An introduction to R for sensory and consumer scientists

Jacob Lahne[1] Leah Hamilton[2]

[1]Virginia Tech, jlahne@vt.edu
[2]Virginia State University, lhamilton@vsu.edu

Contents

Introduction and welcome

Welcome to the Pangborn Sensometrics Workshop "**An introduction to R for sensory and consumer scientists**"!

This workshop is going to be conducted not using slides, but through **livecoding**. That means we are going to run code lines in the console or highlight and run code in scripts and other files. It is also an opportunity and encouragement for you to follow along. Along with introducing ourselves for today's workshop, we're going to discuss a bit about how that will work here.

Introductions

Leah Hamilton, PhD

Leah Hamilton is an Assistant Professor of Sensory & Flavor Science at Virginia State University, in the US. Her primary research interest is flavor language, including the ways that people talk about flavors using their own words in different contexts. In her new position at Virginia State University, she'll be working closely with plant breeders and agricultural scientists to develop sensory-driven specialty crops and support the work of small and underprivileged farmers in the mid-Atlantic US.

Elizabeth Clark, PhD

Elizabeth Clark is a Senior Scientist in Sensory & Consumer Sciences at McCormick & Company Inc. — a global leader in flavor operating in two segments across 170 countries and territories. McCormick's passion for Sensory & Consumer Science has led to published research on a replacement for the Scoville heat method for the sensory determination of pungency in capsicum products (Gillette, Appel, & Leggo, 1984); The Sensory Quality System (SQS): a global quality control solution (King et.al, 2022); the EsSense Profile®— a scientific measurement of the human emotional response to flavor (King & Meiselman, 2010), and The Wellsense Profile™ — a questionnaire to measure consumer

wellness with foods (King et.al, 2015). Leveraging her interests in data analytics & coding, Elizabeth is helping McCormick usher in a new era of sensory research geared toward addressing rapidly evolving challenges faced by global food & beverage companies.

Sébastien Lê, PhD

Sébastien Lê is an Associate Professor of Statistics and Computer science at l'Institut Agro Rennes-Angers, in France. He's a co-author of the FactoMineR and SensoMineR R packages, a co-author of *Analyzing Sensory Data with R*, and a co-author of the Sensory %>% Data %>% Science educational platform. He's always curious and open to all kind of collaborations.

Jacob Lahne, PhD

Jacob Lahne is an Associate Professor of Food Science & Technology at Virginia Tech, in the United States. He runs the Virginia Tech Sensory Evaluation Laboratory, as well as teaching courses in data analytics and coding for food-science research. His main research focuses are sensory data-analysis methodologies and investigating the sensory properties of fermented and distilled foods and beverages. His work was invaluable in putting this workshop together, but unfortunately he won't be joining us in Nantes.

Today's agenda

Today's workshop is going to take ~3 hours, with a break for lunch, and we'll be covering the following material:

1. Crash course in using R
2. Creating, importing, and manipulating data in R
3. Tidy Data Analysis: Rows, Columns, and Groups
    1. What is tidy data?
    2. Subsetting data
    3. Chaining steps together
    4. Making new variables
    5. Groups of rows and split-apply-combine
    6. Groups of columns
4. Tidy Data Analysis: Reshaping and combining tables
    1. Wider and longer data
    2. Combining data frames

    3. Other data-wrangling utilities

5. Data analysis outside the `tidyverse`

    1. Correspondence Analysis overview
    2. Working with binary, count, and character data
    3. Untidying & Retidying data

6. Basics of data visualization in `R`/`ggplot2`

    1. Built-in plots using ca package
    2. Customizing plots with ggplot2

How we're going to run

This workshop is going to be run with **livecoding**, as noted above. This means we won't be using slides or a prepared video, but running through code step by step to show how these tools are used in practice. We encourage **you** to also follow along with livecoding, because the best way to learn coding is to actually do it.

Recommended approach for livecoding

We recommend that you download the pre-made archive of code and data from the workshop github repo. This archive, when unzipped, will have a folder structure and a `.Rproj` file. We recommend that you close out RStudio, unzip the archive, and double click the `.Rproj` file *in that folder*, which will open a new session of RStudio with proper setting (like the home directory) for the files for this workshop.

In that folder, you will find a `data/` folder with the necessary data for the workshop, and a script named `pangborn-all-code.R`. This latter file contains all of the code demonstrated in this workshop for your future reference. You can also follow along with the code at the workshop's page hosted on github.io (which you're reading right now), and which will remain available after this workshop.

Once you're in RStudio, go to the `File > New File > R Script` menu to open a new script. We'll talk about how these work in a minute, but this is basically a workbook for you to store sequential lines of code to be run in the `Console`. It is where you can livecode along! Even though we are giving you all of the code you need right now, you will learn a lot more if you actively follow along, rather than just run that code.

Dealing with errors

Coding means **making mistakes**. This is fine–as you will surely see today, I will make a ton of trivial errors and have to fix things on the fly. If you run into trouble, try looking carefully at what you've done and see if you can see what went wrong. If that fails, we are here to help! Because we have 3 instructors for this workshop, two of us are available to help at any time.

When you run into trouble, please use the **red sticky note** by putting it on the back of your laptop. We'll be keeping an eye out, and someone will come to help you. When you've resolved your problem, take the sticky note back off. This way you don't have to raise your hand and interrupt the workshop, etc. However, if your issue is a common one or something we think is worth noting, don't worry–we'll make time to discuss it!

Chapter 1

A crash course in R

In this section, we're going to go over the basics of R: what the heck you're looking at, how the RStudio IDE works, how to extend R with packages, and some key concepts that will help you work well in R.

1.1 R vs RStudio

In this workshop we are going to be learning the basics of coding for text analysis in R, but we will be using the RStudio interface/IDE! Why am I using R for this workshop? And why do we need this extra layer of program to deal with it?

1.1.1 What is R?

R is a programming language that was built for statistical and numerical analysis. It is not unique in these spaces–most of you are probably familiar with a program like SAS, SPSS, Unscrambler, XLSTAT, JMP, etc. Unlike these, R is **free** and **open-source**. This has two main consequences:

1. R is constantly being extended to do new, useful things because there is a vibrant community of analysts developing tools for it, and the barrier to entry is very low.

2. R doesn't have a fixed set of tasks that it can accomplish, and, in fact, I generally haven't found a data-analysis task I needed to do that I *couldn't* in R.

Because it's a programming **language**, R isn't point-and-click–today we're going to be typing commands into the console, hitting, enter, making errors, and

repeating. But this is a good thing! The power and flexibility of R (and its ability to do most of the things we want) come from the fact that it is a programming language. While learning to use R can *seem* intimidating, the effort to do so will give you a much more powerful suite of tools than the more limited point-and-click alternatives. R is built for research programming (data analysis), rather than for production programming. The only other alternative that is as widely supported in the research community is Python, but–honesty time here–I have never learned Python very well, and so we are learning R. And, in addition, Python doesn't have as good an Interactive Development Environment (IDE, explained further below) as RStudio!

If you open your `R.exe`/R application, you'll see something like this:

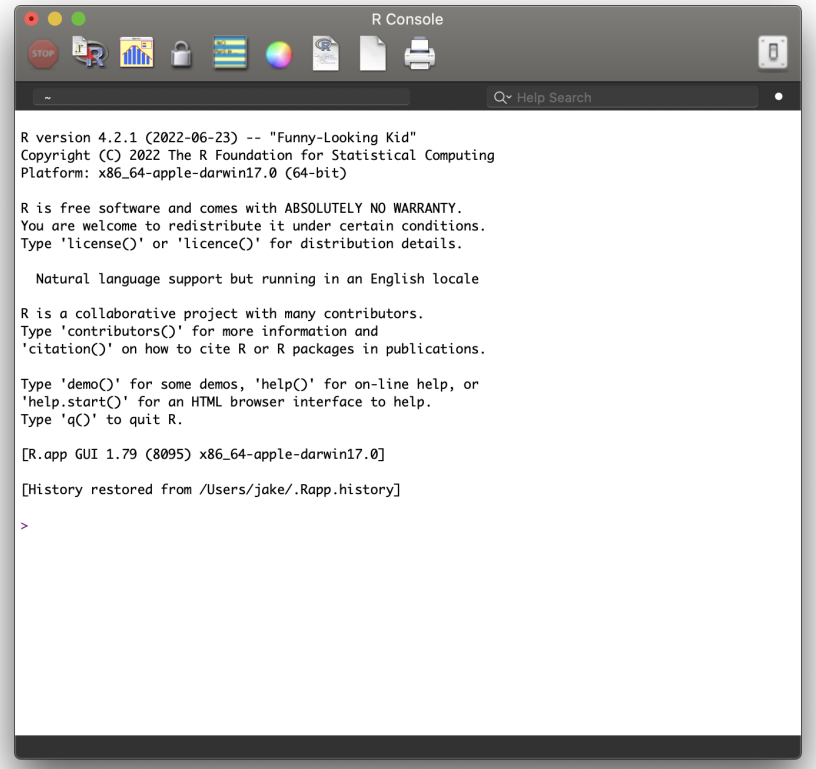


Figure 1.1: The R graphical console

You can also work with R from a shell interface, but I will not be discussing this approach in this workshop.

1.1.2 Great, why are we using RStudio then?

RStudio is an "**I**nteractive **D**evelopment **E**nvironment" (IDE) for working with `R`. Without going into a lot of detail, that means that `R` lives on its own on your computer in a separate directory, and RStudio provides a bunch of better functionality for things like writing multiple files at once, making editing easier, autofilling code, and displaying plots. You can learn more about RStudio here.

With that out of the way, I am going to be sloppy in terminology and say/type "`R`" a lot of the times I mean "RStudio". I will be very clear if the distinction actually matters. RStudio is going to make your life way easier, and when you try to learn Python you are going to be sad :(

1.2 The parts of RStudio

The default layout of RStudio looks something like this (font sizes may vary):

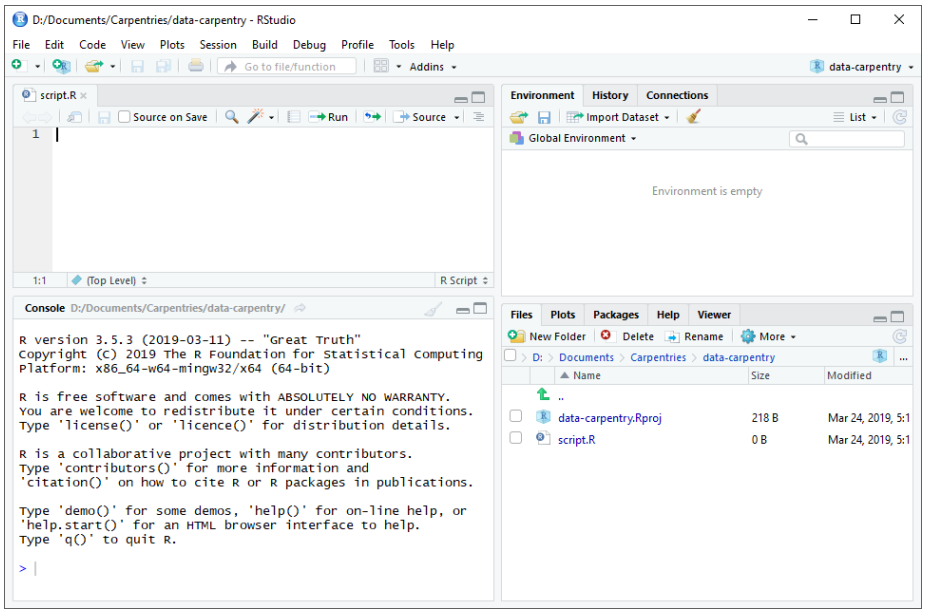


Figure 1.2: RStudio default layout, courtesy of Data Carpentry

RStudio always has 4 "panes" with various functions, which are in tabs (just like a web browser). The key ones for right now to pay attention are:

1. The `Console` tab is the portal to interact directly with `R`. The `>` "prompt" is where you can type and execute commands (by hitting return). You

can try this out right now by using it like a calculator - try `1 + 1` if you like!
2. The `Files` tab shows the files in your **working directory**: like in the Windows Explorer or macOS Finder, files are displayed within folders. You can click on files to open them.
3. The `Help` tab shows documentation for `R` functions and packages–it is useful for learning how to use specific functions.
4. The `Plots` tab shows graphical output, and this is where the data visualizations we'll learn to make will (generally) appear.
5. The `Environment` tab shows the objects that exist in memory in your current `R` session. Without going into details, this is "what you've done" so far: data tables and variables you've created, etc.
6. Finally, the `Scripts` **pane** shows *individual* tabs for each script and other RStudio file. Scripts (and other, more exotic file types like RMarkdown/.Rmd files) are documents that contain multiple `R` commands, like you'd type into the `Console`. However, unlike commands in the `Console`, these commands don't disappear as soon as they're run, and we can string them together to make workflows or even programs. This is where the real power of `R` will come from.

You can change the layout of your Panes (and many other options) by going to the RStudio menu: `Tools > Global Options` and select `Pane Layout`.

You'll notice that my layout for RStudio looks quite different from the default, but you can always orient yourself by seeing what tab or pane I am in–these are always the same. I prefer giving myself more space for writing `R` scripts and markdown files, so I have given that specific Pane more space while minimizing the `History` pane.

While we're in Global Options, please make the following selections:

1. Under `General`, uncheck all the boxes to do with restoring projects and workspaces. We want to make sure our code runs the same time every time (i.e., that our methods are reproducible), and letting RStudio load these will make this impossible:

2. Make your life easier by setting up **autocompletion** for your code. Under the `Code > Completion` options, select the checkboxes to allow using `tab` for autocompletions, and also allowing multiline autocompletions. This means that RStudio will suggest functions and data for you if you hit `tab`, which will make you have to do way less typing:

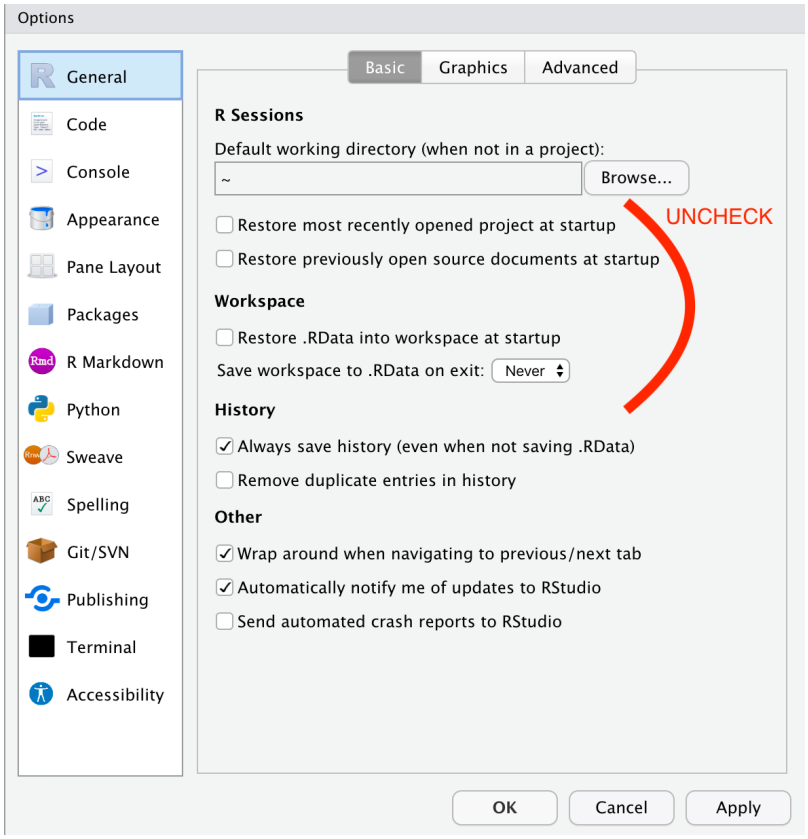


Figure 1.3: Uncheck the options to restore various data and projects at startup.

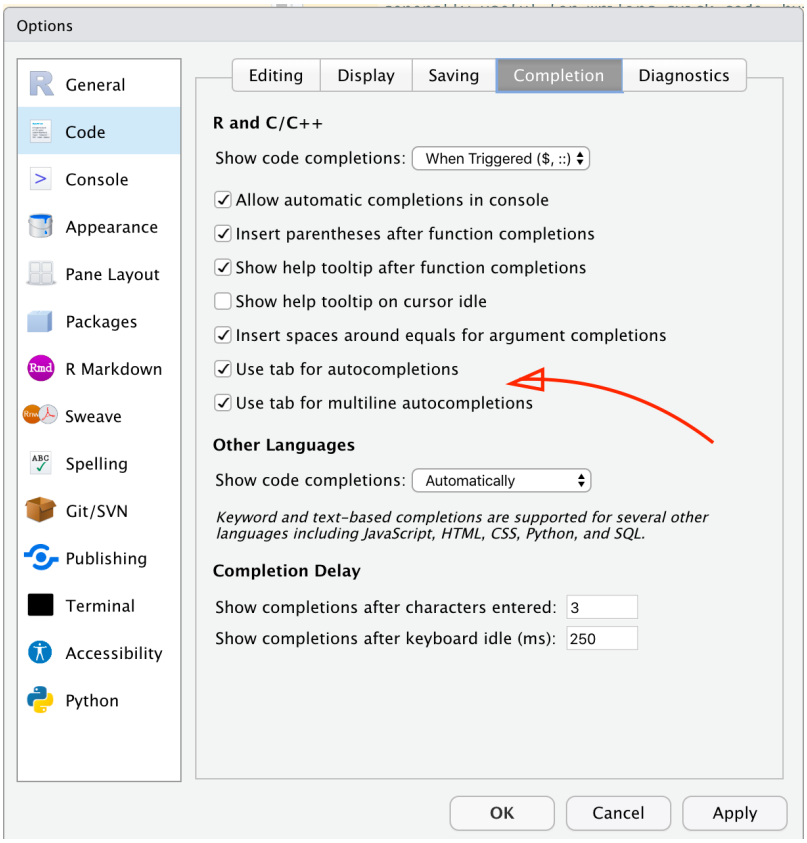


Figure 1.4: Check the boxes for tab and multiline autocompletions.

1.2.1 The "working directory" and why you should care

Before we move on to using R for real, we have one key general computing concept to tackle: the "working directory". The working directory is the folder on your computer in which R will look for files and save files. When you need to tell R to read in data from a file or output a file, you will have to do so **in relation to your working directory**. Therefore, it is important that you know how to find your working directory and change it.

The easiest (but not best) way to do this is to use the Files pane. If you hit the "gear" icon in the Files pane menu, you'll see two commands to do with the working directory. You can Go To Working Directory to show you whatever R currently has set as the working directory. You can then navigate to any directory you want on your hard drive, and use the Set As Working Directory command to make that the working directory.

A better way to do this is to use the R commands getwd() and setwd().

```
getwd() # will print the current working directory
```

```
## [1] "C:/Users/Leah/Documents/R/pangborn-r-tutorial-2023"
```

And we can manually change the working directory by using

```
setwd("Enter/Your/Desired/Directory/Here")
```

Notice that I am not running the second command, because it would cause an error!

When we use R to navigate directories, I recommend **always** using the forward slash: /, even though on Windows systems the typical slash is the backslash: \. R will properly interpret the / for you in the context of your operating system, and this is more consistent with most modern code environments.

1.3 Extending R with packages

One of the key advantages of R is that its open-source nature means that you can extend it to do all sorts of things. For example, for much of this workshop we are going to be going about basic text analysis using the tidytext package. There are various ways to install new packages, but the easiest way is to use the Packages tab. This will show you all the packages you currently have installed as an alphabetical list.

1.3.1 Installing packages

To install a new package, you can select the `Install` button from the `Packages` tab, which will give you a prompt to type the package name in. You can get to the same prompt by going to the `Tools > Install Packages...` menu. On this prompt, you can list packages separated by a comma (`,`), which is convenient. RStudio will also try to help you by autocompleting package names.

You should have already installed the `tidyverse` package as part of your prework for this workshop. Now, let's go ahead and install the `tidytext` package, which we'll use later in this workshop. If you didn't install the `tidyverse` package, you can list it along with the `tidytext` package.

You'll note that hitting `Install` made a line of code appear in your console, something like:

```
install.packages("ca")
```

This is the "true" `R` way to install packages–the function `install.packages()` can be run on the `Console` to install whatever package is quoted inside the parentheses.

You can get `R` packages from a variety of sources. The most common are repositories, like CRAN, which is where you first downloaded `R`. There are others, like Bioconductor, which is used more by the bioinformatics community. You might also sometime download an install a package that isn't on a repository, such one from github (for example this one), but I am not going to cover that in detail here.

1.3.2 Loading packages

To actually use a package, you need to load it using the `library(<name of package>)` command. So, for example, to load the `tidyverse` package we will use the command

```
library(tidyverse)
```

You need to use multiple `library()` commands to load multiple packages, e.g.,

```
library(tidyverse)
library(ca)
```

If you want to know what packages you have loaded, you can run the `sessionInfo()` function, which will tell you a bunch of stuff, including the "attached" packages:

```
sessionInfo()
```

```
## R version 4.3.0 (2023-04-21 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19044)
##
## Matrix products: default
##
##
## locale:
## [1] LC_COLLATE=English_United States.utf8
## [2] LC_CTYPE=English_United States.utf8
## [3] LC_MONETARY=English_United States.utf8
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.utf8
##
## time zone: America/New_York
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] ca_0.71.1       lubridate_1.9.2 forcats_1.0.0   stringr_1.5.0
##  [5] dplyr_1.1.2     purrr_1.0.1     readr_2.1.4     tidyr_1.3.0
##  [9] tibble_3.2.1    ggplot2_3.4.2   tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] bit_4.0.5        gtable_0.3.3     crayon_1.5.2     compiler_4.3.0
##  [5] tidyselect_1.2.0 parallel_4.3.0   scales_1.2.1     yaml_2.3.7
##  [9] fastmap_1.1.1    R6_2.5.1         generics_0.1.3   knitr_1.42
## [13] bookdown_0.34    munsell_0.5.0    pillar_1.9.0     tzdb_0.3.0
## [17] rlang_1.1.1      utf8_1.2.3       stringi_1.7.12   xfun_0.39
## [21] bit64_4.0.5      timechange_0.2.0 cli_3.6.1        withr_2.5.0
## [25] magrittr_2.0.3   digest_0.6.31    grid_4.3.0       vroom_1.6.3
## [29] rstudioapi_0.14  hms_1.1.3        lifecycle_1.0.3  vctrs_0.6.2
## [33] evaluate_0.21    glue_1.6.2       fansi_1.0.4      colorspace_2.1-0
## [37] rmarkdown_2.21   tools_4.3.0      pkgconfig_2.0.3  htmltools_0.5.5
```

Finally, you can also load (and unload) packages using the `Packages` tab, by clicking the checkbox next to the name of the package you want to load (or unload).

1.4 Getting help

With more packages you're going to more frequently run into the need to look up how to do things, which means dealing with help files. You can always get help on a particular function by typing `?<search term>`, which will make the help documentation for whatever you've searched for appear.

For example, try typing the following to get help for the `sessionInfo()` command:

```
?sessionInfo
```

But what if you don't know what to search for?

By typing `??<search term>` you will search **all** help files for the search term. R will return a list of matching articles to you in the help pane. This is considerably slower, since it's searching hundreds or thousands of text files. Try typing `??install` into your console to see how this works.

You will notice that there are two types of results in the help list for install. The help pages should be familiar. But what are "vignettes"? Try clicking on one to find out.

Vignettes are formatted, conversational walkthroughs that are increasingly common (and helpful!) for R packages. Rather than explaining a single function they usually explain some aspect of a package, and how to use it. And, even better for our purposes, they are written in R Markdown. Click the "source" link next to the vignette name in order to see how the author wrote it in R Markdown. This is a great way to learn new tricks.

While you can find vignettes as we just did, a better way is to use the function `browseVignettes()`. This opens a web browser window that lists **all** vignettes installed on your computer. You can then use `cmd/ctrl + F` to search using terms in the web browser and quickly find package names, function names, or topics you are looking for.

1.5 Livecoding along

We've now covered the `Console` tab and the `Scripts` pane. These are both areas in which you can write and execute code, but they work a little differently. The `Console` is the place to run code that is short and easy to type, or that you're experimenting with. It will allow you to write a single line of code, and after you hit `return`, `R` will execute the command. This is great for "interactive programming", but it isn't so great for building up a complex workflow, or for following along with this workshop!

This is why I have recommended that you create a new script to follow along with this workshop. Again, you get a new script by going to `File > New File > R Script`. You can write multiple lines of code and then execute each one in any order (although keeping a logical sequence from top to bottom will help you keep track of what you're doing). In an `R` script, everything is expected to be valid R code.

```
You can't write this in an R script because it is plain text.  This will
cause an error.

# If you want to write text or notes to yourself, use the "#" symbol at the start of
# every line to "comment" out that line.  You can also put "#" in the middle of
# a line in order to add a comment - everything after will be ignored.

1 + 1 # this is valid R syntax

print("hello world") # this is also valid R syntax
```

To run code from your `R` script, put your cursor on the line you want to run and either hit the run button with the green arrow at the top left or (my preferred method) type `cmd + return` (on Mac) or `ctrl + return` (on PC).

Chapter 2

How to work with R

Now that we've all got R up and running, we're going to quickly go over the basic functionality of `R` to make sure everyone is on the same page. If you have ever used `R`, this might include some review, but my hope is that this will be helpful to everyone and get us all on the same page before we launch into some more advanced applications.

We're going to speed through the basics of the console (working interactively with `R`) and then some of the "programming"/"coding" capabilities in `R`.

2.1 Doing math and creating objects in `R`

At it's most basic, `R` can be a calculator. If you type math into the `Console` and hit `return` it will do math for you!

```
2 + 5
```

```
## [1] 7
```

```
1000 / 3.5
```

```
## [1] 285.7143
```

To get the most out of R, though, we are going to want to use its abilities as a programming language. Among some other topics that I won't explicitly cover today, this means using `R` to store values (and later objects that contain multiple lines of values, like you'd get in an Excel spreadsheet).

This set of characters is the **assignment operator**: `<-`. It works like this:

```
x <- 100
hi <- "hello world"
data_set <- rnorm(n = 100, mean = 0, sd = 1)
```

. . . but that didn't do anything! Where's the output? Well, we can do two things. First, look at the "Environment" tab in your RStudio after you run the above code chunk. You'll notice that there are 3 new things there: `x`, `hi`, and `data_set`. In general I am going to call those **objects**–they are stored variables, which R now knows about by name. How did it learn about them? You guessed it: the assignment operator: `<-`.

To be explicit: `x <- 100` can be read in English as "x gets 100" (what a lot of programmers like to say) or, in a clearer but longer way, "assign 100 to a variable called x".

NB: R also allows you to use `=` as an assignment operator. **DO NOT DO THIS!**. There are two good reasons.

1. It is ambiguous, because it is not directional (how are you sure what is getting assigned where?)
2. This makes your code super confusing because `=` is the *only* assignment operator for *arguments* in functions (as in `print(quote = FALSE)` see below for more on this)
3. Anyone who has used R for a while who sees you do this will roll their eyes and kind of make fun of you a little bit

NB2: Because it is directional, it is actually possible to use `->` as an assignment operator as well. What do you think it does? Check and find out.

If you find typing `<-` a pain (reasonably), RStudio provides a keyboard shortcut: either `option + -` on Mac or `Alt + -` on PC.

I am going to use the terms "object" and "variable" interchangeably in this workshop–in other programming languages there are hard distinctions between these concepts, and even in some `R` packages this can be important, but for our purposes I mean the same thing if I am sloppy.

The advantage of storing values into objects is that we can do math with them, change them, and duplicate and store them elsewhere.

```
x / 10
```

```
## [1] 10
```

```
x + 1
```

```
## [1] 101
```

Note that doing math with a variable doesn't change the variable itself. To do that, you have to use the assignment `<-` to change the value of the variable again:

```
x
```

```
## [1] 100
```

```
x <- 100 + 1
x
```

```
## [1] 101
```

2.2 Functions and their arguments in R

Obviously if we're using R as a calculator we might want to do more than basic arithmetic. What about taking the square root of `x`?

```
sqrt(x)
```

```
## [1] 10.04988
```

In fact, we can ask R to do lots of neat stuff, like generate random numbers for us. For example, here are 100 random numbers from a normal distribution with mean of 0 and standard deviation of 1.

```
rnorm(n = 100, mean = 0, sd = 1)
```

```
##   [1]  0.3127088404  0.7811740067 -0.0215795288  1.0235217924 -0.7251987669
##   [6]  0.3727787092  1.8036557121 -0.0178840161  0.9959159908  0.6009265661
##  [11] -0.8688703459  1.3245708592  1.5943777323 -0.8717920674  0.4635823800
##  [16] -0.4508236879 -0.7577804727  0.6180873500  0.9591681449  0.1085341079
##  [21] -0.5628465428  0.9923773209 -0.8125048557 -1.9232796791 -0.5010334053
##  [26] -0.7742001146 -0.0970364010 -1.2308403095  1.5217743317  2.1809419251
##  [31]  0.0602903677 -0.5075360006  0.3387776652  0.6847487314  0.5489826537
##  [36]  1.1707115058  0.7944680476 -0.4693476850 -0.4319151415  1.6035968920
##  [41]  0.4070418717 -0.5357572540  0.2719328442  0.0008122861  0.7409776357
##  [46] -0.8331587604 -0.1526313947 -0.5167037484 -1.0929257235  0.3170995516
##  [51]  0.8632677583 -0.2506240280  0.0385318543  0.2838225075  0.0499514203
##  [56] -0.0946479888 -0.3059885609 -0.7328641563  0.6611459605  1.0402169375
##  [61]  0.9267740581  1.1449916307 -0.5905126178 -0.8354054905 -0.9222455707
##  [66]  0.0119571348  1.1452873421  0.3166335507  0.0209763772 -1.2252522254
```

```
##  [71] -1.7036696849  1.5377006653 -0.5053489443 -1.0450196747  0.0198555816
##  [76]  0.4638127307 -0.0663671849  0.3539525869 -0.5772909553  0.3607914730
##  [81]  1.5871903080 -0.1266737033  0.5548421744 -1.5813153725  0.6244765434
##  [86] -0.4747060467  0.0531680539  0.0663849219 -0.1492569069  0.6111938410
##  [91]  0.4590638118  2.1055792242  0.0835077185 -1.8789901791 -0.7159629742
##  [96]  1.1392029786  0.2485645496  0.4365126039  0.2855317261 -0.9579557834
```

These forms are called a **function** in R. Functions lie at the heart of `R`'s power: they are pre-written scripts that are included with base `R` or added in packages, like the ones we installed. In general, an `R` function will have a form like `<name>(<argument>, <argument>, ...)`. In other words, the function will have a name (that lets `R` know what you're trying to do) followed by an open parenthesis, and inside that a list of arguments, which are variables, objects, values, etc that you "pass" to the function, finally followed by a close parenthesis. In the case of our `sqrt()` function, there is only a single argument: a variable to which the square-root operation will be applied. In the case of the `rnorm()` function there are 3 arguments: the number of values we want, `n`, and the `mean` and standard deviation `sd` of the normal distribution we wish to sample from.

Functions are the `R` tools for which you can make the most use of the help operator: `?`. Try typing `?rnorm` into your console, and when you hit `return` you'll see the help page for the function.

Notice that in the `rnorm()` example we *named* the arguments–we told `R` which was the `n`, `mean`, and `sd`. This is because if we name arguments, we can give them in any order. Otherwise, `R` will try to match the provided values to the arguments **in the order in which they are given in the help file**. This is a major source of errors for newer `R` users!

```
# This will give us 1 value from the normal distribution with mean = 0 and sd = 100
rnorm(1, 0, 100)
```

```
## [1] 27.46005
```

```
# But we can also use named arguments to provide them in any order we wish!
rnorm(sd = 1, n = 100, mean = 0)
```

```
##   [1]  0.846774745  1.000191578 -0.145750723  1.566699892  1.753139051
##   [6] -0.115245345  0.321254828  0.000450397  0.740293204  0.181187066
##  [11] -0.887583241 -0.618598590 -0.330435879 -2.417908767 -0.072796750
##  [16]  0.532909880  0.456909453  0.473676409  1.997337995 -2.005915574
##  [21]  0.841572722  1.135965809 -0.131080007 -1.178496947  2.149888857
##  [26] -0.350743758 -1.077018392  0.631203417 -2.862627387 -1.368542826
##  [31] -1.111228779  1.156583220 -1.269423061 -1.158976889 -0.471091408
##  [36]  1.762110724  0.404541511  2.040268190  1.597849153  1.775415738
```

```
##  [41] -1.700472927 -0.193710062  0.476411479 -0.841765728 -0.190948680
##  [46]  1.040547336  0.412257056 -0.680708660  0.523015149 -2.103943339
##  [51]  0.779978705 -0.760464170 -0.740882618  0.999925500 -1.380059203
##  [56] -0.473940949  1.826746300  1.553454122 -0.744743699  0.250054291
##  [61] -1.663891798 -1.279729088  0.025808513 -0.078183147  1.359307795
##  [66]  0.899439097 -0.363354954  0.608445014  0.692863220  0.139534597
##  [71]  0.093151850 -0.432444071  0.221886011 -0.673139279  2.066764307
##  [76]  1.245291063 -0.654288364 -0.643006370 -0.282278725 -0.625371213
##  [81] -1.113960818  2.596223612 -1.307314828  0.289711362 -1.828386767
##  [86] -2.098665031 -0.397126873  0.780363125 -0.201391022 -1.447829968
##  [91]  1.016883364  0.704308656 -0.534276737  0.077979801  0.391692282
##  [96] -0.881415459 -0.400030573  0.049131806  1.722677999 -0.320569208
```

Programming languages like R are very literal, and we need to be as literal as we can to make them work the way we want them to.

2.3 Reading data into R

So far we've only done very basic things with R, which probably haven't sold you on its power and utility. Let's try doing some things that will hopefully get us a little further towards actual, useful applications.

First off, make sure you have the `tidyverse` package loaded by using the `library()` function.

```
library(tidyverse)
```

Now, we're going to read in the data we're going to use for this workshop using the function `read_csv()`. If you want to learn about this function, use the `?read_csv` command to get some details.

In the workshop archive you downloaded, the `data/` directory has a file called `clt-berry-data.csv`. If you double-click this in your file browser, it will (most likely) open in Excel, and you can take a look at it. The data describe the attributes and liking scores reported by consumers for a variety of berries across multiple CLTs. A total of 969 participants (`Subject Code`) and 23 berries (`Sample Name`) were involved in these tests, with only one species of berry (blackberry, blueberry, raspberry, or strawberry) presented during each CLT. In the actual experimental design, subjects got multiple sample sets (so there are *not* 969 unique subjects), but here we will treat them as unique for ease of description.

As for the survey, panelists were asked JAR, CATA, and free response questions, as well as liking on the Unlabeled Line Scale (`us_*`), 9-point scale (`9pt_*`), or labeled magnitude scale (`lms_*`).

To get this data into R, we have to tell R where it lives on your computer, and what kind of data it is.

2.3.1 Where the data lives

We touched on **working directories** because this is how R "sees" your computer. It will look first in the working directory, and then you will have to tell it where the file is *relative* to that directory. If you have been following along and opened up the `.Rproj` file in the downloaded archive, your working directory should be the archive's top level, which will mean that we only need to point R towards the `data/` folder and then the `clt-berry-data.csv` file. We can check the working directory with the `getwd()` function.

```
getwd()
```

```
## [1] "C:/Users/Leah/Documents/R/pangborn-r-tutorial-2023"
```

Therefore, **relative to the working directory**, the file path to this data is `data/clt-berry-data.csv`. Please note that this is the UNIX convention for file paths: in Windows, the backslash \ is used to separate directories. Happily, RStudio will translate between the two conventions, so you can just follow along with the macOS/UNIX convention in this workshop.

2.3.2 What kind of file are we importing?

The first step is to notice this is a `.csv` file, which stands for **c**omma-**s**eparated **v**alue. This means our data, in raw format, looks something like this:

```
# Comma-separated data

cat_acquisition_order,name,weight\n
1,Nick,9\n
2,Margot,10\n
3,Little Guy,13\n
```

Each line represents a row of data, and each field is separated by a comma (`,`). We can read this kind of data into R by using the `read_csv()` function.

```
read_csv(file = "data/clt-berry-data.csv")
```

```
## Rows: 7507 Columns: 92
## -- Column specification --------------------------------------------------------
```

```
## Delimiter: ","
## chr  (7): Start Time (UTC), End Time (UTC), Sample Name, verbal_likes, verba...
## dbl (83): Subject Code, Participant Name, Serving Position, Sample Identifie...
## lgl  (2): Gender, Age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 7,507 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 7,497 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

Suddenly, we have tabular data (i.e., data in rows and columns), like we'd have in Excel! Now we're getting somewhere. However, before we go forward we'll have to store this data somewhere–right now we're just reading it and throwing it away.

```
berry_data <- read_csv(file = "data/clt-berry-data.csv")
```

```
## Rows: 7507 Columns: 92
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (7): Start Time (UTC), End Time (UTC), Sample Name, verbal_likes, verba...
## dbl (83): Subject Code, Participant Name, Serving Position, Sample Identifie...
## lgl  (2): Gender, Age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

As a note, in many countries the separator (delimiter) will be the semi-colon (`;`), since the comma is used as the decimal marker. To read files formatted this way, you can use the `read_csv2()` function. If you encounter tab-separated values files (`.tsv`) you can use the `read_tsv()` function. If you have more non-standard delimiters, you can use the `read_delim()` function, which will allow you to specify your own delimiter characters. You can also read many other formats of tabular data using the `rio` package ("read input/output"), which can be installed from CRAN (using, as you have learned, `install.packages("rio")`).

2.4 Data in R

Let's take a look at the `Environment` tab. Among some other objects you may have created (like `x`), you should see `berry_data` listed. This is a type of data called a `data.frame` in `R`, and it is going to be, for the most part, the kind of data you interact with most. Let's learn about how these types of objects work by doing a quick review of the basics.

We started by creating an object called `x` and storing a number (`100`) into it. What kind of thing is this?

```
x <- 100
class(x)
```

```
## [1] "numeric"
```

```
typeof(x)
```

```
## [1] "double"
```

`R` has a bunch of basic data types, including the above "numeric" data type, which is a "real number" (in computer terms, a floating-point double as opposed to an integer). It can also store logical values (`TRUE` and `FALSE`), integers, characters/strings (which are what we're really here to deal with) and some more exotic data types you won't encounter very much. What `R` does that makes it good for data analysis is that it stores these all as **vectors**: 1-dimensional arrays of the same type of data. So, in fact, `x` is a length-1 vector of numeric data:

```
length(x)
```

```
## [1] 1
```

The operator to explicitly make a vector in R is the `c()` function, which stands for "combine". So if we want to make a vector of a few values, we use this function as so:

```
y <- c(1, 2, 3, 10, 50)
y
```

```
## [1]  1  2  3 10 50
```

We can also use `c()` to combine pre-existing objects:

```
c(x, y)
```

```
## [1] 100   1   2   3  10  50
```

You can have vectors of other types of objects:

```
animals <- c("fox", "bat", "rat", "cat")
class(animals)
```

```
## [1] "character"
```

If we try to combine vectors of 2 types of data, R will "coerce" the data types to match to the less restrictive type, in the following order: `logical > integer > numeric > character`. So if we combine `y` and `animals`, we'll turn the numbers into their character representations. I mention this because it can be a source of error and confusion when we are working with large datasets, as we may see.

```
c(y, animals)
```

```
## [1] "1"   "2"   "3"   "10"  "50"  "fox" "bat" "rat" "cat"
```

For example, we can divide all the numbers in `y` by `2`, but if we try to divide `c(y, animals)` by `2` we will get an error:

```
c(y, animals) / 2
```

```
## Error in c(y, animals)/2: non-numeric argument to binary operator
```

For vectors (and more complex objects), we can use the `str()` ("structure") function to get some details about their nature and what they contain:

```
str(y)
```

```
##  num [1:5] 1 2 3 10 50
```

```
str(animals)
```

```
##  chr [1:4] "fox" "bat" "rat" "cat"
```

This `str()` function is especially useful when we have big, complicated datasets, like `berry_data`:

```
str(berry_data)
```

```
## spc_tbl_ [7,507 x 92] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ Subject Code                : num [1:7507] 1001 1001 1001 1001 1001 ...
##  $ Participant Name            : num [1:7507] 1001 1001 1001 1001 1001 ...
##  $ Gender                      : logi [1:7507] NA NA NA NA NA NA ...
##  $ Age                         : logi [1:7507] NA NA NA NA NA NA ...
##  $ Start Time (UTC)            : chr [1:7507] "6/13/2019 21:05" "6/13/2019 20:55" "6/
##  $ End Time (UTC)              : chr [1:7507] "6/13/2019 21:09" "6/13/2019 20:59" "6/
##  $ Serving Position            : num [1:7507] 5 3 2 1 4 6 3 1 4 2 ...
##  $ Sample Identifier           : num [1:7507] 1426 3167 4624 5068 7195 ...
##  $ Sample Name                 : chr [1:7507] "raspberry 6" "raspberry 5" "raspberry
##  $ 9pt_appearance              : num [1:7507] 4 8 4 7 7 7 6 8 8 7 ...
##  $ pre_expectation             : num [1:7507] 2 4 2 4 3 4 2 3 5 3 ...
##  $ jar_color                   : num [1:7507] 2 3 2 2 4 4 2 3 3 2 ...
##  $ jar_gloss                   : num [1:7507] 4 3 2 3 3 3 4 3 4 4 ...
##  $ jar_size                    : num [1:7507] 2 3 3 4 3 3 4 3 5 3 ...
##  $ cata_appearance_unevencolor: num [1:7507] 0 0 0 0 1 0 0 1 1 1 ...
##  $ cata_appearance_misshapen  : num [1:7507] 1 0 0 0 1 0 0 0 0 0 ...
##  $ cata_appearance_creased    : num [1:7507] 0 0 0 0 0 0 0 0 1 1 ...
##  $ cata_appearance_seedy      : num [1:7507] 0 0 0 0 0 0 0 0 0 0 ...
##  $ cata_appearance_bruised    : num [1:7507] 0 0 0 0 0 0 0 0 1 1 ...
##  $ cata_appearance_notfresh   : num [1:7507] 1 0 1 0 0 0 0 0 0 0 ...
##  $ cata_appearance_fresh      : num [1:7507] 0 1 0 1 0 1 1 1 1 1 ...
##  $ cata_appearance_goodshape  : num [1:7507] 0 1 0 1 0 1 1 1 0 0 ...
##  $ cata_appearance_goodquality: num [1:7507] 0 1 0 1 0 1 1 1 1 0 ...
##  $ cata_appearance_none        : num [1:7507] 0 0 0 0 0 0 0 0 0 0 ...
##  $ 9pt_overall                 : num [1:7507] 4 9 3 7 4 4 4 7 7 9 ...
##  $ verbal_likes                : chr [1:7507] "Out of the two, there was one that hac
##  $ verbal_dislikes             : chr [1:7507] "There were different flavors coming fr
##  $ 9pt_taste                   : num [1:7507] 4 9 3 6 3 3 4 4 6 9 ...
##  $ grid_sweetness              : num [1:7507] 3 6 3 6 2 3 3 2 2 6 ...
```

```
##  $ grid_tartness              : num [1:7507] 6 5 5 3 5 6 5 5 5 2 ...
##  $ grid_raspberryflavor       : num [1:7507] 4 7 2 6 2 3 2 6 2 7 ...
##  $ jar_sweetness              : num [1:7507] 2 3 2 3 2 1 1 1 1 3 ...
##  $ jar_tartness               : num [1:7507] 4 3 3 3 4 5 4 4 4 3 ...
##  $ cata_taste_floral          : num [1:7507] 0 0 0 1 0 0 0 1 1 1 ...
##  $ cata_taste_berry           : num [1:7507] 1 1 0 1 0 0 0 1 0 1 ...
##  $ cata_taste_green           : num [1:7507] 0 0 0 1 1 1 0 0 1 0 ...
##  $ cata_taste_grassy          : num [1:7507] 0 0 0 0 1 1 1 0 1 0 ...
##  $ cata_taste_fermented       : num [1:7507] 0 0 1 0 0 0 0 0 0 0 ...
##  $ cata_taste_tropical        : num [1:7507] 1 1 0 0 0 0 0 0 0 1 ...
##  $ cata_taste_fruity          : num [1:7507] 1 1 0 1 0 0 0 0 0 1 ...
##  $ cata_taste_citrus          : num [1:7507] 1 0 0 0 0 1 1 0 0 0 ...
##  $ cata_taste_earthy          : num [1:7507] 0 0 0 0 1 0 0 1 0 0 ...
##  $ cata_taste_candy           : num [1:7507] 0 0 1 0 0 0 0 0 0 0 ...
##  $ cata_taste_none            : num [1:7507] 0 0 0 0 0 0 0 0 0 0 ...
##  $ 9pt_texture                : num [1:7507] 6 8 2 8 5 6 6 9 8 7 ...
##  $ grid_seediness             : num [1:7507] 3 5 6 3 5 5 6 4 6 5 ...
##  $ grid_firmness              : num [1:7507] 5 5 5 2 6 5 5 6 5 3 ...
##  $ grid_juiciness             : num [1:7507] 2 5 2 2 2 4 2 4 2 3 ...
##  $ jar_firmness               : num [1:7507] 3 3 4 2 4 3 3 3 3 2 ...
##  $ jar_juciness               : num [1:7507] 2 3 1 2 2 2 1 2 1 3 ...
##  $ post_expectation           : num [1:7507] 1 5 2 4 2 2 2 2 2 5 ...
##  $ price                      : num [1:7507] 1.99 4.99 2.99 4.99 2.99 3.99 3.99 3.99 2.99 4.99
##  $ product_tier               : num [1:7507] 1 3 2 3 1 2 2 2 1 3 ...
##  $ purchase_intent            : num [1:7507] 1 5 2 4 2 2 3 4 2 5 ...
##  $ subject                    : num [1:7507] 1031946 1031946 1031946 1031946 1031946 ...
##  $ test_day                   : chr [1:7507] "Raspberry Day 1" "Raspberry Day 1" "Raspberry Da
##  $ us_appearance              : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ us_overall                 : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ us_taste                   : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ us_texture                 : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ lms_appearance             : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ lms_overall                : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ lms_taste                  : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ lms_texture                : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_appearane_bruised     : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_appearance_goodshapre : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_appearance_goodcolor  : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ grid_blackberryflavor      : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_cinnamon        : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_lemon           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_clove           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_minty           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_grape           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ grid_crispness             : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ jar_crispness              : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
```

```
##  $ jar_juiciness              : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_appearane_creased     : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ grid_blueberryflavor       : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_piney           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_peachy          : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ 9pt_aroma                  : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ grid_strawberryflavor      : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_caramel         : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_grapey          : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_melon           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ cata_taste_cherry          : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ grid_crunchiness           : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ jar_crunch                 : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ us_aroma                   : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ lms_aroma                  : num [1:7507] NA NA NA NA NA NA NA NA NA NA ...
##  $ berry                      : chr [1:7507] "raspberry" "raspberry" "raspberry" "ra
##  $ sample                     : num [1:7507] 6 5 2 3 4 1 6 5 2 3 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   `Subject Code` = col_double(),
##   ..   `Participant Name` = col_double(),
##   ..   Gender = col_logical(),
##   ..   Age = col_logical(),
##   ..   `Start Time (UTC)` = col_character(),
##   ..   `End Time (UTC)` = col_character(),
##   ..   `Serving Position` = col_double(),
##   ..   `Sample Identifier` = col_double(),
##   ..   `Sample Name` = col_character(),
##   ..   `9pt_appearance` = col_double(),
##   ..   pre_expectation = col_double(),
##   ..   jar_color = col_double(),
##   ..   jar_gloss = col_double(),
##   ..   jar_size = col_double(),
##   ..   cata_appearance_unevencolor = col_double(),
##   ..   cata_appearance_misshapen = col_double(),
##   ..   cata_appearance_creased = col_double(),
##   ..   cata_appearance_seedy = col_double(),
##   ..   cata_appearance_bruised = col_double(),
##   ..   cata_appearance_notfresh = col_double(),
##   ..   cata_appearance_fresh = col_double(),
##   ..   cata_appearance_goodshape = col_double(),
##   ..   cata_appearance_goodquality = col_double(),
##   ..   cata_appearance_none = col_double(),
##   ..   `9pt_overall` = col_double(),
##   ..   verbal_likes = col_character(),
##   ..   verbal_dislikes = col_character(),
```

```
##   ..   `9pt_taste` = col_double(),
##   ..   grid_sweetness = col_double(),
##   ..   grid_tartness = col_double(),
##   ..   grid_raspberryflavor = col_double(),
##   ..   jar_sweetness = col_double(),
##   ..   jar_tartness = col_double(),
##   ..   cata_taste_floral = col_double(),
##   ..   cata_taste_berry = col_double(),
##   ..   cata_taste_green = col_double(),
##   ..   cata_taste_grassy = col_double(),
##   ..   cata_taste_fermented = col_double(),
##   ..   cata_taste_tropical = col_double(),
##   ..   cata_taste_fruity = col_double(),
##   ..   cata_taste_citrus = col_double(),
##   ..   cata_taste_earthy = col_double(),
##   ..   cata_taste_candy = col_double(),
##   ..   cata_taste_none = col_double(),
##   ..   `9pt_texture` = col_double(),
##   ..   grid_seediness = col_double(),
##   ..   grid_firmness = col_double(),
##   ..   grid_juiciness = col_double(),
##   ..   jar_firmness = col_double(),
##   ..   jar_juciness = col_double(),
##   ..   post_expectation = col_double(),
##   ..   price = col_double(),
##   ..   product_tier = col_double(),
##   ..   purchase_intent = col_double(),
##   ..   subject = col_double(),
##   ..   test_day = col_character(),
##   ..   us_appearance = col_double(),
##   ..   us_overall = col_double(),
##   ..   us_taste = col_double(),
##   ..   us_texture = col_double(),
##   ..   lms_appearance = col_double(),
##   ..   lms_overall = col_double(),
##   ..   lms_taste = col_double(),
##   ..   lms_texture = col_double(),
##   ..   cata_appearane_bruised = col_double(),
##   ..   cata_appearance_goodshapre = col_double(),
##   ..   cata_appearance_goodcolor = col_double(),
##   ..   grid_blackberryflavor = col_double(),
##   ..   cata_taste_cinnamon = col_double(),
##   ..   cata_taste_lemon = col_double(),
##   ..   cata_taste_clove = col_double(),
##   ..   cata_taste_minty = col_double(),
##   ..   cata_taste_grape = col_double(),
```

```
##   ..   grid_crispness = col_double(),
##   ..   jar_crispness = col_double(),
##   ..   jar_juiciness = col_double(),
##   ..   cata_appearane_creased = col_double(),
##   ..   grid_blueberryflavor = col_double(),
##   ..   cata_taste_piney = col_double(),
##   ..   cata_taste_peachy = col_double(),
##   ..   `9pt_aroma` = col_double(),
##   ..   grid_strawberryflavor = col_double(),
##   ..   cata_taste_caramel = col_double(),
##   ..   cata_taste_grapey = col_double(),
##   ..   cata_taste_melon = col_double(),
##   ..   cata_taste_cherry = col_double(),
##   ..   grid_crunchiness = col_double(),
##   ..   jar_crunch = col_double(),
##   ..   us_aroma = col_double(),
##   ..   lms_aroma = col_double(),
##   ..   berry = col_character(),
##   ..   sample = col_double()
##   .. )
##  - attr(*, "problems")=<externalptr>
```

2.4.1 Subsetting data

Vectors, by nature, are ordered arrays of data (in this case, 1-dimensional arrays). That means they have a first element, a second element, and so on. Our `y` vector has 5 total elements, and our `animals` vector has 4 elements. In `R`, the way to **subset** vectors is to use the `[]` (square brackets) operator. For a 1-dimensional vector, we use this to select one or more elements:

```
y[1]
```

```
## [1] 1
```

```
animals[4]
```

```
## [1] "cat"
```

We can also select multiple elements by using a vector of indices

```
animals[c(1, 2)]
```

```
## [1] "fox" "bat"
```

A shortcut for a sequence of numbers in R is the : (colon) operator, so this is often used for indexing:

```
1:3
```

```
## [1] 1 2 3
```

```
animals[1:3]
```

```
## [1] "fox" "bat" "rat"
```

We often want to use programmatic (or "conditional") logic to subset vectors and more complex datasets. For example, we might want to only select elements of y that are less than 10. To do that, we can use one of R's conditional logic operators: <, >, <=, >=, ==, or !=. These, in order, stand for "less than,"greater than,",","less than or equal to," "greater than or equal to," "equal to," and "not equal to."

```
y < 10
```

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

We can then use this same set of logical values to select only the elements of y for which the condition is TRUE:

```
y[y < 10]
```

```
## [1] 1 2 3
```

This is useful if we have a long vector (frequently) and do not want to list or are not able to list all of the actual indices that we want to select.

2.4.2 Complex vectors/lists: data.frame and tibble

Now that we have the basics of vectors, we can move on to the complex data object we're really interested in: berry_data. This is a type of object called a tibble, which is a cute/fancy version of the more basic R object called a data.frame. These are R's version of the .csv file or your typical Excel file: a rectangular matrix of data, with (usually) columns representing some variable and rows representing some kind of observation. Each row will have a value in each column or will be NA, which is R's specific value to represent **missing data**.

In a `data.frame`, every column has to have only a single data type: a column might be `logical` or `integer` or `character`, but it cannot be a mix. However, each column can be a different type. For example, the first column in our `beer_data`, called `reviews`, is a `character` vector, but the third column, `beer_id`, is a `numeric` column.

We have now moved from 1-dimensional vectors to 2-dimensional data tables, which means we're going to have some new properties to investigate. First off, we might want to know how many rows and columns our data table has:

```
nrow(berry_data)
```

```
## [1] 7507
```

```
ncol(berry_data)
```

```
## [1] 92
```

```
length(berry_data) # Note that length() of a data.frame is the same as ncol()
```

```
## [1] 92
```

We already tried running `str(berry_data)`, which gives us the data types of each column and an example. Some other ways to examine the data include the following:

```
berry_data # simply printing the object
head(berry_data) # show the first few rows
tail(berry_data) # show the last few rows
glimpse(berry_data) # a more compact version of str()
names(berry_data) # get the variable/column names
```

Some of these functions (for example, `glimpse()`) come from the `tidyverse` package, so if you are having trouble running a command, first make sure you have run `library(tidyverse)`.

2.5 Subsetting and wrangling data tables

Since we now have 2 dimensions, our old strategy of using a single number to select a value from a vector won't work! But the `[]` operator still works on data frames and tibbles. We just have to specify coordinates, as `[<row>, <column>]`.

```
berry_data[3, 9] # get the 3rd row, 9th column value
```

```
## # A tibble: 1 x 1
##   `Sample Name`
##   <chr>
## 1 raspberry 2
```

We can continue to use ranges or vectors of indices to select larger parts of the table

```
berry_data[1:6, 9] # get the first 5 rows of the 9th column value
```

```
## # A tibble: 6 x 1
##   `Sample Name`
##   <chr>
## 1 raspberry 6
## 2 raspberry 5
## 3 raspberry 2
## 4 raspberry 3
## 5 raspberry 4
## 6 raspberry 1
```

If we only want to subset on a specific dimension and get everything from the other dimension, we just leave it blank.

```
berry_data[, 1] # get all rows of the 1st column
```

```
## # A tibble: 7,507 x 1
##    `Subject Code`
##             <dbl>
##  1           1001
##  2           1001
##  3           1001
##  4           1001
##  5           1001
##  6           1001
##  7           1002
##  8           1002
##  9           1002
## 10           1002
## # i 7,497 more rows
```

We can also use logical subsetting, just like in vectors. This is very powerful but a bit complicated, so we are going to introduce some `tidyverse` based operators to do this that will make it a lot easier. I will just give an example:

```
berry_data[berry_data$berry == "raspberry", ] # get all raspberry data
```

```
## # A tibble: 2,148 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 2,138 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

In this last example I also introduced the final bit of `tibble` and `data.frame` wrangling we will cover here: the `$` operator. This is the operator to select a single column from a `data.frame` or `tibble`. It gives you back the vector that makes up that column:

```
berry_data$lms_overall # not printed because it is too long!
berry_data$`9pt_overall` # column names starting with numbers or containing special ch
```

One of the nice things about RStudio is that it provides tab-completion for `$`. Go to the console, type "ber" and hit `tab` (or just wait a second or two, if you use the default RStudio settings). You'll see a list of possible matches, with `berry_data` at the top. Hit enter, and this will fill out the typing for you! Now, type "$" and hit `tab` again. You'll see a list of the columns in `berry_data`! This can save a huge amount of typing and memorizing the names of variables and objects.

Now that we've gone over the basics of creating and manipulating objects in `R`, we're going to run through the basics of data manipulation and visualiza-

tion with the `tidyverse`. Before we move on to that topic, let's address **any questions**.

2.6 PSA: not-knowing is normal!

Above, I mentioned "help files". How do we get help when we (inevitably) run into problems in R? There are a couple steps you will find helpful in the future:

1. Look up the help file for whatever you're doing. Do this by using the syntax `?<search item>` (for example `?c` gets help on the vector command) as a shortcut on the console.
2. Search the help files for a term you think is related. Can't remember the command for making a sequence of integers? Go to the "Help" pane in RStudio and search in the search box for "sequence". See if some of the top results get you what you need.
3. The internet. Seriously. I am not kidding even a little bit. R has one of the most active and (surprisingly) helpful user communities I've ever encountered. Try going to google and searching for "How do I make a sequence of numbers in R?" You will find quite a bit of useful help. I find the following sites particularly helpful
    1. Stack Overflow
    2. Cross Validated/Stack Exchange
    3. Seriously, Google will get you most of the way to helpful answers for many basic R questions.

We may come back to this, but I want to emphasize that **looking up help is normal**. I do it all the time. Learning to ask questions in helpful ways, how to quickly parse the information you find, and how to slightly alter the answers to suit your particular situation are key skills.

Chapter 3

Wrangling data with `tidyverse`: Rows, Columns, and Groups

A common saying in data science is that about 90% of the effort in an analysis workflow is in getting data wrangled into the right format and shape, and 10% is actual analysis. In a point and click program like SPSS or XLSTAT we don't think about this as much because the activity of reshaping the data–making it longer or wider as required, finding and cleaning missing values, selecting columns or rows, etc–is often temporally and programmatically separated from the "actual" analysis.

In `R`, this can feel a bit different because we are using the same interface to manipulate our data and to analyze it. Sometimes we'll want to jump back out to a spreadsheet program like Excel or even the command line (the "shell" like `bash` or `zsh`) to make some changes. But in general the tools for manipulating data in `R` are both more powerful and more easily used than doing these activities by hand, and you will make yourself a much more effective analyst by mastering these basic tools.

Here, we are going to emphasize the set of tools from the `tidyverse`, which are extensively documented in Hadley Wickham and Garrett Grolemund's book *R for Data Science*. If you want to learn more, start there!

Before we move on to actually learning the tools, let's make sure we've got our data loaded up.

```
library(tidyverse)
berry_data <- read_csv("data/clt-berry-data.csv")
```

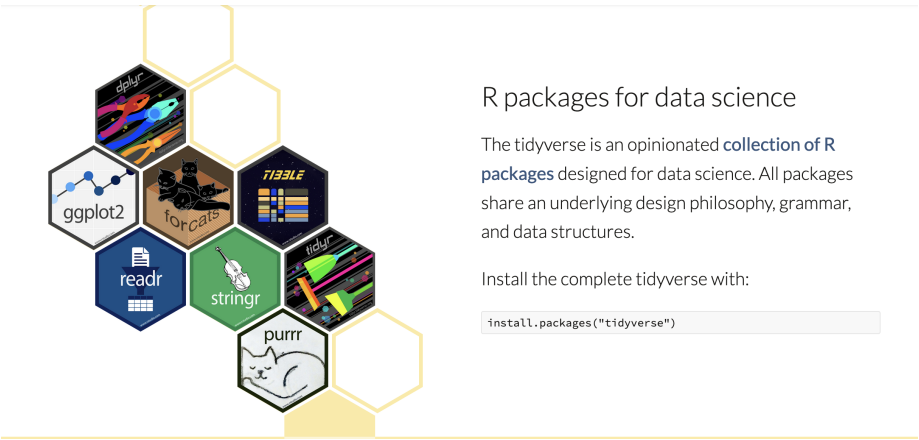


Figure 3.1: The `tidyverse` is associated with this hexagonal iconography.

```
## Rows: 7507 Columns: 92
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (7): Start Time (UTC), End Time (UTC), Sample Name, verbal_likes, verba...
## dbl (83): Subject Code, Participant Name, Serving Position, Sample Identifie...
## lgl  (2): Gender, Age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

3.1 Subsetting data

R's system for **indexing** data frames is clear and sensible for those who are used to programming languages, but not necessarily easy to read.

A common situation in R is wanting to select some rows and some columns of our data–this is called "**subsetting**" our data. But this is less easy than it might be for the beginner in R. Happily, the `tidverse` methods are much easier to read (and modeled after syntax from **SQL**, which may be helpful for some users). Starting with...

3.1.1 `select()` for columns

The first thing we often want to do in a data analysis is to pick a subset of columns, which usually represent variables. If we take a look at our berry data, we see that, for example, we have some columns that contain the answers to

survey questions, some columns that are about the test day or panelist, and some that identify the panelist or berry:

```
glimpse(berry_data)
```

```
## Rows: 7,507
## Columns: 92
## $ `Subject Code`              <dbl> 1001, 1001, 1001, 1001, 1001, 1001, 1002, ~
## $ `Participant Name`          <dbl> 1001, 1001, 1001, 1001, 1001, 1001, 1002, ~
## $ Gender                      <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ Age                         <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ `Start Time (UTC)`          <chr> "6/13/2019 21:05", "6/13/2019 20:55", "6/1~
## $ `End Time (UTC)`            <chr> "6/13/2019 21:09", "6/13/2019 20:59", "6/1~
## $ `Serving Position`          <dbl> 5, 3, 2, 1, 4, 6, 3, 1, 4, 2, 6, 5, 2, 4, ~
## $ `Sample Identifier`         <dbl> 1426, 3167, 4624, 5068, 7195, 9161, 1426, ~
## $ `Sample Name`               <chr> "raspberry 6", "raspberry 5", "raspberry 2~
## $ `9pt_appearance`            <dbl> 4, 8, 4, 7, 7, 7, 6, 8, 8, 7, 9, 8, 5, 5, ~
## $ pre_expectation             <dbl> 2, 4, 2, 4, 3, 4, 2, 3, 5, 3, 4, 5, 3, 3, ~
## $ jar_color                   <dbl> 2, 3, 2, 2, 4, 4, 2, 3, 3, 2, 3, 4, 3, 3, ~
## $ jar_gloss                   <dbl> 4, 3, 2, 3, 3, 3, 4, 3, 4, 4, 2, 4, 3, 3, ~
## $ jar_size                    <dbl> 2, 3, 3, 4, 3, 3, 4, 3, 5, 3, 3, 4, 3, 3, ~
## $ cata_appearance_unevencolor <dbl> 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, ~
## $ cata_appearance_misshapen   <dbl> 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, ~
## $ cata_appearance_creased     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, ~
## $ cata_appearance_seedy       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ cata_appearance_bruised     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, ~
## $ cata_appearance_notfresh    <dbl> 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ~
## $ cata_appearance_fresh       <dbl> 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, ~
## $ cata_appearance_goodshape   <dbl> 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, ~
## $ cata_appearance_goodquality <dbl> 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, ~
## $ cata_appearance_none        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ `9pt_overall`               <dbl> 4, 9, 3, 7, 4, 4, 4, 7, 7, 9, 7, 2, 8, 7, ~
## $ verbal_likes                <chr> "Out of the two, there was one that had a ~
## $ verbal_dislikes             <chr> "There were different flavors coming from ~
## $ `9pt_taste`                 <dbl> 4, 9, 3, 6, 3, 3, 4, 4, 6, 9, 6, 2, 8, 7, ~
## $ grid_sweetness              <dbl> 3, 6, 3, 6, 2, 3, 3, 2, 2, 6, 4, 1, 6, 4, ~
## $ grid_tartness               <dbl> 6, 5, 5, 3, 5, 6, 5, 5, 5, 2, 2, 7, 4, 5, ~
## $ grid_raspberryflavor        <dbl> 4, 7, 2, 6, 2, 3, 2, 6, 2, 7, 2, 2, 6, 5, ~
## $ jar_sweetness               <dbl> 2, 3, 2, 3, 2, 1, 1, 1, 1, 3, 2, 1, 3, 3, ~
## $ jar_tartness                <dbl> 4, 3, 3, 3, 4, 5, 4, 4, 4, 3, 4, 5, 3, 3, ~
## $ cata_taste_floral           <dbl> 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, ~
## $ cata_taste_berry            <dbl> 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, ~
## $ cata_taste_green            <dbl> 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, ~
## $ cata_taste_grassy           <dbl> 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, ~
## $ cata_taste_fermented        <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
```

```
## $ cata_taste_tropical         <dbl> 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, ~
## $ cata_taste_fruity           <dbl> 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, ~
## $ cata_taste_citrus           <dbl> 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, ~
## $ cata_taste_earthy           <dbl> 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, ~
## $ cata_taste_candy            <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ~
## $ cata_taste_none             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ `9pt_texture`               <dbl> 6, 8, 2, 8, 5, 6, 6, 9, 8, 7, 7, 7, 8, 7, ~
## $ grid_seediness              <dbl> 3, 5, 6, 3, 5, 5, 6, 4, 6, 5, 6, 5, 4, 4, ~
## $ grid_firmness               <dbl> 5, 5, 5, 2, 6, 5, 5, 6, 5, 3, 5, 5, 4, 5, ~
## $ grid_juiciness              <dbl> 2, 5, 2, 2, 2, 4, 2, 4, 2, 3, 3, 2, 6, 5, ~
## $ jar_firmness                <dbl> 3, 3, 4, 2, 4, 3, 3, 3, 3, 2, 3, 3, 3, 3, ~
## $ jar_juciness                <dbl> 2, 3, 1, 2, 2, 2, 1, 2, 1, 3, 2, 1, 3, 3, ~
## $ post_expectation            <dbl> 1, 5, 2, 4, 2, 2, 2, 2, 2, 5, 2, 1, 4, 3, ~
## $ price                       <dbl> 1.99, 4.99, 2.99, 4.99, 2.99, 3.99, 3.99, ~
## $ product_tier                <dbl> 1, 3, 2, 3, 1, 2, 2, 2, 1, 3, 2, 1, 2, 2, ~
## $ purchase_intent             <dbl> 1, 5, 2, 4, 2, 2, 3, 4, 2, 5, 3, 1, 5, 5, ~
## $ subject                     <dbl> 1031946, 1031946, 1031946, 1031946, 103194~
## $ test_day                    <chr> "Raspberry Day 1", "Raspberry Day 1", "Ras~
## $ us_appearance               <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ us_overall                  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ us_taste                    <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ us_texture                  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ lms_appearance              <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ lms_overall                 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ lms_taste                   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ lms_texture                 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_appearane_bruised      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_appearance_goodshapre  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_appearance_goodcolor   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ grid_blackberryflavor       <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_cinnamon         <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_lemon            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_clove            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_minty            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_grape            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ grid_crispness              <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ jar_crispness               <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ jar_juiciness               <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_appearane_creased      <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ grid_blueberryflavor        <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_piney            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_peachy           <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ `9pt_aroma`                 <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ grid_strawberryflavor       <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_caramel          <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_grapey           <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
```

```
## $ cata_taste_melon            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ cata_taste_cherry           <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ grid_crunchiness            <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ jar_crunch                  <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ us_aroma                    <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ lms_aroma                   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ berry                       <chr> "raspberry", "raspberry", "raspberry", "ra~
## $ sample                      <dbl> 6, 5, 2, 3, 4, 1, 6, 5, 2, 3, 4, 1, 6, 5, ~
```

So, for example, we might want to determine whether there are differences in liking between the berries, test days, or scales, in which case perhaps we only want the columns starting with `us_`, `lms_`, and `9pt_` along with `berry` and `test_day`. We learned previously that we can do this with numeric indexing:

```
berry_data[, c(10,25,28,45,56:64,81,89:91)]
```

```
## # A tibble: 7,507 x 17
##    `9pt_appearance` `9pt_overall` `9pt_taste` `9pt_texture` test_day
##               <dbl>         <dbl>       <dbl>         <dbl> <chr>
##  1                4             4           4             6 Raspberry Day 1
##  2                8             9           9             8 Raspberry Day 1
##  3                4             3           3             2 Raspberry Day 1
##  4                7             7           6             8 Raspberry Day 1
##  5                7             4           3             5 Raspberry Day 1
##  6                7             4           3             6 Raspberry Day 1
##  7                6             4           4             6 Raspberry Day 1
##  8                8             7           4             9 Raspberry Day 1
##  9                8             7           6             8 Raspberry Day 1
## 10                7             9           9             7 Raspberry Day 1
## # i 7,497 more rows
## # i 12 more variables: us_appearance <dbl>, us_overall <dbl>, us_taste <dbl>,
## #   us_texture <dbl>, lms_appearance <dbl>, lms_overall <dbl>, lms_taste <dbl>,
## #   lms_texture <dbl>, `9pt_aroma` <dbl>, us_aroma <dbl>, lms_aroma <dbl>,
## #   berry <chr>
```

However, this is both difficult for novices to `R` and difficult to read if you are not intimately familiar with the data. It is also rather fragile–what if someone else rearranged the data in your import file? You're just selecting columns by their order, so column 91 is not guaranteed to contain the berry type every time. Not to mention all of the counting was annoying!

Subsetting using names is a little more stable, but still not that readable and takes a lot of typing.

```
berry_data[, c("9pt_aroma","9pt_overall","9pt_taste","9pt_texture","9pt_appearance",
               "lms_aroma","lms_overall","lms_taste","lms_texture","lms_appearance",
               "us_aroma","us_overall","us_taste","us_texture","us_appearance",
               "berry","test_day")]
```

```
## # A tibble: 7,507 x 17
##    `9pt_aroma` `9pt_overall` `9pt_taste` `9pt_texture` `9pt_appearance`
##          <dbl>         <dbl>       <dbl>         <dbl>             <dbl>
##  1          NA             4           4             6                 4
##  2          NA             9           9             8                 8
##  3          NA             3           3             2                 4
##  4          NA             7           6             8                 7
##  5          NA             4           3             5                 7
##  6          NA             4           3             6                 7
##  7          NA             4           4             6                 6
##  8          NA             7           4             9                 8
##  9          NA             7           6             8                 8
## 10          NA             9           9             7                 7
## # i 7,497 more rows
## # i 12 more variables: lms_aroma <dbl>, lms_overall <dbl>, lms_taste <dbl>,
## #   lms_texture <dbl>, lms_appearance <dbl>, us_aroma <dbl>, us_overall <dbl>,
## #   us_taste <dbl>, us_texture <dbl>, us_appearance <dbl>, berry <chr>,
## #   test_day <chr>
```

The `select()` function in `tidyverse` (actually from the `dplyr` package) is the smarter, easier way to do this. It works on data frames, and it can be read as "`select()` the columns from <data frame> that meet the criteria we've set."

```
select(<data frame>, <column 1>, <column 2>, ...)
```

The simplest way to use `select()` is just to name the columns you want!

```
select(berry_data, berry, test_day,
       lms_aroma, lms_overall, lms_taste, lms_texture, lms_appearance) # note the lack
```

```
## # A tibble: 7,507 x 7
##    berry     test_day lms_aroma lms_overall lms_taste lms_texture lms_appearance
##    <chr>     <chr>        <dbl>       <dbl>     <dbl>       <dbl>          <dbl>
##  1 raspberry Raspber~        NA          NA        NA          NA             NA
##  2 raspberry Raspber~        NA          NA        NA          NA             NA
##  3 raspberry Raspber~        NA          NA        NA          NA             NA
##  4 raspberry Raspber~        NA          NA        NA          NA             NA
##  5 raspberry Raspber~        NA          NA        NA          NA             NA
##  6 raspberry Raspber~        NA          NA        NA          NA             NA
##  7 raspberry Raspber~        NA          NA        NA          NA             NA
```

```
##  8 raspberry Raspber~        NA          NA        NA          NA             NA
##  9 raspberry Raspber~        NA          NA        NA          NA             NA
## 10 raspberry Raspber~        NA          NA        NA          NA             NA
## # i 7,497 more rows
```

This is much clearer to the reader.

Getting rid of the double quotes we previously saw around `"column names"` can have some consequences, though. R variable names can *only* contain letters, numbers, and underscores (_), no spaces ( ) or dashes (-), and can't start with numbers, but the tidyverse *will* let you make column names that break these rules. So if you try to do the exact same thing as before to get the data from the 9-point hedonic scale instead of the labeled magnitude scale:

```
 # this will cause an error
select(berry_data, berry, test_day,
       9pt_aroma, 9pt_overall, 9pt_taste, 9pt_texture, 9pt_appearance)
```

You'll run into an error. The solution to this is a different kind of quote called a backtick (‘) which is usually next to the number 1 key in the very top left of QWERTY (US and UK) keyboards. On QWERTZ and AZERTY keyboards, it may be next to the backspace key or one of the alternate characters made by the 7 key. Look at this guide for help finding it in common international keyboard layouts.

```
select(berry_data, berry, test_day, # these "syntactic" column names don't need escaping
       `9pt_aroma`, # only ones starting with numbers...
       `9pt_overall`, `9pt_taste`, `9pt_texture`, `9pt_appearance`,
       `Sample Name`) # or containing spaces
```

```
## # A tibble: 7,507 x 8
##    berry     test_day        ‘9pt_aroma‘ ‘9pt_overall‘ ‘9pt_taste‘ ‘9pt_texture‘
##    <chr>     <chr>                 <dbl>         <dbl>       <dbl>         <dbl>
##  1 raspberry Raspberry Day 1          NA             4           4             6
##  2 raspberry Raspberry Day 1          NA             9           9             8
##  3 raspberry Raspberry Day 1          NA             3           3             2
##  4 raspberry Raspberry Day 1          NA             7           6             8
##  5 raspberry Raspberry Day 1          NA             4           3             5
##  6 raspberry Raspberry Day 1          NA             4           3             6
##  7 raspberry Raspberry Day 1          NA             4           4             6
##  8 raspberry Raspberry Day 1          NA             7           4             9
##  9 raspberry Raspberry Day 1          NA             7           6             8
## 10 raspberry Raspberry Day 1          NA             9           9             7
## # i 7,497 more rows
## # i 2 more variables: ‘9pt_appearance‘ <dbl>, ‘Sample Name‘ <chr>
```

The backticks are only necessary when a column name breaks one of the variable-naming rules, and RStudio will usually fill them in for you if you use tab auto-completion when writing your select() and other tidyverse functions.

You can also use `select()` with a number of helper functions, which use logic to select columns that meet whatever conditions you set. For example, the `starts_with()` helper function lets us give a set of characters we want columns to start with:

```
select(berry_data, starts_with("lms_")) #the double-quotes are back because this isn't
```

```
## # A tibble: 7,507 x 5
##    lms_appearance lms_overall lms_taste lms_texture lms_aroma
##             <dbl>       <dbl>     <dbl>       <dbl>     <dbl>
##  1             NA          NA        NA          NA        NA
##  2             NA          NA        NA          NA        NA
##  3             NA          NA        NA          NA        NA
##  4             NA          NA        NA          NA        NA
##  5             NA          NA        NA          NA        NA
##  6             NA          NA        NA          NA        NA
##  7             NA          NA        NA          NA        NA
##  8             NA          NA        NA          NA        NA
##  9             NA          NA        NA          NA        NA
## 10             NA          NA        NA          NA        NA
## # i 7,497 more rows
```

There are equivalents for the end of column names (`ends_with()`) and text found anywhere in the name (`contains()`).

You can combine these statements together to get subsets of columns however you want:

```
select(berry_data, starts_with("lms_"), starts_with("9pt_"), starts_with("us_"),
       berry, test_day)
```

```
## # A tibble: 7,507 x 17
##    lms_appearance lms_overall lms_taste lms_texture lms_aroma `9pt_appearance`
##             <dbl>       <dbl>     <dbl>       <dbl>     <dbl>             <dbl>
##  1             NA          NA        NA          NA        NA                 4
##  2             NA          NA        NA          NA        NA                 8
##  3             NA          NA        NA          NA        NA                 4
##  4             NA          NA        NA          NA        NA                 7
##  5             NA          NA        NA          NA        NA                 7
##  6             NA          NA        NA          NA        NA                 7
##  7             NA          NA        NA          NA        NA                 6
```

```
##  8            NA          NA        NA          NA        NA                8
##  9            NA          NA        NA          NA        NA                8
## 10            NA          NA        NA          NA        NA                7
## # i 7,497 more rows
## # i 11 more variables: `9pt_overall` <dbl>, `9pt_taste` <dbl>,
## #   `9pt_texture` <dbl>, `9pt_aroma` <dbl>, us_appearance <dbl>,
## #   us_overall <dbl>, us_taste <dbl>, us_texture <dbl>, us_aroma <dbl>,
## #   berry <chr>, test_day <chr>
```

```
select(berry_data, starts_with("jar_"), ends_with("_overall"),
       berry, test_day)
```

```
## # A tibble: 7,507 x 15
##    jar_color jar_gloss jar_size jar_sweetness jar_tartness jar_firmness
##        <dbl>     <dbl>    <dbl>         <dbl>        <dbl>        <dbl>
##  1         2         4        2             2            4            3
##  2         3         3        3             3            3            3
##  3         2         2        3             2            3            4
##  4         2         3        4             3            3            2
##  5         4         3        3             2            4            4
##  6         4         3        3             1            5            3
##  7         2         4        4             1            4            3
##  8         3         3        3             1            4            3
##  9         3         4        5             1            4            3
## 10         2         4        3             3            3            2
## # i 7,497 more rows
## # i 9 more variables: jar_juciness <dbl>, jar_crispness <dbl>,
## #   jar_juiciness <dbl>, jar_crunch <dbl>, `9pt_overall` <dbl>,
## #   us_overall <dbl>, lms_overall <dbl>, berry <chr>, test_day <chr>
```

If you're annoyed at the order that your columns are printing in and how far to the right you have to scroll in your previews, `select()` can also be used to rearrange your columns with a helper function called `everything()`:

```
select(berry_data, everything()) #This selects everything--it doesn't change at all.
```

```
## # A tibble: 7,507 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
```

```
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 7,497 more rows
## # i 87 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

```
select(berry_data, berry, contains("_overall"),
       everything()) #only type the columns you want to move to the front, and everyth
```

```
## # A tibble: 7,507 x 92
##    berry  '9pt_overall' us_overall lms_overall 'Subject Code' 'Participant Name'
##    <chr>          <dbl>      <dbl>       <dbl>          <dbl>              <dbl>
##  1 raspb~             4         NA          NA           1001               1001
##  2 raspb~             9         NA          NA           1001               1001
##  3 raspb~             3         NA          NA           1001               1001
##  4 raspb~             7         NA          NA           1001               1001
##  5 raspb~             4         NA          NA           1001               1001
##  6 raspb~             4         NA          NA           1001               1001
##  7 raspb~             4         NA          NA           1002               1002
##  8 raspb~             7         NA          NA           1002               1002
##  9 raspb~             7         NA          NA           1002               1002
## 10 raspb~             9         NA          NA           1002               1002
## # i 7,497 more rows
## # i 86 more variables: Gender <lgl>, Age <lgl>, 'Start Time (UTC)' <chr>,
## #   'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>, ...
```

`everything()` can be very useful for programming, but if you just need to rearrange an even easier function is `relocate()`, which moves whatever columns you specify to the "left" or "right" of the `data.frame`; you can even get more specific and move to the left or right of specific columns:

```
# By default relocate() moves the selected column(s) to the left of the data.frame
relocate(berry_data, us_overall)
```

```
## # A tibble: 7,507 x 92
##    us_overall `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##         <dbl>          <dbl>               <dbl> <lgl>  <lgl> <chr>
##  1         NA           1001                1001 NA     NA    6/13/2019 21:05
##  2         NA           1001                1001 NA     NA    6/13/2019 20:55
##  3         NA           1001                1001 NA     NA    6/13/2019 20:49
##  4         NA           1001                1001 NA     NA    6/13/2019 20:45
##  5         NA           1001                1001 NA     NA    6/13/2019 21:00
##  6         NA           1001                1001 NA     NA    6/13/2019 21:10
##  7         NA           1002                1002 NA     NA    6/13/2019 20:08
##  8         NA           1002                1002 NA     NA    6/13/2019 19:57
##  9         NA           1002                1002 NA     NA    6/13/2019 20:13
## 10         NA           1002                1002 NA     NA    6/13/2019 20:03
## # i 7,497 more rows
## # i 86 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

```
# ".before/.after = " specify a position relative to another column
relocate(berry_data, `Subject Code`, .before = Gender)
```

```
## # A tibble: 7,507 x 92
##    `Participant Name` `Subject Code` Gender Age   `Start Time (UTC)`
##                 <dbl>          <dbl> <lgl>  <lgl> <chr>
##  1               1001           1001 NA     NA    6/13/2019 21:05
##  2               1001           1001 NA     NA    6/13/2019 20:55
##  3               1001           1001 NA     NA    6/13/2019 20:49
##  4               1001           1001 NA     NA    6/13/2019 20:45
##  5               1001           1001 NA     NA    6/13/2019 21:00
##  6               1001           1001 NA     NA    6/13/2019 21:10
##  7               1002           1002 NA     NA    6/13/2019 20:08
##  8               1002           1002 NA     NA    6/13/2019 19:57
##  9               1002           1002 NA     NA    6/13/2019 20:13
## 10               1002           1002 NA     NA    6/13/2019 20:03
## # i 7,497 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

You may sometimes want to select columns where the data meets some criteria,

rather than the column name or position. The `where()` helper function allows you to insert this kind of programmatic logic into `select()` statements, which gives you the ability to specify columns using any arbitrary function.

```
 #a few functions return one logical value per vector
select(berry_data, where(is.numeric))
```

```
## # A tibble: 7,507 x 83
##    `Subject Code` `Participant Name` `Serving Position` `Sample Identifier`
##             <dbl>              <dbl>              <dbl>                <dbl>
##  1           1001               1001                  5                 1426
##  2           1001               1001                  3                 3167
##  3           1001               1001                  2                 4624
##  4           1001               1001                  1                 5068
##  5           1001               1001                  4                 7195
##  6           1001               1001                  6                 9161
##  7           1002               1002                  3                 1426
##  8           1002               1002                  1                 3167
##  9           1002               1002                  4                 4624
## 10           1002               1002                  2                 5068
## # i 7,497 more rows
## # i 79 more variables: `9pt_appearance` <dbl>, pre_expectation <dbl>,
## #   jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>, ...
```

```
 #otherwise we can use "lambda" functions
select(berry_data, where(~!any(is.na(.))))
```

```
## # A tibble: 7,507 x 38
##    `Subject Code` `Participant Name` `Start Time (UTC)` `End Time (UTC)`
##             <dbl>              <dbl> <chr>              <chr>
##  1           1001               1001 6/13/2019 21:05    6/13/2019 21:09
##  2           1001               1001 6/13/2019 20:55    6/13/2019 20:59
##  3           1001               1001 6/13/2019 20:49    6/13/2019 20:53
##  4           1001               1001 6/13/2019 20:45    6/13/2019 20:48
##  5           1001               1001 6/13/2019 21:00    6/13/2019 21:03
##  6           1001               1001 6/13/2019 21:10    6/13/2019 21:13
##  7           1002               1002 6/13/2019 20:08    6/13/2019 20:11
##  8           1002               1002 6/13/2019 19:57    6/13/2019 20:01
##  9           1002               1002 6/13/2019 20:13    6/13/2019 20:17
## 10           1002               1002 6/13/2019 20:03    6/13/2019 20:07
## # i 7,497 more rows
```

```
## # i 34 more variables: `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `Sample Name` <chr>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodquality <dbl>,
## #   cata_appearance_none <dbl>, grid_sweetness <dbl>, grid_tartness <dbl>, ...
```

The little squiggly symbol, called a tilde (~), is above the backtick on QWERTY keyboards, and it turns everything that comes after it into a "lambda function". We'll talk more about lambda functions later, when we talk about `across()`. You can read it the above as "`select()` the columns from `berry_data` `where()` there are not (`!`) `any()` NA values (`is.na(.)`)".

Besides being easier to write conditions for than indexing with `[]`, `select()` is code that is much closer to how you or I think about what we're actually doing, making code that is more human readable.

3.1.2 `filter()` for rows

So `select()` lets us pick which columns we want. Can we also use it to pick particular observations? No. For that, there's `filter()`.

We learned that, using `[]` indexing, we'd specify a set of rows we want. If we want the first 10 rows of `berry_data`, we'd write

```
berry_data[1:10, ]
```

```
## # A tibble: 10 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
```

```
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>, ...
```

Again, this is not very human readable, and if we reorganize our rows this won't be useful anymore. The `tidyverse` answer to this approach is the `filter()` function, which lets you filter your dataset into specific rows according to *data stored in the table itself.*

```
# let's get survey responses with had a higher-than-average expectation
filter(berry_data, pre_expectation > 3)
```

```
## # A tibble: 2,192 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 20:55
##  2           1001               1001 NA     NA    6/13/2019 20:45
##  3           1001               1001 NA     NA    6/13/2019 21:10
##  4           1002               1002 NA     NA    6/13/2019 20:13
##  5           1002               1002 NA     NA    6/13/2019 20:21
##  6           1002               1002 NA     NA    6/13/2019 20:18
##  7           1004               1004 NA     NA    6/13/2019 21:09
##  8           1004               1004 NA     NA    6/13/2019 20:58
##  9           1004               1004 NA     NA    6/13/2019 20:50
## 10           1005               1005 NA     NA    6/13/2019 20:13
## # i 2,182 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

When using `filter()`, we can specify multiple logical conditions. For example, let's get only raspberries with initially high expectations. If we want only exact matches, we can use the direct `==` operator:

```
filter(berry_data, pre_expectation > 3, berry == "raspberry")
```

```
## # A tibble: 709 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 20:55
##  2           1001               1001 NA     NA    6/13/2019 20:45
##  3           1001               1001 NA     NA    6/13/2019 21:10
```

```
##  4           1002               1002 NA     NA    6/13/2019 20:13
##  5           1002               1002 NA     NA    6/13/2019 20:21
##  6           1002               1002 NA     NA    6/13/2019 20:18
##  7           1004               1004 NA     NA    6/13/2019 21:09
##  8           1004               1004 NA     NA    6/13/2019 20:58
##  9           1004               1004 NA     NA    6/13/2019 20:50
## 10           1005               1005 NA     NA    6/13/2019 20:13
## # i 699 more rows
## # i 87 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

But this won't return, for example, any berries labeled as "Raspberry", with an uppercase R.

```
filter(berry_data, pre_expectation > 3, berry == "Raspberry")
```

```
## # A tibble: 0 x 92
## # i 92 variables: Subject Code <dbl>, Participant Name <dbl>, Gender <lgl>,
## #   Age <lgl>, Start Time (UTC) <chr>, End Time (UTC) <chr>,
## #   Serving Position <dbl>, Sample Identifier <dbl>, Sample Name <chr>,
## #   9pt_appearance <dbl>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_gloss <dbl>, jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>, ...
```

Luckily, we don't have a mix of raspberries and Raspberries. Maybe we want both raspberries and strawberries that panelists had high expectations of:

```
filter(berry_data, pre_expectation > 3, berry == "raspberry" | berry == "strawberry")
```

```
## # A tibble: 1,104 x 92
##    'Subject Code' 'Participant Name' Gender Age   'Start Time (UTC)'
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 20:55
##  2           1001               1001 NA     NA    6/13/2019 20:45
##  3           1001               1001 NA     NA    6/13/2019 21:10
##  4           1002               1002 NA     NA    6/13/2019 20:13
##  5           1002               1002 NA     NA    6/13/2019 20:21
##  6           1002               1002 NA     NA    6/13/2019 20:18
##  7           1004               1004 NA     NA    6/13/2019 21:09
```

```
##  8           1004               1004 NA     NA    6/13/2019 20:58
##  9           1004               1004 NA     NA    6/13/2019 20:50
## 10           1005               1005 NA     NA    6/13/2019 20:13
## # i 1,094 more rows
## # i 87 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

In `R`, the `|` means Boolean `OR`, and the `&` means Boolean `AND`. If you're trying to figure out which one to use, phrase what you want to do with your `filter()` statement by starting "keep only rows that have…". We may want to look at raspberries *and* strawberries, but we want to "keep only rows that have a `berry` type of raspberry *or* strawberry".

We can combine any number of conditions with `&` and `|` to search within our data table. But this can be a bit tedious. The `stringr` package, part of `tidyverse`, gives a lot of utility functions that we can use to reduce the number of options we're individually writing out, similar to `starts_with()` or `contains()` for columns. Maybe we want the results from the first day of each berry type:

```
filter(berry_data, str_detect(test_day, "Day 1"))
```

```
## # A tibble: 2,586 x 92
##    'Subject Code' 'Participant Name' Gender Age   'Start Time (UTC)'
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 2,576 more rows
## # i 87 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

Here, the `str_detect()` function searched for any text that **contains** "Day 1" in the `test_day` column.

3.2 Combining steps with the pipe: %>%

It isn't hard to imagine a situation in which we want to **both** `select()` some columns and `filter()` some rows. There are 3 ways we can do this, one of which is going to be the best for most situations.

Let's imagine we want to get only information about the berries, overall liking, and CATA attributes for blackberries.

First, we can nest functions:

```
select(filter(berry_data, berry == "blackberry"), `Sample Name`, contains("_overall"), contains("
```

```
## # A tibble: 1,495 x 40
##    `Sample Name` `9pt_overall` us_overall lms_overall cata_appearance_unevenco~1
##    <chr>                 <dbl>      <dbl>       <dbl>                       <dbl>
##  1 Blackberry 4              2         NA          NA                           0
##  2 Blackberry 2              5         NA          NA                           0
##  3 Blackberry 1              8         NA          NA                           0
##  4 Blackberry 3              6         NA          NA                           0
##  5 Blackberry 5              8         NA          NA                           0
##  6 Blackberry 4              6         NA          NA                           0
##  7 Blackberry 2              1         NA          NA                           0
##  8 Blackberry 1              8         NA          NA                           0
##  9 Blackberry 3              8         NA          NA                           0
## 10 Blackberry 5              8         NA          NA                           0
## # i 1,485 more rows
## # i abbreviated name: 1: cata_appearance_unevencolor
## # i 35 more variables: cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>,
## #   cata_appearance_goodquality <dbl>, cata_appearance_none <dbl>, ...
```

The problem with this approach is that we have to read it "inside out". First, `filter()` will happen and get us only berries whose `berry` is exactly "blackberry". Then `select()` will get `Sample Name` along with columns that match "*overall" or "cata* in their names. Especially as your code gets complicated, this can be very hard to read (and in the bookdown, doesn't even fit on one line!).

So we might take the second approach: creating intermediates. We might first `filter()`, store that step somewhere, then `select()`:

```
blackberries <- filter(berry_data, berry == "blackberry")
select(blackberries, `Sample Name`, contains("_overall"), contains("cata_"))
```

```
## # A tibble: 1,495 x 40
##    `Sample Name` `9pt_overall` us_overall lms_overall cata_appearance_unevenco~1
##    <chr>                 <dbl>      <dbl>       <dbl>                      <dbl>
##  1 Blackberry 4              2         NA          NA                          0
##  2 Blackberry 2              5         NA          NA                          0
##  3 Blackberry 1              8         NA          NA                          0
##  4 Blackberry 3              6         NA          NA                          0
##  5 Blackberry 5              8         NA          NA                          0
##  6 Blackberry 4              6         NA          NA                          0
##  7 Blackberry 2              1         NA          NA                          0
##  8 Blackberry 1              8         NA          NA                          0
##  9 Blackberry 3              8         NA          NA                          0
## 10 Blackberry 5              8         NA          NA                          0
## # i 1,485 more rows
## # i abbreviated name: 1: cata_appearance_unevencolor
## # i 35 more variables: cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>,
## #   cata_appearance_goodquality <dbl>, cata_appearance_none <dbl>, ...
```

But now we have this intermediate we don't really need cluttering up our `Environment` tab. This is fine for a single step, but if you have a lot of steps in your analysis this is going to get old (and confusing) fast. You'll have to remove a lot of these using the `rm()` command to keep your code clean.

warning: `rm()` will *permanently* delete whatever objects you run it on from your `Environment`, and you will only be able to restore them by rerunning the code that generated them in the first place.

```
rm(blackberries)
```

The final method, and what is becoming standard in modern `R` coding, is the **pipe**, which is written in `tidyverse` as `%>%`. This garbage-looking set of symbols is actually your best friend, you just don't know it yet. I use this tool constantly in my R programming, but I've been avoiding it up to this point because it wasn't a part of base R for the vast majority of its history.

OK, enough background, what the heck *is* a pipe? The term "pipe" comes from what it does: like a pipe, `%>%` let's whatever is on it's left side flow through to the right hand side. It is easiest to read `%>%` as "**AND THEN**".

```
berry_data %>%                              # Start with the berry_data
  filter(berry == "blackberry") %>%         # AND THEN filter to blackberries
  select(`Sample Name`,                     # AND THEN select sample name, overall liking...
         contains("_overall"),
         contains("cata_"))
```

```
## # A tibble: 1,495 x 40
##    `Sample Name` `9pt_overall` us_overall lms_overall cata_appearance_unevenco~1
##    <chr>                 <dbl>      <dbl>       <dbl>                       <dbl>
##  1 Blackberry 4              2         NA          NA                           0
##  2 Blackberry 2              5         NA          NA                           0
##  3 Blackberry 1              8         NA          NA                           0
##  4 Blackberry 3              6         NA          NA                           0
##  5 Blackberry 5              8         NA          NA                           0
##  6 Blackberry 4              6         NA          NA                           0
##  7 Blackberry 2              1         NA          NA                           0
##  8 Blackberry 1              8         NA          NA                           0
##  9 Blackberry 3              8         NA          NA                           0
## 10 Blackberry 5              8         NA          NA                           0
## # i 1,485 more rows
## # i abbreviated name: 1: cata_appearance_unevencolor
## # i 35 more variables: cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>,
## #   cata_appearance_goodquality <dbl>, cata_appearance_none <dbl>, ...
```

In this example, each place there is a %>% I've added a comment saying "AND THEN". This is because that's exactly what the pipe does: it passes whatever happened in the previous step to the next function. Specifically, %>% passes the **results** of the previous line to the **first argument** of the next line.

3.2.1 Pipes require that the lefthand side be a single functional command

This means that we can't directly do something like rewrite `sqrt(1 + 2)` with %>%:

```
1 + 2 %>% sqrt # this is instead computing 1 + sqrt(2)
```

```
## [1] 2.414214
```

Instead, if we want to pass binary operators in a pipe, we need to enclose them in `()` on the line they are in:

```
(1 + 2) %>% sqrt() # Now this computes sqrt(1 + 2) = sqrt(3)
```

```
## [1] 1.732051
```

More complex piping is possible using the curly braces ({}), which create new R environments, but this is more advanced than you will generally need to be.

3.2.2 Pipes always pass the result of the lefthand side to the *first* argument of the righthand side

This sounds like a weird logic puzzle, but it's not, as we can see if we look at some simple math. Let's define a function for use in a pipe that computes the difference between two numbers:

```
subtract <- function(a, b) a - b
subtract(5, 4)
```

```
## [1] 1
```

If we want to rewrite that as a pipe, we can write:

```
5 %>% subtract(4)
```

```
## [1] 1
```

But we can't write

```
4 %>% subtract(5) # this is actually making subtract(4, 5)
```

```
## [1] -1
```

We can explicitly force the pipe to work the way we want it to by using **. as the placeholder for the result of the lefthand side**:

```
4 %>% subtract(5, .) # now this properly computes subtract(5, 4)
```

```
## [1] 1
```

So, when you're using pipes, make sure that the output of the lefthand side *should* be going into the first argument of the righthand side–this is often but not always the case, especially with non-`tidyverse` functions.

3.2.3 Pipes are a pain to type

Typing %>% is no fun. But, happily, RStudio builds in a shortcut for you: macOS is `cmd + shift + M`, Windows is `ctrl + shift + M`.

3.3 Make new columns: `mutate()`

You hopefully are starting to be excited by the relative ease of doing some things in R with `tidyverse` that are otherwise a little bit abstruse. Here's where I think things get really, really cool. The `mutate()` function *creates a new column in the existing dataset.*

We can do this in base R by setting a new name for a column and using the assign (`<-`) operator, but this is clumsy and often requires care to maintain the proper ordering of rows. Often, we want to create a new column temporarily, or to combine several existing columns. We can do this using the `mutate()` function.

Let's say that we want to create a quick categorical variable that tells us whether a berry was rated higher after the actual tasting event than the pre-tasting expectation.

We know that we can use `filter()` to get just the berries with `post_expectation > pre_expectation`:

```
berry_data %>%
  filter(post_expectation > pre_expectation)
```

```
## # A tibble: 1,612 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 20:55
##  2           1002               1002 NA     NA    6/13/2019 20:03
##  3           1004               1004 NA     NA    6/13/2019 20:54
##  4           1005               1005 NA     NA    6/13/2019 20:08
##  5           1005               1005 NA     NA    6/13/2019 19:54
##  6           1006               1006 NA     NA    6/13/2019 18:49
##  7           1007               1007 NA     NA    6/13/2019 18:52
##  8           1009               1009 NA     NA    6/13/2019 20:39
##  9           1010               1010 NA     NA    6/13/2019 20:02
## 10           1010               1010 NA     NA    6/13/2019 20:14
## # i 1,602 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
```

```
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

But what if we want to be able to just see this?

```
berry_data %>%
  mutate(improved = post_expectation > pre_expectation) %>%
  # We'll select just a few columns to help us see the result
  select(`Participant Name`, `Sample Name`, pre_expectation, post_expectation, improved
```

```
## # A tibble: 7,507 x 5
##    `Participant Name` `Sample Name` pre_expectation post_expectation improved
##                 <dbl> <chr>                   <dbl>            <dbl> <lgl>
##  1               1001 raspberry 6                 2                1 FALSE
##  2               1001 raspberry 5                 4                5 TRUE
##  3               1001 raspberry 2                 2                2 FALSE
##  4               1001 raspberry 3                 4                4 FALSE
##  5               1001 raspberry 4                 3                2 FALSE
##  6               1001 raspberry 1                 4                2 FALSE
##  7               1002 raspberry 6                 2                2 FALSE
##  8               1002 raspberry 5                 3                2 FALSE
##  9               1002 raspberry 2                 5                2 FALSE
## 10               1002 raspberry 3                 3                5 TRUE
## # i 7,497 more rows
```

What does the above function do?

`mutate()` is a very easy way to edit your data mid-pipe. So we might want to do some calculations, create a temporary variable using `mutate()`, and then continue our pipe. **Unless we use <- to store our `mutate()`'d data, the results will be only temporary.**

We can use the same kind of functional logic we've been using in other `tidyverse` commands in `mutate()` to get real, powerful results. For example, it might be easier to interpret the individual ratings on the 9-point scale if we can compare them to the `mean()` of all berry ratings. We can do this easily using `mutate()`:

```
# Let's find out the average (mean) rating for the berries on the 9-point scale
berry_data$`9pt_overall` %>% #we need the backticks for $ subsetting whenever we'd nee
  mean(na.rm = TRUE)
```

```
## [1] 5.679346
```

```
# Now, let's create a column that tells us how far each rating is from to the average
berry_data %>%
  mutate(difference_from_average = `9pt_overall` - mean(`9pt_overall`, na.rm = TRUE)) %>%
  # Again, let's select just a few columns
  select(`Sample Name`, `9pt_overall`, difference_from_average)
```

```
## # A tibble: 7,507 x 3
##    `Sample Name` `9pt_overall` difference_from_average
##    <chr>                 <dbl>                   <dbl>
##  1 raspberry 6               4                   -1.68
##  2 raspberry 5               9                    3.32
##  3 raspberry 2               3                   -2.68
##  4 raspberry 3               7                    1.32
##  5 raspberry 4               4                   -1.68
##  6 raspberry 1               4                   -1.68
##  7 raspberry 6               4                   -1.68
##  8 raspberry 5               7                    1.32
##  9 raspberry 2               7                    1.32
## 10 raspberry 3               9                    3.32
## # i 7,497 more rows
```

3.4 Split-apply-combine analyses

Many basic data analyses can be described as *split-apply-combine*: *split* the data into groups, *apply* some analysis into groups, and then *combine* the results.

For example, in our `berry_data` we might want to split the data by each berry sample, calculate the average overall rating and standard deviation of the rating for each, and the generate a summary table telling us these results. Using the `filter()` and `select()` commands we've learned so far, you could probably cobble together this analysis without further tools.

However, `tidyverse` provides two powerful tools to do this kind of analysis:

```
1.  The `group_by()` function takes a data table and groups it by **categorical** values of any c
2.  The `summarize()` function is like `mutate()` for groups created with `group_by()`:
    1.  First, you specify 1 or more new columns you want to calculate for each group
    2.  Second, the function produces 1 value for each group for each new column
```

3.4.1 Groups of data: Split-apply

The first of these tools is `group_by()`, which you *can* use with `mutate()` if you want to use both individual observed values (one per row) and groupwise summary statistics to calculate a new column.

If we wanted to make a column that shows how each individual person's rating differs from the mean of *that specific berry sample*:

```
berry_data %>%
  group_by(`Sample Name`) %>%
  mutate(difference_from_average = `9pt_overall` - mean(`9pt_overall`, na.rm = TRUE)) %
  # Again, let's select just a few columns
  select(`Sample Name`, `9pt_overall`, difference_from_average)
```

```
## # A tibble: 7,507 x 3
## # Groups:   Sample Name [23]
##    `Sample Name` `9pt_overall` difference_from_average
##    <chr>                 <dbl>                   <dbl>
##  1 raspberry 6               4                  -2.30
##  2 raspberry 5               9                   3.04
##  3 raspberry 2               3                  -2.86
##  4 raspberry 3               7                   0.832
##  5 raspberry 4               4                  -1.82
##  6 raspberry 1               4                  -0.968
##  7 raspberry 6               4                  -2.30
##  8 raspberry 5               7                   1.04
##  9 raspberry 2               7                   1.14
## 10 raspberry 3               9                   2.83
## # i 7,497 more rows
```

You might notice that the `mutate()` call hasn't been changed at all from when we did this in the last code chunk using the global average, but that the numbers are in fact different. The tibble also now tells us that it has `Groups: Sample Name[23]`, because of our group_by() call.

It's good practice to `ungroup()` as soon as you're done calculating groupwise statistics, so you don't cause unexpected problems later in the analysis.

```
berry_data %>%
  select(`Sample Name`, `9pt_overall`) %>%
  group_by(`Sample Name`) %>%
  mutate(group_average = mean(`9pt_overall`, na.rm = TRUE),
         difference_from_average = `9pt_overall` - group_average) %>%
  ungroup() %>%
  mutate(grand_average = mean(`9pt_overall`, na.rm = TRUE))
```

```
## # A tibble: 7,507 x 5
##    `Sample Name` `9pt_overall` group_average difference_from_average
##    <chr>                 <dbl>         <dbl>                   <dbl>
##  1 raspberry 6               4          6.30                  -2.30
```

```
##  2 raspberry 5              9         5.96                3.04
##  3 raspberry 2              3         5.86               -2.86
##  4 raspberry 3              7         6.17                0.832
##  5 raspberry 4              4         5.82               -1.82
##  6 raspberry 1              4         4.97               -0.968
##  7 raspberry 6              4         6.30               -2.30
##  8 raspberry 5              7         5.96                1.04
##  9 raspberry 2              7         5.86                1.14
## 10 raspberry 3              9         6.17                2.83
## # i 7,497 more rows
## # i 1 more variable: grand_average <dbl>
```

You'll see that the `Groups` info at the top of the tibble is gone, and that thanks to `ungroup()` every single row has the same grand average.

3.4.2 Split-apply-combine in fewer steps with `summarize()`

So why would we want to use `summarize()`? Well, it's not very convenient to have repeated data if we just want the average rating of each group (or any other groupwise **summary**).

Use `mutate()` when you want to add a **new column with one number for each current row**. Use `summarize()` when you need to use data from **multiple rows to create a summary**.

To accomplish the example above, we'd do the following:

```
berry_summary <-
  berry_data %>%
  filter(!is.na(`9pt_overall`)) %>%                    # only rows with 9-point ratings
  group_by(`Sample Name`) %>%                          # we will create a group for each berry sa
  summarize(n_responses = n(),                         # n() counts number of ratings for each be
            mean_rating = mean(`9pt_overall`),         # the mean rating for each species
            sd_rating = sd(`9pt_overall`),             # the standard deviation in rating
            se_rating = sd(`9pt_overall`) / sqrt(n())) # multiple functions in 1 row

berry_summary
```

```
## # A tibble: 23 x 5
##    `Sample Name` n_responses mean_rating sd_rating se_rating
##    <chr>               <int>       <dbl>     <dbl>     <dbl>
##  1 Blackberry 1           93        5.12      2.23     0.231
##  2 Blackberry 2           93        4.66      2.24     0.232
##  3 Blackberry 3           93        5.65      2.14     0.222
##  4 Blackberry 4           93        5.82      2.10     0.217
```

```
##  5 Blackberry 5           93        5.94      1.99     0.207
##  6 Blueberry 1           105        5.75      1.91     0.186
##  7 Blueberry 2           105        5.85      1.93     0.188
##  8 Blueberry 3           105        5.61      1.92     0.188
##  9 Blueberry 4           105        5.70      2.08     0.203
## 10 Blueberry 5           105        5.38      2.17     0.212
## # i 13 more rows
```

We can use this approach to even get a summary stats table - for example, confidence limits according to the normal distribution:

```
berry_summary %>%
  mutate(lower_limit = mean_rating - 1.96 * se_rating,
         upper_limit = mean_rating + 1.96 * se_rating)
```

```
## # A tibble: 23 x 7
##    `Sample Name` n_responses mean_rating sd_rating se_rating lower_limit
##    <chr>               <int>       <dbl>     <dbl>     <dbl>       <dbl>
##  1 Blackberry 1           93        5.12      2.23     0.231        4.67
##  2 Blackberry 2           93        4.66      2.24     0.232        4.20
##  3 Blackberry 3           93        5.65      2.14     0.222        5.21
##  4 Blackberry 4           93        5.82      2.10     0.217        5.39
##  5 Blackberry 5           93        5.94      1.99     0.207        5.53
##  6 Blueberry 1           105        5.75      1.91     0.186        5.39
##  7 Blueberry 2           105        5.85      1.93     0.188        5.48
##  8 Blueberry 3           105        5.61      1.92     0.188        5.24
##  9 Blueberry 4           105        5.70      2.08     0.203        5.31
## 10 Blueberry 5           105        5.38      2.17     0.212        4.97
## # i 13 more rows
## # i 1 more variable: upper_limit <dbl>
```

Note that in the above example we use `mutate()`, *not* `summarize()`, because we had saved our summarized data. We could also have calculated `lower_limit` and `upper_limit` directly as part of the `summarize()` statement if we hadn't saved the intermediate.

3.5 Groups of columns and across()

It's more common to have groups of **observations** in tidy data, reflected by categorical variables–each `Subject Code` is a group, each `berry` type is a group, each `testing_day` is a group, etc. But we can also have groups of **variables**, as we do in the `berry_data` we've been using!

We have a group of `cata_` variables, a group of liking data with subtypes `9_pt`, `lms_`, `us_`, `_overall`, `_appearance`, etc...

What if we want to count the total number of times each `cata_` attribute was used for one of the berries?

Well, we *can* do this with `summarize()`, but we'd have to type out the names of all 36 columns manually. This is what `select()` helpers are for, and we *can* use them in functions that operate on rows or groups like `filter()`, `mutate()`, and `summarize()` if we use the new `across()` function.

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(.cols = starts_with("cata_"),
                   .fns = sum))
```

```
## # A tibble: 23 x 37
##    `Sample Name` cata_appearance_unevencolor cata_appearance_misshapen
##    <chr>                               <dbl>                     <dbl>
##  1 Blackberry 1                           28                        67
##  2 Blackberry 2                           32                        72
##  3 Blackberry 3                           25                        50
##  4 Blackberry 4                           46                       114
##  5 Blackberry 5                           32                       144
##  6 Blueberry 1                            46                        13
##  7 Blueberry 2                            48                        25
##  8 Blueberry 3                            34                        37
##  9 Blueberry 4                            29                        26
## 10 Blueberry 5                            22                        35
## # i 13 more rows
## # i 34 more variables: cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>,
## #   cata_appearance_notfresh <dbl>, cata_appearance_fresh <dbl>,
## #   cata_appearance_goodshape <dbl>, cata_appearance_goodquality <dbl>,
## #   cata_appearance_none <dbl>, cata_taste_floral <dbl>,
## #   cata_taste_berry <dbl>, cata_taste_green <dbl>, ...
```

You might read this as: `summarize()` `across()` every `.col` that `starts_with("cata_")` by taking the `sum()`.

We can easily expand this to take multiple kinds of summaries for each column, in which case it helps to **name** the functions. `across()` uses **lists** to work with more than one function, so it will look at the **list names** (lefthand-side of the arguments in `list()`) to name the output columns:

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(.cols = starts_with("cata_"),
                                #the sum of binary cata data gives the citation frequenc
                   .fns = list(frequency = sum,
                                #meanwhile, the mean gives the percentage.
                                percentage = mean)))
```

```
## # A tibble: 23 x 73
##    `Sample Name` cata_appearance_unevencolor_frequency cata_appearance_unevenc~1
##    <chr>                                         <dbl>                     <dbl>
##  1 Blackberry 1                                     28                    0.0936
##  2 Blackberry 2                                     32                    0.107
##  3 Blackberry 3                                     25                    0.0836
##  4 Blackberry 4                                     46                    0.154
##  5 Blackberry 5                                     32                    0.107
##  6 Blueberry 1                                      46                    0.147
##  7 Blueberry 2                                      48                    0.153
##  8 Blueberry 3                                      34                    0.109
##  9 Blueberry 4                                      29                    0.0927
## 10 Blueberry 5                                      22                    0.0703
## # i 13 more rows
## # i abbreviated name: 1: cata_appearance_unevencolor_percentage
## # i 70 more variables: cata_appearance_misshapen_frequency <dbl>,
## #   cata_appearance_misshapen_percentage <dbl>,
## #   cata_appearance_creased_frequency <dbl>,
## #   cata_appearance_creased_percentage <dbl>,
## #   cata_appearance_seedy_frequency <dbl>, ...
```

`across()` is capable of taking arbitrarily complicated functions, but you'll notice that we didn't include the parentheses we usually see after a function name for `sum()` and `mean()`. `across()` will just pipe in each column to the `.fns` as the first argument. That means, however, that there's nowhere for us to put additional arguments like `na.rm`.

We can use **lambda functions** to . This basically just means starting each function off with a tilde (~) and telling `across()` where we want our `.cols` to go manually using `.x`.

Remember, the tilde is usually above the backtick on QWERTY keyboards. Try the instructions here and here to type a tilde if you have a non-QWERTY keyboard. If those methods don't work, try this guide for Italian keyboards, this guide for Spanish keyboards, this guide for German, this guide for Norwegian, or this guide for Swedish keyboards.

This will be necessary if we want to take the average of our various liking columns without those pesky `NA`s propogating.

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(.cols = starts_with("9pt_"),
                   .fns = list(mean = mean,
                               sd = sd))) #All NA
```

```
## # A tibble: 23 x 11
##    `Sample Name` `9pt_appearance_mean` `9pt_appearance_sd` `9pt_overall_mean`
##    <chr>                         <dbl>                <dbl>               <dbl>
##  1 Blackberry 1                     NA                   NA                  NA
##  2 Blackberry 2                     NA                   NA                  NA
##  3 Blackberry 3                     NA                   NA                  NA
##  4 Blackberry 4                     NA                   NA                  NA
##  5 Blackberry 5                     NA                   NA                  NA
##  6 Blueberry 1                      NA                   NA                  NA
##  7 Blueberry 2                      NA                   NA                  NA
##  8 Blueberry 3                      NA                   NA                  NA
##  9 Blueberry 4                      NA                   NA                  NA
## 10 Blueberry 5                      NA                   NA                  NA
## # i 13 more rows
## # i 7 more variables: `9pt_overall_sd` <dbl>, `9pt_taste_mean` <dbl>,
## #   `9pt_taste_sd` <dbl>, `9pt_texture_mean` <dbl>, `9pt_texture_sd` <dbl>,
## #   `9pt_aroma_mean` <dbl>, `9pt_aroma_sd` <dbl>
```

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(.cols = starts_with("9pt_"),
                   .fns = list(mean = ~ mean(.x, na.rm = TRUE),
                               sd = ~ sd(.x, na.rm = TRUE))))
```

```
## # A tibble: 23 x 11
##    `Sample Name` `9pt_appearance_mean` `9pt_appearance_sd` `9pt_overall_mean`
##    <chr>                         <dbl>                <dbl>               <dbl>
##  1 Blackberry 1                   6.57                 1.64                5.12
##  2 Blackberry 2                   6.43                 1.61                4.66
##  3 Blackberry 3                   6.96                 1.37                5.65
##  4 Blackberry 4                   5.90                 1.97                5.82
##  5 Blackberry 5                   5.99                 1.82                5.94
##  6 Blueberry 1                    6.75                 1.55                5.75
##  7 Blueberry 2                    6.61                 1.53                5.85
##  8 Blueberry 3                    6.4                  1.68                5.61
##  9 Blueberry 4                    6.45                 1.72                5.70
## 10 Blueberry 5                    6.39                 1.74                5.38
## # i 13 more rows
```

```
## # i 7 more variables: `9pt_overall_sd` <dbl>, `9pt_taste_mean` <dbl>,
## #   `9pt_taste_sd` <dbl>, `9pt_texture_mean` <dbl>, `9pt_texture_sd` <dbl>,
## #   `9pt_aroma_mean` <dbl>, `9pt_aroma_sd` <dbl>
```

The `across()` function is very powerful and also pretty new to the tidyverse. It's probably the least intuitive thing we're covering today other than graphs, in my opinion, but it's also leagues better than the `summarize_at()`, `summarize_if()`, and `summarize_all()` functions that came before.

You can also use `across()` to `filter()` rows based on multiple columns or `mutate()` multiple columns at once, but you don't need to worry about `across()` at all if you know exactly what columns you're working with and don't mind typing them all out!

Chapter 4

Wrangling data with `tidyverse`: Reshaping and combining tables

4.1 Pivot tables- wider and longer data

Users of Excel may be familiar with the idea of pivot tables. These are functions that let us make our data tidier. To quote Wickham and Grolemund:

> here are three interrelated rules which make a dataset tidy:
>
> 1. Each variable must have its own column.
> 2. Each observation must have its own row.
> 3. Each value must have its own cell.

While these authors present "tidiness" of data as an objective property, I'd argue that data is always tidy **for a specific purpose**. For example, our data is relatively tidy with one row per tasting event (one person tasting one berry), but this data still has an unruly number of variables (92 columns!!). You've already learned some tricks for dealing with large numbers of columns at once like `across()` and other functions using select helpers, but we have to do this *every time* we use `mutate()`, `summarize()`, or a similar function.

We could also treat the attribute or question as an independent variable affecting the response. If we take this view, then the tidiest dataset actually has one row for each person's response to a single `question`. If we want to make plots or do other modelling, this **longer** form is often more tractable and lets us do operations on the whole dataset with less code.

We can use the `pivot_longer()` function to change our data to make the implicit variable explicit and to make our data tidier.

```
berry_data %>%
  select(`Subject Code`, `Sample Name`, berry, starts_with("cata_"), starts_with("9pt"))
  pivot_longer(cols = starts_with("cata_"),
               names_prefix = "cata_",
               names_to = "attribute",
               values_to = "presence") ->
  berry_data_cata_long
#The names_prefix will be *removed* from the start of every column name
#before putting the rest of the name in the `names_to` column

berry_data_cata_long
```

```
## # A tibble: 270,252 x 10
##    `Subject Code` `Sample Name` berry `9pt_appearance` `9pt_overall` `9pt_taste`
##             <dbl> <chr>         <chr>             <dbl>         <dbl>       <dbl>
##  1           1001 raspberry 6   rasp~                 4             4           4
##  2           1001 raspberry 6   rasp~                 4             4           4
##  3           1001 raspberry 6   rasp~                 4             4           4
##  4           1001 raspberry 6   rasp~                 4             4           4
##  5           1001 raspberry 6   rasp~                 4             4           4
##  6           1001 raspberry 6   rasp~                 4             4           4
##  7           1001 raspberry 6   rasp~                 4             4           4
##  8           1001 raspberry 6   rasp~                 4             4           4
##  9           1001 raspberry 6   rasp~                 4             4           4
## 10           1001 raspberry 6   rasp~                 4             4           4
## # i 270,242 more rows
## # i 4 more variables: `9pt_texture` <dbl>, `9pt_aroma` <dbl>, attribute <chr>,
## #   presence <dbl>
```

Remember that `tibbles` and `data.frames` can only have one data type per column (`logical > integer > numeric > character`), however! If we have one row for each CATA, JAR, hedonic scale, AND free response question, the `value` column would have a mixture of different data types. This is why we have to tell `pivot_longer()` which `cols` to pull the `names` and `values` from.

Now for each unique combination of `Sample Name` and `Subject Code`, we have 36 rows, one for each CATA question that was asked. The variables that weren't listed in the `cols` argument are just replicated on each of these rows. Each of the 36 rows that represent `Subject Code` 1001's CATA responses for `raspberry 6` has the same `Subject Code`, `Sample Name`, `berry`, and various `9pt_` ratings as the other 35.

Sometimes we want to have "wider" or "untidy" data. We can use `pivot_wider()` to reverse the effects of `pivot_longer()`.

```
berry_data_cata_long %>%
  pivot_wider(names_from = "attribute",
              values_from = "presence",
              names_prefix = "cata_") #pivot_wider *adds* the names_prefix
```

```
## # A tibble: 7,507 x 44
##    `Subject Code` `Sample Name` berry `9pt_appearance` `9pt_overall` `9pt_taste`
##             <dbl> <chr>         <chr>             <dbl>         <dbl>       <dbl>
##  1           1001 raspberry 6   rasp~                 4             4           4
##  2           1001 raspberry 5   rasp~                 8             9           9
##  3           1001 raspberry 2   rasp~                 4             3           3
##  4           1001 raspberry 3   rasp~                 7             7           6
##  5           1001 raspberry 4   rasp~                 7             4           3
##  6           1001 raspberry 1   rasp~                 7             4           3
##  7           1002 raspberry 6   rasp~                 6             4           4
##  8           1002 raspberry 5   rasp~                 8             7           4
##  9           1002 raspberry 2   rasp~                 8             7           6
## 10           1002 raspberry 3   rasp~                 7             9           9
## # i 7,497 more rows
## # i 38 more variables: `9pt_texture` <dbl>, `9pt_aroma` <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodshape <dbl>,
## #   cata_appearance_goodquality <dbl>, cata_appearance_none <dbl>, ...
```

Pivoting is an incredibly powerful and incredibly common data manipulation technique that will become even more powerful when we need to make complex graphs later. Different functions and analyses may require the data in different longer or wider formats, and you will often find yourself starting with even less tidy data than what we've provided.

For an example of this power, let's imagine that we want to compare the 3 different liking scales by normalizing each by the `mean()` and `sd()` of that particular scale, then comparing average liking for each attribute of each berry across the three scales.

```
berry_data %>%
  pivot_longer(cols = starts_with(c("9pt_","lms_","us_")),
               names_to = c("scale", "attribute"),
               names_sep = "_",
               values_to = "rating",
               values_drop_na = TRUE) %>%
  group_by(scale) %>%
  mutate(normalized_rating = (rating - mean(rating)) / sd(rating)) %>%
```

```
  group_by(scale, attribute, berry) %>%
  summarize(avg_liking = mean(normalized_rating)) %>%
  pivot_wider(names_from = scale,
              values_from = avg_liking)
```

```
## `summarise()` has grouped output by 'scale', 'attribute'. You can override
## using the `.groups` argument.
```

```
## # A tibble: 17 x 5
## # Groups:   attribute [5]
##    attribute  berry         `9pt`      lms      us
##    <chr>      <chr>         <dbl>    <dbl>   <dbl>
##  1 appearance blackberry  0.284    0.364    0.327
##  2 appearance blueberry   0.381    0.424    0.462
##  3 appearance raspberry   0.246    0.236    0.223
##  4 appearance strawberry -0.164   -0.226   -0.220
##  5 aroma      strawberry  0.0351  -0.0676  -0.0951
##  6 overall    blackberry -0.177   -0.209   -0.177
##  7 overall    blueberry   0.0234   0.0576   0.166
##  8 overall    raspberry   0.0261   0.0300   0.0707
##  9 overall    strawberry -0.150   -0.179   -0.200
## 10 taste      blackberry -0.289   -0.301   -0.336
## 11 taste      blueberry  -0.0611   0.0202   0.0366
## 12 taste      raspberry  -0.0291  -0.0336  -0.0359
## 13 taste      strawberry -0.306   -0.292   -0.339
## 14 texture    blackberry -0.0467   0.00645 -0.0118
## 15 texture    blueberry   0.159    0.202    0.284
## 16 texture    raspberry  -0.00677 -0.00607  0.0289
## 17 texture    strawberry -0.0602  -0.0531  -0.0768
```

While pivoting may seem simple at first, it can also get pretty confusing! That example required two different pivots! We'll be using these tools throughout the rest of the tutorial, so I wanted to give exposure, but mastering them takes trial and error. I recommend taking a look at the relevant chapter in Wickham and Grolemund for details.

4.2 Combining data

While we've been using a single dataset read in from one `.csv` file, this will not always be the way your data is stored when you start working with it. There may be several surveys you're combining, or a separate file with survey responses, another with your blinding code key, and still another with data from a collaborator.

The `tidyverse` verb that you use for this combination depends on whether your datasets have matching columns/variables (say, data from two different years or locations of a project) or matching rows/observations (say, sensory and chemical data).

The "matching variables" case can also happen if we want to stack the outputs of multiple independent `summarize()` calls, such as making a stacked demographic table. Since our data doesn't have demographic data removed, let's instead try to make a table that reports our sample size for each `berry` type and each of the three kinds of liking scales.

If we wanted the combinations of these levels (e.g., the sample size of 9-pt scale ratings of strawberries), we could use one `summarize()` or `count()` call, but we need to do a little extra work if we want to `summarize()` the whole dataset more than one different way.

```
berry_type_counts <-
  berry_data %>%
  group_by(berry) %>%
  summarize(n = n_distinct(`Subject Code`, test_day)) %>%
  rename(Level = berry)

berry_scale_counts <-
  berry_data %>%
  pivot_longer(ends_with("_overall"),
               names_sep = "_", names_to = c("Scale", NA),
               values_to = "Used", values_drop_na = TRUE) %>%
  group_by(Scale) %>%
  summarize(n = n_distinct(`Subject Code`, test_day)) %>%
  rename(Level = Scale)

#These are both summarizing the same set of observations into different groups:
sum(berry_scale_counts$n)
```

```
## [1] 1301
```

```
sum(berry_type_counts$n)
```

```
## [1] 1301
```

```
#Hence needing two summarize() calls

bind_rows(Berry = berry_type_counts,
          Scale = berry_scale_counts,
          .id = "Variable") #This makes a new column called "Variable" which will
```

```
## # A tibble: 7 x 3
##   Variable Level          n
##   <chr>    <chr>      <int>
## 1 Berry    blackberry   299
## 2 Berry    blueberry    313
## 3 Berry    raspberry    358
## 4 Berry    strawberry   331
## 5 Scale    9pt          423
## 6 Scale    lms          435
## 7 Scale    us           443
```

```
                              #specify which of the two tables each row came from
```

For multiple tables that share the same observations, we could want to add follow-up survey data using the same participants or genetic information about the berries. In the latter case, our table of berry genetics would have less rows, but the `tidyverse` actually handles them with the same verbs.

There *is* a `bind_cols()` function, but it's easy to accidentally have the raspberries on the top in one table and the blueberries on the top in another, or to have one table sorted alphabetically and another by blinding code or participant ID, so it's safer to use the `*_join()` functions if you're adding columns instead of rows. `left_join()` is the most common.

We'll make up some demographic data to join to our existing table.

```
demographics <-
  berry_data %>%
  distinct(`Subject Code`) %>%
  mutate(Age = round(rnorm(n(), 45, 6)),
         Gender = ifelse(rbinom(n(), 1, 0.6), "F", "M"),
         Location = sample(state.name, n(), replace = TRUE)) %>%
  rename(ID = `Subject Code`) #To demonstrate how you can manually configure
                              #the columns used to align the datasets

#And now we can join it:
berry_data %>%
  select(-Age, -Gender) %>% #Getting rid of the empty demographic columns first
  left_join(demographics, by = c("Subject Code" = "ID"))
```

```
## # A tibble: 7,507 x 93
##    `Subject Code` `Participant Name` `Start Time (UTC)` `End Time (UTC)`
##             <dbl>              <dbl> <chr>               <chr>
##  1           1001               1001 6/13/2019 21:05     6/13/2019 21:09
##  2           1001               1001 6/13/2019 20:55     6/13/2019 20:59
##  3           1001               1001 6/13/2019 20:49     6/13/2019 20:53
```

```
##  4           1001               1001 6/13/2019 20:45    6/13/2019 20:48
##  5           1001               1001 6/13/2019 21:00    6/13/2019 21:03
##  6           1001               1001 6/13/2019 21:10    6/13/2019 21:13
##  7           1002               1002 6/13/2019 20:08    6/13/2019 20:11
##  8           1002               1002 6/13/2019 19:57    6/13/2019 20:01
##  9           1002               1002 6/13/2019 20:13    6/13/2019 20:17
## 10           1002               1002 6/13/2019 20:03    6/13/2019 20:07
## # i 7,497 more rows
## # i 89 more variables: `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `Sample Name` <chr>, `9pt_appearance` <dbl>, pre_expectation <dbl>,
## #   jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

`anti_join()` can be used to *remove* data. If you have a list of participants whose responses had data quality issues, you can put it in the second argument of `anti_join()` to return the lefthand table with those entries removed.

We do have some repeat participants in the berry tests, because the actual study involved repeated measures. But in online surveys, repeat answers could be a problem necessitating all data removal. Let's demonstrate how we'd do that with `anti_join()`:

```
problem_participants <-
  berry_data %>%
  group_by(`Participant Name`) %>%
  summarize(sessions = n_distinct(test_day)) %>%
  filter(sessions > 1)

#We don't need to specify by if they share the column name they're joining on
#and NO OTHERS
berry_data %>%
  anti_join(problem_participants)
```

```
## Joining with `by = join_by(`Participant Name`)`
```

```
## # A tibble: 3,755 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
```

```
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 3,745 more rows
## # i 87 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, 'Sample Name' <chr>, '9pt_appearance' <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

`anti_join()` also gives priority to the first/lefthand argument, usually the one you're piping in with `%>%`. It returns the rows in your left tibble that don't have corresponding entries in the righthand one. It also does *not* add the columns that are unique to your right table. There is no `n` column in the output.

4.3 Utilities for data management

Honestly, the amount of power in `tidyverse` is way more than we can cover today, and is covered more comprehensively (obviously) by Wickham and Grolemund. However, I want to name a few more utilities we will make a lot of use of today (and you will want to know about for your own work).

4.3.1 Rename your columns

Often you will import data with bad column names or you'll realize you need to rename variables during your workflow. This is one way to get around having to type a bunch of backticks forever. For this, you can use the `rename()` function:

```
names(berry_data)
```

```
##  [1] "Subject Code"                "Participant Name"
##  [3] "Gender"                      "Age"
##  [5] "Start Time (UTC)"            "End Time (UTC)"
##  [7] "Serving Position"            "Sample Identifier"
##  [9] "Sample Name"                 "9pt_appearance"
## [11] "pre_expectation"             "jar_color"
## [13] "jar_gloss"                   "jar_size"
## [15] "cata_appearance_unevencolor" "cata_appearance_misshapen"
## [17] "cata_appearance_creased"     "cata_appearance_seedy"
```

```
## [19] "cata_appearance_bruised"     "cata_appearance_notfresh"
## [21] "cata_appearance_fresh"       "cata_appearance_goodshape"
## [23] "cata_appearance_goodquality" "cata_appearance_none"
## [25] "9pt_overall"                 "verbal_likes"
## [27] "verbal_dislikes"             "9pt_taste"
## [29] "grid_sweetness"              "grid_tartness"
## [31] "grid_raspberryflavor"        "jar_sweetness"
## [33] "jar_tartness"                "cata_taste_floral"
## [35] "cata_taste_berry"            "cata_taste_green"
## [37] "cata_taste_grassy"           "cata_taste_fermented"
## [39] "cata_taste_tropical"         "cata_taste_fruity"
## [41] "cata_taste_citrus"           "cata_taste_earthy"
## [43] "cata_taste_candy"            "cata_taste_none"
## [45] "9pt_texture"                 "grid_seediness"
## [47] "grid_firmness"               "grid_juiciness"
## [49] "jar_firmness"                "jar_juciness"
## [51] "post_expectation"            "price"
## [53] "product_tier"                "purchase_intent"
## [55] "subject"                     "test_day"
## [57] "us_appearance"               "us_overall"
## [59] "us_taste"                    "us_texture"
## [61] "lms_appearance"              "lms_overall"
## [63] "lms_taste"                   "lms_texture"
## [65] "cata_appearane_bruised"      "cata_appearance_goodshapre"
## [67] "cata_appearance_goodcolor"   "grid_blackberryflavor"
## [69] "cata_taste_cinnamon"         "cata_taste_lemon"
## [71] "cata_taste_clove"            "cata_taste_minty"
## [73] "cata_taste_grape"            "grid_crispness"
## [75] "jar_crispness"               "jar_juiciness"
## [77] "cata_appearane_creased"      "grid_blueberryflavor"
## [79] "cata_taste_piney"            "cata_taste_peachy"
## [81] "9pt_aroma"                   "grid_strawberryflavor"
## [83] "cata_taste_caramel"          "cata_taste_grapey"
## [85] "cata_taste_melon"            "cata_taste_cherry"
## [87] "grid_crunchiness"            "jar_crunch"
## [89] "us_aroma"                    "lms_aroma"
## [91] "berry"                       "sample"
```

```
berry_data %>%
  rename(Sample = `Sample Name`,
         Subject = `Participant Name`) %>%
  select(Subject, Sample, everything()) #no more backticks!
```

```
## # A tibble: 7,507 x 92
##    Subject Sample      `Subject Code` Gender Age   `Start Time (UTC)`
```

```
##      <dbl> <chr>                <dbl> <lgl>  <lgl> <chr>
##  1    1001 raspberry 6           1001 NA     NA    6/13/2019 21:05
##  2    1001 raspberry 5           1001 NA     NA    6/13/2019 20:55
##  3    1001 raspberry 2           1001 NA     NA    6/13/2019 20:49
##  4    1001 raspberry 3           1001 NA     NA    6/13/2019 20:45
##  5    1001 raspberry 4           1001 NA     NA    6/13/2019 21:00
##  6    1001 raspberry 1           1001 NA     NA    6/13/2019 21:10
##  7    1002 raspberry 6           1002 NA     NA    6/13/2019 20:08
##  8    1002 raspberry 5           1002 NA     NA    6/13/2019 19:57
##  9    1002 raspberry 2           1002 NA     NA    6/13/2019 20:13
## 10    1002 raspberry 3           1002 NA     NA    6/13/2019 20:03
## # i 7,497 more rows
## # i 86 more variables: 'End Time (UTC)' <chr>, 'Serving Position' <dbl>,
## #   'Sample Identifier' <dbl>, '9pt_appearance' <dbl>, pre_expectation <dbl>,
## #   jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

You can also rename by position, but be sure you have the right order and don't change the input data later:

```
berry_data %>%
  rename(Subject = 1)
```

```
## # A tibble: 7,507 x 92
##    Subject 'Participant Name' Gender Age   'Start Time (UTC)' 'End Time (UTC)'
##      <dbl>              <dbl> <lgl>  <lgl> <chr>              <chr>
##  1    1001               1001 NA     NA    6/13/2019 21:05    6/13/2019 21:09
##  2    1001               1001 NA     NA    6/13/2019 20:55    6/13/2019 20:59
##  3    1001               1001 NA     NA    6/13/2019 20:49    6/13/2019 20:53
##  4    1001               1001 NA     NA    6/13/2019 20:45    6/13/2019 20:48
##  5    1001               1001 NA     NA    6/13/2019 21:00    6/13/2019 21:03
##  6    1001               1001 NA     NA    6/13/2019 21:10    6/13/2019 21:13
##  7    1002               1002 NA     NA    6/13/2019 20:08    6/13/2019 20:11
##  8    1002               1002 NA     NA    6/13/2019 19:57    6/13/2019 20:01
##  9    1002               1002 NA     NA    6/13/2019 20:13    6/13/2019 20:17
## 10    1002               1002 NA     NA    6/13/2019 20:03    6/13/2019 20:07
## # i 7,497 more rows
## # i 86 more variables: 'Serving Position' <dbl>, 'Sample Identifier' <dbl>,
## #   'Sample Name' <chr>, '9pt_appearance' <dbl>, pre_expectation <dbl>,
## #   jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

4.3.2 Relocate your columns

If you `mutate()` columns or just have a big data set with a lot of variables, often you want to move columns around. This is a pain to do with `[]`, but again `tidyverse` has a utility to move things around easily: `relocate()`.

```
berry_data %>%
  relocate(`Sample Name`) # giving no other arguments will move to front
```

```
## # A tibble: 7,507 x 92
##    `Sample Name` `Subject Code` `Participant Name` Gender Age
##    <chr>                  <dbl>              <dbl> <lgl>  <lgl>
##  1 raspberry 6             1001               1001 NA     NA
##  2 raspberry 5             1001               1001 NA     NA
##  3 raspberry 2             1001               1001 NA     NA
##  4 raspberry 3             1001               1001 NA     NA
##  5 raspberry 4             1001               1001 NA     NA
##  6 raspberry 1             1001               1001 NA     NA
##  7 raspberry 6             1002               1002 NA     NA
##  8 raspberry 5             1002               1002 NA     NA
##  9 raspberry 2             1002               1002 NA     NA
## 10 raspberry 3             1002               1002 NA     NA
## # i 7,497 more rows
## # i 87 more variables: `Start Time (UTC)` <chr>, `End Time (UTC)` <chr>,
## #   `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `9pt_appearance` <dbl>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_gloss <dbl>, jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>, ...
```

You can also use `relocate()` to specify positions

```
berry_data %>%
  relocate(Gender, Age, `Subject Code`, `Start Time (UTC)`,
           `End Time (UTC)`, `Sample Identifier`,
           # move repetitive and empty columns to the end
           .after = berry)
```

```
## # A tibble: 7,507 x 92
##    `Participant Name` `Serving Position` `Sample Name` `9pt_appearance`
##                 <dbl>              <dbl> <chr>                    <dbl>
##  1               1001                  5 raspberry 6                  4
##  2               1001                  3 raspberry 5                  8
##  3               1001                  2 raspberry 2                  4
```

```
##  4              1001                 1 raspberry 3                 7
##  5              1001                 4 raspberry 4                 7
##  6              1001                 6 raspberry 1                 7
##  7              1002                 3 raspberry 6                 6
##  8              1002                 1 raspberry 5                 8
##  9              1002                 4 raspberry 2                 8
## 10              1002                 2 raspberry 3                 7
## # i 7,497 more rows
## # i 88 more variables: pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>,
## #   jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>,
## #   cata_appearance_notfresh <dbl>, cata_appearance_fresh <dbl>,
## #   cata_appearance_goodshape <dbl>, cata_appearance_goodquality <dbl>, ...
```

4.3.3 Remove missing values

Missing values (the `NA`s you've been seeing so much) can be a huge pain, because they make more of themselves.

```
mean(berry_data$price) #This column had no NAs, so we can take the average
```

```
## [1] 2.962896
```

```
mean(berry_data$`9pt_overall`) #This column has some NAs, so we get NA
```

```
## [1] NA
```

Many base R functions that take a vector and return some mathematical function (e.g., `mean()`, `sum()`, `sd()`) have an argument called `na.rm` that can be set to just act as if the values aren't there at all.

```
mean(berry_data$`9pt_overall`, na.rm = TRUE) #We get the average of only the valid num
```

```
## [1] 5.679346
```

```
sum(berry_data$`9pt_overall`, na.rm = TRUE) /
  length(berry_data$`9pt_overall`) #The denominator is NOT the same as the total numbe
```

```
## [1] 1.84974
```

```
sum(berry_data$`9pt_overall`, na.rm = TRUE) /
  sum(!is.na(berry_data$`9pt_overall`)) #The denominator is the number of non-NA values
```

```
## [1] 5.679346
```

However, this isn't always convenient. Sometimes it may be easier to simply get rid of all observations with any missing values, which tidyverse has a handy `drop_na()` function for:

```
berry_data %>%
  drop_na() #All of our rows have *some* NA values, so this returns nothing
```

```
## # A tibble: 0 x 92
## # i 92 variables: Subject Code <dbl>, Participant Name <dbl>, Gender <lgl>,
## #   Age <lgl>, Start Time (UTC) <chr>, End Time (UTC) <chr>,
## #   Serving Position <dbl>, Sample Identifier <dbl>, Sample Name <chr>,
## #   9pt_appearance <dbl>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_gloss <dbl>, jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>, ...
```

```
berry_data %>%
  select(`Participant Name`, `Sample Name`, contains("9pt_")) %>%
  drop_na() #Now we get only respondants who answered all 9-point liking questions.
```

```
## # A tibble: 600 x 7
##    `Participant Name` `Sample Name` `9pt_appearance` `9pt_overall` `9pt_taste`
##                 <dbl> <chr>                    <dbl>         <dbl>       <dbl>
##  1               2001 Strawberry4                  5             5           4
##  2               2001 Strawberry1                  6             2           2
##  3               2001 Strawberry2                  1             6           6
##  4               2001 Strawberry6                  3             3           2
##  5               2001 Strawberry3                  8             7           8
##  6               2001 Strawberry5                  4             6           6
##  7               2002 Strawberry4                  2             4           3
##  8               2002 Strawberry1                  4             6           7
##  9               2002 Strawberry2                  3             6           6
## 10               2002 Strawberry6                  7             7           8
## # i 590 more rows
## # i 2 more variables: `9pt_texture` <dbl>, `9pt_aroma` <dbl>
```

You can also use `drop_na()` with specific columns, which is useful to avoid losing all of your data!

```
berry_data %>%
  drop_na(`9pt_overall`)
```

```
## # A tibble: 2,445 x 92
##    `Subject Code` `Participant Name` Gender Age   `Start Time (UTC)`
##             <dbl>              <dbl> <lgl>  <lgl> <chr>
##  1           1001               1001 NA     NA    6/13/2019 21:05
##  2           1001               1001 NA     NA    6/13/2019 20:55
##  3           1001               1001 NA     NA    6/13/2019 20:49
##  4           1001               1001 NA     NA    6/13/2019 20:45
##  5           1001               1001 NA     NA    6/13/2019 21:00
##  6           1001               1001 NA     NA    6/13/2019 21:10
##  7           1002               1002 NA     NA    6/13/2019 20:08
##  8           1002               1002 NA     NA    6/13/2019 19:57
##  9           1002               1002 NA     NA    6/13/2019 20:13
## 10           1002               1002 NA     NA    6/13/2019 20:03
## # i 2,435 more rows
## # i 87 more variables: `End Time (UTC)` <chr>, `Serving Position` <dbl>,
## #   `Sample Identifier` <dbl>, `Sample Name` <chr>, `9pt_appearance` <dbl>,
## #   pre_expectation <dbl>, jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

Or you may want to remove any columns/variables that have some missing data, which is one of the most common uses of `where()`:

```
#Only 38 columns with absolutely no missing values.
#This loses all of the liking data.
berry_data %>%
  select(where(~none(.x, is.na)))
```

```
## # A tibble: 7,507 x 38
##    `Subject Code` `Participant Name` `Start Time (UTC)` `End Time (UTC)`
##             <dbl>              <dbl> <chr>              <chr>
##  1           1001               1001 6/13/2019 21:05    6/13/2019 21:09
##  2           1001               1001 6/13/2019 20:55    6/13/2019 20:59
##  3           1001               1001 6/13/2019 20:49    6/13/2019 20:53
##  4           1001               1001 6/13/2019 20:45    6/13/2019 20:48
##  5           1001               1001 6/13/2019 21:00    6/13/2019 21:03
##  6           1001               1001 6/13/2019 21:10    6/13/2019 21:13
##  7           1002               1002 6/13/2019 20:08    6/13/2019 20:11
##  8           1002               1002 6/13/2019 19:57    6/13/2019 20:01
##  9           1002               1002 6/13/2019 20:13    6/13/2019 20:17
```

```
## 10           1002               1002 6/13/2019 20:03    6/13/2019 20:07
## # i 7,497 more rows
## # i 34 more variables: `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `Sample Name` <chr>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodquality <dbl>,
## #   cata_appearance_none <dbl>, grid_sweetness <dbl>, grid_tartness <dbl>, ...
```

Both of the above methods guarantee that you will have an output with absolutely no missing data, but may be over-zealous if, say, everyone answered overall liking on one of the three scales and we want to do some work to combine those later. `filter()` and `select()` can be combined to do infinitely complex missing value removal.

```
#You'll notice that only strawberries have any non-NA liking values, actually
berry_data %>%
  select(where(~!every(.x, is.na))) %>% #remove columns with no data
  filter(!(is.na(`9pt_aroma`) & is.na(lms_aroma) & is.na(us_aroma)))
```

```
## # A tibble: 1,986 x 90
##    `Subject Code` `Participant Name` `Start Time (UTC)` `End Time (UTC)`
##             <dbl>              <dbl> <chr>               <chr>
##  1           2001               2001 6/24/2019 20:18     6/24/2019 20:22
##  2           2001               2001 6/24/2019 20:30     6/24/2019 20:35
##  3           2001               2001 6/24/2019 20:23     6/24/2019 20:28
##  4           2001               2001 6/24/2019 20:14     6/24/2019 20:17
##  5           2001               2001 6/24/2019 20:35     6/24/2019 20:39
##  6           2001               2001 6/24/2019 20:08     6/24/2019 20:13
##  7           2002               2002 6/24/2019 20:21     6/24/2019 20:25
##  8           2002               2002 6/24/2019 20:14     6/24/2019 20:17
##  9           2002               2002 6/24/2019 19:59     6/24/2019 20:04
## 10           2002               2002 6/24/2019 20:09     6/24/2019 20:13
## # i 1,976 more rows
## # i 86 more variables: `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `Sample Name` <chr>, `9pt_appearance` <dbl>, pre_expectation <dbl>,
## #   jar_color <dbl>, jar_gloss <dbl>, jar_size <dbl>,
## #   cata_appearance_unevencolor <dbl>, cata_appearance_misshapen <dbl>,
## #   cata_appearance_creased <dbl>, cata_appearance_seedy <dbl>,
## #   cata_appearance_bruised <dbl>, cata_appearance_notfresh <dbl>, ...
```

4.3.4 Counting categorical variables

Often, we'll want to count how many observations are in a group without having to actually count ourselves. Do we have enough observations for each sample?

How many people in each demographic category do we have? Is it balanced?

You've already written code to do this, if you've been following along! `summarize()` is incredibly powerful, and it will happily use *any* function that takes a vector or vectors and returns a single value. This includes categorical or `chr` data!

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(n_responses = n())
```

```
## # A tibble: 23 x 2
##    `Sample Name` n_responses
##    <chr>               <int>
##  1 Blackberry 1          299
##  2 Blackberry 2          299
##  3 Blackberry 3          299
##  4 Blackberry 4          299
##  5 Blackberry 5          299
##  6 Blueberry 1           313
##  7 Blueberry 2           313
##  8 Blueberry 3           313
##  9 Blueberry 4           313
## 10 Blueberry 5           313
## # i 13 more rows
```

We can also do this with a little less typing using `count()`, which is handy if we're repeatedly doing a lot of counting observations in various categories (like for CATA tests and Correspondence Analyses):

```
#Counts the number of observations (rows) of each berry
berry_data %>%
  count(`Sample Name`)
```

```
## # A tibble: 23 x 2
##    `Sample Name`     n
##    <chr>         <int>
##  1 Blackberry 1    299
##  2 Blackberry 2    299
##  3 Blackberry 3    299
##  4 Blackberry 4    299
##  5 Blackberry 5    299
##  6 Blueberry 1     313
##  7 Blueberry 2     313
##  8 Blueberry 3     313
```

```
##  9 Blueberry 4     313
## 10 Blueberry 5     313
## # i 13 more rows
```

```
#Number of observations, *not necessarily* the number of participants!
berry_data %>%
  count(berry)
```

```
## # A tibble: 4 x 2
##   berry          n
##   <chr>      <int>
## 1 blackberry  1495
## 2 blueberry   1878
## 3 raspberry   2148
## 4 strawberry  1986
```

Depending on the shape of your data, the number of rows may or may not be the count you actually want. Maybe we want to know how many people participated in each day of testing, but we have one row per *tasting event*.

We could use `pivot_wider()` to reshape our data first, so we have one row per *completed tasting session*, but since `count()` drops most columns anyways, we only really need one row for each thing we care about. `distinct()` can be handy here. It keeps one row for each **distinct** combination of the columns you give it, getting rid of all other columns so it doesn't have to worry about the fact that one person gave multiple different `9pt_overall` ratings per `test_day`.

```
#Two columns, with one row for each completed tasting session
#(each reflects 5-6 rows in the initial data)
berry_data %>%
  distinct(test_day, `Subject Code`)
```

```
## # A tibble: 1,301 x 2
##    test_day        `Subject Code`
##    <chr>                    <dbl>
##  1 Raspberry Day 1           1001
##  2 Raspberry Day 1           1002
##  3 Raspberry Day 1           1004
##  4 Raspberry Day 1           1005
##  5 Raspberry Day 1           1006
##  6 Raspberry Day 1           1007
##  7 Raspberry Day 1           1008
##  8 Raspberry Day 1           1009
##  9 Raspberry Day 1           1010
## 10 Raspberry Day 1           1011
## # i 1,291 more rows
```

```
#Counts the number of participants per testing day
berry_data %>%
  distinct(test_day, `Subject Code`) %>%
  count(test_day)
```

```
## # A tibble: 12 x 2
##    test_day             n
##    <chr>            <int>
##  1 Blackberry Day 1   108
##  2 Blackberry Day 2    88
##  3 Blackberry Day 3   103
##  4 Blueberry Day 1    102
##  5 Blueberry Day 2    114
##  6 Blueberry Day 3     97
##  7 Raspberry Day 1    131
##  8 Raspberry Day 2    120
##  9 Raspberry Day 3    107
## 10 Strawberry Day 1   108
## 11 Strawberry Day 2   106
## 12 Strawberry Day 3   117
```

4.3.5 Sort your data

More frequently, we will want to rearrange our rows, which can be done with `arrange()`. All you have to do is give `arrange()` one or more columns to sort the data by. You can use either the `desc()` or the `-` shortcut to sort in reverse order. Whether ascending or descending, `arrange()` places missing values at the bottom.

```
berry_data %>%
  arrange(desc(lms_overall)) %>%
  # which berries had the highest liking on the lms?
  select(`Sample Name`, `Participant Name`, lms_overall)
```

```
## # A tibble: 7,507 x 3
##    `Sample Name` `Participant Name` lms_overall
##    <chr>                     <dbl>       <dbl>
##  1 raspberry 2                5135         100
##  2 raspberry 6                7033         100
##  3 Blackberry 4               5273         100
##  4 Blackberry 4               7135         100
##  5 Blackberry 3               7135         100
##  6 Blackberry 5               7135         100
```

```
##  7 Blueberry 5                 5113         100
##  8 Blueberry 6                 5127         100
##  9 Blueberry 3                 7040         100
## 10 Strawberry1                 1273         100
## # i 7,497 more rows
```

You can sort alphabetically as well:

```
# using a dataset of US States for demonstration
tibble(state_name = state.name, area = state.area) %>%
  # sort states reverse-alphabetically
  arrange(desc(state_name))
```

```
## # A tibble: 50 x 2
##    state_name      area
##    <chr>          <dbl>
##  1 Wyoming        97914
##  2 Wisconsin      56154
##  3 West Virginia  24181
##  4 Washington     68192
##  5 Virginia       40815
##  6 Vermont         9609
##  7 Utah           84916
##  8 Texas         267339
##  9 Tennessee      42244
## 10 South Dakota   77047
## # i 40 more rows
```

It's not a bad idea to restart your R session here. Make sure to save your work, but a clean `Environment` is great when we're shifting topics.

You can accomplish this by going to `Session > Restart R` in the menu.

Then, we want to make sure to re-load our packages and import our data.

```
# The packages we're using
library(tidyverse)
library(ca)
```

```
## Warning: package 'ca' was built under R version 4.3.1
```

```
# The dataset
berry_data <- read_csv("data/clt-berry-data.csv")
```

```
## Rows: 7507 Columns: 92
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (7): Start Time (UTC), End Time (UTC), Sample Name, verbal_likes, verba...
## dbl (83): Subject Code, Participant Name, Serving Position, Sample Identifie...
## lgl  (2): Gender, Age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Chapter 5

Untidy Data Analysis

5.1 Correspondence Analysis Overview

Remember what we said about most of the work of analysis being data wrangling? Now that we're about 2/3 of the way through this workshop, it's finally time to talk **data analysis**. The shape you need to wrangle your data *into* is determined by the analysis you want to run. Today, we have a bunch of **categorical data** and we want to understand the overall patterns within it. Which berries are **similar** and which are **different** from each other? Which sensory attributes are driving those differences?

One analysis well-suited to answer these questions for CATA and other categorical data is **Correspondence Analysis**. It's a singular value decomposition-based dimensionality-reduction method, so it's similar to Principal Component Analysis, but can be used when individual observations don't have numerical responses, or when the distances between those numbers (e.g., rankings) aren't meaningful.

We're using Correspondence Analysis as an example, so you can see the tidyverse in action! This is not intended to be a statistics lesson. If you want to understand the theoretical and mathematical underpinnings of Correspondence Analysis and its variations, we recommend Michael Greenacre's book *Correspondence Analysis in Practice*. It was written by the author of the R package we'll be using today (`ca`), and includes R code in the appendix.

5.1.1 The CA package

Let's take a look at this `ca` package!

```
library(ca)

?ca
```

The first thing you'll see is that it wants, specifically, a data frame or matrix as `obj`. We'll be ignoring the `formula` option, because a frequency table stored as a matrix is more flexible: you can use it in other functions like those in the `FactoMineR` package.

A **frequency table** or **contingency table** is one way of numerically representing multiple categorical variables measured on the same set of observations. Each row represents one group of observations, each column represents one level of a given categorical variable, and the cells are filled with the number of observations in that group that fall into that level of the categorical variable.

The help file shows us an example of a frequency table included in base R, so we can take a look at the shape we need to match:

```
data("author")
str(author)
```

```
##  num [1:12, 1:26] 550 515 590 557 589 541 517 592 576 557 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:12] "three daughters (buck)" "drifters (michener)" "lost world (clar
##   ..$ : chr [1:26] "a" "b" "c" "d" ...
```

```
head(author)
```

```
##                                a   b   c   d    e   f   g   h   i j  k   l   m
## three daughters (buck)       550 116 147 374 1015 131 131 493 442 2 52 302 159
## drifters (michener)          515 109 172 311  827 167 136 376 432 8 61 280 146
## lost world (clark)           590 112 181 265  940 137 119 419 514 6 46 335 176
## east wind (buck)             557 129 128 343  996 158 129 571 555 4 76 291 247
## farewell to arms (hemingway) 589  72 129 339  866 108 159 449 472 7 59 264 158
## sound and fury 7 (faulkner)  541 109 136 228  763 126 129 401 520 5 72 280 209
##                                n   o   p q   r   s   t   u   v   w  x   y  z
## three daughters (buck)       534 516 115 4 409 467 632 174  66 155  5 150  3
## drifters (michener)          470 561 140 4 368 387 632 195  60 156 14 137  5
## lost world (clark)           403 505 147 8 395 464 670 224 113 146 13 162 10
## east wind (buck)             479 509  92 3 413 533 632 181  68 187 10 184  4
## farewell to arms (hemingway) 504 542  95 0 416 314 691 197  64 225  1 155  2
## sound and fury 7 (faulkner)  471 589  84 2 324 454 672 247  71 160 11 280  1
```

Now let's take a look at the **tidy** CATA data we need to convert into a frequency table:

```
berry_data %>%
  select(`Sample Name`, `Subject Code`, starts_with("cata_"))
```

```
## # A tibble: 7,507 x 38
##    `Sample Name` `Subject Code` cata_appearance_unevenc~1 cata_appearance_miss~2
##    <chr>                  <dbl>                     <dbl>                   <dbl>
##  1 raspberry 6             1001                         0                       1
##  2 raspberry 5             1001                         0                       0
##  3 raspberry 2             1001                         0                       0
##  4 raspberry 3             1001                         0                       0
##  5 raspberry 4             1001                         1                       1
##  6 raspberry 1             1001                         0                       0
##  7 raspberry 6             1002                         0                       0
##  8 raspberry 5             1002                         1                       0
##  9 raspberry 2             1002                         1                       0
## 10 raspberry 3             1002                         1                       0
## # i 7,497 more rows
## # i abbreviated names: 1: cata_appearance_unevencolor,
## #   2: cata_appearance_misshapen
## # i 34 more variables: cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>,
## #   cata_appearance_notfresh <dbl>, cata_appearance_fresh <dbl>,
## #   cata_appearance_goodshape <dbl>, cata_appearance_goodquality <dbl>, ...
```

This is a pretty typical way for data collection software to save data from categorical questions like CATA and ordinal questions like JAR: one column per attribute. It's also, currently, a `tibble`, which is not in the list of objects that the `ca()` function will take.

We need to **untidy** our data to do this analysis, which is pretty common, but the `tidyverse` functions are still going to make it much easier than trying to reshape the data in base R. The makers of the packages have included some helpful functions for converting data *out of the tidyverse*, which we'll cover in a few minutes.

5.2 Categorical, Character, Binomial, Binary, and Count data

Now, you might notice that, for categorical data, there are an awful lot of numbers in both of these tables. Without giving a whole statistics lesson, I want to take a minute to stress that the **data type** in R or another statistical software (`logical > integer > numeric > character`) is **not** necessarily

the same as your statistical **level of measurement** (categorical/numerical or nominal/ordinal/interval/ratio).

It is your job as a sensory scientist and data analyst to understand what kind of data you have *based on how it was collected*, and select appropriate analyses accordingly.

In `berry_data$cata_*`, the data is represented as a 1 if that panelist checked that attribute for that sample, and a 0 otherwise. This could also be represented as a logical `TRUE` or `FALSE`, but the 0 and 1 convention makes binomial statistics most common, so you'll see it a lot.

You can also pretty easily convert between binomial/binary data stored as `numeric` 0s and 1s and `logical` data.

```
testlogic <- c(TRUE,FALSE,FALSE)
testlogic
```

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

```
class(testlogic) #We start with logical data
```

```
## [1] "logical"
```

```
testnums <- as.numeric(testlogic) #as.numeric() turns FALSE to 0 and TRUE to 1
testnums
```

```
## [1] 1 0 0
```

```
class(testnums) #Now it's numeric
```

```
## [1] "numeric"
```

```
as.logical(testnums) #We can turn numeric to logical data the same way
```

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

```
as.logical(c(0,1,2,3)) #Be careful with non-binary data, though!
```

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

You may also have your categorical data stored as a `character` type, namely if the categories are **mutually exclusive** or if you have free response data where there were not a finite/fixed set of options. The latter can quickly become thorny to deal with, because every single respondent could have their own unique categories that don't align with any others. Julia Silge and David Robinson's book *Text Mining with R* is a good primer for tidying and doing basic analysis of text data.

The first type of `character` data (with a limited number of mutually exclusive categories), thankfully, is much easier to deal with. We could turn the `berry` type variable into four separate **indicator variables** with 0s and 1s, if the analysis called for it, using `pivot_wider()`. Because we're turning *one column* into *multiple columns*, we're making the data wider, even though we don't actually want to make it any shorter.

```
berry_data %>%
  mutate(presence = 1) %>%
  pivot_wider(names_from = berry,
              values_from = presence, values_fill = 0) %>%
  #The rest is just for visibility:
  select(ends_with("berry"), `Sample Name`, everything()) %>%
  arrange(lms_overall)
```

```
## # A tibble: 7,507 x 95
##    cata_taste_berry raspberry blackberry blueberry strawberry `Sample Name`
##               <dbl>     <dbl>      <dbl>     <dbl>      <dbl> <chr>
##  1                0         1          0         0          0 raspberry 2
##  2                0         0          0         0          1 Strawberry2
##  3                0         1          0         0          0 raspberry 2
##  4                0         1          0         0          0 raspberry 1
##  5                0         1          0         0          0 raspberry 5
##  6                0         0          1         0          0 Blackberry 1
##  7                0         0          1         0          0 Blackberry 2
##  8                0         0          0         1          0 Blueberry 2
##  9                0         0          0         1          0 Blueberry 4
## 10                0         0          0         0          1 Strawberry1
## # i 7,497 more rows
## # i 89 more variables: `Subject Code` <dbl>, `Participant Name` <dbl>,
## #   Gender <lgl>, Age <lgl>, `Start Time (UTC)` <chr>, `End Time (UTC)` <chr>,
## #   `Serving Position` <dbl>, `Sample Identifier` <dbl>,
## #   `9pt_appearance` <dbl>, pre_expectation <dbl>, jar_color <dbl>,
## #   jar_gloss <dbl>, jar_size <dbl>, cata_appearance_unevencolor <dbl>,
## #   cata_appearance_misshapen <dbl>, cata_appearance_creased <dbl>, ...
```

And we can see here that `pivot_wider()` increased the number of columns without decreasing the number of rows. By default, it will only combine rows

where every column other than the `names_from` and `values_from` columns are identical.

It's often possible to convert between data types by changing the scope of your focus. **Summary tables** of categorical data can include numerical statistics (and this might give you a clue as to which `tidyverse` verb we're going to be using the most in this chapter). The most common kind of summary statistic for categorical variables is the **count** or **frequency**, which is where frequency tables get their name.

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(starts_with("cata_"), sum))
```

```
## # A tibble: 23 x 37
##    `Sample Name` cata_appearance_unevencolor cata_appearance_misshapen
##    <chr>                               <dbl>                     <dbl>
##  1 Blackberry 1                           28                        67
##  2 Blackberry 2                           32                        72
##  3 Blackberry 3                           25                        50
##  4 Blackberry 4                           46                       114
##  5 Blackberry 5                           32                       144
##  6 Blueberry 1                            46                        13
##  7 Blueberry 2                            48                        25
##  8 Blueberry 3                            34                        37
##  9 Blueberry 4                            29                        26
## 10 Blueberry 5                            22                        35
## # i 13 more rows
## # i 34 more variables: cata_appearance_creased <dbl>,
## #   cata_appearance_seedy <dbl>, cata_appearance_bruised <dbl>,
## #   cata_appearance_notfresh <dbl>, cata_appearance_fresh <dbl>,
## #   cata_appearance_goodshape <dbl>, cata_appearance_goodquality <dbl>,
## #   cata_appearance_none <dbl>, cata_taste_floral <dbl>,
## #   cata_taste_berry <dbl>, cata_taste_green <dbl>, ...
```

Note that the there are some attributes with NA counts. If we reran the analysis with `na.rm = TRUE`, we'd see that these attributes have zero citations for the berries that were `NA` before. This is because some attributes were only relevant for some of the berries. You will have to think about whether and how to include any of these variables in your analysis.

For now, we'll just drop those terms.

```
berry_data %>%
  group_by(`Sample Name`) %>%
  summarize(across(starts_with("cata_"), sum)) %>%
```

```
  select(where(~ none(.x, is.na))) -> berry_tidy_contingency

berry_tidy_contingency
```

```
## # A tibble: 23 x 14
##    `Sample Name` cata_appearance_unevencolor cata_appearance_misshapen
##    <chr>                               <dbl>                     <dbl>
##  1 Blackberry 1                           28                        67
##  2 Blackberry 2                           32                        72
##  3 Blackberry 3                           25                        50
##  4 Blackberry 4                           46                       114
##  5 Blackberry 5                           32                       144
##  6 Blueberry 1                            46                        13
##  7 Blueberry 2                            48                        25
##  8 Blueberry 3                            34                        37
##  9 Blueberry 4                            29                        26
## 10 Blueberry 5                            22                        35
## # i 13 more rows
## # i 11 more variables: cata_appearance_notfresh <dbl>,
## #   cata_appearance_fresh <dbl>, cata_appearance_goodquality <dbl>,
## #   cata_appearance_none <dbl>, cata_taste_floral <dbl>,
## #   cata_taste_berry <dbl>, cata_taste_grassy <dbl>,
## #   cata_taste_fermented <dbl>, cata_taste_fruity <dbl>,
## #   cata_taste_earthy <dbl>, cata_taste_none <dbl>
```

5.3 Untidy Analysis

We have our contingency table now, right? That wasn't so hard! Let's do CA!

```
berry_tidy_contingency %>%
  ca()
```

To explain why this error happens, we're going to need to talk a bit more about base R, since `ca` and many other data analysis packages aren't part of the `tidyverse`. Specifically, we need to talk about matrices and row names.

5.3.1 Untidying Data

Let's take another look at the ways the example `author` dataset are different from our data.

```
class(author)
```

```
## [1] "matrix" "array"
```

```
dimnames(author)
```

```
## [[1]]
##  [1] "three daughters (buck)"       "drifters (michener)"
##  [3] "lost world (clark)"           "east wind (buck)"
##  [5] "farewell to arms (hemingway)" "sound and fury 7 (faulkner)"
##  [7] "sound and fury 6 (faulkner)"  "profiles of future (clark)"
##  [9] "islands (hemingway)"          "pendorric 3 (holt)"
## [11] "asia (michener)"              "pendorric 2 (holt)"
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
class(berry_tidy_contingency)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dimnames(berry_tidy_contingency)
```

```
## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
## [16] "16" "17" "18" "19" "20" "21" "22" "23"
##
## [[2]]
##  [1] "Sample Name"                 "cata_appearance_unevencolor"
##  [3] "cata_appearance_misshapen"   "cata_appearance_notfresh"
##  [5] "cata_appearance_fresh"       "cata_appearance_goodquality"
##  [7] "cata_appearance_none"        "cata_taste_floral"
##  [9] "cata_taste_berry"            "cata_taste_grassy"
## [11] "cata_taste_fermented"        "cata_taste_fruity"
## [13] "cata_taste_earthy"           "cata_taste_none"
```

The example data we're trying to replicate is a `matrix`. This is another kind of tabular data, similar to a `tibble` or `data.frame`. The thing that sets matrices apart is that *every single cell in a matrix has the same data type*. This is a property that a lot of matrix algebra relies upon, like the math underpinning Correspondence Analysis.

Because they're tabular, it's very easy to turn a `data.frame` *into* a `matrix`, like the `ca()` function alludes to in the help files.

```
as.matrix(berry_tidy_contingency)
```

```
##       Sample Name    cata_appearance_unevencolor cata_appearance_misshapen
##  [1,] "Blackberry 1" " 28"                       " 67"
##  [2,] "Blackberry 2" " 32"                       " 72"
##  [3,] "Blackberry 3" " 25"                       " 50"
##  [4,] "Blackberry 4" " 46"                       "114"
##  [5,] "Blackberry 5" " 32"                       "144"
##  [6,] "Blueberry 1"  " 46"                       " 13"
##  [7,] "Blueberry 2"  " 48"                       " 25"
##  [8,] "Blueberry 3"  " 34"                       " 37"
##  [9,] "Blueberry 4"  " 29"                       " 26"
## [10,] "Blueberry 5"  " 22"                       " 35"
## [11,] "Blueberry 6"  " 43"                       " 45"
## [12,] "Strawberry1"  "192"                       " 18"
## [13,] "Strawberry2"  "213"                       " 51"
## [14,] "Strawberry3"  "139"                       " 24"
## [15,] "Strawberry4"  "144"                       " 25"
## [16,] "Strawberry5"  "160"                       " 32"
## [17,] "Strawberry6"  "197"                       " 87"
## [18,] "raspberry 1"  "126"                       " 52"
## [19,] "raspberry 2"  " 99"                       " 58"
## [20,] "raspberry 3"  "184"                       "100"
## [21,] "raspberry 4"  "112"                       " 56"
## [22,] "raspberry 5"  "121"                       " 43"
## [23,] "raspberry 6"  "116"                       " 45"
##       cata_appearance_notfresh cata_appearance_fresh
##  [1,] " 27"                    "191"
##  [2,] " 37"                    "197"
##  [3,] "  9"                    "222"
##  [4,] " 38"                    "170"
##  [5,] " 16"                    "197"
##  [6,] " 20"                    "223"
##  [7,] " 30"                    "203"
##  [8,] " 40"                    "196"
##  [9,] " 24"                    "219"
## [10,] " 39"                    "209"
## [11,] " 30"                    "198"
## [12,] " 99"                    "129"
## [13,] "155"                    " 65"
## [14,] " 55"                    "185"
## [15,] " 73"                    "156"
## [16,] " 80"                    "168"
## [17,] "226"                    " 38"
## [18,] " 60"                    "228"
```

```
## [19,] " 57"                    "216"
## [20,] "111"                    "145"
## [21,] " 75"                    "199"
## [22,] " 59"                    "219"
## [23,] " 61"                    "218"
##       cata_appearance_goodquality cata_appearance_none cata_taste_floral
##  [1,] "172"                        " 3"                "48"
##  [2,] "170"                        " 2"                "32"
##  [3,] "204"                        " 4"                "56"
##  [4,] "152"                        " 1"                "56"
##  [5,] "161"                        " 3"                "56"
##  [6,] "208"                        " 3"                "54"
##  [7,] "202"                        " 3"                "24"
##  [8,] "189"                        " 1"                "64"
##  [9,] "198"                        " 3"                "57"
## [10,] "193"                        " 2"                "49"
## [11,] "201"                        " 5"                "61"
## [12,] " 92"                        " 0"                "56"
## [13,] " 63"                        " 4"                "36"
## [14,] "159"                        " 4"                "39"
## [15,] "121"                        " 1"                "39"
## [16,] "141"                        " 3"                "34"
## [17,] " 43"                        " 0"                "46"
## [18,] "206"                        " 1"                "48"
## [19,] "190"                        " 4"                "63"
## [20,] "127"                        " 1"                "74"
## [21,] "181"                        " 7"                "60"
## [22,] "196"                        " 3"                "67"
## [23,] "169"                        "12"                "60"
##       cata_taste_berry cata_taste_grassy cata_taste_fermented cata_taste_fruity
##  [1,] "155"            " 55"             "72"                " 88"
##  [2,] "127"            " 79"             "70"                " 71"
##  [3,] "161"            " 66"             "64"                "107"
##  [4,] "167"            " 53"             "21"                "129"
##  [5,] "183"            " 61"             "47"                "115"
##  [6,] "156"            " 69"             "22"                "116"
##  [7,] "182"            " 48"             "17"                "134"
##  [8,] "172"            " 73"             "33"                "111"
##  [9,] "171"            " 44"             "35"                "125"
## [10,] "180"            " 56"             "21"                "120"
## [11,] "218"            " 45"             "19"                "155"
## [12,] "204"            " 55"             "53"                "172"
## [13,] "144"            "113"             "59"                "105"
## [14,] "171"            " 77"             "37"                "148"
## [15,] "137"            "125"             "35"                "111"
## [16,] "201"            " 81"             "53"                "164"
```

```
## [17,] " 84"            "113"             "66"                " 82"
## [18,] "173"            "102"             "36"                "122"
## [19,] "217"            " 78"             "31"                "156"
## [20,] "266"            " 29"             "20"                "213"
## [21,] "198"            " 67"             "28"                "151"
## [22,] "231"            " 59"             "20"                "175"
## [23,] "268"            " 36"             "15"                "212"
##       cata_taste_earthy cata_taste_none
##  [1,] " 92"             "13"
##  [2,] "102"             "24"
##  [3,] " 82"             "13"
##  [4,] " 51"             "36"
##  [5,] " 72"             "19"
##  [6,] " 67"             "19"
##  [7,] " 35"             "25"
##  [8,] " 67"             "11"
##  [9,] " 55"             " 9"
## [10,] " 60"             "24"
## [11,] " 41"             " 6"
## [12,] " 64"             "13"
## [13,] "101"             "24"
## [14,] " 57"             "26"
## [15,] "100"             "24"
## [16,] " 69"             "11"
## [17,] "109"             "23"
## [18,] " 90"             "19"
## [19,] "100"             "21"
## [20,] " 48"             "13"
## [21,] " 82"             "17"
## [22,] " 68"             "19"
## [23,] " 47"             "14"
```

This is what the `ca()` function does for us when we give it a `data.frame` or `tibble`. It follows the hierarchy of data types, so you'll see that now every single number is surrounded by quotation marks now (“). It's been converted into the least-restrictive data type in `berry_tidy_contingency`, which is `character`.

Unfortunately, you can't do math on `character` vectors.

```
1 + 2
"1" + "2"
```

It's important to know which row corresponds to which berry, though, so we want to keep the `Sample Name` column *somehow*! This is where `rownames` come in handy, which the `author` data has but our `berry_tidy_contingency` doesn't.

The `tidyverse` doesn't really use row names (it is technically *possible* to have a tibble with `rownames`, but extremely error-prone). The theory is that whatever information you *could* use as `rownames` could be added as another column, and that you may have multiple variables whose combined levels define each row (say the sample and the participant IDs) rather than needing a single less-informative ID unique to each row.

Row names are important to numeric matrices, though, because we can't do math on a matrix of character variables!

The `tidyverse` provides a handy function for this, `column_to_rownames()`:

```
berry_tidy_contingency %>%
  column_to_rownames("Sample Name") -> berry_contingency_df

class(berry_contingency_df)
```

```
## [1] "data.frame"
```

```
dimnames(berry_contingency_df)
```

```
## [[1]]
##  [1] "Blackberry 1" "Blackberry 2" "Blackberry 3" "Blackberry 4" "Blackberry 5"
##  [6] "Blueberry 1"  "Blueberry 2"  "Blueberry 3"  "Blueberry 4"  "Blueberry 5"
## [11] "Blueberry 6"  "Strawberry1"  "Strawberry2"  "Strawberry3"  "Strawberry4"
## [16] "Strawberry5"  "Strawberry6"  "raspberry 1"  "raspberry 2"  "raspberry 3"
## [21] "raspberry 4"  "raspberry 5"  "raspberry 6"
##
## [[2]]
##  [1] "cata_appearance_unevencolor" "cata_appearance_misshapen"
##  [3] "cata_appearance_notfresh"    "cata_appearance_fresh"
##  [5] "cata_appearance_goodquality" "cata_appearance_none"
##  [7] "cata_taste_floral"           "cata_taste_berry"
##  [9] "cata_taste_grassy"           "cata_taste_fermented"
## [11] "cata_taste_fruity"           "cata_taste_earthy"
## [13] "cata_taste_none"
```

Note that you have to double-quote (" ") column names for `column_to_rownames()`. No idea why. I just do what `?column_to_rownames` tells me.

`berry_contingency_df` is all set for the `ca()` function now, but if you run into any functions (like many of those in `FactoMineR` and other packages) that need matrices, you can always use `as.matrix()` on the results of `column_to_rownames()`.

`column_to_rownames()` will almost always be the cleanest way to untidy your data, but there are some other functions that may be handy if you need a

different data format, like a vector. You already know about $-subsetting, but you can also use `pull()` to pull one column out of a `tibble` as a vector using tidyverse syntax, so it fits easily at the end or in the middle of a series of piped steps.

```
berry_data %>%
  pivot_longer(starts_with("cata_"),
               names_to = "attribute",
               values_to = "presence") %>%
  filter(presence == 1) %>%
  count(attribute) %>%
  #Arranges the highest-cited CATA terms first
  arrange(desc(n)) %>%
  #Pulls the attribute names as a vector, in the order above
  pull(attribute)
```

```
##  [1] "cata_appearance_fresh"       "cata_taste_berry"
##  [3] "cata_appearance_goodquality" "cata_appearance_goodshape"
##  [5] "cata_taste_fruity"           "cata_appearance_goodcolor"
##  [7] "cata_appearance_unevencolor" "cata_taste_earthy"
##  [9] "cata_taste_grassy"           "cata_appearance_notfresh"
## [11] "cata_taste_citrus"           "cata_appearance_misshapen"
## [13] "cata_taste_floral"           "cata_appearance_bruised"
## [15] "cata_taste_fermented"        "cata_appearance_goodshapre"
## [17] "cata_appearance_seedy"       "cata_taste_peachy"
## [19] "cata_taste_none"             "cata_taste_tropical"
## [21] "cata_taste_candy"            "cata_taste_green"
## [23] "cata_appearance_creased"     "cata_taste_piney"
## [25] "cata_taste_lemon"            "cata_taste_grapey"
## [27] "cata_taste_cherry"           "cata_appearane_bruised"
## [29] "cata_taste_melon"            "cata_taste_grape"
## [31] "cata_taste_clove"            "cata_taste_caramel"
## [33] "cata_appearance_none"        "cata_appearane_creased"
## [35] "cata_taste_minty"            "cata_taste_cinnamon"
```

In summary: - Reshape your data in the `tidyverse` and then change it as needed for analysis. - If you need a `data.frame` or `matrix` with `rownames` set, use `column_to_rownames()`. - Use `as.matrix()` carefully, only on tabular data with **all the same data type**. - `as.matrix()` may not work on `tibbles` *at all* in older versions of the `tidyverse`, so it's always safest to go `tibble` -> `data.frame` -> `matrix`. - If you need a vector, use `pull()`.

5.3.2 Data Analysis

Okay, are you ready? Our data is *finally* in the shape and format we needed. You're ready to run multivariate statistics in R.

Ready?

Are you sure?

```
ca(berry_contingency_df) -> berry_ca_res
```

Yep, that's it.

There are other options you can read about in the help files, if you need a more sophisticated analysis, but most of the time, if I need to change something, it's with the way I'm arranging my data *before* analysis rather than fundamentally changing the `ca()` call.

In general, I find it easiest to do all of the filtering and selecting *on the tibble* so I can use the handy `tidyverse` functions, before I untidy the data, but you can also include extra rows or columns in your contingency table (as long as they're also numbers!!) and then tell the `ca()` function which columns are active and supplemental. This may be an easier way to compare a few different analyses with different variables or levels of summarization, rather than having to make a bunch of different contingency matrices for each.

```
berry_data %>%
  select(`Sample Name`, contains(c("cata_", "9pt_", "lms_", "us_"))) %>%
  summarize(across(contains("cata_"), ~ sum(.x, na.rm = TRUE)),
            across(contains(c("9pt_","lms_","us_")), ~ mean(.x, na.rm = TRUE)), .by = 
  column_to_rownames("Sample Name") %>%
  #You have to know the NUMERIC indices to do it this way.
  ca(supcol = 37:51)
```

```
##
##  Principal inertias (eigenvalues):
##            1        2        3        4        5        6        7
## Value      0.295123 0.148008 0.109166 0.033574 0.016535 0.010195 0.007532
## Percentage 46.03%   23.09%   17.03%   5.24%    2.58%    1.59%    1.17%
##            8        9        10       11       12       13       14
## Value      0.00576  0.003221 0.002931 0.00244  0.001342 0.001226 0.001024
## Percentage 0.9%     0.5%     0.46%    0.38%    0.21%    0.19%    0.16%
##            15       16       17       18       19       20       21
## Value      0.000914 0.000631 0.000562 0.00036  0.000227 0.000155 0.000123
## Percentage 0.14%    0.1%     0.09%    0.06%    0.04%    0.02%    0.02%
##            22
## Value      4.8e-05
```

```
## Percentage 0.01%
##
##
##  Rows:
##         raspberry 6 raspberry 5 raspberry 2 raspberry 3 raspberry 4 raspberry 1
## Mass       0.047104    0.047309    0.046566    0.048898    0.046797    0.046745
## ChiDist    0.701154    0.597664    0.544036    0.778030    0.603364    0.588880
## Inertia    0.023157    0.016899    0.013782    0.029599    0.017036    0.016210
## Dim. 1     0.539265    0.541430    0.498086    0.695131    0.597405    0.528684
## Dim. 2     0.936377    0.758876    0.559180    0.455674    0.602968    0.588395
##         Blackberry 4 Blackberry 2 Blackberry 1 Blackberry 3 Blackberry 5
## Mass        0.038160     0.039979     0.040569     0.041850     0.039672
## ChiDist     0.987037     1.105960     1.166375     1.113777     0.910491
## Inertia     0.037177     0.048901     0.055191     0.051915     0.032888
## Dim. 1     -1.589369    -1.864827    -1.959700    -1.884164    -1.525017
## Dim. 2     -0.699024    -0.999143    -1.019725    -0.767436    -0.525974
##         Blueberry 1 Blueberry 4 Blueberry 2 Blueberry 3 Blueberry 5 Blueberry 6
## Mass       0.040774    0.041235    0.040979    0.042209    0.041363    0.043234
## ChiDist    0.703408    0.648461    0.608480    0.638924    0.636871    0.635531
## Inertia    0.020174    0.017339    0.015172    0.017231    0.016777    0.017462
## Dim. 1    -0.284508   -0.300322   -0.151757   -0.294232   -0.287190   -0.261262
## Dim. 2     1.230837    1.164323    1.114938    1.095328    1.124883    1.167484
##         Strawberry4 Strawberry1 Strawberry2 Strawberry6 Strawberry3 Strawberry5
## Mass       0.043875    0.046643    0.043849    0.043824    0.043106    0.045259
## ChiDist    0.711993    1.037185    0.879538    1.102282    0.613945    0.636763
## Inertia    0.022242    0.050176    0.033921    0.053247    0.016248    0.018351
## Dim. 1     0.915361    1.082511    1.092504    1.155009    0.762252    0.816533
## Dim. 2    -0.968446   -1.424927   -1.542342   -1.688415   -0.549006   -0.794523
##
##
##  Columns:
##         cata_appearance_unevencolor cata_appearance_misshapen
## Mass                       0.056074                  0.031240
## ChiDist                    0.624470                  0.632738
## Inertia                    0.021867                  0.012507
## Dim. 1                     0.954959                 -0.541613
## Dim. 2                    -0.794647                 -0.478410
##         cata_appearance_creased cata_appearance_seedy cata_appearance_bruised
## Mass                   0.006663              0.015838                0.027986
## ChiDist                1.107665              1.738786                1.105926
## Inertia                0.008175              0.047884                0.034228
## Dim. 1                 1.443844              1.752834                1.463469
## Dim. 2                -0.717471             -2.632660               -1.526336
##         cata_appearance_notfresh cata_appearance_fresh
## Mass                    0.036417              0.107406
## ChiDist                 0.754184              0.277362
```

```
## Inertia                0.020714             0.008263
## Dim. 1                 0.889423            -0.307694
## Dim. 2                -1.022464             0.452318
##         cata_appearance_goodshape cata_appearance_goodquality
## Mass                     0.093311                    0.095797
## ChiDist                  0.550292                    0.296273
## Inertia                  0.028257                    0.008409
## Dim. 1                   0.660358                   -0.332910
## Dim. 2                   0.965680                    0.529804
##         cata_appearance_none cata_taste_floral cata_taste_berry
## Mass                0.001794          0.030215         0.106766
## ChiDist             0.789222          0.223386         0.192937
## Inertia             0.001117          0.001508         0.003974
## Dim. 1             -0.037264         -0.093179        -0.001625
## Dim. 2              0.690895          0.230931         0.291399
##         cata_taste_green cata_taste_grassy cata_taste_fermented
## Mass            0.007945          0.040595             0.022399
## ChiDist         1.679549          0.354579             0.501741
## Inertia         0.022411          0.005104             0.005639
## Dim. 1          1.022314          0.128887            -0.319566
## Dim. 2          1.693699         -0.500564            -1.022115
##         cata_taste_tropical cata_taste_fruity cata_taste_citrus
## Mass                0.010687          0.078985          0.032471
## ChiDist             1.669061          0.225740          0.650213
## Inertia             0.029771          0.004025          0.013728
## Dim. 1              1.067362          0.148197          0.820933
## Dim. 2              1.699332          0.249386          0.714076
##         cata_taste_earthy cata_taste_candy cata_taste_none
## Mass             0.042517         0.008278        0.010841
## ChiDist          0.298491         1.656958        0.405144
## Inertia          0.003788         0.022727        0.001779
## Dim. 1          -0.072856         1.067984       -0.127751
## Dim. 2          -0.463882         1.685638       -0.298570
##         cata_appearane_bruised cata_appearance_goodshapre
## Mass                 0.004869                   0.020169
## ChiDist              2.101907                   2.040213
## Inertia              0.021513                   0.083953
## Dim. 1              -3.226349                  -3.308097
## Dim. 2              -2.103976                  -2.154730
##         cata_appearance_goodcolor cata_taste_cinnamon cata_taste_lemon
## Mass                     0.058944            0.000948        0.006023
## ChiDist                  1.107046            2.121291        2.088847
## Inertia                  0.072239            0.004267        0.026278
## Dim. 1                  -1.735961           -3.328963       -3.317694
## Dim. 2                   0.698509           -2.141693       -2.201850
##         cata_taste_clove cata_taste_minty cata_taste_grape
```

```
## Mass            0.002383         0.001256         0.003998
## ChiDist         2.103524         2.036370         2.072312
## Inertia         0.010546         0.005207         0.017169
## Dim. 1         -3.313433        -3.286100        -3.318432
## Dim. 2         -2.132266        -2.105219        -2.191008
##         cata_appearane_creased cata_taste_piney cata_taste_peachy
## Mass                 0.001281         0.006407          0.011353
## ChiDist              1.920294         1.831149          1.144327
## Inertia              0.004725         0.021483          0.014867
## Dim. 1              -0.489183        -0.500874          0.309423
## Dim. 2               2.954184         3.007418          0.883482
##         cata_taste_caramel cata_taste_grapey cata_taste_melon cata_taste_cherry
## Mass              0.001896          0.005843         0.004767          0.005638
## ChiDist           1.794449          1.668738         1.711006          1.718018
## Inertia           0.006107          0.016271         0.013955          0.016641
## Dim. 1            1.768170          1.775356         1.792723          1.761788
## Dim. 2           -2.946463         -2.971754        -3.038976         -2.919189
##         9pt_appearance (*) 9pt_overall (*) 9pt_taste (*) 9pt_texture (*)
## Mass                    NA              NA            NA              NA
## ChiDist           0.130697        0.102893      0.113246        0.096145
## Inertia                 NA              NA            NA              NA
## Dim. 1           -0.186168       -0.075320     -0.076711       -0.101059
## Dim. 2            0.143741        0.095335      0.129310        0.066306
##         9pt_aroma (*) lms_appearance (*) lms_overall (*) lms_taste (*)
## Mass               NA                 NA              NA            NA
## ChiDist           NaN           0.463025        0.502031      0.687212
## Inertia            NA                 NA              NA            NA
## Dim. 1            NaN          -0.588691        0.019500      0.024678
## Dim. 2            NaN           0.687830        0.735511      1.065640
##         lms_texture (*) lms_aroma (*) us_appearance (*) us_overall (*)
## Mass                NA            NA                NA             NA
## ChiDist       0.331814           NaN          0.169497       0.128866
## Inertia             NA            NA                NA             NA
## Dim. 1        -0.198320           NaN         -0.234714      -0.084274
## Dim. 2         0.436500           NaN          0.211381       0.195086
##         us_taste (*) us_texture (*) us_aroma (*)
## Mass              NA             NA           NA
## ChiDist     0.149586       0.111795          NaN
## Inertia           NA             NA           NA
## Dim. 1     -0.070870      -0.117917          NaN
## Dim. 2      0.225916       0.128712          NaN
```

5.3.3 Retidying Data

What does the `ca()` function actually give us?

```
berry_ca_res %>%
  str()
```

```
## List of 16
##  $ sv        : num [1:12] 0.2958 0.1659 0.133 0.0724 0.0647 ...
##  $ nd        : logi NA
##  $ rownames  : chr [1:23] "Blackberry 1" "Blackberry 2" "Blackberry 3" "Blackberry 4
##  $ rowmass   : num [1:23] 0.0392 0.0394 0.0412 0.0401 0.0429 ...
##  $ rowdist   : num [1:23] 0.364 0.399 0.376 0.379 0.468 ...
##  $ rowinertia: num [1:23] 0.00519 0.00625 0.00583 0.00575 0.00941 ...
##  $ rowcoord  : num [1:23, 1:12] -0.676 -0.461 -1.025 -0.476 -0.738 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:23] "Blackberry 1" "Blackberry 2" "Blackberry 3" "Blackberry 4" .
##   .. ..$ : chr [1:12] "Dim1" "Dim2" "Dim3" "Dim4" ...
##  $ rowsup    : logi(0)
##  $ colnames  : chr [1:13] "cata_appearance_unevencolor" "cata_appearance_misshapen"
##  $ colmass   : num [1:13] 0.0848 0.0473 0.0551 0.1625 0.1449 ...
##  $ coldist   : num [1:13] 0.618 0.599 0.755 0.285 0.31 ...
##  $ colinertia: num [1:13] 0.0324 0.017 0.0314 0.0132 0.0139 ...
##  $ colcoord  : num [1:13, 1:12] 1.9366 0.0589 2.4533 -0.9373 -0.9973 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:13] "cata_appearance_unevencolor" "cata_appearance_misshapen" "ca
##   .. ..$ : chr [1:12] "Dim1" "Dim2" "Dim3" "Dim4" ...
##  $ colsup    : logi(0)
##  $ N         : num [1:23, 1:13] 28 32 25 46 32 46 48 34 29 22 ...
##  $ call      : language ca.matrix(obj = as.matrix(obj))
##  - attr(*, "class")= chr "ca"
```

It's a list with many useful things. You can think of a list as kinda like a data frame, because each item has a **name** (like columns in data frames), except they can be any length/size/shape. It's *not* tabular, so you can't use [1,2] for indexing rows and columns, but you *can* use $ indexing if you know the name of the data you're after.

You're unlikely to need to worry about the specifics. Just remember that lists can be $-indexed.

There are a few things we may want out of the list that `ca()` gives us, and we can see descriptions of them in plain English by typing `?ca`. These are the ones we'll be using:

```
berry_ca_res$rowcoord #the standard coordinates of the row variable (berry)
```

```
##                       Dim1        Dim2        Dim3       Dim4         Dim5
## Blackberry 1 -0.676422039  1.55801517  0.02889809 -2.1207457  0.421035244
```

```
## Blackberry 2 -0.460988133  2.16220762  0.44500036 -0.6763648 -0.401344347
## Blackberry 3 -1.025439995  1.02861582  0.58873600 -1.5512728  0.149090611
## Blackberry 4 -0.475531553  0.73822421 -2.14355819  1.8024546 -1.054134662
## Blackberry 5 -0.737948756  1.55341431 -2.29069697 -0.3768377 -1.478964909
## Blueberry 1  -0.942208700 -0.16697306  1.34533986  0.9721393  0.341902827
## Blueberry 2  -0.891383712 -0.73549822  0.46896273  1.3567656 -0.058683995
## Blueberry 3  -0.693395862  0.30840131  0.49784640  0.2667007  1.452589890
## Blueberry 4  -1.062846799 -0.19768031  0.54029846 -0.6011662  1.453852344
## Blueberry 5  -0.903946503  0.01258835  0.29565501  1.3192008  1.250783134
## Blueberry 6  -0.911518977 -0.80933135 -0.45676687 -0.1166850  1.177305333
## Strawberry1   1.123787175 -1.29134698  0.30029870 -1.9371408 -0.269928992
## Strawberry2   2.186526847  0.39020970  0.46639303 -0.0688568 -0.454494991
## Strawberry3   0.216458365 -0.68408859  0.84099468  0.1817546 -1.474293484
## Strawberry4   0.779921416  0.31126088  1.56706081  1.0718960 -1.737282915
## Strawberry5   0.595885377 -0.65590705  0.58277734 -1.2198681 -0.852687372
## Strawberry6   2.890807350  1.51710457 -0.40231055  0.8846827  2.338783198
## raspberry 1   0.002784249  0.24770479  0.77952820  0.7647137 -0.914998126
## raspberry 2  -0.181932219 -0.05470297  0.05660525  0.2948372 -0.159554062
## raspberry 3   0.714829561 -1.32650761 -2.11426476 -0.3861240  0.086736116
## raspberry 4   0.041631118 -0.30336432 -0.10035713  0.3256547  0.412286985
## raspberry 5  -0.177484391 -0.92080171 -0.04037283  0.3511281 -0.003814396
## raspberry 6  -0.239619991 -1.64119492 -0.69498878 -0.3495728  0.259177191
##                      Dim6         Dim7        Dim8        Dim9       Dim10
## Blackberry 1  0.416339437 -0.087205574  0.55527540  0.66487134  0.7992141
## Blackberry 2  1.353241139 -0.143661403  0.31809116  0.82018151  1.3393069
## Blackberry 3 -0.154464899 -0.009161493  0.17303241 -0.07476922 -1.5859313
## Blackberry 4  0.604303775  1.121173149  1.61580941 -0.44455523 -1.3036853
## Blackberry 5 -0.652553200 -0.503141105 -0.95611885 -0.76061578 -0.2891770
## Blueberry 1  -0.708842996  0.275826588  0.50857290  0.94363142 -1.0496073
## Blueberry 2   3.082539058  0.538179530 -1.16634324 -0.43368257  1.1514695
## Blueberry 3  -1.525570006  0.767208280 -0.17612888 -1.96339520 -0.1572088
## Blueberry 4  -0.121844931  0.606785673 -0.26607669  0.49314835 -0.9894949
## Blueberry 5   0.820976326  0.475499585  1.30639341 -0.73014319  1.4665355
## Blueberry 6  -0.582502705 -0.563767160 -1.33239601 -1.94597487 -0.2792107
## Strawberry1   0.003367573  1.980558028  1.56631289 -0.17821445 -0.0413237
## Strawberry2   0.333203747 -1.237545485  0.45004439 -0.95562013  0.1394031
## Strawberry3   1.174373617  0.381298231  0.06195272  0.41063042 -2.2704383
## Strawberry4  -1.577928830 -0.329844134  0.71300614 -1.32879302  0.5226638
## Strawberry5   0.761511155  0.054789811 -1.58761812 -1.24962440  0.5416557
## Strawberry6   0.483756496  0.160111078 -0.57887026  0.34590501 -0.8335918
## raspberry 1  -0.901156816  0.498483211 -2.30064883  1.53815305  0.2120347
## raspberry 2  -1.041068949 -0.750258551  1.07509047  0.55799211  1.7404352
## raspberry 3  -0.577370181  1.175513729 -0.54472390  0.75571365  0.9693176
## raspberry 4  -0.435287990 -1.543719337 -0.11575622  1.57370723 -0.2755199
## raspberry 5  -0.678042187  0.444107106  0.28159320  1.09950507  0.3868386
## raspberry 6   0.697218086 -2.755894177  0.71155298  0.04135319 -0.5673104
```

```
##                    Dim11        Dim12
## Blackberry 1 -0.95983642 -0.009253265
## Blackberry 2  0.68731238  0.267060332
## Blackberry 3 -0.69717059 -1.168003875
## Blackberry 4 -0.22875124 -0.894559086
## Blackberry 5  0.74883806  1.156346478
## Blueberry 1  -0.33120184 -0.408429808
## Blueberry 2  -0.67185606 -0.519204536
## Blueberry 3  -0.29921710  1.327706362
## Blueberry 4   1.24943134  0.309664418
## Blueberry 5   0.54425299  1.516555388
## Blueberry 6  -1.24454179 -1.159488529
## Strawberry1   0.49720288  0.715403472
## Strawberry2  -2.84848835  1.293593154
## Strawberry3  -0.36073715  0.084475487
## Strawberry4   1.45255686 -0.678893230
## Strawberry5   1.25779330 -1.359994255
## Strawberry6   1.22987860 -0.669642985
## raspberry 1  -0.04656501  1.543167644
## raspberry 2  -0.10284611 -1.758554927
## raspberry 3  -0.40241340 -0.328363440
## raspberry 4  -0.43059527 -0.823009716
## raspberry 5  -0.56755742  0.383635858
## raspberry 6   1.44742716  1.252192647
```

```
berry_ca_res$colcoord #the standard coordinates of the column variable (attribute)
```

```
##                                   Dim1        Dim2        Dim3       Dim4
## cata_appearance_unevencolor  1.9366137 -1.17322492  0.32072486 -0.3691085
## cata_appearance_misshapen    0.0589441  2.05113772 -3.66060544  0.3531013
## cata_appearance_notfresh     2.4532684  0.16152199 -0.22489458  0.9952467
## cata_appearance_fresh       -0.9372699  0.02467928  0.44045357  0.1240773
## cata_appearance_goodquality -0.9973472  0.08087827  0.50595892  0.4849384
## cata_appearance_none        -0.7937563 -1.48244462 -0.16563013 -1.4180918
## cata_taste_floral           -0.2815016 -0.15570463 -0.66082635 -0.2243516
## cata_taste_berry            -0.3325407 -0.79022119 -0.46021047 -0.4352527
## cata_taste_grassy            0.5978264  1.20258057  1.48981642  1.2955302
## cata_taste_fermented         0.4853398  2.23871239  0.82509530 -3.9589988
## cata_taste_fruity           -0.1224089 -1.18256562 -0.46550861 -0.2273194
## cata_taste_earthy            0.3549054  1.43796995  0.85550318 -0.3491357
## cata_taste_none              0.1207429  0.96539923 -0.03486454  3.5345743
##                                   Dim5       Dim6         Dim7       Dim8
## cata_appearance_unevencolor -1.62005938 -0.3774736   0.30154586 -0.6335252
## cata_appearance_misshapen   -1.00603555 -0.2124960  -0.23927488 -1.0951641
## cata_appearance_notfresh     2.86890300  1.0263872  -0.05545059 -0.3850610
```

```
## cata_appearance_fresh       -0.10863293  0.1414962   0.04397075 -0.2062730
## cata_appearance_goodquality  0.52975420  0.3327724   0.34250974 -1.0731454
## cata_appearance_none         0.69017805  3.2913350 -17.00616406  1.3294917
## cata_taste_floral            1.49728372 -2.8193315   0.86406459  1.8336607
## cata_taste_berry             0.03456135  0.0893703  -0.21406092  0.2679412
## cata_taste_grassy           -0.96800100 -1.0235468  -0.48915622 -0.7017379
## cata_taste_fermented         0.10594194  1.8898178   1.03877228  0.3000662
## cata_taste_fruity           -0.14530173  0.2930711   0.08434588  0.4291713
## cata_taste_earthy           -0.01891776 -1.2013919  -0.99644609  1.3593894
## cata_taste_none             -2.05265315  3.7122409   1.34673737  4.8331118
##                                    Dim9      Dim10       Dim11       Dim12
## cata_appearance_unevencolor  1.22816392 -0.4084541 -0.62254913  0.31727241
## cata_appearance_misshapen    0.26709095 -0.1111936  0.11384684 -0.11854340
## cata_appearance_notfresh     0.06475031  0.2590012  0.65050805  0.36937808
## cata_appearance_fresh        0.81170689 -0.1700575  1.47757553  1.05774152
## cata_appearance_goodquality  0.47718892 -0.1537466 -1.36294181 -0.79924373
## cata_appearance_none         1.45853357 -7.0630004 -2.76411379  0.82437204
## cata_taste_floral            0.12638909 -2.2281270 -0.79295431  0.61533873
## cata_taste_berry            -0.98297215  1.2341189 -0.68568963  0.98439275
## cata_taste_grassy           -2.52321861 -0.6268628  0.30656609  0.11219358
## cata_taste_fermented        -0.78605241 -1.2096748 -0.26064110  0.09947975
## cata_taste_fruity           -0.65205090 -0.3340588  1.11961579 -1.89705359
## cata_taste_earthy            1.24888861  2.0332988  0.06165682 -1.23209187
## cata_taste_none              0.41381134 -0.8624509 -1.47668343  0.52466551
```

```
berry_ca_res$sv       #the singular value for each dimension
```

```
##  [1] 0.295838428 0.165865288 0.132976201 0.072363931 0.064690580 0.044807235
##  [7] 0.038031617 0.029774848 0.024918839 0.021300214 0.014415082 0.008117431
```

```
berry_ca_res$sv %>%   #which are useful in calculating the % inertia of each dimension
  {.^2 / sum(.^2)}
```

```
##  [1] 0.5920550825 0.1861075348 0.1196191706 0.0354239712 0.0283096845
##  [6] 0.0135815468 0.0097845873 0.0059972484 0.0042005733 0.0030691695
## [11] 0.0014056824 0.0004457488
```

```
#The column and row masses (in case you want to add your own supplementary variables
#after the fact):

#the row masses
berry_ca_res$rowmass
```

```
##  [1] 0.03919516 0.03935024 0.04121113 0.04008684 0.04287819 0.03938901
```

```
##  [7] 0.03783826 0.03985423 0.03857486 0.03915639 0.04136621 0.04446771
## [13] 0.04392494 0.04345972 0.04229666 0.04640614 0.04318834 0.04896488
## [19] 0.05001163 0.05160115 0.04780181 0.04962394 0.04935256
```

```
#the column masses
berry_ca_res$colmass
```

```
##  [1] 0.084825929 0.047259052 0.055090331 0.162479646 0.144917423 0.002713809
##  [7] 0.045708304 0.161510429 0.061409630 0.033883849 0.119485152 0.064317283
## [13] 0.016399163
```

The *main* results of CA are the row and column coordinates, which are in two matrices with the same columns. We can tidy them with the reverse of `column_to_rownames()`, `rownames_to_column()`, and then we can use `bind_rows()` to combine them.

```
berry_row_coords <- berry_ca_res$rowcoord %>%
  #rownames_to_column() works on data.frames, not matrices
  as.data.frame() %>%
  #This has to be the same for both to use bind_rows()!
  rownames_to_column("Variable")

#Equivalent to the above, and works on matrices
berry_col_coords <- berry_ca_res$colcoord %>%
  as_tibble(rownames = "Variable")

berry_ca_coords <- bind_rows(Berry = berry_row_coords,
                             Attribute = berry_col_coords,
                             .id = "Type")

berry_ca_coords
```

```
##         Type                  Variable         Dim1        Dim2        Dim3
## 1      Berry          Blackberry 1 -0.676422039  1.55801517  0.02889809
## 2      Berry          Blackberry 2 -0.460988133  2.16220762  0.44500036
## 3      Berry          Blackberry 3 -1.025439995  1.02861582  0.58873600
## 4      Berry          Blackberry 4 -0.475531553  0.73822421 -2.14355819
## 5      Berry          Blackberry 5 -0.737948756  1.55341431 -2.29069697
## 6      Berry           Blueberry 1 -0.942208700 -0.16697306  1.34533986
## 7      Berry           Blueberry 2 -0.891383712 -0.73549822  0.46896273
## 8      Berry           Blueberry 3 -0.693395862  0.30840131  0.49784640
## 9      Berry           Blueberry 4 -1.062846799 -0.19768031  0.54029846
## 10     Berry           Blueberry 5 -0.903946503  0.01258835  0.29565501
## 11     Berry           Blueberry 6 -0.911518977 -0.80933135 -0.45676687
```

```
## 12     Berry                Strawberry1  1.123787175 -1.29134698  0.30029870
## 13     Berry                Strawberry2  2.186526847  0.39020970  0.46639303
## 14     Berry                Strawberry3  0.216458365 -0.68408859  0.84099468
## 15     Berry                Strawberry4  0.779921416  0.31126088  1.56706081
## 16     Berry                Strawberry5  0.595885377 -0.65590705  0.58277734
## 17     Berry                Strawberry6  2.890807350  1.51710457 -0.40231055
## 18     Berry                raspberry 1  0.002784249  0.24770479  0.77952820
## 19     Berry                raspberry 2 -0.181932219 -0.05470297  0.05660525
## 20     Berry                raspberry 3  0.714829561 -1.32650761 -2.11426476
## 21     Berry                raspberry 4  0.041631118 -0.30336432 -0.10035713
## 22     Berry                raspberry 5 -0.177484391 -0.92080171 -0.04037283
## 23     Berry                raspberry 6 -0.239619991 -1.64119492 -0.69498878
## 24 Attribute cata_appearance_unevencolor  1.936613700 -1.17322492  0.32072486
## 25 Attribute   cata_appearance_misshapen  0.058944103  2.05113772 -3.66060544
## 26 Attribute    cata_appearance_notfresh  2.453268365  0.16152199 -0.22489458
## 27 Attribute       cata_appearance_fresh -0.937269938  0.02467928  0.44045357
## 28 Attribute cata_appearance_goodquality -0.997347176  0.08087827  0.50595892
## 29 Attribute        cata_appearance_none -0.793756318 -1.48244462 -0.16563013
## 30 Attribute           cata_taste_floral -0.281501563 -0.15570463 -0.66082635
## 31 Attribute            cata_taste_berry -0.332540749 -0.79022119 -0.46021047
## 32 Attribute           cata_taste_grassy  0.597826399  1.20258057  1.48981642
## 33 Attribute        cata_taste_fermented  0.485339794  2.23871239  0.82509530
## 34 Attribute           cata_taste_fruity -0.122408859 -1.18256562 -0.46550861
## 35 Attribute           cata_taste_earthy  0.354905352  1.43796995  0.85550318
## 36 Attribute             cata_taste_none  0.120742891  0.96539923 -0.03486454
##          Dim4         Dim5         Dim6         Dim7        Dim8        Dim9
## 1  -2.1207457  0.421035244  0.416339437 -0.087205574  0.55527540  0.66487134
## 2  -0.6763648 -0.401344347  1.353241139 -0.143661403  0.31809116  0.82018151
## 3  -1.5512728  0.149090611 -0.154464899 -0.009161493  0.17303241 -0.07476922
## 4   1.8024546 -1.054134662  0.604303775  1.121173149  1.61580941 -0.44455523
## 5  -0.3768377 -1.478964909 -0.652553200 -0.503141105 -0.95611885 -0.76061578
## 6   0.9721393  0.341902827 -0.708842996  0.275826588  0.50857290  0.94363142
## 7   1.3567656 -0.058683995  3.082539058  0.538179530 -1.16634324 -0.43368257
## 8   0.2667007  1.452589890 -1.525570006  0.767208280 -0.17612888 -1.96339520
## 9  -0.6011662  1.453852344 -0.121844931  0.606785673 -0.26607669  0.49314835
## 10  1.3192008  1.250783134  0.820976326  0.475499585  1.30639341 -0.73014319
## 11 -0.1166850  1.177305333 -0.582502705 -0.563767160 -1.33239601 -1.94597487
## 12 -1.9371408 -0.269928992  0.003367573  1.980558028  1.56631289 -0.17821445
## 13 -0.0688568 -0.454494991  0.333203747 -1.237545485  0.45004439 -0.95562013
## 14  0.1817546 -1.474293484  1.174373617  0.381298231  0.06195272  0.41063042
## 15  1.0718960 -1.737282915 -1.577928830 -0.329844134  0.71300614 -1.32879302
## 16 -1.2198681 -0.852687372  0.761511155  0.054789811 -1.58761812 -1.24962440
## 17  0.8846827  2.338783198  0.483756496  0.160111078 -0.57887026  0.34590501
## 18  0.7647137 -0.914998126 -0.901156816  0.498483211 -2.30064883  1.53815305
## 19  0.2948372 -0.159554062 -1.041068949 -0.750258551  1.07509047  0.55799211
## 20 -0.3861240  0.086736116 -0.577370181  1.175513729 -0.54472390  0.75571365
```

```
## 21  0.3256547  0.412286985 -0.435287990  -1.543719337 -0.11575622  1.57370723
## 22  0.3511281 -0.003814396 -0.678042187   0.444107106  0.28159320  1.09950507
## 23 -0.3495728  0.259177191  0.697218086  -2.755894177  0.71155298  0.04135319
## 24 -0.3691085 -1.620059383 -0.377473620   0.301545855 -0.63352523  1.22816392
## 25  0.3531013 -1.006035553 -0.212496004  -0.239274881 -1.09516405  0.26709095
## 26  0.9952467  2.868903004  1.026387237  -0.055450592 -0.38506099  0.06475031
## 27  0.1240773 -0.108632925  0.141496178   0.043970746 -0.20627303  0.81170689
## 28  0.4849384  0.529754205  0.332772420   0.342509736 -1.07314538  0.47718892
## 29 -1.4180918  0.690178046  3.291335009 -17.006164060  1.32949174  1.45853357
## 30 -0.2243516  1.497283724 -2.819331519   0.864064594  1.83366072  0.12638909
## 31 -0.4352527  0.034561355  0.089370302  -0.214060919  0.26794118 -0.98297215
## 32  1.2955302 -0.968001002 -1.023546788  -0.489156221 -0.70173788 -2.52321861
## 33 -3.9589988  0.105941940  1.889817842   1.038772276  0.30006618 -0.78605241
## 34 -0.2273194 -0.145301727  0.293071101   0.084345882  0.42917126 -0.65205090
## 35 -0.3491357 -0.018917764 -1.201391871  -0.996446089  1.35938942  1.24888861
## 36  3.5345743 -2.052653151  3.712240877   1.346737365  4.83311179  0.41381134
##         Dim10       Dim11        Dim12
## 1   0.7992141 -0.95983642 -0.009253265
## 2   1.3393069  0.68731238  0.267060332
## 3  -1.5859313 -0.69717059 -1.168003875
## 4  -1.3036853 -0.22875124 -0.894559086
## 5  -0.2891770  0.74883806  1.156346478
## 6  -1.0496073 -0.33120184 -0.408429808
## 7   1.1514695 -0.67185606 -0.519204536
## 8  -0.1572088 -0.29921710  1.327706362
## 9  -0.9894949  1.24943134  0.309664418
## 10  1.4665355  0.54425299  1.516555388
## 11 -0.2792107 -1.24454179 -1.159488529
## 12 -0.0413237  0.49720288  0.715403472
## 13  0.1394031 -2.84848835  1.293593154
## 14 -2.2704383 -0.36073715  0.084475487
## 15  0.5226638  1.45255686 -0.678893230
## 16  0.5416557  1.25779330 -1.359994255
## 17 -0.8335918  1.22987860 -0.669642985
## 18  0.2120347 -0.04656501  1.543167644
## 19  1.7404352 -0.10284611 -1.758554927
## 20  0.9693176 -0.40241340 -0.328363440
## 21 -0.2755199 -0.43059527 -0.823009716
## 22  0.3868386 -0.56755742  0.383635858
## 23 -0.5673104  1.44742716  1.252192647
## 24 -0.4084541 -0.62254913  0.317272405
## 25 -0.1111936  0.11384684 -0.118543402
## 26  0.2590012  0.65050805  0.369378079
## 27 -0.1700575  1.47757553  1.057741523
## 28 -0.1537466 -1.36294181 -0.799243733
## 29 -7.0630004 -2.76411379  0.824372037
```

```
## 30 -2.2281270 -0.79295431  0.615338734
## 31  1.2341189 -0.68568963  0.984392755
## 32 -0.6268628  0.30656609  0.112193577
## 33 -1.2096748 -0.26064110  0.099479751
## 34 -0.3340588  1.11961579 -1.897053591
## 35  2.0332988  0.06165682 -1.232091871
## 36 -0.8624509 -1.47668343  0.524665515
```

We could also add on any columns that have one value for each product *and* each attribute (or fill in the gaps with `NA`s). Maybe we want a column with the `rowmass`es and `colmass`es. These are vectors, so it would be handy if we could wrangle them into tibbles first.

You can use either `tibble()` or `data.frame()` to make vectors in the same order into a table. They have basically the same usage. Just make sure you name your columns!

```
berry_rowmass <- tibble(Variable = berry_ca_res$rownames,
                        Mass = berry_ca_res$rowmass)

berry_rowmass
```

```
## # A tibble: 23 x 2
##    Variable       Mass
##    <chr>         <dbl>
##  1 Blackberry 1 0.0392
##  2 Blackberry 2 0.0394
##  3 Blackberry 3 0.0412
##  4 Blackberry 4 0.0401
##  5 Blackberry 5 0.0429
##  6 Blueberry 1  0.0394
##  7 Blueberry 2  0.0378
##  8 Blueberry 3  0.0399
##  9 Blueberry 4  0.0386
## 10 Blueberry 5  0.0392
## # i 13 more rows
```

If you have an already-named vector, `enframe()` is a handy shortcut to making a two-column tibble, but unfortunately this isn't how the `ca` package structures its output.

```
named_colmasses <- berry_ca_res$colmass
names(named_colmasses) <- berry_ca_res$colnames

berry_colmass <- named_colmasses %>%
```

```
  enframe(name = "Variable",
          value = "Mass")

berry_colmass
```

```
## # A tibble: 13 x 2
##    Variable                        Mass
##    <chr>                          <dbl>
##  1 cata_appearance_unevencolor 0.0848
##  2 cata_appearance_misshapen   0.0473
##  3 cata_appearance_notfresh    0.0551
##  4 cata_appearance_fresh       0.162
##  5 cata_appearance_goodquality 0.145
##  6 cata_appearance_none        0.00271
##  7 cata_taste_floral           0.0457
##  8 cata_taste_berry            0.162
##  9 cata_taste_grassy           0.0614
## 10 cata_taste_fermented        0.0339
## 11 cata_taste_fruity           0.119
## 12 cata_taste_earthy           0.0643
## 13 cata_taste_none             0.0164
```

And now we can use `bind_rows()` and `left_join()` to jigsaw these together.

```
bind_rows(berry_colmass, berry_rowmass) %>%
  left_join(berry_ca_coords, by = "Variable")
```

```
## # A tibble: 36 x 15
##    Variable    Mass Type     Dim1    Dim2   Dim3   Dim4    Dim5    Dim6     Dim7
##    <chr>      <dbl> <chr>   <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
##  1 cata_ap~ 0.0848  Attr~  1.94   -1.17    0.321 -0.369 -1.62   -0.377    0.302
##  2 cata_ap~ 0.0473  Attr~  0.0589  2.05   -3.66   0.353 -1.01   -0.212   -0.239
##  3 cata_ap~ 0.0551  Attr~  2.45    0.162  -0.225  0.995  2.87    1.03    -0.0555
##  4 cata_ap~ 0.162   Attr~ -0.937   0.0247  0.440  0.124 -0.109   0.141    0.0440
##  5 cata_ap~ 0.145   Attr~ -0.997   0.0809  0.506  0.485  0.530   0.333    0.343
##  6 cata_ap~ 0.00271 Attr~ -0.794  -1.48   -0.166 -1.42   0.690   3.29   -17.0
##  7 cata_ta~ 0.0457  Attr~ -0.282  -0.156  -0.661 -0.224  1.50   -2.82     0.864
##  8 cata_ta~ 0.162   Attr~ -0.333  -0.790  -0.460 -0.435  0.0346  0.0894  -0.214
##  9 cata_ta~ 0.0614  Attr~  0.598   1.20    1.49   1.30  -0.968  -1.02    -0.489
## 10 cata_ta~ 0.0339  Attr~  0.485   2.24    0.825 -3.96   0.106   1.89     1.04
## # i 26 more rows
## # i 5 more variables: Dim8 <dbl>, Dim9 <dbl>, Dim10 <dbl>, Dim11 <dbl>,
## #   Dim12 <dbl>
```

In summary: - Many analyses will give you **lists** full of every possible piece of data you could need, which aren't necessarily tabular. - If you need to turn a **tabular data** with `rownames` into a tibble, use `rownames_to_column()` or `as_tibble()`. - If you need to turn a **named vector** into a two-column table, use `enframe()` - If you need to turn **multiple vectors** into a table, use `tibble()` or `data.frame()`. - You can combine multiple tables together using `bind_rows()` and `left_join()`, if you manage your column names and the orders of your vectors carefully.

Like with our *untidying* process, the shape you need to get your data into during *retidying* is ultimately decided by what you want to do with it next. Correspondence Analysis is primarily a graphical method, so next we're going to talk about graphing functions in R in our last substantive chapter. By the end, you will be able to make the plots we showed in the beginning!

Let's take a quick moment to **save our data** before we move on, though, so we don't have to rerun our `ca()` whenever we restart `R` to make more changes to our graph.

As we've shown before, you can save tabular data easily:

```
berry_ca_coords %>%
  write_csv("data/berry_ca_coords.csv")

berry_col_coords %>%
  write_csv("data/berry_ca_col_coords.csv")

berry_row_coords %>%
  write_csv("data/berry_ca_row_coords.csv")
```

But `.csv` is a **tabular format**, so it's a little harder to save the whole non-tabular list of `berry_ca_res` as a table. There's a lot of stuff we may need later, though, so just in case we can save it as an `.Rds` file:

```
berry_ca_res %>%
  write_rds("data/berry_ca_results.rds")
```

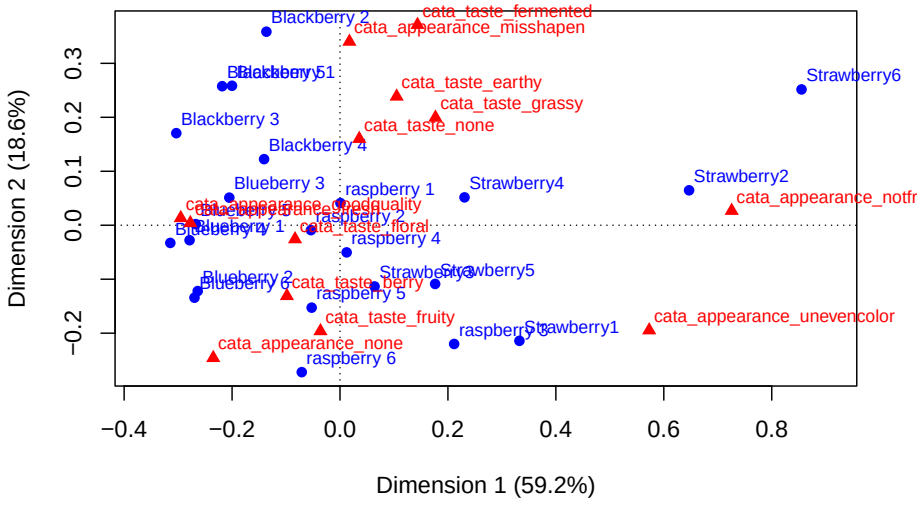
Chapter 6

Data visualization basics with ggplot2

6.1 Built-in plots with the ca package

Normally, we present CA results as a graph. You can use the base R `plot()` function *directly* on the output of `ca()` for quick results.

```
plot(berry_ca_res)
```



You can learn some things from this already: Strawberries 1 & 2 had some appearance defects that made them noticably different from the others. The blackberries were generally more earthy, while many of the raspberries and strawberries had more fruity/berry flavor. Often, multivariate data analysis packages have pretty good defaults built into their packages that you should take advantage of to quickly look at your data.

But it's hard to see what's going on with the blueberries. A lot of the text is impossible to read. And some additional color- or shape-coding with a key would be helpful.

6.2 Basics of Tidy Graphics

If you want to make this look publication-friendly by getting rid of overlapping text, changing font size and color, color-coding your berries, etc, you *can* do this with base `R`'s plotting functions. The help files for `?plot.ca` and pages 262-268 in Greenacre's *Correspondence Analysis in Practice* demonstrate the wide variety of options available (although Greenacre also explains on pages 283-284 that he used a variety of non-base `R` tools to make figures for the book). In general, however, it's much easier to use the tidyverse package `ggplot2` to manipulate graphs.

`ggplot2` provides a standardized, programmatic interface for data visualization, in contrast to the piecemeal approach common to base `R` graphics plotting. This means that, while the syntax itself can be challenging to learn, syntax for

different tasks differs in logical and predictable ways and it works well together with other `tidyverse` functions and principles (like `select()` and `filter()`).

The schematic elements of a ggplot are as follows:

```
# The ggplot() function creates your plotting environment.  We often save it to a variable in R s
p <- ggplot(mapping = aes(x = <a variable>, y = <another variable>, ...),
            data = <your data>)

# Then, you can add various ways of plotting data to make different visualizations.
p +
  geom_<your chosen way of plotting>(...) +
  theme_<your chosen theme> +
  ...
```

In graphical form, the following diagram (from VT Professor JP Gannon) gives an intuition of what is happening:

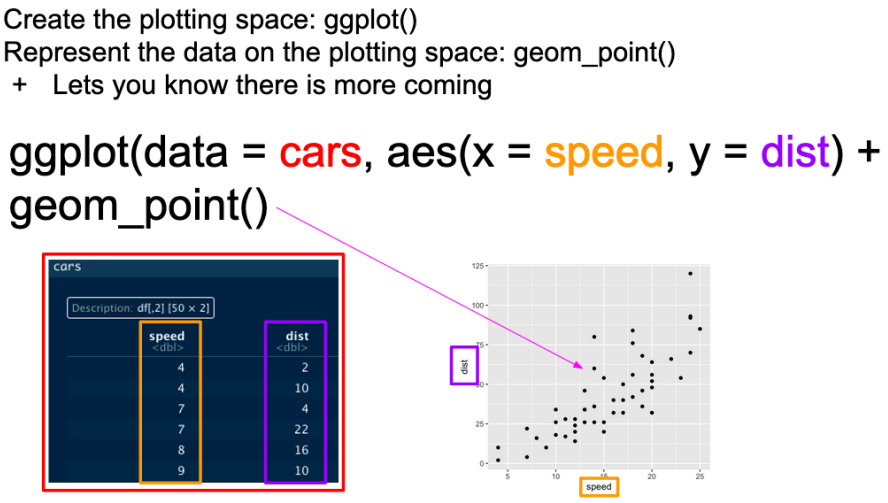


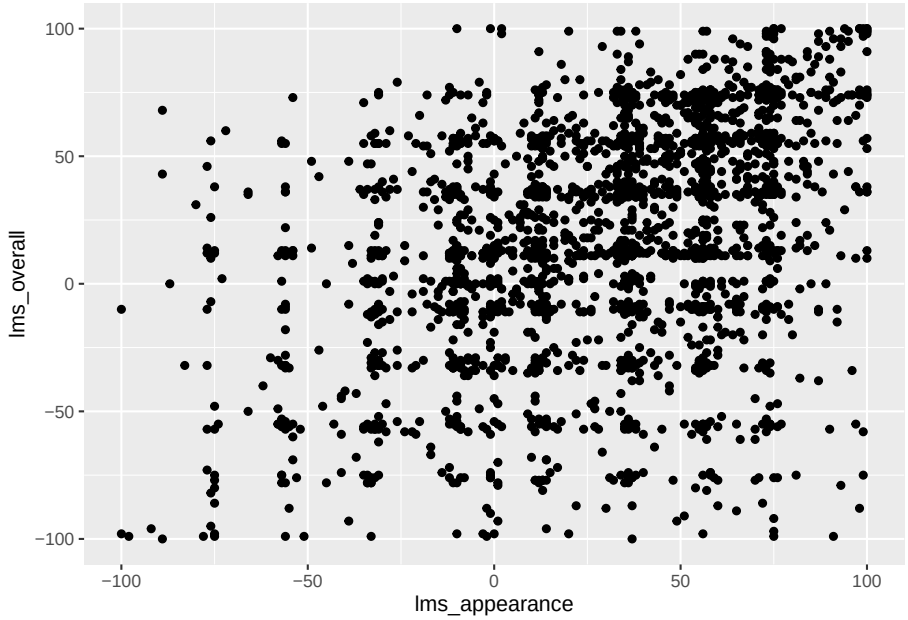
Figure 6.1: Basic ggplot mappings. Color boxes indicate where the elements go in the function and in the plot.

Since ggplot2 is a tidyverse package, we do need to (re)tidy the data (and, often, keep reshaping it after that). In general, `ggplot2` works best with data in "long" or "tidy" format: one row for every observation or point.

By default, the scatterplot function needs one row per point with one column of coordinates for each axis (normally x and y):

```
berry_data %>%
  # Here we set up the base plot
  ggplot(mapping = aes(x = lms_appearance, y = lms_overall)) +
   # Here we tell our base plot to add points
  geom_point()
```

```
## Warning: Removed 5005 rows containing missing values (`geom_point()`).
```

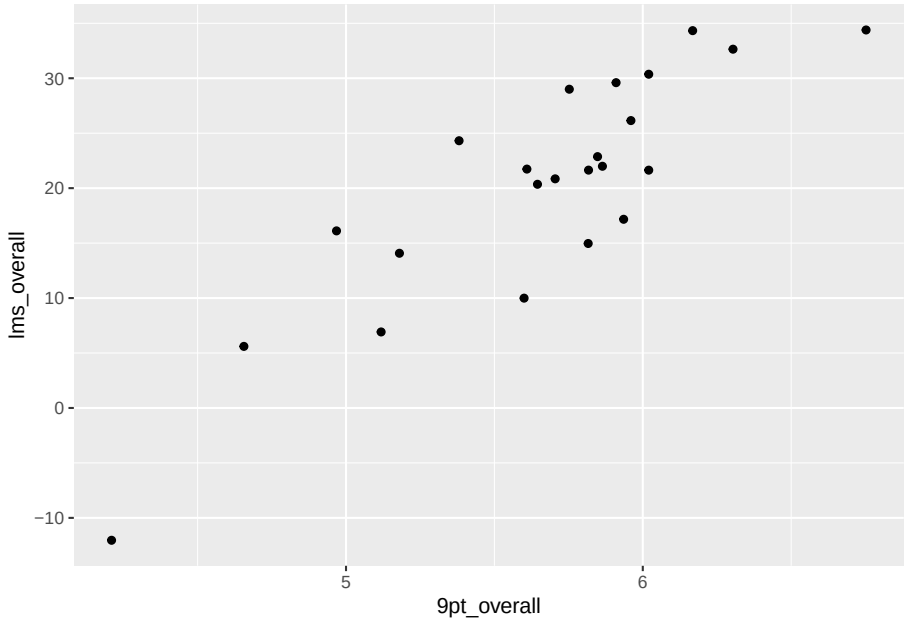


This doesn't look all that impressive–partly because the data being plotted itself isn't that sensible, and partly because we haven't made many changes. If we want to plot some **summary** data, maybe one point per berry sample, we can use the familiar `tidyverse` functions to reshape our data and pipe it into `ggplot()`.

```
berry_data %>%
  select(`Sample Name`, contains(c("9pt_","lms_","us_"))) %>%
  summarize(across(everything(), ~ mean(.x, na.rm = TRUE)), .by = `Sample Name`) ->
  berry_average_likings

berry_average_likings %>%
  nrow()
```

```
## [1] 23
```

```
berry_average_likings %>%
  ggplot(aes(x = `9pt_overall`, y = `lms_overall`)) +
   #23 points, one per row
  geom_point()
```



This plot has fewer overlapping points and less noise, so it's a lot more informative. But it still doesn't look that good, with the underscores in the axis labels, the printer-unfriendly grey background, etc. Let's start looking at the pieces that make up a `ggplot` so we can change them.

6.2.1 The `aes()` function and `mapping =` argument

The `ggplot()` function takes two arguments that are essential, as well as some others you'll only use in specific cases. The first, `data =`, is straightforward, and you'll usually be passing data to the function at the end of some pipeline using `%>%`

The second, `mapping =`, is less clear. This argument requires the `aes()` function, which can be read as the "aesthetic" function. The way that this function works is quite complex, and really not worth digging into here, but it's the place where you **tell `ggplot()` what part of your data is going to connect to what part of the plot**. So, if we write `aes(x = rating)`, we can read this in our heads as "the values of x will be mapped from the 'rating' column".

This sentence tells us the other important thing about `ggplot()` and the `aes()` mappings: **mapped variables each have to be in their own column**. This is another reason that `ggplot()` requires tidy data.

6.2.2 Adding layers with `geom_*()` functions

In the above example, we added (literally, using `+`) a function called `geom_point()` to the base `ggplot()` call. This is functionally a "layer" of our plot, that tells `ggplot2` how to actually visualize the elements specified in the `aes()` function–in the case of `geom_point()`, we create a point for each row's combination of `x = lms_overall` and `y = 9pt_overall`.

```
berry_average_likings %>%
  select(lms_overall, `9pt_overall`)
```

```
## # A tibble: 23 x 2
##    lms_overall `9pt_overall`
##          <dbl>         <dbl>
##  1       32.6           6.30
##  2       26.1           5.96
##  3       22.0           5.86
##  4       34.3           6.17
##  5       15.0           5.82
##  6       16.1           4.97
##  7       21.6           5.82
##  8        5.60          4.66
##  9        6.92          5.12
## 10       20.4           5.65
## # i 13 more rows
```

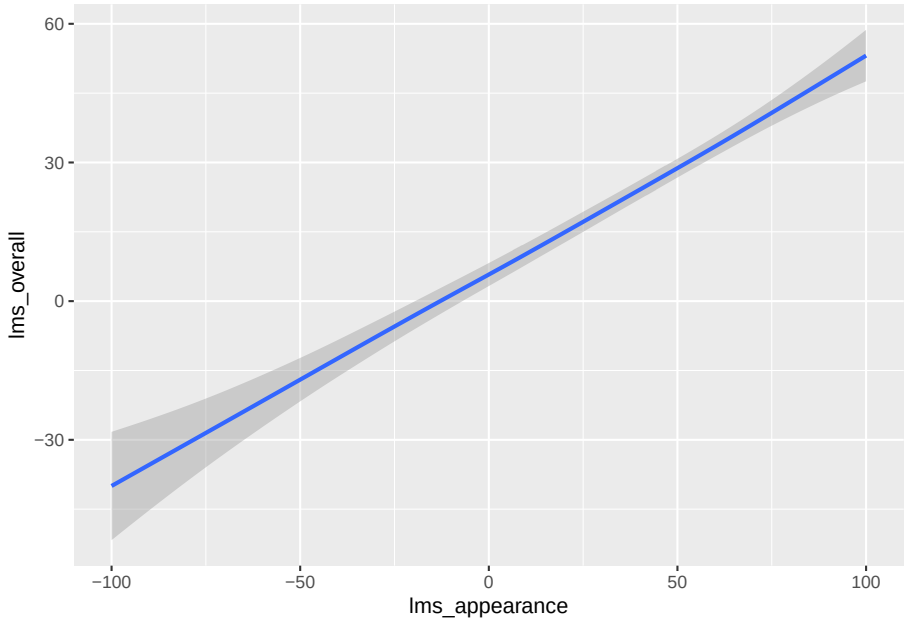
There are many `geom_*()` functions in `ggplot2`, and many others defined in other accessory packages like `ggrepel`. These determine what symbols and spatial arrangements are used to represent each row, and are the heart of visualizations. Some `geom_*()` functions do some summarization of their own, making them more appropriate for raw data.

We can see this by swapping out the `geom_*()` in our initial scatterplot on the Labeled Magnitude Scale liking data:

```
berry_data %>%
  ggplot(mapping = aes(x = lms_appearance, y = lms_overall)) +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```

```
## Warning: Removed 5005 rows containing non-finite values (‘stat_smooth()‘).
```

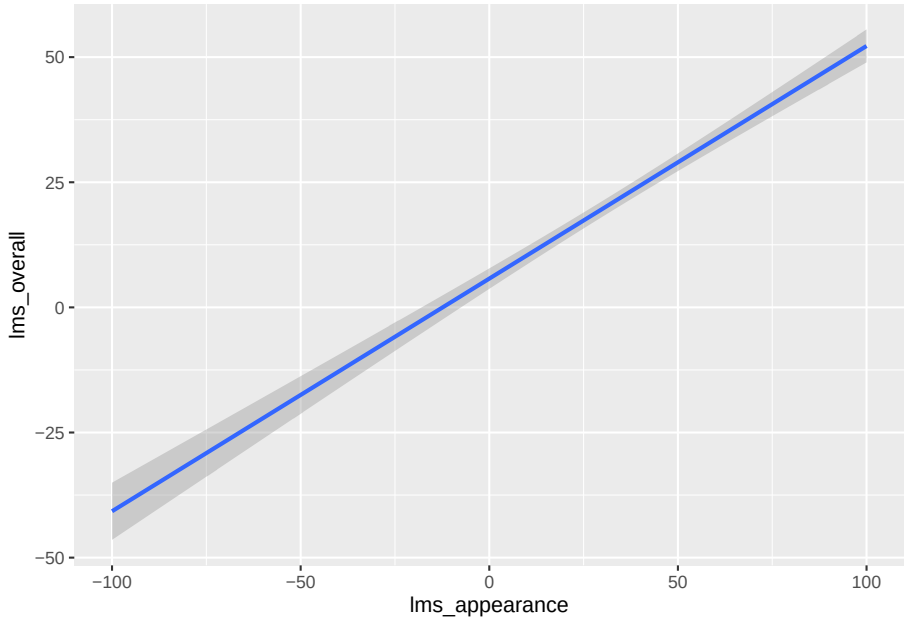


`geom_smooth()` fits a smoothed line to our data. By default, it will use either Local Polynomial Regression or the Generalized Additive Model, depending on the size of your data (here, you can see that it chose `gam`, the Generalized Additive Model). You can specify models manually, using the `method` argument of `geom_smooth()`:

```
berry_data %>%
  ggplot(mapping = aes(x = lms_appearance, y = lms_overall)) +
  geom_smooth(method = "lm")
```

```
## ‘geom_smooth()‘ using formula = ’y ~ x’
```

```
## Warning: Removed 5005 rows containing non-finite values (‘stat_smooth()‘).
```



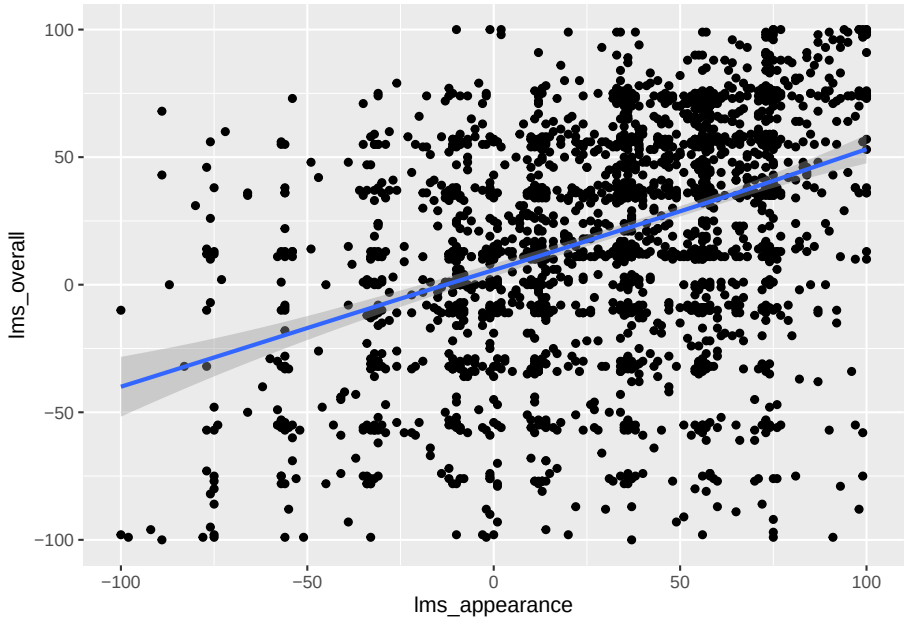
We can also combine layers, as the term implies:

```
berry_data %>%
  ggplot(mapping = aes(x = lms_appearance, y = lms_overall)) +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```

```
## Warning: Removed 5005 rows containing non-finite values (`stat_smooth()`).
```

```
## Warning: Removed 5005 rows containing missing values (`geom_point()`).
```

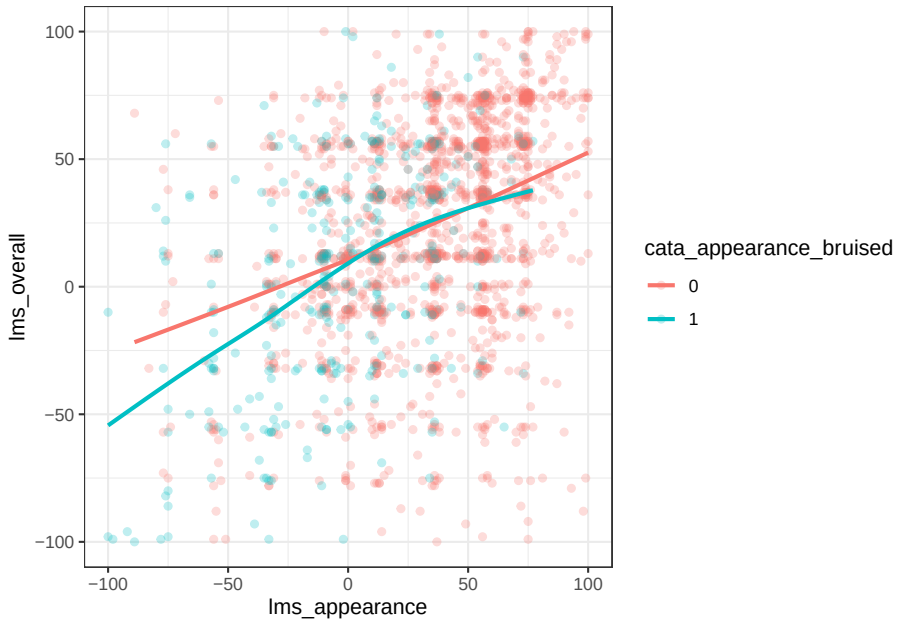


Note that we don't need to tell *either* `geom_smooth()` or `geom_point()` what `x` and `y` are–they "inherit" them from the `ggplot()` function to which they are added (`+`), which defines the plot itself.

What other arguments can be set to aesthetics? Well, we can set other visual properties like **color**, **size**, **transparency** (called "alpha"), and so on. For example, let's try to look at whether there is a relationship between berry condition (proxied here by `cata_appearance_bruised`) and overall liking.

```
berry_data %>%
  #ggplot will drop NA values for you, but it's good practice to
  #think about what you want to do with them:
  drop_na(lms_overall, cata_appearance_bruised) %>%
  #color, shape, linetype, and other aesthetics that would add a key
  #don't like numeric data types. The quick-and-dirty solution:
  mutate(across(starts_with("cata_"), as.factor)) %>%
  ggplot(mapping = aes(x = lms_appearance, y = lms_overall,
                       color = cata_appearance_bruised)) +
  geom_point(alpha = 1/4) +
  geom_smooth(se = FALSE) +
  theme_bw()
```

```
## `geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



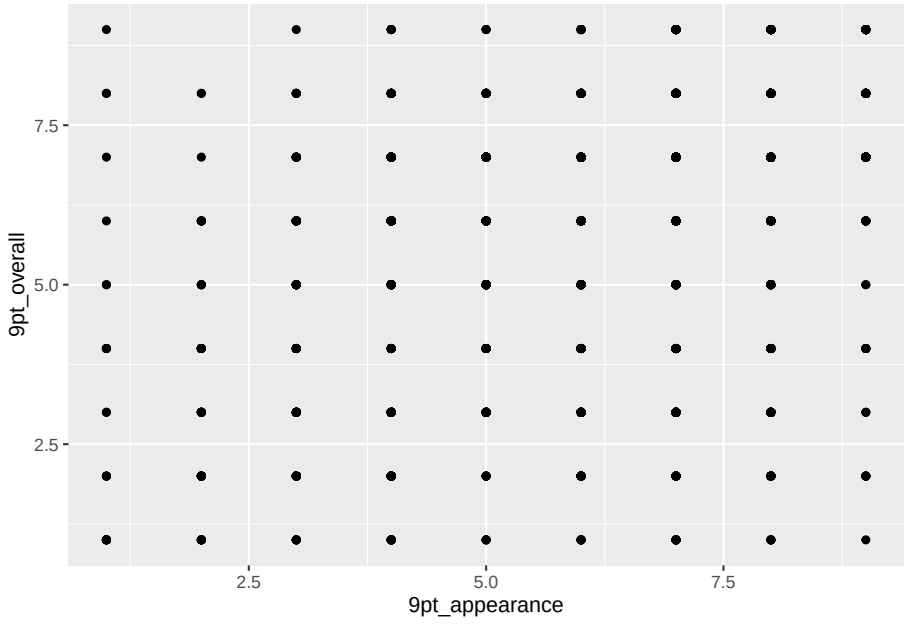
We can see that more of the blue dots for samples with a bruised appearance in the lower left of the figure–it has a negative influence on the ratings of overall liking *and* appearance.

6.2.3 Geoms for categorical, ordinal, or unevenly-distributed data

You may notice that we've been using the Labeled Magnitude Scale data so far, rather than the data from the other two scales. That's because adding `geom_point()` to the 9-point hedonic scale data looks like this:

```
berry_data %>%
  ggplot(mapping = aes(x = `9pt_appearance`, y = `9pt_overall`)) +
  geom_point()
```

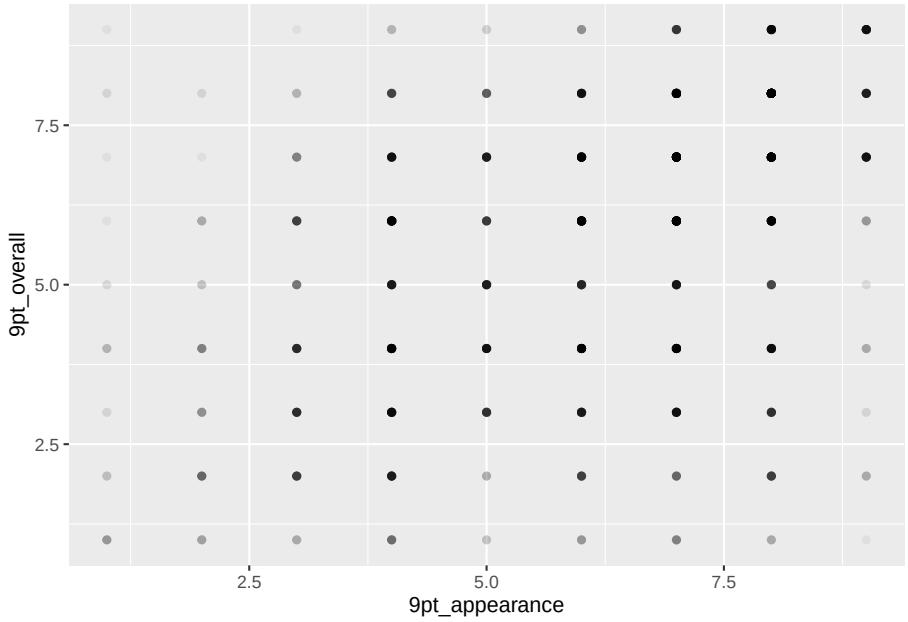
```
## Warning: Removed 5062 rows containing missing values (‘geom_point()‘).
```



Each of those points actually represents many individual ratings of berries, possibly hundreds. There are almost certainly fewer people giving the berries a 1 for appearance and a 9 for overall liking than there are people rating each a 6. This also makes it a good demonstration of how `ggplot` handles the transparency of overlapping points:

```
berry_data %>%
  ggplot(mapping = aes(x = `9pt_appearance`, y = `9pt_overall`)) +
  geom_point(alpha = 0.05)
```

```
## Warning: Removed 5062 rows containing missing values (‘geom_point()‘).
```



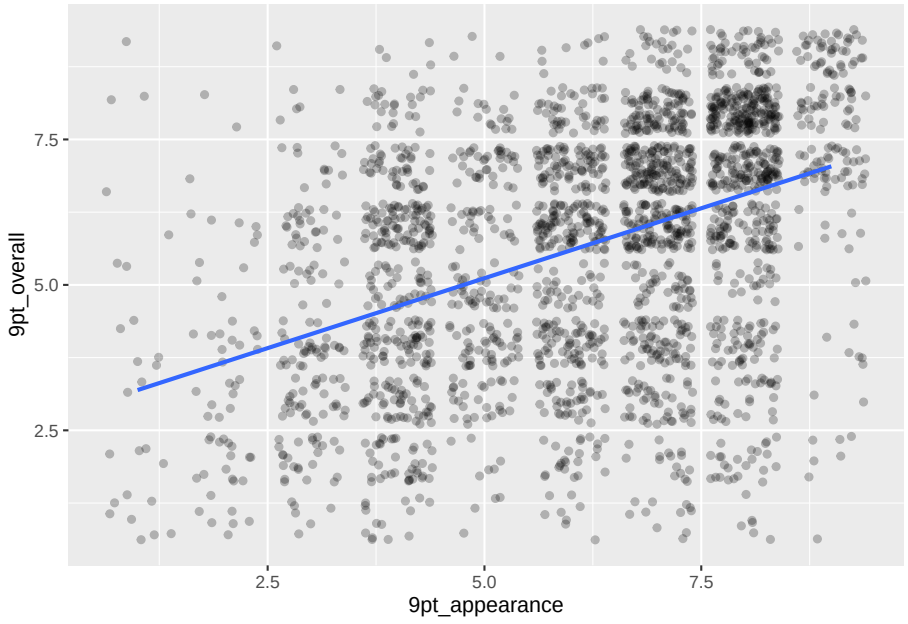
But the actual solution to this problem, instead of the hacky pseudo-heat map, is `geom_jitter()`, which applies a small random x and y offset to each point:

```
berry_data %>%
  ggplot(mapping = aes(x = `9pt_appearance`, y = `9pt_overall`)) +
  geom_jitter(alpha = 1/4) +
  geom_smooth(method = "lm", se = FALSE)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

```
## Warning: Removed 5062 rows containing non-finite values (`stat_smooth()`).
```

```
## Warning: Removed 5062 rows containing missing values (`geom_point()`).
```

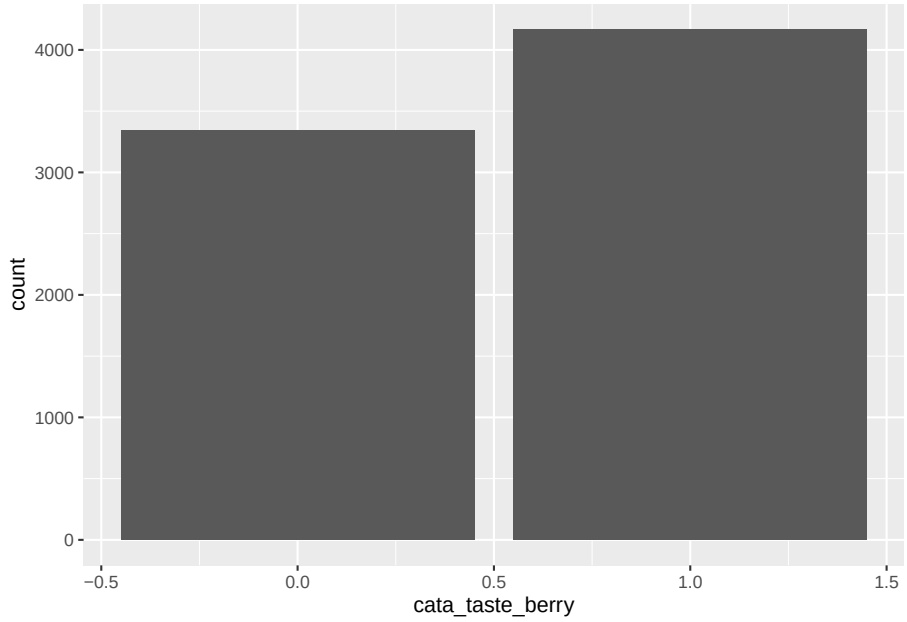


You can see there are some overlapping points left, but this gives us a much better idea of the shape, along with the summarizing `geom_smooth()`. Since there are only 9 possible values on the hedonic scale while the continuous Labeled Magnitude Scale allows people to select numbers in-between scale labels, `geom_jitter()` can be thought of as simulating this random human scale usage after the fact.

If you'd like to look at the variation of a single categorical or discrete variable, a bar plot is more appropriate. `geom_bar()` is another **summarizing** geom, similar to `geom_smooth()`, as it expects a discrete `x` variable and *one row per observation*. It will **count** the number of rows in each **group** and use those counts to plot the bar heights, one bar per group. (Note that you can override or tweak this behavior using additional arguments.)

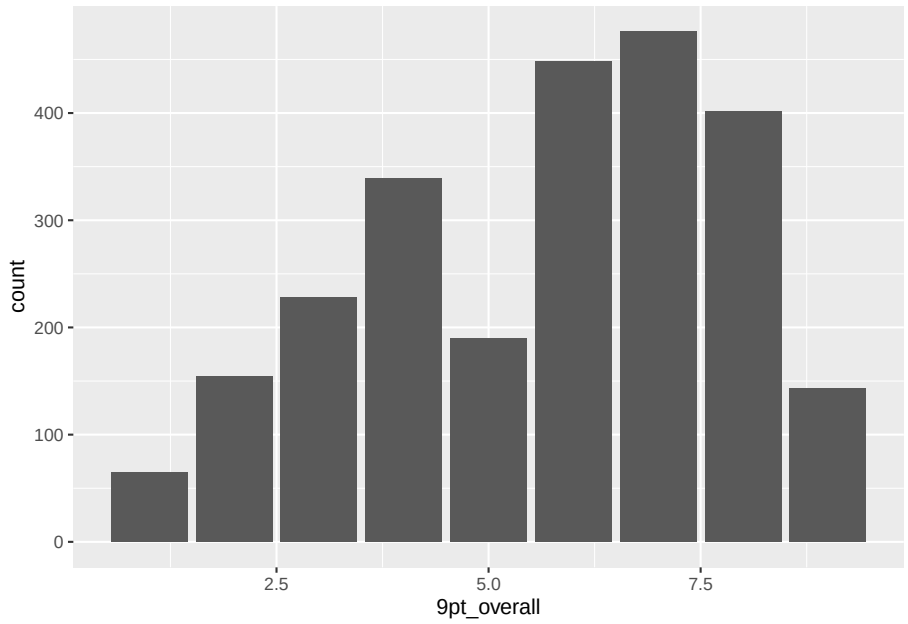
`geom_histogram()` is the version for numeric data, which will also calculate bins for you.

```
#geom_bar() is for when you already have discrete data, it just counts:
berry_data %>%
  ggplot(aes(x = cata_taste_berry)) +
  geom_bar()
```




```
berry_data %>%
  ggplot(aes(x = `9pt_overall`)) +
  geom_bar()
```

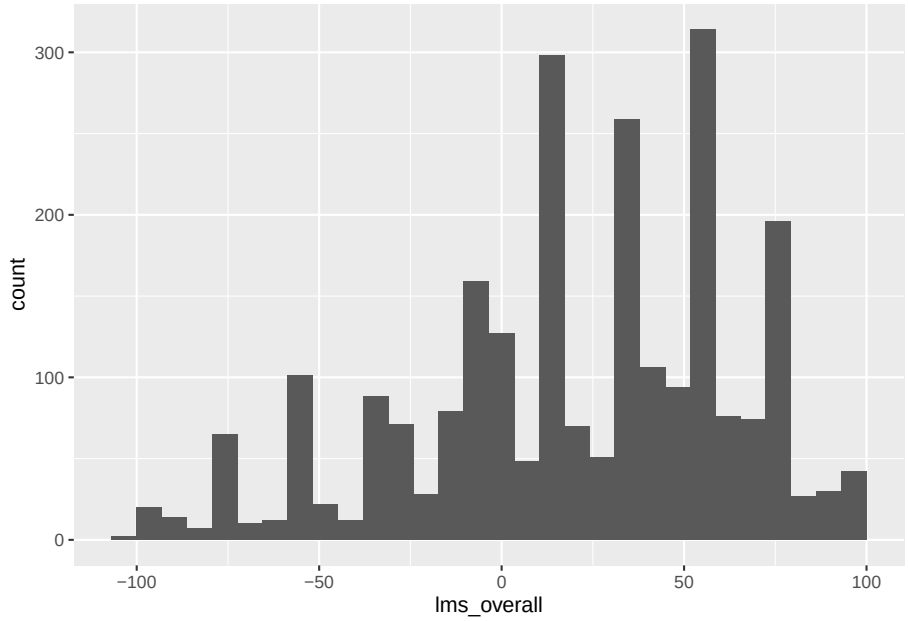
```
## Warning: Removed 5062 rows containing non-finite values (‘stat_count()‘).
```




```
#and geom_histogram() is for continuous data, it counts and bins:
berry_data %>%
  ggplot(aes(x = `lms_overall`)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 5005 rows containing non-finite values (`stat_bin()`).
```

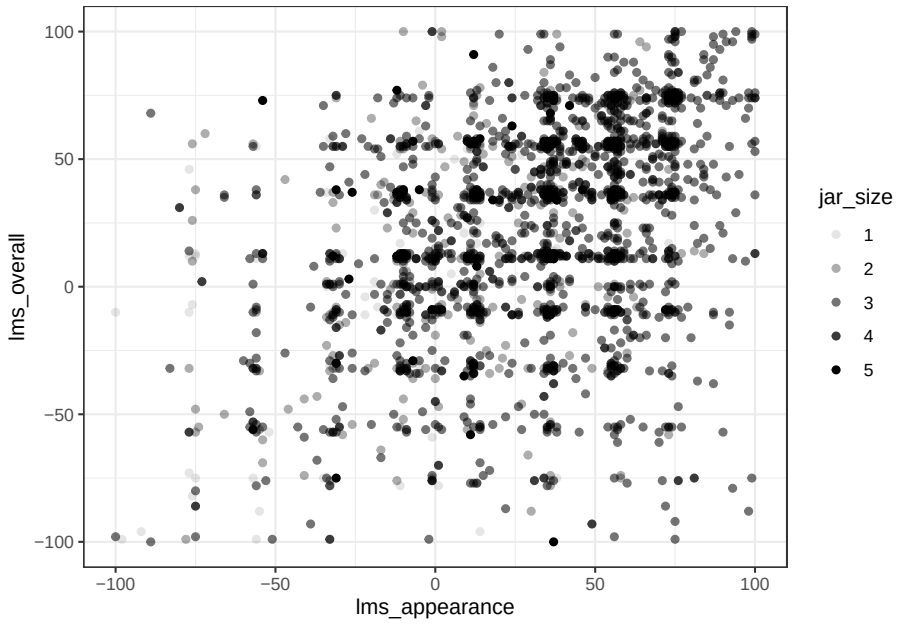


6.2.4 Arguments inside and outside of `aes()`

In some previous plots, we've seen some aesthetic elements specified directly inside of geom functions like `geom_point(alpha = 1/4)`, without using `aes()` to **map** a variable to this aesthetic. If we want every point or geom to have the same, fixed look (the same transparency, the same color, etc), we *don't* wrap it in the `aes()` function. `aes()` ties a visual element to a variable.

Note that we *can* map `alpha` to a variable, just like `color`:

```
berry_data %>%
  drop_na(lms_overall, cata_appearance_bruised) %>%
  ggplot(aes(x = lms_appearance, y = lms_overall)) +
  # We can set new aes() mappings in individual layers, as well as the plot itself
  geom_point(aes(alpha = jar_size)) +
  #Unlike color, alpha will accept numeric variables for mapping
  theme_bw()
```

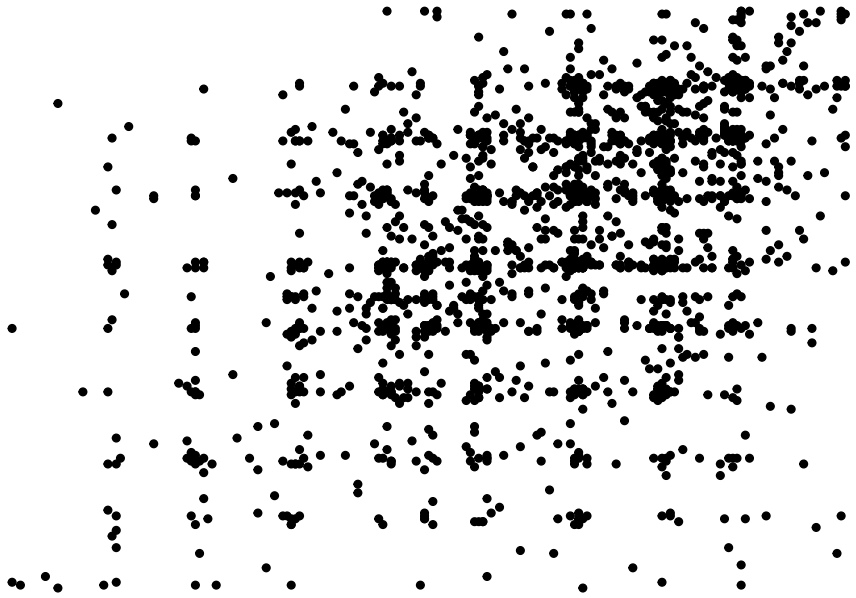


Color would be a better way to represent this relationship, however, as semitransparent points can overlap and appear indistinguishable from a single, darker point.

6.2.5 Using `theme_*()` to change visual options quickly

In the last plot, notice that we have changed from the default (and to my mind unattractive) grey background of `ggplot2` to a black and white theme. This is by adding a `theme_bw()` call to the list of commands. `ggplot2` includes a number of default `theme_*()` functions, and you can get many more through other `R` packages. They can have subtle to dramatic effects:

```
berry_data %>%
  drop_na(lms_overall, cata_appearance_bruised) %>%
  ggplot(aes(x = lms_appearance, y = lms_overall)) +
  geom_point() +
  theme_void()
```



You can also edit every last element of the plot's theme using the base `theme()` function, which is powerful but a little bit tricky to use.

6.2.6 Changing aesthetic elements with `scale_*()` functions

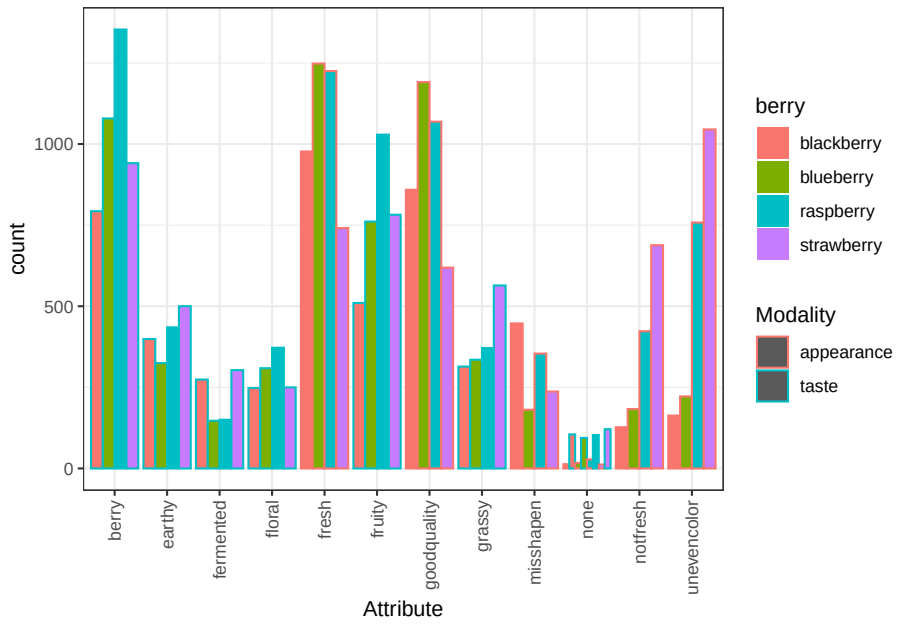
But what about the color of the *points*? None of the themes change the colors used for drawing geoms, or the color scales used for showing categories or additional variables.

The symbols, colors, or other signifiers mapped to aesthetic variables by `mapping()` are are controlled by the `scale_*()` functions. In my experience, the most frequently encountered scales are those for color: either `scale_fill_*()` for solid objects (like the bars in a histogram) or `scale_color_*()` for lines and points (like the outlines of the histogram bars).

Scale functions work by telling `ggplot()` *how* to map aesthetic variables to visual elements. The `viridis` package is a good starting place for color and fill scales, as its `scale_color_viridis_*()` functions provide color-blind and (theoretically) print-safe color palettes.

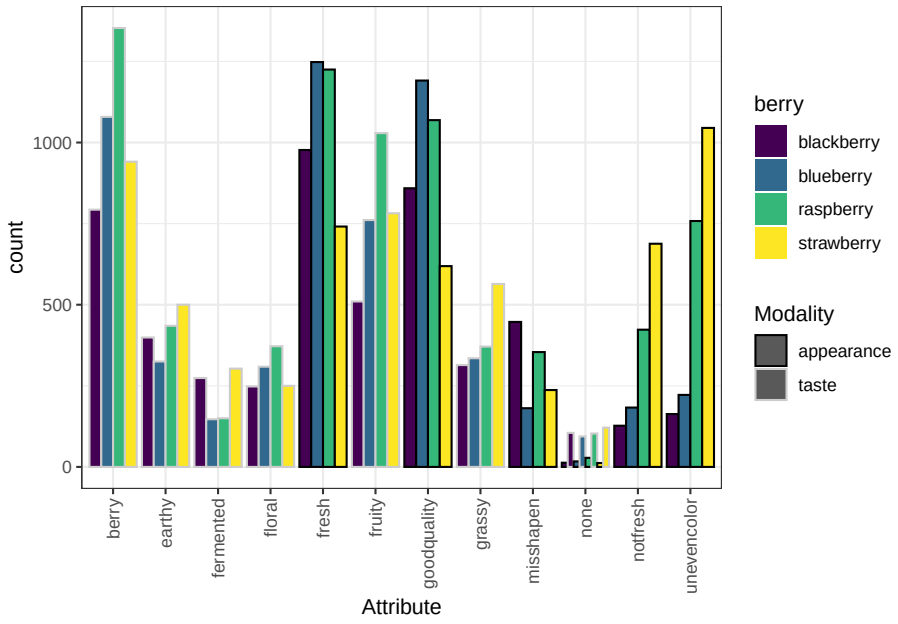
```
# To effectively plot all of the cata attributes on a bar chart, the data
# needs to be longer (one geom_bar() per group, not per column!)
# and we'll remove columns with NAs for now.
berry_cata_long <-
```

```
  berry_data %>%
  select(where(~none(.x, is.na))) %>%
  pivot_longer(starts_with("cata_"),
               names_to = c(NA, "Modality", "Attribute"), names_sep = "_",
               values_to = "Presence")

# And now we can use this for plotting
p <-
  berry_cata_long %>%
  filter(Presence == 1) %>%
  ggplot(aes(x = Attribute, fill = berry, color = Modality)) +
  geom_bar(position = position_dodge()) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))

p
```



We can take a saved plot (like `p`) and use scales to change how it is visualized.

```
p +
  scale_fill_viridis_d() +
  scale_color_grey(start = 0, end = 0.8) #For bar plots, color is the outline!
```

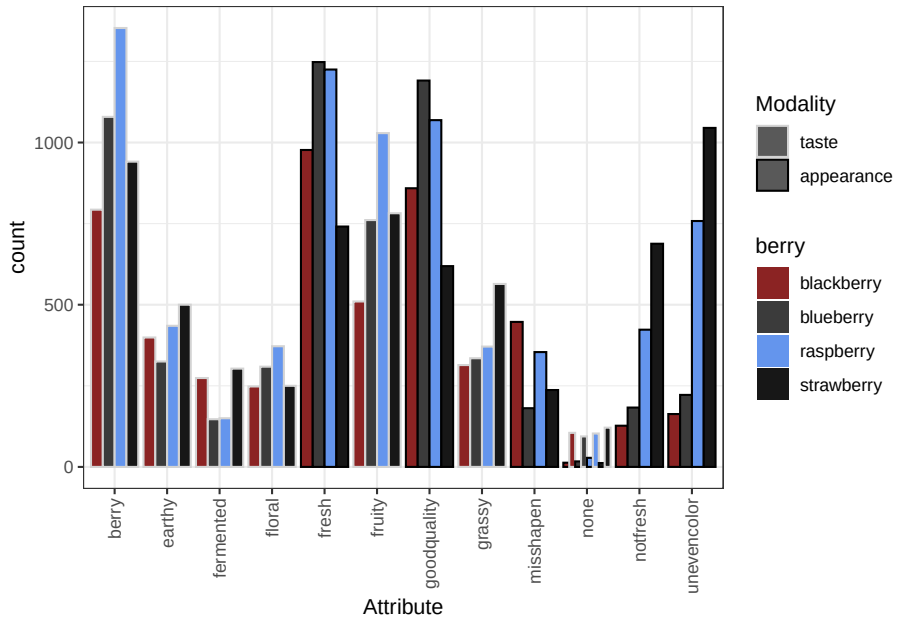


ggplot2 has a broad range of built-in options for scales, but there are many others available in add-on packages that build on top of it. You can also build your own scales using the `scale_*_manual()` functions, in which you give a vector of the same length as your mapped aesthetic variable in order to set up the visual assignment. That sounds jargon-y, so here is an example:

```
# We'll pick 14 random colors from the colors R knows about
random_colors <- print(colors()[sample(x = 1:length(colors()), size = 10)])
```

```
##  [1] "brown4"         "grey23"         "cornflowerblue" "gray9"
##  [5] "mediumpurple2"  "gray16"         "aliceblue"      "darkseagreen"
##  [9] "peachpuff"      "thistle1"
```

```
p +
  scale_fill_manual(values = random_colors) +
  scale_color_manual(breaks = c("taste", "appearance"),
                     values = c("lightgrey", "black"))
```

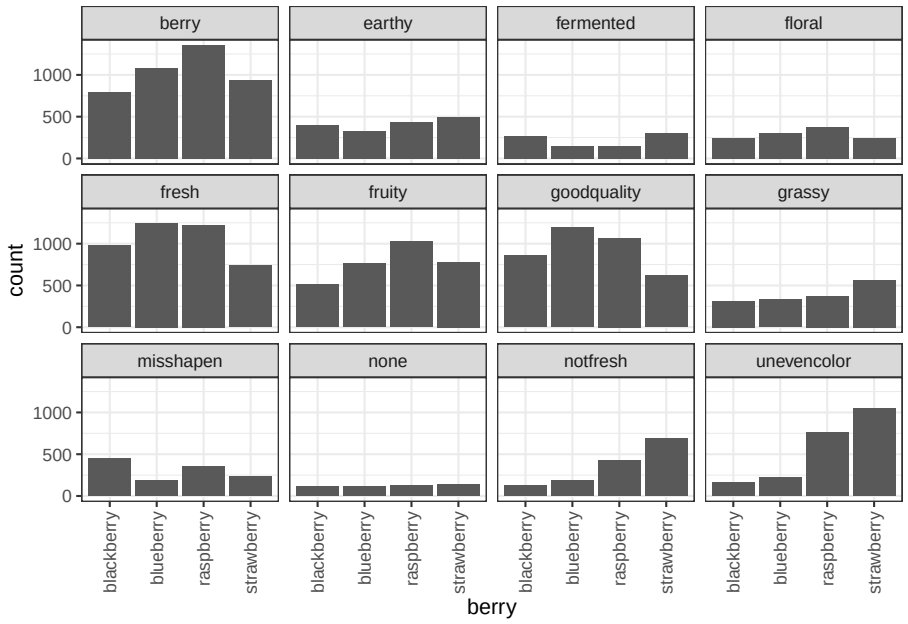


6.2.7 Finally, `facet_*()`

The last powerful tool I want to show off is the ability of `ggplot2` to make what Edward Tufte called "small multiples": breaking out the data into multiple, identical plots by some categorical classifier in order to show trends more effectively.

The bar plot we were just looking at is quite busy, even without displaying all 36 CATA questions. Instead, let's see how we can break out separate plots, for example, different CATA attributes into "small multiple" facet plots to get a look at trends between berries one attribute at a time.

```
berry_cata_long %>%
  filter(Presence == 1) %>%
  ggplot(aes(x = berry)) +
  geom_bar() +
  theme_bw() +
  facet_wrap(~ Attribute) +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```

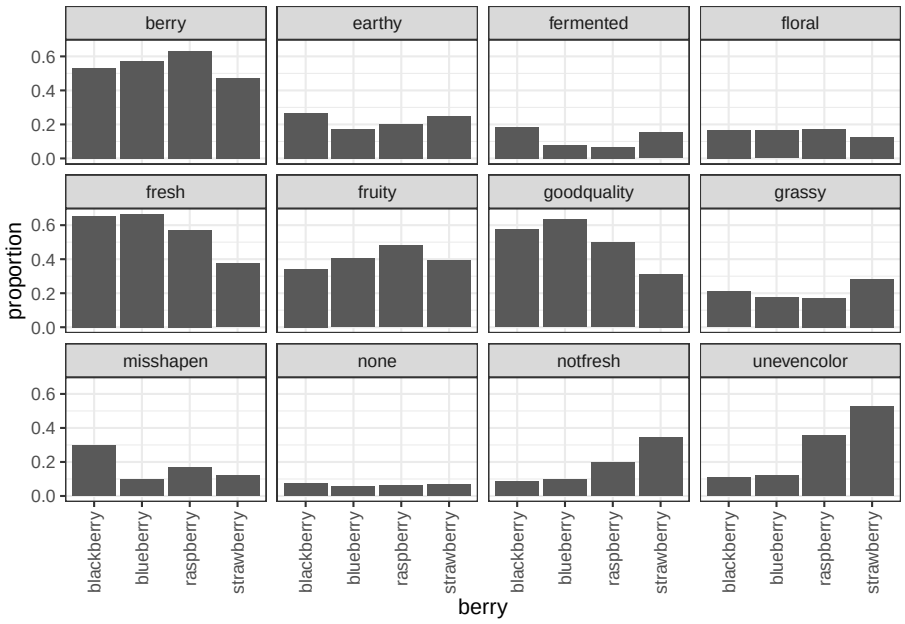


We can still compare the facets in this case, because they all share X and Y axes. The "none" attribute was checked much less often than the other attributes, for example. We can also see that uneven color was a more common problem among the raspberries and strawberries than the blueberries and blackberries, and that strawberries and blackberries more commonly had fermented flavor.

It would be good to go a step further and plot the percentages, rather than the raw counts, since not every berry had the exact same number of participants. We can use `geom_col()` instead of `geom_bar()` to do our own summarizing:

```
berry_cata_long %>%
  group_by(berry, Attribute, Modality) %>%
  summarize(proportion = mean(Presence)) %>%
  ggplot(aes(x = berry, y = proportion)) +
  geom_col() +
  theme_bw() +
  facet_wrap(~ Attribute) +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```

```
## `summarise()` has grouped output by 'berry', 'Attribute'. You can override
## using the `.groups` argument.
```

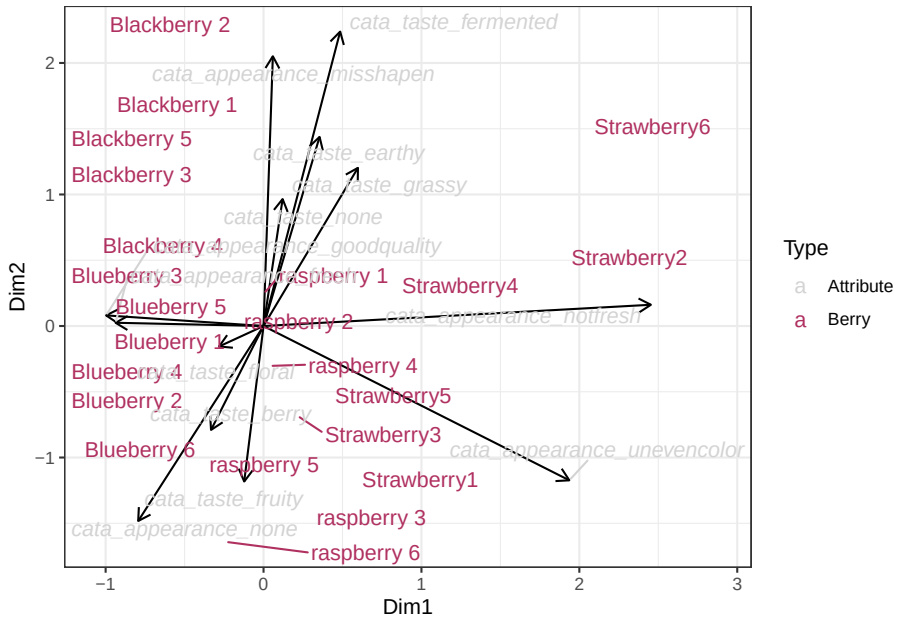


Both plots show that blueberries and raspberries are more commonly described by "berry flavor", but looking at the proportions instead of the raw counts reveals that there aren't strong differences in floral flavor across the berry types.

6.2.8 The ggplot rabbithole

Like many things we're introducing today, you can make infinitely-complicated graphs using these same basic semantics. `ggplot2` is a world of exceptions! You could eventually end up having to do something like this, where each `geom_*` has different `data`:

```
ggplot() +
  geom_segment(aes(xend = Dim1, yend = Dim2), x = 0, y = 0,
               arrow = arrow(length = unit(0.25, "cm")),
               data = berry_col_coords) +
  geom_text_repel(aes(x = Dim1, y = Dim2, label = Variable, color = Type,
                      fontface = ifelse(Type == "Attribute", "italic", "plain")),
                  data = berry_ca_coords) +
  scale_color_manual(breaks = c("Attribute","Berry"),
                     values = c("lightgrey","maroon")) +
  theme_bw()
```



It's the fact that the arrows need `xend` and `yend` instead of `x` and `y` like the text, as well as the fact that there are only arrows for half the data, that make it easier to give each geom its own `data`. There are simpler (and possibly better) ways to display the same information as this plot, which we'll cover next.

If you ever find yourself tearing your hair out over a complicated plot, remember this section. Some resources you may find helpful for further reading and troubleshooting include:

1. Kieran Healy's "Data Visualization: a Practical Introduction".
2. The plotting section of R for Data Science.
3. Hadley Wickham's core reference textbook on ggplot2.

6.3 Better CA plots with ggplot2

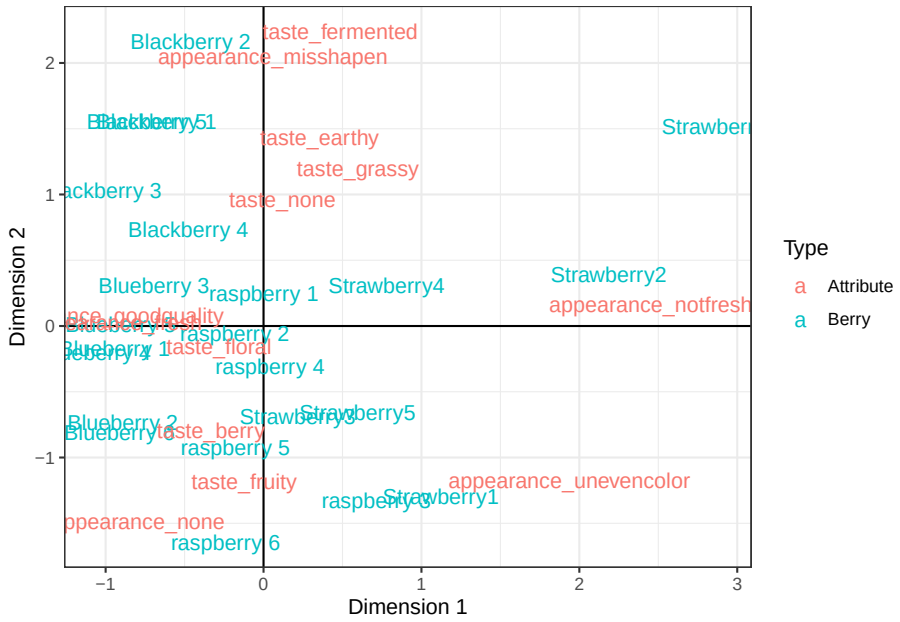
We need to think about what we want to plot. The CA maps are **scatterplots** where each point is a **sample** (row variable) or **attribute** (column variable). These coordinates are in the list that the `ca()` function outputs, but in two separate matrices. We already learned how to combine them into *one* where the columns are *axes* and the rows are the *variables* from our initial dataset, with a column specifying whether each row is a sample or an attribute variable.

```
berry_ca_coords
```

```
## # A tibble: 36 x 14
##    Type  Variable     Dim1    Dim2    Dim3   Dim4    Dim5   Dim6     Dim7   Dim8
##    <chr> <chr>       <dbl>   <dbl>   <dbl>  <dbl>   <dbl>  <dbl>    <dbl>  <dbl>
##  1 Berry Blackberr~ -0.676  1.56    0.0289 -2.12   0.421   0.416 -0.0872   0.555
##  2 Berry Blackberr~ -0.461  2.16    0.445  -0.676 -0.401   1.35  -0.144    0.318
##  3 Berry Blackberr~ -1.03   1.03    0.589  -1.55   0.149  -0.154 -0.00916  0.173
##  4 Berry Blackberr~ -0.476  0.738  -2.14    1.80  -1.05    0.604  1.12     1.62
##  5 Berry Blackberr~ -0.738  1.55   -2.29   -0.377 -1.48   -0.653 -0.503   -0.956
##  6 Berry Blueberry~ -0.942 -0.167   1.35    0.972  0.342  -0.709  0.276    0.509
##  7 Berry Blueberry~ -0.891 -0.735   0.469   1.36  -0.0587  3.08   0.538   -1.17
##  8 Berry Blueberry~ -0.693  0.308   0.498   0.267  1.45   -1.53   0.767   -0.176
##  9 Berry Blueberry~ -1.06  -0.198   0.540  -0.601  1.45   -0.122  0.607   -0.266
## 10 Berry Blueberry~ -0.904  0.0126  0.296   1.32   1.25    0.821  0.475    1.31
## # i 26 more rows
## # i 4 more variables: Dim9 <dbl>, Dim10 <dbl>, Dim11 <dbl>, Dim12 <dbl>
```

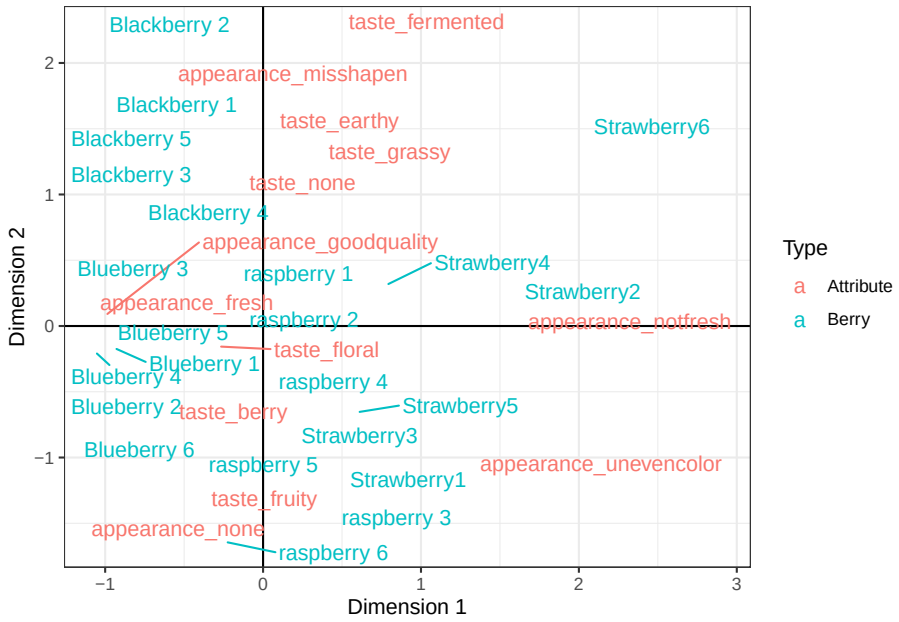
This will get us pretty far.

```
berry_ca_coords %>%
  mutate(Variable = str_remove(Variable, "cata_")) %>%
  ggplot(aes(x = Dim1, y = Dim2, color = Type, label = Variable)) +
  geom_hline(color="black", yintercept = 0) +
  geom_vline(color="black", xintercept = 0) +
  geom_text() +
  theme_bw() +
  xlab("Dimension 1") +
  ylab("Dimension 2")
```



`geom_text()` is similar to `geom_point()`, but instead of having a point with a given **`shape`**, it places **`text`** on the plot which you can pull directly from your data using the **`label`** aesthetic. We can make this even more readable using `geom_text_repel()`, a very similar geom out of the `ggrepel` package:

```
berry_ca_coords %>%
  mutate(Variable = str_remove(Variable, "cata_")) %>%
  ggplot(aes(x = Dim1, y = Dim2, color = Type, label = Variable)) +
  geom_hline(color="black", yintercept = 0) +
  geom_vline(color="black", xintercept = 0) +
  geom_text_repel() +
  theme_bw() +
  xlab("Dimension 1") +
  ylab("Dimension 2")
```

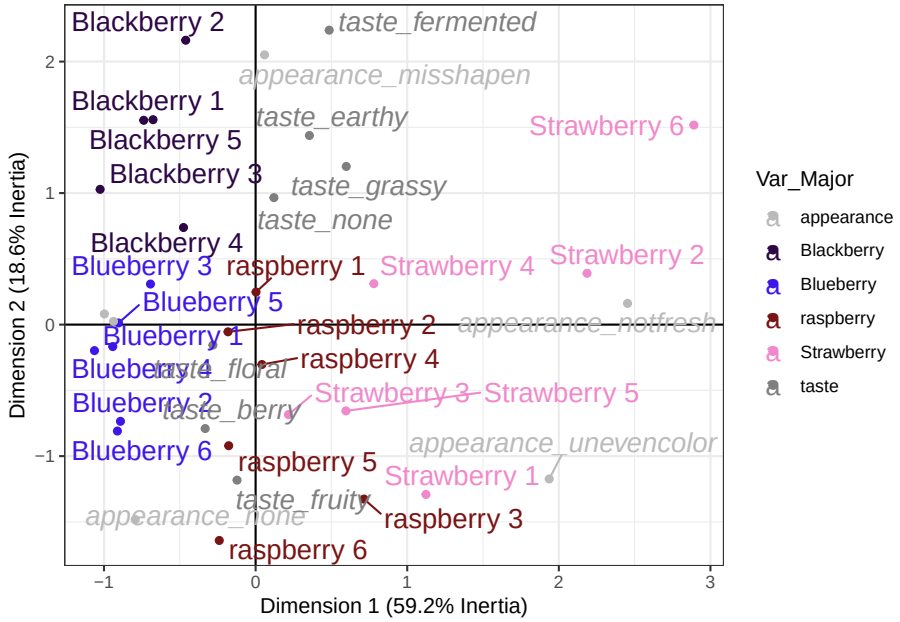


With a little bit of extra work, we can also add the % inertia to each dimension label, tweak the colors with `color_scale_manual`, and make the text a bit bigger.

```
berry_ca_res$sv %>%
  {str_c("Dimension ", 1:length(.), " (", round(100 * .^2 / sum(.^2), 1), "% Inertia)")} ->
  berry_cata_ca_dimnames

berry_ca_coords %>%
  mutate(Variable = str_remove(Variable, "cata_"),
         Variable = str_replace(Variable, "Strawberry", "Strawberry "),
         font = ifelse(Type == "Attribute", "italic", "plain")) %>%
  separate(Variable, c("Var_Major", "Var_Minor"), remove = FALSE) %>%
  ggplot(aes(x = Dim1, y = Dim2, fontface = font, color = Var_Major, label = Variable)) +
  geom_hline(color="black", yintercept = 0) +
  geom_vline(color="black", xintercept = 0) +
  geom_point() +
  geom_text_repel(size = 5) +
  theme_bw() +
  xlab(berry_cata_ca_dimnames[1]) +
  ylab(berry_cata_ca_dimnames[2]) +
  scale_color_manual(values = c("appearance" = "#bababa",
                                "taste" = "#7f7f7f",
                                "Blackberry" = "#2d0645",
                                "Blueberry" = "#3e17e8",
```

```
                                    "raspberry" = "#7a1414",
                                    "Strawberry" = "#f089cb"))
```

```
## Warning: ggrepel: 2 unlabeled data points (too many overlaps). Consider
## increasing max.overlaps
```



This plot still isn't really what I'd call "publication ready". A lot of the final tweaking will depend on the exact size you want, but regardless I'd probably continue adjusting the labels, zoom the plot out a bit, and consider only displaying CATA terms with a high enough `berry_ca_res$colinertia` so the plot was a bit less cluttered.

You can tweak forever. And I'd encourage you to go ahead and try to do whatever you can think of, right now, to make this graph more readable!

For now, this is the culmination of your new data-wrangling, analysis, and graphing skills. We can see which berries are more fresh-tasting and have no notable appearance attributes (blueberries and blackberries), which berries are the worst-looking (strawberries 2 and 6), and we could identify berries anywhere along our roughly earthy/fermented to fruity/berry Dimension 2 (from blackberry 2 to raspberries 3 and 6).

This is also the same set of skills that you'll need for PCA, MDS, DISTATIS, ANOVA, text analysis, and any other computational or statistical task in R.

Chapter 7

Wrap-up and further resources

Let's look back at what we were aiming to do today:

> In this tutorial, we will introduce the audience to the R statistical programming environment and the RStudio Interactive Development Environment (IDE) with the aim of developing sufficient basic skills to conduct multivariate analyses (like Correspondence Analysis) on sensory and consumer datasets. We will provide a learning dataset for the analysis—a set of free response comments and overall liking scores from a central location test on berries. We will teach participants how to import, manipulate, and plot data using user-friendly, "tidy" R programming. All resources used in the tutorial are open-source and will remain available to attendees, including an R script covering the full workflow.
>
> At the end of the tutorial, attendees will be able to prepare raw sensory data for common multivariate analyses or visual representations in R.

We have managed to touch on all of these topics, but of course we have taken the most cursory look at each. I hope what we've gone over today has inspired you, sure, but I mostly hope it has shown you **how much you can do with just a little knowledge**. My journey in terms of learning data science with `R` has been all about building my coding ability incrementally. My code looks more like this than anything else, but I am able to get so much done:

By developing your ability to code (in `R` or Python, or whatever works for you– Julia?) you will open up a whole set of analyses that you would otherwise be unable to access.

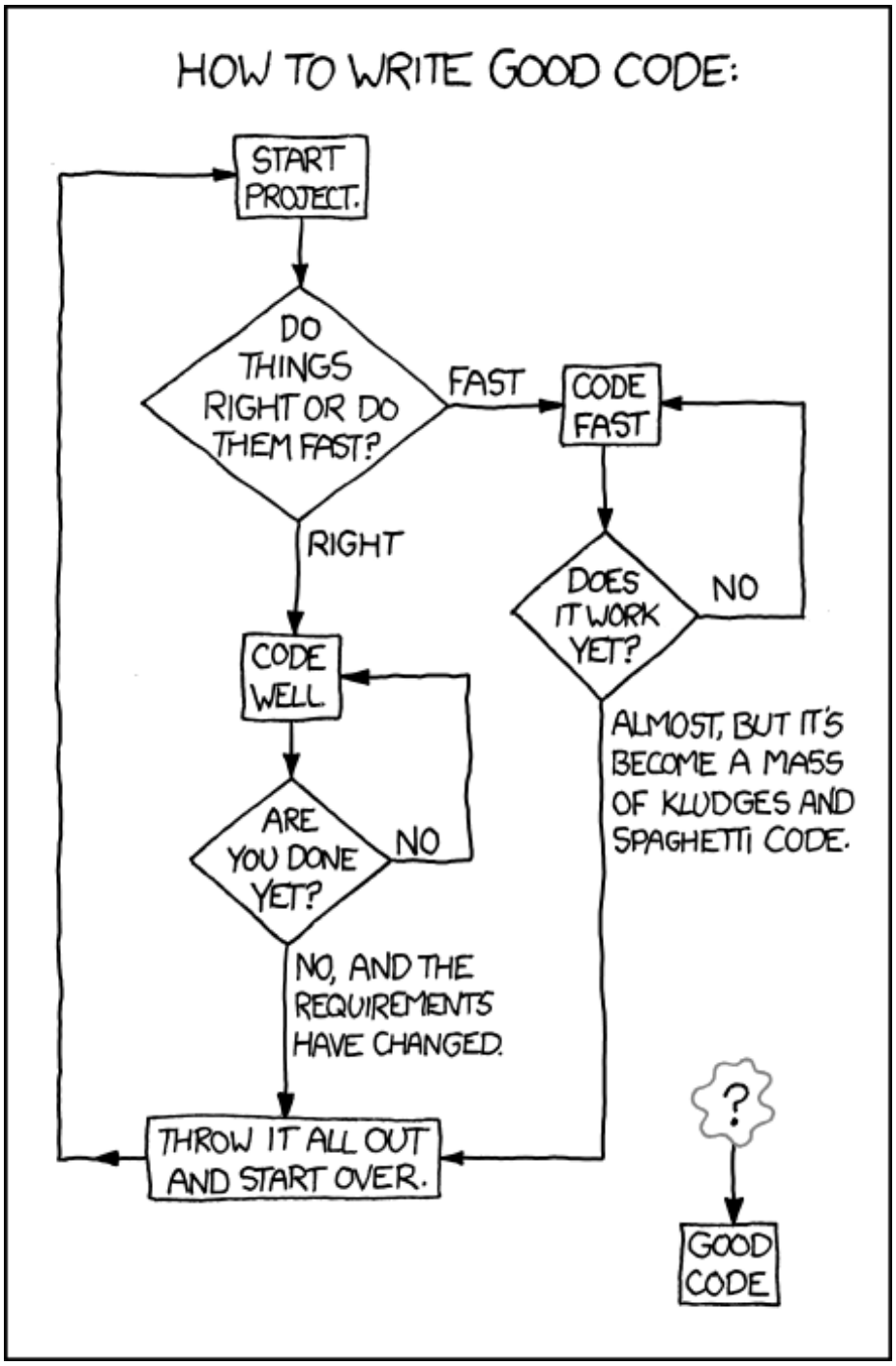


Figure 7.1: What does good code even look like? (via XKCD)

7.1 Getting help

1. Look up the help file for whatever you're doing. Do this by using the syntax `?<search item>` (for example `?c` gets help on the vector command) as a shortcut on the console.
2. Search the help files for a term you think is related. Can't remember the command for making a sequence of integers? Go to the "Help" pane in RStudio and search in the search box for "sequence". See if some of the top results get you what you need.
3. The internet. Seriously. I am not kidding even a little bit. R has one of the most active and (surprisingly) helpful user communities I've ever encountered. Try going to google and searching for "How do I make a sequence of numbers in R?" You will find quite a bit of useful help. I find the following sites particularly helpful
    1. Stack Overflow
    2. Cross Validated/Stack Exchange
    3. Seriously, Google will get you most of the way to helpful answers for many basic R questions.

I want to emphasize that **looking up help is normal**. I do it all the time. Learning to ask questions in helpful ways, how to quickly parse the information you find, and how to slightly alter the answers to suit your particular situation are key skills.

7.2 Learning more with Sensometrics Society

This workshop was organized and sponsored by the Sensometrics Society. Want to learn more? We are hosting our biennial conference **in Paris 3-6 June, 2024**:

In general, Sensometrics has a focus on methods and skills for the analysis of sensory data. If you're interested in contributing to this focus (or just participating), please see our call for papers:

> Please submit an abstract for either an oral or poster presentation by following the instructions on the conference website through. All submissions need to be received by **December 15, 2023**. All accepted contributions are invited to submit a full paper for inclusion in a virtual special issue of Food Quality and Preference (FQAP).

7.3 Further reading/resources

1. General `R` programming

Figure 7.2: Sensometrics Society biannual conference banner

    1. Data Carpentry's *R for Social Scientists* (and, really the courses from The Carpentries in general)
    2. Wickham & Grolemund's *R for Data Science*
    3. The stat545 course website
    4. Healy's *Data Visualization*
    5. My own (somewhat opinionated and eccentric) course from VT: FST 5984

2. Text analysis

    1. Jurafsky & Martin's seminal textbook, *Speech & Language Processing*
    2. Silge & Robinson's *Text Mining with R*
    3. Chollet et al.'s *Deep Learning with R*

We will also be presenting more coding demonstrations in R this week, as part of the "Applying Natural Language Processing tools for sensory and consumer data" workshop from 15:45-17:15 on August 23rd in Room 200! We'd love to see you there if you're at all interested in working with text data (like the free response comments in the `berry_data`).

7.4 Questions/Comments

If you get stuck, feel free to find us during the conference, or email us at **jlahne at vt dot edu** and **lhamilton at vsu dot edu**. I'd love to learn about what you're working on!