Introduction to Statistics and Data Science

Elizabeth Tipton Arend M Kuyper Danielle Sass
Kaitlyn G. Fitzgerald Adapted from ModernDive by
Chester Ismay and Albert Y. Kim

2022-09-22

Table of contents

Welcome

This is the website for **Introduction to Statistics and Data Science**. This book starts you down the path of learning how to think with data using R. You'll learn the basics of how to engage, explore, and examine many types of data arising from several contexts. Hopefully you'll have fun and see how valuable it is to be able to critically think with data.

> ⚠ Warning
>
> Please note that this is a "development version" of this book for the new design of STAT 202. Meaning this is a work in progress being edited and updated as we go.
> We would appreciate any feedback on typos and errors.

This open textbook is produced with support from Northwestern University Libraries and The Alumnae of Northwestern University.

Figure 1

Figure 2

License

Preface

Help! I'm new to R and RStudio and I need to learn about them! However, I'm completely new to coding! What do I do?

Figure 3

Figure 4

If you're asking yourself this question, then you've come to the right place! Start with our "Introduction for Students".

Introduction for students

This book assumes no prerequisites: no algebra, no calculus, and no prior programming/coding experience. This is intended to be a gentle introduction to the practice of analyzing data and answering questions using data the way statisticians, data scientists, data journalists, and other researchers would.

In Figure 5 we present a flowchart of what you'll cover in this book. You'll first get started with data in Chapter 1, where you'll learn about the difference between R and RStudio, start coding in R, understand what R packages are, and explore your first dataset: all domestic departure flights from a New York City airport in 2013. Then

1. **Data Exploration**: You'll assemble your data science toolbox using `tidyverse` packages. In particular:
    - Ch. 2: Visualizing data via the `ggplot2` package.
    - Ch. 3: Wrangling data via the `dplyr` package.
    - Ch. 4: Understanding the concept of "tidy" data as a standardized data input format for all packages in the `tidyverse`
2. **Data Modeling**: Using these data science tools, you'll start performing data modeling. In particular:

- Ch. 5: Constructing basic regression models.
- Ch. 6: Constructing multiple regression models.

3. **Statistical Theory**: Now you'll learn about the role of randomization in making inferences and the general frameworks used to make inferences in statistics. In particular:
   - Ch. 7: Randomization and causality.
   - Ch. 8: Populations and generalizability.
   - Ch. 9: Sampling distributions.
4. **Statistical Inference**: You'll learn to combine your newly acquired data analysis and modeling skills with statistical theory to make inferences. In particular:
   - Ch. 10: Building confidence intervals.
   - Ch. 11: Calculating p-values.
   - Ch. 12: Conducting hypothesis tests.

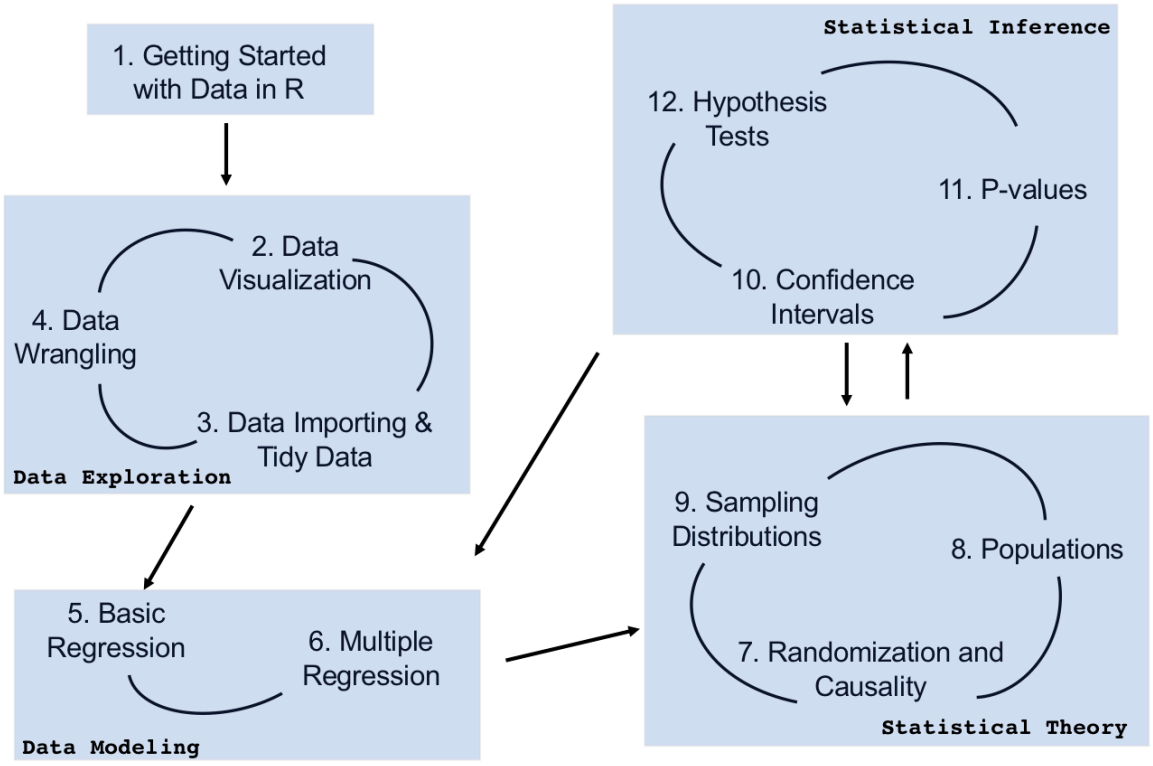


Figure 5: Course Flowchart

What you will learn from this book

We hope that by the end of this book, you'll have learned

1. How to use R to explore data.
2. How to generate research questions and hypotheses.
3. How to think like a statistician and the role of chance in your data.
4. How to answer statistical questions using tools like confidence intervals and hypothesis tests.
5. How to effectively create "data stories" using these tools.

What do we mean by data stories? We mean any analysis involving data that engages the reader in answering questions with careful visuals and thoughtful discussion, such as How strong is the relationship between per capita income and crime in Chicago neighborhoods? and How many f**ks does Quentin Tarantino give (as measured by the amount of swearing in his films)?. Further discussions on data stories can be found in this Think With Google article.

For other examples of data stories constructed by students like yourselves, look at the final projects for two courses that have previously used a version of this book:

- Middlebury College MATH 116 Introduction to Statistical and Data Sciences using student collected data.
- Pacific University SOC 301 Social Statistics using data from the fivethirtyeight R package.

This book will help you develop your "data science toolbox", including tools such as data visualization, data formatting, data wrangling, and data modeling using regression. With these tools, you'll be able to perform the entirety of the "data/science pipeline" while building data communication skills.

In particular, this book will lean heavily on data visualization. In today's world, we are bombarded with graphics that attempt to convey ideas. We will explore what makes a good graphic and what the standard ways are to convey relationships with data. You'll also see the use of visualization to introduce concepts like mean, median, standard deviation, distributions, etc. In general, we'll use visualization as a way of building almost all of the ideas in this book.

To impart the statistical lessons in this book, we have intentionally minimized the number of mathematical formulas used and instead have focused on developing a conceptual understanding via data visualization, statistical computing, and simulations. We hope this is a more intuitive experience than the way statistics has traditionally been taught in the past and how it is commonly perceived.

Finally, you'll learn the importance of literate programming. By this we mean you'll learn how to write code that is useful not just for a computer to execute but also for readers to understand exactly what your analysis is doing and how you did it. This is part of a greater effort to encourage reproducible research (see subsection *Reproducible research* for more details). Hal Abelson coined the phrase that we will follow throughout this book:

> "Programs must be written for people to read, and only incidentally for machines to execute."

We understand that there may be challenging moments as you learn to program. We still continue to struggle and find ourselves often using web searches to find answers and reach out to colleagues for help. In the long run though, we all can solve problems faster and more elegantly via programming. We wrote this book as our way to help you get started and you should know that there is a huge community of R users that are always happy to help everyone along as well. This community exists in particular on the internet on various forums and websites such as stackoverflow.com.

Data/science pipeline

You may think of statistics as just being a bunch of numbers. We commonly hear the phrase "statistician" when listening to broadcasts of sporting events. Statistics (in particular, data analysis), in addition to describing numbers like with baseball batting averages, plays a vital role in all of the sciences. You'll commonly hear the phrase "statistically significant" thrown around in the media. You'll see articles that say "Science now shows that chocolate is good for you." Underpinning these claims is data analysis and a theoretical model relating the data collected in a sample to a larger population. By the end of this book, you'll be able to better understand whether these claims should be trusted or whether we should be wary. Inside data analysis are many sub-fields that we will discuss throughout this book (though not necessarily in this order):

- data collection
- data wrangling
- data visualization
- data modeling
- statistical inference
- correlation and regression
- interpretation of results
- data communication/storytelling

These sub-fields are summarized in what Grolemund and Wickham term the "Data/Science Pipeline" in Figure 6.

We will begin by digging into the gray **Understand** portion of the cycle with data visualization, then with a discussion on what is meant by tidy data and data wrangling, and then conclude by talking about interpreting and discussing the results of our models via **Communication**. These steps are vital to any statistical analysis. But why should you care about statistics? "Why did they make me take this class?"

There's a reason so many fields require a statistics course. Scientific knowledge grows through an understanding of statistical significance and data analysis. You needn't be intimidated by

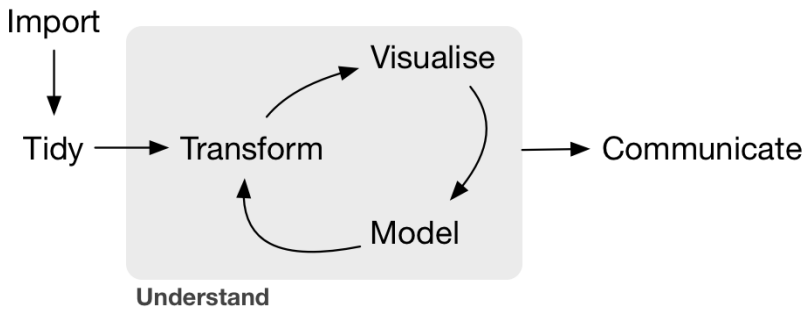


Figure 6: Data/Science Pipeline

statistics. It's not the beast that it used to be and, paired with computation, you'll see how reproducible research in the sciences particularly increases scientific knowledge.

Reproducible research

> "The most important tool is the *mindset*, when starting, that the end product will be reproducible." – Keith Baggerly

Another goal of this book is to help readers understand the importance of reproducible analyses. The hope is to get readers into the habit of making their analyses reproducible from the very beginning. This means we'll be trying to help you build new habits. This will take practice and be difficult at times. You'll see just why it is so important for you to keep track of your code and well-document it to help yourself later and any potential collaborators as well.

Copying and pasting results from one program into a word processor is not the way that efficient and effective scientific research is conducted. It's much more important for time to be spent on data collection and data analysis and not on copying and pasting plots back and forth across a variety of programs.

In a traditional analysis if an error was made with the original data, we'd need to step through the entire process again: recreate the plots and copy and paste all of the new plots and our statistical analysis into your document. This is error prone and a frustrating use of time. We'll see how to use R Markdown to get away from this tedious activity so that we can spend more time doing science.

> "We are talking about *computational* reproducibility." - Yihui Xie

Reproducibility means a lot of things in terms of different scientific fields. Are experiments conducted in a way that another researcher could follow the steps and get similar results? In this book, we will focus on what is known as **computational reproducibility**. This refers to being able to pass all of one's data analysis, data-sets, and conclusions to someone else and have them get exactly the same results on their machine. This allows for time to be

spent interpreting results and considering assumptions instead of the more error prone way of starting from scratch or following a list of steps that may be different from machine to machine.

Part I

Getting started

1 Getting Started with Data in R

Before we can start exploring data in R, there are some key concepts to understand first:

1. What are R and RStudio?
2. How do I code in R?
3. What are R packages?

We'll introduce these concepts in upcoming Sections 1.1 - 1.3 If you are already somewhat familiar with these concepts, feel free to skip to Section 1.4 where we'll introduce our first data set: all domestic flights departing a New York City airport in 2013. This is a dataset we will explore in depth in this book.

1.1 What are R and RStudio?

For much of this book, we will assume that you are using R via RStudio. First time users often confuse the two. At its simplest:

- R is like a car's engine.
- RStudio is like a car's dashboard.

R: Engine	RStudio: Dashboard
	

More precisely, R is a programming language that runs computations while RStudio is an *integrated development environment (IDE)* that provides an interface by adding many convenient features and tools. So just as having access to a speedometer, rearview mirrors, and a

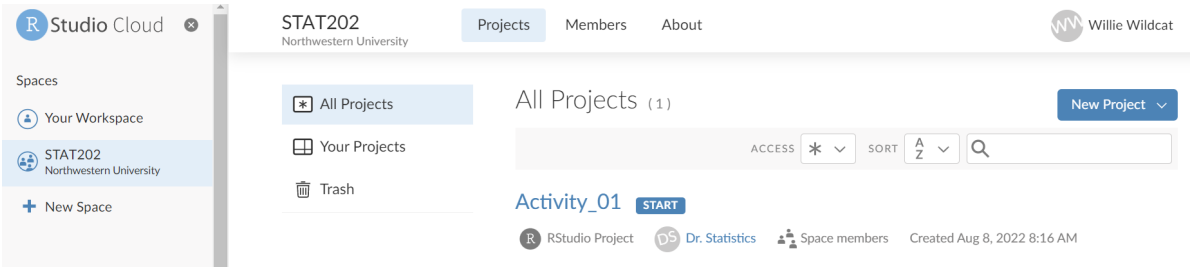
navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

1.1.1 Using RStudio Cloud

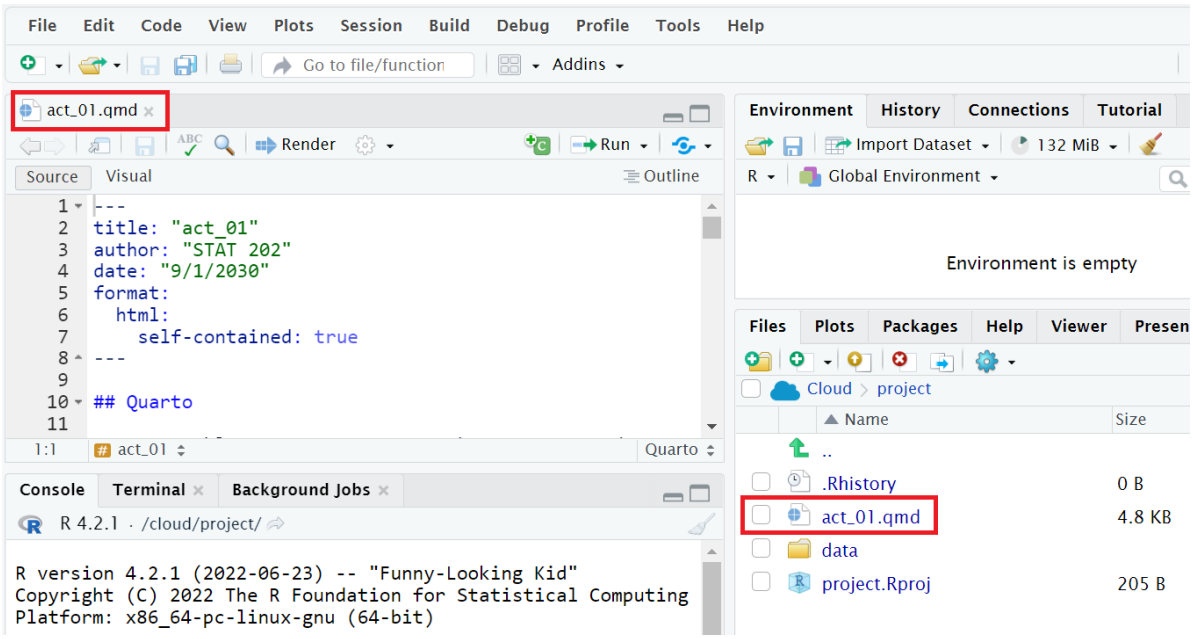
RStudio Cloud (https://rstudio.cloud) is a hosted version of RStudio that allows you to begin coding directly from your browser - there is no software to install and nothing to configure on your computer.

To begin using RStudio Cloud use the link provided by your instructor to gain access to the classroom workspace. You will be prompted to create a free account or log in if you have an existing account.

After you open RStudio Cloud, you should now have access to the classroom under 'Spaces' on the left hand side (in this case 'STAT202').



Throughout this course you will be working on various activities. Once the instructor has made an activity available you will click on the classroom Workspace (STAT202) to access the available projects. To begin working on an activity click 'Start'. Once that activity project is open navigate to the 'File' pane and open the Quarto '.qmd' file.



You can use RStudio Cloud for personal use as well by creating projects in 'Your Workspace'. However, RStudio Cloud limits the number of projects and amount of accessible time so it is recommended that you later install the software on your own computer.

1.1.2 Installing R and RStudio on your personal computer

> **Note about RStudio Server or RStudio Cloud**: If your instructor has provided you with a link and access to RStudio Server or RStudio Cloud, then you can skip this section. We do recommend after a few months of working on RStudio Server/Cloud that you return to these instructions to install this software on your own computer though. You will first need to download and install both R and RStudio (Desktop version) on your computer. It is important that you install R first and then install RStudio second.

1. **You must do this first:** Download and install R.
    - If you are a Windows user: Click on "Download R for Windows", then click on "base", then click on the Download link.
    - If you are macOS user: Click on "Download R for (Mac) OS X", then under "Latest release:" click on R-X.X.X.pkg, where R-X.X.X is the version number. For example, the latest version of R as of August 10, 2019 was R-3.6.1.
2. **You must do this second:** Download and install RStudio.
    - Scroll down to "Installers for Supported Platforms" near the bottom of the page.

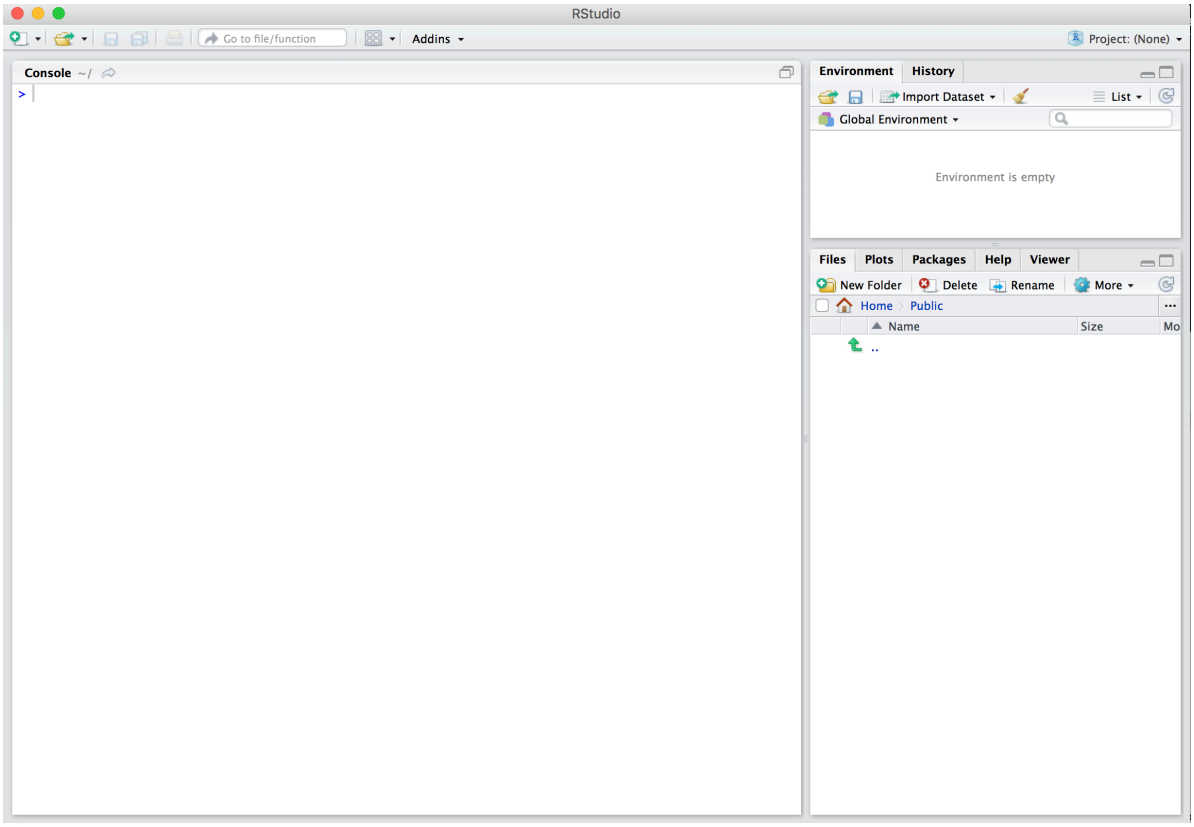
- Click on the download link corresponding to your computer's operating system.

1.1.3 Using R via RStudio

Recall our car analogy from above. Much as we don't drive a car by interacting directly with the engine but rather by interacting with elements on the car's dashboard, we won't be using R directly but rather we will use RStudio's interface. After you install R and RStudio on your computer, you'll have two new programs AKA applications you can open. We will always work in RStudio and not R. In other words:

R: Do not open this	RStudio: Open this
	

After you open RStudio, you should see the following:



Note the three panes, which are three panels dividing the screen: The *Console pane*, the *Files pane*, and the *Environment pane*. Over the course of this chapter, you'll come to learn what purpose each of these panes serve.

1.2 How do I code in R?

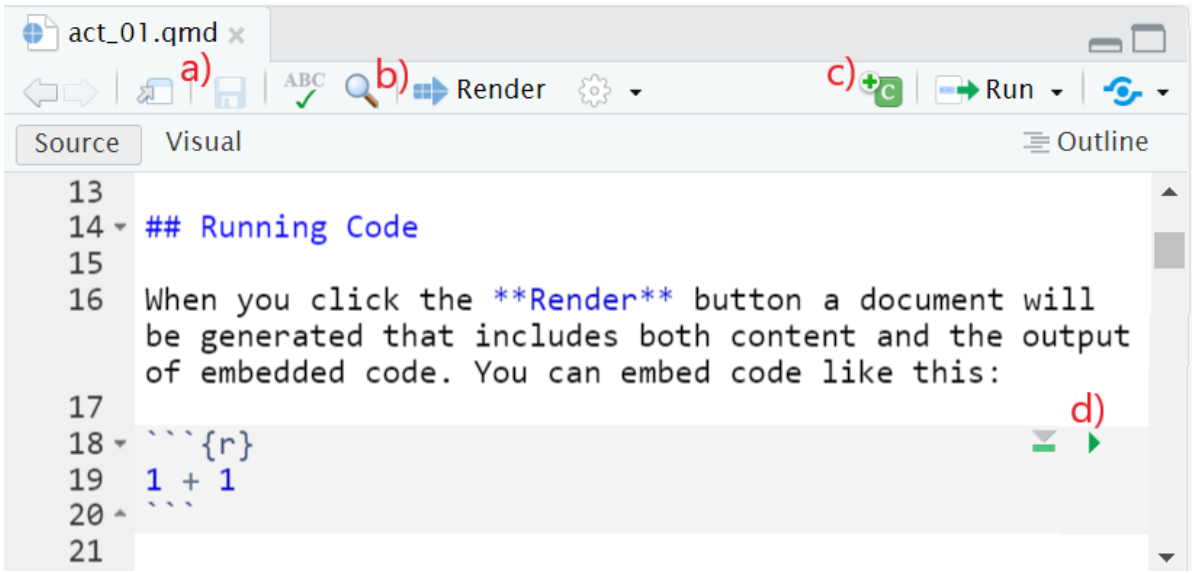
Now that you're set up with R and RStudio, you are probably asking yourself "OK. Now how do I use R?" The first thing to note as that unlike other statistical software programs like Excel, STATA, or SAS that provide point and click interfaces, R is an interpreted language, meaning you have to enter in R commands written in R code. In other words, you have to code/program in R. Note that we'll use the terms "coding" and "programming" interchangeably in this book.

While it is not required to be a seasoned coder/computer programmer to use R, there is still a set of basic programming concepts that R users need to understand. Consequently, while this book is not a book on programming, you will still learn just enough of these basic programming concepts needed to explore and analyze data effectively.

1.2.1 Creating your first Quarto document

Quarto allows you to easily create a document which combines your code, the results from your code, as well as any text that accompanies the analysis. To create a new Quarto file, in RStudio select File>New File>Quarto Document. Then, you will see a window pop-up titled *New Quarto Document*. Here, you specify the type of file you wish to create. HTML is generally the recommended document type since it does not have traditional *page* separators like PDF and Word do. You can also choose a title and author for your document using their respective fields. Finally, select *Create* to create your new Quarto file. You will see it appear as a tab in your RStudio session. Click the *save icon* to save your new document.

The following is an example of a Quarto document:



a) Save your document.
b) Click *Render* to compile your Quarto document into the file type that you specified. The file will be saved in your *Files pane*. This will also save your document.
c) Insert a new code chunk in your document where the cursor is located. You will often have many code chunks in your document.
d) Run the current code chunk.

When you create your Quarto file and *Render* it into a document, the chunks are run in order and any output from them is shown in the document, in the order and location that their respective chunk appears. Sometimes you may wish to type code or analyze data without it printing in the document. If that is the case, you type the code in the *Console* rather than in the *.qmd* file.

While you read through this book, it will be helpful to have a Quarto document open so you can copy code provided and paste it into a code chunk to run.

1.2.2 Basic programming concepts and terminology

We now introduce some basic programming concepts and terminology. Instead of asking you to learn all these concepts and terminology right now, we'll guide you so that you'll "learn by doing." Note that in this book we will always use a different font to distinguish regular text from `computer_code`. The best way to master these topics is, in our opinions, "learning by doing" and lots of repetition.

- Basics:
  - *Console*: Where you enter in commands.

  - *Running code*: The act of telling R to perform an action by giving it commands in the console.
  - *Objects*: Where values are saved in R. In order to do useful and interesting things in R, we will want to *assign* a name to an object. For example we could do the following assignments: `x <- 44 - 20` and `three <- 3`. This would allow us to run `x + three` which would return `27`.
  - *Data types*: Integers, doubles/numerics, logicals, and characters.

In RStudio try typing the following code into the console or code chunk.

```
x <- 44-20
three <- 3
x+three
```

```
[1] 27
```

You should see `x` and `three` appear as stored objects in the *Environment* pane. Anything you store in the *Environment* pane can be referenced and used later. R can also be used as a calculator, notice how it evaluates `x+three`.

- *Vectors*: A series of values. These are created using the `c()` function, where `c()` stands for "combine" or "concatenate". For example: `c(6, 11, 13, 31, 90, 92)`.
- *Factors*: *Categorical data* are represented in R as factors.
- *Data frames*: Data frames are like rectangular spreadsheets: they are representations of datasets in R where the rows correspond to *observations* and the columns correspond to *variables* that describe the observations. We'll cover data frames later in Section Section 1.4.
- *Conditionals*:
  - Testing for equality in R using `==` (and not `=` which is typically used for assignment). Ex: `2 + 1 == 3` compares `2 + 1` to `3` and is correct R code, while `2 + 1 = 3` will return an error.
  - Boolean algebra: `TRUE/FALSE` statements and mathematical operators such as `<` (less than), `<=` (less than or equal), and `!=` (not equal to).
  - Logical operators: `&` representing "and" as well as `|` representing "or." Ex: `(2 + 1 == 3) & (2 + 1 == 4)` returns `FALSE` since both clauses are not `TRUE` (only the first clause is `TRUE`). On the other hand, `(2 + 1 == 3) | (2 + 1 == 4)` returns `TRUE` since at least one of the two clauses is `TRUE`.
- *Functions*, also called *commands*: Functions perform tasks in R. They take in inputs called *arguments* and return outputs. You can either manually specify a function's arguments or use the function's *default values*.

This list is by no means an exhaustive list of all the programming concepts and terminology needed to become a savvy R user; such a list would be so large it wouldn't be very useful, especially for novices. Rather, we feel this is a minimally viable list of programming concepts and terminology you need to know before getting started. We feel that you can learn the rest as you go. Remember that your mastery of all of these concepts and terminology will build as you practice more and more.

1.2.3 Errors, warnings, and messages

One thing that intimidates new R and RStudio users is how it reports *errors*, *warnings*, and *messages*. R reports errors, warnings, and messages in a glaring red font, which makes it seem like it is scolding you. However, seeing red text in the console is not always bad.

R will show red text in the console pane in three different situations:

- **Errors**: When the red text is a legitimate error, it will be prefaced with "Error in…" and try to explain what went wrong. Generally when there's an error, the code will not run. For example, we'll see in Subsection 1.3.3 if you see `Error in ggplot(...) : could not find function "ggplot"`, it means that the `ggplot()` function is not accessible because the package that contains the function (`ggplot2`) was not loaded with `library(ggplot2)`. Thus you cannot use the `ggplot()` function without the `ggplot2` package being loaded first.
- **Warnings**: When the red text is a warning, it will be prefaced with "Warning:" and R will try to explain why there's a warning. Generally your code will still work, but with some caveats. For example, you will see in Chapter 2 if you create a scatterplot based on a dataset where one of the values is missing, you will see this warning: `Warning: Removed 1 rows containing missing values (geom_point)`. R will still produce the scatterplot with all the remaining values, but it is warning you that one of the points isn't there.
- **Messages**: When the red text doesn't start with either "Error" or "Warning", it's *just a friendly message*. You'll see these messages when you load *R packages* in the upcoming Subsection 1.3.2 or when you read data saved in spreadsheet files with the `read_csv()` function as you'll see in Chapter 4. These are helpful diagnostic messages and they don't stop your code from working. Additionally, you'll see these messages when you install packages too using `install.packages()`.

Remember, when you see red text in the console, *don't panic*. It doesn't necessarily mean anything is wrong. Rather:

- If the text starts with "Error", figure out what's causing it. Think of errors as a red traffic light: something is wrong!

- If the text starts with "Warning", figure out if it's something to worry about. For instance, if you get a warning about missing values in a scatterplot and you know there are missing values, you're fine. If that's surprising, look at your data and see what's missing. Think of warnings as a yellow traffic light: everything is working fine, but watch out/pay attention.
- Otherwise the text is just a message. Read it, wave back at R, and thank it for talking to you. Think of messages as a green traffic light: everything is working fine.

1.2.4 Tips on learning to code

Learning to code/program is very much like learning a foreign language, it can be very daunting and frustrating at first. Such frustrations are very common and it is very normal to feel discouraged as you learn. However just as with learning a foreign language, if you put in the effort and are not afraid to make mistakes, anybody can learn.

Here are a few useful tips to keep in mind as you learn to program:

- **Remember that computers are not actually that smart**: You may think your computer or smartphone are "smart," but really people spent a lot of time and energy designing them to appear "smart." Rather you have to tell a computer everything it needs to do. Furthermore the instructions you give your computer can't have any mistakes in them, nor can they be ambiguous in any way.
- **Take the "copy, paste, and tweak" approach**: Especially when learning your first programming language, it is often much easier to taking existing code that you know works and modify it to suit your ends, rather than trying to write new code from scratch. We call this the *copy, paste, and tweak* approach. So early on, we suggest not trying to write code from memory, but rather take existing examples we have provided you, then copy, paste, and tweak them to suit your goals. Don't be afraid to play around!
- **The best way to learn to code is by doing**: Rather than learning to code for its own sake, we feel that learning to code goes much smoother when you have a goal in mind or when you are working on a particular project, like analyzing data that you are interested in.
- **Practice is key**: Just as the only method to improving your foreign language skills is through practice, practice, and practice; so also the only method to improving your coding is through practice, practice, and practice. Don't worry however; we'll give you plenty of opportunities to do so!

1.3 What are R packages?

Another point of confusion with many new R users is the idea of an R package. R packages extend the functionality of R by providing additional functions, data, and documentation. They are written by a world-wide community of R users and can be downloaded for free from the internet. For example, among the many packages we will use in this book are the `ggplot2` package for data visualization in Chapter 2, the `dplyr` package for data wrangling in Chapter 3, and the `moderndive` package that accompanies this book.

A good analogy for R packages is they are like apps you can download onto a mobile phone:

R: A new phone	R Packages: Apps you can download
	

So R is like a new mobile phone: while it has a certain amount of features when you use it for the first time, it doesn't have everything. R packages are like the apps you can download onto your phone from Apple's App Store or Android's Google Play.

Let's continue this analogy by considering the Instagram app for editing and sharing pictures. Say you have purchased a new phone and you would like to share a recent photo you have taken on Instagram. You need to:

1. *Install the app*: Since your phone is new and does not include the Instagram app, you need to download the app from either the App Store or Google Play. You do this once and you're set. You might do this again in the future any time there is an update to the app.
2. *Open the app*: After you've installed Instagram, you need to open the app.

Once Instagram is open on your phone, you can then proceed to share your photo with your friends and family. The process is very similar for using an R package. You need to:

1. *Install the package*: This is like installing an app on your phone. Most packages are not installed by default when you install R and RStudio. Thus if you want to use a package for the first time, you need to install it first. Once you've installed a package, you likely won't install it again unless you want to update it to a newer version.

2. *"Load" the package*: "Loading" a package is like opening an app on your phone. Packages are not "loaded" by default when you start RStudio on your computer; you need to "load" each package you want to use every time you start RStudio.

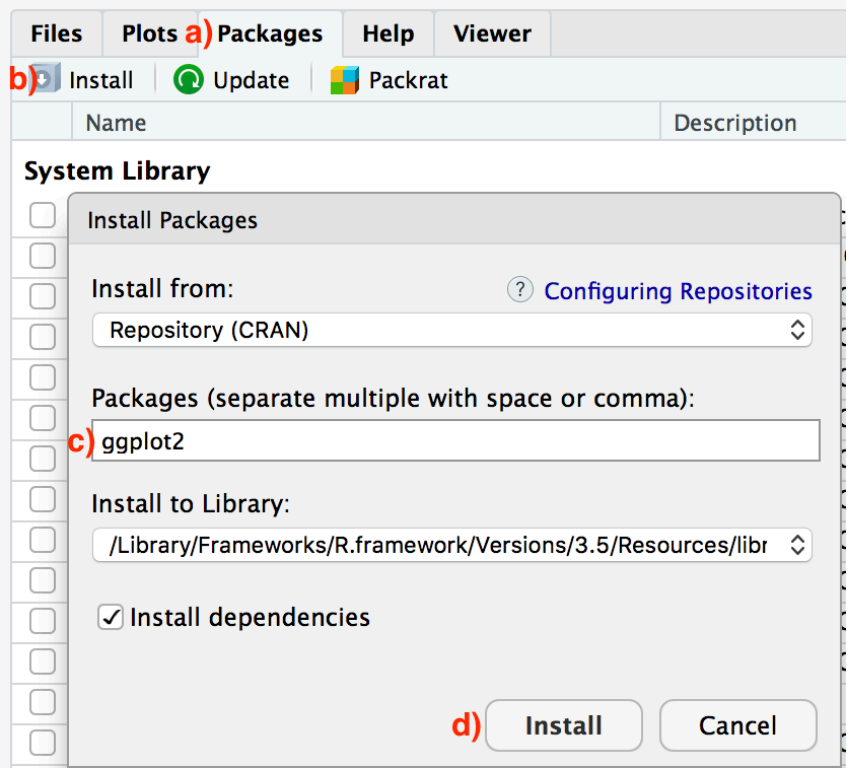
Let's now show you how to perform these two steps for the `ggplot2` package for data visualization.

1.3.1 Package installation

> **Note about RStudio Server/Cloud**: If your instructor has provided you with a link and access to RStudio Server/Cloud, you probably will not need to install packages, as they have likely been pre-installed for you by your instructor. That being said, it is still a good idea to know this process for later on when you are not using RStudio Server/Cloud, but rather RStudio Desktop on your own computer.

There are two ways to install an R package. For example, to install the `ggplot2` package:

1. **Easy way**: In the Files pane of RStudio:
    a) Click on the "Packages" tab
    b) Click on "Install"
    c) Type the name of the package under "Packages (separate multiple with space or comma):" In this case, type `ggplot2`
    d) Click "Install"



2. **Slightly harder way**: An alternative but slightly less convenient way to install a package is by typing `install.packages("ggplot2")` in the Console pane of RStudio and hitting enter. Note you must include the quotation marks.

Much like an app on your phone, you only have to install a package once. However, if you want to update an already installed package to a newer verions, you need to re-install it by repeating the above steps.

> Learning Check 1.1
>
> Repeat the above installing steps for the `dplyr`, `nycflights13`, and `knitr` packages. This will install the earlier mentioned `dplyr` package, the `nycflights13` package containing data on all domestic flights leaving a NYC airport in 2013, and the `knitr` package for writing reports in R.

1.3.2 Package loading

Recall that after you've installed a package, you need to "load" it, in other words open it. We do this by using the `library()` command. For example, to load the `ggplot2` package, run the

following code in the Console pane. What do we mean by "run the following code"? Either type or copy & paste the following code into the Console pane and then hit the enter key.

```
library(ggplot2)
```

If after running the above code, a blinking cursor returns next to the `>` "prompt" sign, it means you were successful and the `ggplot2` package is now loaded and ready to use. If however, you get a red "error message" that reads...

```
Error in library(ggplot2) : there is no package called 'ggplot2'
```

... it means that you didn't successfully install it. In that case, go back to the previous subsection "Package installation" and install it.

> Learning Check 1.2
>
> "Load" the `dplyr`, `nycflights13`, and `knitr` packages as well by repeating the above steps.

1.3.3 Package use

One extremely common mistake new R users make when wanting to use particular packages is that they forget to "load" them first by using the `library()` command we just saw. Remember: *you have to load each package you want to use every time you start RStudio.* If you don't first "load" a package, but attempt to use one of its features, you'll see an error message similar to:

```
Error: could not find function
```

R is telling you that you are trying to use a function in a package that has not yet been "loaded." Almost all new users forget do this when starting out, and it is a little annoying to get used to. However, you'll remember with practice.

1.4 Explore your first dataset

Let's put everything we've learned so far into practice and start exploring some real data! Data comes to us in a variety of formats, from pictures to text to numbers. Throughout this book, we'll focus on datasets that are saved in "spreadsheet"-type format; this is probably the most common way data are collected and saved in many fields. Remember from Subsection 1.2.2

that these "spreadsheet"-type datasets are called *data frames* in R; we will focus on working with data saved as data frames throughout this book.

Let's first load all the packages needed for this chapter, assuming you've already installed them. Read Section 1.3 for information on how to install and load R packages if you haven't already.

```
library(nycflights13)
library(dplyr)
library(knitr)
```

At the beginning of all subsequent chapters in this text, we'll always have a list of packages that you should have installed and loaded to work with that chapter's R code.

1.4.1 `nycflights13` package

Many of us have flown on airplanes or know someone who has. Air travel has become an ever-present aspect in many people's lives. If you live in or are visiting a relatively large city and you walk around that city's airport, you see gates showing flight information from many different airlines. And you will frequently see that some flights are delayed because of a variety of conditions. Are there ways that we can avoid having to deal with these flight delays?

We'd all like to arrive at our destinations on time whenever possible. (Unless you secretly love hanging out at airports. If you are one of these people, pretend for the moment that you are very much anticipating being at your final destination.) Throughout this book, we're going to analyze data related to flights contained in the `nycflights13` package (Wickham 2021). Specifically, this package contains five data sets saved in five separate data frames with information about all domestic flights departing from New York City in 2013. These include Newark Liberty International (EWR), John F. Kennedy International (JFK), and LaGuardia (LGA) airports:

- `flights`: Information on all 336,776 flights
- `airlines`: A table matching airline names and their two letter IATA airline codes (also known as carrier codes) for 16 airline companies
- `planes`: Information about each of 3,322 physical aircraft used.
- `weather`: Hourly meteorological data for each of the three NYC airports. This data frame has 26,115 rows, roughtly corresponding to the $365 \times 24 \times 3 = 26{,}280$ possible hourly measurements one can observe at three locations over the course of a year.
- `airports`: Airport names, codes, and locations for 1,458 destination airports.

1.4.2 `flights` data frame

We will begin by exploring the `flights` data frame that is included in the `nycflights13` package and getting an idea of its structure. Run the following code in your console (either by typing it or cutting & pasting it): it loads in the `flights` dataset into your Console. Note depending on the size of your monitor, the output may vary slightly.

```
flights
```

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
   <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
 1  2013     1     1      517        515       2     830     819      11 UA
 2  2013     1     1      533        529       4     850     830      20 UA
 3  2013     1     1      542        540       2     923     850      33 AA
 4  2013     1     1      544        545      -1    1004    1022     -18 B6
 5  2013     1     1      554        600      -6     812     837     -25 DL
 6  2013     1     1      554        558      -4     740     728      12 UA
 7  2013     1     1      555        600      -5     913     854      19 B6
 8  2013     1     1      557        600      -3     709     723     -14 EV
 9  2013     1     1      557        600      -3     838     846      -8 B6
10  2013     1     1      558        600      -2     753     745       8 AA
# ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>, and abbreviated variable names
#   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
#   5: arr_delay
```

Let's unpack this output:

- `A tibble: 336,776 x 19`: A `tibble` is a kind of data frame used in R. This particular data frame has
  - `336,776` rows
  - `19` columns corresponding to 19 variables describing each observation
- `year month day dep_time sched_dep_time dep_delay arr_time` are different columns, in other words variables, of this data frame.
- We then have the first 10 rows of observations corresponding to 10 flights.
- `... with 336,766 more rows, and 11 more variables:` indicating to us that 336,766 more rows of data and 11 more variables could not fit in this screen.

Unfortunately, this output does not allow us to explore the data very well. Let's look at different tools to explore data frames.

1.4.3 Exploring data frames

Among the many ways of getting a feel for the data contained in a data frame such as `flights`, we present three functions that take as their "argument", in other words their input, the data frame in question. We also include a fourth method for exploring one particular column of a data frame:

1. Using the `View()` function built for use in RStudio. We will use this the most.
2. Using the `glimpse()` function, which is included in the `dplyr` package.
3. Using the `kable()` function, which is included in the `knitr` package.
4. Using the `$` operator to view a single variable in a data frame.

1. `View()`:

Run `View(flights)` in your Console in RStudio, either by typing it or cutting & pasting it into the Console pane, and explore this data frame in the resulting pop-up viewer. You should get into the habit of always `View`ing any data frames that come your way. Note the capital "V" in `View`. R is case-sensitive so you'll receive an error is you run `view(flights)` instead of `View(flights)`.

> Learning Check 1.3
>
> What does any *ONE* row in this `flights` dataset refer to?
>
> a. Data on an airline
> b. Data on a flight
> c. Data on an airport
> d. Data on multiple flights

By running `View(flights)`, we see the different *variables* listed in the columns and we see that there are different types of variables. Some of the variables like `distance`, `day`, and `arr_delay` are what we will call *quantitative* variables. These variables are numerical in nature. Other variables here are *categorical*.

Note that if you look in the leftmost column of the `View(flights)` output, you will see a column of numbers. These are the row numbers of the dataset. If you glance across a row with the same number, say row 5, you can get an idea of what each row corresponds to. In other words, this will allow you to identify what object is being referred to in a given row. This is often called the *observational unit*. The *observational unit* in this example is an individual flight departing New York City in 2013. You can identify the observational unit by determining what "thing" is being measured or described by each of the variables.

2. `glimpse()`:

The second way to explore a data frame is using the `glimpse()` function included in the `dplyr` package. Thus, you can only use the `glimpse()` function after you've loaded the `dplyr` package. This function provides us with an alternative method for exploring a data frame:

```
glimpse(flights)
```

```
Rows: 336,776
Columns: 19
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

We see that `glimpse()` will give you the first few entries of each variable in a row after the variable. In addition, the *data type* (see Subsection 1.2.2) of the variable is given immediately after each variable's name inside `< >`. Here, `int` and `dbl` refer to "integer" and "double", which are computer coding terminology for quantitative/numerical variables. In contrast, `chr` refers to "character", which is computer terminology for text data. Text data, such as the `carrier` or `origin` of a flight, are categorical variables. The `time_hour` variable is an example of one more type of data type: `dttm`. As you may suspect, this variable corresponds to a specific date and time of day. However, we won't work with dates in this class and leave it to a more advanced book on data science.

Learning Check 1.4

What are some examples in this dataset of **categorical** variables? What makes them different than **quantitative** variables?

3. `kable()`:

The another way to explore the entirety of a data frame is using the `kable()` function from the `knitr` package. Let's explore the different carrier codes for all the airlines in our dataset two ways. Run both of these lines of code in your Console:

```
airlines
kable(airlines)
```

At first glance, it may not appear that there is much difference in the outputs. However when using tools for document production such as Quarto, the latter code produces output that is much more legible and reader-friendly.

4. `$` operator

Lastly, the `$` operator allows us to explore a single variable within a data frame. For example, run the following in your console

```
airlines
airlines$name
```

We used the `$` operator to extract only the `name` variable and return it as a vector of length 16. We will only be occasionally exploring data frames using this operator, instead favoring the `View()` and `glimpse()` functions.

1.4.4 Help files

Another nice feature of R is the help system. You can get help in R by entering a `?` before the name of a function or data frame in question and you will be presented with a page showing the documentation. For example, let's look at the help file for the `flights` data frame:

```
?flights
```

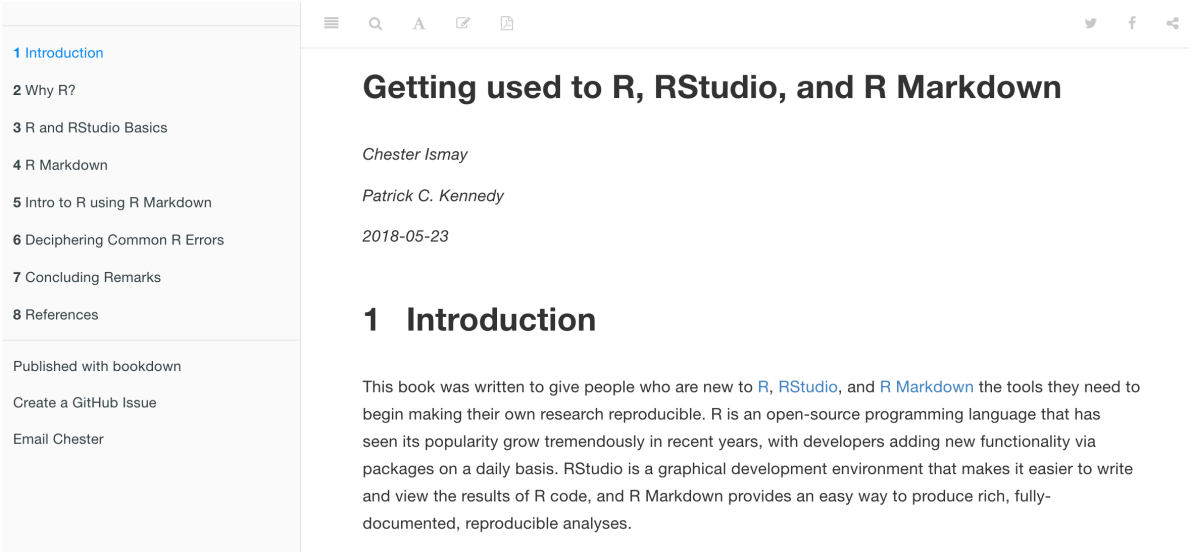
A help file should pop-up in the Help pane of RStudio. If you have questions about a function or data frame included in an R package, you should get in the habit of consulting the help file right away.

1.5 Conclusion

We've given you what we feel are the most essential concepts to know before you can start exploring data in R. Is this chapter exhaustive? Absolutely not. To try to include everything in this chapter would make the chapter so large it wouldn't be useful!

1.5.1 Additional resources

If you are completely new to the world of coding, R, and RStudio and feel you could benefit from a more detailed introduction, we suggest you check out Chester Ismay's short book Getting used to R, RStudio, and R Markdown (Ismay 2016), which includes screencast recordings that you can follow along and pause as you learn. While this book teaches R Markdown it it important to note that everything in R Markdown is transferable to Quarto. R Markdown and Quarto are both tools used for reproducible research but R Markdown is fundamentally tied to R while Quarto is a multi-language platform. For a getting started guide on Quarto, we suggest the Quarto Getting Started webpage



1.6 Practice Problems

> Concept

1. Which type of document do we use to both code and write explanations?

a) R Script
b) Quarto Document
c) HTML file
d) R Notebook

2. Which type of red text in the console pane generally means that your code will not run?

a) error
b) warning
c) message

3. If you place the operator ? before the name of a function or data frame, then you will be presented with a page showing the documentation for the respective function or data frame.

a) TRUE
b) FALSE

4. If you type $8/2 == 4$ into the console, what will the output be?

a) TRUE
b) FALSE
c) NA
d) 0
e) 4

5. If you type $3^2! = 9$ into the console, what will the output be?

a) TRUE
b) FALSE
c) NA
d) 0
e) 9

6. If you type $5 * 3$ into the console, what will the output be?

a) TRUE
b) FALSE
c) NA
d) 8
e) 15

7. What does any ONE row in this flights dataset refer to?

a) Data on an airline
b) Data on a flight

c) Data on an airport
d) Data on multiple flights

8. In the flights dataset, `air_time` and `arr_delay` are which type of variables?

a) string
b) categorical
c) quantitative
d) character
e) dataframe

Application

9. In a code chunk, first define a variable `z` to be the product of 12 and 31, then define a variable called `add_on` to be the number 12. Print the output of `z + add_on`.

10. Consider the `titanic` data set included in the package `ISDSdatasets`. This is one of the most popular data sets used for understanding machine learning basics, and you will likely see this data set in the future if you continue on in your studies to machine learning.

```
Use the `glimpse()` function from the `dplyr` package to explore and describe the dataset.
```

Advanced

For the following problems we will use the `titanic` data set to learn additional data exploration techniques.

11. Use the function `head()` on the `titanic` dataset. What does it do? Based on this, what do you expect the function `tail()` does?

12. The function `unique()`, when used on a specific variable within a data set, returns a vector of the values of the variable with duplicate elements removed. Try using the function `unique()` on the variable `Embarked`.

Part II

Data Exploration via the tidyverse

2 Data Visualization

We begin the development of your data science toolbox with data visualization. By visualizing our data, we gain valuable insights that we couldn't initially see from just looking at the raw data in spreadsheet form. We will use the `ggplot2` package as it provides an easy way to customize your plots. `ggplot2` is rooted in the data visualization theory known as *The Grammar of Graphics* (Wilkinson 2005).

At the most basic level, graphics/plots/charts (we use these terms interchangeably in this book) provide a nice way for us to get a sense for how quantitative variables compare in terms of their center (where the values tend to be located) and their spread (how they vary around the center). Graphics should be designed to emphasize the findings and insight you want your audience to understand. This does however require a balancing act. On the one hand, you want to highlight as many meaningful relationships and interesting findings as possible; on the other you don't want to include so many as to overwhelm your audience.

As we will see, plots/graphics also help us to identify patterns and outliers in our data. We will see that a common extension of these ideas is to compare the *distribution* of one quantitative variable (i.e., what the spread of a variable looks like or how the variable is *distributed* in terms of its values) as we go across the levels of a different categorical variable.

Packages Needed

Let's load all the packages needed for this chapter (this assumes you've already installed them). Read Section 1.3 for information on how to install and load R packages.

```
library(nycflights13)
library(ggplot2)
library(dplyr)
```

2.1 The Grammar of Graphics

We begin with a discussion of a theoretical framework for data visualization known as "The Grammar of Graphics," which serves as the foundation for the `ggplot2` package. Think of how we construct sentences in English to form sentences by combining different elements,

like nouns, verbs, particles, subjects, objects, etc. However, we can't just combine these elements in any arbitrary order; we must do so following a set of rules known as a linguistic grammar. Similarly to a linguistic grammar, "The Grammar of Graphics" define a set of rules for constructing *statistical graphics* by combining different types of *layers*. This grammar was created by Leland Wilkinson (Wilkinson 2005) and has been implemented in a variety of data visualization software including R.

2.1.1 Components of the Grammar

In short, the grammar tells us that:

> **A statistical graphic is a `mapping` of `data` variables to `aesthetic` attributes of `geometric` objects.**

Specifically, we can break a graphic into three essential components:

1. `data`: the data set composed of variables that we map.
2. `geom`: the geometric object in question. This refers to the type of object we can observe in a plot. For example: points, lines, and bars.
3. `aes`: aesthetic attributes of the geometric object. For example, x-position, y-position, color, shape, and size. Each assigned aesthetic attribute can be mapped to a variable in our data set.

You might be wondering why we wrote the terms `data`, `geom`, and `aes` in a computer code type font. We'll see very shortly that we'll specify the elements of the grammar in R using these terms. However, let's first break down the grammar with an example.

2.1.2 Gapminder data

In February 2006, a statistician named Hans Rosling gave a TED talk titled "The best stats you've ever seen" where he presented global economic, health, and development data from the website gapminder.org. For example, for the 142 countries included from 2007, let's consider only the first 6 countries when listed alphabetically in Table 2.1.

Table 2.1: Gapminder 2007 Data: First 6 of 142 countries

Country	Continent	Life Expectancy	Population	GDP per Capita
Afghanistan	Asia	43.8	31889923	975
Albania	Europe	76.4	3600523	5937
Algeria	Africa	72.3	33333216	6223
Angola	Africa	42.7	12420476	4797
Argentina	Americas	75.3	40301927	12779

Country	Continent	Life Expectancy	Population	GDP per Capita
Australia	Oceania	81.2	20434176	34435

Each row in this table corresponds to a country in 2007. For each row, we have 5 columns:

1. **Country**: Name of country.
2. **Continent**: Which of the five continents the country is part of. (Note that "Americas" includes countries in both North and South America and that Antarctica is excluded.)
3. **Life Expectancy**: Life expectancy in years.
4. **Population**: Number of people living in the country.
5. **GDP per Capita**: Gross domestic product (in US dollars).

Now consider Figure 2.1, which plots this data for all 142 countries in the data.

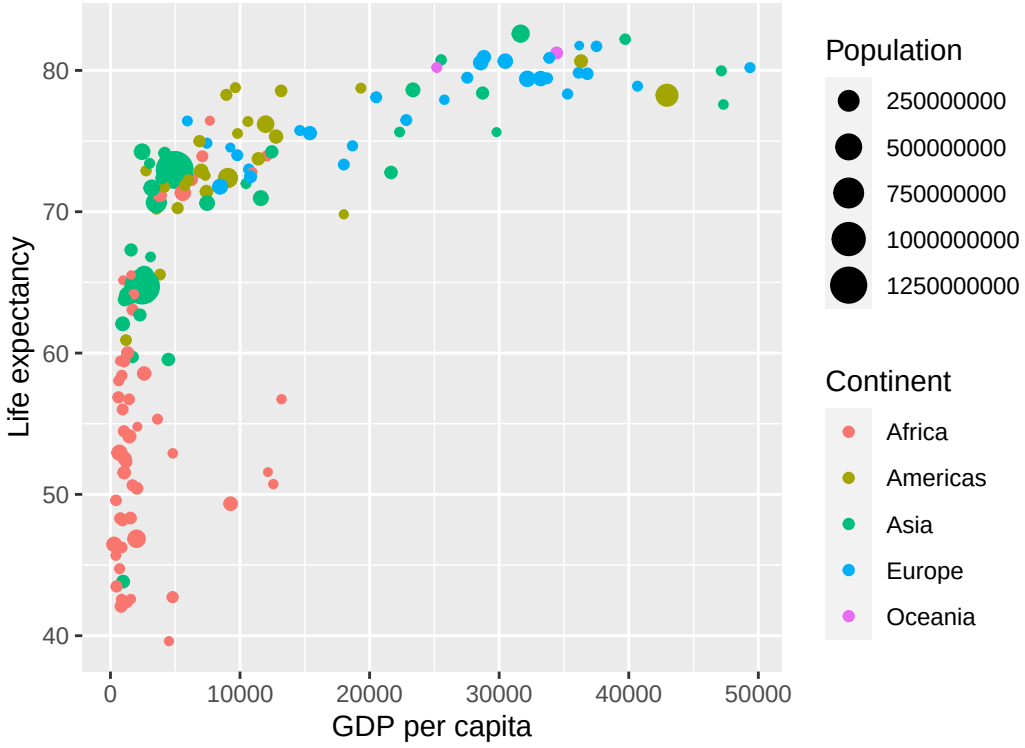


Figure 2.1: Life Expectancy over GDP per Capita in 2007

Let's view this plot through the grammar of graphics:

1. The `data` variable **GDP per Capita** gets mapped to the `x`-position `aes`thetic of the points.

2. The `data` variable **Life Expectancy** gets mapped to the `y`-position `aes`thetic of the points.
3. The `data` variable **Population** gets mapped to the `size` `aes`thetic of the points.
4. The `data` variable **Continent** gets mapped to the `color` `aes`thetic of the points.

We'll see shortly that `data` corresponds to the particular data frame where our data is saved and a "data variable" corresponds to a particular column in the data frame. Furthermore, the type of `geom`etric object considered in this plot are points. That being said, while in this example we are considering points, graphics are not limited to just points. Other plots involve lines while others involve bars.

Let's summarize the three essential components of the Grammar in Table 2.2.

Table 2.2: Summary of Grammar of Graphics for this plot

data variable	aes	geom
GDP per Capita	x	point
Life Expectancy	y	point
Population	size	point
Continent	color	point

2.1.3 Other components

There are other components of the Grammar of Graphics we can control as well. As you start to delve deeper into the Grammar of Graphics, you'll start to encounter these topics more frequently. In this book however, we'll keep things simple and only work with the two additional components listed below:

- `facet`ing breaks up a plot into small multiples corresponding to the levels of another variable (Section 2.6)
- `position` adjustments for barplots (Section 2.8)

Other more complex components like `scales` and `coord`inate systems are left for a more advanced text such as R for Data Science (Grolemund and Wickham 2016). Generally speaking, the Grammar of Graphics allows for a high degree of customization of plots and also a consistent framework for easily updating and modifying them.

2.1.4 ggplot2 package

In this book, we will be using the `ggplot2` package for data visualization, which is an implementation of the Grammar of Graphics for R (Wickham et al. 2022). As we noted earlier, a lot of the previous section was written in a computer code type font. This is because the various

components of the Grammar of Graphics are specified in the `ggplot()` function included in the `ggplot2` package, which expects at a minimum as arguments (i.e. inputs):

- The data frame where the variables exist: the `data` argument.
- The mapping of the variables to aesthetic attributes: the `mapping` argument which specifies the `aes`thetic attributes involved.

After we've specified these components, we then add *layers* to the plot using the `+` sign. The most essential layer to add to a plot is the layer that specifies which type of `geom`etric object we want the plot to involve: points, lines, bars, and others. Other layers we can add to a plot include layers specifying the plot title, axes labels, visual themes for the plots, and facets (which we'll see in Section 2.6.

Let's now put the theory of the Grammar of Graphics into practice.

2.2 Five Named Graphs - The 5NG

In order to keep things simple, we will only focus on five types of graphics in this book, each with a commonly given name. We term these "five named graphs" the **5NG**:

1. scatterplots
2. linegraphs
3. boxplots
4. histograms
5. barplots

We will discuss some variations of these plots, but with this basic repertoire of graphics in your toolbox you can visualize a wide array of different variable types. Note that certain plots are only appropriate for categorical variables and while others are only appropriate for quantitative variables. You'll want to quiz yourself often as we go along on which plot makes sense a given a particular problem or data set.

2.3 5NG#1: Scatterplots

The simplest of the 5NG are *scatterplots*, also called bivariate plots. They allow you to visualize the relationship between two numerical variables. While you may already be familiar with scatterplots, let's view them through the lens of the Grammar of Graphics. Specifically, we will visualize the relationship between the following two numerical variables in the `flights` data frame included in the `nycflights13` package:

1. `dep_delay`: departure delay on the horizontal "x" axis and
2. `arr_delay`: arrival delay on the vertical "y" axis

for Alaska Airlines flights leaving NYC in 2013. This requires paring down the data from all 336,776 flights that left NYC in 2013, to only the 714 *Alaska Airlines* flights that left NYC in 2013.

What this means computationally is: we'll take the `flights` data frame, extract only the 714 rows corresponding to Alaska Airlines flights, and save this in a new data frame called `alaska_flights`. Run the code below to do this:

```
alaska_flights <- flights %>%
  filter(carrier == "AS")
```

For now we suggest you ignore how this code works; we'll explain this in detail in Chapter 3 when we cover data wrangling. However, convince yourself that this code does what it is supposed to by running `View(alaska_flights)`: it creates a new data frame `alaska_flights` consisting of only the 714 Alaska Airlines flights.

We'll see later in Chapter 3 on data wrangling that this code uses the `dplyr` package for data wrangling to achieve our goal: it takes the `flights` data frame and `filter`s it to only return the rows where `carrier` is equal to `"AS"`, Alaska Airlines' carrier code. Other examples of carrier codes include "AA" for American Airlines and "UA" for United Airlines. Recall from Section 1.2 that testing for equality is specified with `==` and not `=`. Fasten your seat belts and sit tight for now however, we'll introduce these ideas more fully in Chapter 3.

> Learning Check 2.1
>
> Take a look at both the `flights` and `alaska_flights` data frames by running `View(flights)` and `View(alaska_flights)`. In what respect do these data frames differ?

2.3.1 Scatterplots via geom_point

Let's now go over the code that will create the desired scatterplot, keeping in mind our discussion on the Grammar of Graphics in Section 2.1. We'll be using the `ggplot()` function included in the `ggplot2` package.

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

Let's break this down piece-by-piece:

- Within the `ggplot()` function, we specify two of the components of the Grammar of Graphics as arguments (i.e. inputs):
    1. The `data` frame to be `alaska_flights` by setting `data = alaska_flights`.

2. The `aes`thetic `mapping` by setting `aes(x = dep_delay, y = arr_delay)`. Specifically:
   - the variable `dep_delay` maps to the `x` position aesthetic
   - the variable `arr_delay` maps to the `y` position aesthetic

- We add a layer to the `ggplot()` function call using the `+` sign. The layer in question specifies the third component of the grammar: the `geom`etric object. In this case the geometric object are points, set by specifying `geom_point()`.

After running the above code, you'll notice two outputs: a warning message and the graphic shown in Figure 2.2. Let's first unpack the warning message:

```
Warning: Removed 5 rows containing missing values (geom_point).
```

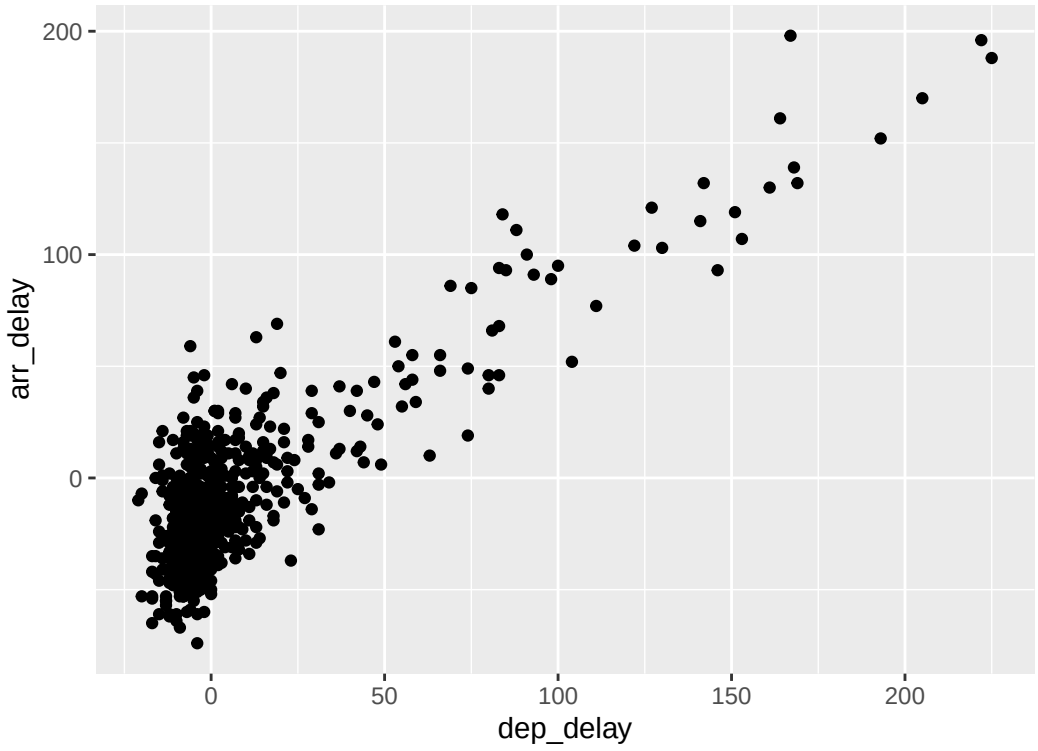


Figure 2.2: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013

After running the above code, R returns a warning message alerting us to the fact that 5 rows were ignored due to them being missing. For 5 rows either the value for `dep_delay` or `arr_delay` or both were missing (recorded in R as `NA`), and thus these rows were ignored in our plot. Turning our attention to the resulting scatterplot in Figure 2.2, we see that a positive

relationship exists between `dep_delay` and `arr_delay`: as departure delays increase, arrival delays tend to also increase. We also note the large mass of points clustered near (0, 0).

Before we continue, let's consider a few more notes on the layers in the above code that generated the scatterplot:

- Note that the `+` sign comes at the end of lines, and not at the beginning. You'll get an error in R if you put it at the beginning.
- When adding layers to a plot, you are encouraged to start a new line after the `+` so that the code for each layer is on a new line. As we add more and more layers to plots, you'll see this will greatly improve the legibility of your code.
- To stress the importance of adding layers in particular the layer specifying the `geom`etric object, consider Figure 2.3 where no layers are added. A not very useful plot!

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay))
```

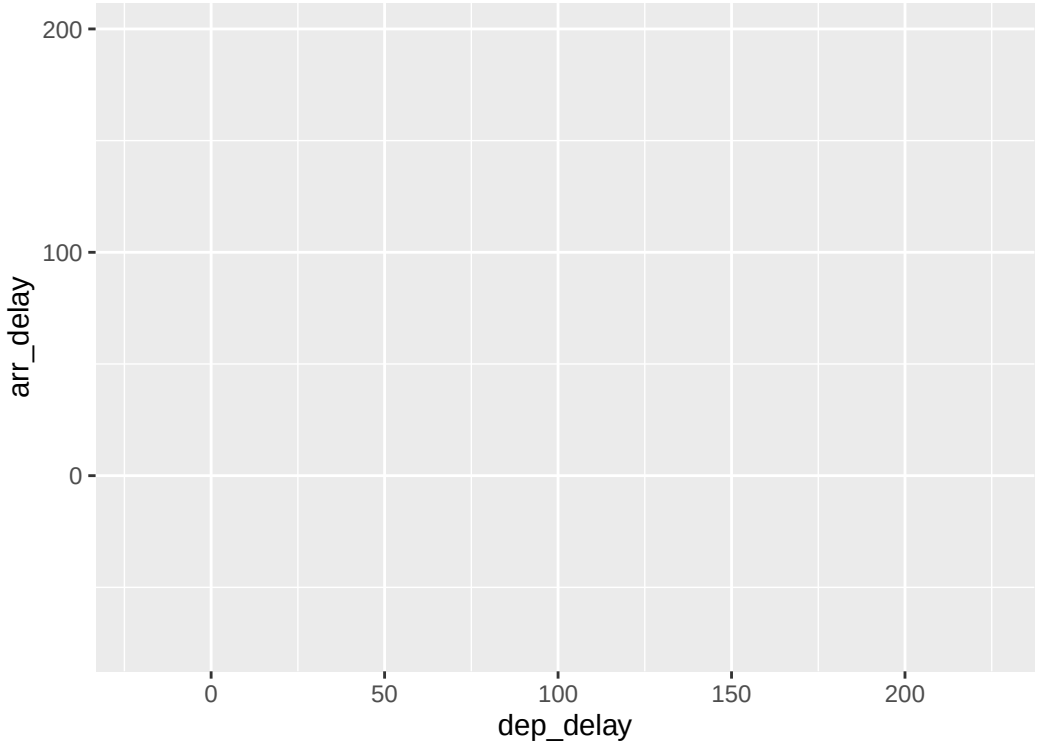


Figure 2.3: Plot with no layers

Learning Check 2.2

What are some practical reasons why `dep_delay` and `arr_delay` have a positive relationship?

Learning Check 2.3

What variables (not necessarily in the `flights` data frame) would you expect to have a negative correlation (i.e. a negative relationship) with `dep_delay`? Why? Remember that we are focusing on numerical variables here.

Learning Check 2.4

Why do you believe there is a cluster of points near (0, 0)? What does (0, 0) correspond to in terms of the Alaskan flights?

Learning Check 2.5

What are some other features of the plot that stand out to you?

Learning Check 2.6

Create a new scatterplot using different variables in the `alaska_flights` data frame by modifying the example above.

2.3.2 Over-plotting

The large mass of points near (0, 0) in Figure 2.2 can cause some confusion as it is hard to tell the true number of points that are plotted. This is the result of a phenomenon called *overplotting*. As one may guess, this corresponds to values being plotted on top of each other *over* and *over* again. It is often difficult to know just how many values are plotted in this way when looking at a basic scatterplot as we have here. There are two methods to address the issue of overplotting:

1. By adjusting the transparency of the points.
2. By adding a little random "jitter", or random "nudges", to each of the points.

Method 1: Changing the transparency

The first way of addressing overplotting is by changing the transparency of the points by using the `alpha` argument in `geom_point()`. By default, this value is set to `1`. We can change this to any value between `0` and `1`, where `0` sets the points to be 100% transparent and `1`

sets the points to be 100% opaque. Note how the following code is identical to the code in Section 2.3 that created the scatterplot with overplotting, but with `alpha = 0.2` added to the `geom_point()`:

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.2)
```

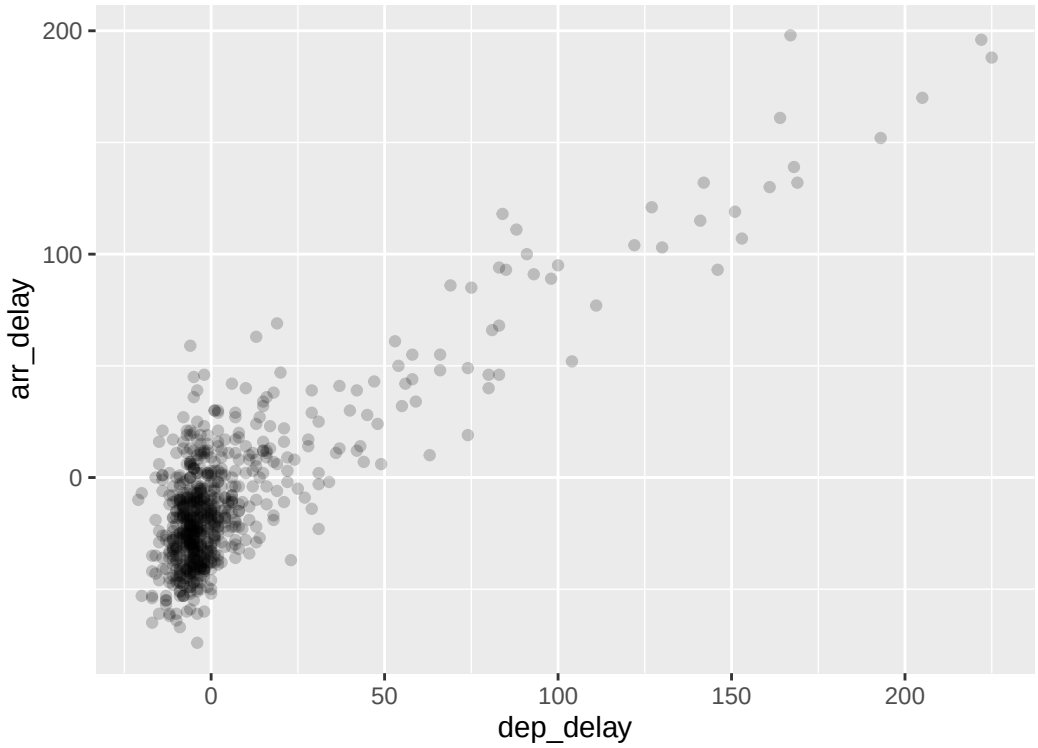


Figure 2.4: Delay scatterplot with alpha = 0.2

The key feature to note in Figure 2.4 is that the transparency of the points is cumulative: areas with a high-degree of overplotting are darker, whereas areas with a lower degree are less dark. Note furthermore that there is no `aes()` surrounding `alpha = 0.2`. This is because we are not mapping a variable to an aesthetic attribute, but rather merely changing the default setting of `alpha`. In fact, you'll receive an error if you try to change the second line above to read `geom_point(aes(alpha = 0.2))`.

Method 2: Jittering the points

The second way of addressing overplotting is by *jittering* all the points, in other words give each point a small nudge in a random direction. You can think of "jittering" as shaking the

points around a bit on the plot. Let's illustrate using a simple example first. Say we have a data frame `jitter_example` with 4 rows of identical value 0 for both `x` and `y`:

```
# A tibble: 4 x 2
      x     y
  <dbl> <dbl>
1     0     0
2     0     0
3     0     0
4     0     0
```

We display the resulting scatterplot in Figure 2.5; observe that the 4 points are superimposed on top of each other. While we know there are 4 values being plotted, this fact might not be apparent to others.

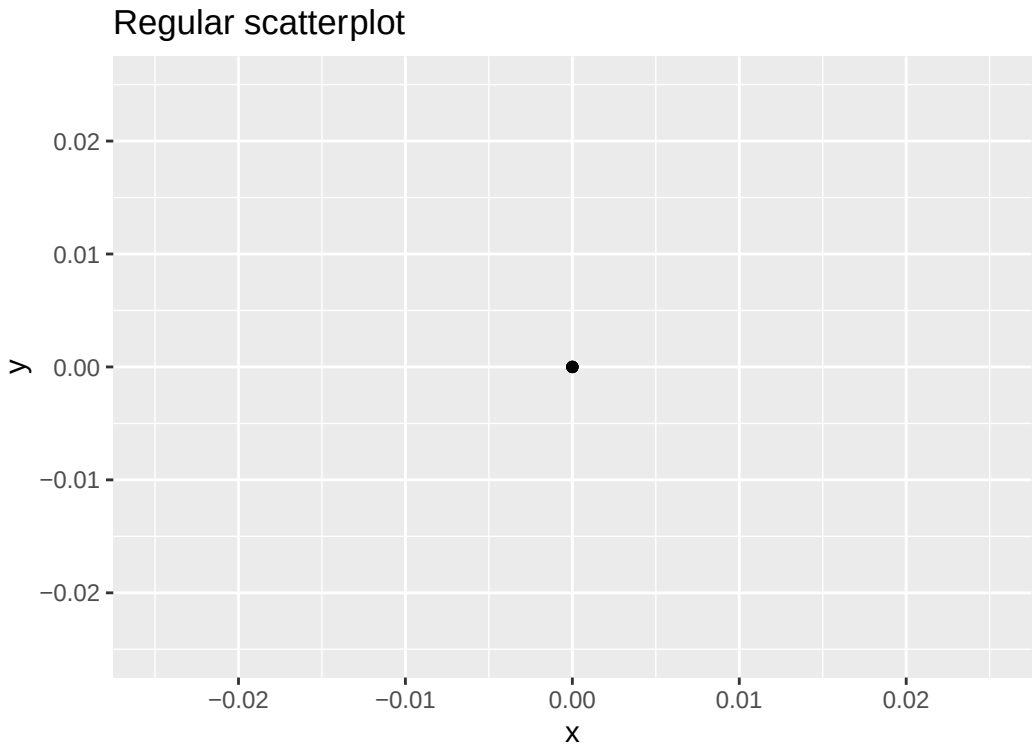


Figure 2.5: Regular scatterplot of jitter example data

In Figure 2.6 we instead display a *jittered scatterplot* where each point is given a random "nudge." It is now plainly evident that this plot involves four points. Keep in mind that jittering is strictly a visualization tool; even after creating a jittered scatterplot, the original values saved in `jitter_example` remain unchanged.

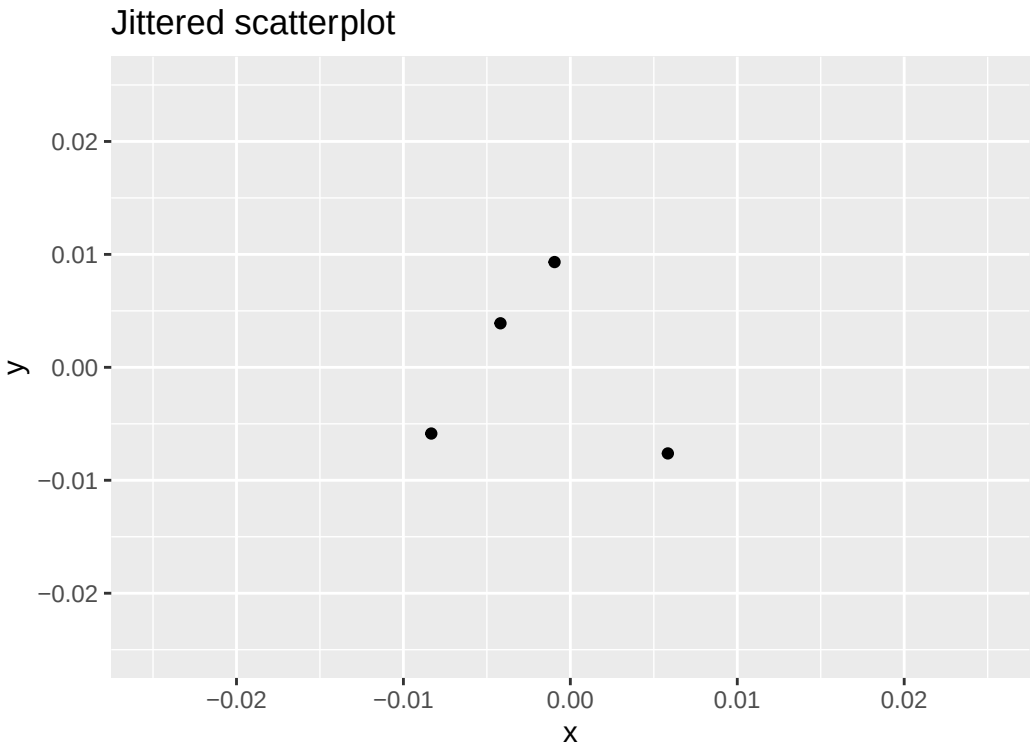


Figure 2.6: Jittered scatterplot of jitter example data

To create a jittered scatterplot, instead of using `geom_point()`, we use `geom_jitter()`. To specify how much jitter to add, we adjust the `width` and `height` arguments. This corresponds to how hard you'd like to shake the plot in units corresponding to those for both the horizontal and vertical variables (in this case minutes).

```
ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30)
```

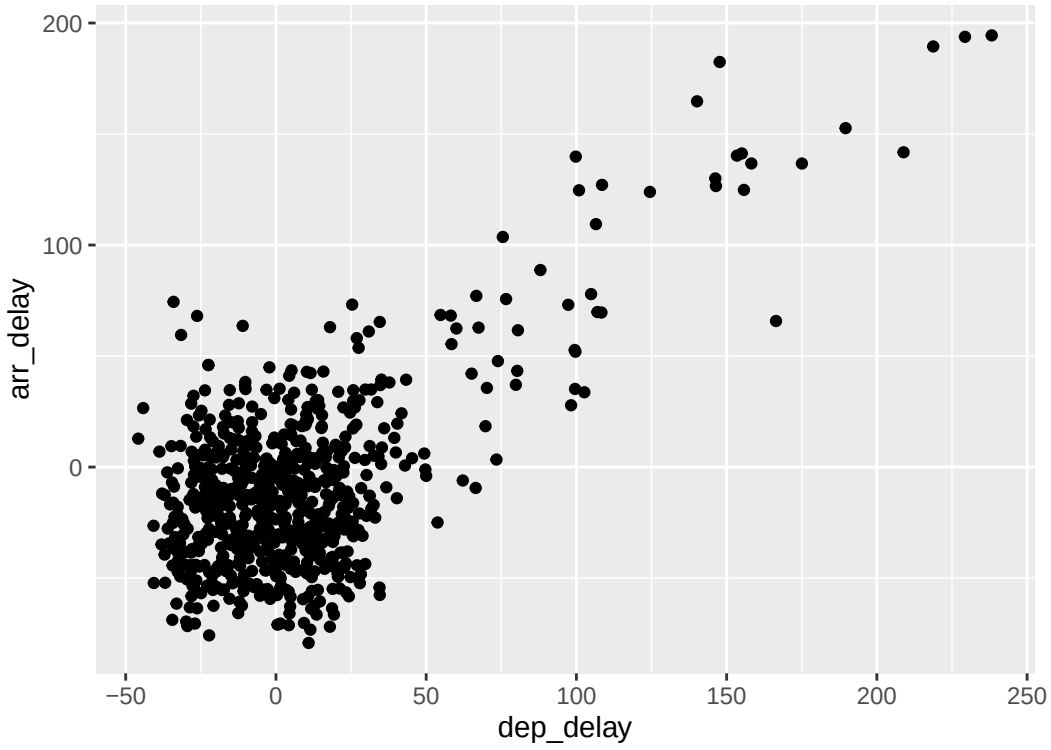


Figure 2.7: Jittered delay scatterplot

Observe how the above code is identical to the code that created the scatterplot with overplotting in Subsection 2.3.1, but with `geom_point()` replaced with `geom_jitter()`.

The resulting plot in Figure 2.7 helps us a little bit in getting a sense for the overplotting, but with a relatively large data set like this one (714 flights), it can be argued that changing the transparency of the points by setting `alpha` proved more effective. In terms of how much jitter one should add using the `width` and `height` arguments, it is important to add just enough jitter to break any overlap in points, but not so much that we completely alter the overall pattern in points.

Learning Check 2.7

Why is setting the `alpha` argument value useful with scatterplots? What further information does it give you that a regular scatterplot cannot?

Learning Check 2.8

After viewing the Figure 2.4 above, give an approximate range of arrival delays and departure delays that occur the most frequently. How has that region changed compared to when you observed the same plot without the `alpha = 0.2` set in Figure 2.2?

2.3.3 Summary

Scatterplots display the relationship between two numerical variables. They are among the most commonly used plots because they can provide an immediate way to see the trend in one variable versus another. However, if you try to create a scatterplot where either one of the two variables is not numerical, you might get strange results. Be careful!

With medium to large data sets, you may need to play around with the different modifications one can make to a scatterplot. This tweaking is often a fun part of data visualization, since you'll have the chance to see different relationships come about as you make subtle changes to your plots.

2.4 5NG#2: Linegraphs

The next of the five named graphs are linegraphs. Linegraphs show the relationship between two numerical variables when the variable on the x-axis, also called the *explanatory* variable, is of a sequential nature; in other words there is an inherent ordering to the variable. The most common example of linegraphs have some notion of time on the x-axis: hours, days, weeks, years, etc. Since time is sequential, we connect consecutive observations of the variable on the y-axis with a line. Linegraphs that have some notion of time on the x-axis are also called *time series* plots. Linegraphs should be avoided when there is not a clear sequential ordering to the variable on the x-axis. Let's illustrate linegraphs using another data set in the `nycflights13` package: the `weather` data frame.

Let's get a sense for the `weather` data frame:

- Explore the `weather` data by running `View(weather)`.
- Run `?weather` to bring up the help file.

We can see that there is a variable called `temp` of hourly temperature recordings in Fahrenheit at weather stations near all three airports in New York City: Newark (`origin` code `EWR`), JFK, and La Guardia (`LGA`). Instead of considering hourly temperatures for all days in 2013 for all three airports however, for simplicity let's only consider hourly temperatures at only Newark airport for the first 15 days in January.

Recall in Section 2.3 we used the `filter()` function to only choose the subset of rows of `flights` corresponding to Alaska Airlines flights. We similarly use `filter()` here, but by using the `&` operator we only choose the subset of rows of `weather` where

1. The `origin` is `"EWR"` and
2. the `month` is January and
3. the `day` is between `1` and `15`

```
early_january_weather <- weather %>%
  filter(origin == "EWR" & month == 1 & day <= 15)
```

Learning Check 2.9

Take a look at both the `weather` and `early_january_weather` data frames by running `View(weather)` and `View(early_january_weather)`. In what respect do these data frames differ?

Learning Check 2.10

`View()` the `flights` data frame again. Why does the `time_hour` variable uniquely identify the hour of the measurement whereas the `hour` variable does not?

2.4.1 Linegraphs via geom_line

Let's plot a linegraph of hourly temperatures in `early_january_weather` by using `geom_line()` instead of `geom_point()` like we did for scatterplots:

```
ggplot(data = early_january_weather, mapping = aes(x = time_hour, y = temp)) +
  geom_line()
```

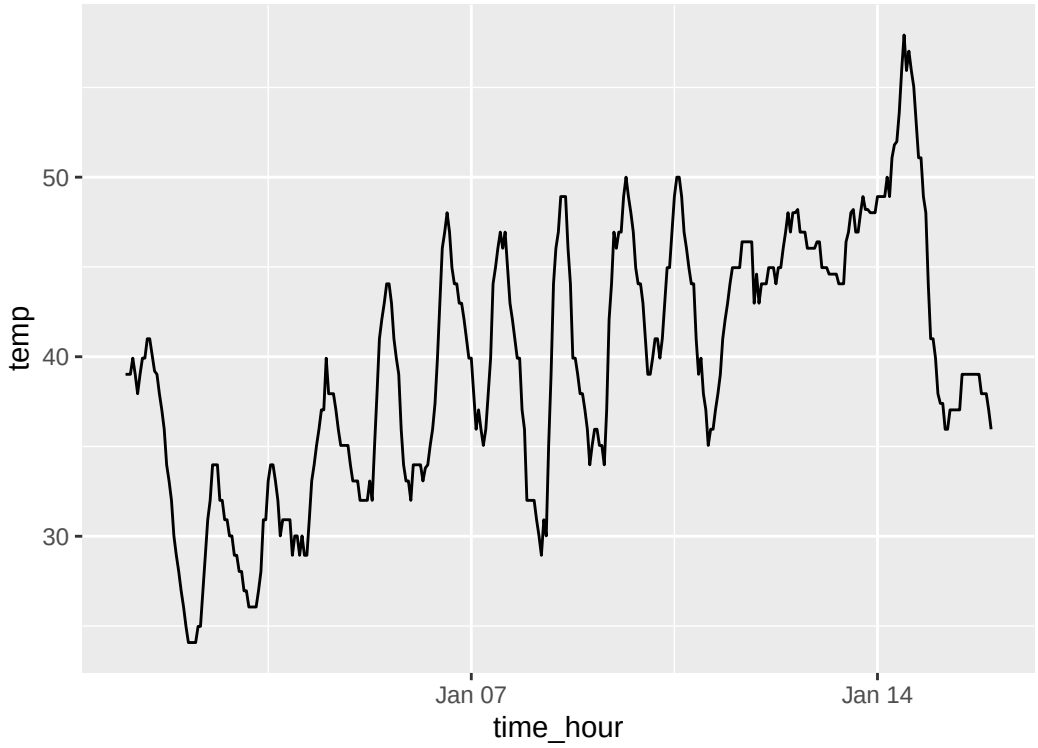


Figure 2.8: Hourly Temperature in Newark for January 1-15, 2013

Much as with the `ggplot()` code that created the scatterplot of departure and arrival delays for Alaska Airlines flights in Figure 2.2, let's break down the above code piece-by-piece in terms of the Grammar of Graphics:

- Within the `ggplot()` function call, we specify two of the components of the Grammar of Graphics as arguments:
    1. The `data` frame to be `early_january_weather` by setting `data = early_january_weather`
    2. The `aes`thetic mapping by setting `aes(x = time_hour, y = temp)`. Specifically:
        - the variable `time_hour` maps to the `x` position aesthetic.
        - the variable `temp` maps to the `y` position aesthetic
- We add a layer to the `ggplot()` function call using the `+` sign. The layer in question specifies the third component of the grammar: the `geom`etric object in question. In this case the geometric object is a `line`, set by specifying `geom_line()`.

Learning Check 2.11

Why should linegraphs be avoided when there is not a clear ordering of the horizontal axis?

Learning Check 2.12

Why are linegraphs frequently used when time is the explanatory variable on the x-axis?

Learning Check 2.13

Plot a time series of a variable other than `temp` for Newark Airport in the first 15 days of January 2013.

2.4.2 Summary

Linegraphs, just like scatterplots, display the relationship between two numerical variables. However it is preferred to use linegraphs over scatterplots when the variable on the x-axis (i.e. the explanatory variable) has an inherent ordering, like some notion of time.

2.5 5NG#3: Histograms

Let's consider the `temp` variable in the `weather` data frame once again, but unlike with the linegraphs in Section 2.4, let's say we don't care about the relationship of temperature to time, but rather we only care about how the values of `temp` *distribute*. In other words:

1. What are the smallest and largest values?
2. What is the "center" value?
3. How do the values spread out?
4. What are frequent and infrequent values?

One way to visualize this *distribution* of this single variable `temp` is to plot them on a horizontal line as we do in Figure 2.9:

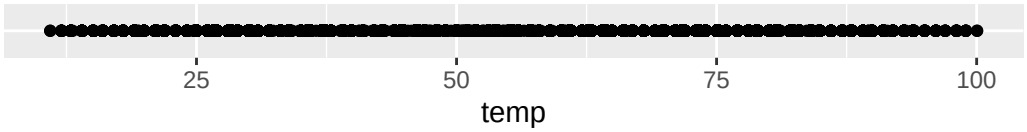


Figure 2.9: Plot of Hourly Temperature Recordings from NYC in 2013

This gives us a general idea of how the values of `temp` distribute: observe that temperatures vary from around 11°F up to 100°F. Furthermore, there appear to be more recorded temperatures between 40°F and 60°F than outside this range. However, because of the high degree of overlap in the points, it's hard to get a sense of exactly how many values are between, say, 50°F and 55°F.

What is commonly produced instead of the above plot is known as a *histogram*. A histogram is a plot that visualizes the *distribution* of a numerical value as follows:

1. We first cut up the x-axis into a series of *bins*, where each bin represents a range of values.
2. For each bin, we count the number of observations that fall in the range corresponding to that bin.
3. Then for each bin, we draw a bar whose height marks the corresponding count.

Let's drill-down on an example of a histogram, shown in @fig-histogramexample.

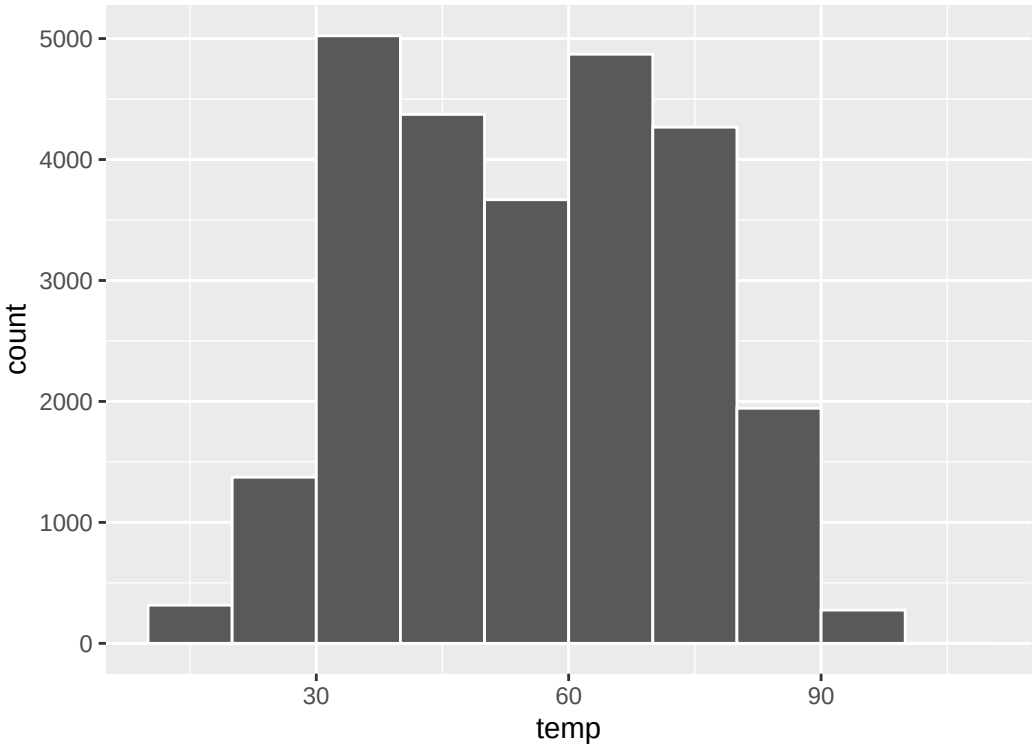


Figure 2.10: Example histogram

Observe that there are three bins of equal width between 30°F and 60°F, thus we have three bins of width 10°F each: one bin for the 30-40°F range, another bin for the 40-50°F range, and another bin for the 50-60°F range. Since:

1. The bin for the 30-40°F range has a height of around 5000, this histogram is telling us that around 5000 of the hourly temperature recordings are between 30°F and 40°F.
2. The bin for the 40-50°F range has a height of around 4300, this histogram is telling us that around 4300 of the hourly temperature recordings are between 40°F and 50°F.
3. The bin for the 50-60°F range has a height of around 3500, this histogram is telling us that around 3500 of the hourly temperature recordings are between 50°F and 60°F.

The remaining bins all have a similar interpretation.

2.5.1 Histograms via geom_histogram

Let's now present the `ggplot()` code to plot your first histogram! Unlike with scatterplots and linegraphs, there is now only one variable being mapped in `aes()`: the single numerical variable `temp`. The y-aesthetic of a histogram gets computed for you automatically. Furthermore, the geometric object layer is now a `geom_histogram()`

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram()
```

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

```
Warning: Removed 1 rows containing non-finite values (stat_bin).
```

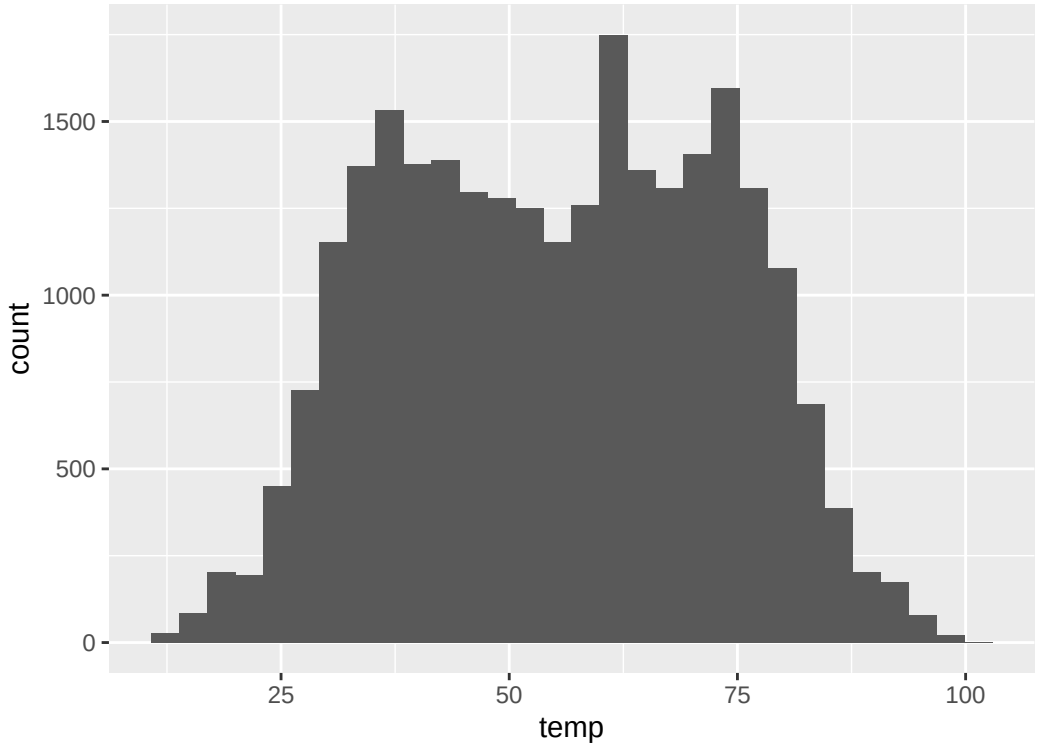


Figure 2.11: Histogram of hourly temperatures at three NYC airports

Let's unpack the messages R sent us first. The first message is telling us that the histogram was constructed using `bins = 30`, in other words 30 equally spaced bins. This is known in computer programming as a default value; unless you override this default number of bins with a number you specify, R will choose 30 by default. We'll see in the next section how to change this default number of bins. The second message is telling us something similar to the warning message we received when we ran the code to create a scatterplot of departure and arrival delays for Alaska Airlines flights in Figure 2.2: that because one row has a missing `NA` value for `temp`, it was omitted from the histogram. R is just giving us a friendly heads up that this was the case.

Now's let's unpack the resulting histogram in Figure 2.11. Observe that values less than 25°F as well as values above 80°F are rather rare. However, because of the large number of bins, its hard to get a sense for which range of temperatures is covered by each bin; everything is one giant amorphous blob. So let's add white vertical borders demarcating the bins by adding a `color = "white"` argument to `geom_histogram()`:

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(color = "white")
```

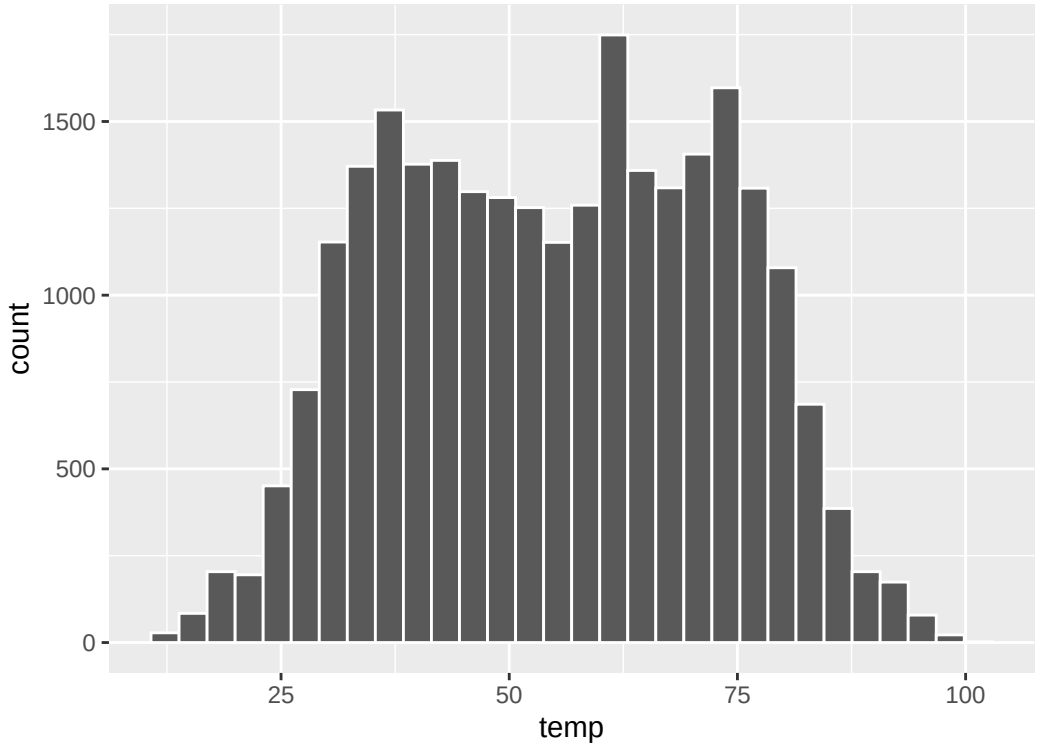


Figure 2.12: Histogram of hourly temperatures at three NYC airports with white borders

We can now better associate ranges of temperatures to each of the bins. We can also vary the color of the bars by setting the `fill` argument. Run `colors()` to see all 657 possible choice of colors!

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(color = "white", fill = "steelblue")
```

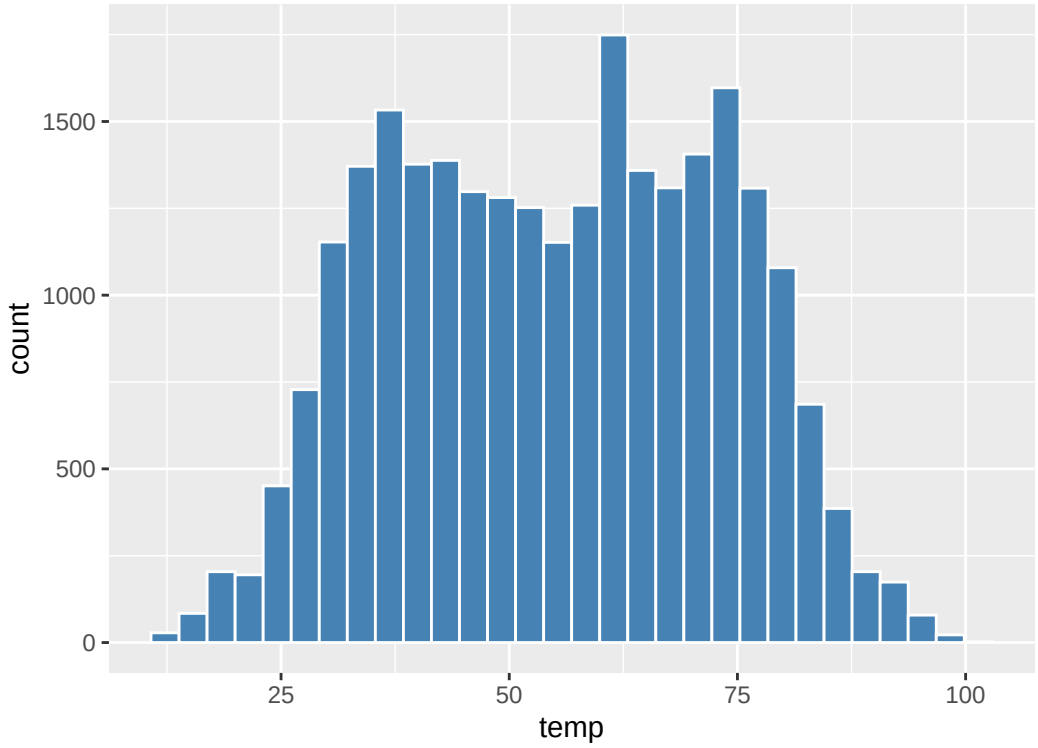


Figure 2.13: Histogram of hourly temperatures at three NYC airports with white borders

2.5.2 Adjusting the bins

Observe in both Figure 2.12 and Figure 2.13 that in the 50-75°F range there appear to be roughly 8 bins. Thus each bin has width 25 divided by 8, or roughly 3.12°F which is not a very easily interpretable range to work with. Let's now adjust the number of bins in our histogram in one of two methods:

1. By adjusting the number of bins via the `bins` argument to `geom_histogram()`.
2. By adjusting the width of the bins via the `binwidth` argument to `geom_histogram()`.

Using the first method, we have the power to specify how many bins we would like to cut the x-axis up in. As mentioned in the previous section, the default number of bins is 30. We can override this default, to say 40 bins, as follows:

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(bins = 40, color = "white")
```

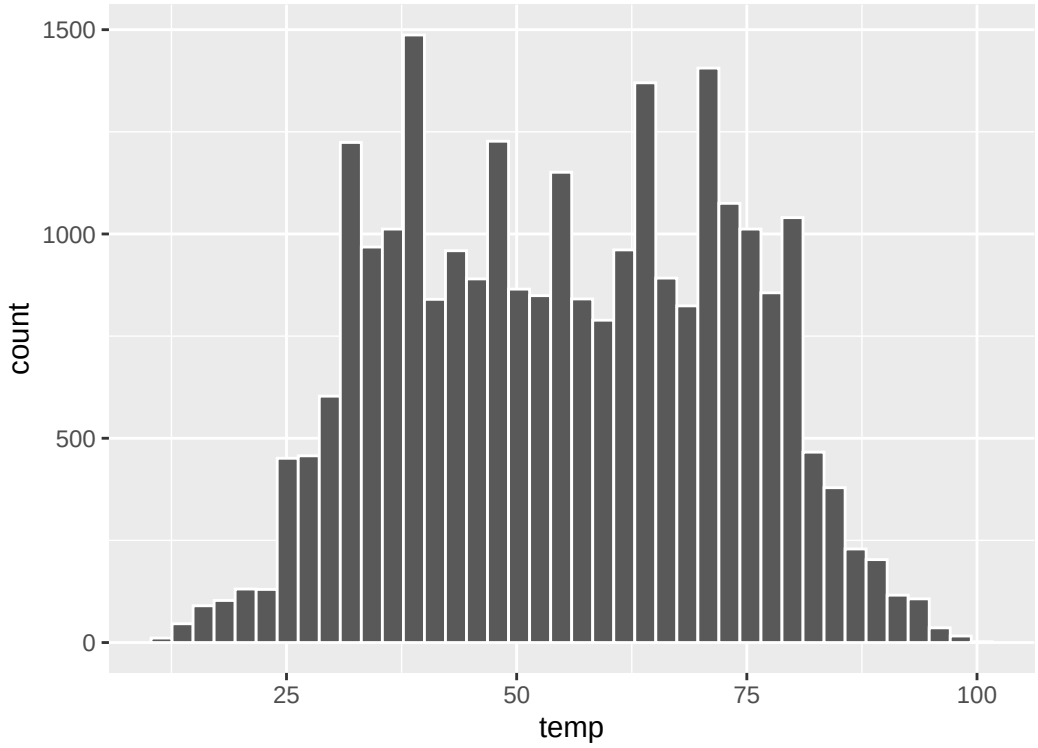


Figure 2.14: Histogram with 40 bins

Using the second method, instead of specifying the number of bins, we specify the width of the bins by using the `binwidth` argument in the `geom_histogram()` layer. For example, let's set the width of each bin to be 10°F.

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 10, color = "white")
```

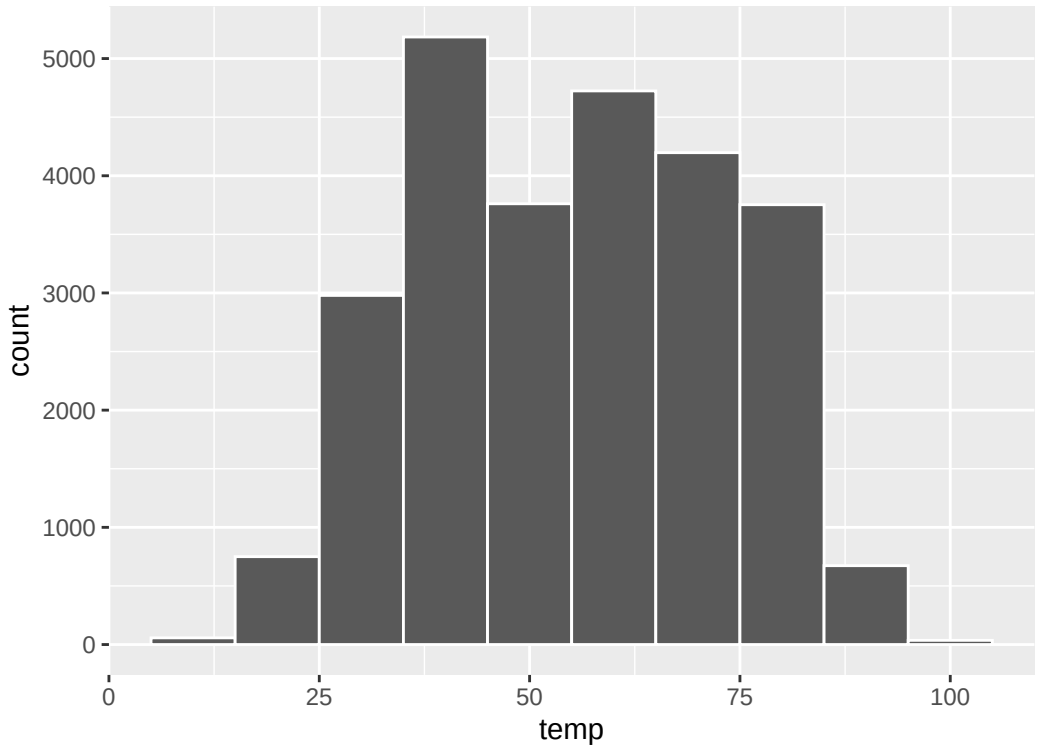


Figure 2.15: Histogram with binwidth 10

Learning Check 2.14

What does changing the number of bins from 30 to 40 tell us about the distribution of temperatures?

Learning Check 2.15

Would you classify the distribution of temperatures as symmetric or skewed?

Learning Check 2.16

What would you guess is the "center" value in this distribution? Why did you make that choice?

Learning Check 2.17

Is this data spread out greatly from the center or is it close? Why?

2.5.3 Summary

Histograms, unlike scatterplots and linegraphs, present information on only a single numerical variable. Specifically, they are visualizations of the distribution of the numerical variable in question.

2.6 Facets

Before continuing the 5NG, let's briefly introduce a new concept called *faceting*. Faceting is used when we'd like to split a particular visualization of variables by another variable. This will create multiple copies of the same type of plot with matching x and y axes, but whose content will differ.

For example, suppose we were interested in looking at how the histogram of hourly temperature recordings at the three NYC airports we saw in Section 2.5 differed by month. We would "split" this histogram by the 12 possible months in a given year, in other words plot histograms of `temp` for each `month`. We do this by adding `facet_wrap(~ month)` layer.

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 5, color = "white") +
  facet_wrap(~ month)
```

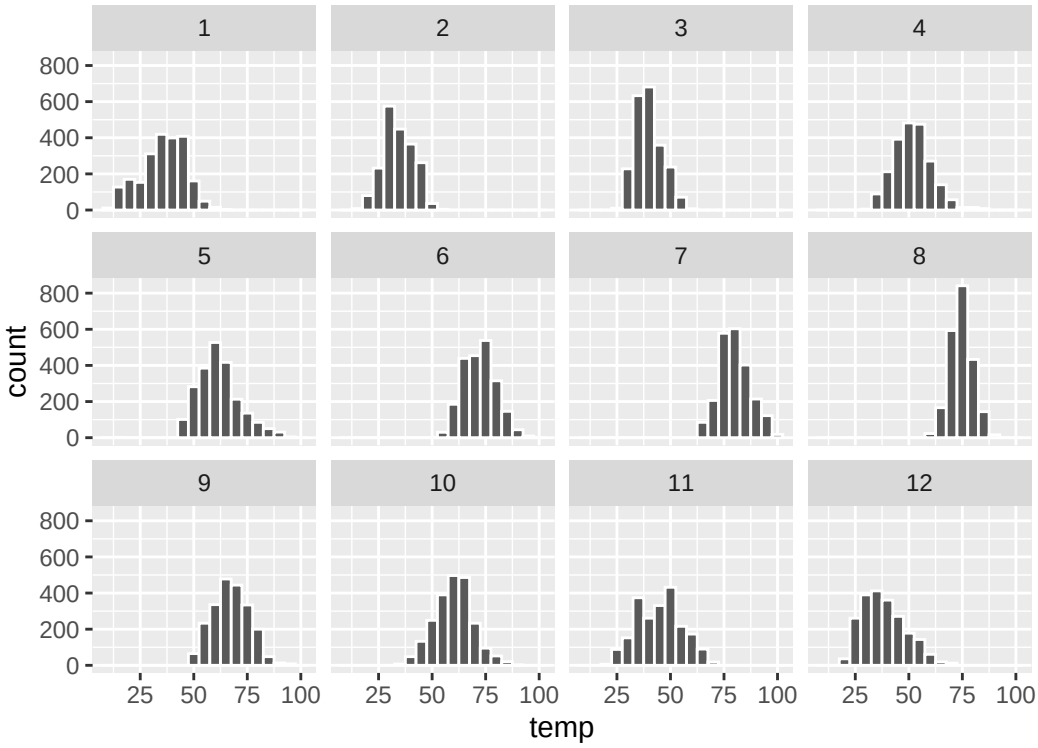


Figure 2.16: Faceted histogram

Note the use of the tilde `~` before `month` in `facet_wrap()`. The tilde is required and you'll receive the error `Error in as.quoted(facets) : object 'month' not found` if you don't include it before `month` here. We can also specify the number of rows and columns in the grid by using the `nrow` and `ncol` arguments inside of `facet_wrap()`. For example, say we would like our faceted plot to have 4 rows instead of 3. Add the `nrow = 4` argument to `facet_wrap(~ month)`

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 5, color = "white") +
  facet_wrap(~ month, nrow = 4)
```

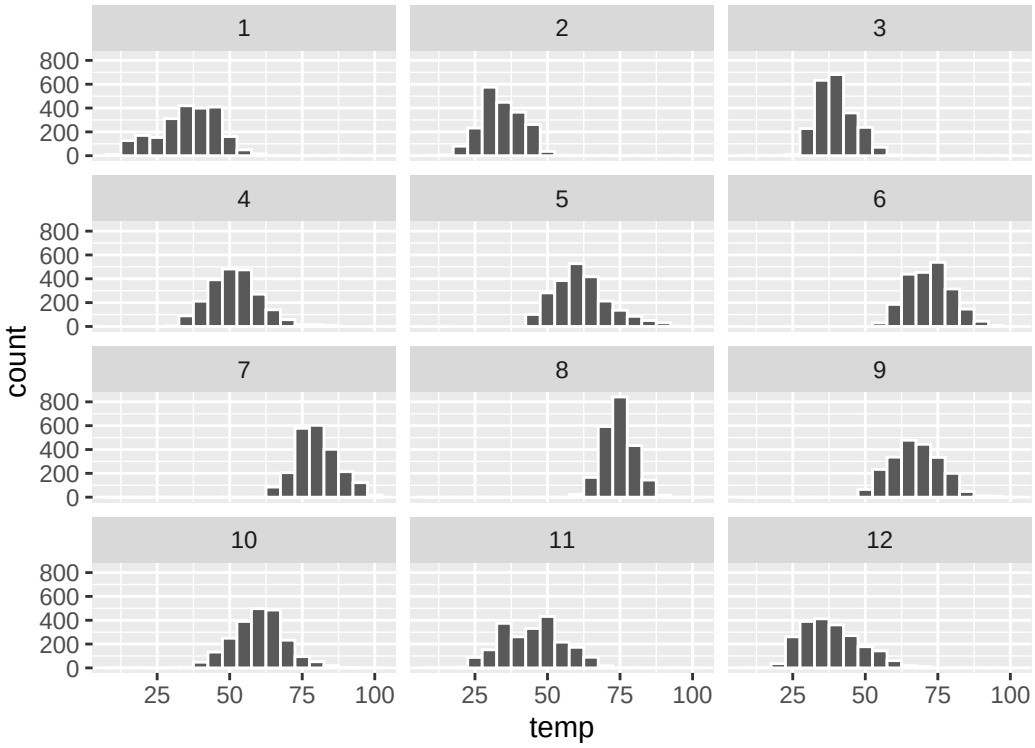


Figure 2.17: Faceted histogram with 4 instead of 3 rows

Observe in both Figure 2.16 and Figure 2.17 that as we might expect in the Northern Hemisphere, temperatures tend to be higher in the summer months, while they tend to be lower in the winter.

Learning Check 2.18

What other things do you notice about the faceted plot above? How does a faceted plot help us see relationships between two variables?

Learning Check 2.19

What do the numbers 1-12 correspond to in the plot above? What about 25, 50, 75, 100?

Learning Check 2.20

For which types of data sets would these types of faceted plots not work well in comparing relationships between variables? Give an example describing the nature of these variables

and other important characteristics.

Learning Check 2.21

Does the `temp` variable in the `weather` data set have a lot of variability? Why do you say that?

2.7 5NG#4: Boxplots

While faceted histograms are one visualization that allows us to compare distributions of a numerical variable split by another variable, another visualization that achieves this same goal are *side-by-side boxplots*. A boxplot is constructed from the information provided in the *five-number summary* of a numerical variable (see Appendix A). To keep things simple for now, let's only consider hourly temperature recordings for the month of November in Figure 2.18.

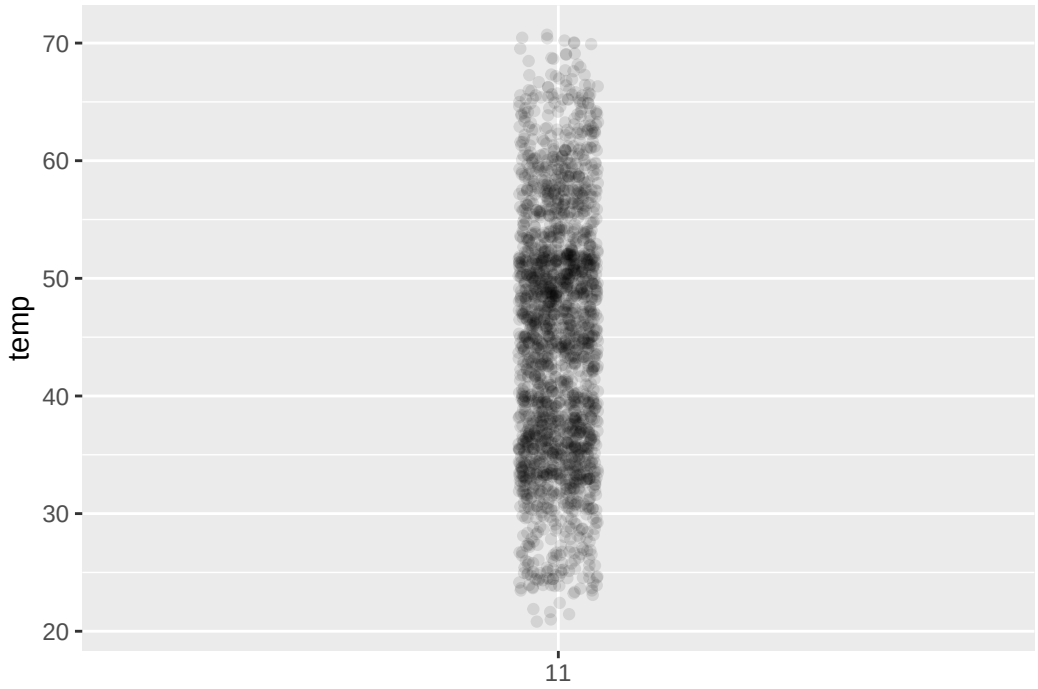


Figure 2.18: November temperatures

These 2141 observations have the following five-number summary:

1. Minimum: 21.02°F

2. First quartile AKA 25th percentile: 35.96°F
3. Median AKA second quartile AKA 50th percentile: 44.96°F
4. Third quartile AKA 75th percentile: 51.98°F
5. Maximum: 71.06°F

Let's mark these 5 values with dashed horizontal lines in Figure 2.19.

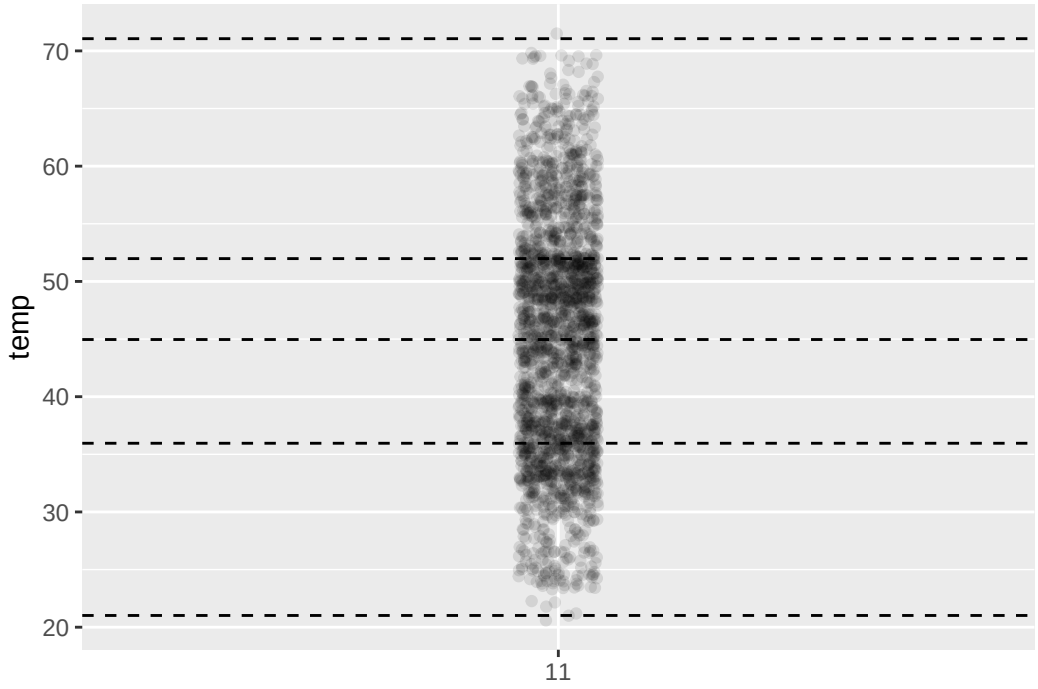


Figure 2.19: November temperatures

Let's add the boxplot underneath these points and dashed horizontal lines in Figure 2.20.

What the boxplot does summarize the 2141 points by emphasizing that:

1. 25% of points (about 534 observations) fall below the bottom edge of the box, which is the first quartile of 35.96°F. In other words 25% of observations were colder than 35.96°F.
2. 25% of points fall between the bottom edge of the box and the solid middle line, which is the median of 44.96°F. In other words 25% of observations were between 35.96 and 44.96°F and 50% of observations were colder than 44.96°F.
3. 25% of points fall between the solid middle line and the top edge of the box, which is the third quartile of 51.98°F. In other words 25% of observations were between 44.96 and 51.98°F and 75% of observations were colder than 51.98°F.

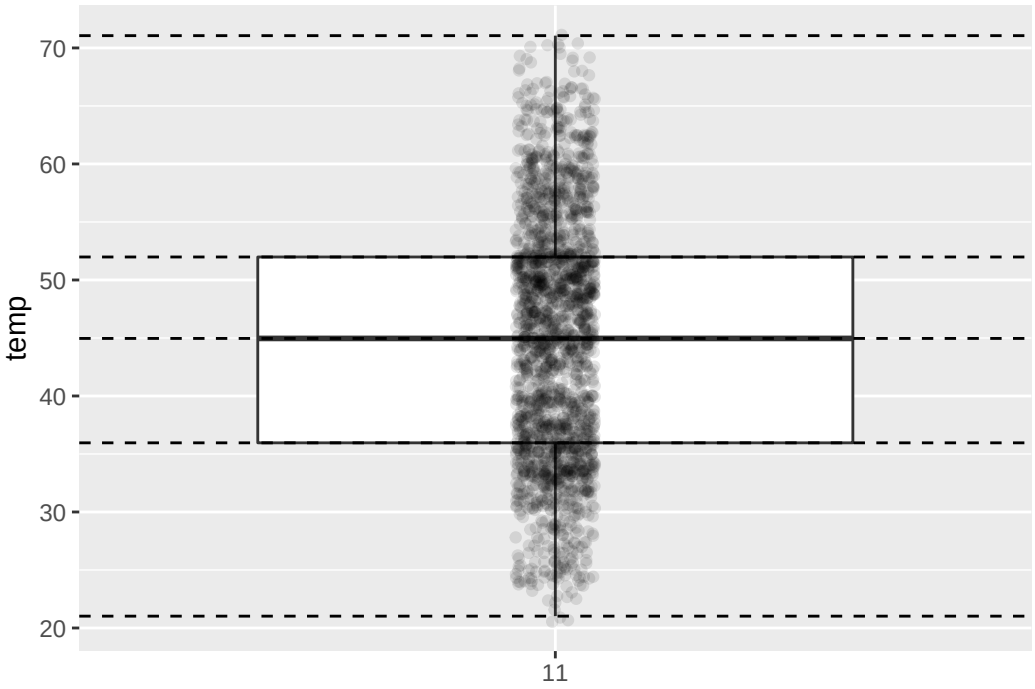


Figure 2.20: November temperatures

4. 25% of points fall over the top edge of the box. In other words 25% of observations were warmer than 51.98°F.
5. The middle 50% of points lie within the *interquartile range* between the first and third quartile of 51.98 - 35.96 = 16.02°F.

Lastly, for clarity's sake let's remove the points but keep the dashed horizontal lines in Figure 2.21.

We can now better see the *whiskers* of the boxplot. They stick out from either end of the box all the way to the minimum and maximum observed temperatures of 21.02°F and 71.06°F respectively. However, the whiskers don't always extend to the smallest and largest observed values. They in fact can extend no more than 1.5 × the interquartile range from either end of the box, in this case 1.5 × 16.02°F = 24.03°F from either end of the box. Any observed values outside this whiskers get marked with points called *outliers*, which we'll see in the next section.

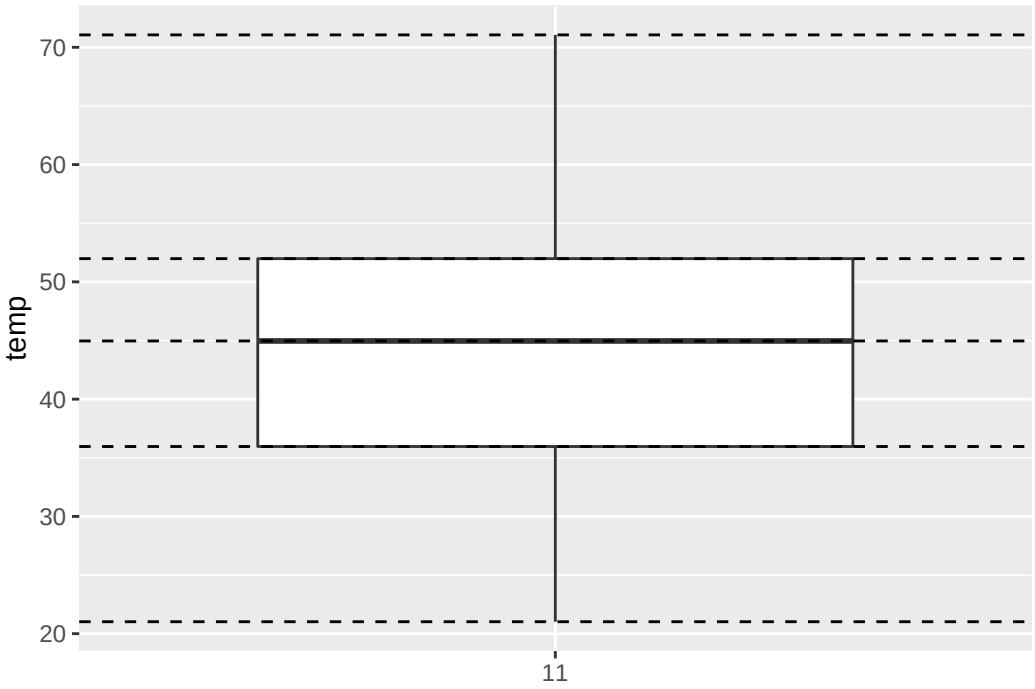


Figure 2.21: November temperatures

2.7.1 Boxplots via geom_boxplot

Let's now create a side-by-side boxplot of hourly temperatures split by the 12 months as we did above with the faceted histograms. We do this by mapping the `month` variable to the x-position aesthetic, the `temp` variable to the y-position aesthetic, and by adding a `geom_boxplot()` layer:

```
ggplot(data = weather, mapping = aes(x = month, y = temp)) +
  geom_boxplot()
```

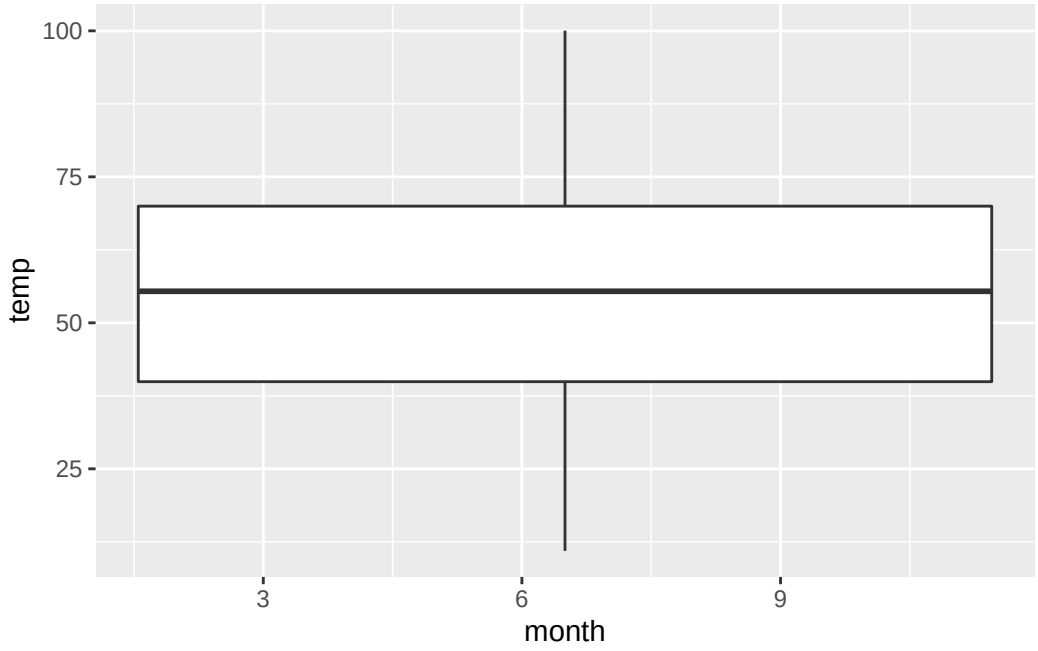


Figure 2.22: Invalid boxplot specification

```
Warning messages:
1: Continuous x aesthetic -- did you forget aes(group=...)?
2: Removed 1 rows containing non-finite values (stat_boxplot).
```

Observe in Figure 2.22 that this plot does not provide information about temperature separated by month. The warning messages clue us in as to why. The second warning message is identical to the warning message when plotting a histogram of hourly temperatures: that one of the values was recorded as `NA` missing. However, the first warning message is telling us that we have a "continuous", or numerical variable, on the x-position aesthetic. Boxplots however require a categorical variable on the x-axis.

We can convert the numerical variable `month` into a categorical variable by using the `factor()` function. So after applying `factor(month)`, month goes from having numerical values 1, 2, ..., 12 to having labels "1", "2", ..., "12."

```
ggplot(data = weather, mapping = aes(x = factor(month), y = temp)) +
  geom_boxplot()
```

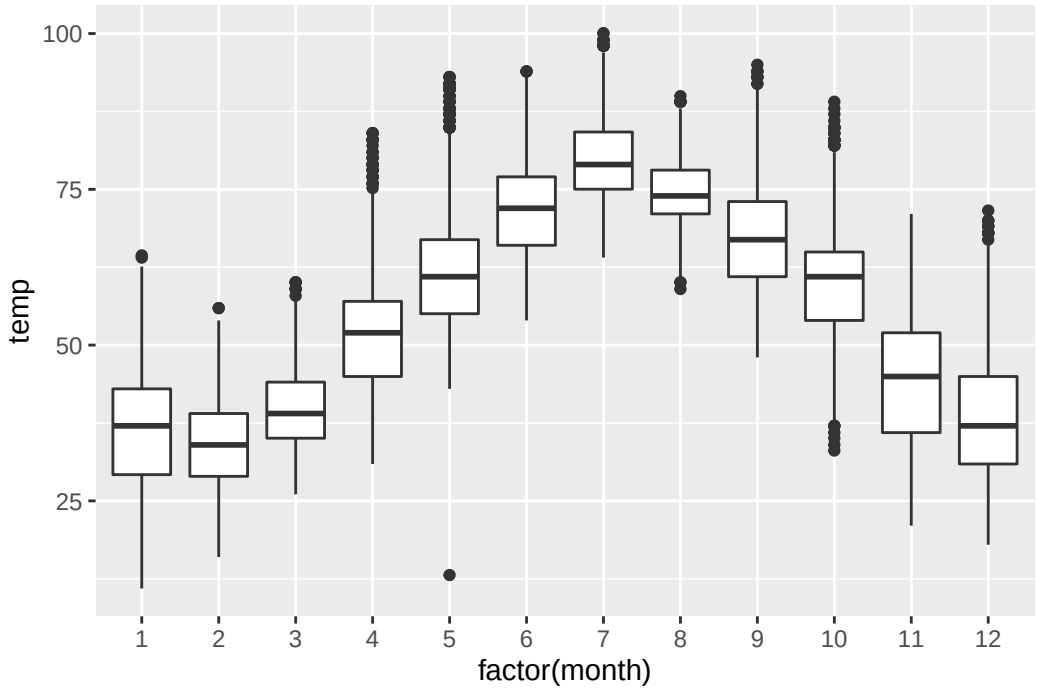


Figure 2.23: Temp by month boxplot

The resulting Figure 2.23 shows 12 separate "box and whiskers" plots with the features we saw earlier focusing only on November:

- The "box" portions of this visualization represent the 1st quartile, the median AKA the 2nd quartile, and the 3rd quartile.
- The "length" of each box, i.e. the value of the 3rd quartile minus the value of the 1st quartile, is the *interquartile range*. It is a measure of spread of the middle 50% of values, with longer boxes indicating more variability.
- The "whisker" portions of these plots extend out from the bottoms and tops of the boxes and represent points less than the 25th percentile and greater than the 75th percentiles respectively. They're set to extend out no more than $1.5 \times IQR$ units away from either end of the boxes. We say "no more than" because the ends of the whiskers have to correspond to observed temperatures. The length of these whiskers show how the data outside the middle 50% of values vary, with longer whiskers indicating more variability.
- The dots representing values falling outside the whiskers are called *outliers*. These can be thought of as anomalous values.

It is important to keep in mind that the definition of an outlier is somewhat arbitrary and not absolute. In this case, they are defined by the length of the whiskers, which are no

more than $1.5 \times IQR$ units long. Looking at this plot we can see, as expected, that summer months (6 through 8) have higher median temperatures as evidenced by the higher solid lines in the middle of the boxes. We can easily compare temperatures across months by drawing imaginary horizontal lines across the plot. Furthermore, the height of the 12 boxes as quantified by the interquartile ranges are informative too; they tell us about variability, or spread, of temperatures recorded in a given month.

Learning Check 2.22

What does the dot at the bottom of the plot for May correspond to? Explain what might have occurred in May to produce this point.

Learning Check 2.23

Which months have the highest variability in temperature? What reasons can you give for this?

Learning Check 2.24

We looked at the distribution of the numerical variable `temp` split by the numerical variable `month` that we converted to a categorical variable using the `factor()` function. Why would a boxplot of `temp` split by the numerical variable `pressure` similarly converted to a categorical variable using the `factor()` not be informative?

Learning Check 2.25

Boxplots provide a simple way to identify outliers. Why may outliers be easier to identify when looking at a boxplot instead of a faceted histogram?

2.7.2 Summary

Side-by-side boxplots provide us with a way to compare and contrast the distribution of a quantitative variable across multiple levels of another categorical variable. One can see where the median falls across the different groups by looking at the center line in the boxes. To see how spread out the variable is across the different groups, look at both the width of the box and also how far the whiskers stretch out away from the box. Outliers are even more easily identified when looking at a boxplot than when looking at a histogram as they are marked with points.

2.8 5NG#5: Barplots

Both histograms and boxplots are tools to visualize the distribution of numerical variables. Another common task is visualize the distribution of a categorical variable. This is a simpler task, as we are simply counting different categories, also known as *levels*, of a categorical variable. Often the best way to visualize these different counts, also known as *frequencies*, is with a barplot (also known as a barchart). One complication, however, is how your data is represented: is the categorical variable of interest "pre-counted" or not? For example, run the following code that manually creates two data frames representing a collection of fruit: 3 apples and 2 oranges.

```
fruits <- tibble(
  fruit = c("apple", "apple", "orange", "apple", "orange")
  )

fruits_counted <- tibble(
  fruit = c("apple", "orange"),
  number = c(3, 2)
  )
```

We see both the `fruits` and `fruits_counted` data frames represent the same collection of fruit. Whereas `fruits` just lists the fruit individually…

```
# A tibble: 5 x 1
  fruit
  <chr>
1 apple
2 apple
3 orange
4 apple
5 orange
```

… `fruits_counted` has a variable `number` which represents pre-counted values of each fruit.

```
# A tibble: 2 x 2
  fruit  number
  <chr>   <dbl>
1 apple       3
2 orange      2
```

Depending on how your categorical data is represented, you'll need to use add a different `geom` layer to your `ggplot()` to create a barplot, as we now explore.

2.8.1 Barplots via geom_bar or geom_col

Let's generate barplots using these two different representations of the same basket of fruit: 3 apples and 2 oranges. Using the `fruits` data frame where all 5 fruits are listed individually in 5 rows, we map the `fruit` variable to the x-position aesthetic and add a `geom_bar()` layer.

```
ggplot(data = fruits, mapping = aes(x = fruit)) +
  geom_bar()
```

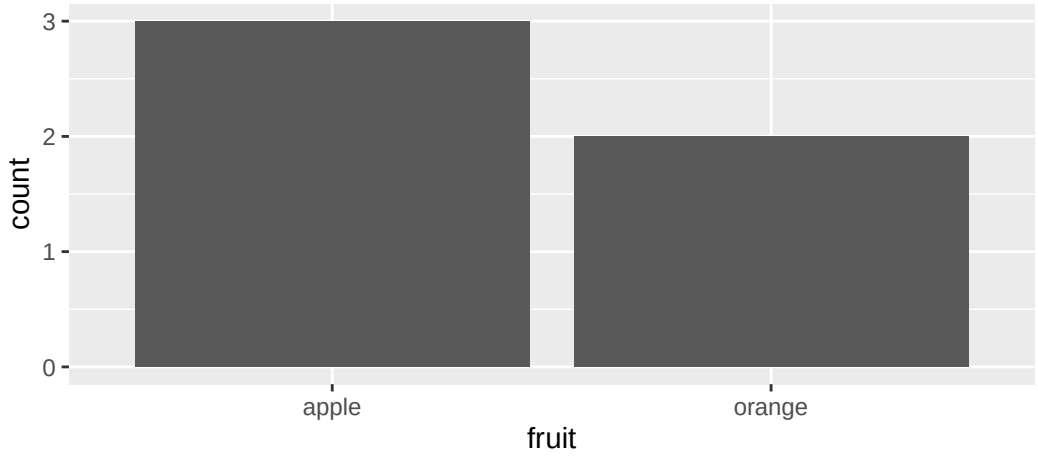


Figure 2.24: Barplot when counts are not pre-counted

However, using the `fruits_counted` data frame where the fruit have been "pre-counted", we map the `fruit` variable to the x-position aesthetic as with `geom_bar()`, but we also map the `count` variable to the y-position aesthetic, and add a `geom_col()` layer.

```
ggplot(data = fruits_counted, mapping = aes(x = fruit, y = number)) +
  geom_col()
```

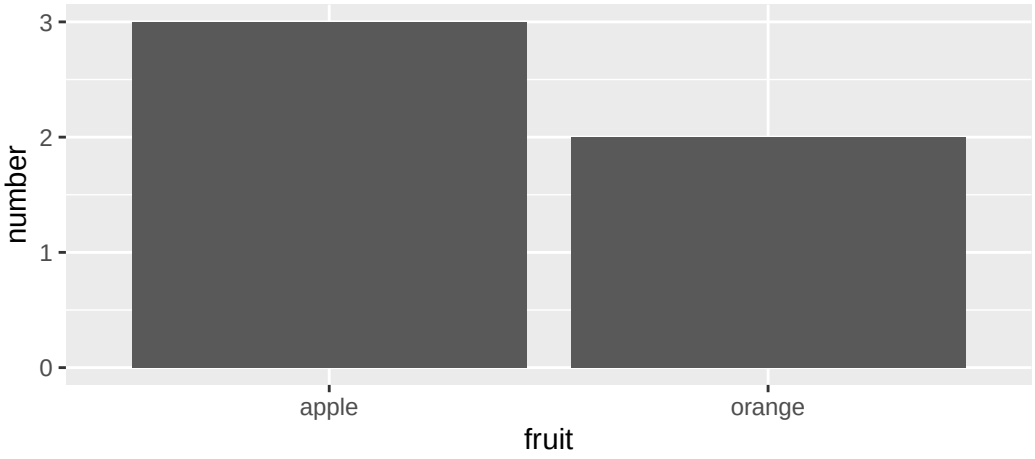


Figure 2.25: Barplot when counts are pre-counted

Compare the barplots in Figure 2.24 and Figure 2.25. They are identical because they reflect count of the same 5 fruit. However depending on how our data is saved, either pre-counted or not, we must add a different `geom` layer. When the categorical variable whose distribution you want to visualize is:

- Is not pre-counted in your data frame: use `geom_bar()`.
- Is pre-counted in your data frame, use `geom_col()` with the y-position aesthetic mapped to the variable that has the counts.

Let's now go back to the `flights` data frame in the `nycflights13` package and visualize the distribution of the categorical variable `carrier`. In other words, let's visualize the number of domestic flights out of the three New York City airports each airline company flew in 2013. Recall from Section 1.4.3 when you first explored the `flights` data frame you saw that each row corresponds to a flight. In other words the `flights` data frame is more like the `fruits` data frame than the `fruits_counted` data frame above, and thus we should use `geom_bar()` instead of `geom_col()` to create a barplot. Much like a `geom_histogram()`, there is only one variable in the `aes()` aesthetic mapping: the variable `carrier` gets mapped to the `x`-position.

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

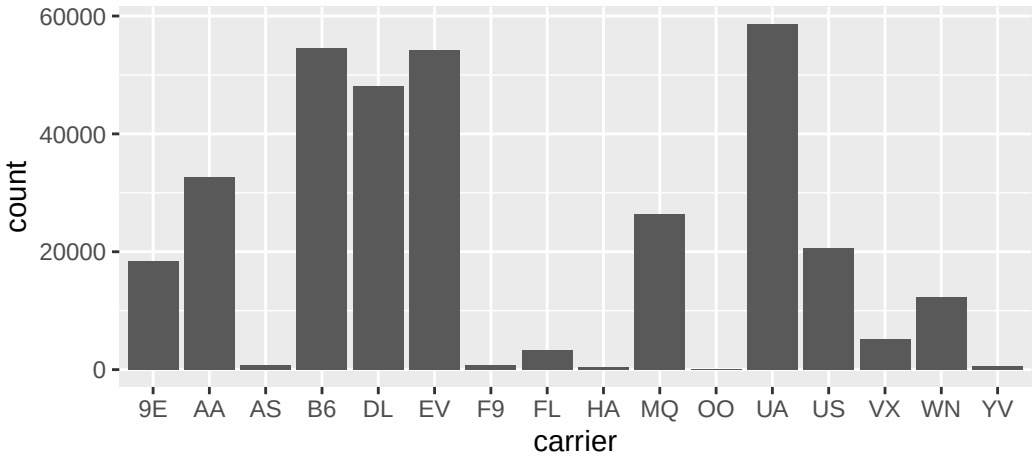


Figure 2.26: Number of flights departing NYC in 2013 by airline using `geom_bar()`

Observe in Figure 2.26 that United Air Lines (`UA`), JetBlue Airways (`B6`), and ExpressJet Airlines (`EV`) had the most flights depart New York City in 2013. If you don't know which airlines correspond to which carrier codes, then run `View(airlines)` to see a directory of airlines. For example: AA is American Airlines; B6 is JetBlue Airways; DL is Delta Airlines; EV is ExpressJet Airlines; MQ is Envoy Air; while UA is United Airlines.

Alternatively, say you had a data frame `flights_counted` where the number of flights for each `carrier` was pre-counted like in Table 2.3.

Table 2.3: Number of flights pre-counted for each carrier

carrier	number
UA	58665
B6	54635
EV	54173
DL	48110
AA	32729
MQ	26397
US	20536
9E	18460
WN	12275
VX	5162
FL	3260
AS	714
F9	685
YV	601

carrier	number
HA	342
OO	32

In order to create a barplot visualizing the distribution of the categorical variable `carrier` in this case, we would use `geom_col()` instead with `x` mapped to `carrier` and `y` mapped to `number` as seen below. The resulting barplot would be identical to Figure 2.26.

```
ggplot(data = flights_table, mapping = aes(x = carrier, y = number)) +
  geom_col()
```

Learning Check 2.26

Why are histograms inappropriate for visualizing categorical variables?

Learning Check 2.27

What is the difference between histograms and barplots?

Learning Check 2.28

How many Envoy Air flights departed NYC in 2013?

Learning Check 2.29

What was the seventh highest airline in terms of departed flights from NYC in 2013? How could we better present the table to get this answer quickly?

2.8.2 Must avoid pie charts!

Unfortunately, one of the most common plots seen today for categorical data is the pie chart. While they may seem harmless enough, they actually present a problem in that humans are unable to judge angles well. As Naomi Robbins describes in her book "Creating More Effective Graphs" (Robbins 2013), we overestimate angles greater than 90 degrees and we underestimate angles less than 90 degrees. In other words, it is difficult for us to determine relative size of one piece of the pie compared to another.

Let's examine the same data used in our previous barplot of the number of flights departing NYC by airline in Figure 2.26, but this time we will use a pie chart in Figure 2.27.

Try to answer the following questions:

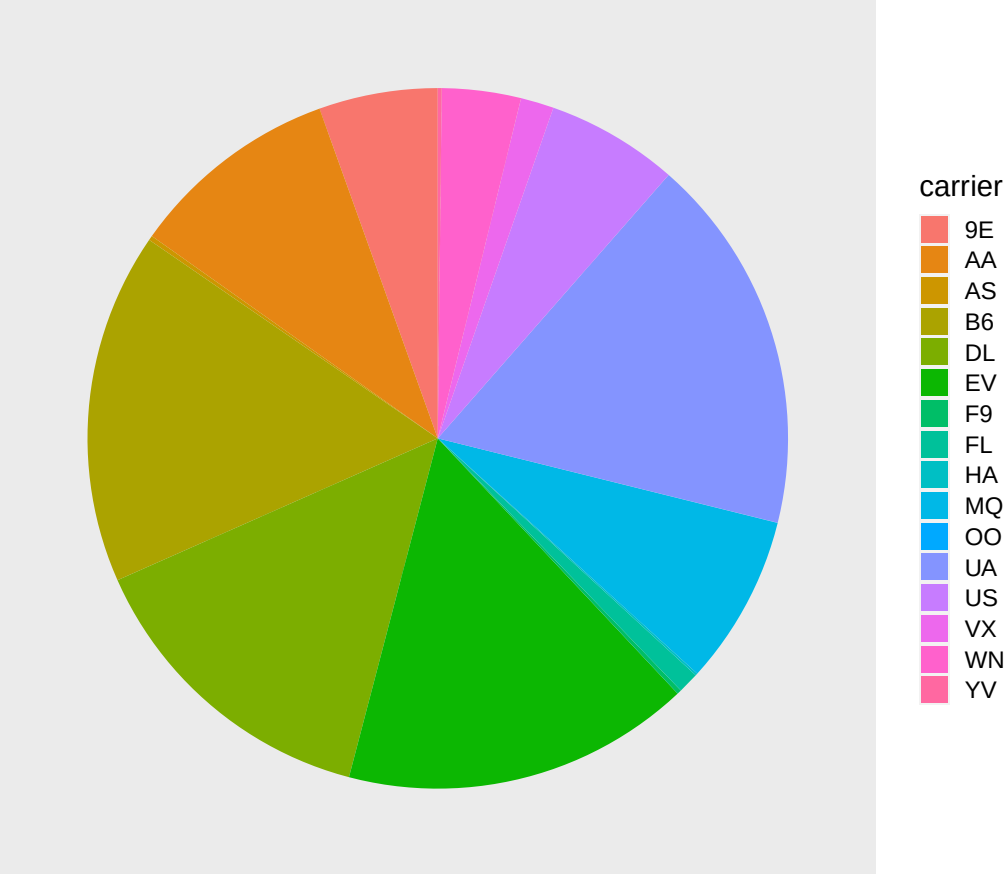


Figure 2.27: The dreaded pie chart

- How much larger the portion of the pie is for ExpressJet Airlines (`EV`) compared to US Airways (`US`),
- What the third largest carrier is in terms of departing flights, and
- How many carriers have fewer flights than United Airlines (`UA`)?

While it is quite difficult to answer these questions when looking at the pie chart in Figure 2.27, we can much more easily answer these questions using the barchart in Figure Figure 2.26. This is true since barplots present the information in a way such that comparisons between categories can be made with single horizontal lines, whereas pie charts present the information in a way such that comparisons between categories must be made by comparing angles.

There may be one exception of a pie chart not to avoid courtesy Nathan Yau at FlowingData.com, but we will leave this for the reader to decide:

Figure 2.28: The only good pie chart

Learning Check 2.30

Why should pie charts be avoided and replaced by barplots?

Learning Check 2.31

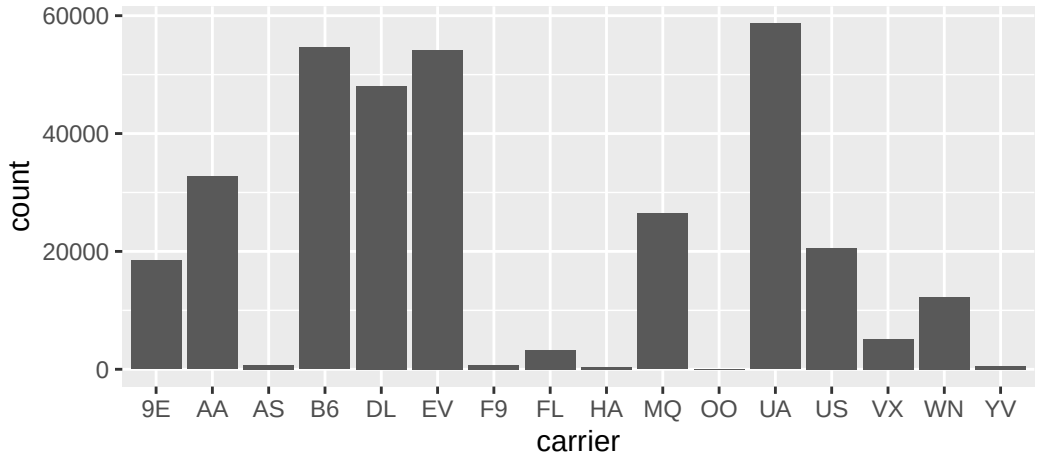
Why do you think people continue to use pie charts?

2.8.3 Two categorical variables

Barplots are the go-to way to visualize the frequency of different categories, or levels, of a single categorical variable. Another use of barplots is to visualize the *joint* distribution of

two categorical variables at the same time. Let's examine the *joint* distribution of outgoing domestic flights from NYC by `carrier` and `origin`, or in other words the number of flights for each `carrier` and `origin` combination. For example, the number of WestJet flights from `JFK`, the number of WestJet flights from `LGA`, the number of WestJet flights from `EWR`, the number of American Airlines flights from `JFK`, and so on. Recall the `ggplot()` code that created the barplot of `carrier` frequency in Figure 2.26:

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```



We can now map the additional variable `origin` by adding a `fill = origin` inside the `aes()` aesthetic mapping; the `fill` aesthetic of any bar corresponds to the color used to fill the bars.

```
ggplot(data = flights, mapping = aes(x = carrier, fill = origin)) +
  geom_bar()
```

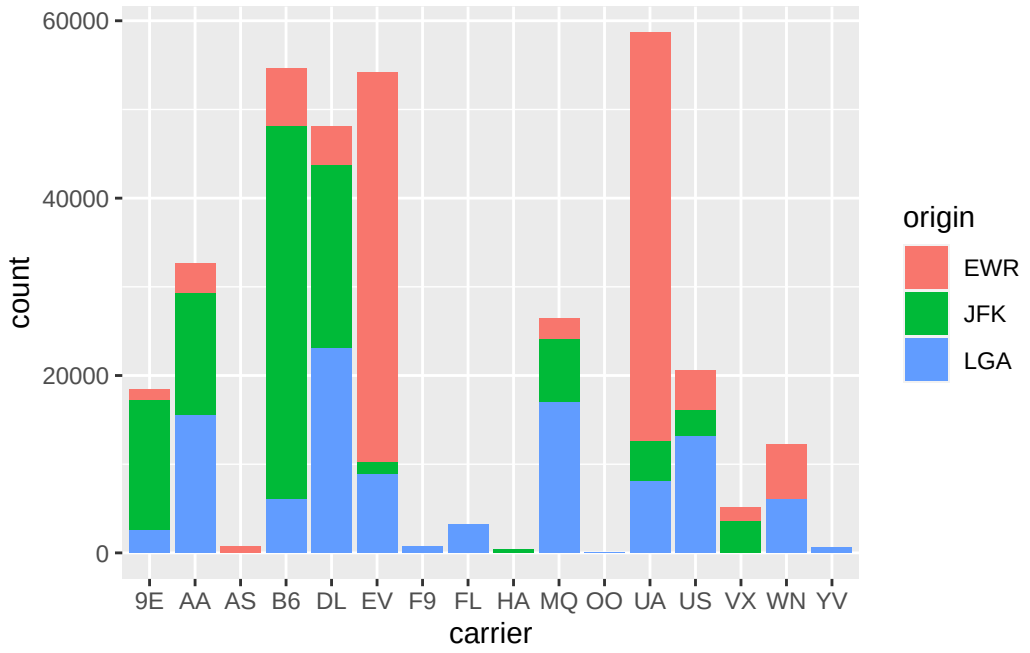


Figure 2.29: Stacked barplot comparing the number of flights by carrier and origin

Figure 2.29 is an example of a *stacked barplot*. While simple to make, in certain aspects it is not ideal. For example, it is difficult to compare the heights of the different colors between the bars, corresponding to comparing the number of flights from each `origin` airport between the carriers.

Before we continue, let's address some common points of confusion amongst new R users. First, note that `fill` is another aesthetic mapping much like `x`-position; thus it must be included within the parentheses of the `aes()` mapping. The following code, where the `fill` aesthetic is specified outside the `aes()` mapping will yield an error. This is a fairly common error that new `ggplot` users make:

```
ggplot(data = flights, mapping = aes(x = carrier), fill = origin) +
  geom_bar()
```

Second, the `fill` aesthetic corresponds to the color used to fill the bars, while the `color` aesthetic corresponds to the color of the outline of the bars. Observe in Figure 2.30 that mapping `origin` to `color` and not `fill` yields grey bars with different colored outlines.

```
ggplot(data = flights, mapping = aes(x = carrier, color = origin)) +
  geom_bar()
```

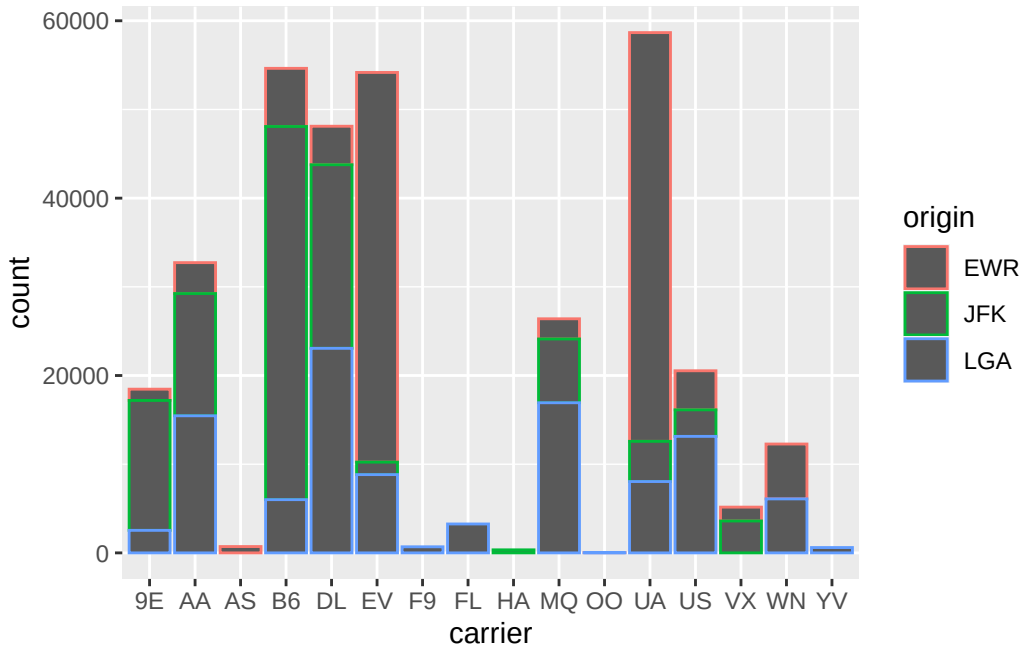


Figure 2.30: Stacked barplot with color aesthetic used instead of fill

Learning Check 2.32

What kinds of questions are not easily answered by looking at the above figure?

Learning Check 2.33

What can you say, if anything, about the relationship between airline and airport in NYC in 2013 in regards to the number of departing flights?

Another alternative to stacked barplots are *side-by-side barplots*, also known as a *dodged barplot*. The code to created a side-by-side barplot is identical to the code to create a stacked barplot, but with a `position = "dodge"` argument added to `geom_bar()`. In other words, we are overriding the default barplot type, which is a stacked barplot, and specifying it to be a side-by-side barplot.

```
ggplot(data = flights, mapping = aes(x = carrier, fill = origin)) +
  geom_bar(position = "dodge")
```

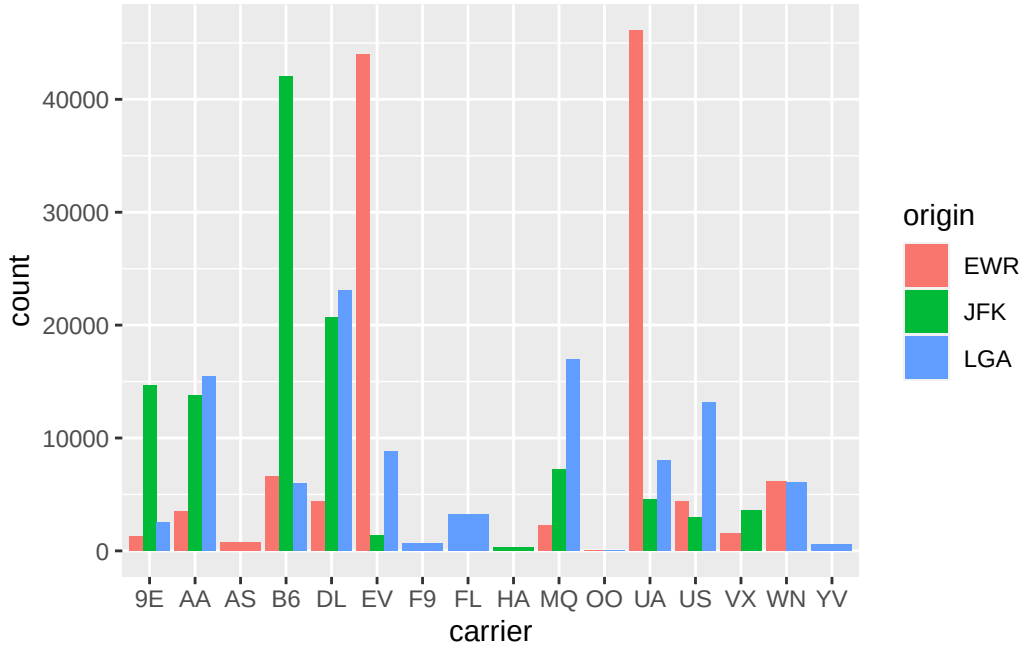


Figure 2.31: Side-by-side AKA dodged barplot comparing the number of flights by carrier and origin

Learning Check 2.34

Why might the side-by-side (AKA dodged) barplot be preferable to a stacked barplot in this case?

Learning Check 2.35

What are the disadvantages of using a side-by-side (AKA dodged) barplot, in general?

Lastly, another type of barplot is a *faceted barplot*. Recall in Section 2.6 we visualized the distribution of hourly temperatures at the 3 NYC airports *split* by month using facets. We apply the same principle to our barplot visualizing the frequency of `carrier` split by `origin`: instead of mapping `origin`

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar() +
  facet_wrap(~ origin, ncol = 1)
```

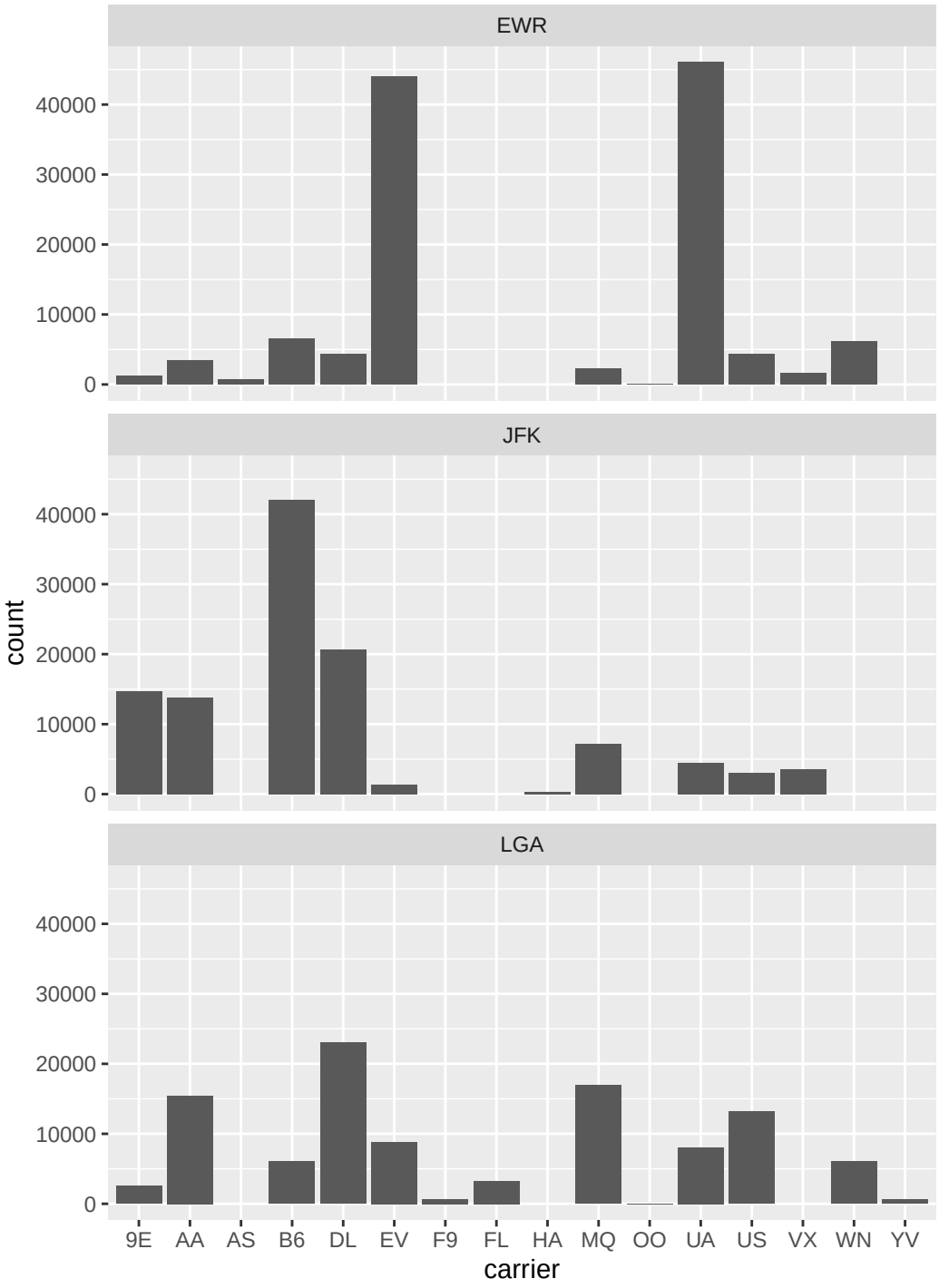


Figure 2.32: Faceted barplot comparing the number of flights by carrier and origin

Learning Check 2.36

Why is the faceted barplot preferred to the side-by-side and stacked barplots in this case?

Learning Check 2.37

What information about the different carriers at different airports is more easily seen in the faceted barplot?

2.8.4 Summary

Barplots are the preferred way of displaying the distribution of a categorical variable, or in other words the frequency with which the different categories called *levels* occur. They are easy to understand and make it easy to make comparisons across levels. When trying to visualize two categorical variables, you have many options: stacked barplots, side-by-side barplots, and faceted barplots. Depending on what aspect of the joint distribution you are trying to emphasize, you will need to make a choice between these three types of barplots.

2.9 Conclusion

2.9.1 Summary table

Let's recap all five of the Five Named Graphs (5NG) in Table 2.4 summarizing their differences. Using these 5NG, you'll be able to visualize the distributions and relationships of variables contained in a wide array of datasets. This will be even more the case as we start to map more variables to more of each `geom`etric object's `aes`thetic attribute options, further unlocking the awesome power of the `ggplot2` package.

Table 2.4: Summary of 5NG

Named graph	Shows	Geometric object	Notes
Scatterplot	Relationship between 2 numerical variables	`geom_point()`	
Linegraph	Relationship between 2 numerical variables	`geom_line()`	Used when there is a sequential order to x-variable e.g. time
Histogram	Distribution of 1 numerical variable	`geom_histogram()`	Facetted histograms show the distribution of 1 numerical variable split by the values of another variable

Named graph	Shows	Geometric object	Notes
Boxplot	Distribution of 1 numerical variable split by the values of another variable	`geom_boxplot()`	
Barplot	Distribution of 1 categorical variable	`geom_bar()` when counts are not pre-counted, `geom_col()` when counts are pre-counted	Stacked, side-by-side, and faceted barplots show the joint distribution of 2 categorical variables

2.9.2 Argument specification

Run the following two segments of code. First this:

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

then this:

```
ggplot(flights, aes(x = carrier)) +
  geom_bar()
```

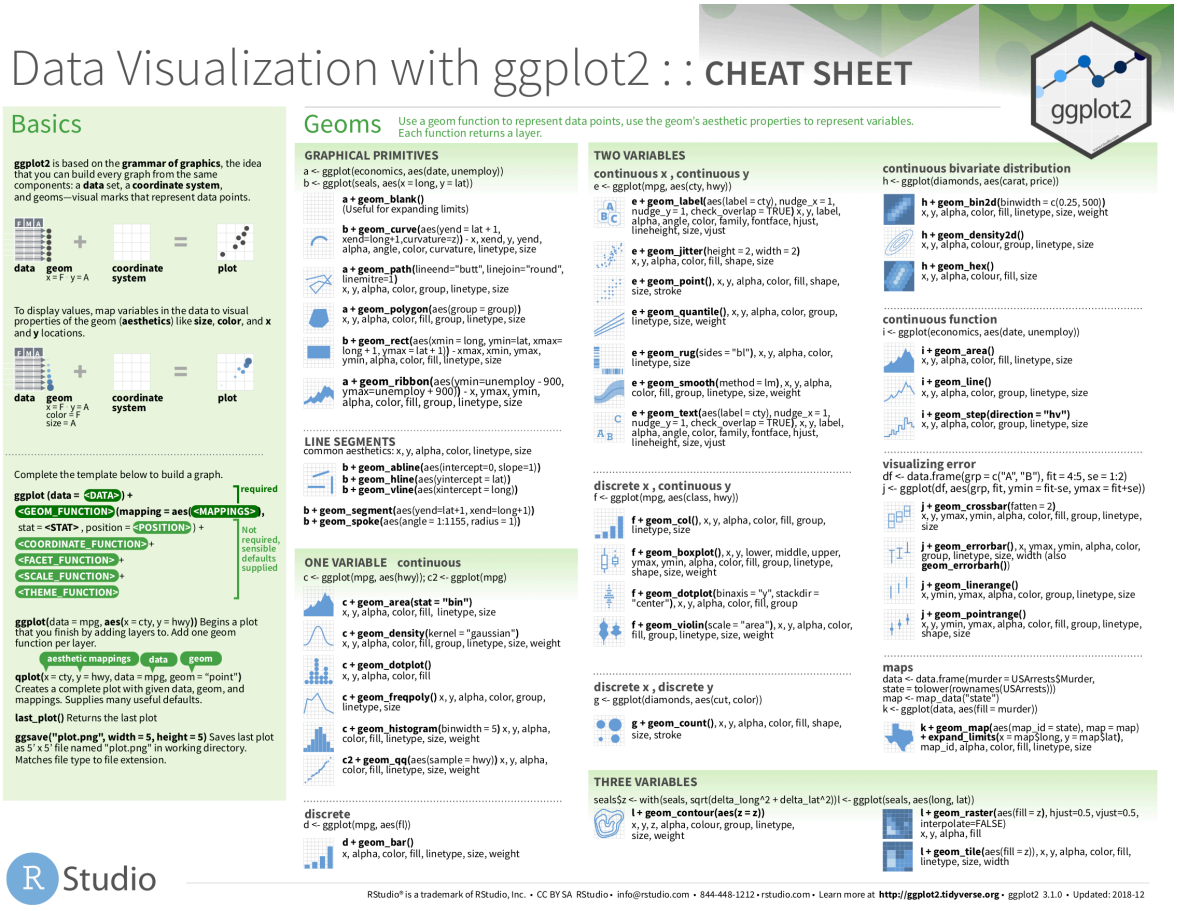
You'll notice that that both code segments create the same barplot, even though in the second segment we omitted the `data =` and `mapping =` code argument names. This is because the `ggplot()` by default assumes that the `data` argument comes first and the `mapping` argument comes second. So as long as you specify the data frame in question first and the `aes()` mapping second, you can omit the explicit statement of the argument names `data =` and `mapping =`.

Going forward for the rest of this book, all `ggplot()` will be like the second segment above: with the `data =` and `mapping =` explicit naming of the argument omitted and the default ordering of arguments respected.

2.9.3 Additional resources

If you want to further unlock the power of the `ggplot2` package for data visualization, we suggest you that you check out RStudio's "Data Visualization with ggplot2" cheatsheet. This cheatsheet summarizes much more than what we've discussed in this chapter, in particular the many more than the 5 `geom` geometric objects we covered in this Chapter, while providing quick and easy to read visual descriptions.

You can access this cheatsheet by going to the RStudio Menu Bar -> Help -> Cheatsheets -> "Data Visualization with ggplot2":



2.9.4 What's to come

Recall in Figure 2.2 in Section 2.3 we visualized the relationship between departure delay and arrival delay for Alaska Airlines flights. This necessitated paring or filtering down the `flights` data frame to a new data frame `alaska_flights` consisting of only `carrier == AS` flights first:

```
alaska_flights <- flights %>%
  filter(carrier == "AS")

ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

Furthermore recall in Figure 2.8 in Section 2.4 we visualized hourly temperature recordings at Newark airport only for the first 15 days of January 2013. This necessitated paring or fitlering down the `weather` data frame to a new data frame `early_january_weather` consisting of hourly temperature recordings only for `origin == "EWR"`, `month == 1`, and day less than or equal to `15` first:

```
early_january_weather <- weather %>%
  filter(origin == "EWR" & month == 1 & day <= 15)

ggplot(data = early_january_weather, mapping = aes(x = time_hour, y = temp)) +
  geom_line()
```

These two code segments were a preview of Chapter 3 on data wrangling where we'll delve further into the `dplyr` package. Data wrangling is the process of transforming and modifying existing data with the intent of making it more appropriate for analysis purposes. For example, the two code segments used the `filter()` function to create new data frames (`alaska_flights` and `early_january_weather`) by choosing only a subset of rows of existing data frames (`flights` and `weather`). In this next chapter, we'll formally introduce the `filter()` and other data wrangling functions as well as the *pipe operator* `%>%` which allows you to combine multiple data wrangling actions into a single sequential *chain* of actions. On to Chapter 3 on data wrangling!

3 Data Wrangling

So far in our journey, we've seen how to look at data saved in data frames using the `glimpse()` and `View()` functions in Chapter 1 on and how to create data visualizations using the `ggplot2` package in Chapter 2. In particular we studied what we term the "five named graphs" (5NG):

1. scatterplots via `geom_point()`
2. linegraphs via `geom_line()`
3. boxplots via `geom_boxplot()`
4. histograms via `geom_histogram()`
5. barplots via `geom_bar()` or `geom_col()`

We created these visualizations using the "Grammar of Graphics", which maps variables in a data frame to the aesthetic attributes of one the above 5 `geom`etric objects. We can also control other aesthetic attributes of the geometric objects such as the size and color as seen in the Gapminder data example in Figure Figure 2.1.

Recall however in Section 2.9.4 we discussed that for two of our visualizations we needed transformed/modified versions of existing data frames. Recall for example the scatterplot of departure and arrival delay *only* for Alaska Airlines flights. In order to create this visualization, we needed to first pare down the `flights` data frame to a new data frame `alaska_flights` consisting of only `carrier == "AS"` flights using the `filter()` function.

```
alaska_flights <- flights %>%
  filter(carrier == "AS")

ggplot(data = alaska_flights, mapping = aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

In this chapter, we'll introduce a series of functions from the `dplyr` package that will allow you to take a data frame and

1. `filter()` its existing rows to only pick out a subset of them. For example, the `alaska_flights` data frame above.
2. `summarize()` one of its columns/variables with a *summary statistic*. Examples include the median and interquartile range of temperatures as we saw in Section 2.7 on boxplots.

3. `group_by()` its rows. In other words assign different rows to be part of the same *group* and report summary statistics for each group separately. For example, say perhaps you don't want a single overall average departure delay `dep_delay` for all three `origin` airports combined, but rather three separate average departure delays, one for each of the three `origin` airports.
4. `mutate()` its existing columns/variables to create new ones. For example, convert hourly temperature recordings from °F to °C.
5. `arrange()` its rows. For example, sort the rows of `weather` in ascending or descending order of `temp`.
6. `join()` it with another data frame by matching along a "key" variable. In other words, merge these two data frames together.

Notice how we used `computer code` font to describe the actions we want to take on our data frames. This is because the `dplyr` package for data wrangling that we'll introduce in this chapter has intuitively verb-named functions that are easy to remember.

We'll start by introducing the pipe operator `%>%`, which allows you to combine multiple data wrangling verb-named functions into a single sequential *chain* of actions.

Packages Needed

Let's load all the packages needed for this chapter (this assumes you've already installed them). If needed, read Section 1.3 for information on how to install and load R packages.

```
library(dplyr)
library(ggplot2)
library(nycflights13)
```

3.1 The pipe operator: %>%

Before we start data wrangling, let's first introduce a very nifty tool that gets loaded along with the `dplyr` package: the pipe operator `%>%`. Say you would like to perform a hypothetical sequence of operations on a hypothetical data frame `x` using hypothetical functions `f()`, `g()`, and `h()`:

1. Take `x` *then*
2. Use `x` as an input to a function `f()` *then*
3. Use the output of `f(x)` as an input to a function `g()` *then*
4. Use the output of `g(f(x))` as an input to a function `h()`

One way to achieve this sequence of operations is by using nesting parentheses as follows:

```
h(g(f(x)))
```

The above code isn't so hard to read since we are applying only three functions: `f()`, then `g()`, then `h()`. However, you can imagine that this can get progressively harder and harder to read as the number of functions applied in your sequence increases. This is where the pipe operator `%>%` comes in handy. `%>%` takes one output of one function and then "pipes" it to be the input of the next function. Furthermore, a helpful trick is to read `%>%` as "then." For example, you can obtain the same output as the above sequence of operations as follows:

```
x %>%
  f() %>%
  g() %>%
  h()
```

You would read this above sequence as:

1. Take `x` *then*
2. Use this output as the input to the next function `f()` *then*
3. Use this output as the input to the next function `g()` *then*
4. Use this output as the input to the next function `h()`

So while both approaches above would achieve the same goal, the latter is much more human-readable because you can read the sequence of operations line-by-line. But what are the hypothetical `x`, `f()`, `g()`, and `h()`? Throughout this chapter on data wrangling:

- The starting value `x` will be a data frame. For example: `flights`.
- The sequence of functions, here `f()`, `g()`, and `h()`, will be a sequence of any number of the 6 data wrangling verb-named functions we listed in the introduction to this chapter. For example: `filter(carrier == "AS")`.
- The result will be the transformed/modified data frame that you want. For example: a data frame consisting of only the subset of rows in `flights` corresponding to Alaska Airlines flights.

Much like when adding layers to a `ggplot()` using the `+` sign at the end of lines, you form a single *chain* of data wrangling operations by combining verb-named functions into a single sequence with pipe operators `%>%` at the end of lines. So continuing our example involving Alaska Airlines flights, we form a chain using the pipe operator `%>%` and save the resulting data frame in `alaska_flights`:

```
alaska_flights <- flights %>%
  filter(carrier == "AS")
```

Keep in mind, there are many more advanced data wrangling functions than just the 6 listed in the introduction to this chapter; you'll see some examples of these in Section 3.8. However, just with these 6 verb-named functions you'll be able to perform a broad array of data wrangling tasks for the rest of this book.

3.2 `filter()` rows

Figure 3.1: Diagram of filter()

The `filter()` function here works much like the "Filter" option in Microsoft Excel; it allows you to specify criteria about the values of a variable in your dataset and then filters out only those rows that match that criteria. We begin by focusing only on flights from New York City to Portland, Oregon. The `dest` code (or airport code) for Portland, Oregon is `"PDX"`. Run the following and look at the resulting spreadsheet to ensure that only flights heading to Portland are chosen here:

```
portland_flights <- flights %>%
  filter(dest == "PDX")

View(portland_flights)
```

Note the following:

- The ordering of the commands:
  - Take the `flights` data frame `flights` *then*
  - `filter` the data frame so that only those where the `dest` equals `"PDX"` are included.
- We test for equality using the double equal sign `==` and not a single equal sign `=`. In other words `filter(dest = "PDX")` will yield an error. This is a convention across

many programming languages. If you are new to coding, you'll probably forget to use the double equal sign `==` a few times before you get the hang of it.

You can use other mathematical operations beyond just `==` to form criteria:

- `>` corresponds to "greater than"
- `<` corresponds to "less than"
- `>=` corresponds to "greater than or equal to"
- `<=` corresponds to "less than or equal to"
- `!=` corresponds to "not equal to". The `!` is used in many programming languages to indicate "not".

Furthermore, you can combine multiple criteria together using operators that make comparisons:

- `|` corresponds to "or"
- `&` corresponds to "and"

To see many of these in action, let's filter `flights` for all rows that:

- Departed from JFK airport and
- Were heading to Burlington, Vermont (`"BTV"`) or Seattle, Washington (`"SEA"`) and
- Departed in the months of October, November, or December.

Run the following:

```
btv_sea_flights_fall <- flights %>%
  filter(origin == "JFK" & (dest == "BTV" | dest == "SEA") & month >= 10)

View(btv_sea_flights_fall)
```

Note that even though colloquially speaking one might say "all flights leaving Burlington, Vermont *and* Seattle, Washington," in terms of computer operations, we really mean "all flights leaving Burlington, Vermont *or* leaving Seattle, Washington." For a given row in the data, `dest` can be "BTV", "SEA", or something else, but not "BTV" and "SEA" at the same time. Furthermore, note the careful use of parentheses around the `dest == "BTV" | dest == "SEA"`.

We can often skip the use of `&` and just separate our conditions with a comma. In other words the code above will return the identical output `btv_sea_flights_fall` as this code below:

```
btv_sea_flights_fall <- flights %>%
  filter(origin == "JFK", (dest == "BTV" | dest == "SEA"), month >= 10)

View(btv_sea_flights_fall)
```

Let's present another example that uses the `!` "not" operator to pick rows that *don't* match a criteria. As mentioned earlier, the `!` can be read as "not." Here we are filtering rows corresponding to flights that didn't go to Burlington, VT or Seattle, WA.

```
not_BTV_SEA <- flights %>%
  filter(!(dest == "BTV" | dest == "SEA"))

View(not_BTV_SEA)
```

Again, note the careful use of parentheses around the `(dest == "BTV" | dest == "SEA")`. If we didn't use parentheses as follows:

```
flights %>%
  filter(!dest == "BTV" | dest == "SEA")
```

We would be returning all flights not headed to `"BTV"` *or* those headed to `"SEA"`, which is an entirely different resulting data frame.

Now say we have a large list of airports we want to filter for, say `BTV`, `SEA`, `PDX`, `SFO`, and `BDL`. We could continue to use the `|` or operator as so:

```
many_airports <- flights %>%
  filter(dest == "BTV" | dest == "SEA" | dest == "PDX" | dest == "SFO" | dest == "BDL")

View(many_airports)
```

but as we progressively include more airports, this will get unwieldy. A slightly shorter approach uses the `%in%` operator:

```
many_airports <- flights %>%
  filter(dest %in% c("BTV", "SEA", "PDX", "SFO", "BDL"))

View(many_airports)
```

What this code is doing is filtering `flights` for all flights where `dest` is in the list of airports `c("BTV", "SEA", "PDX", "SFO", "BDL")`. Recall from Chapter 1 that the `c()` function "combines" or "concatenates" values in a vector of values. Both outputs of `many_airports` are the same, but as you can see the latter takes much less time to code.

As a final note we point out that `filter()` should often be among the first verbs you apply to your data. This cleans your dataset to only those rows you care about, or put differently, it narrows down the scope of your data frame to just the observations your care about.

Learning Check 3.1

What's another way of using the "not" operator `!` to filter only the rows that are not going to Burlington VT nor Seattle WA in the `flights` data frame? Test this out using the code above.

3.3 `summarize()` variables

The next common task when working with data is to return *summary statistics*: a single numerical value that summarizes a large number of values, for example the mean/average or the median. Other examples of summary statistics that might not immediately come to mind include the sum, the smallest value AKA the minimum, the largest value AKA the maximum, and the standard deviation; they are all summaries of a large number of values.

Figure 3.2: Summarize diagram from Data Wrangling with dplyr and tidyr cheatsheet

Figure 3.3: Another summarize diagram from Data Wrangling with dplyr and tidyr

Let's calculate the mean and the standard deviation of the temperature variable `temp` in the `weather` data frame included in the `nycflights13` package (See Appendix A). We'll do this in one step using the `summarize()` function from the `dplyr` package and save the results in a new data frame `summary_temp` with columns/variables `mean` and the `std_dev`. Note you can also use the UK spelling of `summarise()`.

The `weather` data frame's many rows will bow be collapsed into a single row of just the summary values, in this case the mean and standard deviation:

```
summary_temp <- weather %>%
  summarize(mean = mean(temp), std_dev = sd(temp))

summary_temp
```

```
# A tibble: 1 x 2
   mean std_dev
  <dbl>   <dbl>
1    NA      NA
```

Why are the values returned `NA`? As we saw in Section 2.3.1 when creating the scatterplot of departure and arrival delays for `alaska_flights`, `NA` is how R encodes *missing values* where `NA` indicates "not available" or "not applicable." If a value for a particular row and a particular column does not exist, `NA` is stored instead. Values can be missing for many reasons. Perhaps the data was collected but someone forgot to enter it? Perhaps the data was not collected at all because it was too difficult? Perhaps there was an erroneous value that someone entered that has been correct to read as missing? You'll often encounter issues with missing values when working with real data.

Going back to our `summary_temp` output above, by default any time you try to calculate a summary statistic of a variable that has one or more `NA` missing values in R, then `NA` is returned. To work around this fact, you can set the `na.rm` argument to `TRUE`, where `rm` is short for "remove"; this will ignore any `NA` missing values and only return the summary value for all non-missing values.

The code below computes the mean and standard deviation of all non-missing values of `temp`. Notice how the `na.rm=TRUE` are used as arguments to the `mean()` and `sd()` functions individually, and not to the `summarize()` function.

```
summary_temp <- weather %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))

summary_temp
```

```
# A tibble: 1 x 2
   mean std_dev
  <dbl>   <dbl>
1  55.3    17.8
```

However, one needs to be cautious whenever ignoring missing values as we've done above. In the upcoming Learning Checks we'll consider the possible ramifications of blindly sweeping rows with missing values "under the rug." This is in fact why the `na.rm` argument to any summary statistic function in R has is set to `FALSE` by default; in other words, do not ignore rows with missing values by default. R is alerting you to the presence of missing data and you should by mindful of this missingness and any potential causes of this missingness throughout your analysis.

What are other functions for summary statistics can we use inside the `summarize()` verb? We can use any function in R that takes many values and returns just one. Here are just a few:

- `mean()`: the mean AKA the average
- `sd()`: the standard deviation, which is a measure of spread
- `min()` and `max()`: the minimum and maximum values respectively
- `IQR()`: Interquartile range
- `sum()`: the sum
- `n()`: a count of the number of rows/observations in each group. This particular summary function will make more sense when `group_by()` is covered in Section 3.4.

Learning Check 3.2

Say a doctor is studying the effect of smoking on lung cancer for a large number of patients who have records measured at five year intervals. She notices that a large number of patients have missing data points because the patient has died, so she chooses to ignore these patients in her analysis. What is wrong with this doctor's approach?

Learning Check 3.3

Modify the above `summarize()` function to create `summary_temp` to also use the `n()` summary function: `summarize(count = n())`.
What does the returned value correspond to?

Learning Check 3.4

Why doesn't the following code work?

```
summary_temp <- weather %>%
  summarize(mean = mean(temp, na.rm = TRUE)) %>%
  summarize(std_dev = sd(temp, na.rm = TRUE))
```

Hiint: Run the code line by line instead of all at once, and then look at the data. In other words, run `summary_temp <- weather %>% summarize(mean = mean(temp, na.rm = TRUE))` first.

3.4 `group_by()` rows

Figure 3.4: Group by and summarize diagram from Data Wrangling with dplyr and tidyr cheatsheet

Say instead of the a single mean temperature for the whole year, you would like 12 mean temperatures, one for each of the 12 months separately? In other words, we would like to compute the mean temperature split by month AKA sliced by month AKA aggregated by month. We can do this by "grouping" temperature observations by the values of another variable, in this case by the 12 values of the variable `month`. Run the following code:

```
summary_monthly_temp <- weather %>%
  group_by(month) %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))

summary_monthly_temp
```

```
# A tibble: 12 x 3
   month  mean std_dev
   <int> <dbl>   <dbl>
 1     1  35.6   10.2
 2     2  34.3    6.98
 3     3  39.9    6.25
 4     4  51.7    8.79
 5     5  61.8    9.68
```

```
 6     6  72.2    7.55
 7     7  80.1    7.12
 8     8  74.5    5.19
 9     9  67.4    8.47
10    10  60.1    8.85
11    11  45.0   10.4
12    12  38.4    9.98
```

This code is identical to the previous code that created `summary_temp`, but with an extra `group_by(month)` added before the `summarize()`. Grouping the `weather` dataset by `month` and then applying the `summarize()` functions yields a data frame that displays the mean and standard deviation temperature split by the 12 months of the year.

It is important to note that the `group_by()` function doesn't change data frame by itself. Rather it changes the *meta-data*, or data about the data, specifically the group structure. It is only after we apply the `summarize()` function that the data frame changes. For example, let's consider the `diamonds` data frame included in the `ggplot2` package. Run this code, specifically in the console:

```
diamonds
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Observe that the first line of the output reads `# A tibble: 53,940 x 10`. This is an example of meta-data, in this case the number of observations/rows and variables/columns in `diamonds`. The actual data itself are the subsequent table of values.

Now let's pipe the `diamonds` data frame into `group_by(cut)`. Run this code, specifically in the console:

```
diamonds %>%
  group_by(cut)
```

```
# A tibble: 53,940 x 10
# Groups:   cut [5]
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Observe that now there is additional meta-data: `# Groups: cut [5]` indicating that the grouping structure meta-data has been set based on the 5 possible values AKA levels of the categorical variable `cut`: `"Fair"`, `"Good"`, `"Very Good"`, `"Premium"`, `"Ideal"`. On the other hand observe that the data has not changed: it is still a table of 53,940 $\times$ 10 values.

Only by combining a `group_by()` with another data wrangling operation, in this case `summarize()` will the actual data be transformed.

```
diamonds %>%
  group_by(cut) %>%
  summarize(avg_price = mean(price))
```

```
# A tibble: 5 x 2
  cut       avg_price
  <ord>         <dbl>
1 Fair          4359.
2 Good          3929.
3 Very Good     3982.
4 Premium       4584.
5 Ideal         3458.
```

If we would like to remove this group structure meta-data, we can pipe the resulting data frame into the `ungroup()` function. Observe how the `# Groups: cut [5]` meta-data is no longer present. Run this code, specifically in the console:

```
diamonds %>%
  group_by(cut) %>%
  ungroup()
```

```
# A tibble: 53,940 x 10
   carat cut       color clarity depth table price     x     y     z
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
# ... with 53,930 more rows
```

Let's now revisit `n()` thw counting summary function introduced in the previous section. For example, suppose we'd like to count how many flights departed each of the three airports in New York City:

```
by_origin <- flights %>%
  group_by(origin) %>%
  summarize(count = n())

by_origin
```

```
# A tibble: 3 x 2
  origin  count
  <chr>   <int>
1 EWR    120835
2 JFK    111279
3 LGA    104662
```

We see that Newark (`"EWR"`) had the most flights departing in 2013 followed by `"JFK"` and lastly by LaGuardia (`"LGA"`). Note there is a subtle but important difference between `sum()` and `n()`; While `sum()` returns the sum of a numerical variable, `n()` returns counts of the the number of rows/observations.

3.4.1 Grouping by more than one variable

You are not limited to grouping by one variable! Say you wanted to know the number of flights leaving each of the three New York City airports *for each month*, we can also group by a second variable `month`: `group_by(origin, month)`. We see there are 36 rows to `by_origin_monthly` because there are 12 months for 3 airports (`EWR`, `JFK`, and `LGA`).

```
by_origin_monthly <- flights %>%
  group_by(origin, month) %>%
  summarize(count = n())

by_origin_monthly
```

```
# A tibble: 36 x 3
# Groups:   origin [3]
   origin month count
   <chr>  <int> <int>
 1 EWR        1  9893
 2 EWR        2  9107
 3 EWR        3 10420
 4 EWR        4 10531
 5 EWR        5 10592
 6 EWR        6 10175
 7 EWR        7 10475
 8 EWR        8 10359
 9 EWR        9  9550
10 EWR       10 10104
# ... with 26 more rows
```

Why do we `group_by(origin, month)` and not `group_by(origin)` and then `group_by(month)`? Let's investigate:

```
by_origin_monthly_incorrect <- flights %>%
  group_by(origin) %>%
  group_by(month) %>%
  summarize(count = n())
```

```
by_origin_monthly_incorrect
```

```
# A tibble: 12 x 2
   month count
   <int> <int>
 1     1 27004
 2     2 24951
 3     3 28834
 4     4 28330
 5     5 28796
 6     6 28243
 7     7 29425
 8     8 29327
 9     9 27574
10    10 28889
11    11 27268
12    12 28135
```

What happened here is that the second `group_by(month)` overrode the group structure metadata of the first `group_by(origin)`, so that in the end we are only grouping by `month`. The lesson here is if you want to `group_by()` two or more variables, you should include all these variables in a single `group_by()` function call.

> Learning Check 3.5

(LC3.6) Recall from Chapter @ref(viz) when we looked at plots of temperatures by months in NYC. What does the standard deviation column in the `summary_monthly_temp` data frame tell us about temperatures in New York City throughout the year?

> Learning Check 3.7
>
> What code would be required to get the mean and standard deviation temperature for each day in 2013 for NYC?

> Learning Check 3.8
>
> Recreate `by_monthly_origin`, but instead of grouping via `group_by(origin, month)`, group variables in a different order `group_by(month, origin)`.
> What differs in the resulting dataset?

Learning Check 3.9

How could we identify how many flights left each of the three airports for each `carrier`?

Learning Check 3.10

How does the `filter` operation differ from a `group_by` followed by a `summarize`?

3.5 `mutate` existing variables

Figure 3.5: Mutate diagram from Data Wrangling with dplyr and tidyr cheatsheet

Another common transformation of data is to create/compute new variables based on existing ones. For example, say you are more comfortable thinking of temperature in degrees Celsius °C and not degrees Farenheit °F. The formula to convert temperatures from °F to °C is:

$$\text{temp in C} = \frac{\text{temp in F} - 32}{1.8}$$

We can apply this formula to the `temp` variable using the `mutate()` function, which takes existing variables and mutates them to create new ones.

```
weather <- weather %>%
  mutate(temp_in_C = (temp-32)/1.8)

View(weather)
```

Note that we have overwritten the original `weather` data frame with a new version that now includes the additional variable `temp_in_C`. In other words, the `mutate()` command outputs a new data frame which then gets saved over the original `weather` data frame. Furthermore, note how in `mutate()` we used `temp_in_C = (temp-32)/1.8` to create a new variable `temp_in_C`.

Why did we overwrite the data frame `weather` instead of assigning the result to a new data frame like `weather_new`, but on the other hand why did we *not* overwrite `temp`, but instead created a new variable called `temp_in_C`? As a rough rule of thumb, as long as you are not losing original information that you might need later, it's acceptable practice to overwrite existing data frames. On the other hand, had we used `mutate(temp = (temp-32)/1.8)` instead of `mutate(temp_in_C = (temp-32)/1.8)`, we would have overwritten the original variable `temp` and lost its values.

Let's compute average monthly temperatures in both °F and °C using the similar `group_by()` and `summarize()` code as in the previous section.

```
summary_monthly_temp <- weather %>%
  group_by(month) %>%
  summarize(
    mean_temp_in_F = mean(temp, na.rm = TRUE),
    mean_temp_in_C = mean(temp_in_C, na.rm = TRUE)
    )

summary_monthly_temp
```

```
# A tibble: 12 x 3
   month mean_temp_in_F mean_temp_in_C
   <int>          <dbl>          <dbl>
 1     1           35.6           2.02
 2     2           34.3           1.26
 3     3           39.9           4.38
 4     4           51.7          11.0
 5     5           61.8          16.6
 6     6           72.2          22.3
 7     7           80.1          26.7
 8     8           74.5          23.6
 9     9           67.4          19.7
10    10           60.1          15.6
11    11           45.0           7.22
12    12           38.4           3.58
```

Let's consider another example. Passengers are often frustrated when their flights depart late, but change their mood a bit if pilots can make up some time during the flight to get them to their destination close to the original arrival time. This is commonly referred to as "gain" and we will create this variable using the `mutate()` function.

```
flights <- flights %>%
  mutate(gain = dep_delay - arr_delay)
```

Let's take a look at `dep_delay`, `arr_delay`, and the resulting `gain` variables for the first 5 rows in our new `flights` data frame:

```
# A tibble: 5 x 3
  dep_delay arr_delay  gain
      <dbl>     <dbl> <dbl>
1         2        11    -9
2         4        20   -16
3         2        33   -31
4        -1       -18    17
5        -6       -25    19
```

The flight in the first row departed 2 minutes late but arrived 11 minutes late, so its "gained time in the air" is actually a loss of 9 minutes, hence its `gain` is `-9`. Contrast this to the flight in the fourth row which departed a minute early (`dep_delay` of `-1`) but arrived 18 minutes early (`arr_delay` of `-18`), so its "gained time in the air" is 17 minutes, hence its `gain` is `+17`.

Let's look at summary measures of this `gain` variable and even plot it in the form of a histogram:

```
gain_summary <- flights %>%
  summarize(
    min = min(gain, na.rm = TRUE),
    q1 = quantile(gain, 0.25, na.rm = TRUE),
    median = quantile(gain, 0.5, na.rm = TRUE),
    q3 = quantile(gain, 0.75, na.rm = TRUE),
    max = max(gain, na.rm = TRUE),
    mean = mean(gain, na.rm = TRUE),
    sd = sd(gain, na.rm = TRUE),
    missing = sum(is.na(gain))
  )

gain_summary
```

min	q1	median	q3	max	mean	sd	missing
-196	-3	7	17	109	5.66	18	9430

We've recreated the `summary` function we saw in Chapter 2 here using the `summarize` function in `dplyr`.

```
ggplot(data = flights, mapping = aes(x = gain)) +
  geom_histogram(color = "white", bins = 20)
```

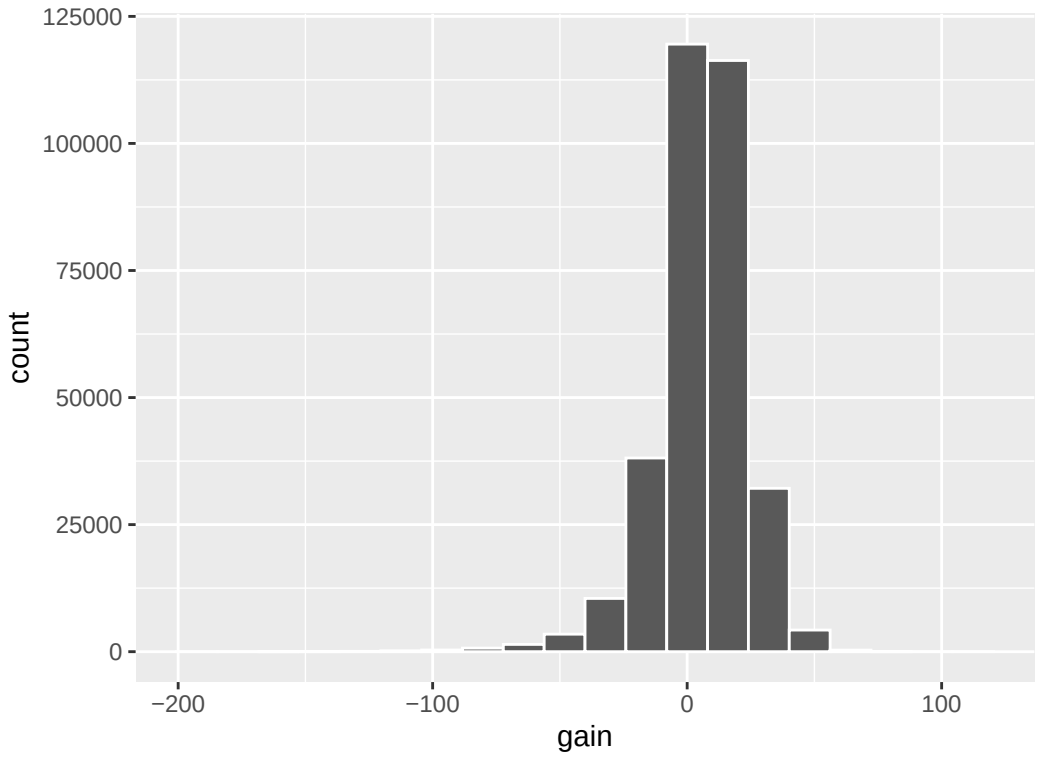


Figure 3.6: Histogram of gain variable

We can also create multiple columns at once and even refer to columns that were just created in a new column. Hadley and Garrett produce one such example in Chapter 5 of "R for Data Science" (Grolemund and Wickham 2016):

```
flights <- flights %>%
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

Learning Check 3.11

What do positive values of the `gain` variable in `flights` correspond to? What about negative values? And what about a zero value?

Learning Check 3.12

Could we create the `dep_delay` and `arr_delay` columns by simply subtracting `dep_time` from `sched_dep_time` and similarly for arrivals? Try the code out and explain any differences between the result and what actually appears in `flights`.

Learning Check 3.13

What can we say about the distribution of `gain`? Describe it in a few sentences using the plot and the `gain_summary` data frame values.

3.6 `arrange()` and sort rows

One of the most common tasks people working with data would like to perform is sort the data frame's rows in alphanumeric order of the values in a variable/column. For example, when calculating a median by hand requires you to first sort the data from the smallest to highest in value and then identify the "middle" value. The `dplyr` package has a function called `arrange()` that we will use to sort/reorder a data frame's rows according to the values of the specified variable. This is often used after we have used the `group_by()` and `summarize()` functions as we will see.

Let's suppose we were interested in determining the most frequent destination airports for all domestic flights departing from New York City in 2013:

```
freq_dest <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n())

freq_dest
```

```
# A tibble: 105 x 2
   dest  num_flights
   <chr>       <int>
 1 ABQ           254
 2 ACK           265
```

```
 3 ALB           439
 4 ANC             8
 5 ATL         17215
 6 AUS          2439
 7 AVL           275
 8 BDL           443
 9 BGR           375
10 BHM           297
# ... with 95 more rows
```

Observe that by default the rows of the resulting `freq_dest` data frame are sorted in alphabetical order of `dest` destination. Say instead we would like to see the same data, but sorted from the most to the least number of flights `num_flights` instead:

```
freq_dest %>%
  arrange(num_flights)
```

```
# A tibble: 105 x 2
   dest  num_flights
   <chr>       <int>
 1 LEX             1
 2 LGA             1
 3 ANC             8
 4 SBN            10
 5 HDN            15
 6 MTJ            15
 7 EYW            17
 8 PSP            19
 9 JAC            25
10 BZN            36
# ... with 95 more rows
```

This is actually giving us the opposite of what we are looking for: the rows are sorted with the least frequent destination airports displayed first. To switch the ordering to be descending instead of ascending we use the `desc()` function, which is short for "descending":

```
freq_dest %>%
  arrange(desc(num_flights))
```

```
# A tibble: 105 x 2
```

```
   dest  num_flights
   <chr>       <int>
 1 ORD         17283
 2 ATL         17215
 3 LAX         16174
 4 BOS         15508
 5 MCO         14082
 6 CLT         14064
 7 SFO         13331
 8 FLL         12055
 9 MIA         11728
10 DCA          9705
# ... with 95 more rows
```

In other words, `arrange()` sorts in ascending order by default unless you override this default behavior by using `desc()`.

3.7 `join` data frames

Another common data transformation task is "joining" or "merging" two different datasets. For example in the `flights` data frame the variable `carrier` lists the carrier code for the different flights. While the corresponding airline names for `"UA"` and `"AA"` might be somewhat easy to guess (United and American Airlines), what airlines have codes? `"VX"`, `"HA"`, and `"B6"`? This information is provided in a separate data frame `airlines`.

```
View(airlines)
```

We see that in `airlines`, `carrier` is the carrier code while `name` is the full name of the airline company. Using this table, we can see that `"VX"`, `"HA"`, and `"B6"` correspond to Virgin America, Hawaiian Airlines, and JetBlue respectively. However, wouldn't it be nice to have all this information in a single data frame instead of two separate data frames? We can do this by "joining" i.e. "merging" the `flights` and `airlines` data frames.

Note that the values in the variable `carrier` in the `flights` data frame match the values in the variable `carrier` in the `airlines` data frame. In this case, we can use the variable `carrier` as a *key variable* to match the rows of the two data frames. Key variables are almost always identification variables that uniquely identify the observational units. This ensures that rows in both data frames are appropriately matched during the join. Hadley and Garrett (Grolemund and Wickham 2016) created the following diagram to help us understand how the different datasets are linked by various key variables:

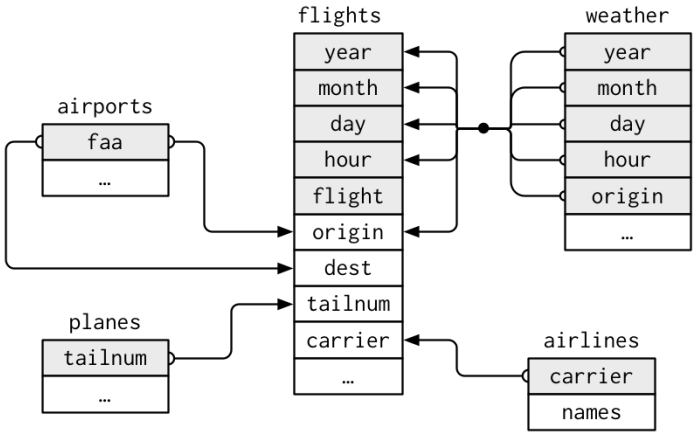


Figure 3.7: Data relationships in nycflights13 from R for Data Science

3.7.1 Matching "key" variable names

In both the `flights` and `airlines` data frames, the key variable we want to join/merge/match the rows of the two data frames by have the same name: `carriers`. We make use of the `inner_join()` function to join the two data frames, where the rows will be matched by the variable `carrier`.

```
flights_joined <- flights %>%
  inner_join(airlines, by = "carrier")

View(flights)
View(flights_joined)
```

Observe that the `flights` and `flights_joined` data frames are identical except that `flights_joined` has an additional variable `name` whose values correspond to the airline company names drawn from the `airlines` data frame.

A visual representation of the `inner_join()` is given below (Grolemund and Wickham 2016). There are other types of joins available (such as `left_join()`, `right_join()`, `outer_join()`, and `anti_join()`), but the `inner_join()` will solve nearly all of the problems you'll encounter in this book.

3.7.2 Different "key" variable names

Say instead you are interested in the destinations of all domestic flights departing NYC in 2013 and ask yourself:

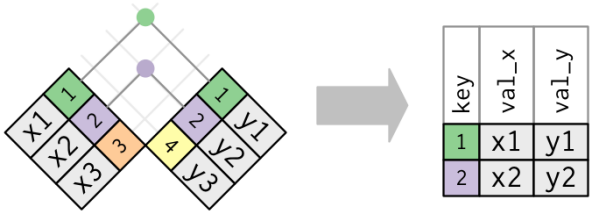


Figure 3.8: Diagram of inner join from R for Data Science

- "What cities are these airports in?"
- "Is `"ORD"` Orlando?"
- "Where is `"FLL"`?

The `airports` data frame contains airport codes:

```
View(airports)
```

However, considering the visual representation (Figure 3.7) of the relations between the datasets `airports` and `flights`, we see that:

- the `airports` data frame the airport code is in the variable `faa`
- the `flights` data frame the airport codes are in the variables `origin` and `dest`

We need to join these two data frames soo that we can identify the destination cities. For example, our `inner_join()` operation will use the `by = c("dest" = "faa")` argument, which allows us to join two data frames where the key variable has a different name:

```
flights_with_airport_names <-  flights %>%
  inner_join(airports, by = c("dest" = "faa"))

View(flights_with_airport_names)
```

Let's construct the sequence of commands that computes the number of flights from NYC to each destination, but also includes information about each destination airport:

```
named_dests <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n()) %>%
  arrange(desc(num_flights)) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  rename(airport_name = name)
```

```
named_dests
```

```
# A tibble: 101 x 9
   dest  num_flights airport_name             lat    lon   alt    tz dst   tzone
   <chr>       <int> <chr>                  <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
 1 ORD         17283 Chicago Ohare Intl      42.0  -87.9   668    -6 A     Amer~
 2 ATL         17215 Hartsfield Jackson At~  33.6  -84.4  1026    -5 A     Amer~
 3 LAX         16174 Los Angeles Intl        33.9 -118.    126    -8 A     Amer~
 4 BOS         15508 General Edward Lawren~  42.4  -71.0    19    -5 A     Amer~
 5 MCO         14082 Orlando Intl            28.4  -81.3    96    -5 A     Amer~
 6 CLT         14064 Charlotte Douglas Intl  35.2  -80.9   748    -5 A     Amer~
 7 SFO         13331 San Francisco Intl      37.6 -122.     13    -8 A     Amer~
 8 FLL         12055 Fort Lauderdale Holly~  26.1  -80.2     9    -5 A     Amer~
 9 MIA         11728 Miami Intl              25.8  -80.3     8    -5 A     Amer~
10 DCA          9705 Ronald Reagan Washing~  38.9  -77.0    15    -5 A     Amer~
# ... with 91 more rows
```

In case you didn't know, `"ORD"` is the airport code of Chicago O'Hare airport and `"FLL"` is the main airport in Fort Lauderdale, Florida, which we can now see in the `airport_name` variable in the resulting `named_dests` data frame.

3.7.3 Multiple "key" variables

Say instead we are in a situation where we need to join by multiple variables. For example, in Figure 3.7 above we see that in order to join the `flights` and `weather` data frames, we need more than one key variable: `year`, `month`, `day`, `hour`, and `origin`. This is because the combination of these 5 variables act to uniquely identify each observational unit in the `weather` data frame: hourly weather recordings at each of the 3 NYC airports.

We achieve this by specifying a vector of key variables to join by using the `c()` function for "combine" or "concatenate" that we saw earlier:

```
flights_weather_joined <- flights %>%
  inner_join(weather, by = c("year", "month", "day", "hour", "origin"))

View(flights_weather_joined)
```

Learning Check 3.14

Looking at Figure 3.7, when joining `flights` and `weather` (or, in other words, matching the hourly weather values with each flight), why do we need to join by all of `year`, `month`,

`day`, `hour`, and `origin`, and not just `hour`?

Learning Check 3.15

What surprises you about the top 10 destinations from NYC in 2013?

3.7.4 Normal forms

The data frames included in the `nycflights13` package are in a form that minimizes redundancy of data. For example, the `flights` data frame only saves the `carrier` code of the airline company; it does not include the actual name of the airline. For example the first row of `flights` has `carrier` equal to `UA`, but does it does not include the airline name "United Air Lines Inc." The names of the airline companies are included in the `name` variable of the `airlines` data frame. In order to have the airline company name included in `flights`, we could join these two data frames as follows:

```
joined_flights <- flights %>%
  inner_join(airlines, by = "carrier")

View(joined_flights)
```

We are capable of performing this join because each of the data frames have *keys* in common to relate one to another: the `carrier` variable in both the `flights` and `airlines` data frames. The *key* variable(s) that we join are often *identification variables* we mentioned previously.

This is an important property of what's known as **normal forms** of data. The process of decomposing data frames into less redundant tables without losing information is called **normalization**. More information is available on Wikipedia.

Learning Check 3.16

What are some advantages of data in normal forms? What are some disadvantages?

3.8 Other verbs

Here are some other useful data wrangling verbs that might come in handy:

- `select()` only a subset of variables/columns
- `rename()` variables/columns to have new names
- Return only the `top_n()` values of a variable

3.8.1 `select()` variables

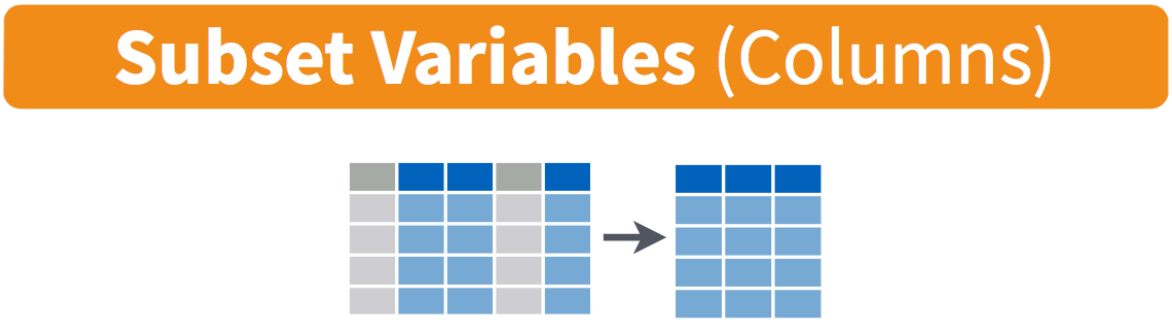


Figure 3.9: Select diagram from Data Wrangling with dplyr and tidyr cheatsheet

We've seen that the `flights` data frame in the `nycflights13` package contains 19 different variables. You can identify the names of these 19 variables by running the `glimpse()` function from the `dplyr` package:

```
glimpse(flights)
```

However, say you only need two of these variables, say `carrier` and `flight`. You can `select()` these two variables:

```
flights %>%
  select(carrier, flight)
```

This function makes exploring data frames with a very large number of variables easier for humans to process by restricting consideration to only those we care about, like our example with `carrier` and `flight` above. This might make viewing the dataset using the `View()` spreadsheet viewer more digestible. However, as far as the computer is concerned, it doesn't care how many additional variables are in the data frame in question, so long as `carrier` and `flight` are included.

Let's say instead you want to drop i.e deselect certain variables. For example, take the variable `year` in the `flights` data frame. This variable isn't quite a "variable" in the sense that all the values are `2013` i.e. it doesn't change. Say you want to remove the `year` variable from the data frame; we can deselect `year` by using the `-` sign:

```
flights_no_year <- flights %>%
  select(-year)

glimpse(flights_no_year)
```

Another way of selecting columns/variables is by specifying a range of columns:

```
flight_arr_times <- flights %>%
  select(month:day, arr_time:sched_arr_time)

flight_arr_times
```

The `select()` function can also be used to reorder columns in combination with the `everything()` helper function. Let's suppose we'd like the `hour`, `minute`, and `time_hour` variables, which appear at the end of the `flights` dataset, to appear immediately after the `year`, `month`, and `day` variables while keeping the rest of the variables. In the code below `everything()` picks up all remaining variables.

```
flights_reorder <- flights %>%
  select(year, month, day, hour, minute, time_hour, everything())

glimpse(flights_reorder)
```

Lastly, the helper functions `starts_with()`, `ends_with()`, and `contains()` can be used to select variables/column that match those conditions. For example:

```
flights_begin_a <- flights %>%
  select(starts_with("a"))

flights_begin_a
```

```
flights_delays <- flights %>%
  select(ends_with("delay"))

flights_delays
```

```
flights_time <- flights %>%
  select(contains("time"))

flights_time
```

3.8.2 `rename()` variables

Another useful function is `rename()`, which as you may have guessed renames one column to another name. Suppose we want `dep_time` and `arr_time` to be `departure_time` and

`arrival_time` instead in the `flights_time` data frame:

```
flights_time <- flights %>%
  select(contains("time")) %>%
  rename(departure_time = dep_time,
         arrival_time = arr_time)

glimpse(flights_time)
```

Note that in this case we used a single `=` sign within the `rename()`, for example `departure_time = dep_time`. This is because we are not testing for equality like we would using `==`, but instead we want to assign a new variable `departure_time` to have the same values as `dep_time` and then delete the variable `dep_time`. It's easy to forget if the new name comes before or after the equals sign. I usually remember this as "New Before, Old After" or NBOA.

3.8.3 `slice()` data by a variable

We can return observations with maximum or minimum values of a variable using the `slice_mac()` or `slice_min()`. For example, we can get the observations the top 10 destination airports using the example from Section 3.7.2. Observe that we set the number of values to return to `n = 10` and `wt = num_flights` to indicate that we want the rows of corresponding to the top 10 values of `num_flights`. See the help file for `top_n()` by running `?top_n` for more information.

```
named_dests %>%
  slice_max(n = 10, order_by =  num_flights)

named_dests %>%
  slice_min(n = 10, order_by =  num_flights)
```

Learning Check 3.17

What are some ways to select all three of the `dest`, `air_time`, and `distance` variables from `flights`? Give the code showing how to do this in at least three different ways.

Learning Check 3.18

How could one use `starts_with`, `ends_with`, and `contains` to select columns from the `flights` data frame? Provide three different examples in total: one for `starts_with`, one for `ends_with`, and one for `contains`.

Learning Check 3.19

Why might we want to use the `select` function on a data frame?

Learning Check 3.20

Create a new data frame that shows the top 5 airports with the largest arrival delays from NYC in 2013.

3.9 Conclusion

3.9.1 Summary table

Let's recap our data wrangling verbs in Table 3.2. Using these verbs and the pipe `%>%` operator from @sec-piping, you'll be able to write easily legible code to perform almost all the data wrangling necessary for the rest of this book.

Table 3.2: Summary of data wrangling verbs

Verb	Data wrangling operation
`filter()`	Pick out a subset of rows
`summarize()`	Summarize many values to one using a summary statistic function like `mean()`, `median()`, etc.
`group_by()`	Add grouping structure to rows in data frame. Note this does not change values in data frame, rather only the meta-data
`mutate()`	Create new variables by mutating existing ones
`arrange()`	Arrange rows of a data variable in ascending (default) or `desc`ending order
`inner_join()`	Join/merge two data frames, matching rows by a key variable

Learning Check 3.21

Let's now put your newly acquired data wrangling skills to the test!
An airline industry measure of a passenger airline's capacity is the available seat miles, which is equal to the number of seats available multiplied by the number of miles or kilometers flown summed over all flights. So for example say an airline had 2 flights using a plane with 10 seats that flew 500 miles and 3 flights using a plane with 20 seats that flew 1000 miles, the available seat miles would be $2 \times 10 \times 500 + 3 \times 20 \times 1000 =$ 70,000 seat miles.
Using the datasets included in the `nycflights13` package, compute the available seat miles for each airline sorted in descending order. After completing all the necessary data

wrangling steps, the resulting data frame should have 16 rows (one for each airline) and 2 columns (airline name and available seat miles). Here are some hints:

1. **Crucial**: Unless you are very confident in what you are doing, it is worthwhile to not start coding right away, but rather first sketch out on paper all the necessary data wrangling steps not using exact code, but rather high-level *pseudocode* that is informal yet detailed enough to articulate what you are doing. This way you won't confuse *what* you are trying to do (the algorithm) with *how* you are going to do it (writing `dplyr` code).
2. Take a close look at all the datasets using the `View()` function: `flights`, `weather`, `planes`, `airports`, and `airlines` to identify which variables are necessary to compute available seat miles.
3. Figure 3.7 above showing how the various datasets can be joined will also be useful.
4. Consider the data wrangling verbs in Table 3.2) as your toolbox!

3.9.2 Additional resources

If you want to further unlock the power of the `dplyr` package for data wrangling, we suggest you that you check out RStudio's "Data Transformation with dplyr" cheatsheet. This cheatsheet summarizes much more than what we've discussed in this chapter, in particular more-intermediate level and advanced data wrangling functions, while providing quick and easy to read visual descriptions.

You can access this cheatsheet by going to the RStudio Menu Bar -> Help -> Cheatsheets -> "Data Transformation with dplyr":

On top of data wrangling verbs and examples we presented in this section, if you'd like to see more examples of using the `dplyr` package for data wrangling check out Chapter 5 of Garrett Grolemund and Hadley Wickham's and Garrett's book (Grolemund and Wickham 2016).

3.9.3 What's to come?

So far in this book, we've explored, visualized, and wrangled data saved in data frames that are in spreadsheet-type format: rectangular with a certain number of rows corresponding to observations and a certain number of columns corresponding to variables describing the observations.

We'll see in Chapter 4 that there are actually two ways to represent data in spreadsheet-type rectangular format: 1) "wide" format and 2) "tall/narrow" format also known in R circles as "tidy" format. While the distinction between "tidy" and non-"tidy" formatted data is very

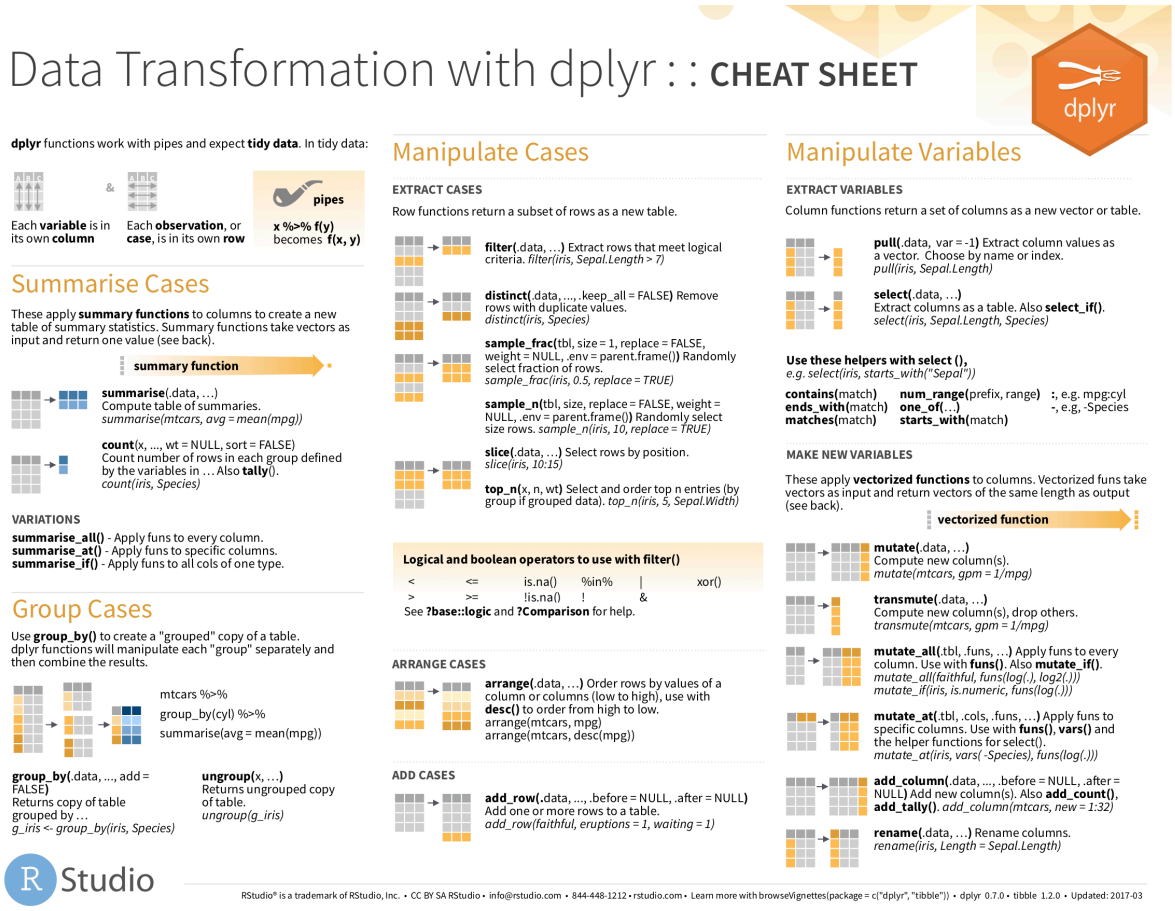


Figure 3.10: Data Transformation with dplyr cheatsheat

subtle, it has very important implications for whether or not we can use the `ggplot2` package for data visualization and the `dplyr` package for data wrangling.

Furthermore, we've only explored, visualized, and wrangled data saved within R packages. What if you have spreadsheet data saved in a Microsoft Excel, Google Sheets, or "Comma-Separated Values" (CSV) file that you would like to analyze? In Chapter 4, we'll show you how to import this data into R using the `readr` package.

4 Data Importing & "Tidy Data"

In Subsection 1.2.2 we introduced the concept of a data frame: a rectangular spreadsheet-like representation of data in R where the rows correspond to observations and the columns correspond to variables describing each observation. In Section 1.4, we started exploring our first data frame: the `flights` data frame included in the `nycflights13` package. In Chapter 2 we created visualizations based on the data included in `flights` and other data frames such as `weather`. In Chapter 3, we learned how to wrangle data, in other words take existing data frames and transform/ modify them to suit our analysis goals.

In this final chapter of the "Data Science via the tidyverse" portion of the book, we extend some of these ideas by discussing a type of data formatting called "tidy" data. You will see that having data stored in "tidy" format is about more than what the colloquial definition of the term "tidy" might suggest: having your data "neatly organized." Instead, we define the term "tidy" in a more rigorous fashion, outlining a set of rules by which data can be stored, and the implications of these rules for analyses.

Although knowledge of this type of data formatting was not necessary for our treatment of data visualization in Chapter 2 and data wrangling in Chapter 3 since all the data was already in "tidy" format, we'll now see this format is actually essential to using the tools we covered in these two chapters. Furthermore, it will also be useful for all subsequent chapters in this book when we cover regression and statistical inference. First however, we'll show you how to import spreadsheet data for use in R.

Packages Needed

Let's load all the packages needed for this chapter (this assumes you've already installed them). If needed, read Section @ref(packages) for information on how to install and load R packages.

```
library(dplyr)
library(ggplot2)
library(readr)
library(tidyr)
library(nycflights13)
library(fivethirtyeight)
```

4.1 Importing data

Up to this point, we've almost entirely used data stored inside of an R package. Say instead you have your own data saved on your computer or somewhere online? How can you analyze this data in R? Spreadsheet data is often saved in one of the following formats:

- A *Comma Separated Values* `.csv` file. You can think of a `.csv` file as a bare-bones spreadsheet where:
  - Each line in the file corresponds to one row of data/one observation.
  - Values for each line are separated with commas. In other words, the values of different variables are separated by commas.
  - The first line is often, but not always, a *header* row indicating the names of the columns/variables.
- An Excel `.xlsx` file. This format is based on Microsoft's proprietary Excel software. As opposed to a bare-bones `.csv` files, `.xlsx` Excel files contain a lot of meta-data, or put more simply, data about the data. (Recall we saw a previous example of meta-data in Section 3.4 when adding "group structure" meta-data to a data frame by using the `group_by()` verb.) Some examples of spreadsheet meta-data include the use of bold and italic fonts, colored cells, different column widths, and formula macros.
- A Google Sheets file, which is a "cloud" or online-based way to work with a spreadsheet. Google Sheets allows you to download your data in both comma separated values `.csv` and Excel `.xlsx` formats however: go to the Google Sheets menu bar -> File -> Download as -> Select "Microsoft Excel" or "Comma-separated values."

We'll cover two methods for importing `.csv` and `.xlsx` spreadsheet data in R: one using the R console and the other using RStudio's graphical user interface, abbreviated a GUI.

4.1.1 Using the console

First, let's import a Comma Separated Values `.csv` file of data directly off the internet. The `.csv` file `dem_score.csv` accessible at https://moderndive.com/data/dem_score.csv contains ratings of the level of democracy in different countries spanning 1952 to 1992. Let's use the `read_csv()` function from the `readr` package to read it off the web, import it into R, and save it in a data frame called `dem_score`

```
library(readr)
dem_score <- read_csv("https://moderndive.com/data/dem_score.csv")
dem_score
```

```
# A tibble: 96 x 10
   country    `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
   <chr>       <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
 1 Albania        -9     -9     -9     -9     -9     -9     -9     -9      5
 2 Argentina      -9     -1     -1     -9     -9     -9     -8      8      7
 3 Armenia        -9     -7     -7     -7     -7     -7     -7     -7      7
 4 Australia      10     10     10     10     10     10     10     10     10
 5 Austria        10     10     10     10     10     10     10     10     10
 6 Azerbaijan     -9     -7     -7     -7     -7     -7     -7     -7      1
 7 Belarus        -9     -7     -7     -7     -7     -7     -7     -7      7
 8 Belgium        10     10     10     10     10     10     10     10     10
 9 Bhutan        -10    -10    -10    -10    -10    -10    -10    -10    -10
10 Bolivia        -4     -3     -3     -4     -7     -7      8      9      9
# ... with 86 more rows
```

In this `dem_score` data frame, the minimum value of `-10` corresponds to a highly autocratic nation whereas a value of `10` corresponds to a highly democratic nation. We'll revisit the `dem_score` data frame in a case study in the upcoming Section 4.3.

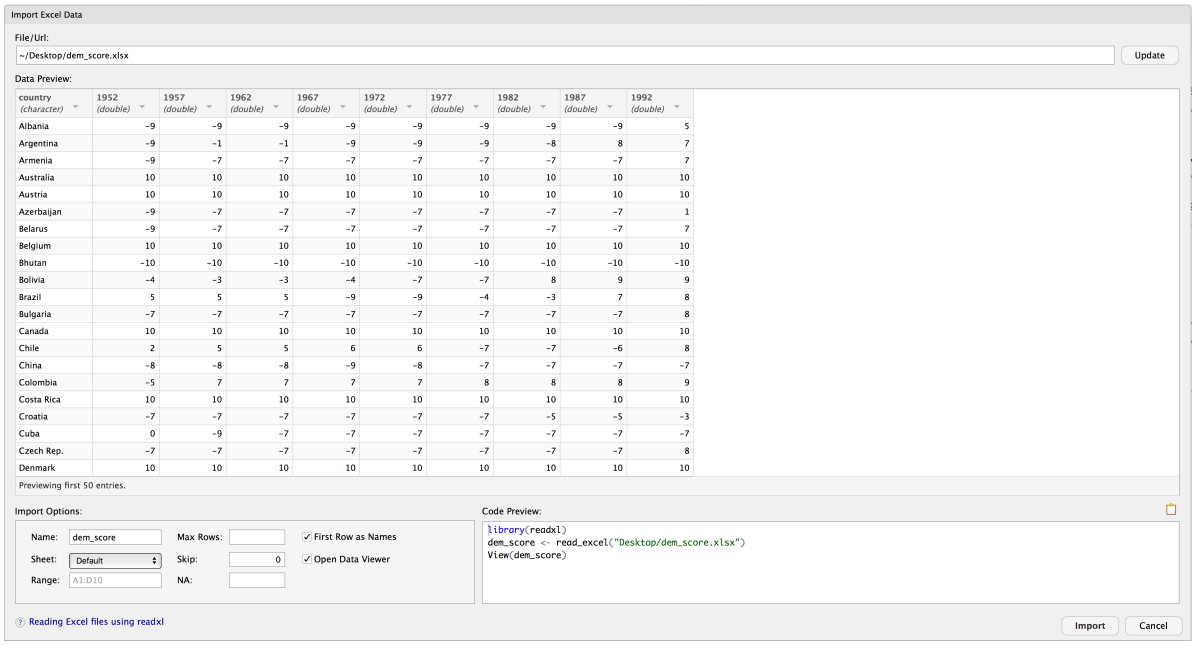
Note that the `read_csv()` function included in the `readr` package is different than the `read.csv()` function that comes installed with R by default. While the difference in the names might seem near meaningless (an `_` instead of a `.`), the `read_csv()` function is in our opinion easier to use since it can more easily read data off the web and generally imports data at a much faster speed.

4.1.2 Using RStudio's interface

Let's read in the exact same data saved in Excel format, but this time via RStudio's graphical interface instead of via the R console. First download the Excel file `dem_score.xlsx` by clicking here, then

1. Go to the Files panel of RStudio.
2. Navigate to the directory i.e. folder on your computer where the downloaded `dem_score.xlsx` Excel file is saved.
3. Click on `dem_score.xlsx`.
4. Click "Import Dataset…"

At this point you should see an image like this:



After clicking on the "Import" button on the bottom right RStudio, RStudio will save this spreadsheet's data in a data frame called `dem_score` and display its contents in the spreadsheet viewer. Furthermore, note in the bottom right of the above image there exists a "Code Preview": you can copy and paste this code to reload your data again later automatically instead of repeating the above manual point-and-click process.

4.2 Tidy data

Let's now switch gears and learn about the concept of "tidy" data format by starting with a motivating example. Let's consider the `drinks` data frame included in the `fivethirtyeight` data. Run the following:

```
drinks
```

```
# A tibble: 193 x 5
   country       beer_servings spirit_servings wine_servings total_litres_~1
   <chr>                 <int>           <int>         <int>            <dbl>
 1 Afghanistan               0               0             0              0
 2 Albania                  89             132            54              4.9
 3 Algeria                  25               0            14              0.7
 4 Andorra                 245             138           312             12.4
 5 Angola                  217              57            45              5.9
```

```
 6 Antigua & Barbuda           102             128            45             4.9
 7 Argentina                   193              25           221             8.3
 8 Armenia                      21             179            11             3.8
 9 Australia                   261              72           212            10.4
10 Austria                     279              75           191             9.7
# ... with 183 more rows, and abbreviated variable name
#   1: total_litres_of_pure_alcohol
```

After reading the help file by running `?drinks`, we see that `drinks` is a data frame containing results from a survey of the average number of servings of beer, spirits, and wine consumed for 193 countries. This data was originally reported on the data journalism website FiveThirtyEight.com in Mona Chalabi's article "Dear Mona Followup: Where Do People Drink The Most Beer, Wine And Spirits?"

Let's apply some of the data wrangling verbs we learned in Chapter 3 on the `drinks` data frame. Let's

1. `filter()` the `drinks` data frame to only consider 4 countries (the United States, China, Italy, and Saudi Arabia) then
2. `select()` all columns except `total_litres_of_pure_alcohol` by using `-` sign, then
3. `rename()` the variables `beer_servings`, `spirit_servings`, and `wine_servings` to `beer`, `spirit`, and `wine` respectively

and save the resulting data frame in `drinks_smaller`.

```
drinks_smaller <- drinks %>%
  filter(country %in% c("USA", "China", "Italy", "Saudi Arabia")) %>%
  select(-total_litres_of_pure_alcohol) %>%
  rename(beer = beer_servings, spirit = spirit_servings, wine = wine_servings)

drinks_smaller
```

```
# A tibble: 4 x 4
  country       beer spirit  wine
  <chr>        <int>  <int> <int>
1 China           79    192     8
2 Italy           85     42   237
3 Saudi Arabia     0      5     0
4 USA            249    158    84
```

Using the `drinks_smaller` data frame, how would we create the side-by-side AKA dodged barplot in Figure 4.1? Recall we saw barplots displaying two categorical variables in Section 2.8.3.

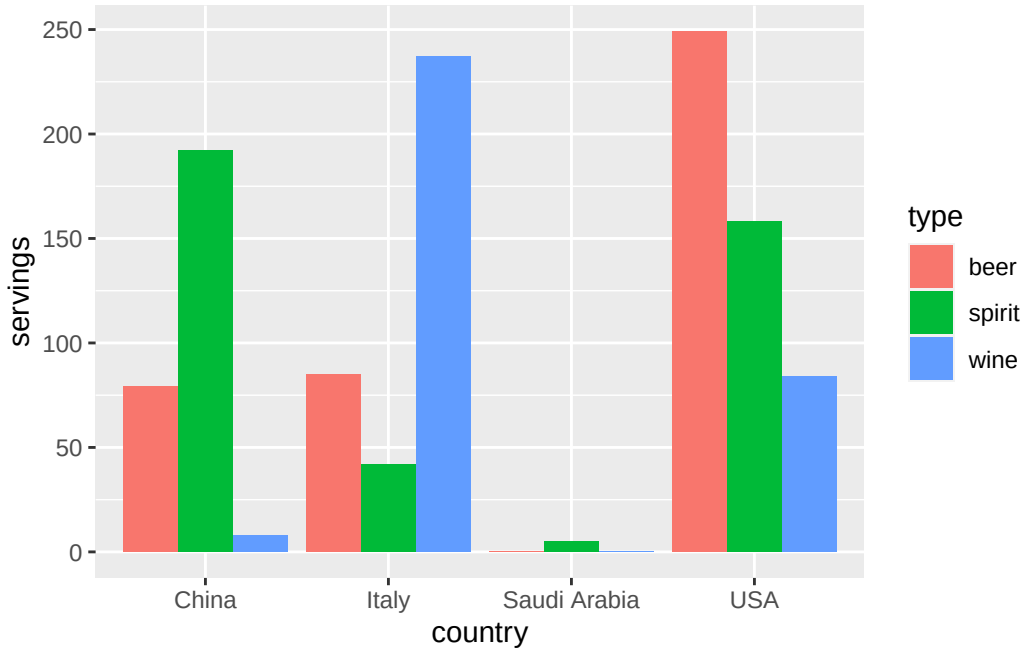


Figure 4.1: Alcohol consumption in 4 countries

Let's break down the Grammar of Graphics:

1. The categorical variable `country` with four levels (China, Italy, Saudi Arabia, USA) would have to be mapped to the `x`-position of the bars.
2. The numerical variable `servings` would have to be mapped to the `y`-position of the bars, in other words the height of the bars.
3. The categorical variable `type` with three levels (beer, spirit, wine) who have to be mapped to the `fill` color of the bars.

Observe however that `drinks_smaller` has *three separate variables* for `beer`, `spirit`, and `wine`, whereas in order to recreate the side-by-side AKA dodged barplot in Figure 4.1 we would need a *single variable* `type` with three possible values: `beer`, `spirit`, and `wine`, which we would then map to the `fill` aesthetic. In other words, for us to be able to create the barplot in Figure 4.1, our data frame would have to look like this:

```
drinks_smaller_tidy
```

```
# A tibble: 12 x 3
   country      type   servings
   <chr>        <chr>     <int>
 1 China        beer         79
 2 China        spirit      192
 3 China        wine          8
 4 Italy        beer         85
 5 Italy        spirit       42
 6 Italy        wine        237
 7 Saudi Arabia beer          0
 8 Saudi Arabia spirit        5
 9 Saudi Arabia wine          0
10 USA          beer        249
11 USA          spirit      158
12 USA          wine         84
```

Let's compare the `drinks_smaller_tidy` with the `drinks_smaller` data frame from earlier:

```
drinks_smaller
```

```
# A tibble: 4 x 4
  country       beer spirit  wine
  <chr>        <int>  <int> <int>
1 China           79    192     8
2 Italy           85     42   237
3 Saudi Arabia     0      5     0
4 USA            249    158    84
```

Observe that while `drinks_smaller` and `drinks_smaller_tidy` are both rectangular in shape and contain the same 12 numerical values (3 alcohol types $\times$ 4 countries), they are formatted differently. `drinks_smaller` is formatted in what's known as "wide" format, whereas `drinks_smaller_tidy` is formatted in what's known as "long/narrow". In the context of using R, long/narrow format is also known as "tidy" format. Furthermore, in order to use the `ggplot2` and `dplyr` packages for data visualization and data wrangling, your input data frames *must* be in "tidy" format. So all non-"tidy" data must be converted to "tidy" format first.

Before we show you how to convert non-"tidy" data frames like `drinks_smaller` to "tidy" data frames like `drinks_smaller_tidy`, let's go over the explicit definition of "tidy" data.

4.2.1 Definition of "tidy" data

You have surely heard the word "tidy" in your life:

- "Tidy up your room!"
- "Please write your homework in a tidy way so that it is easier to grade and to provide feedback."
- Marie Kondo's best-selling book *The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing* and Netflix TV series *Tidying Up with Marie Kondo*.
- "I am not by any stretch of the imagination a tidy person, and the piles of unread books on the coffee table and by my bed have a plaintive, pleading quality to me - 'Read me, please!' " - Linda Grant

What does it mean for your data to be "tidy"? While "tidy" has a clear English meaning of "organized", "tidy" in the context of data science using R means that your data follows a standardized format. We will follow Hadley Wickham's definition of *tidy data* here (Wickham 2014):

> A dataset is a collection of values, usually either numbers (if quantitative) or strings AKA text data (if qualitative). Values are organised in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a city) across attributes.
>
> Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In *tidy data*:
>
> 1. Each variable forms a column.
> 2. Each observation forms a row.
> 3. Each type of observational unit forms a table.

For example, say you have the following table of stock prices in Table 4.1:

Table 4.1: Stock Prices (Non-Tidy Format)

Date	Boeing Stock Price	Amazon Stock Price	Google Stock Price
2009-01-01	$173.55	$174.90	$174.34
2009-01-02	$172.61	$171.42	$170.04

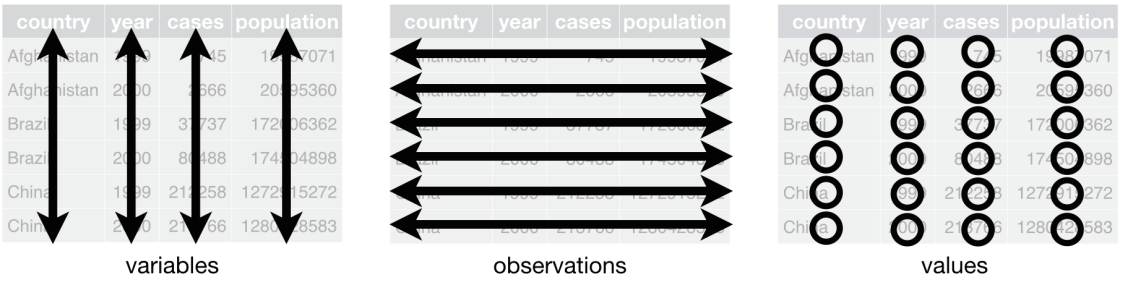


Figure 4.2: Tidy data graphic from R for Data Science

Although the data are neatly organized in a rectangular spreadsheet-type format, they are not in tidy format because while there are three variables corresponding to three unique pieces of information (Date, Stock Name, and Stock Price), there are not three columns. In "tidy" data format each variable should be its own column, as shown in Table 4.2). Notice that both tables present the same information, but in different formats.

Table 4.2: Stock Prices (Tidy Format)

Date	Stock Name	Stock Price
2009-01-01	Boeing	$173.55
2009-01-01	Amazon	$174.90
2009-01-01	Google	$174.34
2009-01-02	Boeing	$172.61
2009-01-02	Amazon	$171.42
2009-01-02	Google	$170.04

Now we have the requisite three columns Date, Stock Name, and Stock Price. On the other hand, consider the data in Table @ref(tab:tidy-stocks-2).

Table 4.3: Date, Boeing Price, Weather Data

Date	Boeing Price	Weather
2009-01-01	$173.55	Sunny
2009-01-02	$172.61	Overcast

In this case, even though the variable "Boeing Price" occurs just like in our non-"tidy" data in Table 4.1), the data *is* "tidy" since there are three variables corresponding to three unique pieces of information: Date, Boeing stock price, and the weather that particular day.

Learning Check 4.1

What are common characteristics of "tidy" data frames?

Learning Check 4.2

What makes "tidy" data frames useful for organizing data?

4.2.2 Converting to "tidy" data

In this book so far, you've only seen data frames that were already in "tidy" format. Furthermore for the rest of this book, you'll mostly only see data frames that are already in "tidy" format as well. This is not always the case however with data in the wild. If your original data frame is in wide i.e. non-"tidy" format and you would like to use the `ggplot2` package for data visualization or the `dplyr` package for data wrangling, you will first have to convert it to "tidy" format using the `pivot_longer()` function in the `tidyr` package (Wickham and Girlich 2022).

Going back to our `drinks_smaller` data frame from earlier:

```
drinks_smaller
```

```
# A tibble: 4 x 4
  country       beer spirit  wine
  <chr>        <int>  <int> <int>
1 China           79    192     8
2 Italy           85     42   237
3 Saudi Arabia     0      5     0
4 USA            249    158    84
```

We convert it to "tidy" format by using the `pivot_longer()` function from the `tidyr` package as follows:

```
# tidy drinks_smaller
drinks_smaller_tidy <- drinks_smaller %>%
  pivot_longer(
    cols = -country,
    names_to = "type",
    values_to = "servings"
  )
```

```
# print
drinks_smaller_tidy
```

```
# A tibble: 12 x 3
   country      type   servings
   <chr>        <chr>     <int>
 1 China        beer         79
 2 China        spirit      192
 3 China        wine          8
 4 Italy        beer         85
 5 Italy        spirit       42
 6 Italy        wine        237
 7 Saudi Arabia beer          0
 8 Saudi Arabia spirit        5
 9 Saudi Arabia wine          0
10 USA          beer        249
11 USA          spirit      158
12 USA          wine         84
```

We set the arguments to `pivot_longer()` as follows:

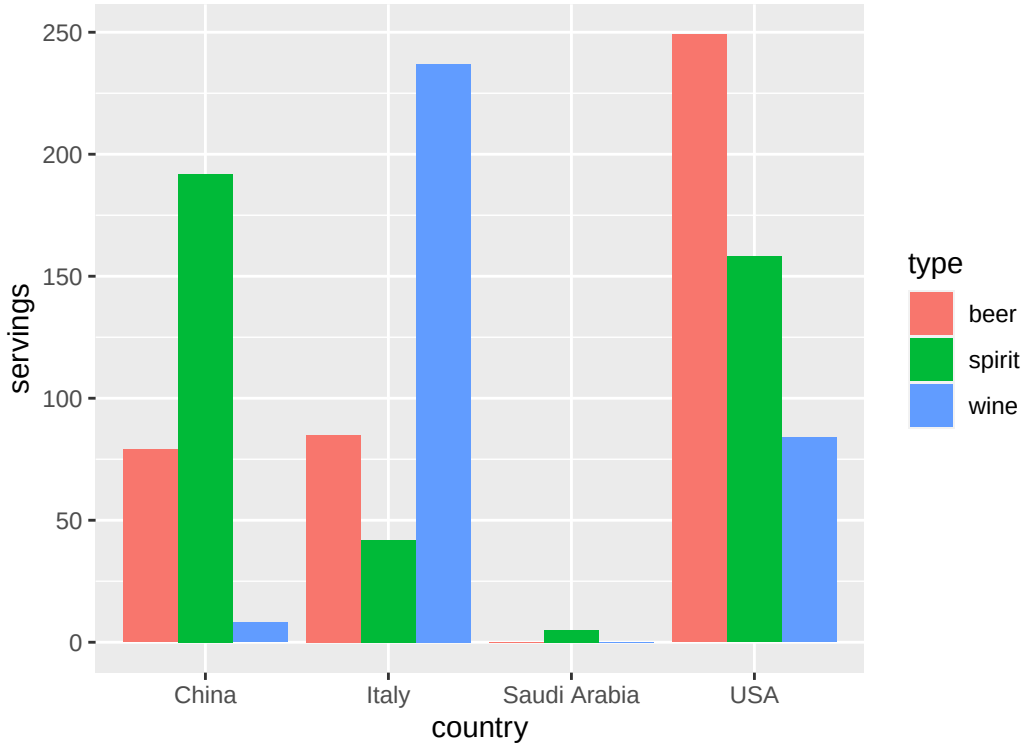
1. The first argument, `cols`, are the columns you either want to or don't want to tidy. Observe how we set this to `-country` indicating that we don't want to tidy the `country` variable in `drinks_smaller` which leaves `beer`, `spirit`, and `wine` to be tidied.
2. `names_to` is the name of the column/variable in the new "tidy" frame that contains the column names of the original data frame that you want to tidy. Observe how we set `names_to = "type"` and in the resulting `drinks_smaller_tidy` the column `type` contains the three types of alcohol `beer`, `spirit`, and `wine`.
3. `values_to` is the name of the column/variable in the "tidy" frame that contains the rows and columns of values in the original data frame you want to tidy. Observe how we set `values_to = "servings"` and in the resulting `drinks_smaller_tidy` the column `servings` contains the $4 \times 3 = 12$ numerical values.

The first argument, `cols`, is a little nuanced, so let's consider another example. Note the code below is very similar, but now the first argument species which columns we'd want to tidy `c(beer, spirit, wine)`, instead of the columns we don't want to tidy `-country`. Note the use of `c()` to create a vector of the columns in `drinks_smaller` that we'd like to tidy. If you run the code below, you'll see that the result is as `drinks_smaller_tidy`.

```
# tidy drinks_smaller
drinks_smaller %>%
  pivot_longer(
    cols = c(beer, spirit, wine),
    names_to = "type",
    values_to = "servings"
    )
```

With our `drinks_smaller_tidy` "tidy" format data frame, we can now produce a side-by-side AKA dodged barplot using `geom_col()` and not `geom_bar()`, since we would like to map the `servings` variable to the `y`-aesthetic of the bars.

```
ggplot(drinks_smaller_tidy, aes(x=country, y=servings, fill=type)) +
  geom_col(position = "dodge")
```



Converting "wide" format data to "tidy" format often confuses new R users. The only way to learn to get comfortable with the `pivot_longer()` function is with practice, practice, and more practice. For example, see the examples in the bottom of the help file for `pivot_longer()` by running `?pivot_longer`. We'll show another example of using `pivot_longer()` to convert a

"wide" formatted data frame to "tidy" format in Section 4.3. For other examples of converting a dataset into "tidy" format, check out the different functions available for data tidying and a case study using data from the World Health Organization in R for Data Science (Grolemund and Wickham 2016).

Learning Check 4.3

Take a look the `airline_safety` data frame included in the `fivethirtyeight` data. Run the following:

```
airline_safety
```

After reading the help file by running `?airline_safety`, we see that `airline_safety` is a data frame containing information on different airlines companies' safety records. This data was originally reported on the data journalism website FiveThirtyEight.com in Nate Silver's article "Should Travelers Avoid Flying Airlines That Have Had Crashes in the Past?". Let's ignore the `incl_reg_subsidiaries` and `avail_seat_km_per_week` variables for simplicity:

```
airline_safety_smaller <- airline_safety %>%
  select(-c(incl_reg_subsidiaries, avail_seat_km_per_week))

airline_safety_smaller
```

```
# A tibble: 56 x 7
   airline               incidents_85_99 fatal~1 fatal~2 incid~3 fatal~4 fatal~5
   <chr>                           <int>   <int>   <int>   <int>   <int>   <int>
 1 Aer Lingus                          2       0       0       0       0       0
 2 Aeroflot                           76      14     128       6       1      88
 3 Aerolineas Argentinas               6       0       0       1       0       0
 4 Aeromexico                          3       1      64       5       0       0
 5 Air Canada                          2       0       0       2       0       0
 6 Air France                         14       4      79       6       2     337
 7 Air India                           2       1     329       4       1     158
 8 Air New Zealand                     3       0       0       5       1       7
 9 Alaska Airlines                     5       0       0       5       1      88
10 Alitalia                            7       2      50       4       0       0
# ... with 46 more rows, and abbreviated variable names
#   1: fatal_accidents_85_99, 2: fatalities_85_99, 3: incidents_00_14,
#   4: fatal_accidents_00_14, 5: fatalities_00_14
```

This data frame is not in "tidy" format. How would you convert this data frame to be

in "tidy" format, in particular so that it has a variable `incident_type_years` indicating the incident type/year and a variable `count` of the counts?

4.2.3 `nycflights13` package

Recall the `nycflights13` package with data about all domestic flights departing from New York City in 2013 that we introduced in Section 1.4 and used extensively in Chapter 2 on data visualization and Chapter 3 on data wrangling. Let's revisit the `flights` data frame by running `View(flights)`. We saw that `flights` has a rectangular shape with each of its 336,776 rows corresponding to a flight and each of its 19 columns corresponding to different characteristics/measurements of each flight. This matches exactly with our definition of "tidy" data from above.

1. Each variable forms a column.
2. Each observation forms a row.

But what about the third property of "tidy" data?

3. Each type of observational unit forms a table.

Recall that we also saw in Section 1.4.3 that the observational unit for the `flights` data frame is an individual flight. In other words, the rows of the `flights` data frame refer to characteristics/measurements of individual flights. Also included in the `nycflights13` package are other data frames with their rows representing different observational units (Wickham 2021):

- `airlines`: translation between two letter IATA carrier codes and names (16 in total). i.e. the observational unit is an airline company.
- `planes`: construction information about each of 3,322 planes used. i.e. the observational unit is an aircraft.
- `weather`: hourly meteorological data (about 8705 observations) for each of the three NYC airports. i.e. the observational unit is an hourly measurement.
- `airports`: airport names and locations. i.e. the observational unit is an airport.

The organization of the information into these five data frames follow the third "tidy" data property: observations corresponding to the same observational unit should be saved in the same table i.e. data frame. You could think of this property as the old English expression: "birds of a feather flock together."

4.3 Case study: Democracy in Guatemala

In this section, we'll show you another example of how to convert a data frame that isn't in "tidy" format i.e. "wide" format, to a data frame that is in "tidy" format i.e. "long/narrow" format. We'll do this using the `pivot_longer()` function from the `tidyr` package again. Furthermore, we'll make use of some of the `ggplot2` data visualization and `dplyr` data wrangling tools you learned in Chapters Chapter 2 and Chapter 3.

Let's use the `dem_score` data frame we imported in Section 4.1, but focus on only data corresponding to Guatemala.

```
guat_dem <- dem_score %>%
  filter(country == "Guatemala")

guat_dem
```

```
# A tibble: 1 x 10
  country   `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
  <chr>      <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 Guatemala      2     -6     -5      3      1     -3     -7      3      3
```

Now let's produce a *time-series plot* showing how the democracy scores have changed over the 40 years from 1952 to 1992 for Guatemala. Recall that we saw time-series plot in Section 2.4 on creating linegraphs using `geom_line()`. Let's lay out the Grammar of Graphics we saw in Section 2.1.

First we know we need to set `data = guat_dem` and use a `geom_line()` layer, but what is the aesthetic mapping of variables. We'd like to see how the democracy score has changed over the years, so we need to map:

- `year` to the x-position aesthetic and
- `democracy_score` to the y-position aesthetic

Now we are stuck in a predicament, much like with our `drinks_smaller` example in sec-tidy-data-ex. We see that we have a variable named `country`, but its only value is `"Guatemala"`. We have other variables denoted by different year values. Unfortunately, the `guat_dem` data frame is not "tidy" and hence is not in the appropriate format to apply the Grammar of Graphics and thus we cannot use the `ggplot2` package. We need to take the values of the columns corresponding to years in `guat_dem` and convert them into a new "key" variable called `year`. Furthermore, we'd like to take the democracy scores on the inside of the table and turn them into a new "value" variable called `democracy_score`. Our resulting data frame will thus have three columns: `country`, `year`, and `democracy_score`.

Recall that the `pivot_longer()` function in the `tidyr` package can complete this task for us:

```
guat_dem_tidy <- guat_dem %>%
  pivot_longer(
    cols = -country,
    names_to = "year",
    values_to = "democracy_score"
  )

guat_dem_tidy
```

```
# A tibble: 9 x 3
  country   year  democracy_score
  <chr>     <chr>           <dbl>
1 Guatemala 1952                2
2 Guatemala 1957               -6
3 Guatemala 1962               -5
4 Guatemala 1967                3
5 Guatemala 1972                1
6 Guatemala 1977               -3
7 Guatemala 1982               -7
8 Guatemala 1987                3
9 Guatemala 1992                3
```

We set the arguments to `pivot_longer()` as follows:

1. The first argument, `cols`, indicatesthe columns you either want to or don't want to tidy. Observe how we set this to `-country` indicating that we don't want to tidy the `country` variable in `guat_dem` which leaves `1952` through `1992` to be tidied.
2. `names_to` is the name of the column/variable in the new "tidy" frame that contains the column names of the original data frame that you want to tidy. Observe how we set `names_to = "year"` and in the resulting `guat_dem_tidy` the column `year` contains the years where the Guatemala's democracy score were measured.
3. `values_to` is the name of the column/variable in the "tidy" frame that contains the rows and columns of values in the original data frame you want to tidy. Observe how we set `values_to = "democracy_score"` and in the resulting `guat_dem_tidy` the column `democracy_score` contains the $1 \times 9 = 9$ democracy scores.

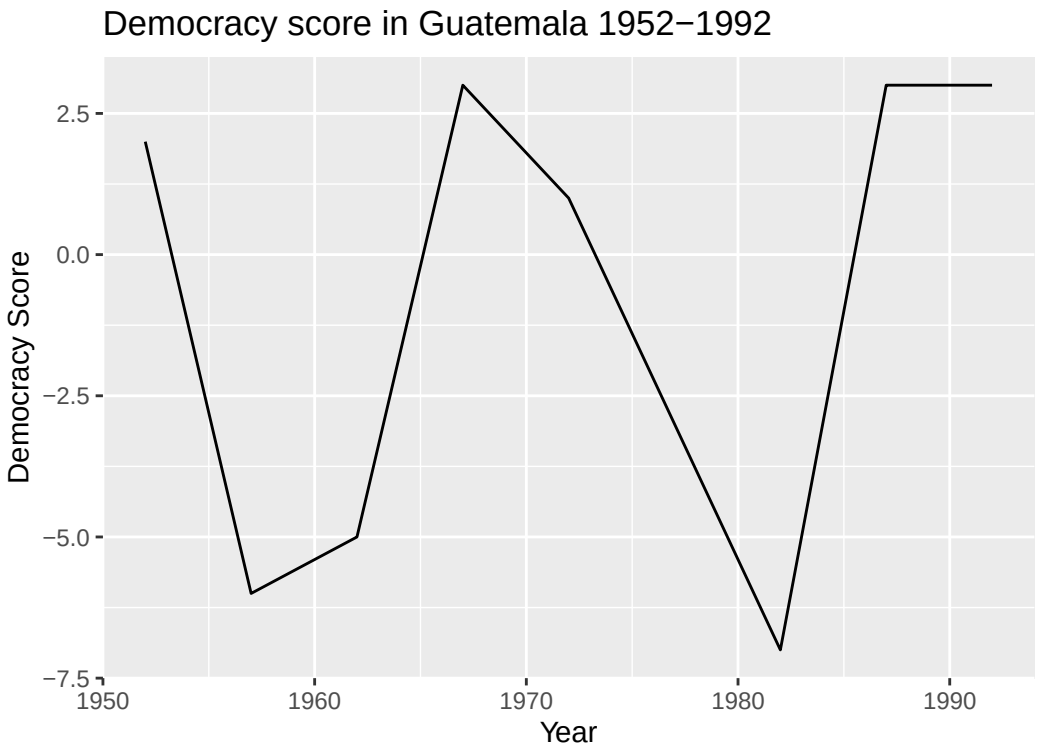
However, observe in the output for `guat_dem_tidy` that the `year` variable is of type `chr` or character. Before we can plot this variable on the x-axis, we need to convert it into a

numerical variable using the `as.numeric()` function within the `mutate()` function, which we saw in Section 3.5 on mutating existing variables to create new ones.

```
guat_dem_tidy <- guat_dem_tidy %>%
  mutate(year = as.numeric(year))
```

We can now create the plot to show how the democracy score of Guatemala changed from 1952 to 1992 using a `geom_line()`:

```
ggplot(guat_dem_tidy, aes(x = year, y = democracy_score)) +
  geom_line() +
  labs(
    x = "Year",
    y = "Democracy Score",
    title = "Democracy score in Guatemala 1952-1992"
  )
```



Learning Check 4.4

Convert the `dem_score` data frame into a tidy data frame and assign the name of `dem_score_tidy` to the resulting long-formatted data frame.

Learning Check 4.5

Read in the life expectancy data stored at https://moderndive.com/data/le_mess.csv and convert it to a tidy data frame.

4.4 Conclusion

4.4.1 `tidyverse` package

Notice at the beginning of the chapter we loaded the following four packages, which are among the four of the most frequently used R packages for data science:

```
library(dplyr)
library(ggplot2)
library(readr)
library(tidyr)
```

There is a much quicker way to load these packages than by individually loading them as we did above: by installing and loading the `tidyverse` package. The `tidyverse` package acts as an "umbrella" package whereby installing/loading it will install/load multiple packages at once for you. So after installing the `tidyverse` package as you would a normal package, running this:

```
library(tidyverse)
```

would be the same as running this:

```
library(ggplot2)
library(dplyr)
library(tidyr)
library(readr)
library(purrr)
library(tibble)
library(stringr)
library(forcats)
```

You've seen the first 4 of the these packages: `ggplot2` for data visualization, `dplyr` for data wrangling, `tidyr` for converting data to "tidy" format, and `readr` for importing spreadsheet data into R. The remaining packages (`purrr`, `tibble`, `stringr`, and `forcats`) are left for a more advanced book; check out R for Data Science to learn about these packages.

The `tidyverse` "umbrella" package gets its name from the fact that all functions in all its constituent packages are designed to that all inputs/argument data frames are in "tidy" format and all output data frames are in "tidy" format as well. This standardization of input and output data frames makes transitions between the various functions in these packages as seamless as possible.

4.4.2 Additional resources

If you want to learn more about using the `readr` and `tidyr` package, we suggest you that you check out RStudio's "Data Import" cheatsheet. You can access this cheatsheet by going to RStudio's cheatsheet page and searching for "Data Import Cheat Sheet".

4.4.3 What's to come?

Congratulations! We've completed the "Data Science via the tidyverse" portion of this book! We'll now move to the "data modeling" portion in Chapters -Chapter 5 and -Chapter 6, where you'll leverage your data visualization and wrangling skills to model relationships between different variables in data frames.

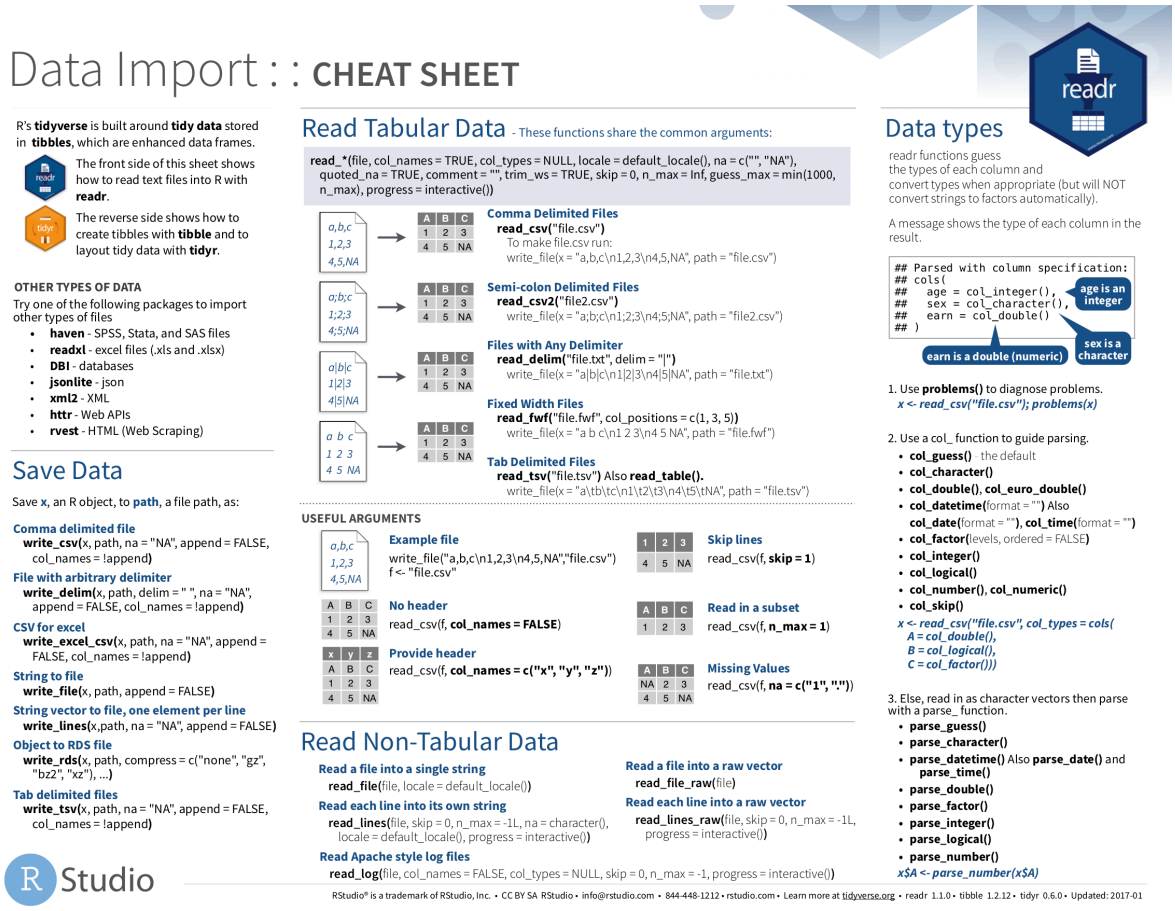


Figure 4.3: Data Import cheatsheat

Part III

Data Modeling

5 Basic Regression

Under Construction

Currently working on content transfer from previous version of the book.

6 Multiple Regression

Under Construction

Currently working on content transfer from previous version of the book.

Part IV

Statistical Theory

7 Randomization and Causality

Under Construction

Currently working on content transfer from previous version of the book.

8 Populations and Generalizability

Under Construction

Currently working on content transfer from previous version of the book.

9 Sampling Distributions

Under Construction

Currently working on content transfer from previous version of the book.

Part V

Statistical Inference

10 Confidence Intervals

Under Construction

Currently working on content transfer from previous version of the book.

11 P-values

Under Construction

Currently working on content transfer from previous version of the book.

12 Hypothesis tests

Under Construction

Currently working on content transfer from previous version of the book.

13 Putting it all together

Under Construction

Currently working on content transfer from previous version of the book.

References

Grolemund, Garrett, and Hadley Wickham. 2016. *R for Data Science*. http://r4ds.had.co.nz/.

Ismay, Chester. 2016. *Getting Used to r, RStudio, and r Markdown*. http://ismayc.github.io/rbasics-book.

Robbins, Naomi. 2013. *Creating More Effective Graphs*. Chart House.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* Volume 59 (Issue 10). https://www.jstatsoft.org/index.php/jss/article/view/v059i10/v59i10.pdf.

———. 2021. *Nycflights13: Flights That Departed NYC in 2013*. https://github.com/hadley/nycflights13.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. 2022. *Ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. https://CRAN.R-project.org/package=ggplot2.

Wickham, Hadley, and Maximilian Girlich. 2022. *Tidyr: Tidy Messy Data*. https://CRAN.R-project.org/package=tidyr.

Wilkinson, Leland. 2005. *The Grammar of Graphics (Statistics and Computing)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

A Statistical Background

Under Construction

Currently working on content transfer from previous version of the book.