R Tutorial

Tobias Gerstenberg

October 5, 2021

Load packages

Settings

```
# settings about how comments are displayed
opts_chunk$set(comment = "",
               fig.show = "hold")

# suppress grouping warning
options(dplyr.summarise.inform = F)

# set default plotting theme and text size
theme_set(theme_classic() +
            theme(text = element_text(size = 24)))

# set default color scheme in ggplot
options(ggplot2.discrete.color = RColorBrewer::brewer.pal(9,"Set1"))
options(ggplot2.discrete.fill = RColorBrewer::brewer.pal(9,"Set1"))
```

Getting ready

Installing R

Go on this link to download R: https://cloud.r-project.org/

Select the version that works for your operating system, and download the latest release (R-4.1.1).

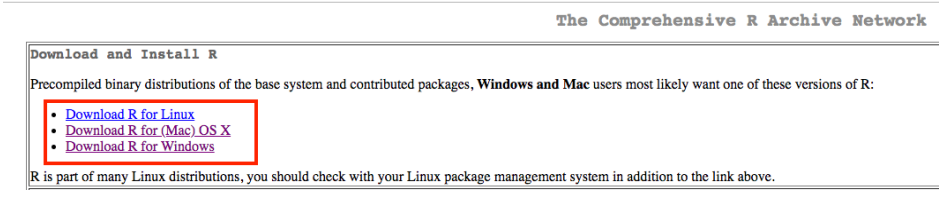


Figure 1: Download R.

Once you've downloaded R, install it following the instructions on the screen.

Installing R Studio

Go on this link to download R Studio: https://www.rstudio.com/products/rstudio/download/#download

And then download the version that works for your operating system.

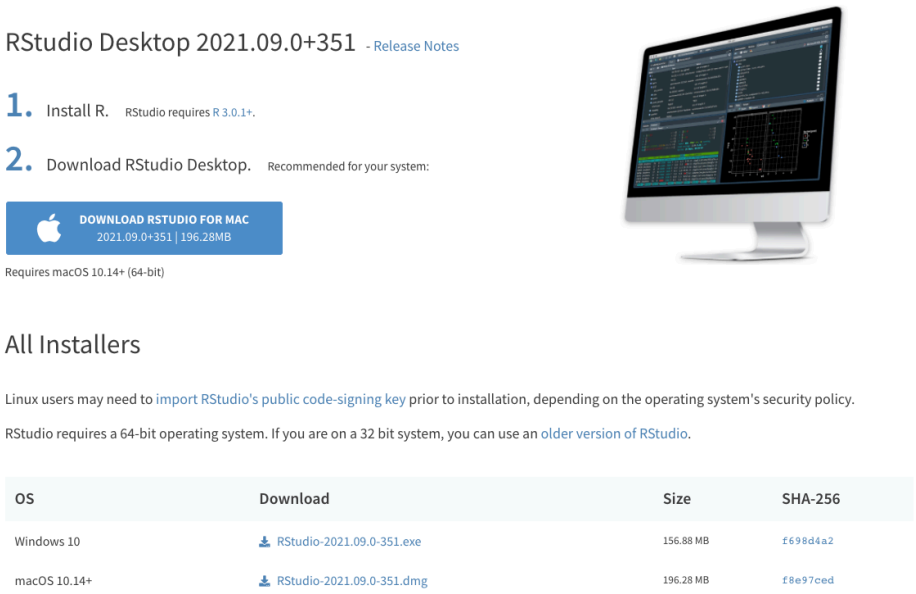


Figure 2: Download R Studio.

Once you've downloaded R Studio, install it following the instructions on the screen.

Why R?

- What R is very good at:
  - data visualization
  - data manipulation
  - statistics
  - project documentation with R Markdown
- What Python is very good at:
  - you can do everything with Python (whereas R is a more specialized language)
  - Python is the main language for machine learning (deep learning)
  - you can program experiments in Python
- Depending on what research you do, I strongly suggest to learn Python, too!
- In my research, I use both languages together.

Setting things up

R Studio

R Studio is a great integrated development environment (IDE) in which you can do all your R coding.

Before we get started, let's change some of the settings in R Studio first.

Make sure that:

- Restore .RData into workspace at startup is *unselected*
- Save workspace to .RData on exit is set to *Never*

This can otherwise cause problems with reproducibility and weird behavior between R sessions because certain things may still be saved in your workspace.

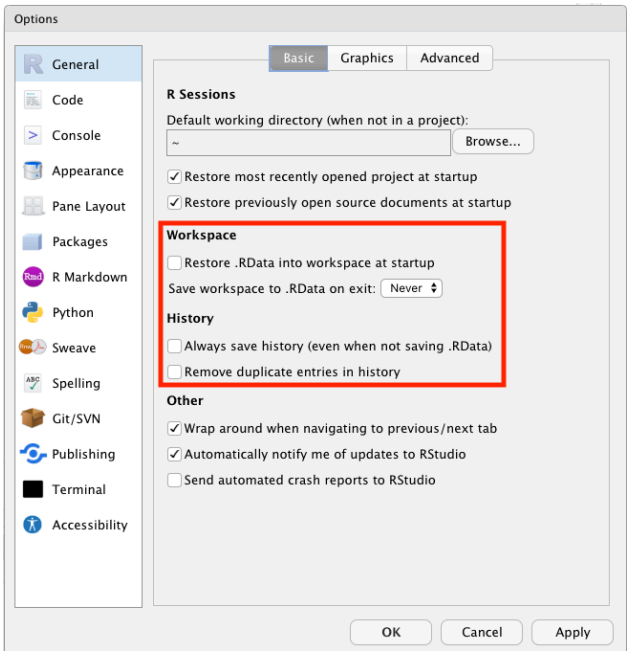


Figure 3: General preferences.

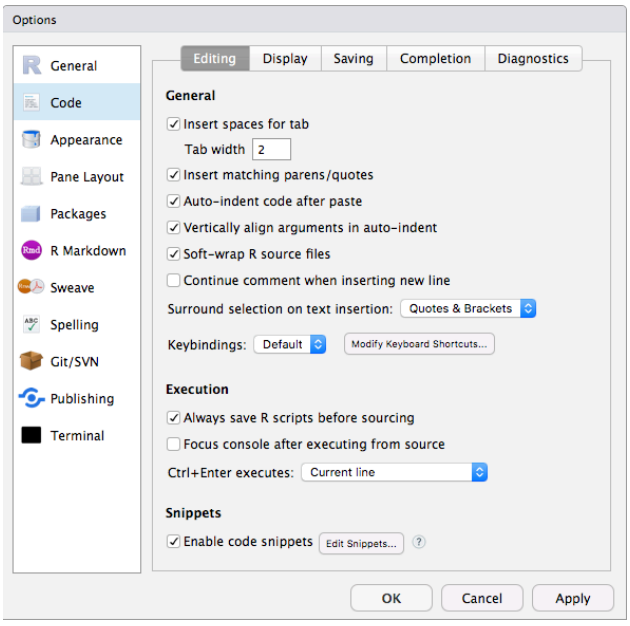


Figure 4: Code window preferences.

This makes sure that each time we run R Studio, we are starting with a fresh environment rather than still having variables saved from a previous run (which can cause trouble).

Make sure that:

- Soft-wrap R source files is *selected*

This way you don't have to scroll horizontally. At the same time, avoid writing long single lines of code. For example, instead of writing code like so:

```
ggplot(data = diamonds, aes(x = cut, y = price)) +
  stat_summary(fun = "mean", geom = "bar", color = "black", fill = "lightblue", width = 0.85) +
  stat_summary(fun.data = "mean_cl_boot", geom = "linerange", size = 1.5) +
  labs(title = "Price as a function of quality of cut", subtitle = "Note: The price is in US dollars",
```

You may want to write it this way instead:

```
ggplot(data = diamonds, aes(x = cut, y = price)) +
  # display the means
  stat_summary(fun = "mean",
               geom = "bar",
               color = "black",
               fill = "lightblue",
               width = 0.85) +
  # display the error bars
  stat_summary(fun.data = "mean_cl_boot",
               geom = "linerange",
               size = 1.5) +
  # change labels
  labs(title = "Price as a function of quality of cut",
       subtitle = "Note: The price is in US dollars", # we might want to change this later
       tag = "A",
       x = "Quality of the cut",
       y = "Price")
```

This makes it much easier to see what's going on, and you can easily add comments to individual lines of code.

Here is cheatsheet with more useful information about R Studio:

- R Studio cheatsheet

Getting help

There are a few different ways to get help in R. You can either put a `?` in front of the function you'd like to learn more about, or use the `help()` function.

```
?print
help("print")
```

> **Tip**: To see the help file, hover over a function (or dataset) with the mouse (or select the text) and then press `F1`.

I recommend using `F1` to get to help files – it's the fastest way!

R help files can sometimes look a little cryptic. Most R help files have the following sections (copied from here):

Title: A one-sentence overview of the function.

Description: An introduction to the high-level objectives of the function, typically about one paragraph long.

Usage: A description of the syntax of the function (in other words, how the function is called). This is where you find all the arguments that you can supply to the function, as well as any default values of these arguments.

Arguments: A description of each argument. Usually this includes a specification of the class (for example, character, numeric, list, and so on). This section is an important one to understand, because arguments are frequently a cause of errors in R.

Details: Extended details about how the function works, provides longer descriptions of the various ways to call the function (if applicable), and a longer discussion of the arguments.

Value: A description of the class of the value returned by the function.

See also: Links to other relevant functions. In most of the R editors, you can click these links to read the Help files for these functions.

Examples: Worked examples of real R code that you can paste into your console and run.

Here is the help file for the `print()` function:

print {base} R Documentation

Print Values

Description

`print` prints its argument and returns it *invisibly* (via `invisible(x)`). It is a generic function which means that new printing methods can be easily added for new `class`es.

Usage

```
print(x, ...)

## S3 method for class 'factor'
print(x, quote = FALSE, max.levels = NULL,
      width = getOption("width"), ...)

## S3 method for class 'table'
print(x, digits = getOption("digits"), quote = FALSE,
      na.print = "", zero.print = "0",
      right = is.numeric(x) || is.complex(x),
      justify = "none", ...)

## S3 method for class 'function'
print(x, useSource = TRUE, ...)
```

Arguments

`x`	an object used to select a method.
`...`	further arguments passed to or from other methods.
`quote`	logical, indicating whether or not strings should be printed with surrounding quotes.
`max.levels`	integer, indicating how many levels should be printed for a factor; if `0`, no extra "Levels" line will be printed. The default, `NULL`, entails choosing `max.levels` such that the levels print on one line of width `width`.
`width`	only used when `max.levels` is NULL, see above.
`digits`	minimal number of *significant* digits, see `print.default`.
`na.print`	character string (or `NULL`) indicating `NA` values in printed output, see `print.default`.
`zero.print`	character specifying how zeros (`0`) should be printed; for sparse tables, using `"."` can produce more readable results, similar to printing sparse matrices in **Matrix**.
`right`	logical, indicating whether or not strings should be right aligned.
`justify`	character indicating if strings should left- or right-justified or left alone, passed to `format`.
`useSource`	logical indicating if internally stored source should be used for printing when present, e.g., if `options(keep.source = TRUE)` has been in use.

Details

The default method, `print.default` has its own help page. Use `methods("print")` to get all the methods for the `print` generic.

`print.factor` allows some customization and is used for printing `ordered` factors as well.

`print.table` for printing `table`s allows other customization. As of R 3.0.0, it only prints a description in case of a table with 0-extents (this can happen if a classifier has no valid data).

See `noquote` as an example of a class whose main purpose is a specific `print` method.

References

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*. Wadsworth & Brooks/Cole.

See Also

The default method `print.default`, and help for the methods above; further `options`, `noquote`.

For more customizable (but cumbersome) printing, see `cat`, `format` or also `write`. For a simple prototypical print method, see `.print.via.format` in package **tools**.

Examples

```
require(stats)

ts(1:20)  #-- print is the "Default function" --> print.ts(.) is called
for(i in 1:3) print(1:i)

## Printing of factors
attenu$station ## 117 levels -> 'max.levels' depending on width

## ordered factors: levels  "11 < 12 < .."
esoph$agegp[1:12]
esoph$alcgp[1:12]

## Printing of sparse (contingency) tables
set.seed(521)
t1 <- round(abs(rt(200, df = 1.8)))
t2 <- round(abs(rt(200, df = 1.4)))
table(t1, t2) # simple
print(table(t1, t2), zero.print = ".") # nicer to read

## same for non-integer "table":
T <- table(t2,t1)
T <- T * (1+round(rlnorm(length(T)))/4)
print(T, zero.print = ".") # quite nicer,
print.table(T[,2:8] * 1e9, digits=3, zero.print = ".")
## still slightly inferior to  Matrix::Matrix(T)  for larger T

## Corner cases with empty extents:
table(1, NA) # < table of extent 1 x 0 >
```

Figure 5: Help file for the print() function.

It can take some time until these are really helpful. Until then, **google things**! R has a very active community with a large number of posts on stackoverflow and other online forums. Often it's enough to just copy and paste the error you're getting into google and then looking at the first solution that's been endorsed by others.

Installing and maintaining packages

What makes R powerful is the large number of packages that have been written for R. You can install a new package like so:

```
install.packages("tidyverse")
```

You can also install multiple packages at the same time, by concatenating the package names using the `c()` function ("c" stands for "concatenate"):

```
install.packages(c("tidyverse","broom"))
```

To make sure that your packages remain up to date, you can go to `Tools > Check for Package Updates ...` in R Studio.

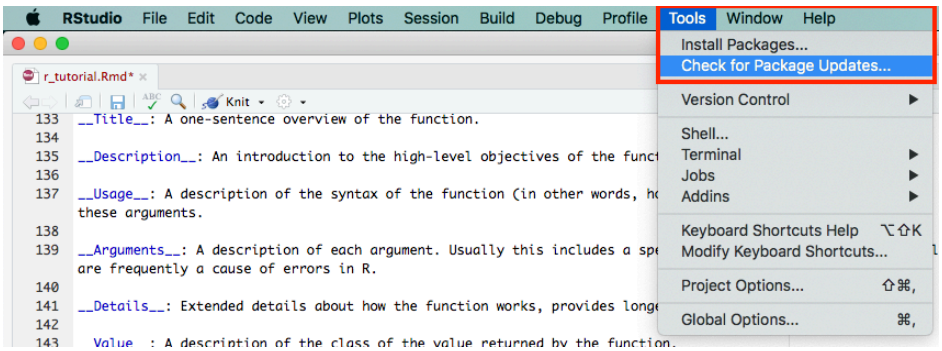


Figure 6: Help file for the print() function.

You can then click `Select All` and then `Install Updates`.

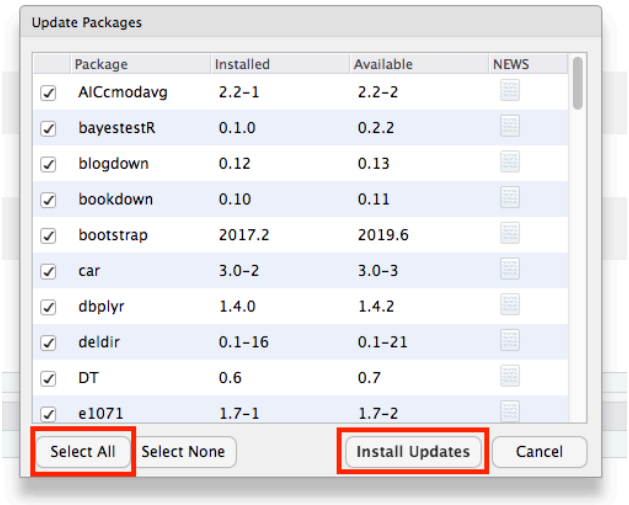


Figure 7: Help file for the print() function.

R Studio might ask you to restart your R session before updating the packages.

R Markdown

R Markdown files are a great way of organizing one's code. This tutorial is written using R Markdown! Most importantly, you can put R code straight into your R Markdown file so that you can have everything in one place. Indeed, you can write a full paper in R Markdown if you like (using the package papaja).

There are two main ways of putting code into your R Markdown document. Most often, you will create a code chunk and put the code into that chunk, like so:

```r
a = 1 + 2
print(a)
```

```
[1] 3
```

You can also evaluate R code in line with other text like so: The value of `a` is 3.

The nice thing about these code chunks is that they show you the output directly underneath the chunk when you run it. This is also true for plots. This means you can focus on one place rather than needing to shift back and forth between multiple windows.

And a big advantage of using R Markdown is that you can render the file in different formats by "knitting" it. For example, I've created the ".html" file using this R Markdown file. This is a great way of sharing your code with others and contributing to open science this way.

You can also use R Markdown to build academic homepages, and to write online books.

You can find some more information about R Markdown in the cheatsheets here:

- R Markdown cheatsheet

General structure

An RMarkdown file has four key components:

1. YAML header
2. Headings to structure the document
3. Text
4. Code chunks

The YAML (*Y*et *A*nother *M*arkdown *L*anguage) header specifies general options such as whether you'd like to have a table of content displayed, and in what output format you want to create your report (e.g. html or pdf). Notice that the YAML header cares about indentation, so make sure to get that right!

Headings are very useful for structuring your RMarkdown file. For your reports, it's often a good idea to have one header for each code chunk. The outline viewer here on the right is great for navigating large analysis files.

Text is self-explanatory.

Code chunks is where the coding happens. You can add one via the Insert button above, or via the shortcut `cmd + option + i` (the much cooler way of doing it!)

Code chunks can have code chunk options which we can set by clicking on the cog symbol on the right. You can also give code chunks a name, so that we can refer to it in text. I've named the one above "another-code-chunk". Make sure to have no white space or underscore in a code chunk name.

Some general advice

Before diving into R, here are a few more general tips.

Use R projects

By using R projects you make sure that the working directory is set correctly. You can then open multiple R projects at the same time without any conflicts between the projects (otherwise, you might overwrite variables from one script with the variables of another script using the same environment). For this tutorial, I've created the `r_tutorial.Rproj` file.

You can find out more about R projects here

Naming folders and files

I suggest to always use lower case characters and avoid whitespace in folder and file names. Either use "_" or "-" instead of a white space. Some programs (e.g. LaTeX) cannot deal with white spaces in file paths.

Always use relative paths

In your R Markdown file, make sure to always use relative paths rather than full paths. For example, you'll see below how I import the data like so `"../data/top2018songs.csv"` (relative path) rather than so `"/Users/tobi/Documents/work/projects_git/r_tutorial/data/data/top2018songs.csv"` (absolute path).

Using relative paths has the advantage that your collaborators can run code just like you can. If you were to use an absolute path, then your collaborator wouldn't be able to run the file without changing the path first.

Naming variables, functions, etc.

Personally, I like to name things consistently so that I have no trouble finding stuff even when I open up a project that I haven't worked on for a while. I try to use the following naming conventions:

Table 1: Some naming conventions I adopt to make my life easier.

name	use
df.thing	for data frames
l.thing	for lists
fun.thing	for functions
tmp.thing	for temporary variables

Always load all packages at the top

This way, other collaborators will directly see what packages they may need to install before running the code.

From top to bottom

Make sure that a script can be executed from top to bottom. For example, you don't want it to be the case that in order to run code chunk 2, you have to run code chunk 3 first.

Keep your projects organized

This github repository uses a project structure that I like. I recommend keeping data, figures, and code separate. Using the same structure in different projects really helps to keep things organized, and to find things quickly.

Learn keyboard shortcuts!

Learning keyboard shortcuts will speed up your workflow immensely! You can view the default keyboard shortcuts here: `Tools > Keyboard Shortcuts Help`

You can also modify and add keyboard shortcuts via `Tools > Modify Keyboard Shortcuts...`

For the very eager among, you can also take a look at snippets. Snippets allow you to define code macros for pieces of code that you use often (e.g. particular kinds of plots that you like making). You can find out more about how snippets in R Studio work here.

Here are some shortcuts I use a lot:

- `cmd + enter`: run selected code
- `cmd + shift + enter`: run code chunk
- `cmd + i`: re-indent selected code
- `cmd + shift + c`: comment/uncomment several lines of code
- `cmd + shift + d`: duplicate line underneath
- `cmd + shift + p`: open command palette (useful for learning shortcuts)
- `tab`: for auto-completion when writing code
- set up your own shortcuts to do useful things like
  - switch focus between tabs
  - move tabs to the left/right
  - jump up and down between code chunks
  - make a new cursor for each instance of the selected text
  - knitting documents

Don't write past the vertical rule in code blocks

This way, your code will look nice when you knit your R Markdown file into a html or a pdf output.

Keep your code tidy

Figure 8: Tidy code and data sparks joy!!!

This code block here is difficult to read:

```
ggplot(df.plot,aes(x = money,
                   y=happiness))+geom_point()+
geom_smooth(method="lm")
```

This code block is much easier to read:

```
ggplot(data = df.plot,
       mapping = aes(x = money,
                     y = happiness)) +
  geom_point() +
  geom_smooth(method = "lm")
```

- Use consistent indentation. RStudio makes it easy to write nice code. It figures out where to put the next line of code when you press `ENTER`. And if things ever get messy, just select the code of interest and hit `cmd + i` to re-indent the code.
- Use named arguments for functions. For example, write `ggplot(data = df.plot)` instead of `ggplot(df.plot)`. Using argument names makes it easier for others to read your code. Coming from another programming language, you might not get what `seq(1, 11, 2)` means, and it'll be easier to understand `seq(from = 1, to = 11, by = 2)` – Ah, this is a sequence from 1 to 11 in steps of 2!
- Use white spaces between names and arguments, and around `+`, `=`, `-`, etc.
- Always have a line break after `+` in ggplot2 or after using the pipe `%>%` (which we will discuss later). This makes it easier to just run parts of your code if you want to test stuff, and to comment out parts of your code, too.

Here are some more tips on how to write nice code in R:

- Advanced R style guide

R syntax

There are two main ways to code in R, one is called "base R" and the other is called "tidyverse". The "tidyverse" is a collection of powerful packages that work very well with one another It's the modern way of coding in R, and this tutorial uses the tidyverse. That said, it's still important to know how to write things using "base R".

This cheatsheet summarizes some of the key aspects of "base R"

- base R cheatsheet

The pipe %>%

A key part of coding in the tidyverse is using the pipe operator `%>%` (pronounce "then"). What's great about the pipe operator is that it allows us to write code in the order which makes sense: first I want to do this with the data, then I want to do that, then I want to print out the result. Let's consider the following example of making and eating a cake (thanks to https://twitter.com/dmi3k/status/1191824875842879489?s=09). This would be the traditional way of writing some code:

```
eat(slice(bake(put(pour(mix(ingredients), into = baking_form), into = oven), time = 30), pieces = 6, 1)
```

To see what's going on here, we need to read the code inside out. That is, we have to start in the innermost bracket, and then work our way outward.

However, there is a natural causal ordering to these steps and wouldn't it be nice if we could just write code in that order? Thanks to the pipe operator `%>%` we can! Here is the same example using the pipe:

```
ingredients %>%
  mix %>%
  pour(into = baking_form) %>%
  put(into = oven) %>%
  bake(time = 30) %>%
  slice(pieces = 6) %>%
  eat(1)
```

This code is much easier to read and write, since it represents the order in which we want to do things!

Abstractly, the pipe operator does the following:

> `f(x)` can be rewritten as `x %>% f()`

For example, in standard R, we would write:

```
x = 1:3

# standard R
sum(x)
```

```
[1] 6
```

With the pipe, we can rewrite this as:

```
x = 1:3

# with the pipe
x %>% sum()
```

```
[1] 6
```

This doesn't seem super useful yet, but just hold on a little longer.

> `f(x, y)` can be rewritten as `x %>% f(y)`

So, we could rewrite the following standard R code …

```
# rounding pi to 6 digits, standard R
round(pi, digits = 6)
```

```
[1] 3.141593
```

… by using the pipe:

```
# rounding pi to 6 digits, standard R
pi %>% round(digits = 6)
```

```
[1] 3.141593
```

Here is another example:

```
a = 3
b = 4
sum(a, b) # standard way
```

```
[1] 7
```

```
a %>% sum(b) # the pipe way
```

```
[1] 7
```

The pipe operator inserts the result of the previous computation as a first element into the next computation. So, `a %>% sum(b)` is equivalent to `sum(a, b)`. We can also specify to insert the result at a different position via the `.` operator. For example:

```
a = 1
b = 10
b %>% seq(from = a, to = .)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Here, I used the `.` operator to specify that I woud like to insert the result of `b` where I've put the `.` in the `seq()` function.

> `f(x, y)` can be rewritten as `y %>% f(x, .)`

Still not to thrilled about the pipe? We can keep going though (and I'm sure you'll be convinced eventually.)

> `h(g(f(x)))` can be rewritten as `x %>% f() %>% g() %>% h()`

For example, consider that we want to calculate the root mean squared error (RMSE) between prediction and data.

Here is how the RMSE is defined:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

where $\hat{y}_i$ denotes the prediction, and y_i the actually observed value.

In base R, we would do the following.

```
data = c(1, 3, 4, 2, 5)
prediction = c(1, 2, 2, 1, 4)

# calculate root mean squared error
rmse = sqrt(mean((prediction - data)^2))
print(rmse)
```

```
[1] 1.183216
```

Using the pipe operator makes the operation more intuitive:

```
data = c(1, 3, 4, 2, 5)
prediction = c(1, 2, 2, 1, 4)

# calculate root mean squared error the pipe way
rmse = (prediction-data)^2 %>%
  mean() %>%
  sqrt() %>%
  print()
```

```
[1] 1.183216
```

First, we calculate the squared error, then we take the mean, then the square root, and then print the result.

The pipe operator `%>%` is similar to the `+` used in `ggplot2` (for those of you who know this one already). It allows us to take step-by-step actions in a way that fits the causal order in which we want to do things.

> **Tip**: The keyboard shortcut for the pipe operator is: `cmd/ctrl + shift + m` **Definitely learn this one** – you'll use the pipe a lot!!

Tip: Code is generally easier to read when the pipe `%>%` is at the end of a line (just like the `+` in `ggplot2`).

A key advantage of using the pipe is that you don't have to save intermediate computations as new variables and this helps to keep your environment nice and clean!

Practice

Let's practice the pipe operator.

```
# here are some numbers
x = seq(from = 1, to = 5, by = 1)

# taking the log the standard way
log(x)
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
# now take the log the pipe way (write your code below)
```

```
# some more numbers
x = seq(from = 10, to = 5, by = -1)

# the standard way
mean(round(sqrt(x), digits = 2))
```

```
[1] 2.721667
```

```
# the pipe way (write your code below)
```

Doing stuff

Loading packages

The order in which packages in R are loaded matters!

```
library("tidyverse")
library("MASS")
```

versus

```
library("MASS")
library("tidyverse")
```

Both the `MASS` package and the `tidyverse` packages have a function called `select()`. In R, whichever package is loaded later, overwrites the functions of earlier loaded packages with the same name.

You can refer to functions from specific packages by adding the package name at the beginning. For example, this command would use the `select()` function from the `MASS` package `MASS::select()`, while this command would use the function from the `dplyr` package `dplyr::select()` (irrespective in which order you've loaded the packages). However, adding the package name to a function each time it's called is cumbersome. That's why we want to make sure to load the packages whose functions we use most frequently last.

In particular, I'd suggest to always load `library("tidyverse")` last because it loads a large number of often used functions.

Importing data

We can import a comma-separated-value (csv) file like so (you can ignore the `mutate()` part for now):

```
df.data = read_csv(file = "../data/top2018songs.csv") %>%
  mutate(rank = 1:nrow(.))
```

Table 2: Description of the different columns in the data frame.

column	description
id	Spotify URI of the song
name	Name of the song
artists	Artist(s) of the song
danceability	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
energy	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
key	The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C /D , 2 = D, and so on.
loudness	The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typical range between -60 and 0 db.
mode	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
speechiness	Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
acousticness	A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
instrumentalness	Predicts whether a track contains no vocals. 'Ooh' and 'aah' sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly 'vocal'. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.
liveness	Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
valence	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
tempo	The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.
duration_ms	The duration of the track in milliseconds.
time_signature	An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure).

The quickest way to take a look at your data is to hover your mouse over a variable of a data frame, and press `F2`.

Let's take a look at the top of the data frame:

```
df.data %>%
  print()
```

```
# A tibble: 100 x 17
   id      name     artists danceability energy   key loudness  mode speechiness
   <chr>   <chr>    <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 6DCZcS~ God's P~ Drake          0.754  0.449     7    -9.21     1      0.109
 2 3ee8Jm~ SAD!     XXXTEN~        0.74   0.613     8    -4.88     1      0.145
 3 0e7ipj~ rocksta~ Post M~        0.587  0.535     5    -6.09     0      0.0898
 4 3swc6W~ Psycho ~ Post M~        0.739  0.559     8    -8.01     1      0.117
 5 2G7V7z~ In My F~ Drake          0.835  0.626     1    -5.83     1      0.125
 6 7dt6x5~ Better ~ Post M~        0.68   0.563    10    -5.84     1      0.0454
 7 58q2HK~ I Like ~ Cardi B        0.816  0.726     5    -4.00     0      0.129
 8 7ef4Dl~ One Kis~ Calvin~        0.791  0.862     9    -3.24     0      0.11
 9 76cy1W~ IDGAF    Dua Li~        0.836  0.544     7    -5.98     1      0.0943
10 08bNPG~ FRIENDS  Marshm~        0.626  0.88      9    -2.38     0      0.0504
# ... with 90 more rows, and 8 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>
```

Here is a cheatsheet with more information about how to import data into R:

- importing data cheatsheet

Data visualiztion

How not to visualize data

We should always take a look at the data first.

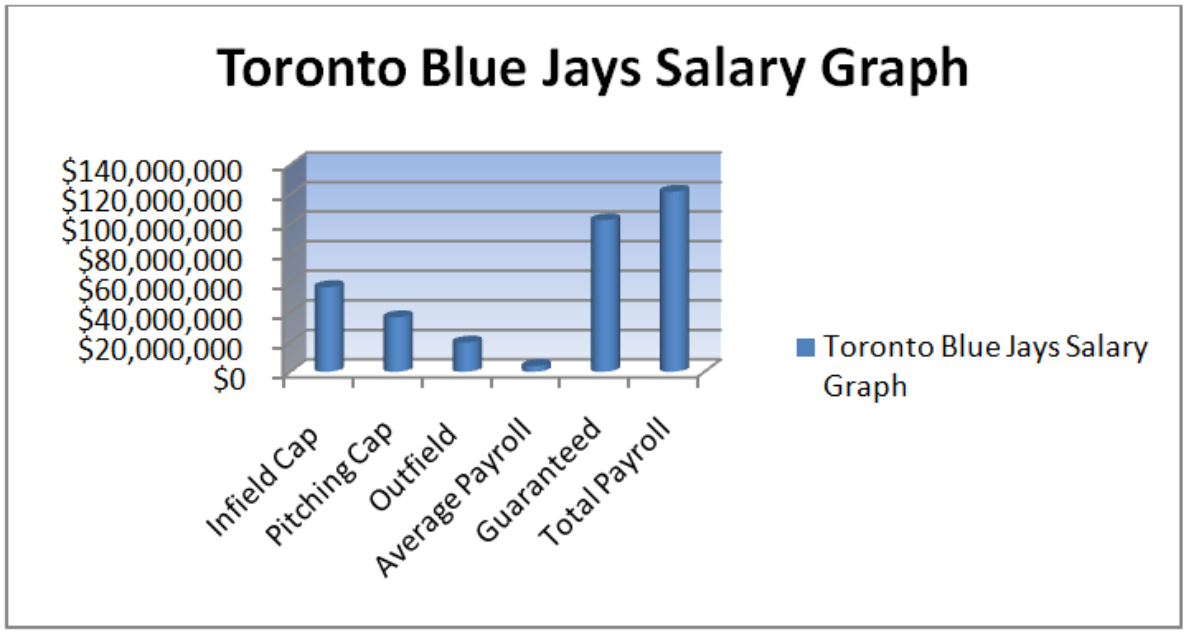


Figure 9: A not so good plot.

This second plots reminded me of the following:

Just because two lines look similar, doesn't mean that anything interesting is going on – it certainly doesn't

Figure 10: Another could-be-improved plot.

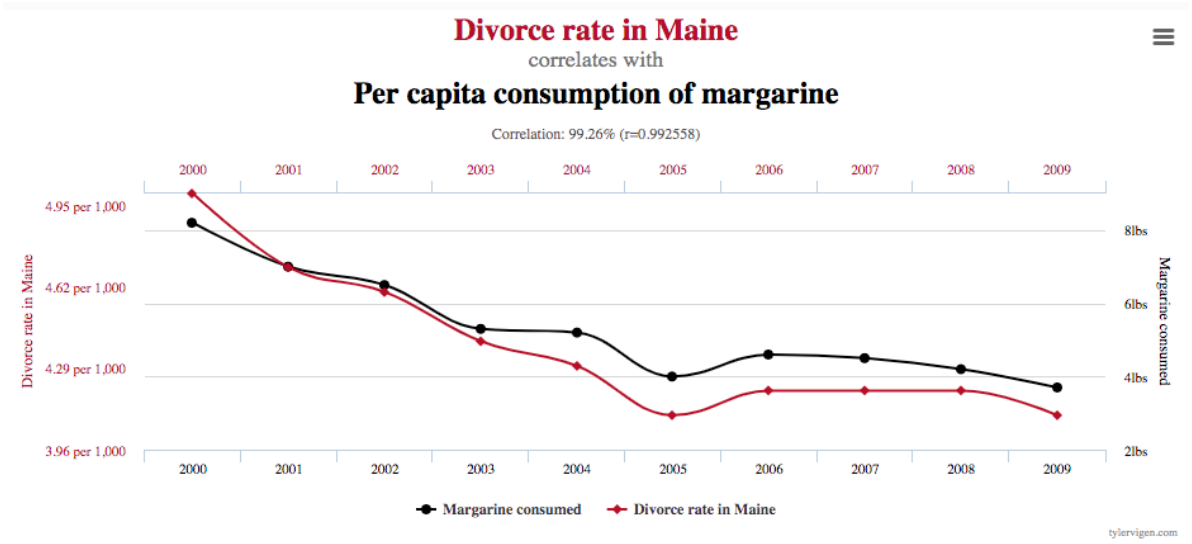


Figure 11: Correlation is not causation.

mean that the two phenomena represented by the lines are causally connected. For more inspiration check out this site https://www.tylervigen.com/spurious-correlations.

Why you should always visualize your data first

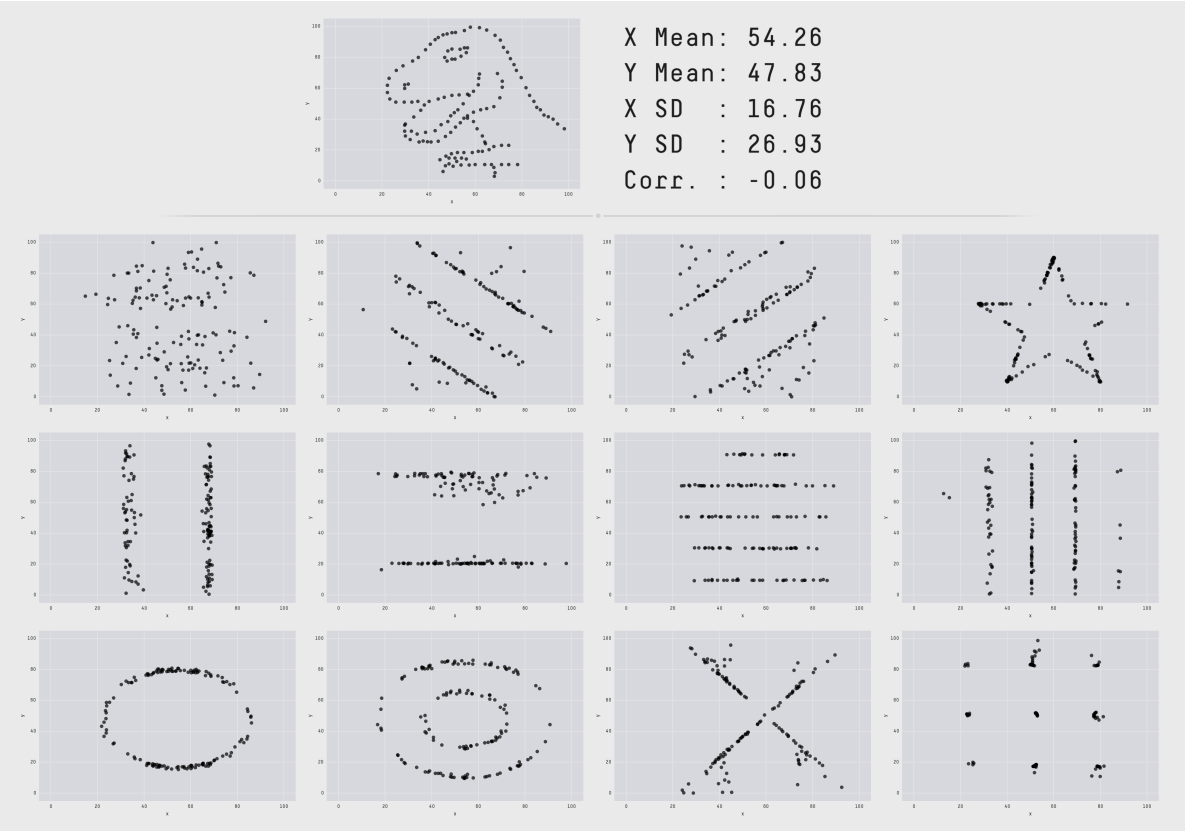


Figure 12: **The Datasaurus Dozen**. While different in appearance, each dataset has the same summary statistics to two decimal places (mean, standard deviation, and Pearson's correlation).

The data sets in Figure @ref(fig:datasaurus) all share the same summary statistics. Clearly, the data sets are not the same though.

Tip: Always plot the data first!

Here is the paper from which I took Figure @ref(fig:datasaurus). It explains how the figures were generated and shows more examples for how summary statistics and some kinds of plots are insufficient to get a good sense for what's going on in the data.

Visualizing data using `ggplot2`

`ggplot2` defines a grammar of graphics. One of the great things is that you can make a variety of different kinds of plots without ever having to change your data frame.

Here is how you would make a scatter plot:

```
ggplot(data = df.data,
       mapping = aes(x = danceability,
                     y = valence)) +
  geom_point()
```

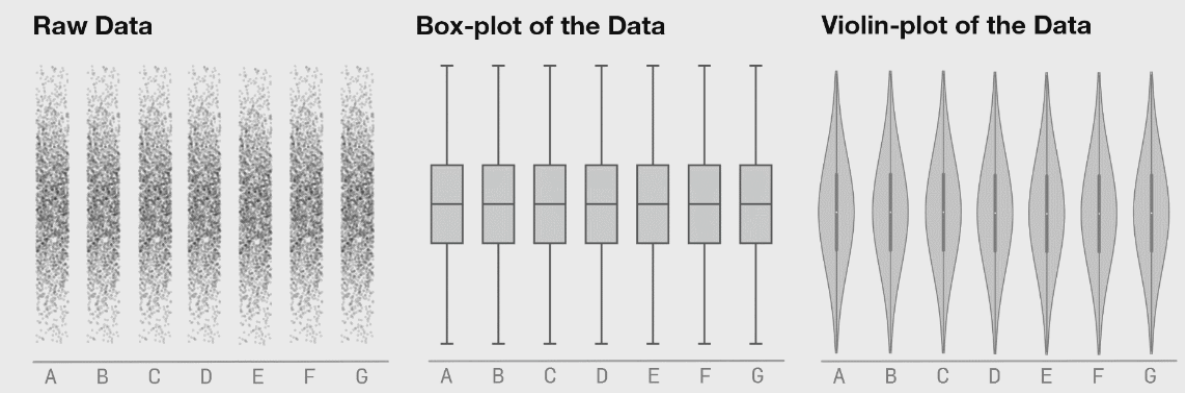
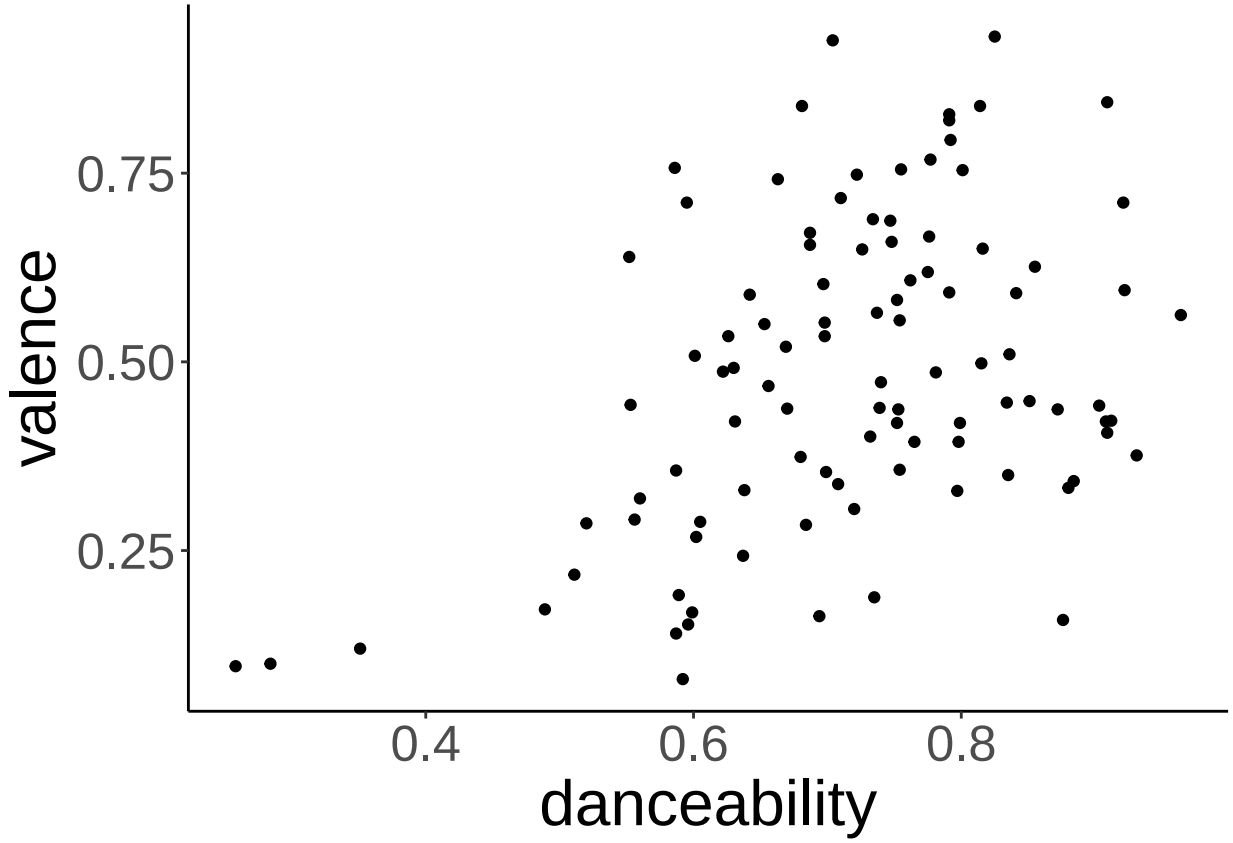
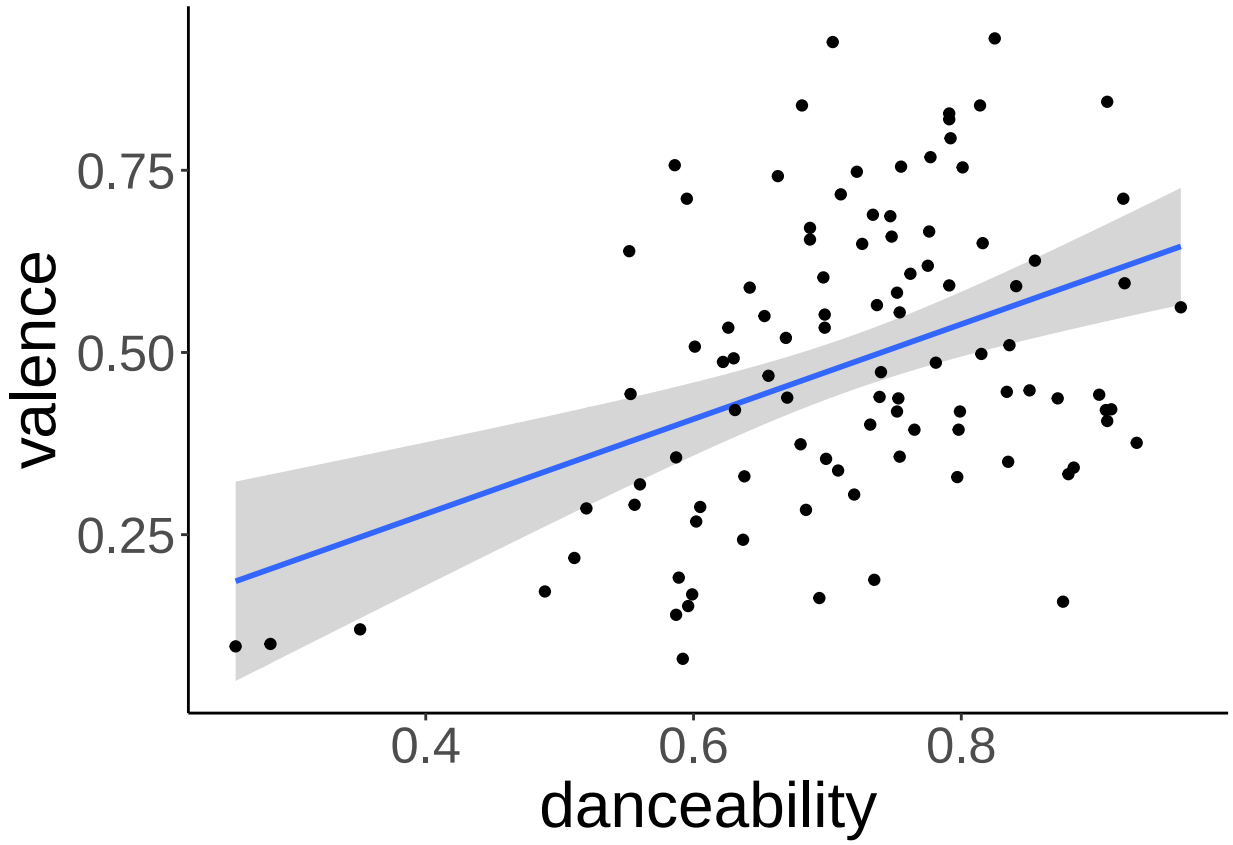


Figure 13: Boxplots can be misleading.



Adding a best-fitting linear regression line to the scatter plot is simple:

```
ggplot(data = df.data,
       mapping = aes(x = danceability,
                     y = valence)) +
  geom_smooth(method = "lm") +
  geom_point()
```



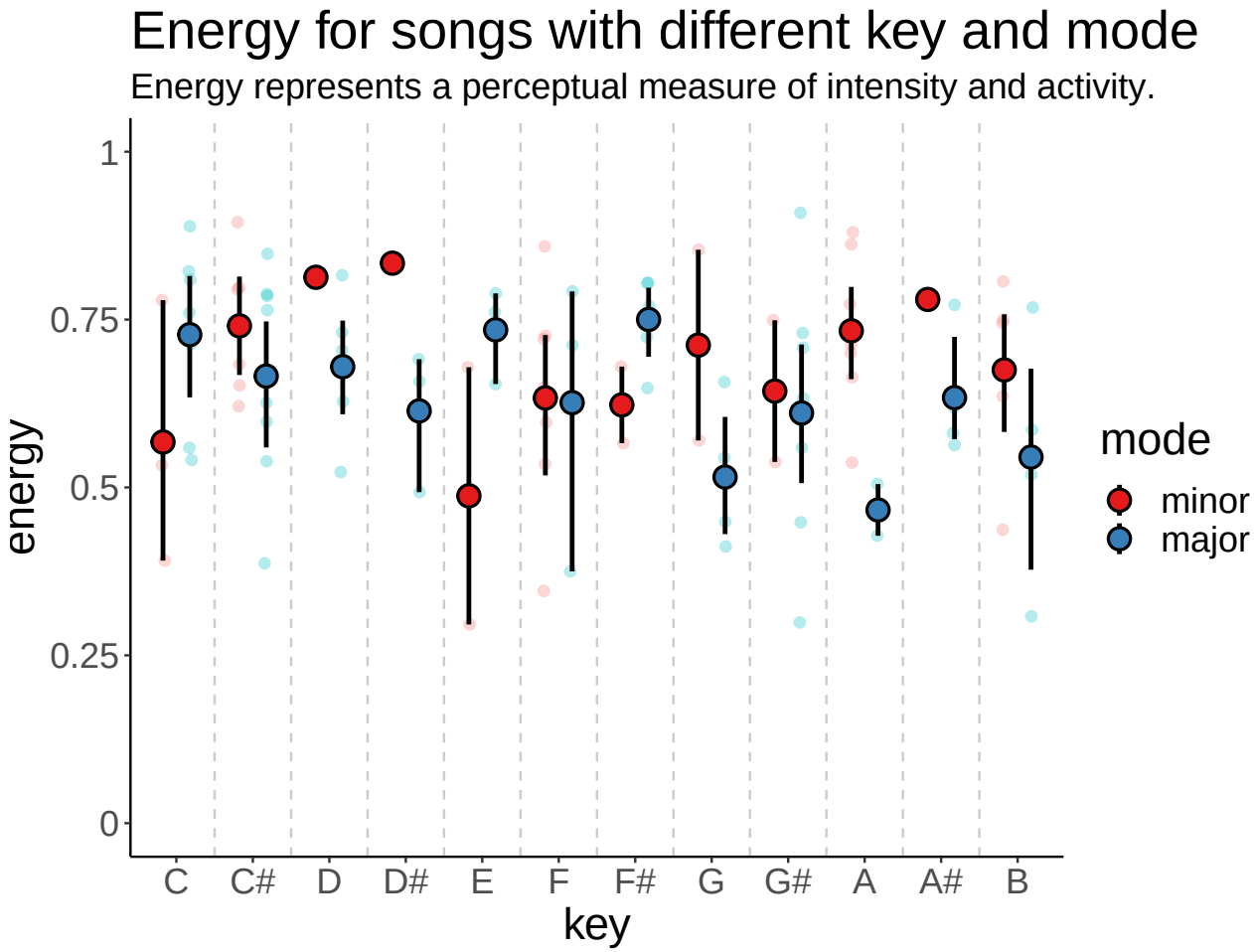
Here is a more involved plot that shows some of the things you can do with `ggplot2`:

```
df.plot = df.data %>%
  mutate(mode = factor(mode,
                       levels = c(0, 1),
                       labels = c("minor", "major")),
         key = factor(key,
                      levels = 0:11,
                      labels = c("C", "C#", "D", "D#",
                                 "E", "F", "F#", "G",
                                 "G#", "A", "A#", "B")))

ggplot(data = df.plot,
       mapping = aes(x = key,
                     y = energy,
                     group = mode,
                     fill = mode)) +
  # add individual data points
  geom_point(mapping = aes(color = mode),
             position = position_jitterdodge(dodge.width = 0.7,
                                             jitter.width = 0.1,
                                             jitter.height = 0),
             size = 2,
             alpha = 0.3) +
  # add means with error bars
  stat_summary(fun.data = "mean_cl_boot",
               geom = "pointrange",
```

```
              position = position_dodge(width = 0.7),
              size = 1,
              shape = 21) +
  # add the vertical lines
  geom_vline(xintercept = seq(from = 1.5, to = 11.5, by = 1),
             linetype = 2,
             color = "gray80") +
  # set title and subtitle of plot
  labs(title = "Energy for songs with different key and mode",
       subtitle = "Energy represents a perceptual measure of intensity and activity.") +
  # change the y-axis
  scale_y_continuous(breaks = seq(0, 1, 0.25),
                     labels = seq(0, 1, 0.25),
                     limits = c(0, 1)) +
  # set the fill color
  scale_fill_brewer(palette = "Set1") +
  # change the plotting theme
  theme_classic() +
  # adjust the text size
  theme(text = element_text(size = 20),
        plot.subtitle = element_text(size = 16))

# let's save the plot
ggsave(filename = "../figures/plots/energy_key_mode.pdf",
       width = 8,
       height = 6)
```



Here are some cheatsheets with data visualization info:

- ggplot2 cheatsheet
- data visualization principles cheatsheet

Practice time Make a scatter plot that shows `energy` on the x-axis and `tempo` on the y-axis.

```
# write your code here
```

Play around with the scatter plot that you've just created by incorporating some of the elements I've used in the more complex plot above. For example, you could try the following:

- change the size of the points
- change the color of the points
- change the text of the x-axis and y-axis title
- add a regression line
- add a horizontal line that intersects the y-axis at 100
- add color = mode to the `aes()` function and figure out what this does

```
# write your code here
```

Data manipulation

Visualizing data is fun! But often, we need to spend quite a bit of time beating data into the right shape first. We want our data to be tidy – a tidy data frame has one row per observation. Once we have a tidy data frame, plotting things using `ggplot2` becomes a breeze. Unfortunately, many data files aren't tidy at

all to start off with. For example, if you use Qualtrics to run your experiment, the data output will be far from tidy. So we have to learn how to beat our data into shape.

We often want to do things our data frame such as filter out certain observations, select a subset of the columns, rename variables, sort the rows, create new variables, and summarize the data in different ways. Here, we'll take a quick look at these data transformations in R.

filter

Let's filter out only the songs by the artist "Drake".

```
df.data %>%
  filter(artists == "Drake")
```

```
# A tibble: 4 x 17
  id        name    artists danceability energy   key loudness  mode speechiness
  <chr>     <chr>   <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
1 6DCZcSsp~ God's ~ Drake          0.754  0.449     7    -9.21     1      0.109
2 2G7V7zsV~ In My ~ Drake          0.835  0.626     1    -5.83     1      0.125
3 3CA9pLiw~ Nice F~ Drake          0.586  0.909     8    -6.47     1      0.0705
4 0TlLq3lA~ Nonstop Drake          0.912  0.412     7    -8.07     1      0.124
# ... with 8 more variables: acousticness <dbl>, instrumentalness <dbl>,
#   liveness <dbl>, valence <dbl>, tempo <dbl>, duration_ms <dbl>,
#   time_signature <dbl>, rank <int>
```

We can add multiple filters like so:

```
df.data %>%
  filter(artists == "Drake" & danceability > 0.8)
```

```
# A tibble: 2 x 17
  id        name    artists danceability energy   key loudness  mode speechiness
  <chr>     <chr>   <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
1 2G7V7zsV~ In My ~ Drake          0.835  0.626     1    -5.83     1       0.125
2 0TlLq3lA~ Nonstop Drake          0.912  0.412     7    -8.07     1       0.124
# ... with 8 more variables: acousticness <dbl>, instrumentalness <dbl>,
#   liveness <dbl>, valence <dbl>, tempo <dbl>, duration_ms <dbl>,
#   time_signature <dbl>, rank <int>
```

select()

Let's say we are only interested in a subset of the columns. We can use `select()` to do so:

```
df.data %>%
  select(name, artists, rank)
```

```
# A tibble: 100 x 3
   name                          artists        rank
   <chr>                         <chr>         <int>
 1 God's Plan                    Drake             1
 2 SAD!                          XXXTENTACION      2
 3 rockstar (feat. 21 Savage)    Post Malone       3
 4 Psycho (feat. Ty Dolla $ign) Post Malone       4
 5 In My Feelings                Drake             5
 6 Better Now                    Post Malone       6
 7 I Like It                     Cardi B           7
 8 One Kiss (with Dua Lipa)      Calvin Harris     8
 9 IDGAF                         Dua Lipa          9
```

```
10 FRIENDS                      Marshmello       10
# ... with 90 more rows
```

We can also deselect variables like so:

```
df.data %>%
  select(-id)
```

```
# A tibble: 100 x 16
   name           artists   danceability energy   key loudness  mode speechiness
   <chr>          <chr>            <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 God's Plan     Drake            0.754  0.449     7    -9.21     1      0.109
 2 SAD!           XXXTENTA~        0.74   0.613     8    -4.88     1      0.145
 3 rockstar (fea~ Post Mal~        0.587  0.535     5    -6.09     0      0.0898
 4 Psycho (feat.~ Post Mal~        0.739  0.559     8    -8.01     1      0.117
 5 In My Feelings Drake            0.835  0.626     1    -5.83     1      0.125
 6 Better Now     Post Mal~        0.68   0.563    10    -5.84     1      0.0454
 7 I Like It      Cardi B          0.816  0.726     5    -4.00     0      0.129
 8 One Kiss (wit~ Calvin H~        0.791  0.862     9    -3.24     0      0.11
 9 IDGAF          Dua Lipa         0.836  0.544     7    -5.98     1      0.0943
10 FRIENDS        Marshmel~        0.626  0.88      9    -2.38     0      0.0504
# ... with 90 more rows, and 8 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>
```

Now we have a data frame that has all the columns except for the `id` column.

`rename()`

Renaming variables is simple!

```
df.data %>%
  rename(song = name,
         singer = artists)
```

```
# A tibble: 100 x 17
   id       song     singer danceability energy   key loudness  mode speechiness
   <chr>    <chr>    <chr>         <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 6DCZcSs~ God's P~ Drake         0.754  0.449     7    -9.21     1      0.109
 2 3ee8Jmj~ SAD!     XXXTE~        0.74   0.613     8    -4.88     1      0.145
 3 0e7ipj0~ rocksta~ Post ~        0.587  0.535     5    -6.09     0      0.0898
 4 3swc6WT~ Psycho ~ Post ~        0.739  0.559     8    -8.01     1      0.117
 5 2G7V7zs~ In My F~ Drake         0.835  0.626     1    -5.83     1      0.125
 6 7dt6x5M~ Better ~ Post ~        0.68   0.563    10    -5.84     1      0.0454
 7 58q2HKr~ I Like ~ Cardi~        0.816  0.726     5    -4.00     0      0.129
 8 7ef4Dls~ One Kis~ Calvi~        0.791  0.862     9    -3.24     0      0.11
 9 76cy1WJ~ IDGAF    Dua L~        0.836  0.544     7    -5.98     1      0.0943
10 08bNPGL~ FRIENDS  Marsh~        0.626  0.88      9    -2.38     0      0.0504
# ... with 90 more rows, and 8 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>
```

`arrange()`

Let's rearrange the rows of the data frame to show the most danceable song first (since all we really care about is danceability!!).

```
df.data %>%
  arrange(desc(danceability))
```

```
# A tibble: 100 x 17
   id      name     artists danceability energy   key loudness  mode speechiness
   <chr>   <chr>    <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 6vN77l~ Yes Ind~ Lil Ba~        0.964  0.346     5    -9.31     0      0.53
 2 2E124G~ FEFE (f~ 6ix9ine        0.931  0.387     1    -9.13     1      0.412
 3 4qKcDk~ Look Al~ BlocBo~        0.922  0.581    10    -7.50     1      0.27
 4 0JP9xo~ Moonlig~ XXXTEN~        0.921  0.537     9    -5.72     0      0.0804
 5 0TlLq3~ Nonstop  Drake          0.912  0.412     7    -8.07     1      0.124
 6 6n4U3T~ Walk It~ Migos          0.909  0.628     2    -5.46     1      0.201
 7 3xcCix~ Bella    Wolfine        0.909  0.493     3    -6.69     1      0.0735
 8 7KXjTS~ HUMBLE.  Kendri~        0.908  0.621     1    -6.64     0      0.102
 9 3V8UKq~ Te Bot?~ Nio Ga~        0.903  0.675    11    -3.44     0      0.214
10 5IaHrV~ Taste (~ Tyga           0.884  0.559     0    -7.44     1      0.12
# ... with 90 more rows, and 8 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>
```

Note how I've used the `desc()` function here to arrange the data frame in descending order. To sort the data frame starting with the least danceable song, we would simply do:

```
df.data %>%
  arrange(danceability)
```

```
# A tibble: 100 x 17
   id      name     artists danceability energy   key loudness  mode speechiness
   <chr>   <chr>    <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 1j4kHk~ Dusk Ti~ ZAYN           0.258  0.437    11    -6.59     0      0.039
 2 2xGjte~ This Is~ Keala ~        0.284  0.704     2    -7.28     1      0.186
 3 0u2P5u~ lovely ~ Billie~        0.351  0.296     4   -10.1      0      0.0333
 4 1gm616~ Call Ou~ The We~        0.489  0.598     1    -4.93     1      0.036
 5 0s3nno~ Lucid D~ Juice ~        0.511  0.566     6    -7.23     0      0.2
 6 7vGuf3~ Silence  Marshm~        0.52   0.761     4    -3.09     1      0.0853
 7 5WvAo7~ No Brai~ DJ Kha~        0.552  0.76      0    -4.71     1      0.342
 8 3EPXxR~ Be Alri~ Dean L~        0.553  0.586    11    -6.32     1      0.0362
 9 75ZvA4~ I Fall ~ Post M~        0.556  0.538     8    -5.41     0      0.0382
10 0d2iYf~ Eastsid~ benny ~        0.56   0.68      6    -7.65     0      0.321
# ... with 90 more rows, and 8 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>
```

The DJ better not play "Dusk Till Dawn - Radio Edit" the next time I go out!

`mutate()`

We can create new variables using `mutate()`.

```
df.data %>%
  mutate(dance_energy = danceability + energy)
```

```
# A tibble: 100 x 18
   id      name     artists danceability energy   key loudness  mode speechiness
   <chr>   <chr>    <chr>          <dbl>  <dbl> <dbl>    <dbl> <dbl>       <dbl>
 1 6DCZcS~ God's P~ Drake          0.754  0.449     7    -9.21     1      0.109
```

```
 2 3ee8Jm~ SAD!     XXXTEN~        0.74   0.613     8    -4.88     1      0.145
 3 0e7ipj~ rocksta~ Post M~        0.587  0.535     5    -6.09     0      0.0898
 4 3swc6W~ Psycho ~ Post M~        0.739  0.559     8    -8.01     1      0.117
 5 2G7V7z~ In My F~ Drake          0.835  0.626     1    -5.83     1      0.125
 6 7dt6x5~ Better ~ Post M~        0.68   0.563    10    -5.84     1      0.0454
 7 58q2HK~ I Like ~ Cardi B        0.816  0.726     5    -4.00     0      0.129
 8 7ef4Dl~ One Kis~ Calvin~        0.791  0.862     9    -3.24     0      0.11
 9 76cy1W~ IDGAF    Dua Li~        0.836  0.544     7    -5.98     1      0.0943
10 08bNPG~ FRIENDS  Marshm~        0.626  0.88      9    -2.38     0      0.0504
# ... with 90 more rows, and 9 more variables: acousticness <dbl>,
#   instrumentalness <dbl>, liveness <dbl>, valence <dbl>, tempo <dbl>,
#   duration_ms <dbl>, time_signature <dbl>, rank <int>, dance_energy <dbl>
```

Let's take a look at the song with the most combined danceability and energy:

```
df.data %>%
  mutate(dance_energy = danceability + energy) %>%
  select(name, artists, dance_energy) %>%
  arrange(desc(dance_energy))
```

```
# A tibble: 100 x 3
   name                                                artists           dance_energy
   <chr>                                               <chr>                    <dbl>
 1 1, 2, 3 (feat. Jason Derulo & De La Ghetto)         Sofia Reyes               1.69
 2 One Kiss (with Dua Lipa)                            Calvin Harris             1.65
 3 Dura                                                Daddy Yankee              1.64
 4 Taki Taki (with Selena Gomez, Ozuna & Cardi B)      DJ Snake                  1.64
 5 Stir Fry                                            Migos                     1.63
 6 Criminal                                            Natti Natasha             1.63
 7 ?chame La Culpa                                     Luis Fonsi                1.62
 8 Feel It Still                                       Portugal. The Man         1.60
 9 Jackie Chan                                         Ti?sto                    1.58
10 Te Bot? - Remix                                     Nio Garcia                1.58
# ... with 90 more rows
```

Sofia Reyes wins!

group_by() and summarize()

Grouping and summarizing is a very powerful combination! For example, let's say that we are interested in what the average rank of each artist is who had more than one song in the top 100. Here is how we could go about it.

First, I group the data frame by the `artists` variable, and then I summarize what information I would like by group. Here, I calculate the mean rank, the standard deviation of the rank, and the number of hits (using the `n()` function) per artist. I then filter out only those artists who had more than 1 hit in the top 100, and arrange the data frame starting with the artists with the most hits.

```
df.data %>%
  group_by(artists) %>%
  summarize(mean_rank = mean(rank),
            sd_rank = sd(rank),
            n_hits = n()) %>%
  filter(n_hits > 1) %>%
  arrange(desc(n_hits)) %>%
  ungroup()
```

```
# A tibble: 18 x 4
   artists         mean_rank sd_rank n_hits
   <chr>               <dbl>   <dbl>  <int>
 1 Post Malone          33.2   35.4       6
 2 XXXTENTACION         41.2   33.3       6
 3 Drake                20.2   28.3       4
 4 Ed Sheeran           47     33.0       3
 5 Marshmello           42.3   28.7       3
 6 Ariana Grande        41.5   34.6       2
 7 Calvin Harris        49.5   58.7       2
 8 Camila Cabello       24     18.4       2
 9 Clean Bandit         64.5   46.0       2
10 Dua Lipa             17     11.3       2
11 Imagine Dragons      49.5    7.78      2
12 Kendrick Lamar       49.5   47.4       2
13 Khalid               57     42.4       2
14 Maroon 5             41.5   38.9       2
15 Migos                78      5.66      2
16 Ozuna                86      2.83      2
17 Selena Gomez         43      7.07      2
18 The Weeknd           61.5   16.3       2
```

Looks like Post Malone was killing it in 2018!

Here is more information about how to transform your data:

- Data transformation cheatsheet

Some more important verbs

Beating data into the shape we'd like it to be can be frustrating. So it's good practice to learn how to do it, so that you can get to the fun stuff as quickly as possible (such as making cool looking plots!).

Unfortunately, we won't have the time to look into data wrangling in this tutorial. Here is a table of some of the data manipulation verbs that you want to check out and play around with:

Table 3: Important data wrangling verbs to check out.

verb	description
`pivot_longer()`	transform a data frame from wide to long format
`pivot_wider()`	transform a data frame from long to wide format
`unite()`	unite multiple columns into one
`separate()`	separate a single column into several columns
`left_join()`	combine information from multiple data frames into one

Here is the data wrangling cheatsheet (data wrangling will take some time to get familiar with):

- Data wrangling cheatsheet

Practice time

What was the longest song in the Spotify top 100 of 2018?

```
# write your code here
```

What was the mean liveliness of all songs by Drake?

Statistics

As we've seen, R is great for plotting and data wrangling. It's also great for doing statistics! Again, We won't have the time to go into it in this class. Many of your statistical needs will be met by the following functions:

- Linear model `lm()`: for when you have independent observations.
- Linear mixed effects models `lmer()` (using `library("lme4")`): for when your data points aren't independent (e.g. when you have repeated observations from the same participants in your experiment).
- Bayesian models `brm()` (using `library("brms")`): if you'd like to try out some Bayesian data analysis. of your statistical needs will be met by the follow of your statistical needs will be met by the following functions:

Getting help

The best way to help others help you is by making a reproducible example (also called "reprex"). The "reprex" package makes it easy to generate a reproducible example that you can then share with others.

You can install the package like so:

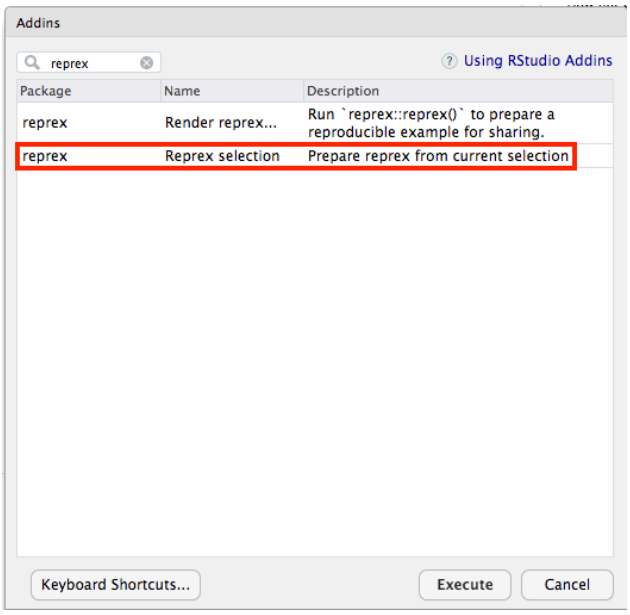
```
install.packages("reprex")
```

Now, you have a new RStudio add-in that you can use for making reproducible examples. First, select that code that you want to use for generating the example:

```
library("ggplot2")

ggplot(data = mpg,
       mapping = aes(x = class,
                     y = cty)) +
  geom_bar()
```

Then go to `Tools > Addins > Browse Addins` and select `Reprex selection` (see Figure):



An alternative way is to fire up the command console via `cmd + shift + p` and then search for the command there (this is much quicker!).

You'll get the following output of running reprex on this code, which you can then email to your colleague, or share on stackoverflow when posting a question.

```
library("ggplot2")

ggplot(data = mpg,
       mapping = aes(x = class,
                     y = cty)) +
  geom_bar()
#> Error: stat_count() can only have an x or y aesthetic.
```

Using reprex, you will make sure that the other person will be able to recreate the error message that you got (because it runs the code with a clear environment – i.e. without any packages already loaded, are variables that you may have stored in your environment).

For example, if you were to run reprex on this piece of code …

```
ggplot(data = mpg,
       mapping = aes(x = class,
                     y = cty)) +
  geom_bar()
```

… the output would be the following:

```
ggplot(data = mpg,
       mapping = aes(x = class,
                     y = cty)) +
  geom_bar()
#> Error in ggplot(data = mpg, mapping = aes(x = class, y = cty)): could not find function "ggplot"
```

You can learn more about the reprex package here: https://github.com/tidyverse/reprex

Where can I learn more?

Here is a list of excellent free online books that you should check out!

- R for Data Science: The tidyverse bible.
- Data Visualization – A practical introduction (by Kieran Healy): Great resourse for data visualization with `ggplot2`.
- Fundamentals of Data Visualization: Another excellent resource for data visualization with `ggplot2`.
- R graphics cookbook: Quick intro to the the most common graphs with `ggplot2`.
- Statistical thinking for the 21st century: Course notes for psychology undergraduate statistics course taught by Russ Poldrack here at Stanford.
- Statistical methods for behavioral and social sciences: Course notes for grad statistics course I teach (notes are not fully self-explanatory though).
- YaRrr! The Pirate's Guide to R: Nice general introduction to R (using mostly base R).
- Learning statistics with R: Introduction to statistics using R (using mostly base R).

And of course, google and stack overflow will be your best friends when figuring stuff out!

Session information

```
sessionInfo()
```

```
R version 4.1.1 (2021-08-10)
Platform: x86_64-apple-darwin17.0 (64-bit)
Running under: macOS Big Sur 10.16

Matrix products: default
```

```
BLAS:   /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] forcats_0.5.1   stringr_1.4.0   dplyr_1.0.7     purrr_0.3.4
 [5] readr_2.0.2     tidyr_1.1.4     tibble_3.1.5    ggplot2_3.3.5
 [9] tidyverse_1.3.1 knitr_1.36

loaded via a namespace (and not attached):
 [1] nlme_3.1-153        fs_1.5.0            lubridate_1.7.10
 [4] bit64_4.0.5         RColorBrewer_1.1-2  httr_1.4.2
 [7] tools_4.1.1         backports_1.2.1     utf8_1.2.2
[10] R6_2.5.1            rpart_4.1-15        Hmisc_4.5-0
[13] DBI_1.1.1           mgcv_1.8-37         colorspace_2.0-2
[16] nnet_7.3-16         withr_2.4.2         tidyselect_1.1.1
[19] gridExtra_2.3       bit_4.0.4           compiler_4.1.1
[22] cli_3.0.1           rvest_1.0.1         htmlTable_2.2.1
[25] xml2_1.3.2          labeling_0.4.2      scales_1.1.1
[28] checkmate_2.0.0     digest_0.6.28       foreign_0.8-81
[31] rmarkdown_2.11      base64enc_0.1-3     jpeg_0.1-9
[34] pkgconfig_2.0.3     htmltools_0.5.2     dbplyr_2.1.1
[37] fastmap_1.1.0       highr_0.9           htmlwidgets_1.5.4
[40] rlang_0.4.11        readxl_1.3.1        rstudioapi_0.13
[43] farver_2.1.0        generics_0.1.0      jsonlite_1.7.2
[46] vroom_1.5.5         magrittr_2.0.1      Formula_1.2-4
[49] Matrix_1.3-4        Rcpp_1.0.7          munsell_0.5.0
[52] fansi_0.5.0         lifecycle_1.0.1     stringi_1.7.4
[55] yaml_2.2.1          grid_4.1.1          parallel_4.1.1
[58] crayon_1.4.1        lattice_0.20-45     haven_2.4.3
[61] splines_4.1.1       hms_1.1.1           pillar_1.6.3
[64] reprex_2.0.1        glue_1.4.2          evaluate_0.14
[67] latticeExtra_0.6-29 data.table_1.14.2   modelr_0.1.8
[70] png_0.1-7           vctrs_0.3.8         tzdb_0.1.2
[73] cellranger_1.1.0    gtable_0.3.0        assertthat_0.2.1
[76] xfun_0.26           broom_0.7.9         survival_3.2-13
[79] cluster_2.1.2       ellipsis_0.3.2
```

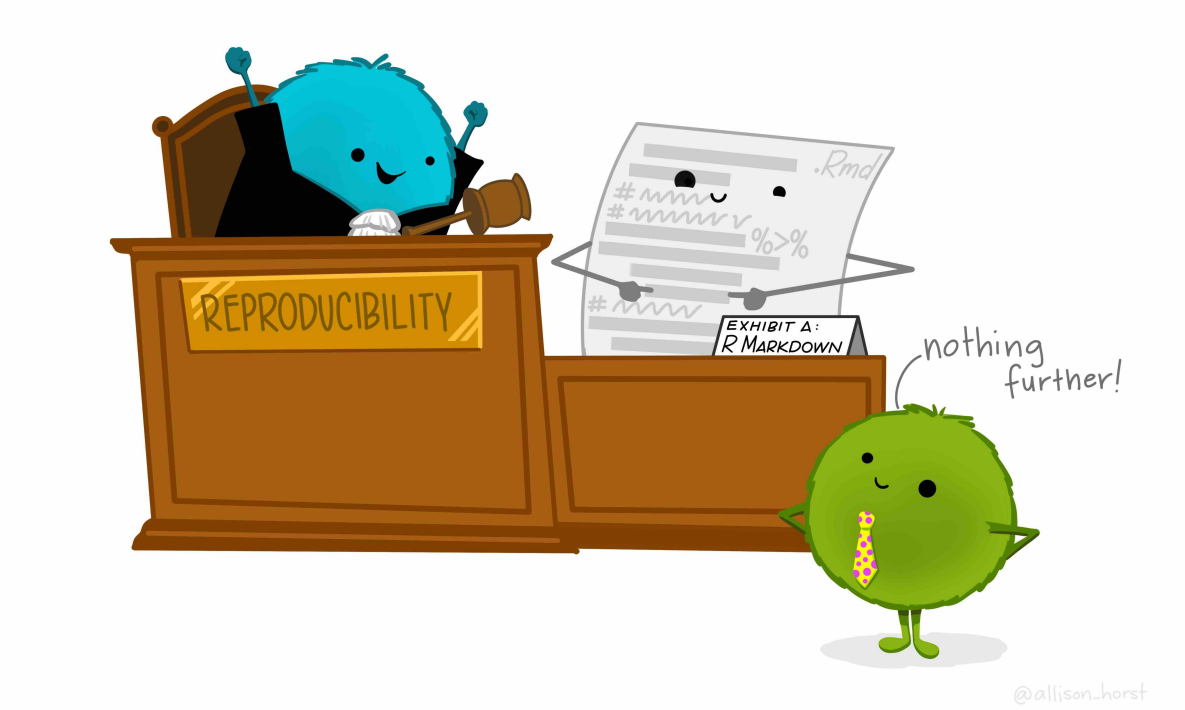


Figure 14: Defense at the reproducibility court.