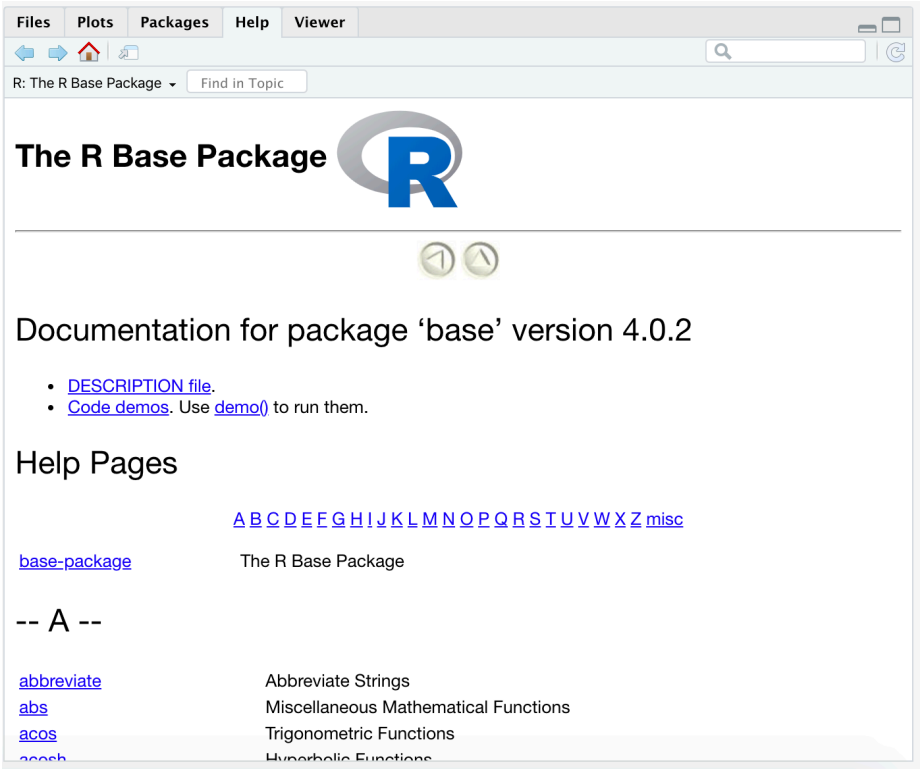
Finding the packages already on your computer

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



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.



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

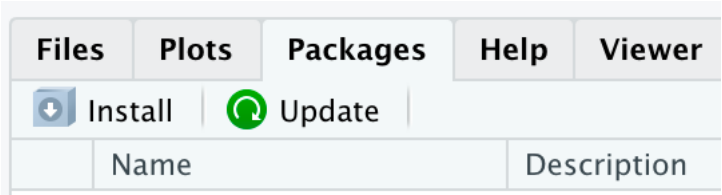
There are a couple ways to download and install a new `R` packae on your computer. Most packages are available from an open-source repository known as `CRAN` (which stands for Comprehensive `R` Archive Network). However, an increasingly common practice is to release packages on a public repository such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

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] "aVPj]+fFM9Wl_S')p{x`i?h6"
```

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] "7;j9a2PDu=}1 /<sKG\\B&vJp"
```

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

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

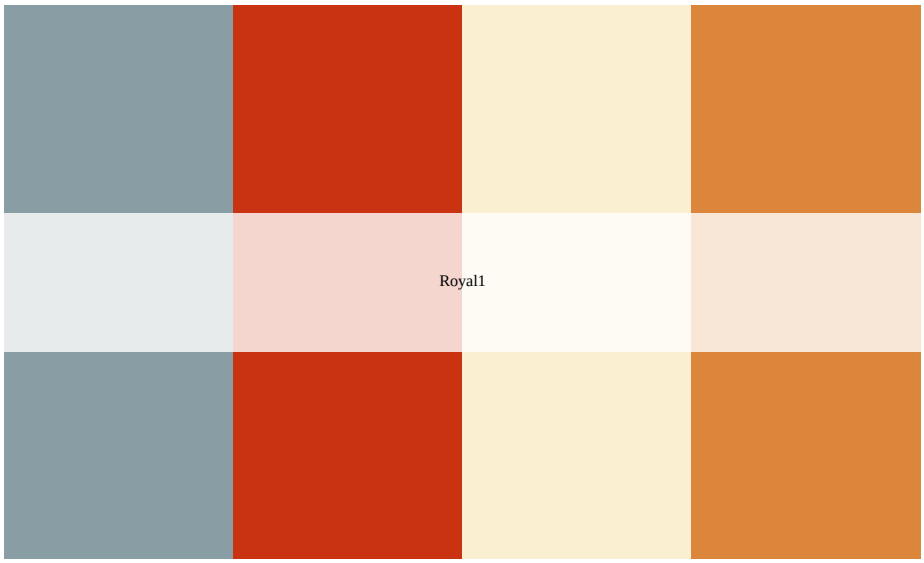
```
# Example from CRAN
install.packages("wesanderson")

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

2. Load the package

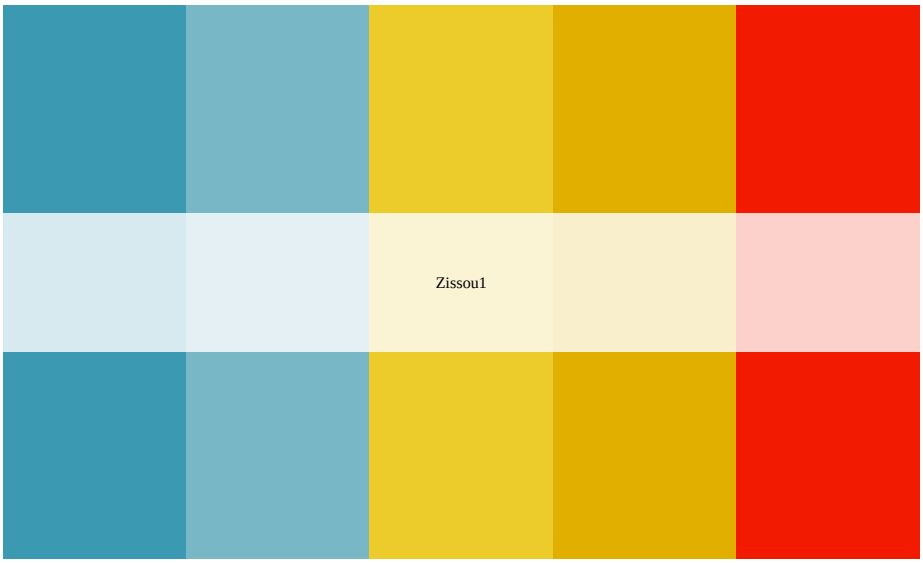
```
# Example
library(wesanderson)
```

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.)

A note on 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.

A note on 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.

Quick overview

You install them from CRAN with install.packages("x"). You use them in R with library("x"). You get help on them with help(package = "x").

Review assigment: let's install some packages

1. Install the `babynames` package.
2. Install `ggplot2`.
3. Install `dplyr`.
4. Install `RColorBrewer`.
5. Install `tidyr`.
6. Install `gapminder`.
7. Install `readr`.
8. Write 7 lines of code which load the above packages.

Chapter 13

Basics of `ggplot`

(Refer heavily to https://ggplot2-book.org/introduction.html)

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 syntax for creating different plots using using `ggplot2`

13.1 Thinking about data visualization

Let's start with a video: https://youtu.be/5Zg-C8AAIGg.

And one by Hans Rosling too: https://www.youtube.com/watch?v=FACK2knC08E

What is `ggplot`?

`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.

The name and concept

"GG" stands for "grammar of graphics", with "grammar" meaning "the fundamental principles or rules of an art or science" (**?**). The most well-known "grammar of graphics" was written in 2005 and laid out some abstract principles for describing statistical graphics (**?**). The basic idea is that all graphs can be described using a *layered* grammar, and that all graphs have the same general elements…

- data
- geometric objects
- aesthetics (mapping) of variables to objects

… whereas some graphs have additional elements…

- statistical transformations
- scales
- facets

Practice

Before getting to examples, hands-on together practice.

DEPRECATED BELOW THIS LINE

A practical example

Let's get practical (we'll get back to the theory later).

First, let's read in some data on health from the World Bank:

Canvas Canvas + variables (mapping) Canvas + variables (mapping) + geometric objects

Learning examples

Perfecting the canvas

Adjust y / x limits

Add a different background

Change x / y labels

Aesthetic attributes of the geoms

A scatterplot

Add a line of best fit

Add title / subtitle / caption

Review assignment

Note: Under construction!

Other resources

Note: Under construction!

(PART) Basic R workflow

Chapter 14

Importing data

Learning goals

- Understand what a `.csv` file is, and why they are important in data science
- How to format your data for easily importing data in `R`
- How to load, or "read", your data into `R`
- 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 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.

`.csv` files

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

station	year	doy	sit	x	y	n	bhvr	id
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:

hw-master.csv

```
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,2004,235,2004082201,NA,NA,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
```

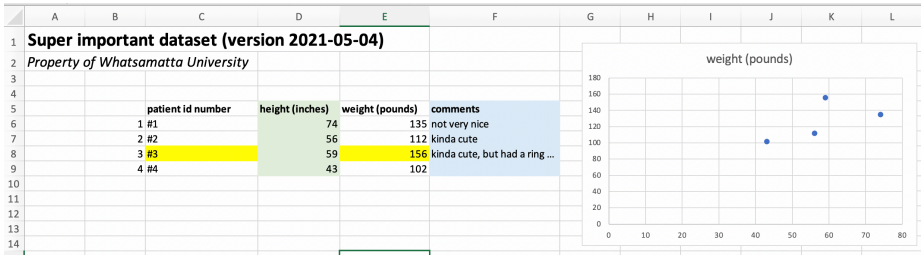
This looks scary, but it is actually really simple. A `.csv` is a simple text file in which each row is on a new line and columns of data are separated by commas. As a simple text file, there is no fancy formatting. There are no "Sheets" or "Tabs", as you would find in GoogleSheets or Excel; it is a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

Data format requirements

For those of us used to working in *Excel* 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	

Workflow for formatting your data

Below is the general worfklow for preparing your data for `R` is the following:

1. 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.

2. Remove blank columns before and in the middle of your data.

3. Remove elements such as graphs.

4. 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.

5. 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.

6. Remove all commas and hashtags from your dataset. You can do this with the 'Find & Replace' feature built-into in most spreadsheet editors.

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`, there are three key functions you should know:

`read.csv()`

This is the base function for reading in a `.csv`.

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

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

```
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!
```

`readr::read_csv()`

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

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

```
df <- readr::read_csv("super_data.csv")
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`:

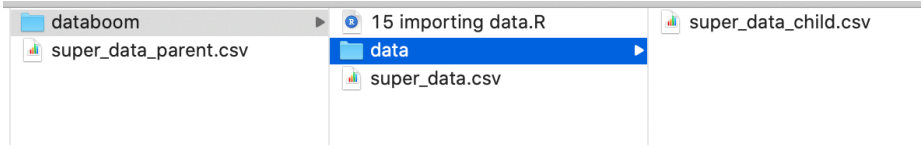
```
df <- readRDS("super_data.rds")
df
# A tibble: 4 x 4
  patient_id height_in weight_lb comment
       <dbl>     <dbl>     <dbl> <chr>
1          1        74       135 not very nice
2          2        56       112 kinda cute
3          3        59       156 kinda cute but had a ring
4          4 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

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:

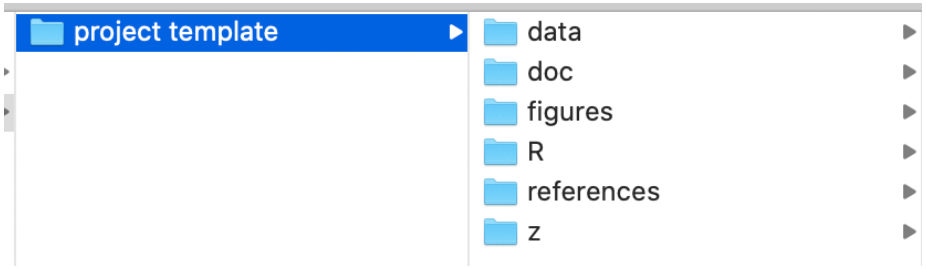
```
df <- read.csv("./data/super_data.csv")
```

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 a 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")
```

Review exercise

NOTE: Under construction!

Chapter 15

Dataframes

Dataframes & other data structures

Learning goals

- Practice exploring, summarizing, and filtering dataframes
- Understand the importance of *tidy* datasets
- Understand what the `tidyverse` is and why it is awesome

A **vector** is the most basic data structure in `R`, and the other structures are built out of vectors.

As a data scientist, the most common data structure you will be working with is a **dataframe**, which is essentially a spreadsheet: a dataset with rows and columns, in which each column represents is a vector of the same class of data.

We will explore dataframes in detail later, but here is a sneak peak at what they look like:

```
df <- data.frame(x=300:310,
           y=600:610)
df
     x   y
1  300 600
2  301 601
3  302 602
4  303 603
5  304 604
6  305 605
7  306 606
8  307 607
9  308 608
```

```
10 309 609
11 310 610
```

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`.

Typically you will "read" data into Rstudio from a folder on your computer (to be covered later). However, R (and many R packages) come with built-in, clean data

To see what's available

```
data()
```

The two other most common data structures are **matrices** and **lists**, but we will wait on learning about those. For now, focus on becoming comfortable using vectors and dataframes.

Exercise 4

Let's create a new object named `animals`. This is going to be a dataframe with 4 different columns: `species`, `height`, `color`, `veg` (whether or not the animal is a vegetarian).

15.0.1 More on dataframes

In data science, the most common data structure you will be using – by far – is the **dataframe**. You were introduced to this data structure in Module 10, but a more familiar and detailed orientation is worthwhile.

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)
# A tibble: 6 x 12
     id make  model    year class   trans  drive     cyl displ fuel    hwy   cty
  <dbl> <chr> <chr>   <dbl> <chr>   <chr>  <chr>   <dbl> <dbl> <chr> <dbl> <dbl>
1 13309 Acura 2.2CL/~  1997 Subcom~ Autom~ Front-~     4   2.2 Regu~    26    20
2 13310 Acura 2.2CL/~  1997 Subcom~ Manua~ Front-~     4   2.2 Regu~    28    22
3 13311 Acura 2.2CL/~  1997 Subcom~ Autom~ Front-~     6   3   Regu~    26    18
4 14038 Acura 2.3CL/~  1998 Subcom~ Autom~ Front-~     4   2.3 Regu~    27    19
5 14039 Acura 2.3CL/~  1998 Subcom~ Manua~ Front-~     4   2.3 Regu~    29    21
6 14040 Acura 2.3CL/~  1998 Subcom~ Autom~ Front-~     6   3   Regu~    26    17
# 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 (see module 10 for a refreshers). In addition to those functions you already know, we've added some new functions to your inventory of useful functions.

Exploration

```
tail(vehicles) # head() and tail() summarize the beginning and end of the object
# A tibble: 6 x 12
     id make  model    year class   trans   drive    cyl displ fuel    hwy   cty
  <dbl> <chr> <chr>   <dbl> <chr>   <chr>   <chr>  <dbl> <dbl> <chr> <dbl> <dbl>
1 28868 Yugo  GV Plu~  1990 Minico~ Manual~ Front~     4   1.3 Regu~    27    21
2  6635 Yugo  GV Plu~  1990 Subcom~ Manual~ Front~     4   1.3 Regu~    28    23
3  3157 Yugo  GV/GVX   1987 Subcom~ Manual~ Front~     4   1.1 Regu~    29    24
4  5497 Yugo  GV/GVX   1989 Subcom~ Manual~ Front~     4   1.1 Regu~    29    24
5  5498 Yugo  GV/GVX   1989 Subcom~ Manual~ Front~     4   1.3 Regu~    28    23
6  1745 Yugo  Gy/yug~  1986 Minico~ Manual~ Front~     4   1.1 Regu~    29    22
names(vehicles) # get names of columns
 [1] "id"    "make"  "model" "year"  "class" "trans" "drive" "cyl"   "displ"
[10] "fuel"  "hwy"   "cty"
```