Applied Econometrics: An Intuitive Field Guide with R

Matt Dobra

2025-07-01

Table of contents

Preface

In preparation for my first semester teaching Econometrics using R back in 2021, I prepared a series of R Notebooks for use in class. Not only did I use these notebooks to teach the material in class, I also provided them to students for their reference use to study, do homework, and so forth. At one point, in referring to this collection of notebooks, a student told me that they liked my book and found it useful for learning. My first instinct was to think *"it's not really a book, it is just a collection of teaching notes."* Well, after I got over the shock of students actually doing the reading, that is!

But this student's statement stuck with me, and I realized that, while it wasn't really presented or written like a book, it was awfully close to being a book. The notebooks, when executed, were something in the realm of 250 pages of almost entirely original work (I think I "borrowed" something like 2 graphs, which I have since created my own versions of from scratch) and relied on data that was freely available via a variety of R packages. Plus, the work of turning it into a proper book appealed to me as an excuse to expand my knowledge of `rmarkdown` (Allaire et al. 2024), learn how to use `bookdown` (Xie 2024), and clean up a few things that needed to be revised. So, that became my summer project in 2022.

Fast forward 3 years, and I've had enough experience using this text and teaching the class that it is time for a second edition. There are a number of significant changes since the first edition:

- **Significant rewrites to the introduction to R chapter:** The previous edition did not do this terribly well, in my opinion. It blurred the lines between base R and the Tidyverse, and again between R and RMarkdown, combining all of this into a single chapter. This edition breaks this chapter up into three distinct chapters; Introduction to R and RStudio, Data Wrangling with the Tidyverse, and Literate Programming. Additionally, the chapter introducing R and RStudio has added a discussion of important computing habits that I have noticed students often lack, like working with directories and file organization, troubleshooting strategies, and so on.
- **Switch from RMarkdown and Bookdown to Quarto:** As Posit (*née*

RStudio) seems to be moving in this direction, so shall I. I made the switch on the fly in my Fall 2024 class as it seemed that Quarto would be generally more beginner friendly than RMarkdown, which I (thankfully) found to be true in practice. This should allow for a number of other innovations in the book to improve readability, including:
  - **Callouts**: These are added to more effectively guidepost the reader through the material. Several callout types have been added, including sections developing statistica intuition (Data Storytelling), tips for literate programming (May the Format Be With You), R and general computer tips (Tip from the Helpdesk), and more.
  - **Code Folding**: The previous edition often suffered from walls of code, which could get unwieldy. Adding code folding for code chunks improves both the appearance and readability of the book.
  - **Referencing**: The first edition needed significant improvement in this area, Quarto's use of LaTeX referencing will help enable this, allowing for cleaner and more consistent citations.
  - **Cross-Platform Output**: For the life of me, I could never get `bookdown` to create a .pdf file. Quarto does make this easier! that said, the .pdf version of this book does have some severe formatting oddities.
- **General improvements to tone:** One of my goals for the new edition is to keep the writing light, conversational, and approachable. I've made a conscious effort to inject a little more fun and whimsy to the text–adding memes, jokes, and quirky commentary–to better appeal to the college-aged audience. Some of my puns are downright awful, trying to hit that sweet spot of cringe that they are so awful that they are good! My hope is that these touches make learning econometrics and R a bit less technically daunting and intimidating (and dare I say it, dry!) and, just maybe, even enjoyable—or least entertaining!

Motivation

This book originated from teaching materials I developed while serving as Professor of Economics at Methodist University for an undergraduate econometrics course (ECO 3160). In developing this course for a group of students with diverse mathematical backgrounds, I struggled to find "off-the-shelf" textbooks that matched both the level of preparedness of incoming students and the style of course I felt would be valuable to my students. After all, this course had only college algebra and an introductory-level applied business statistics as a prerequisite. At the same time, it needed to serve as preparation for both the capstone course for Economics majors and the data mining and data analytics courses for Business Analytics majors.

The absence of a calculus or computer programming prerequisite greatly informs the approach taken in this text. The calculus and linear algebra underpinnings of

econometrics are minimized, appearing only in a few targeted places as callouts. For example, Chapter Chapter **??** uses basic calculus to demonstrate finding the maximum or minimum in a quadratic regression, and also discusses the *dummy variable trap* using matrix concepts. Likewise, the focus on using R emphasizes scripting, rather than coding or programming. Relatively little time is spent on coding logic, loops, or advanced programming techniques. Instead, the book adopts a more "cookbook" approach, with the following goals:

1. Understand the underlying intuition of various econometric procedures.
2. Identify when specific procedures are generally appropriate.
3. Diagnose some of the most common issues that may arise.
4. Learn how to use R to carry out these procedures effectively.

Who This Book is For

Though initially developed for a specific audience, I believe that this book addresses a common need across undergraduate economics education. Many programs, particularly at liberal arts institutions, have students with limited mathematical preparation entering econometrics courses, similar to the students I was teaching. However, even at institutions with stronger quantitative requirements, students often struggle to connect abstract mathematical concepts with practical applications. I believe that this text can serve both as a standalone resource for programs with minimal math prerequisites and as an intuition-building supplement to more mathematically rigorous textbooks, helping bridge the gap between theory and application.

I've made a deliberate choice to make the code visible throughout this book, though the HTML version enables `code-folding` for those who prefer a cleaner reading experience. While many technical books hide code to focus on concepts, seeing the actual R commands is essential to the learning process. This transparency allows readers to follow along step-by-step, experiment with variations, and develop a deeper understanding of both the statistical concepts and their implementation.

A Humble Request

This book is freely available for anyone to read. Unfortunately, I have no way of knowing who, if anybody, *is* actually reading it. So my humble request is to simply drop me a note if you read this book. Let me know how you found it. Did someone suggest it to you? Did you stumble across it online? What is your purpose in using it? Was it assigned to you in a class, are you assigning it to students, or are you just reading it for funsies? If you spot a typo or an error, let me know. If you love it—or hate it—tell me why. Got a great idea for a new

meme? I'd love to hear it. Even your opinions on the existing memes are fair game. Just let me know!

If I get a few nice comments, I might even add a section to this book the next time I revise it with some of them. An even more interesting section might result if I get less-than-nice comments! Anyhow, I'm pretty easy to find–probably the easiest is on social media at Twitter or Facebook. A quick google search of my name should reveal an academic email address, currently matt.dobra [at] bhsu [dot] edu should you be an email aficionado.

Acknowledgements

While the text of this book is, to the best of my knowledge, of my own writing, it does make use of a variety of data repositories (created by others) that are part of the following R packages:

- `datasets` (2024a) (the datasets built into base R)
- `wooldridge` (Shea 2023)
- `AER` (Kleiber and Zeileis 2008)
- `fivethirtyeight` (Kim, Ismay, and Chunn 2018)
- `ggplot2`(Wickham 2016)
- `carData` (Fox, Weisberg, and Price 2022)
- `dplyr` (Wickham et al. 2023)
- `Stat2Data` (Cannon et al. 2019)
- `openintro` (Çetinkaya-Rundel et al. 2024)
- `Ecdat` (Croissant and Graves 2022)

Additionally, there are places in which data is retrieved live from the web via such packages as WDI (Arel-Bundock 2022), quantmod (Ryan and Ulrich 2024), etc.

About the Author

Matt Dobra, Ph.D. is Dean of the College of Business at Black Hills State University. Prior to this administrative role, he spent nearly two decades teaching economics courses at various levels, with particular focus on econometrics, principles classes, and applied microeconomics. His research interests lie primarily in resource economics and public economics, with published work appearing in journals such as the European Economic Review, Decision Sciences, and Resources Policy. He holds a Ph.D. and M.A. in Economics from George Mason University, a Graduate Certificate in Higher Education from Monash University, and a B.A. in History from Loyola University, New Orleans.

Chapter 1

Introduction

This brief chapter introduces the concept of econometrics by explaining what it is, who uses it, and how it connects to other disciplines, setting the stage for why it is such a valuable set of tools in both economics and disciplines beyond. It also provides an overview of the book, outlining its focus on applied econometrics and what you can expect to learn. By the end of this short chapter, you will have a clearer understanding of the role econometrics plays in analyzing data, answering questions, and informing decisions, as well as how this book will guide you through its practical application using R.

1.1 What is Econometrics?

Econometrics is the branch of economics that applies statistics to economic theory. Since statistics is a sub-discipline of mathematics, some math is unavoidable in econometrics. However, the emphasis of this book is not *econometric theory* but rather it is on *applied econometrics*. Our focus will be more on the application of econometric techniques and the interpretation of econometric results, rather than delving deeply into the mathematical underpinnings of the subject. That said, it is assumed that your level of mathematical understanding is at least equivalent to having taken a college algebra and an introductory statistics course.

Less mathy does not necessarily mean easier. While the theoretical underpinning of econometrics are found in calculus, linear algebra, and probability theory, the application of econometrics requires at least an intuitive understanding of the math and a solid grasp of computer coding and programming skills. In short, while this text will attempt to keep the explicit use of calculus and linear algebra to a minimum, to successfully *do* econometrics will require us to focus heavily on learning basic programming skills using the R language.

In many ways, my goal is to see how far I can help you push your econometric

and statistical intuition without resorting to "doing the math." Which means that, if, after working through this book, you want to dig deeper into these topics, you probably won't get very far without also augmenting your math skills.

1.2 Overview of the Book

Before getting into econometrics, we will start with the basics. The next chapter features an introduction to the R language and the RStudio programming environment. After that, we continue to build our statistical computing skills by learning about data wrangling using the Tidyverse in Chapter **??** and literate programming using Quarto in Chapter **??**. Finally, we use R to do many of the things it is already assumed you know how to do using other methods; for example, calculating descriptive statistics and making graphs (Chapter **??**), and conducting basic inferential statistics (e.g. hypothesis testing, ANOVA) in Chapter **??**. In many ways, this first section of the text is designed to teach you the programming basics needed for this book by showing you how to use R to accomplish tasks that (I hope!) you already know how to do.

From there, we transition into a study of econometrics itself, beginning with the fundamental building block: ordinary least squares (OLS) regression. OLS will be introduced in Chapter **??** and extended in Chapter **??** and Chapter **??**. OLS regression will create our basis for understanding to tackle more sophisticated methods, like probit/logit modeling in Chapter **??**, time series in Chapter **??**, and myriad other advanced techniques found in the latter half of the book.

The fundamental methods of OLS, probit/logit, and time series are not only key tools in economics, but are key tools across a wide variety of disciplines throughout the social sciences, hard sciences, and quantitative areas of business. While people in these areas may be using these techniques for slightly different purposes, or may call them by different names, at the end of the day, developing an understanding of these techniques provides you with a versatile skillset that is useful in a wide variety of contexts well beyond academic economics.

If at any point you find yourself stuck or wanting to go further, the final chapter, Chapter **??**, includes a curated list of excellent texts and books. These resources can guide you toward mastering more advanced econometric techniques, programming skills, and contemporary topics in econometrics, analytics, and visualization.

1.3 A Word About AI Tools

I want to warn you now: the learning curve for this stuff can be pretty steep, and it is natural to feel frustrated or overwhelmed early on when learning econometrics and R. However, I want to strongly urge you to resist the temptation

turn to an AI to do the heavy lifting for you before you have built the foundation upon which to understand both what AI is doing, and why it is doing it. While tools like ChatGPT or other code-generating AI can seem like a quick solution, relying on them too early can short-circuit the learning process. You may get code that works, but without knowing why or how it works, you won't develop the deeper understanding of econometric concepts and R programming that this book aims to teach. Worse, you may get code that produces a result that is nonsensical, but lack the understanding to identify or correct it.

AI can be a powerful tool, but used incorrectly it is an impediment to learning. If you skip the hard work of grappling with the material yourself—writing code, fixing errors, and thinking through problems—you will likely become overly dependent on AI-generated answers and miss out on the skills that make you truly capable. At its core, econometrics is about understanding data, interpreting results, and drawing conclusions—skills that require a strong foundation in both the theory and the application. If you don't take the time to develop that foundation, you won't be able to evaluate whether AI's output is correct, appropriate, or meaningful, which can lead to mistakes that are easy to miss and difficult to correct.

1.4 Final Thoughts

If you invest the time needed to work through this book, you should not only have developed a solid intuition of econometric and statistical techniques, but also the practical skills to apply them using R. Whether you pursue work in economics, finance, business analytics, or anything else grounded in quantitative reasoning, these tools will give you a solid foundation for making sense of data. And who knows-maybe you'll have some fun along the way?

Chapter 2

R and RStudio

The focus of this chapter is to introduce the computer program R and its companion environment, RStudio. This chapter assumes that this is your first experience with programming/coding of any kind, so we will begin with some very basic questions: what are R and RStudio, how do you get them, and how do you install them?

After guiding you through the installation of R and RStudio, we'll take a brief tour of RStudio, highlighting key options and setup choices to help improve your experience and workflow. Finally, the chapter will close with an introduction to some essential features and conventions in R, providing a foundation for everything to come.

2.1 The Basics

Let's start with the basics. R is an open source programming and scripting language widely used for statistical analysis. While it is a core tool in econometrics, it has tons of other applications beyond econometrics as well. It is a commonly used program in any academic discipline that relies upon statistical analysis, so researchers across the social sciences (economics, sociology, psychology, etc.) and physical sciences (epidemiology, biology, meteorology, etc.) rely on R for data analysis. Moreover, there is an increasing use of R for data visualization and analytics, making R popular in the business world for tasks like marketing analytics, business analytics, data mining, financial and research analysis, data journalism, actuarial science, and more. In short, R is highly versatile, widely applicable, and growing in demand.

While the focus of this text is on using R for econometrics, the fundamental methods of econometrics–particularly OLS, logit, and time series analysis–are the fundamental methods of nearly all fields utilizing R as well. Of course, R is not the only program out there that are capable of data analysis and

visualization. In the world of economics, the more commonly used software packages include SAS, SPSS, Stata, and to lesser extents economists utilize programs like Eviews, Matlab, Python, and even Microsoft Excel.

Overall, R stands out as likely the most widely used tool across disciplines, which adds to its overall usefulness.

Pros of R:

- Free and open source-no cost to use!
- Powerful and extensible-thousands of community-created packages.
- Purpose-built for statistics-ideal for econometrics and data analysis.
- Widely adopted and growing-it's *probably* here to stay.
- Strong online community-extensive help, resources, and forums available.
- Reproducible research-tools like Quarto allow you to seamlessly integrate code, analysis, and reporting.
- Cross-platform compatibility-works on Windows, Mac, and Linux with no issues.

Some Cons of R:

- Steep learning curve-it can be challenging for beginners.
- Package overload-difficult to keep track of the many packages and their updates.
- Inconsistent syntax-coding style and conventions can vary widely between packages, creating confusion.
- Mediocre in-program help-some built-in documentation is hit-or-miss.
- GUI-phobic-If you are are point-and-click person, R will be tough to get used to.

While R is both the name of the program and the name of the coding language, it is not typically used on its own. Typically, R is nested inside of another program called RStudio. RStudio is an IDE (Integrated Development Environment)–a fancy term for an app that makes programming a lot easier because it helps keep track of everything you are working on in one place.

RStudio provides a comprehensive set of tools for writing, testing, debugging, and running code in a single enviroment. It offers useful features like color coding and auto-formatting to aid in the process of writing and debugging code, auto-completion for commands and variables that are already loaded into the programming environment, tools to help you keep track of mismatched parenthesis and quotation marks, etc. Coding in R is way easier and more efficient when done within RStudio. The best part? Both R and RStudio are **free** and work in Windows, Mac, or Linux systems.

At this point, if you have not yet installed these pieces of software, you should do so.

1. Install R
2. Install RStudio

3. Profit

Go ahead and do it now; I'll wait!

Tip from the Helpdesk: First Things First - Installation Order Matters!

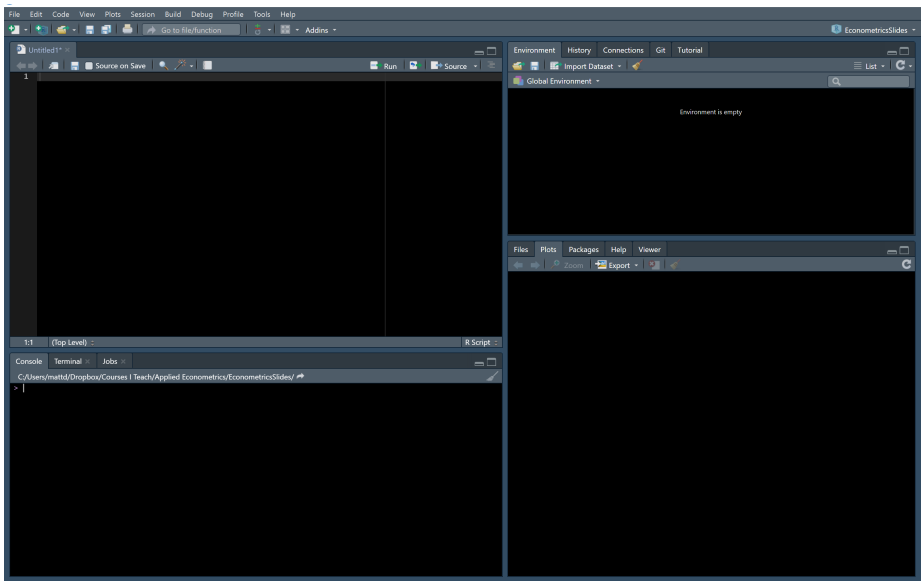
Always install R before RStudio-it saves you a lot of headaches!
This means you need to:

1. Download R: Head to CRAN and grab the latest version for your operating system.
2. Find the installer: Search your downloads folder for the R installer file.
3. Run the installer: Double-click the file and follow the installation prompts. You will know you are done when the installation finishes and you see a message confirming that R has been successfully installed. If it opens the R file, go ahead and close it–this step is complete, and you will likely never run R outside of RStudio.
4. Install RStudio: Once R is installed, download and install RStudio from Posit's website. You may need to click on "Open Source" in the ribbon at the top and choose "RStudio IDE," or use this direct link. Go to your downloads folder, double-click the RStudio installer file, and follow the prompts to complete the installation.
5. Open RStudio: After installation, open RStudio. It should automatically detect your R installation and get you ready to start coding. Again, you likely will never need to open R directly–RStudio is where

all the magic happens!
If you encounter any issues, you may find it useful to head over to YouTube and search for an installation walkthrough.

2.2 A Tour of RStudio

Let's start with a brief overview of the RStudio environment. The basic layout of Rstudio has 4 panes:



Note: The layout in the image may look slightly different from RStudio's default setup, but the differences are minor.

Tip from the Helpdesk: Pane Management

If you don't see four panes, don't worry! You can easily get them to show up with the keyboard code Ctrl + Alt + Shift + 0 (Cmd + Alt + Shift + 0 on a Mac), or via the menu bar by clicking View → Panes → Show All Panes.
That said, it's sometimes useful to minimize one or more of your panes. When writing this book, for example, I typically only have the Editor Pane and the Viewer Pane visible!

2.2.1 Bottom Left: The Console Pane

The console pane is where R lives. You can use R interactively here by typing commands directly into the console. When you hit Enter, the results are processed immediately. R is a command-line interpreter, meaning it evaluates one command at a time as you type it.

However, most of your work will rely on scripts-sets of commands saved in a file that are executed when you tell R to run them. Scripts ensure your work is reproducible and organized. A typical R session will include both interactive commands (quick tests, checks) and scripted work for larger tasks. Scripts are housed in the editor pane.

2.2.2 Top Left: The Editor Pane

This is where you will spend most of your time in RStudio. This is where you write and save your scripts, which are sets of R commands that RStudio can execute sequentially. You can run a whole script through the console pane at once with the run button, or you can send one line at a time to the console pane by selecting a line (or lines) by highlighting them with your mouse and typing Ctrl-Enter (Cmd-Enter on Mac). Unlike typing directly into the console, scripts allow you to save your work, test code in sections, and reproduce your results later. You can have multiple scripts or files open at once, each in its own tab, allowing you to switch between tasks easily.

> Tip from the Helpdesk: Files Don't Save Themselves
>
> If you see an asterisk (*) next to the name of a file on its tab, it means the file has unsaved changes. Saving is like voting in Chicago; one should do it early and often! Frequent saving will help you avoid losing your work! You can save a file by clicking the Save icon or using the shortcut Ctrl + S (Windows) or Cmd + S (Mac).

If you're working with datasets loaded into memory, you can take a peek at them in a spreadsheet-style view in the editor pane. This feature is helpful for quickly inspecting your data and ensuring it looks as expected. One way to do this is by using the `View()` command in the console pane. For example, if you have a data frame called `mydata`, typing `View(mydata)` in the console will display the dataset in the editor window. Keep in mind, this viewer is not a proper spreadsheet-you can't manually edit the data here. It's purely for exploration and inspection.

RStudio also provides smart formatting in the editor pane to make scripting easier, including:

- Color coding: Functions, variables, strings, and comments are visually distinct.

- Bracket matching: Highlighted parentheses, braces, and brackets make it easier to spot mismatches.
- Indentation: RStudio automatically aligns code blocks for better readability.
- Code folding: Collapsible sections of code let you focus on specific parts of your script.

2.2.3 Top Right: The Environment Pane

The environment pane (top right) shows all the objects and variables currently loaded into memory. This includes datasets, lists and vectors, functions, and any other objects you create during your R session. For example, if you load a dataset, its name will appear in this pane along with information about its size and structure.

One particularly useful feature is the ability to click on a data frame to open it in a spreadsheet-style view in the editor pane, making it easy to examine your data visually–basically a shortcut for the `View()` command discussed above. You can also see summaries of objects (like the dimensions of a dataset or the class of an object) without needing to type additional commands.

The environment pane also includes buttons and tools for managing your workspace:

- Search Bar: Quickly locate specific objects in memory, which is especially useful when working with many variables.
- Clear Workspace: The broom button will remove all objects from memory to start fresh (though use this with caution!).
- Import Dataset Button: Launches a wizard to import data from external files (e.g. CSV, Excel). This is a beginner-friendly way to learn how to load data into R.
- List View vs. Grid View: Toggle between a detailed list of objects and a more compact grid format.

Additionally, the environment pane interacts with RStudio projects, which we will talk about later, displaying the workspace associated with your current project. This ensures your work remains organized and makes it easy to pick up where you left off.

Think of the environment pane as your workspace dashboard. It gives you a clear view of everything you've created or loaded into R during your session, helping you keep track of your data and variables as you work.

2.2.4 Bottom Right: The Utility Pane

The bottom right pane in RStudio is versatile and ultimately houses a bunch of useful stuff that doesn't really fit with the vibe of the other panes. It's kinda like a super organized junk drawer. The most commonly used tabs include:

- Files: Browse your project's folder structure directly within RStudio. You can navigate to files, open scripts or datasets, and create new folders or files as needed.
- Plots: View graphs and visualizations generated by your code. If you create multiple plots, RStudio allows you to scroll through them using the arrows in this tab. You can also export plots as image files (e.g., PNG or PDF) for use in reports or presentations.
- Packages: Manage the R packages installed on your system. Packages are add-ons that extend R's functionality, providing specialized tools for specific tasks. You don't need to worry about the details yet, we'll cover packages and how to use them later in this chapter.
- Help: Access R's built-in documentation. When you run a help command (e.g., `?lm`), the output will appear here. You can also search manually for functions, commands, or packages using the search bar.
- Viewer: Used for displaying HTML outputs or other rendered content. In the Chapter on Literate Programming, Chapter **??**, you'll see how this tab becomes important for viewing rendered reports or interactive documents.

The flexibility of the utility pane ensures that you can quickly access the tools you need while working, whether that's managing files, visualizing data, installing packages, or seeking help with R functions.

2.2.5 Menu Bar

The menu bar is the last stop on our tour of the RStudio environment. It contains options and commands typical of menu bars for most programs–options like *File*, *Edit*, *View*, *Help*, etc. At some point you may want to look through these to see what RStudio is capable of, but for now, here are a few useful things to highlight.

- *Code Menu*: Ensure the following 3 items are toggled on as they make reading and debugging your code easier.
  - Soft Wrap Long Lines: without this, long lines of code will have you scrolling horizontally for days!
  - Rainbow Parentheses and Rainbow Fenced Divs: These options color match parentheses in different colors, making it much easier to see which opening parenthesis goes with which closing one. When your code gets complex with nested functions, this visual aid can save you from the frustration of hunting down mismatched parentheses.
- *View Menu* → *Panes*: Here you can find an option to *Show All Panes*. This is useful for situations where you accidentally close one of your four panes and want them back.
- *Help Menu* → *Cheatsheets*: RStudio provides a ton of coding cheatsheets that are incredibly useful for when you are trying to figure out how to do things. You can find a more comprehensive list of cheatsheets at https://rstudio.com/resources/cheatsheets/. Cheatsheets are designed to be printed out on a single double-sided sheet of paper and kept handy

while working–you may find this especially valuable while learning new packages.

- *Help Menu → Keyboard Shortcuts Help*: As you code, you'll find that many repetitive tasks can be done more quickly and easily using keyboard shortcuts. RStudio provides this list of such shortcuts. At this point, looking at that list may be a bit daunting, but after you've become a bit more accustomed to using R, take a look at the list of keyboard shortcuts. You will likely find a few that you will soon view as indispensable. Here are the three I use the most:
  - Ctrl + Shift + M: This shortcut creates the pipe operator, `|>`, which will be introduced in Chapter **??** on Data Wrangling.
  - Ctrl + Alt + I: This key combination will create a new code chunk in Quarto, which we will discuss in Chapter **??** Literate Programming.
  - Ctrl + Shift + C: converts a line of script into a comment, useful for documenting scripts or temporarily ignoring lines when debugging code. More on this later this chapter.
- *Tools Menu → Global Options*: This opens an options window with a variety of settings to customize RStudio. While most of the defaults are fine for you to start with, there are a few useful things to check out in here.
  - *General Tab*: There is the option to set a *Default working directory (when not in a project)*. Your working directory is simply the folder on your computer where R will look for files by default and where it will save output unless told otherwise. It's like R's "home base" on your computer. You should check out the Tip from the Helpdesk on RStudio Projects below, but I'd strongly suggest configuring this option right away. If, for example, you are using R as part of a class, create a folder on your hard drive for that class and set that folder as your working directory for R. Or, if you are simply teaching yourself R for fun, create a folder for learning R and set that as a working directory.
  - *Code Tab*: Ensure that *Use native pipe operator, |>* is ticked.
  - *Appearance Tab*: Here, you can set the theme of RStudio. If you want a dark theme, make sure you change the theme to "Modern" or "Sky". I would suggest choosing a theme that has a lot of contrasting colors to make reading your code easier. You want the five colors of text in the sample display window to "pop," because The different colors in the editor help you identify various parts of your code at a glance. I typically use Tomorrow Night Bright, but I also quite like Ambiance, Chaos, Cobalt, and Vibrant Ink.
  - *Packages Tab*: This enables you to determine where R looks to download packages–the community-developed functions and programs that greatly extend R's power. For the most part you want to ensure that the *Primary CRAN repository* is set to *Global (CDN) - RStudio*. If, however, you find yourself unable to install any packages, you will want to come back here and change your repository, likely to

the repository closest to you geographically.

Tip from the Helpdesk: Projects: the Bento Boxes of R

Just as a bento box keeps your sushi separate from your tempura, RStudio Projects keep your different coding tasks neatly compartmentalized. Each project has its own workspace, history, and working directory, so your econometrics homework won't mix with your data analysis for work. While you might not need projects yet as you're just learning R, they become invaluable when you're managing multiple research projects or juggling different clients at work.
To create a project:

1. Go to File → New Project…
2. Choose to create either a new folder or use an existing one
3. Give your project a name that helps you remember what it's for

When you open a project, RStudio automatically puts you in the right folder and loads the files you had open the last time you were working on the project…no more hunting around your computer for that script you were working on last week!
For this book, consider creating a dedicated project called "Learning R" or "Econometrics" to keep all your practice work organized in one place.

2.3 R Essentials

With our whirlwind tour of RStudio out of the way, it's time to get our hands dirty and start using R. For now, we have two choices for how we use R (we will explore a third option in Chapter **??** Literate Programming). We can either:

1. Type code directly into the console window (bottom left), or
2. Create an R Script by using *File* → *New File* → *R Script*, enter our code into the script, and then manually run the lines from the script.

While it sounds like option 1 is simpler, usually option 2 is the way to go. Why? Scripts allow you to save your work, easily see what you've already done, and make corrections to your code without retyping everything. They are an essential tool for coding, and they make learning easier as they provide a history of what you have already done. At the end of the day:

- Scripting makes your work reproducible. You, or someone else, can re-run your code and get the same results.
- Scripting makes it easier to work on a project over multiple sessions. You can easily pick up where you left off, even after a long break
- Scripting helps you document your work. By adding comments (more on this below) to your code, you can explain your thought process, clarify complex sections, and make your script understandable for both your future self and others who might work with it.

2.3.1 Code vs. Comments

A line in an R script will generally be one of two things–code, or a comment.

- Code: A command or a set of instructions to tell the computer what to do
- Comment: Notes that R will ignore entirely as it goes through your script

So, if R ignores comments, then what is their purpose?

Comments are something that you write for yourself, or people you are working with. You may work on a script with another person who has no idea what your code is all about, or you may not look at a script for a few weeks and have forgotten what you were trying to accomplish! Putting comments in your code is a way of making notes and passing them to people with whom you are working and/or your future self.

It is easy to identify the difference between a line of code and a comment in R–the hashtag symbol `#` precedes a comment, and R ignores anything on a line following a hashtag as well. You can see two examples of comments in the code below:

```
# This line is a comment!
2+2 # This line has both code and a comment!
```

In the example above, the first line began with a `#`, so R would just ignore it and move on. The second line starts with code, and R will read and execute everything in the line up until the `#` and then pretend like the rest of the line doesn't exist.

Comments are your way of documenting code, making it more understandable and easier to share. For now, start practicing by adding simple comments to your scripts as you learn, and concentrate on adding comments that you think will help you make sense of your code later–what will the future you want to know when you return to your script next week, next month, or next year?

2.3.2 Building Blocks of R: Objects and Functions

While R is capable of being an incredibly elaborate calculator and doing things like add $2 + 2$, it is capable of much, much more than just that. For the most part, everything you do in R fits into one of three categories. You are either:

1. Creating *objects* (everything in R is an object),
2. Performing *functions* on objects to create new objects or modify existing objects, or
3. Exploring and examining these objects to understand their structure or content.

I will do my best to separate these three tasks in this section of the book, but unfortunately, a clean separation is not entirely possible. Some objects are

created by functions, looking at objects requires you to use functions, and so on.

2.3.2.1 Objects

2.3.2.1.1 Types of Objects

Let's get started with basic object types and object assignment. At its core, R is built around objects, which are like little boxes that hold information. The three most fundamental object types in R are:

- Values: These are the simplest objects, holding a single piece of information (like the number `42`, or the word `France`).
- Vectors: These are a collection of values, all of the same type (like a list of numbers: `1, 2, 3, 4`).
- Data Frames: These are like tables or spreadsheets, where rows and columns hold different kinds of data.

We begin with the simplest object type, the value.

While using R you will spend a lot of time creating, defining, and manipulating objects. The preferred way of creating an object is with the assignment operator, `<-`. This is literally the less than symbol `<` and a hyphen `-` slapped together. The keyboard shortcut for the assignment operator is Alt + `-` (Option + `-` on a Mac). While you technically can use an equals sign (`=`) instead, that's generally frowned upon as bad form in R programming. Let's start by creating an object `q` and assigning to `q` the value 42 with the following code.

```
q <- 42
```

You can verify that this worked by looking at the environment window-recall, that's the top right pane. The first column tells you the name of your object, `q`, and the second column shows you something about that object. Because `q` is a numeric value (technically, R would call this a numeric vector of length 1), R shows you exactly what the value of `q` is: 42.

> Tip from the Helpdesk: Script it and Rip it
>
> So, how do you make the assignment of `q <- 42` happen? One option is to type that line of code into your console and hit Enter. However, if you are taking my advice and doing this with a script, you will have noticed that entering `q <- 42` into your script and hitting Enter did nothing more remarkable than simply advancing your cursor to the next line of your script.
> There are a few ways to execute code from a script. The simplest method–and the one I use most often–is to put your cursor on the line of code you want to run in your script and hit Ctrl + Enter (Cmd + Enter on a Mac) instead of just Enter. This is actually an example of a super useful

keyboard shortcut! Pressing Ctrl + Enter does two things; this will *both* advance your cursor to the next line of script *and* execute the line of code you just typed.
Alternately, if you look at the ribbon at the top of your script, you will see a button that says *Run*. You can use this button one of two ways. You can either put your cursor on the line of code you want to execute and click the button, which will run that line of code, *or* you can select a large chunk of code with your mouse (this can be one line, multiple lines, or the whole script) and then click the button. This will run every line of code you highlighted!

Another way of verifying that `q` has a value of 42 is to simply type `q` as a line of code.

```
q
```

```
[1] 42
```

When you do this, R will return the value of `q`, here the number `42`. Something that throws off many people when they start using R is not realizing that everything in R is *case-sensitive*. While `q` is `42`, unless you have already assigned something to `Q`, it will tell you there is an error:

```
Q
```

```
Error in eval(expr, envir, enclos): object 'Q' not found
```

The error you see will likely be a little different (you should see `Error: object 'Q' not found"`) than what is printed here, but it will be an error nonetheless. As I said, R takes being case-sensitive very seriously!

There are a couple simple rules that govern the naming of objects. The only characters allowed in the name of an object are letters, numbers, periods, or underscores, and object names must begin with a letter. I could create objects with names like `q1`, `q.1` or `q_1`, but not something like `1q` (because it starts with a number) or q!1 (because `!` is an invalid character).

```
q1 <- 8675309
q.1 <- 2.718
q_1 <- 3.142
```

The fact that `q` is an object that contains the number `42` will remain in R's memory until R is restarted, I overwrite `q`, or I remove `q` (which we'll talk about when we get to functions, a bit later). Overwriting a variable is easy; simply assign a new value to an existing variable name:

```
q <- 420
```

```
q
```

```
[1] 420
```

Objects don't have to be be just numbers. They can hold character values as well–essentially, words or text

```
a <- "Hello World"
a
```

```
[1] "Hello World"
```

Even though 1 is a number, wrapping it in quotation marks means R treats it like a character value, not a numeric one.

```
b <- "1"
```

A second important type of object is a *vector*, a fancy, mathy term for a list. To make a vector, we will need to use the **concatenate** command `c()`. The next bit of code creates two vectors, each with the first few numbers of a couple recognizable sequences:

```
list_squares <- c(1, 4, 9, 16, 25, 36, 49)
list_fib <- c(1, 1, 2, 3, 5, 8, 13, 21, 35, 56)
```

Vectors can also include characters:

```
countries <- c("USA", "Canada", "Mexico")
```

All the elements of a list have to be of the same type. This next bit of code tries to create a list that has two values in it that are characters and one value that is a number.

```
mixed_list <- c("Mt. Everest", 14.6, "Armadillo")
```

R will not allow this mixing of value types within a vector. It will create the list, but it will force everything to be of the same type. Since `"Mt. Everest"` and `"Armadillo"` cannot be forced into being numbers, R will convert `14.6` to be a character.

The last of the fundamental object types to discuss is the data frame. Typically, datasets in R exist in the form of a data frame. Think of a data frame as being a table or a spreadsheet. Each individual column in a data frame is like a vector, in that all of the values in that column are required to be of the same type. A well structured data frame is often described as **tidy data**, meaning that each row is a single observation, and each column represents a different variable.

To make this more concrete, let's use an example. R contains a number of inbuilt data frames for tutorial purposes, the most iconic of which is the `iris` dataset. This data frame includes the measurements of 150 flowers–50 each from 3 different species of *iris*. Let's go ahead and load in the `iris` data frame with the command `data(iris)`.

```
data(iris)
```

For now, you should see `iris` show up in your Environment window as a *Value* with the rest of the objects you may have created so far. You might notice that it lists the object contents as , which simply means that R is waiting for you to actually do something with the `iris` object before it actually loads the data in. This is fine, this is done simply to conserve your computer's memory. As soon as you start to work with the `iris` data, the object will move into the *Data* category in the Environment window, and you will see that it has "150 obs. of 5 variables," which means the dataset has 150 rows and 5 columns.

When we get to functions later in this chapter, we will explore a wide variety of ways to get more information about a data frame. At this point, however, let's just take a quick look at what is in the `iris` data with two functions, `str()` and `head()`. The *structure* function, `str()`, will show us a bit of information about each variable, while `head()` will print out the first six lines of data. These two functions combine to give you a quick and useful snapshot of what the data look like generally. Let's start with `head()`:

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

When you run `head()`, you'll see the first six rows of the dataset printed in the console. Each row represents a single flower, and each column corresponds to a specific feature or measurement of that flower, such as `Petal.Length` or `Species`. This tidy format-where rows are observations and columns are variables-makes the iris dataset a great example of tidy data.

Next, looking at the structure allows us to learn a little more about how each of the variables are stored:

```
str(iris)
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...
```

Each row corresponds to one of the columns of the data frame, and next to the column names, you can see the types of data that each column con-

tains. The first four variables, `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`, are physical measurements of each flower and their type is listed as `num`–a *number*–which is pretty self explanatory! The other variable, `Species`, is stored as a *Factor*, which is a special type of character value. It is a character in the sense that it is not numeric, but R is also recognizing it as being categorical, so it views, for example, all rows in which *Species* = "*setosa*" as being part of the same group.

Here is a list of the most common variable types you will encounter in R:

- Number: Sometimes you will see `num`, sometimes these are listed as `dbl`, or *double*. Doubles are are just numbers that can have decimal values.
- Factor: Often shortened as `fctr`, a factor is a character value where R recognizes the variable as being categorical.
- Character: These can be abbreviated as `chr`. A character value where R does not recognize the variable as being categorical.
- Logical: In shorthand `lgl`, a logical variable is one where the only two possible values are `TRUE` and `FALSE`.

2.3.2.1.2 Subsetting and Extracting From Objects

We often want to extract elements from our objects. In the case of a value, it is simple, as it only has one element. If we want to know what `q` is, we can simply refer to it as `q`. Extracting elements from vectors and data frames is a bit trickier, and requires us to use brackets (`[]`) for subsetting.

We will start with extracting elements from a vector.

Element extraction is extremely powerful and useful in R. Understanding this first requires an understanding of *indexing*, which essentially refers to the position of an element in an object; the first element of any object is at index position 1, the second element is at index position 2, and so forth.

The following commands extracts the fourth element from `list_squares` (the fourth square number is 16) and the sixth element from `list_fib` (the 6th number in the Fibonacci sequence is 8):

```
list_squares[4]
```

```
[1] 16
```

```
list_fib[6]
```

```
[1] 8
```

We can extract all but certain elements with the negative sign; this is called negative indexing. Let's see `list_squares` without the third element:

```
list_squares[-3]
```

```
[1]  1  4 16 25 36 49
```

Another technique is to pull a specific subset. Say we want the 3rd through 6th element of the `list_squares` vector. The colon operator (:) generates a sequence of numbers from the first to the last number, inclusive. In this case, 3:6 creates the sequence 3, 4, 5, 6, allowing you to extract these specific elements.

```
list_squares[3:6]
```

```
[1]  9 16 25 36
```

A very common use of this functionality is to extract based on a condition, or *filter* your data. Filtering is very useful when working with large datasets, or trying to work with specific subsets of a larger data set. For example, the next command will extract all the elements of `list_fib` that are greater than 10:

```
list_fib[list_fib>10]
```

```
[1] 13 21 35 56
```

Filtering like this is a bit trickier than simply using indexes because of the use of the comparison operator to evaluate whether each element in `list_fib` is greater than 10. The most common comparison operators in R are:

- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to
- `!=`: Not equal to
- `==`: Equal to

The `==` comparator is probably the trickiest one to wrap your mind around; note the double equals sign! To differentiate between `=` and `==`, I like to read `=` as "is" or "equals" and `==` as "is equal to". This mental distinction is usually pretty good for helping me determine which symbol to use, as it helps clarify that `==` is used when comparing values.

Extracting elements from a data frame is decidely more complex than from a vector because you need to specify both rows and columns. That said, most of the techniques found above in the vector subsetting section work here. The general syntax for subsetting a data frame is `dataframe[row, column]`

- The number before the comma refers to the row(s) you want to extract.
- The number after the comma refers to the column(s) you want to extract.
- Leaving one of these blank will include all rows or all columns, respectively.

Let's turn back to the `iris` dataset for some concrete examples.

Suppose we want to extract the value in the third row and fifth column, which corresponds to the species of the third flower:

```
iris[3, 5]
```

```
[1] setosa
```

```
Levels: setosa versicolor virginica
```

You can extract an entire row or column by leaving one of the indices blank. Let's extract the entire 73rd row:

```
iris[73, ]
```

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
73          6.3         2.5          4.9         1.5 versicolor
```

And now, the entire third column (`Petal.Length`):

```
iris[, 3]
```

```
  [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4
 [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2
 [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0
 [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0
 [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0
 [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3
[109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0
[127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9
[145] 5.7 5.2 5.0 5.2 5.4 5.1
```

To extract a specific subset of rows and columns, specify both indices. For example, let's extract rows 1–3 and columns 1–2 (Sepal.Length and Sepal.Width):

```
iris[1:3, 1:2]
```

```
  Sepal.Length Sepal.Width
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
```

Things get tricker when we want to filter specific rows or columns. Filtering rows based on a condition is similar to subsetting vectors but applied to the rows of the data frame. Suppose we want to just look at the `versicolor` irises.

```
iris[iris$Species == "versicolor", ]
```

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
51           7.0         3.2          4.7         1.4 versicolor
52           6.4         3.2          4.5         1.5 versicolor
53           6.9         3.1          4.9         1.5 versicolor
54           5.5         2.3          4.0         1.3 versicolor
55           6.5         2.8          4.6         1.5 versicolor
56           5.7         2.8          4.5         1.3 versicolor
57           6.3         3.3          4.7         1.6 versicolor
58           4.9         2.4          3.3         1.0 versicolor
59           6.6         2.9          4.6         1.3 versicolor
60           5.2         2.7          3.9         1.4 versicolor
```

```
61          5.0         2.0          3.5         1.0 versicolor
62          5.9         3.0          4.2         1.5 versicolor
63          6.0         2.2          4.0         1.0 versicolor
64          6.1         2.9          4.7         1.4 versicolor
65          5.6         2.9          3.6         1.3 versicolor
66          6.7         3.1          4.4         1.4 versicolor
67          5.6         3.0          4.5         1.5 versicolor
68          5.8         2.7          4.1         1.0 versicolor
69          6.2         2.2          4.5         1.5 versicolor
70          5.6         2.5          3.9         1.1 versicolor
71          5.9         3.2          4.8         1.8 versicolor
72          6.1         2.8          4.0         1.3 versicolor
73          6.3         2.5          4.9         1.5 versicolor
74          6.1         2.8          4.7         1.2 versicolor
75          6.4         2.9          4.3         1.3 versicolor
76          6.6         3.0          4.4         1.4 versicolor
77          6.8         2.8          4.8         1.4 versicolor
78          6.7         3.0          5.0         1.7 versicolor
79          6.0         2.9          4.5         1.5 versicolor
80          5.7         2.6          3.5         1.0 versicolor
81          5.5         2.4          3.8         1.1 versicolor
82          5.5         2.4          3.7         1.0 versicolor
83          5.8         2.7          3.9         1.2 versicolor
84          6.0         2.7          5.1         1.6 versicolor
85          5.4         3.0          4.5         1.5 versicolor
86          6.0         3.4          4.5         1.6 versicolor
87          6.7         3.1          4.7         1.5 versicolor
88          6.3         2.3          4.4         1.3 versicolor
89          5.6         3.0          4.1         1.3 versicolor
90          5.5         2.5          4.0         1.3 versicolor
91          5.5         2.6          4.4         1.2 versicolor
92          6.1         3.0          4.6         1.4 versicolor
93          5.8         2.6          4.0         1.2 versicolor
94          5.0         2.3          3.3         1.0 versicolor
95          5.6         2.7          4.2         1.3 versicolor
96          5.7         3.0          4.2         1.2 versicolor
97          5.7         2.9          4.2         1.3 versicolor
98          6.2         2.9          4.3         1.3 versicolor
99          5.1         2.5          3.0         1.1 versicolor
100         5.7         2.8          4.1         1.3 versicolor
```

You may have noticed that the complexity just ramped up with that last line of code! A few things need to be explained:

- The " " around `versicolor` indicate that `versicolor` is a character string. In R, text values (also called character values) must always be wrapped in either single (') or double (") quotation marks to differentiate them

from object names, numeric values, or commands. Without the quotes, R would think you were referring to an object named versicolor, which doesn't exist unless you've defined it earlier.

- The $ operator is used to refer to a specific column within a data frame by its name, rather than by its index position. Think of it as saying, "Hey, R, show me just this column." In this case, iris$Species pulls out the Species column from the iris data frame, which contains the species classification for each flower.
- After the condition (`iris$Species == "versicolor"`), there's a comma followed by nothing. This syntax tells R to filter rows based on the condition, but keep all columns. If we wanted to filter specific rows and specific columns, we'd include an additional condition after the comma.

The key difference between subsetting vectors and data frames is that, when extracting elements or filtering data frames, remember that you must specify both rows and columns. This added complexity makes data frame subsetting more powerful but also trickier than working with vectors.

Subsetting and extracting elements from data frames is a powerful skill that allows you to focus on specific parts of your data, whether by selecting rows, columns, or filtering based on conditions. These techniques are essential for data analysis and are widely applicable across various projects.

In Chapter **??** on Data Wrangling, we'll explore the **Tidyverse**, a set of R add-ons designed to make working with data more intuitive. I personally prefer the Tidyverse, and we will make more use if it throughout the text than the methogs above. The Tidyverse introduces methods for subsetting and extracting data that most people find easier to learn and use, especially with larger datasets or more complex operations. That said, it's still worth learning these foundational techniques. Even as the Tidyverse simplifies and streamlines many tasks, understanding these basics will give you greater insight into how R works and will be invaluable when you encounter situations where Tidyverse tools might not be the best fit.

2.3.2.2 Functions

Let's talk about functions! Functions are the heavy lifters of R-they handle most of the actual work in your scripts, and using them is the key to nearly everything you will do in R. In this section, we'll break down what functions are, how they work, and why they're so important. I'll also introduce a few super useful functions to get you started, but the goal here isn't to take a deep dive into specific functions. Instead, we're focusing on the big picture: understanding how functions operate so you can use them effectively.

Functions are how we get R to do things. They take input, process it, and (usually) give us some output. Functions are so central to R that you've already used some-commands like `head()`, `str()`, and `data()` are all functions!

One of the first functions to know is rm(), which stands for "remove." This handy function lets you delete objects from your environment when they're no longer needed. Do you no longer need `q`?

```
rm(q)
```

Poof! It's gone. Do you want a reset button? Get rid of all the objects in your environment? This is a bit of a trickier command, but:

```
rm(list = ls())
```

Be careful with this one; it's the nuclear option!

Beyond `head()` and `str()`, there are a few useful tools for exploring your data:

- `View()`: Opens your data frame in a spreadsheet-style viewer. Great for getting a bird's-eye view.
- `summary()`: Provides basic summary statistics for each column in a data frame. Perfect for a quick look at your data's distribution. This function has a lot of other uses for other object types as well; we will look at them later.
- `class()`: Tells you the data type of an object. Is it a vector? A data frame? Something else? You can also feed a particular column into `class()` to learn about the type of variable it is.
- `names()`: Returns a list of the variable names in a dataset. This also has a lot of uses for different object types.
- `levels()`: Useful for exploring factors (categorical variables). This shows you the categories that exist in a factor variable.

Let's use some of these new functions with the `iris` data set. But first, since I just nuked my workspace with `rm(list = ls())`, I need to get it back!

```
data(iris)
```

The results of `View(iris)` won't show up in this document, because technically, `View()` is not an R command, it's an RStudio only command. If you execute the line of code below, you should see a spreadsheet-like view of the dataset pop up in your Editor pane. While `View()` is very useful in small datasets, when you start to encounter massive ones `View()` can get sluggish.

```
View(iris)
```

We can get a brief overview of some of the summary statistics of the data with the `summary()` function:

```
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
```

```
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
       Species
 setosa    :50
 versicolor:50
 virginica :50
```

For the `iris` data, it creates numerical summary statistics of the 4 numeric variables and a tabulation of the categorical variable.

Next, we can see the result of the `class()` function:

```
class(iris)
```

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

This is not that useful. Where `class()` shines with data is when you look for the class of a variable:

```
class(iris$Species)
```

```
[1] "factor"
```

The `levels()` function is really useful when looking at factor variables–knowing the variable is categorical is good, but what if you want to know what the possible values for your factor variable are? This is where `levels()` comes in handy:

```
levels(iris$Species)
```

```
[1] "setosa"     "versicolor" "virginica"
```

Functions often have more than one input, called arguments, that let you customize what the function does. Let's look at an example using the `round()` function, a fairly simple function which rounds numbers to a specified number of decimal places. The syntax for round() is `round(x, digits)`, where `x` is what you want to round and `digits` is the number of decimal places to round to.

Let's work with a couple examples to see how this works. Let's create an object called `pi` that is the mathematical constant π to 10 decimal places.

```
pi <- 3.1415926535
```

The "proper" way to specify arguments is with the `=` sign, so we might round `pi` to three decimal places by typing:

```
round(x = pi, digits = 3)
```

```
[1] 3.142
```

If we are going to do it the "proper" way, then it doesn't matter which order we put our arguments. Note that reversing the order of the arguments gives the same result:

```
round(digits = 3, x = pi)
```

```
[1] 3.142
```

Once you become more accustomed to specific functions, and you learn the default order of the arguments, you can dispense with the `x =` and `digits =` and just enter the arguments directly into the function. If you do this, R assumes that we are putting our arguments into the function in the order of `(x, digits)`, so the easiest way to round `pi` to 3 decimal places is:

```
round(pi, 3)
```

```
[1] 3.142
```

Strictly speaking, the `digits` is an *optional* argument; you can specify a number of places to round to, but if you don't, there is a default value pre-programmed into R. In this case, the default value is 0. If I want `pi` rounded to the nearest whole number, I could type:

```
round(pi)
```

```
[1] 3
```

> **Tip**
>
> ######Tip from the Helpdesk: Parental Advisory Explicit Content
> When should you use the `=` sign to specify arguments explicitly? It really depends on the situation:
>
> - Be Explicit: If you're new to a function or sharing your code with others, using = makes your intentions crystal clear. For example, `round(x = pi, digits = 3)` leaves no doubt about what you're doing, even to someone seeing the code for the first time. Explicit arguments also prevent mistakes when functions have many inputs, or when you're skipping over optional arguments. This is especially helpful if you only want to change one or two specific arguments from their default values while leaving the rest untouched.
> - Skip the Explicitness: Once you've mastered a function and know its argument order by heart, you can skip the = for a quicker, cleaner look. For instance, `round(pi, 3)` works just as well and is easier to read when the function is simple and the meaning is obvious.

> Ultimately, the choice between explicit and implicit argument calling depends on your context. I suggest that you prioritize clarity and explicitness when learning, debugging, or sharing code. As you gain experience and familiarity with functions, you can adopt a more streamlined approach for speed and simplicity. Find what works best for you!

A final useful bit to note: you will often want to save the output of a function as an object. This can be done with the assignment operator. For example:

```
pi3 <- round(pi, 3)
```

This creates an object called `pi3` that is `pi` rounded to three decimal places. Note that creating the object doesn't create any output! However, you should be able to see `pi3` in your environment window, and I can use it in my code if I want:

```
pi3
```

```
[1] 3.142
```

So, let's stretch our legs a bit and combine some of the techniques we've seen thus far. Suppose we want to calculate the average petal length for each of the three different iris types, and to save each of these averages as an object. How might we do this? We need one function that we haven't seen yet, the `mean()` function, but the rest of this task can be accomplished with what we've learned so far. We'll use the mean() function to calculate the average, combined with subsetting and filtering techniques to isolate each species. For example, this line of code will calculate the mean petal length of setosa irises and put it into the object `mean_setosa`:

```
mean_setosa <- mean(iris$Petal.Length[iris$Species == "setosa"])
```

How does this work?

- `iris$Petal.Length`: This selects the Petal.Length column from the iris dataset.
- `[iris$Species == "setosa"]`: This subsets the Petal.Length column to include only the rows where the species is setosa.
- `mean()`: This calculates the average of the subsetted petal lengths.
- Assignment with `<-`: Finally, we assign the calculated mean to a new object called `mean_setosa`.

How would you go about calculating the means of the versicolor and virginica irises? Think about it, then check the answer below:

```
mean_versicolor <- mean(iris$Petal.Length[iris$Species == "versicolor"])
mean_virginica <- mean(iris$Petal.Length[iris$Species == "virginica"])
```

If I told you that the function to calculate a standard deviation was `sd()`, do

you think you could repeat this task for the standard deviation for each iris type?

Tip from the Helpdesk: Help! I Need Somebody!

Stuck on a function? Encountered an error that makes no sense? Welcome to programming-this happens to everyone. Here are some strategies to get unstuck:

- Read the Error Message: The error messages in R often feel cryptic at first, but they usually contain valuable clues. Carefully read the error message and pay attention to the function or object it mentions. For example, `Error in mean(x) : 'x' must be numeric` is telling you that there is a mismatch between what the function `mean(x)` wants you to use as an argument and what you actually used as an argument. Or, in English, you are trying to calculate the average of something that isn't a set of numbers!
- Google It: If the error message doesn't make sense, copy and paste it into Google (or your search engine of choice). Chances are, someone else has encountered the exact same issue before. Include "R" in your search query to narrow results to relevant discussions.
- Search Discussion Boards: Websites like Stack Overflow are goldmines for R troubleshooting. Search the error message, function name, or problem description-many questions have already been asked and answered. If you don't find an exact match, consider posting your question (and include a clear explanation with reproducible code!).
- Check R's Built-In Help: Use `?function_name` (e.g., `?sd`) or `help(function_name)` in the console to access R's documentation for a function. While this can be dense, it often includes examples that can clarify how a function works.
- Experiment and Simplify: If the error seems mysterious, try breaking your code into smaller chunks and running them one by one. This can help pinpoint where things are going wrong.
- Ask for Help (Politely): If all else fails, reach out to someone knowledgeable, such as a colleague, professor, or online community. When asking for help, provide context, include the full error message, and share your code (or a simplified version of it). Clear questions get faster, more helpful answers.

Remember, troubleshooting is an essential skill in programming, and learning how to solve problems independently will make you a stronger coder. Most importantly-don't get discouraged! Every coder, no matter how experienced, hits roadblocks. I genuinely feel that I've learned more R from fixing mistakes in my code than any other way!

2.3.3 Expanding R: Packages

Every command this far has used what is called **Base R**. Base R is the basic software that contains the R programming language and many statistical and graphical tools. However, R is also extensible via packages, user-written sets of commands that are typically open-source (e.g. freely available) that expand upon the capabilities of R.

Most R packages uploaded to the CRAN network–recall, we configured your preferred CRAN server earlier in this chapter–and are relatively easy to install. Packages in R must be installed before they can be used, *and* they must also be loaded into memory every time you use them.

For example, let's say I want to calculate the mean rate of return of an asset. I know that the proper method for calculating a mean rate of return is to calculate a geometric mean, not an arithmetic mean, because arithmetic means tend to overstate average rates of return. Unfortunately, Base R doesn't have a function to calculate a geometric mean, so I will need to find a package that another R user has written that includes such a function. It turns out that there is a package called `EnvStats` (Millard 2013) that includes a function called `geoMean()` which does precisely that.

To install a package. you use the command `install.packages()` and put the name of the package to be installed in quotation marks inside the parentheses:

```
install.packages("EnvStats")
```

> Tip from the Helpdesk: You only need to install a package once
>
> It is generally bad idea to include an `install.packages()` command within a script, because this generally leads to attempting to reinstall packages repeatedly which is a waste of time and often breaks your code anyhow.
> If you insist on putting the `install.packages()` command into your script, only run it once, and then comment it out by adding a hashtag (`#`) before the command!
> Think about it like this: `install.packages()` is like going into the app store on your phone to get an app, `library()` is like clicking the icon on your phone. You wouldn't reinstall Instagram every time you want to use it, right?

Once a package is installed, I need to let R know when I want to use it. When you open R via RStudio, the only thing that starts right away is Base R, so the only commands you can use natively are those from Base R. If I want to use the `geoMean()` function from within the `EnvStats` package I just installed, I need to let R know where to find the `geoMean()` function. There are two ways of doing so.

The first method (and frankly, the less preferred method) uses the double colon operator - `::` - and has the general syntax of `library::function`. To see this in action, let's create a vector with 6 months of rates of return for an asset:

```
ror6 <- 1 + c(.04, .13, -.03, .11, -.05, .08)
```

Let's assume I want to calculate the average rate of return, which is where the geometric mean comes in. Next, let's use the double colon method to calculate the geometric mean using the `geoMean()` function from the `EnvStats` package:

```
EnvStats::geoMean(ror6)
```

```
[1] 1.044461
```

The `geoMean(ror6)` function shows that the rate of return of my asset is 4.4%.

The double colon operator is useful if you only plan on using a function from a particular library once in a script or coding session, as it doesn't require R to load everything in the package in at once. This conserves your computer memory, and weird things can happen when you have too many packages loaded at once. However, it is often easier to simply load the library into memory so you can access the function without typing `::` all over the place. Loading a package into memory is accomplished with the `library()` command. So if I wanted to use the `EnvStats` package, I would type `library(EnvStats)` (unlike with the `install.packages()` function, this time I don't need quotation marks) into R and then I could use all of the functions contained within. This next code chunk first loads `EnvStats`, so I can directly use `geoMean()` in the following line.

```
library(EnvStats)
geoMean(ror6)
```

```
[1] 1.044461
```

Generally speaking, the `library()` approach is used far more often than the `::` approach.

> Tip from the Helpdesk: Deep Dive into Packages with Vignettes
>
> Many R packages include vignettes-detailed guides and examples to help you get started. To explore vignettes for a package, use browseVignettes("package_name") or vignette("topic", "package_name"). They're like mini-tutorials designed to show off the package's capabilities!

As you continue through this book, you'll encounter various R add-ons and tools that extend its functionality. Each package will be introduced when it becomes relevant, along with guidance on how to use it effectively. This approach ensures that you won't need to learn everything all at once, making the process more manageable and less overwhelming.

2.4 Wrapping Up

This chapter has laid the foundation for your journey with R, introducing you to some essential tools, terminology, and concepts. From understanding the relationship between R and RStudio to diving into objects, functions, and even extending R's capabilities with add-ons like packages, we've taken the first steps toward mastering this powerful programming environment.

As you move forward, remember that R is both a tool and a language. Like learning any new language, the key to success is consistent practice, patience, and curiosity. Don't be afraid to experiment and make mistakes. Embrace the mistakes; it is the through the fixing of them that learning occurs. I want to reinforce a sentiment I mentioned in Chapter **??** Introduction; the learning curve might be pretty steep, and if you are feeling frustrated or overwhelmed right now, that's understandable. But the worst thing you could do right now in terms of learning R is to turn to an AI to do the work for you. Sure, you'll get code that works. But you won't have learned a thing.

In the next chapter, we'll dive deeper into data wrangling, where you'll learn how to clean, reshape, and manipulate data using more advanced techniques. You'll also get to know the Tidyverse, a game-changing set of tools that make working with data in R even more intuitive and enjoyable.

Chapter 3

Data Wrangling and the Tidyverse

This chapter will take a deeper dive into the techniques used in manipulating and transforming data.

In the world of business and data analytics, as well as when applying the techniques of econometrics in a real-world business scenario, a significant amount of time is spent on **ETL**-Extract, Transform, Load-when managing data. The data we wish to analyze often exists in different places, is saved in incompatible formats, is measured in conflicting or unnatural units, is not recorded in formats that are consistent with the types of questions we want to ask, or is just plain messy. Thus, analyzing data often requires a lot of heavy work up front to:

- **Extract** the data: The data needs to be pulled in to R from somewhere else. Sometimes this is as simple as loading in nice and tidy CSV file that is ready to go, but sometimes we need to grab data from lots of sources and Frankenstein them together.
- **Transform** the data: We often need to combine datasets and/or variables in a way that allows us to conduct the types of analysis we want to. This is especially difficult when the data are coming from different sources, as data from different sources often doesn't fit together neatly. Even if the extraction process is straightforward, we may need to identify problems in the data, convert units of measure, and so forth.
- **Load** the data: In an R-based econometrics workflow, the load step can be a bit less obvious, especially if we are not exporting the results of the extract and transform steps into some other software or data warehouse for analysis. In this case, and thus in a lot of what this book does, the transform and load processes are combined into one. However, if we take a bit more of an expansive view of the concept of load, the work we will be discussing in Chapter **??** Literate Programming is a sort of load process,

as we will be loading our data and/or analysis into a document format (HTML, PDF, DOC, etc.), which is the endpoint of our analysis.

If it helps, I think a reasonable metaphor for the ETL process is cooking dinner for your family. The extract process is when you go to the store (or multiple stores) to buy ingredients, pull the knives and pots and pans out of your cabinets, etc. The transform process is washing, peeling, seasoning, chopping, cooking, and so forth; taking the raw materials and turning them into something usable for your final product. The load process, then, is plating your food and serving it to your family or guests. Hopefully someone else is doing the dishes!

While we will be using a variety of packages in this chapter, the one that will be the star of the show is the `tidyverse` (Wickham et al. 2019) package. Technically, the `tidyverse` is a collection of packages, and when we run the `library(tidyverse)` command a slew of packages will load; the ones that will be front-and-center in this chapter are `dplyr` (Wickham et al. 2023) and `readxl` (Wickham and Bryan 2023), but we will make use of many of the other `tidyverse` packages throughout the book.

Before we get started, you should ensure that you have already installed and loaded into R the packages that will be featured in this chapter:

```
library(tidyverse) # Loads the tidyverse family of packages
library(readxl) # Allows us to import Excel worksheets into R
library(openintro) # A data-centric package
library(gapminder) # Provides a dataset that we will use for examples
```

In general, I will begin each chapter with a code chunk that loads in each of the libraries that will be utilized within it. This approach keeps everything organized and should help you get ready to dive into the rest of the chapter.

Tip from the Helpdesk: Library First Before You Go Go

A good habit when scripting is to load all necessary libraries at the very beginning of your script. For example, because most scripts I write use the `tidyverse` in some way, as a course of habit the first line in nearly every script I write starts with:

```
library(tidyverse)
```

Why? This ensures that all the tools you need are available up front, avoiding mid-script interruptions. If you are well into your script and discover that you need another package, add its `library()` line to the top of your script-not scattered throughout or at the end.
This tip will become even more important when we get to Chapter **??** Literate Programming, so we may as well start to develop good habits right now!
Also, one more reminder: keep `install.packages()` out of your scripts.

This command should generally only be run manually, outside of your script, and only once per package.

3.1 Loading Data into R

There are three primary ways of loading data into R.

- Importing from a file: This involves using Base R functions like `read.csv()` or specialized functions in packages like `readr` to read data in from comma-separated values (CSV), text files, or even Excel Workbooks. Many other statistical software packages, such as SAS, Stata, and SPSS, have their own proprietary data formats; these too are typically importable, though they often require specialized packages like `haven` (Wickham, Miller, and Smith 2023).
- Programmatically downloading data: A massive variety of packages exist that allow an R user to pull data into R from other sources. These include packages for web scraping, connecting to databases like SQL, accessing API to pull data from public sources, and so forth. While incredibly versatile, this method often requires specific packages and coding techniques tailored to the source, making it somewhat beyond the scope of this book. However, there's at least one example later in @sec-assumptions where we'll see this in action.
- Pulling data from an R library: We've already seen that R includes data that can be accessed via the `data()` command in Chapter **??** R and RStudio. In addition to the 100 or so datasets built into Base R (technically they are in the `datasets` (2024b) package that is autoloaded when you start R, but that's splitting hairs), many R packages include datasets that are just as easily loaded into R with `data()`. In fact, there are a lot of R packages that are primarily data libraries that basically exist for the purpose of supplying datasets for teaching and learning.

This book will mostly stick to the third method because it guarantees that the data you're working with matches exactly what's used in the examples. For your own projects, though, you'll probably spend a lot of time importing your own files or programmatically downloading data-essential skills for real-world analysis. I'll share a few quick tips on importing files, but when it comes to programmatic downloads, your best bet is to dive into the documentation for the specific packages you're working with.

When importing data from a file, the easiest workflow is typically:

1. Ensure that the data you are about to import is ready to be imported. Generally this means using software like Microsoft Excel to to ensure that the first row (and only the first row) are variable names, that you have *good* variable names (see the *What's in a Name?* tip below), there are

no blank rows or columns, and that the data is in a file format R can handle (R generally does best with CSV or Excel files). In other words, you want your data to be *tidy*, as discussed in Chapter **??** R and RStudio. This step may not always be possible-Excel may not be able to handle opening larger datasets. However, if the dataset is reasonably small, it is worth doing this step first, as it will help make what comes next-data wrangling-significantly easier.

Tip from the Helpdesk: What's in a Name?

You can make your coding experience go a lot smoother if you follow some basic guidance when it comes to naming your variables. While R is fairly permissive when it comes to naming variables, here are some tips and conventions to keep in mind when starting out

- Be descriptive and have variables with names like `age` and `income` instead of `x1` and `x2`. It's not a big deal if those are the only two variables in your dataset, but more often than not, you have dozens of variables and you don't want to have to keep a secret decoder chart handy just to figure out what's what-just because we call it code doesn't mean we want it to be undecipherable!
- Make consistent use of capitalization and naming conventions. A couple common ones are snake_case (`var_name`), where you use only lowercase characters and put underscores between words, and camelCase (`varName`), where you start with a lower case word, have no spacing between words, and capitalize subsequent words. I personally use snake_case, as you will see throughout this book, however not everybody does. When you are working with data you get from someone else, or working on a project with other people, you either need to adapt or do the work to change the variable names to your preferred convention, which we will discuss later in this chapter. Ultimately, which convention you find works best for doesn't matter; the key is to be consistent and it will make your coding go much more smoothly.
- Even though you don't strictly have to, you should try to follow the same rules for variable names as are required of object names. That is, start them with letters, and only include letters, numbers, underscores, and periods.
- Avoid using spaces in variable names at all costs-even though it is allowed, it is a pain to code with variables with spaces in their names. This is often the biggest issue with importing Excel files, which are designed to be viewed by the user and therefore their authors tend to prioritize variable names that are convenient for humans to read, not computers.

2. Use the *Files* tab in the utility pane (bottom right pane in RStudio) to

navigate to the directory where the file you want to import is located. One of the benefits of working with RStudio Projects is that, if you put the file you are importing in the project directory, locating it is easy and the code you generate later will be more straightforward.

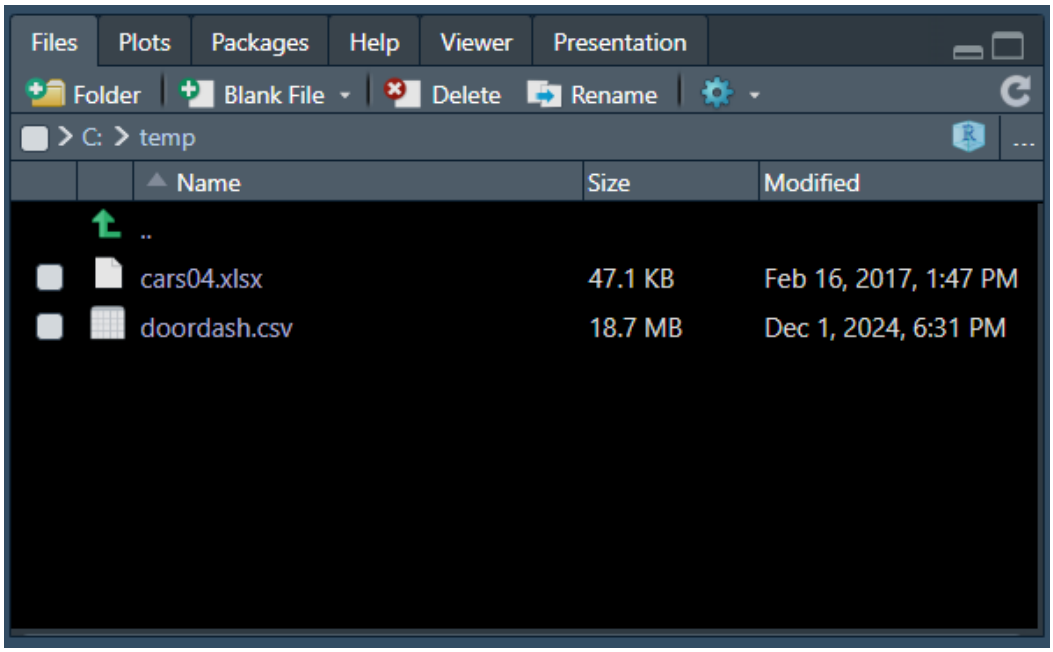


Figure 3.1: The *Files* tab in the Utility Pane

3. Click on the file you want to import, and then choose *Import Dataset…*, which opens up an import wizard. We will be opening the `cars04.xlsx` file, which will ultimately requires us to use the `readxl` package (Wickham and Bryan 2023).

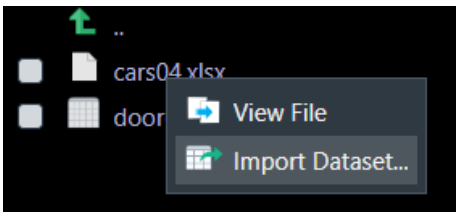


Figure 3.2: Choose *Import Dataset…*

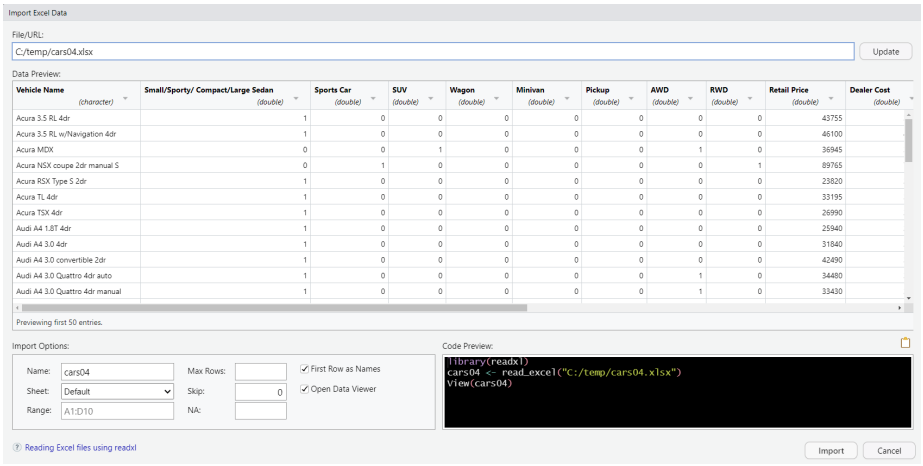


Figure 3.3: Excel Import Wizard

4. The import wizard should show you a preview of the first 50 rows of data. Make sure that it looks like the data should import correctly. Check your column names and variable types, especially if you skipped step 1 above. Here, I can see that I have my work cut out for me when it comes to variable names-the naming conventions are gross, there are spaces and characters I want to avoid (the variable named `Small/Sporty/ Compact/Large Sedan` hurts my soul). As this is a small data set, I might want to open it up in excel first, rename everything there, and then redo the import. Or, I just anticipate doing this work with code. Regardless, at this point you want to evaluate whether or not you want to do the import right now, or if you want to clean up the file more before you import it. Let's assume that I want to just import the data now. The bottom right area of the import wizard has a section called *Code Preview* with the code that will import the data. Clicking the import button will start the process, but you should also copy and paste this code into your script so you can use it later. Note that if you follow my advice from @sec-basicR R and Rstudio, you might want to actually put the `library()` calls at the beginning of your script. The third line, `View(cars04)`, just opens the newly created object in your viewer and you should probably toggle it off via the checkbox in the Import Options.

And voilà, you're done! While these steps outline a general workflow for importing data into R, the specifics can vary depending on the file type, the structure of your data, and the packages you use. If you're not diligent about using RStudio Projects and working directories, the process can become trickier-you might need to include full file paths in your import code, which can be cumbersome and error-prone, especially when sharing your scripts with others or working on network drives.

While importing data from files is the most common method in the real world, this book will primarily use datasets included in various R packages. This approach ensures that you, dear reader, can access the exact same data used in the examples without any hassle. And, as luck would have it, it just so turns out that there is a slightly different (and thankfully pre-formatted!) version of the `cars04` data in the `openintro` package (Çetinkaya-Rundel et al. 2024)! Accessing this data is as simple as loading the library:

```
library(openintro)
```

And then loading the data into memory as an object with a single command:

```
data(cars04)
```

Simply executing the command `data(cars04)` creates an object called `cars04` in your workspace. If you would prefer to give the dataset a different name than its default, you can do so with the assignment operator after the `openintro` library has been loaded.

```
cars2004 <- cars04
```

Typically, datasets loaded in this way will have a data dictionary that you can check out with the help command. Either `?cars04` or `help(cars04)` will open a help file in the Utility Pane, and you can scroll through the file there to see more about what is in the dataset. A couple important notes on this feature:

- You have to use the original dataset name, not something you assigned to it! For instance, in the above code chunk, we made an object called `cars2004` from the `cars04` data. However, `?cars2004` won't work!
- The quality of these help files can vary-some are more useful and descriptive than others!

3.2 Data Wrangling

The term "data wrangling" refers to the process of cleaning, organizing, combining, and transforming raw data into a format that will make it easier to analyze and interpret. This typically involves tasks like fixing errors, reformatting variables, creating and deleting variables, and reorganizing data through sorting, grouping, and filtering. How we wrangle the data is informed by the types of questions we wish to answer; thus, it is an essential first step in data analysis, because we need ensure that the data is appropriately structured to support our goals.

The `dplyr` (Wickham et al. 2023) package in the `tidyverse` (Wickham et al. 2019) houses most of the primary data wrangling functions. The six most important `dplyr` verbs to remember are:

- `select()` - Selects columns
- `filter()` - Filters rows

- `arrange()` - Re-orders rows
- `mutate()` - Creates new columns
- `summarize()` - summarizes variables
- `group_by()` - allows you to split-apply-recombine data along with `ungroup()`

Ok, I cheated a bit and snuck a 7th one in there. This section will cover each of these verbs, and I'll certainly be throwing in a few more along the way that I like to use (`distinct()`, `%in%`, `drop_na()`, `n()`, `case_when()`, and `if_else()` come immediately to mind). This section will by no means be a comprehensive review of the `dplyr` or the `tidyverse` and how they can be used; it should, however, provide a solid foundation upon which to build. For a deeper dive in the `tidyverse` family of packages, a good next step would be to check out the free book *R for Data Science* (Wickham and Grolemund 2017).

On their own, each verb is not that powerful. But when combined with others, they become mighty-kind of like the Infinity Gauntlet, the Deathly Hallows and/or the Care Bear Stare. As a result, this section will start with simple examples and quickly build to more complex, powerful operations.

3.2.1 The Pipe Operator

Before we can dig too deeply into these verbs, we need to learn about the pipe operator, `|>`. The keyboard shortcut in RStudio for the pipe is Ctrl + Shift + M (if this shortcut produces `%>%` instead, go to **Tools** → **Global Options** → **Code** and ensure that "Use Native Pipe Operator" is toggled on); this shortcut is worth memorizing, as you will use it a lot!

We use pipes to pass the results from one line into the next line, which creates a logical, step-by-step workflow. Typically, any data wrangling we do is going to begin with something like this:

```
cars04 |>
  # Data Wrangling!
```

This is telling R that I want to start with the `cars04` dataset and put it into whatever I'm doing on the next line. Note that this does not permanently change what is in `cars04`; it starts with `cars04`, pipes it into whatever comes next where I put the `# Data Wrangling!` placeholder, but that's it. It doesn't save it anywhere. In general, when we are data wrangling, we want to retain the changes we made to the dataset, so we need to pair this with the assignment operator.

```
cars_data <- cars04 |>
  # Data Wrangling!
```

This workflow tells R to create a new dataset called `cars_data` that will store the result of my data wrangling that follows.

Another common option is to overwrite `cars04` with our wrangled data:

```
cars04 <- cars04 |>
  # Data Wrangling!
```

This is fine, but keep in mind that once you have wrangled, you can't always unwrangle. So if whatever follows this line of code doesn't work right, you will have to redo everything up to this point, including reloading in the data and any intermediate steps taken before you got to this stage in your work. To avoid losing your original data, always consider creating a new object for the transformed data.

> Tip from the Helpdesk: A Tale of Two Pipes
>
> There are technically *two* pipe operators that are commonly used in R, `|>` and `%>%`. The first one, `|>`, is part of Base R, whereas the second one, `%>%`, is part of the `magrittr` package in `tidyverse`. I will be using the Base R version, but both do the same thing; pass the output of one line into the next line.
> If you are searching for R help on the internet, it is highly likely that you will see a lot of people using the `%>%` version. This is because the `|>` version was only added to Base R in 2021, whereas the `%>%` pipe in the tidyverse has been around a lot longer. If you see a `%>%`, just treat it like `|>`.
> However, a word of caution: `%>%` requires loading the `tidyverse`. If you use code with `%>%` without first running `library(tidyverse)`, you'll encounter an error like `Error: could not find function "%>%"`. To avoid this, either replace `%>%` with `|>` or make sure to load the `tidyverse` first.

Pipes can be, and are quite often, chained. I will explain more about the commands in this next chunk when I get to the relevant sections later in this chapter, but to illustrate the concept of chaining pipes, let's say I want a list of the cars and prices in `cars04` that had an MSRP (manufacturer's suggested retail price) over $75,000, and I want them in order from most to least expensive. This means I want to get rid of all observations that don't match my criteria (`filter`), get rid of all the other data about these cars (`select`, and put them in order (`arrange`). I will do these commands in order, piping the results of one command into the next line:

```
cars04 |>
  filter(msrp > 75000) |>
  select(name, msrp) |>
  arrange(-msrp)
```

```
# A tibble: 17 x 2
   name                                          msrp
```

```
   <chr>                                        <int>
 1 Porsche 911 GT2 2dr                         192465
 2 Mercedes-Benz CL600 2dr                     128420
 3 Mercedes-Benz SL600 convertible 2dr         126670
 4 Mercedes-Benz SL55 AMG 2dr                  121770
 5 Mercedes-Benz CL500 2dr                      94820
 6 Mercedes-Benz SL500 convertible 2dr          90520
 7 Acura NSX coupe 2dr manual S                 89765
 8 Jaguar XKR convertible 2dr                   86995
 9 Mercedes-Benz S500 4dr                       86970
10 Audi RS 6 4dr                                84600
11 Porsche 911 Carrera 4S coupe 2dr (convert)   84165
12 Jaguar XKR coupe 2dr                         81995
13 Dodge Viper SRT-10 convertible 2dr           81795
14 Porsche 911 Carrera convertible 2dr (coupe)  79165
15 Mercedes-Benz G500                           76870
16 Porsche 911 Targa coupe 2dr                  76765
17 Cadillac XLR convertible 2dr                 76200
```

When looking at code like this, the best way to read the `|>` is to say the words "and then" in your mind when you are deciphering the code. In other words, the code above says to take the `cars04` dataset, *and then* filter it, *and then* select columns, *and then* arrange the data based on the `msrp` variable.

> Tip from the Helpdesk: One Line, One Thing
>
> While it's not always possible, in general, you want each line of code to do one thing and exactly one thing. Sure, breaking it up over several lines as I did in the main text makes your code look longer. But, breaking your code in to clear, single-task steps makes it easier to read, debug, and understand. For example, that last code chunk *could* have been written like this, and it would work just fine:
>
> ```
> cars04 |> filter(msrp > 75000) |> select(name, msrp) |> arrange(-msrp)
> ```
>
> While it would work, in general, you should strive to make your code readable. Keeping the mantra of "one line, one thing" and having each line of code accomplish one task goes a long way toward making your code cleaner and your life easier.

3.2.2 Select

The `select()` function is pretty straightforward. It used to select columns-typically, it is used to focus on or remove columns from our data. There are a number of reasons we may want to do this, for example:

- Simplifying visualizations: Maybe we are creating a table for a presenta-

tion, and while the dataset has 40 variables, we only want to include 3 or 4 of the variables in our chart.

- Sensitive information: There may be personally identifiable information (PII) in a dataset, so if we want to share the data, we need to remove the PII first.
- Huge datasets: If we have a huge dataset, working with it can be cumbersome for a computer. Eliminating variables we don't need will free up memory. Similarly, email systems may not be able to cope with huge datasets, and online cloud storage is not unlimited, so having streamlined data can be useful.

The two primary ways `select()` is used is to either, a) choose a specific subset of variables, or b) to choose *all but* a specific subset of variables. To see both of these in action, let's load up the `gapminder` data from the `gapminder` package and check out the first few rows of data.

```
data(gapminder)
head(gapminder)
```

```
# A tibble: 6 x 6
  country     continent  year lifeExp      pop gdpPercap
  <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Afghanistan Asia       1957    30.3  9240934      821.
3 Afghanistan Asia       1962    32.0 10267083      853.
4 Afghanistan Asia       1967    34.0 11537966      836.
5 Afghanistan Asia       1972    36.1 13079460      740.
6 Afghanistan Asia       1977    38.4 14880372      786.
```

For the first example, let's select only the `country`, `year`, and `gdpPercap` variables:

```
gapminder |>
  select(country, year, gdpPercap) |>
  head()
```

```
# A tibble: 6 x 3
  country      year gdpPercap
  <fct>       <int>     <dbl>
1 Afghanistan  1952      779.
2 Afghanistan  1957      821.
3 Afghanistan  1962      853.
4 Afghanistan  1967      836.
5 Afghanistan  1972      740.
6 Afghanistan  1977      786.
```

Alternately, we can use the `select()` command to choose everything *but* one or more variables. For this functionality, put a `-` sign before any variables you

want to get rid of in your data. Let's get rid of the `continent` and `pop` variables.

```
gapminder |>
  select(-continent,
         -pop) |>
  head()
```

```
# A tibble: 6 x 4
  country      year lifeExp gdpPercap
  <fct>       <int>   <dbl>     <dbl>
1 Afghanistan  1952    28.8      779.
2 Afghanistan  1957    30.3      821.
3 Afghanistan  1962    32.0      853.
4 Afghanistan  1967    34.0      836.
5 Afghanistan  1972    36.1      740.
6 Afghanistan  1977    38.4      786.
```

> Tip from the Helpdesk: One Line, One Thing, Strikes Again
>
> You may have noticed a subtle difference between the two preceding sets of code: the second one follows the *One Line, One Thing* rule more closely by placing each variable in the `select()` command on its own line.
> At first glance, this may seem like overkill. But trust me, as we get into more complicated coding, this syntax will make your code much easier to read, debug, and modify!

In short, `select()` provides a simple and straightforward way to tailor your dataset to include or exclude specific variables. We will continue to make use of `select()` as we move through the list of essential `dplyr` verbs.

3.2.3 Filter

Whereas `select()` helps us choose which columns we wanted to have in our dataset, `filter()` is like a sieve for the rows in our dataset-it allows us to keep only the rows that meet specific conditions. Filtering is generally a bit more complex than selecting, because it typically involves the use of logical conditions.

3.2.3.1 Basic Comparitors

The most basic comparitors are the ones you will get the most use out of: `>`,`<`, `>=`,`<=`, `!=`, and `==`. Don't forget, for equals we use `==` instead of `=`!

Let's use a couple of these. First, let's filter the `gapminder` dataset to only include data from 1977:

```
gapminder |>
  filter(year == 1977)
```

```
# A tibble: 142 x 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1977    38.4 14880372      786.
 2 Albania     Europe     1977    68.9  2509048     3533.
 3 Algeria     Africa     1977    58.0 17152804     4910.
 4 Angola      Africa     1977    39.5  6162675     3009.
 5 Argentina   Americas   1977    68.5 26983828    10079.
 6 Australia   Oceania    1977    73.5 14074100    18334.
 7 Austria     Europe     1977    72.2  7568430    19749.
 8 Bahrain     Asia       1977    65.6   297410    19340.
 9 Bangladesh  Asia       1977    46.9 80428306      660.
10 Belgium     Europe     1977    72.8  9821800    19118.
# i 132 more rows
```

Note that if we had written `filter(year = 1977)`, the code wouldn't have worked. We would have received an error that says something like: "Error in filter(gapminder, year = 1977) : unused argument (year = 1977)."

Let's use filter to see every country with `gdpPercap` less than or equal to 300.

```
gapminder |>
  filter(gdpPercap <= 300)
```

```
# A tibble: 4 x 6
  country          continent  year lifeExp      pop gdpPercap
  <fct>            <fct>     <int>   <dbl>    <int>     <dbl>
1 Congo, Dem. Rep. Africa     2002    45.0 55379852      241.
2 Congo, Dem. Rep. Africa     2007    46.5 64606759      278.
3 Guinea-Bissau    Africa     1952    32.5   580653      300.
4 Lesotho          Africa     1952    42.1   748747      299.
```

3.2.3.2 Factor and Character Variables

When applying a condition to a character variable, you need to wrap the condition in quotation marks.

For example, let's filter the data to only include Canada:

```
gapminder |>
  filter(country == "Canada")
```

```
# A tibble: 12 x 6
   country continent  year lifeExp      pop gdpPercap
   <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
 1 Canada  Americas   1952    68.8 14785584    11367.
 2 Canada  Americas   1957    70.0 17010154    12490.
 3 Canada  Americas   1962    71.3 18985849    13462.
 4 Canada  Americas   1967    72.1 20819767    16077.
```

```
 5 Canada  Americas   1972    72.9 22284500    18971.
 6 Canada  Americas   1977    74.2 23796400    22091.
 7 Canada  Americas   1982    75.8 25201900    22899.
 8 Canada  Americas   1987    76.9 26549700    26627.
 9 Canada  Americas   1992    78.0 28523502    26343.
10 Canada  Americas   1997    78.6 30305843    28955.
11 Canada  Americas   2002    79.8 31902268    33329.
12 Canada  Americas   2007    80.7 33390141    36319.
```

Had we forgotten the quotation marks and entered `filter(country == Canada)`, R would throw an error.

Or, perhaps we want to get rid of Canada from our dataset?

```
gapminder |>
  filter(country != "Canada")
```

```
# A tibble: 1,692 x 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# i 1,682 more rows
```

Remember, spelling and capitalization count and R is super strict about it. For instance, the `gapminder` dataset codes North Korea as `Korea, Dem. Rep.`. If we want to filter the data for North Korea, `filter(country == "Korea, Dem. Rep.")` works, but none of the following would as they don't **exactly** match:

- `filter(country == "Korea, Dem Rep")`
- `filter(country == "North Korea")`
- `filter(country == "N Korea")`
- `filter(country == "korea, dem. rep.")`
- `filter(country == "Democratic People's Republic of Korea")`

When working with datasets, It is important to always check for the exact formatting of character variables, as even small discrepancies can cause your code to break in spectacular fashion.

3.2.3.3 Multiple Filters

You can include multiple filters in the same `filter()` command, separated by commas. This is equivalent to a Boolean **AND** condition, meaning that only rows that meet all the specified conditions will be included in the result. Alternately, you can use a Boolean **OR** in a filter with the `|` operator. This means that rows that match *any* of the conditions will be included in the result. Let's see how these work with some examples:

First, let's filter our data to only include European countries in 1952. And since we only have European countries in our data, let's combine this with a `select()` function; since we only have European countries, the `continent` variable is kinda meaningless:

```
gapminder |>
  filter(continent == "Europe",
         year == 1952) |>
  select(-continent)
```

```
# A tibble: 30 x 5
   country                 year lifeExp      pop gdpPercap
   <fct>                  <int>   <dbl>    <int>     <dbl>
 1 Albania                 1952    55.2  1282697     1601.
 2 Austria                 1952    66.8  6927772     6137.
 3 Belgium                 1952    68    8730405     8343.
 4 Bosnia and Herzegovina  1952    53.8  2791000      974.
 5 Bulgaria                1952    59.6  7274900     2444.
 6 Croatia                 1952    61.2  3882229     3119.
 7 Czech Republic          1952    66.9  9125183     6876.
 8 Denmark                 1952    70.8  4334000     9692.
 9 Finland                 1952    66.6  4090500     6425.
10 France                  1952    67.4 42459667     7030.
# i 20 more rows
```

Because the `filter()` command works sequentially, this is equivalent to the following:

```
gapminder |>
  filter(continent == "Europe") |>
  filter(year == 1952) |>
  select(-continent)
```

Either option is fine. For me, the more complex the filtering operation, the more I tend to lean toward the second version. For example, working with the Boolean **or** filter is a bit trickier. With an or, we are going to filter not based on whether or not **all** of a set of conditions are true, but rather based on if **any** of a set of conditions are true. For these, you need to include all the possibilities in the same filter argument, and separate the different possible qualifying conditions with a `|`. For example, let's get wrangle our data to get

2007 data for countries in either Oceania or Asia:

```
gapminder |>
  filter(continent == "Asia" | continent== "Oceania") |>
  filter(year == 2007)
```

```
# A tibble: 35 x 6
   country          continent  year lifeExp        pop gdpPercap
   <fct>            <fct>     <int>   <dbl>      <int>     <dbl>
 1 Afghanistan      Asia       2007    43.8   31889923      975.
 2 Australia        Oceania    2007    81.2   20434176    34435.
 3 Bahrain          Asia       2007    75.6     708573    29796.
 4 Bangladesh       Asia       2007    64.1  150448339     1391.
 5 Cambodia         Asia       2007    59.7   14131858     1714.
 6 China            Asia       2007    73.0 1318683096     4959.
 7 Hong Kong, China Asia       2007    82.2    6980412    39725.
 8 India            Asia       2007    64.7 1110396331     2452.
 9 Indonesia        Asia       2007    70.6  223547000     3541.
10 Iran             Asia       2007    71.0   69453570    11606.
# i 25 more rows
```

While the `|` or is powerful, when you have a lot of different possible allowable conditions, the syntax might get cumbersome. For example, if I had a list of 6 countries I wanted to collect in my data, that `filter()` line might get a bit ugly. This is a good place to use a special comparator, the `%in%`. Let's wrangle out the 1992 data for each of the 6 Central American countries in the `gapminder` data (Belize is not in the dataset):

```
gapminder |>
  filter(country %in% c("Costa Rica",
                        "El Salvador",
                        "Guatemala",
                        "Honduras",
                        "Nicaragua",
                        "Panama")) |>
  filter(year == 1992)
```

```
# A tibble: 6 x 6
  country     continent  year lifeExp     pop gdpPercap
  <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
1 Costa Rica  Americas   1992    75.7 3173216     6160.
2 El Salvador Americas   1992    66.8 5274649     4444.
3 Guatemala   Americas   1992    63.4 8486949     4439.
4 Honduras    Americas   1992    66.4 5077347     3082.
5 Nicaragua   Americas   1992    65.8 4017939     2170.
6 Panama      Americas   1992    72.5 2484997     6619.
```

In addition to being simpler, the `%in%` syntax is very useful for combining with

lists. For example, if I already have a list of Central American countries in a list called `countries`, I can simply use the `%in%` to refer to that list:

```
countries <- c("Costa Rica",
               "El Salvador",
               "Guatemala",
               "Honduras",
               "Nicaragua",
               "Panama")
gapminder |>
  filter(country %in% countries,
         year == 1992)
```

```
# A tibble: 6 x 6
  country     continent  year lifeExp     pop gdpPercap
  <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
1 Costa Rica  Americas   1992    75.7 3173216     6160.
2 El Salvador Americas   1992    66.8 5274649     4444.
3 Guatemala   Americas   1992    63.4 8486949     4439.
4 Honduras    Americas   1992    66.4 5077347     3082.
5 Nicaragua   Americas   1992    65.8 4017939     2170.
6 Panama      Americas   1992    72.5 2484997     6619.
```

Finally, another valuable use of filtering is to remove rows with missing values from your data. To see this in action, let's turn to the `cars04` data and look at the results of `summary()` to identify which variables have missing values.

```
summary(cars04)
```

```
     name           sports_car         suv            wagon
 Length:428         Mode :logical   Mode :logical   Mode :logical
 Class :character   FALSE:379       FALSE:368       FALSE:398
 Mode  :character   TRUE :49        TRUE :60        TRUE :30


  minivan          pickup        all_wheel       rear_wheel
 Mode :logical   Mode :logical   Mode :logical   Mode :logical
 FALSE:408       FALSE:404       FALSE:336       FALSE:318
 TRUE :20        TRUE :24        TRUE :92        TRUE :110


      msrp         dealer_cost        eng_size          ncyl
 Min.   : 10280   Min.   :  9875   Min.   :1.300   Min.   :-1.000
```

```
 1st Qu.: 20334   1st Qu.: 18866   1st Qu.:2.375   1st Qu.: 4.000
 Median : 27635   Median : 25295   Median :3.000   Median : 6.000
 Mean   : 32775   Mean   : 30015   Mean   :3.197   Mean   : 5.776
 3rd Qu.: 39205   3rd Qu.: 35710   3rd Qu.:3.900   3rd Qu.: 6.000
 Max.   :192465   Max.   :173560   Max.   :8.300   Max.   :12.000

    horsepwr        city_mpg        hwy_mpg          weight       wheel_base
 Min.   : 73.0   Min.   :10.00   Min.   :12.00   Min.   :1850   Min.   : 89.0
 1st Qu.:165.0   1st Qu.:17.00   1st Qu.:24.00   1st Qu.:3102   1st Qu.:103.0
 Median :210.0   Median :19.00   Median :26.00   Median :3474   Median :107.0
 Mean   :215.9   Mean   :20.09   Mean   :26.91   Mean   :3577   Mean   :108.2
 3rd Qu.:255.0   3rd Qu.:21.00   3rd Qu.:29.00   3rd Qu.:3974   3rd Qu.:112.0
 Max.   :500.0   Max.   :60.00   Max.   :66.00   Max.   :7190   Max.   :144.0
                 NA's   :14      NA's   :14      NA's   :2      NA's   :2
     length          width
 Min.   :143.0   Min.   :64.00
 1st Qu.:177.0   1st Qu.:69.00
 Median :186.0   Median :71.00
 Mean   :185.1   Mean   :71.29
 3rd Qu.:193.0   3rd Qu.:73.00
 Max.   :227.0   Max.   :81.00
 NA's   :26      NA's   :28
```

It seems that there are some missing values (`NA's`) for a few of the variables: `city_mpg`, `hwy_mpg`, `weight`, `wheel_base`, `length`, and `width`. Missing values can cause errors, so we might want to explore ways to get rid of them. When dealing missing values, `is.na()`, `!is.na()`, and `drop_na()` come in handy. Let's focus on the `hwy_mpg` variable.

We can use `is.na()` to see which rows have a missing value. For example, to see which cars we don't have data on highway MPG for, we type:

```
cars04 |>
  select(name, hwy_mpg) |>
  filter(is.na(hwy_mpg))
```

```
# A tibble: 14 x 2
   name                                hwy_mpg
   <chr>                                 <int>
 1 Mazda3 i 4dr                             NA
 2 Mitsubishi Lancer ES 4dr                 NA
 3 Mitsubishi Lancer LS 4dr                 NA
 4 Mazda3 s 4dr                             NA
 5 Mitsubishi Galant ES 2.4L 4dr            NA
 6 Mitsubishi Lancer OZ Rally 4dr auto      NA
 7 Pontiac Bonneville GXP 4dr               NA
 8 Volkswagen Phaeton 4dr                   NA
```

```
 9 Volkswagen Phaeton W12 4dr               NA
10 Dodge Viper SRT-10 convertible 2dr       NA
11 Pontiac GTO 2dr                          NA
12 Ford Excursion 6.8 XLT                   NA
13 Mitsubishi Lancer Sportback LS           NA
14 GMC Sierra HD 2500                       NA
```

This shows rows where `hwy_mpg` is missing.

If we wanted to analyze the highway MPG variable, keeping these observation in the dataset is not very helpful, and in fact may cause R to throw errors. This is where we might want to use `!is.na()`, which is the opposite of `is.na()`; it filters rows for which we *do* have data!

```
cars04 |>
  filter(!is.na(hwy_mpg))
```

```
# A tibble: 414 x 19
   name         sports_car suv   wagon minivan pickup all_wheel rear_wheel  msrp
   <chr>        <lgl>      <lgl> <lgl> <lgl>   <lgl>  <lgl>     <lgl>      <int>
 1 Chevrolet A~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      11690
 2 Chevrolet A~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      12585
 3 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      14610
 4 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      14810
 5 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      16385
 6 Dodge Neon ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13670
 7 Dodge Neon ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      15040
 8 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13270
 9 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13730
10 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      15460
# i 404 more rows
# i 10 more variables: dealer_cost <int>, eng_size <dbl>, ncyl <int>,
#   horsepwr <int>, city_mpg <int>, hwy_mpg <int>, weight <int>,
#   wheel_base <int>, length <int>, width <int>
```

Finally, `drop_na()` is like an extreme version of filtering with `!is.na()`. This is a `tidyverse` command that is like a special version of `filter()` that drops any row that has *any* missing values.

```
cars04 |>
  drop_na()
```

```
# A tibble: 387 x 19
   name         sports_car suv   wagon minivan pickup all_wheel rear_wheel  msrp
   <chr>        <lgl>      <lgl> <lgl> <lgl>   <lgl>  <lgl>     <lgl>      <int>
 1 Chevrolet A~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      11690
 2 Chevrolet A~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      12585
 3 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      14610
```

```
 4 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      14810
 5 Chevrolet C~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      16385
 6 Dodge Neon ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13670
 7 Dodge Neon ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      15040
 8 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13270
 9 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      13730
10 Ford Focus ~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      15460
# i 377 more rows
# i 10 more variables: dealer_cost <int>, eng_size <dbl>, ncyl <int>,
#   horsepwr <int>, city_mpg <int>, hwy_mpg <int>, weight <int>,
#   wheel_base <int>, length <int>, width <int>
```

Tip from the Helpdesk: Cleaning with a Toothbrush or a Flamethrower?

Be careful with deciding between `filter(!is.na())` and `drop_na()`!

- `filter(!is.na())` is like cleaning with a toothbrush-precise and targeted but takes more effort.
- `drop_na()` is the flamethrower-quick and thorough but risks removing more rows than necessary.

Before you pick your weapon of choice, always explore your data to understand where the missing values are and how much cleaning you actually need! Especially with large datasets, `drop_na()` can be particularly aggressive, so double-check which columns contain missing values before using it.

3.2.4 Arrange

The `arrange()` function allows us to sort a dataset by one or more particular columns. Arranging is often useful when creating tables, prepping data for visualization, or identifying extreme values.

The default behavior of `arrange()` is to sort numbers from lowest to highest, or strings alphabetically. For example, we can see the cars with the worst gas mileage:

```
cars04 |>
  arrange(city_mpg) |>
  select(name,
         city_mpg)
```

```
# A tibble: 428 x 2
   name                                city_mpg
   <chr>                                  <int>
 1 Hummer H2                                 10
 2 Land Rover Range Rover HSE                12
```

```
 3 Land Rover Discovery SE                  12
 4 Mercedes-Benz CL600 2dr                  13
 5 Mercedes-Benz SL600 convertible 2dr      13
 6 GMC Yukon XL 2500 SLT                    13
 7 Lincoln Navigator Luxury                 13
 8 Nissan Pathfinder Armada SE              13
 9 Lexus LX 470                             13
10 Lincoln Aviator Ultimate                 13
# i 418 more rows
```

Or we can see the cars alphabetically:

```
cars04 |>
  arrange(name)
```

```
# A tibble: 428 x 19
   name         sports_car suv   wagon minivan pickup all_wheel rear_wheel  msrp
   <chr>        <lgl>      <lgl> <lgl> <lgl>   <lgl>  <lgl>     <lgl>      <int>
 1 Acura 3.5 R~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      43755
 2 Acura 3.5 R~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      46100
 3 Acura MDX    FALSE      TRUE  FALSE FALSE   FALSE  TRUE      FALSE      36945
 4 Acura NSX c~ TRUE       FALSE FALSE FALSE   FALSE  FALSE     TRUE       89765
 5 Acura RSX T~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      23820
 6 Acura TL 4dr FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      33195
 7 Acura TSX 4~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      26990
 8 Audi A4 1.8~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      25940
 9 Audi A4 3.0~ FALSE      FALSE FALSE FALSE   FALSE  FALSE     FALSE      31840
10 Audi A4 3.0~ FALSE      FALSE FALSE FALSE   FALSE  TRUE      FALSE      34480
# i 418 more rows
# i 10 more variables: dealer_cost <int>, eng_size <dbl>, ncyl <int>,
#   horsepwr <int>, city_mpg <int>, hwy_mpg <int>, weight <int>,
#   wheel_base <int>, length <int>, width <int>
```

We can reverse the order with a - sign in front of the sorted variable. Here are the 10 most powerful cars in 2004:

```
cars04 |>
  arrange(-horsepwr) |>
  select(name,
         horsepwr) |>
  slice(1:10)
```

```
# A tibble: 10 x 2
   name                                horsepwr
   <chr>                                  <int>
 1 Dodge Viper SRT-10 convertible 2dr       500
 2 Mercedes-Benz CL600 2dr                  493
 3 Mercedes-Benz SL55 AMG 2dr               493
```

```
 4 Mercedes-Benz SL600 convertible 2dr      493
 5 Porsche 911 GT2 2dr                     477
 6 Audi RS 6 4dr                           450
 7 Volkswagen Phaeton W12 4dr              420
 8 Jaguar S-Type R 4dr                     390
 9 Jaguar XJR 4dr                          390
10 Jaguar XKR coupe 2dr                    390
```

The `slice(1:10)` line tells R to just show us the first 10 lines of data.

If we specify multiple variables for the `arrange()`, it will sort by the first variable first, and then "break ties" with the subsequent variables. Here, we arrange by both engine size and horsepower:

```
cars04 |>
  arrange(-eng_size,
          -horsepwr) |>
  select(name,
         eng_size,
         horsepwr) |>
  slice(1:10)
```

```
# A tibble: 10 x 3
   name                                eng_size horsepwr
   <chr>                                  <dbl>    <int>
 1 Dodge Viper SRT-10 convertible 2dr       8.3      500
 2 Ford Excursion 6.8 XLT                   6.8      310
 3 Volkswagen Phaeton W12 4dr               6        420
 4 Cadillac Escalade EXT                    6        345
 5 GMC Yukon XL 2500 SLT                    6        325
 6 Hummer H2                                6        316
 7 Chevrolet Silverado SS                   6        300
 8 GMC Sierra HD 2500                       6        300
 9 Chevrolet Corvette 2dr                   5.7      350
10 Chevrolet Corvette convertible 2dr       5.7      350
```

Above, we can see that the cars are arranged by `eng_size` in descending order, and ties in the `eng_size` variable are broken by the second `arrange()` variable, `horsepwr`; the six automobiles with a 6.0 liter engine are arranged from most- to least-powerful. Apparently, some 6.0 liter engines are more equal than others.

> Data Storytelling: R Crunches Numbers, You Analyze Data
>
> Remember that R has no concept of whether high or low values are "good" or "bad" - that contextual understanding comes from you!
> For example, when measuring fuel efficiency: - In the US, we use Miles Per Gallon (MPG) where *HIGHER* values indicate better efficiency, but

- In most other countries, they use Liters per 100 Kilometers (L/100km) where *LOWER* values indicate better efficiency!
If we were to convert our `city_mpg` to the international standard using the `mutate()` function we will learn about in the next section of this chapter:

```
cars04 |>
  mutate(l_per_100k = 235.21/city_mpg) |>
  arrange(l_per_100k) |>
  select(name, city_mpg, l_per_100k) |>
  slice(1:10)
```

```
# A tibble: 10 x 3
   name                                          city_mpg l_per_100k
   <chr>                                            <int>      <dbl>
 1 Honda Insight 2dr (gas/electric)                    60       3.92
 2 Toyota Prius 4dr (gas/electric)                     59       3.99
 3 Honda Civic Hybrid 4dr manual (gas/electric)        46       5.11
 4 Volkswagen Jetta GLS TDI 4dr                        38       6.19
 5 Honda Civic HX 2dr                                  36       6.53
 6 Toyota Echo 2dr manual                              35       6.72
 7 Toyota Echo 4dr                                     35       6.72
 8 Toyota Echo 2dr auto                                33       7.13
 9 Honda Civic DX 2dr                                  32       7.35
10 Honda Civic LX 4dr                                  32       7.35
```

The cars would appear in the same order, but we'd be sorting from lowest to highest L/100km instead of highest to lowest MPG. The story remains the same, but how we tell it changes based on the audience and context. This reminder applies to all data analysis: GDP, test scores, temperatures - the meaning of "good" or "bad" values comes from your domain knowledge, not from R.

Generally speaking, `arrange()` performs a couple useful tasks. The first is for aesthetics; sorting tables alphabetically or from biggest to smallest in a certain variable adds visual appeal and readability for your audience. The second is for exploring your data and uncovering trends, like identifying the most expensive or least efficient car-or, for example, finding out which car had the most powerful engine in 2004; the Dodge Viper SRT-10 convertible 2dr.

3.2.4.1 Slice: Where Arrange Meets Filter

Let's take a bit of a deeper dive into the `slice` family of functions, which is what you would get if arrange and filter had an unholy data wrangling baby. There's a whole family of `slice_` functions that elegantly combine the tasks of arranging and filtering:

- `slice_head(n = 5)` grabs the first 5 observations
- `slice_tail(n = 5)` takes the last 5 observations

- `slice_min(var, n = 5)` selects the 5 rows with the lowest values of variable `var`
- `slice_max(var, n = 5)` selects the 5 rows with the highest values of variable `var`

As you can see, `slice_max()` and `slice_min()` combine the tasks of arranging and filtering/slicing into one line! Let's see a couple examples in action:

The 7 most powerful cars:

```
cars04 |>
  slice_max(horsepwr, n = 7) |>
  select(name, horsepwr)
```

```
# A tibble: 7 x 2
  name                                horsepwr
  <chr>                                  <int>
1 Dodge Viper SRT-10 convertible 2dr       500
2 Mercedes-Benz CL600 2dr                  493
3 Mercedes-Benz SL55 AMG 2dr               493
4 Mercedes-Benz SL600 convertible 2dr      493
5 Porsche 911 GT2 2dr                      477
6 Audi RS 6 4dr                            450
7 Volkswagen Phaeton W12 4dr               420
```

The 9 least expensive cars:

```
cars04 |>
  slice_min(msrp, n = 9) |>
  select(name, msrp)
```

```
# A tibble: 9 x 2
  name                       msrp
  <chr>                     <int>
1 Kia Rio 4dr manual        10280
2 Hyundai Accent 2dr hatch  10539
3 Toyota Echo 2dr manual    10760
4 Saturn Ion1 4dr           10995
5 Kia Rio 4dr auto          11155
6 Toyota Echo 4dr           11290
7 Toyota Echo 2dr auto      11560
8 Chevrolet Aveo 4dr        11690
9 Hyundai Accent GL 4dr     11839
```

These functions allow you to accomplish in one step what would otherwise require both an arrange() and a slice() or head() function, making your code more concise and readable.

For more flexibility, the original slice() function lets you select any arbitrary

positions. Before, I used `slice(1:10)` to grab the first 10 rows of the sorted dataset, but I could have grabbed any arbitrary set of rows. For example, here are the cars in rows 26-29 of the dataset:

```
cars04 |>
  slice(26:29) |>
  select(name)
```

```
# A tibble: 4 x 1
  name
  <chr>
1 Kia Spectra GSX 4dr hatch
2 Mazda3 i 4dr
3 Mini Cooper
4 Mitsubishi Lancer ES 4dr
```

3.2.5 Mutate

Mutate allows us to create a new column within a dataset or overwrite an existing one. This can be used to rescale variables, change data formatting, or create all-new measures. This one is pretty powerful; we will start with some simple mutates, and work our way up to more complex ones.

Let's start with a simple rescaling. Say, for example, we would prefer to measure our engine size variable `eng_size` in cubic centimeters as opposed to liters. As there are 1,000 ccs in a liter, we can:

```
cars04 <- cars04 |>
  mutate(eng_size_cc = eng_size * 1000)

cars04 |>
  select(name, eng_size, eng_size_cc) |>
  slice_head(n = 10)
```

```
# A tibble: 10 x 3
   name                        eng_size eng_size_cc
   <chr>                          <dbl>       <dbl>
 1 Chevrolet Aveo 4dr               1.6        1600
 2 Chevrolet Aveo LS 4dr hatch      1.6        1600
 3 Chevrolet Cavalier 2dr           2.2        2200
 4 Chevrolet Cavalier 4dr           2.2        2200
 5 Chevrolet Cavalier LS 2dr        2.2        2200
 6 Dodge Neon SE 4dr                2          2000
 7 Dodge Neon SXT 4dr               2          2000
 8 Ford Focus ZX3 2dr hatch         2          2000
 9 Ford Focus LX 4dr                2          2000
10 Ford Focus SE 4dr                2          2000
```

We could have, in principle, overwritten the old variable by specifying `mutate(eng_size = eng_size * 1000)`. However, adding a new column ensures that the original data remains untouched, allowing flexibility for further analysis.

We can also perform calculations with multiple variables. Suppose we want to calculate which car has the largest MSRP markup over dealer invoice in percentage terms? This means we want to calculate:

$$Markup = \frac{MSRP - DealerCost}{DealerCost}$$

```
cars04 <- cars04 |>
  mutate(markup = (msrp - dealer_cost)/dealer_cost)

cars04 |>
  slice_max(markup, n = 10) |>
  select(name, markup)
```

```
# A tibble: 10 x 2
   name                                          markup
   <chr>                                          <dbl>
 1 Porsche 911 Carrera 4S coupe 2dr (convert)     0.166
 2 Lexus LX 470                                   0.148
 3 Lexus SC 430 convertible 2dr                   0.148
 4 Lexus LS 430 4dr                               0.148
 5 Lexus GS 430 4dr                               0.147
 6 Lexus GX 470                                   0.147
 7 Porsche Boxster convertible 2dr                0.145
 8 Porsche Boxster S convertible 2dr              0.144
 9 Porsche 911 Targa coupe 2dr                    0.144
10 Porsche 911 Carrera convertible 2dr (coupe)    0.144
```

Looks like Porsche and Lexus had pretty big markups, between 14% and 17%! Another name for a markup is a profit margin; sometimes math has practical uses!

> Tip from the Helpdesk: Aunt Sally Who Needs to be Constantly Excused
>
> When using `mutate()` (or any other calculations in R), don't forget the order of operations-PEMDAS: Parentheses, Exponents, Multiplication/Division (from left to right), and Addition/Subtraction (from left to right). If your math isn't mathing as expected, double-check your parentheses to ensure everything is grouped the way you want!
> For example, compare these two calculations:

```
cars04 |>
  mutate(markup = (msrp - dealer_cost)/dealer_cost) |>
  mutate(markup2 = msrp - dealer_cost/dealer_cost) |>
  select(name, msrp, dealer_cost, markup, markup2) |>
  slice_max(markup, n = 10)
```

```
# A tibble: 10 x 5
   name                                      msrp dealer_cost markup markup2
   <chr>                                    <int>       <int>  <dbl>   <dbl>
 1 Porsche 911 Carrera 4S coupe 2dr (convert)  84165       72206  0.166   84164
 2 Lexus LX 470                              64800       56455  0.148   64799
 3 Lexus SC 430 convertible 2dr              63200       55063  0.148   63199
 4 Lexus LS 430 4dr                          55750       48583  0.148   55749
 5 Lexus GS 430 4dr                          48450       42232  0.147   48449
 6 Lexus GX 470                              45700       39838  0.147   45699
 7 Porsche Boxster convertible 2dr           43365       37886  0.145   43364
 8 Porsche Boxster S convertible 2dr         52365       45766  0.144   52364
 9 Porsche 911 Targa coupe 2dr               76765       67128  0.144   76764
10 Porsche 911 Carrera convertible 2dr (coupe) 79165       69229  0.144   79164
```

The first calculation is the same as I did in the text, so it correctly calculates:

$$Markup = \frac{MSRP - DealerCost}{DealerCost}$$

The second calculation ignores the fact that PEMDAS exists, and will result in R prioritizing division over subtraction because there no parentheses in the numerator. Thus, the second markup equation calculation will calculate:

$$Markup = MSRP - \frac{DealerCost}{DealerCost}$$

So while the first calculation accurately calculates the dealer profit margin, the second one simplifies to:
$Markup = MSRP - 1$
Clearly, not what I am going for!
The big takeaway here: Aunt Sally is your best friend when working with equations in R. You can't afford to forget PEMDAS! A little extra care upfront saves you from incorrect calculations-and major headaches-later on!

Let's ramp this up a bit. We will use `mutate()` to do something a bit more sophisticated and make a new variable that denotes whether or not a car is domestic (to the US market) or foreign. Looking at the `name` variable, it seems that the vehicle name always starts with the make as the first word, and the model and trim line after that. We can exploit this pattern! We will proceed in

two steps.

- First, we will use the `word()` command out of the `stringr` package (Wickham 2023)-`stringr` is part of the `tidyverse` and is useful for working with character (sometimes called string) variables-to create a variable that has the car make.
- Second, we will use the new car make variable to identify foreign vs. domestic cars, and create a new variable from that.

Let's start by creating a new variable called `make` which is the first word in the `name` variable:

```
cars04 <- cars04 |>
  mutate(make = word(name, 1))
```

After doing something like this, it is useful to double-check that we got the results we want:

```
cars04 |>
  select(name, make) |>
  slice(1:10)
```

```
# A tibble: 10 x 2
   name                          make
   <chr>                         <chr>
 1 Chevrolet Aveo 4dr            Chevrolet
 2 Chevrolet Aveo LS 4dr hatch   Chevrolet
 3 Chevrolet Cavalier 2dr        Chevrolet
 4 Chevrolet Cavalier 4dr        Chevrolet
 5 Chevrolet Cavalier LS 2dr     Chevrolet
 6 Dodge Neon SE 4dr             Dodge
 7 Dodge Neon SXT 4dr            Dodge
 8 Ford Focus ZX3 2dr hatch      Ford
 9 Ford Focus LX 4dr             Ford
10 Ford Focus SE 4dr             Ford
```

So far, so good. Let's take a deeper look by using the the `distinct()` command; `distinct()` is a special kind of `filter()` that drops any rows that are identical to other rows in the data. So the combination of `select(make)` followed by `distinct()` will give me a list of every `make` in the data!

```
cars04 |>
  select(make) |>
  distinct() |>
  arrange(make)
```

```
# A tibble: 42 x 1
   make
   <chr>
```

```
 1 Acura
 2 Audi
 3 BMW
 4 Buick
 5 CMC
 6 Cadillac
 7 Chevrolet
 8 Chrvsler
 9 Chrysler
10 Dodge
# i 32 more rows
```

Looking at just this sample, we can already spot two likely typos: "CMC" and "Chrvsler". The complete list contains 42 unique makes that we ought to scrutinize. It's always a good idea to doublecheck your work along the way, and the fact that we've already found something fishy suggests that we are likely to find more issues.

Remember the earlier tip about *R Crunches Numbers, You Analyze Data?* This is a perfect example-R has no idea that these entries are errors. It's our knowledge of the real world from which these data come that allows us to identify these inconsistencies.

If you are working through this example in R yourself, I encourage you to look through the 42 makes we created to look for other issues. I'm going to filter for the values that seem suspicious to me:

```
cars04 |>
  filter(make %in% c("CMC", "Chrvsler", "Mazda3", "Mazda6", "Land")) |>
  select(name, make)
```

```
# A tibble: 8 x 2
  name                         make
  <chr>                        <chr>
1 Mazda3 i 4dr                 Mazda3
2 Mazda3 s 4dr                 Mazda3
3 Mazda6 i 4dr                 Mazda6
4 Chrvsler PT Cruiser GT 4dr   Chrvsler
5 CMC Yukon 1500 SLE           CMC
6 Land Rover Range Rover HSE   Land
7 Land Rover Discovery SE      Land
8 Land Rover Freelander SE     Land
```

It certainly would seem that:

- CMC should say GMC
- Chrvsler should say Chrysler
- The two Mazda variants should simply be Mazdas.
- Land should be Land Rover

As our goal is to create a foreign/domestic variable, we can either:

- Fix the errors in the original data,
- Change the way we construct the `make` variable, or
- Work around the errors.

Let's change our `make` variable, as it is generally preferable to *not* edit original data. We will make use of a couple new functiions, `mutate()` and `case_when()`.

```
cars04 <- cars04 |>
  mutate(make = case_when(
    make == "Chrvsler" ~ "Chrysler",
    make == "CMC" ~ "GMC",
    make == "Mazda3" ~ "Mazda",
    make == "Mazda6" ~ "Mazda",
    make == "Land" ~ "Land Rover",
    TRUE ~ make
  ))
```

This is probably the trickiest line of code we've encountered so far, so let's break it down a bit. This code corrects typos and standardizes values in the make column of the `cars04` dataset using the `case_when()` function within `mutate()`. It evaluates each row in the make column and applies specific replacements based on defined conditions. For example, if a row contains `"Chrvsler"`, it is replaced with `"Chrysler"`, while `"CMC"` is corrected to `"GMC"`. Similarly, rows with `"Mazda3"` or `"Mazda6"` are standardized to `"Mazda"`. Any value not explicitly matched by these conditions is left unchanged, thanks to the `TRUE ~ make` clause at the end, which acts as a catch-all to preserve unmatched values. Basically, how this line works is that if R looks the value of `make` in a line and doesn't see one of the values we are looking for, but it does see *something* (e.g. it is `TRUE` that `make` has a value), then it will replace the value of `make` with what is already there!

We can double-check our `make` variable to make sure we got the result we wanted:

```
cars04 |>
  select(make) |>
  distinct() |>
  arrange(make)
```

```
# A tibble: 38 x 1
   make
   <chr>
 1 Acura
 2 Audi
 3 BMW
 4 Buick
 5 Cadillac
 6 Chevrolet
```

```
 7 Chrysler
 8 Dodge
 9 Ford
10 GMC
# i 28 more rows
```

This looks fixed!

Now that we have our `make` variable, we can create a list of brands that are domestic and create our new variable:

```
domestic_cars <- c("GMC",
                   "Chevrolet",
                   "Ford",
                   "Saturn",
                   "Dodge",
                   "Jeep",
                   "Pontiac",
                   "Buick",
                   "Chrysler",
                   "Oldsmobile",
                   "Lincoln",
                   "Cadillac",
                   "Mercury",
                   "Hummer")

# If above, I instead opted to work around the errors, I could have included "CMC" and "Chrvsler"

cars04 <- cars04 |>
  mutate(origin = if_else(make %in% domestic_cars, "Domestic", "Foreign"))
```

In this code, we start by creating a list called `domestic_cars` that are all the brands in the dataset that were GM, Ford, or Chrysler makes back in 2004. Then, we use the `if_else()` function to create our `origin` variable; `if_else()` looks to see if the make of each car is included in the list of `domestic_cars`, and if it is it returns the first value, and if not it returns the second value.

In this case, for a car make like "Chevrolet", when R checks if "Chevrolet" is in the `domestic_cars` list and finds that it is, R will assign "Domestic" to the origin variable for that row. On the other hand, consider a car make like "Toyota". R will check to see if "Toyota" is in the `domestic_cars` list. Nope! Therefore, R assigns "Foreign" to the origin variable for that row.

Again, we should double-check to see if this worked correctly:

```
cars04 |>
  select(make, origin) |>
  distinct()
```

```
# A tibble: 38 x 2
   make       origin
   <chr>      <chr>
 1 Chevrolet  Domestic
 2 Dodge      Domestic
 3 Ford       Domestic
 4 Honda      Foreign
 5 Hyundai    Foreign
 6 Kia        Foreign
 7 Mazda      Foreign
 8 Mini       Foreign
 9 Mitsubishi Foreign
10 Nissan     Foreign
# i 28 more rows
```

And there we go, it looks like we nailed it! This example took us through a complete data wrangling workflow - from identifying problems, to investigating them, to implementing a solution, and finally verifying our results. While it was certainly more complex than our earlier examples, this complexity mirrors real-world data challenges. By combining multiple tidyverse functions like `mutate()`, `case_when()`, `distinct()`, and `if_else()`, we've demonstrated how these tools work together to transform messy data into structured, useful information. The thought process we followed here - carefully examining data, making decisions based on domain knowledge, implementing changes step by step, and verifying results - is the essence of effective data wrangling.

3.2.6 Summarize

The `summarize()` function is used to collapse a dataset into a smaller dataset of summary statistics. To get our heads around the basics of summarize, let's use it to create some summary statistics of our `cars04` data.

First, what are some common things we want to get summary statistics of. Measures of central tendency, range, sums, counts, things like that. Here is a short list of the functions I most commonly nest inside of `summarize()`:

- Central Tendency: `mean()`, `median()`, `weighted.mean()`
- Spread: `sd()`
- Range: `max()`, `min()`
- Count: `n()`, `n_distinct()`

Let's use some of these functions to summarize the cars04() dataset. The general syntax of `summarize()` is `summarize(new_var = function)`, so let's create some summary statistics:

```
cars04 |>
  summarize(makes = n_distinct(make),
            count = n(),
```

```
            avg_hp = mean(horsepwr),
            avg_mpg = mean(city_mpg))
```

```
# A tibble: 1 x 4
  makes count avg_hp avg_mpg
  <int> <int>  <dbl>   <dbl>
1    38   428   216.      NA
```

So we can see that there are 38 different makes of cars in the dataset, 428 different cars in total, an average horsepower of 215.89, but I see a problem with the avg_mpg function – it is spitting out an `NA`! This is happening because there are some cars for which we do not have data on their fuel efficiency:

```
cars04 |>
  filter(is.na(city_mpg)) |>
  select(name, city_mpg)
```

```
# A tibble: 14 x 2
   name                                city_mpg
   <chr>                                  <int>
 1 Mazda3 i 4dr                              NA
 2 Mitsubishi Lancer ES 4dr                  NA
 3 Mitsubishi Lancer LS 4dr                  NA
 4 Mazda3 s 4dr                              NA
 5 Mitsubishi Galant ES 2.4L 4dr             NA
 6 Mitsubishi Lancer OZ Rally 4dr auto       NA
 7 Pontiac Bonneville GXP 4dr                NA
 8 Volkswagen Phaeton 4dr                    NA
 9 Volkswagen Phaeton W12 4dr                NA
10 Dodge Viper SRT-10 convertible 2dr        NA
11 Pontiac GTO 2dr                           NA
12 Ford Excursion 6.8 XLT                    NA
13 Mitsubishi Lancer Sportback LS            NA
14 GMC Sierra HD 2500                        NA
```

To work around this, we will add the `na.rm = TRUE` option to our mean function.

```
cars04 |>
  summarize(makes = n_distinct(make),
            count = n(),
            avg_hp = mean(horsepwr),
            avg_mpg = mean(city_mpg, na.rm = TRUE))
```

```
# A tibble: 1 x 4
  makes count avg_hp avg_mpg
  <int> <int>  <dbl>   <dbl>
1    38   428   216.    20.1
```

In general, if you get an `NA` response to a numerical function that you think should work, `NA`'s in the dataset are often the culprit!

Admittedly, `summarize()` is only moderately useful on its own. But when you combine it with grouping techniques (next subsection), it goes hard. So let's jump right into that.

3.2.7 Group_by and Ungroup

The `group_by()` function allows us to create virtual splits of a dataset by segmenting it into multiple smaller data sets. It is typically used in conjunction with other `dplyr` commands to execute functions on each data segment, most typically `summarize()` and `mutate()`. Depending on what you do with the subsetted data, you may need to `ungroup()` to remove the grouping effect. I admit that this description may seem a bit abstract, so let's do a few practical examples, starting with some simple ones and working up to more complicated ones, to see just how powerful `group_by()` can be.

Consider our modified `cars04` dataset in which we constructed the `make` variable above. Let's say we want to get a sense of which car brands tend to be expensive and which tend to be less expensive. One way to accomplish this would be to calculate the `mean()` of `msrp` for each value of `make`, which is easily done with the following code:

```
cars04 |>
  group_by(make) |>
  summarize(mean_msrp = mean(msrp)) |>
  arrange(-mean_msrp)
```

```
# A tibble: 38 x 2
   make          mean_msrp
   <chr>             <dbl>
 1 Porsche          83565
 2 Jaguar           61580.
 3 Mercedes-Benz    60657.
 4 Cadillac         50474.
 5 Hummer           49995
 6 Land Rover       45832.
 7 Lexus            44215.
 8 Audi             43308.
 9 BMW              43285.
10 Acura            42939.
# i 28 more rows
```

Pretty slick for 4 lines of code, eh? Let's think through what the code is doing. We start by piping `cars04` into the `group_by()` function; the way I conceptualize this is that we are taking the dataset of 428 cars and virtually breaking it up into 38 separate datasets. Anything that happens next will be done to

every single one of those datasets individually. The next line is to `summarize()`, which means R is going to make a new data set out of whatever commands we put in the summarize. As the summary we want is `mean(msrp)`, R is going to go through each of the 38 datasets, one for each car make, and calculate `mean(msrp)` in each and every single one of them. The result will be a new variable called `mean_msrp` which contains the average for each `make`. The `make` variable also winds up in the new data set, so we know average mean comes from which `make`. The final line is optional-I just was curious which `make` would come out on top!

By combining `group_by()` and `summarize()` we have the ability to create a highly customizable table of summary statistics for our data. Let's add a bit onto this code, just to stretch our legs a bit:

```
cars04 |>
  group_by(make) |>
  summarize(mean_msrp = mean(msrp),
            most_expensive = max(msrp),
            least_expensive = min(msrp),
            models = n(),
            domestic = first(origin)) |>
  arrange(-mean_msrp)
```

```
# A tibble: 38 x 6
   make          mean_msrp most_expensive least_expensive models domestic
   <chr>             <dbl>          <int>           <int>  <int> <chr>
 1 Porsche           83565          192465           43365      7 Foreign
 2 Jaguar            61580.          86995           29995     12 Foreign
 3 Mercedes-Benz     60657.         128420           26060     26 Foreign
 4 Cadillac          50474.          76200           30835      8 Domestic
 5 Hummer            49995           49995           49995      1 Domestic
 6 Land Rover        45832.          72250           25995      3 Foreign
 7 Lexus             44215.          64800           31045     11 Foreign
 8 Audi              43308.          84600           25940     19 Foreign
 9 BMW               43285.          73195           28495     20 Foreign
10 Acura             42939.          89765           23820      7 Foreign
# i 28 more rows
```

Now, in addition to including the `mean()` of MSRP, we have included the maximum and minimum, along with the total number of car styles from each manufacturer in the data set. We also did something a little fancy with the last line of code; I wanted to include the foreign/domestic designation in my results, but `summarize()` requires a function. Since I know that the value for `origin` is constant within each `make` (e.g. all Hondas are foreign, all Fords are domestic), I just grabbed the `first()` value!

In addition to `summarize()`, `mutate()` is also a really useful function to use with `group_by()`. Let's say we are curious as to which cars are the most fuel efficient

given their body type. In other words, while we expect sports cars and SUVs to have worse MPG than sedans, we may be interested in knowing which sports cars and SUVs have the best fuel efficiency. To start, let's use `case_when()` in a similar way to above to turn the logical variables of `sports_car`, `suv`, and so forth into a single variable called `bodystyle`.

```
cars04 <- cars04 |>
  mutate(bodystyle = case_when(
    sports_car == TRUE ~ "sports car",
    suv == TRUE ~ "suv",
    wagon == TRUE ~ "wagon",
    minivan == TRUE ~ "minivan",
    pickup == TRUE ~ "pickup",
    TRUE ~ "sedan"
  ))
```

This code works in a very similar way to the code above when we cleaned up the `make` variable. Now, we want to take our data and `group_by(bodystyle)` and look for cars with `city_mpg` that is at least 25% higher than the average:

```
cars04 <- cars04 |>
  group_by(bodystyle) |>
  mutate(avg_mpg = mean(city_mpg, na.rm = TRUE)) |>
  mutate(rel_mpg = city_mpg/avg_mpg) |>
  mutate(efficient = case_when(
    rel_mpg > 1.25 ~ "efficient",
    rel_mpg < 0.75 ~ "gas guzzler",
    TRUE ~ "neither")) |>
  ungroup()
```

Before taking a look at the results, let's break down this code.

- First, `group_by(bodystyle)` splits our data into separate groups - one for each body style. This means whatever we do next happens separately for sedans, SUVs, sports cars, and so on.
- Next, we create a new variable `avg_mpg` that calculates the average city MPG for each body style. The `na.rm = TRUE` part tells R to ignore any missing values when calculating these averages. Without this, a single NA would ruin our calculations! I know this because my first draft of this code forgot about this and I was baffled by my lack of results.
- Then we create `rel_mpg` by dividing each car's MPG by the average for its body style. This gives us a ratio where 1.0 means "exactly average for its class," 1.5 means "50% better than average," and 0.5 means "half as efficient as average."
- After that, we use `case_when()` to label each car based on this ratio. Cars that are at least 25% more efficient than their class average get labeled

"efficient," while cars that are at least 25% less efficient get the unfortunate "gas guzzler" label, though I suspect that Hummer owners view that as a badge of honor! Everything in between is "neither."

- Finally, we `ungroup()` to remove the grouping structure

The neat thing about this approach is that we're comparing apples to apples. A Honda Civic might be more fuel-efficient than any SUV in absolute terms, but that's not a fair comparison. This way, we can identify the most efficient vehicles within each category.

> **Tip from the Helpdesk: The Danger of Groupthink**
>
> Why am I ungrouping here but not before? It's because before I was using `summarize()`, but now I'm using `mutate()`. This is an important distinction!
> When you use `summarize()` with a grouped dataset, R automatically removes one level of grouping for you after it creates the summary. This makes sense - if you're calculating averages by make, your new dataset is already at the make level, so the grouping has done its job.
> But `mutate()` is sneaky - it doesn't touch your grouping at all! It adds new columns while keeping all your rows and maintaining whatever groups you created. If you forget to `ungroup()` after using `mutate()`, any calculations you do later will still respect that grouping, which can lead to some pretty confusing results.
> I've been burned by this more times than I care to admit. I'll be looking at your results thinking "what the heck is going on here?!" only to realize I'm still grouped from five steps ago.
> Bottom line: After using `group_by()` with `mutate()`, add an `ungroup()` at the end unless you specifically want to keep working with those groups. Your future self will thank you!

Now that we've done the work, let's take a look. Which cars are fuel efficient for their bodystyle, which are the gas guzzlers?

```
cars04 |>
  select(name, bodystyle, city_mpg, avg_mpg, rel_mpg, efficient) |>
  filter(efficient == "efficient")
```

```
# A tibble: 40 x 6
   name                        bodystyle city_mpg avg_mpg rel_mpg efficient
   <chr>                       <chr>        <int>   <dbl>   <dbl> <chr>
 1 Chevrolet Aveo 4dr          sedan           28    21.8    1.29 efficient
 2 Chevrolet Aveo LS 4dr hatch sedan           28    21.8    1.29 efficient
 3 Dodge Neon SE 4dr           sedan           29    21.8    1.33 efficient
 4 Dodge Neon SXT 4dr          sedan           29    21.8    1.33 efficient
 5 Honda Civic DX 2dr          sedan           32    21.8    1.47 efficient
```

```
 6 Honda Civic HX 2dr          sedan           36    21.8    1.65 efficient
 7 Honda Civic LX 4dr          sedan           32    21.8    1.47 efficient
 8 Hyundai Accent 2dr hatch    sedan           29    21.8    1.33 efficient
 9 Hyundai Accent GL 4dr       sedan           29    21.8    1.33 efficient
10 Hyundai Accent GT 2dr hatch sedan           29    21.8    1.33 efficient
# i 30 more rows
```

```
cars04 |>
  select(name, bodystyle, city_mpg, avg_mpg, rel_mpg, efficient) |>
  filter(efficient == "gas guzzler")
```

```
# A tibble: 12 x 6
   name                            bodystyle city_mpg avg_mpg rel_mpg efficient
   <chr>                           <chr>        <int>   <dbl>   <dbl> <chr>
 1 Audi S4 Quattro 4dr             sedan           14    21.8   0.643 gas guzz~
 2 Mercedes-Benz C32 AMG 4dr       sedan           16    21.8   0.735 gas guzz~
 3 Mercedes-Benz CL500 2dr         sedan           16    21.8   0.735 gas guzz~
 4 Mercedes-Benz CL600 2dr         sedan           13    21.8   0.597 gas guzz~
 5 Mercedes-Benz E500 4dr          sedan           16    21.8   0.735 gas guzz~
 6 Mercedes-Benz S500 4dr          sedan           16    21.8   0.735 gas guzz~
 7 Mercedes-Benz SL600 convertible~ sports c~      13    18.6   0.699 gas guzz~
 8 Hummer H2                       suv             10    16.2   0.617 gas guzz~
 9 Land Rover Range Rover HSE      suv             12    16.2   0.741 gas guzz~
10 Land Rover Discovery SE         suv             12    16.2   0.741 gas guzz~
11 Audi S4 Avant Quattro           wagon           15    21.0   0.715 gas guzz~
12 Infiniti FX45                   wagon           15    21.0   0.715 gas guzz~
```

3.3 Wrapping Up

The focus of this chapter was twofold - you should have learned how to get data into R, and some of the basics of data manipulation and data wrangling using the tidyverse suite of functions.

Throughout this chapter, we've relied heavily on the tidyverse, and for good reason. Though strictly speaking, much of what we've discussed in this section can be done via Base R as well, most people coding in the R world use the `tidyverse` for data wrangling. In general, the `tidyverse` family of packages has an internal logic that better vibes with how humans think because the code is written roughly in the same order as the actions being performed.

To show you what I mean, here are two code snippets that both do the exact same thing-they calculate the weighted mean of GDP per capita by continent in 2007 in the `gapminder` dataset. First, the `tidyverse` version:

```
gapminder |>
  filter(year == 2007) |>
  group_by(continent) |>
```

```
  summarize(gdp = weighted.mean(gdpPercap, w = pop))
```

```
# A tibble: 5 x 2
  continent    gdp
  <fct>      <dbl>
1 Africa     2561.
2 Americas  21603.
3 Asia       5432.
4 Europe    25244.
5 Oceania   32885.
```

This uses the typical tidyverse thought process:

- Pipe the data in
- Filter the data for the relevant year
- Split the data into subgroups by continent
- Summarize the data with the statistic you want

Now, compare this to what I think is likely the simplest Base R method:

```
gap_07 <- gapminder[gapminder$year == 2007, ]
result <- data.frame(continent = character(), gdp = numeric(), stringsAsFactors = FALSE)
continents <- unique(gap_07$continent)
for (cont in continents) {
  subset_data <- gap_07[gap_07$continent == cont, ]
  weighted_gdp <- weighted.mean(subset_data$gdpPercap, subset_data$pop)
  result <- rbind(result, data.frame(continent = cont, gdp = weighted_gdp))
}

result
```

```
  continent       gdp
1      Asia  5432.372
2    Europe 25244.048
3    Africa  2560.930
4  Americas 21602.746
5   Oceania 32884.555
```

Here's the basic logic of the above code:

- Line 1 filters the data to the relevant year.
- Line 2 initializes an empty data frame where the results will be stored.
- Line 3 creates a list of each of the continents in the dataset.
- Lines 4-8 loop over the continents in the list created in Line 3, so the following steps are performed 5 times, once for each continent.
  - Line 4 initializes what is being looped over.
  - Line 5 subsets the data subset for that continent.
  - Line 6 calculates the weighted mean for the continent.

  - Line 7 adds the continent name and weighted mean that was just calculated to the data frame created in Line 2.
  - Line 8 closes the loop.
- Line 9 prints the results.

This frankly makes my brain hurt. The tidyverse accomplishes this in a logical sequence-`filter()`, `group_by()`, `summarize()`, `select()`. With Base R, you're juggling data subsets, loops, and manual assignment. While both work, one is clearly more intuitive for everyday data analysis.

By now, you should have a solid foundation in importing data and using tidyverse tools like `select()`, `filter()`, `mutate()`, `group_by()`, and `summarize()` to begin exploring and manipulating datasets. In the next chapters, we'll build on these skills to perform more sophisticated analyses.

3.4 End of Chapter Exercises

Cars Data

1. Country of Origin Classification: Using the cars04 dataset and following the approach we used to classify cars as foreign or domestic, create a new variable that identifies the country of origin for each car. Classify cars into major automobile producing countries: US, Japan, Germany, South Korea, Italy, Sweden, UK, etc. Then create a summary table showing the average price, fuel efficiency, and horsepower by country of origin.

2. Price Efficiency by Body Style: Similar to how we identified fuel-efficient cars within each body style, identify which cars are the best value in their category. Create a new variable that compares each car's price to the average price for its body style. Classify cars as "bargain" (at least 25% below average price), "premium" (at least 25% above average price), or "average". Which cars offer the most features or performance for below-average prices in their category?

3. Car Market Segmentation: Using the cars04 dataset, create a new categorical variable that classifies cars into market segments based on both price and body style (e.g., "Budget Sedan", "Luxury SUV", "Mid-range Sports Car"). Then provide summary statistics for each segment, including the number of models, average price, and average fuel efficiency.

4. Rethinking Fuel Efficiency: Consider the results of the analysis above where we identified the gas guzzlers. Based on these most recent results – if you were to answer the question of which car in the `cars04` dataset is the least fuel efficient, which one would you pick? Is it the Hummer, with its whopping 10 MPG? Or is a better case to be made for the Mercedes-Benz CL600 2dr, which is tied with about a dozen other cars in the dataset for 4th worst? Or something else? There is no right answer here - what's more important is how you think through this question.

Gapminder Data

5. Population Distribution: Using the gapminder dataset, calculate what percentage of the world's population lived on each continent for each of the years in the dataset. Create a summary showing how these percentages have shifted over time from 1952 to 2007.

6. Demographic Shifts: Using the gapminder dataset, calculate the ratio of GDP per capita between the richest and poorest countries in each continent for each time period. Has global inequality increased or decreased within each continent between 1952 and 2007?

7. Economic Growth: Using the gapminder dataset, identify which countries experienced the greatest economic growth between 1952 and 2007. Calculate the percentage increase in GDP per capita for each country over this period, and display the top 15 countries with the highest growth rates. Include their starting and ending GDP values, and their continent.

Chapter 4

Literate Programming

Have you ever been working on a report where you needed to move stuff from Excel into Word or PowerPoint? It's the worst, right? Tables that looked perfect in Excel suddenly lose all their formatting. Charts need to be screenshot, cropped, and pasted-and they still look awful, because the fonts don't match, the sizes are slightly off, the pictures are fuzzy, etc.

And let's talk about what happens when your data changes. You know the drill: open Excel, update your numbers, regenerate your chart, take a new screenshot, crop it again (slightly differently this time, of course), then hunt through your document to find all the places where you need to paste the new version. Oh, and don't forget to update any text that references specific numbers from your analysis. "The average was 42.3" now needs to be "The average was 43.7"… but did you catch all of them? Good luck with that.

I've been there too. I once got so frustrated with this back-and-forth that I tried writing an entire research paper in Excel. (Spoiler alert: Excel is not a word processor. It did not go well.) I've also spent hours formatting tables in Word, only to realize that not all of my tables were consistent in their rounding conventions (some rounded to two decimal places, others rounded to three), which meant I needed to start fresh on half of them. Ugh!

The problem is obvious-we need the calculation power of spreadsheet software combined with the formatting capabilities of document or presentation software. We need a way to do our analysis AND write our text in the same place, so everything stays connected and consistent. Wouldn't it be nice if we could just say "calculate the average of this column" right in the middle of a sentence, and have the correct number appear automatically? Or create a beautiful chart with actual code instead of clicking through endless menus, and have that chart update itself when the underlying data changes?

The good news? This dream workflow actually exists-and it's available right

here in R! What we need is an example of what is known as *Literate Programming*. The concept, introduced by Donald Knuth in 1984 (Knuth 1984), is pretty simple: combine code and regular writing in a single document. Your analysis and your narrative live together, so when one changes, the other updates automatically. No more copy-paste nightmares, no more inconsistent rounding between tables, no more hunting through your document to update all those numbers by hand!

In the R world, there are tools designed specifically for this purpose. For years, **RMarkdown** was the go-to solution, but since 2022, it's been largely replaced by **Quarto**—a more powerful system developed by Posit (the company behind RStudio). Quarto is what I used to create this entire book, and in this chapter, I'll show you how it can make your data-driven documents both easier to create and better looking.

4.1 Quarto

So, what is Quarto, and how do you use it? Basically, it's a file type that allows you to mix normal writing with code and spits out a single document. This chapter will guide you through the basics of using Quarto (Allaire and Dervieux 2024). If you want more details on Quarto, you can check out their website at https://quarto.org. And if you want a deeper dive than what is in this chapter, the official Quarto guide is really useful (https://quarto.org/docs/guide/) (Quarto Development Team 2024).

Quarto can create all kinds of documents-HTML web pages, PDFs, Word documents, PowerPoint slides, interactive dashboards, and more. For simplicity's sake, we'll focus on HTML documents in this chapter since they're the most versatile and easiest to get started with. For many of you, HTML may sound like a foreign concept, but it's probably the file type that you have experienced the most often in your life! HTML is the file type that web browsers display. So, when you make an HTML file with Quarto, you're basically creating your own mini-website!

While Quarto documents can create a bunch of different types of files, the Quarto file itself lives on your computer as a .qmd file. Let's jump right in and create a Quarto document:

1. In RStudio, go to File → New File → Quarto Document
2. A dialog box will appear asking for some basic information
3. Give your document a title (like "My First Quarto Document")
4. Add your name as the author
5. Make sure HTML is selected as the output format
6. For now, leave the "Use visual markdown editor" box unchecked (I'll explain why in a minute)
7. Click "Create"

Congratulations! You've just created your first .qmd file. But wait—it already has some content in it. What's all this?

4.1.1 Anatomy of a Quarto Document

Let's talk about the architecture of a quarto document. Each quarto document generally has 3 types of sections:

- YAML
  - The YAML is the section at the very top surrounded by three dashes (`---`) on both sides. YAML stands for "Yet Another Markup Language" and it is like the control panel for your document—it tells Quarto how to format your final output. You only have one YAML section per document, and it's always at the top
- Markdown Language
  - The main body of your document is written in markdown language. If you've ever used a simple text editor but wanted to add some formatting, that's basically what markdown is. Instead of clicking buttons for bold or italics, you wrap text in special symbols. For example, wrapping a word with asterisks (*) will make it appear in italics in the final document; if your markdown file says *econometrics*, it will appear in print as *econometrics* in the final version.
- Code Chunks
  - Incorporating code, and its results, is the star feature of Quarto. Code goes into something called a **code chunk**. While Quarto can be used with more than just R (e.g. Python, Julia, etc), the focus here will be on R chunks. You can identify a code chunk in the code because it starts with ```` ```{r} ```` and it closes with ```` ``` ````.

To transform the .qmd file into the final result, you need to **Render** it. The keyboard shortcut to render a .qmd file is ctrl-shift-k (cmd-shift-k on a Mac), and if you have a .qmd file open in your editor pane, you should see a render button on the toolbar over your document. The process or rendering your file saves the .qmd file, and the rendered (HTML, PDF, etc) file automatically winds up in the directory that your .qmd file is in.

Quarto will work its magic, running all your code and formatting your text, and create an HTML file in the same folder as your .qmd file.

> Tip from the Helpdesk: Projects Make Perfect
>
> It's a good habit to work with Quarto documents in an RStudio Project. Why? Because when you render a .qmd file, the output goes in the same folder as the source file. If you don't use a project, you might end up with

rendered documents scattered all over your computer.
Remember the Projects tip from Chapter **??**? This is exactly why they're so valuable. If you haven't created a project yet, flip back to that section for a quick refresher on how to set one up. Your future self will thank you when you're not playing hide-and-seek with your files!

In the next sections, we'll dive deeper into each of these components, starting with the YAML header.

4.2 YAML

The YAML controls many of the general settings of your document. The nearly universally chosen things (technically you don't *need* a YAML, but going without one is more of the "hold my beer" approach and is better left for later as more of an advanced technique) to include in your YAML are typically generated with the new Quarto Document dialog box:

- Title - The name of your document, it shows up at the top in big letters.
- Author Name - Hey! That's you! This also shows up in your document, just under the title.
- Default editor type - the two options are source and visual. Don't stress this choice too much, you can easily switch between the two on the fly with the buttons on your editor window toolbar. Personally, I often switch between the two frequently while working on a document!
- Output document format - the default is an HTML document, but other options include PDF, Word, and various slideshow formats.

While these are the basics you need to get started, here are a few additional options you will generally want:

- Table of Contents - Essentially a clickable navigation menu, including this is a great addition to an HTML file. Quarto will make this for you automatically based on your headers (we'll talk about this in the section on Markdown Language), it adds a touch of refinement and requires almost no extra work! Specify that you want one by including the following code under your `html:` options:
  - `toc: true`
- Stand-Alone file options - If you want to send documents as a single HTML file and not worry about sending all the pictures, tables, equations, etc inside the HTML file separately (spoiler alert: you probably do!), add this under your `html:` options. They make the files larger, but without these options your final products may be missing things:
  - `embed-resources: true`

- Code Chunk Defaults
  - More on this when we get to the section on Code Chunks, but these let you set a default way for Quarto to deal with the R code you include in your report. For example, let's say you are making a document that includes a dozen graphs. Do you want the body of the text to show the reader what the code was that generated the graphs (appropriate for homework or rough drafts), or do you want the final report to only have the polished graphs (appropriate for term papers and slick presentations)?

- Themes
  - Quarto has a bunch of built in themes to make your document look snazzy, which you can set using `theme:` under the `html:` option. Play around with these to see if there is one you particularly like. You can also click the links below to see what they look like in a sample document:
    * default
    * cerulean
    * cosmo
    * cyborg
    * darkly
    * flatly
    * journal
    * litera
    * lumen
    * lux
    * materia
    * minty
    * morph
    * pulse
    * quartz
    * sandstone
    * simplex
    * sketchy
    * slate
    * solar
    * spacelab
    * superhero
    * united
    * vapor
    * yeti
    * zephyr

Tip from the Helpdesk: You Need to Give the YAML Space!

YAML is super finicky with regards to tabs and spacing. For example, below is the YAML for a Quarto handout I once gave my class:

```
title: "Notes for Quarto"
author: "Matt Dobra"
format:
  html:
    theme: sandstone
    embed-resources: true
    toc: true
    html-math-method: katex
    self-contained-math: true
editor: visual
bibliography: references.bib
```

The indentation in this document is actually serving a purpose-it is creating a hierarchy. I have specified `html:` as the `format:`, and the five lines under `html:` modify the `html:`. If I get rid of all the tabs (indents) in the YAML, and try to render with the following YAML, Quarto will throw a little tantrum and refuse to render the document:

```
title: "Notes for Quarto"
author: "Matt Dobra"
format:
html:
theme: journal
embed-resources: true
toc: true
html-math-method: katex
self-contained-math: true
editor: visual
bibliography: references.bib
```

Even though the text is identical between the two documents, the tabs are different and that makes all the difference here. The tabs tell the YAML what parts of the code are subsets of something else.

While the YAML is the first thing in your Quarto document, it is usually the thing you spend the least amount of time on. The YAML header controls how your document looks and behaves, but these settings will make more sense to you as you work with what you're actually formatting. The main body of every Quarto document alternates between sections written in Markdown Language for your narrative and Code Chunks that execute your R code.

4.3 Markdown Language

Following the YAML, you will generally be switching between using Markdown Language and Code Chunks. The Markdown Language sections are where the text of your document appears, and you have two options on how to write it: using the visual editor or the source editor.

If this book is your first foray into coding of any sort, you will probably be more comfortable with the visual editor, as editing in the visual editor mode is quite reminiscent of the sort of format editing you might find in any number of web-based platform like Google Docs, Gmail, Reddit, and so forth.

Within the visual editor, you can modify text just like you would in a word processor - select it and use one of the options in the formatting ribbon to change how it looks, or use normal keyboard shortcuts (e.g. ctrl-b is bold, ctrl-i is italics, etc.). You can also do things like create bulleted lists, insert pictures, and more.

You may, at first glance, be underwhelmed by the visual editor. You might think it feels a bit like a barebones version of a word processor. And honestly, that's a fair assessment. For basic stuff-that is to say, writing documents that aren't going to include R code in them-you may prefer Word as a simpler solution. After all, it's a lot easier to collaborate with normies using Word or Google Docs than it is in R! However, much of the power of the markdown language is more useful in doing research; for example, Quarto can automatically write bibliographies and do inline citations for you, if you invest a little time learning about it. In my opinion, these things are a lot easier to do well in Quarto than in Microsoft Word.

Because the visual editor is fairly straightforward and similar to editors you might have used before (Word, text editors on online message boards and Canvas/Blackboard classrooms, etc), I do not feel the need to go into great detail here on the basics. Instead, let me give you a few useful tips that are unique to writing Quarto documents.

4.3.1 Source Editor Basics

If you choose to work in the source editor (or just want to understand what's happening behind the scenes in the visual editor), here are the most common markdown formatting codes you'll use:

- Headers: Use `#` symbols at the beginning of a line. More `#` symbols mean lower level headers:
  - `# Main Header` (equivalent to Header 1)
  - `## Section Header` (equivalent to Header 2)
  - `### Subsection Header` (equivalent to Header 3)

- Text Formatting:
  - `*italic text*` or `_italic text_` for *italic text*
  - `**bold text**` or `__bold text__` for **bold text**
  - `~~strikethrough~~` for ~~strikethrough~~
  - `` `code` `` for `code` formatting
- Lists:
  - Unordered lists use `-`, `*`, or `+` at the beginning of a line
  - Ordered lists use numbers (`1.`, `2.`, etc.)
  - Indent with spaces to create nested lists
- **Links and Images**:
  - `[Link text](URL)` creates a hyperlink
  - `![Alt text](image-path)` inserts an image
- **Quotes and Code Blocks**:
  - Start a line with `>` for a blockquote
  - Wrap code blocks in triple backticks (“‘) for multiline code

The nice thing about source editing is that once you learn these basics, you can often type faster than clicking through menus in the visual editor. Plus, you'll develop a better understanding of how Quarto documents are structured behind the scenes.

May the Format Be With You: Editor Hopping

I find myself switching between the visual and source editors constantly while working. The visual editor is super helpful when you're starting out or when you need to do something you don't do very often (like inserting a table or fancy equation). But for regular writing and basic formatting, the source editor is often quicker once you get the hang of it.

I think most people who use Quarto regularly tend to drift toward the source editor over time. It's typically faster for typing, doesn't lag with bigger documents, and gives you more direct control. That said, there's absolutely nothing wrong with sticking with the visual editor if that's what you prefer!

My advice? Get comfortable jumping back and forth between the two using the buttons at the top of your editor pane. Use whichever one feels right for what you're doing at the moment. Don't be surprised if you find yourself using the source editor more and more as you get better at Quarto - that's a pretty common pattern.

4.3.2 Inline Code

Sometimes you want to include R calculations directly within your text as part of a natural flowing sentence. This is called **inline code**, and it's perfect for situations where you want to reference specific values, calculations, or results without breaking the flow of your writing.

For example, instead of writing "The average fuel efficiency was approximately 20.1 miles per gallon," you can have R calculate and insert the exact value automatically. This means when your data changes, your text updates automatically too-no more hunting through your document to manually update numbers!

To use inline code, wrap your code with backticks, and put the language name first in curly braces: '{r} code here'. We are using R, but Quarto can work with other languages like Python or Julia too. When you render your document, Quarto will execute the code and replace it with the result. Here are some practical examples; if your Markdown Language says:

- The dataset contains '{r} nrow(cars04)' observations.
- The average fuel economy was '{r} round(mean(cars04$city_mpg, na.rm = TRUE), 1)' miles per gallon.

When rendered, these markdown lines with inline code would appear as:

- The dataset contains 428 observations.
- The average fuel economy was 20.1 miles per gallon.

Your initial thought might be to wonder about the utility of inline code. Why not just crunch the numbers on the side and type them directly into your text? That's fair, but inline code has a couple advantages.

- It automatically updates numbers in your text when data changes.
- It greatly reduces the likelihood of typos.
- This approach is actually faster if you have a lot of them. For example, if I am going to include in the text the means of all the variables, it's easier to copy and paste the code and change the variable name than to calculate every mean and type it in.
- For systematic report writing, where you write the same report repeatedly just with different data, this saves a ton of time.

You generally want to keep your inline code super simple-basic calculations and single values only. If you need to do something more complex, that's where Code Chunks (discussed next) come in.

4.3.3 Use header styles, not large bold fonts to create sections.

Don't use bold/font size for section headers. Instead, use header formats, either in the source editor with octotharpes (`#`), or in the visual editor via the drop-down box that usually says "Normal" to choose "Header 1", "Header 2", etc. This enables Quarto to automatically create an interactive table of contents for your documents. It also creates the ability to have interactive tags in your documents so the reader can click on a link to take them to different part of the document. Additionally, it is useful for the outline feature to the right of the editor in RStudio, so you can easily navigate your document while working on it.

Admittedly, this is far more useful in large documents than in shorter reports or homework, but if good habits are no harder to do than bad habits, why not do it the way that's better in the long run?

4.3.4 Equations can be typeset using LaTeX

Mathematical equations can be be typeset beautifully in R. The easiest way to create an equation is via the visual editor using the insert menu → LaTeX Math - it's worth spending a little time learning some basic math typesetting. You can also create equations via the source editor by wrapping your LaTeX typesetting with dollar signs.

Here are a few of the essential LaTeX formatting styles that are particularly useful in statistics:

- basic Greek Letters (typically a \ followed by the name of the letter) – some examples
  - `\alpha` gives α
  - `\beta` gives β
  - upper and lowercase sigma come from `\Sigma` and `\sigma`, respectively (Σ and σ).
- subscripts and superscripts
  - `x_1` is x_1
  - `\beta_1` is β_1
  - `x^2` is x^2
  - by default you only get your first character after the _ or ^ to be sub- or superscripted, so for more complicated stuff you need to include curly braces
    - `x^{10}` is x^{10}
    - `\beta_{13}`is β_{13}
- accents like hats and bars
  - `\bar{x}` is $\bar{x}$
  - `\hat{\beta}` is $\hat{\beta}$
  - `\hat{\beta_{11}}` is $\hat{\beta_{11}}$

- fractions
  - `\frac{1}{4}` is $\frac{1}{4}$
  - `Z_i = \frac{x_1 - \bar{x}}{\sigma_x}` is $Z_i = \frac{x_1 - \bar{x}}{\sigma_x}$

Your LaTeX typesetting only works inside a LaTeX math block, which you can create either by inserting it into a document from the visual editor or by wrapping your LaTeX code inside of dollar signs. If you look at the math inside the source viewer, you will see it wrapped inside the dollar signs. For example, if you looked at the source code for β_1 you would see `$\beta_1$`.

As you may have noticed, these typesetting options can be mix-and-matched to create some elaborate equations. We can add these techniques to other math typesetting options that you can find online to get some impressive stuff! For example:

The bivariate OLS equation:

$$y_i = \hat{\alpha} + \hat{\beta} X_i + e_i$$

The normal distribution:

$$X \sim \mathcal{N}(\mu, \sigma^2) f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

May the Format Be With You: Equations make your YAML Grow

If you are going to include LaTeX math in an html file, and you want a self-contained file (e.g. you specified `embed-resources: true` above), you need to include more options in your YAML because Quarto treats the math stuff differently from the images and charts. To make this work, add the following lines to your YAML under the `html:` formatting options:

```
format:
  html:
    (other html options)
    html-math-method: katex
    self-contained-math: true
```

4.4 Code Chunks

Incorporating code, and its results, is the star feature of Quarto. It's why we fired up RStudio instead of Word, Powerpoint, or Excel! The part of a Quarto document where we insert code is called a **code chunk**. While Quarto can be used with more more languages than just R (e.g. Python, Julia, etc), the focus here will be on R chunks. The easiest way to start a code chunk in the visual

editor is, on an empty line, hit the / button and the first option is likely an r chunk. Alternately, the keyboard shortcut `ctrl-alt-i` creates a code chunk as well. In the source viewer this looks like three backticks followed by {r} (```` ```{r} ````), and the chunk is closed below with three more backticks (```` ``` ````). Once RStudio recognizes the code chunk, it adds three icons to the top right corner of it; a gear icon that lets you set chunk options, and two run options-the down arrow runs all code chunks above it and the right arrow runs that particular chunk.

Two things happen inside a code chunk: chunk options and code:

- Code: This will typically be data tasks; data wrangling, statistical analysis, constructing graphs, and the like.

- Chunk Options: The options specify how Quarto will include your R work into your final document. Do you your final document to show your code or just the results? Or neither? Do you want your final document to include your code but not actually *run* the code? This is what chunk options control.

This chapter will not focus so much on the code as it will on the options; the rest of the book is about learning how to write the code that goes into the code chunks, and as this chapter is about making pretty documents, what we really want to focus on here is how to control what R does (and does not) include in the rendered document.

Options are specified in one of three different ways. First, all options have a default. If you don't specify anything at all, Quarto has a default behavior which is basically to include everything possible in your Quarto document: the code, the results, any messages or warnings you get. You *usually* don't want this behaviour, which is why we are going to take a deeper look at chunk options. The second way you can set your chunk options is by changing the default options for your document via the YAML. I'll talk more about this option in a **May the Format Be With You** tip below. The third way you can set a chunk option is by specifying it at the top of your code chunk before your actual code.

Some of the most common chunk options can be selected with the gear icon, and you can see the chunk options declared for any a particular chunk in both the source and visual editors; they appear at the top of the chunk on lines that start with a `#\|`, a.k.a. a "hash pipe." Yes, that's the actual name, not another one of my off-color jokes!

Let's start by going through some of the most useful options for your hash pipes, and I'll talk about a couple typical use cases below. Useful options for your hash pipes:

4.4.1 Display Options (How your code appears)

- `#| echo:` - Controls whether your code shows up in the final document.
  - `true` (default): Shows your code. Perfect for homework, tutorials, or methodology explanations, collaboration.
  - `false`: Hides your code. Usually you want this for formal reports or presentations.
- `#| code-fold:` - Gives readers the option to peek at your code if they want.
  - `false` (default): Code always visible.
  - `true`: Code is hidden initially but it can be shown with a click. This is ideal for documents where some readers might want technical details, or turning in homework. It's the default in this book!

4.4.2 Execution Options (How your code runs)

- `#| eval:` - Controls whether the code executes.
  - `true` (default): Runs the code. Use for most normal situations. Most of the time you want this.
  - `false`: Skips execution. Useful when you're showing code that would crash spectacularly, take forever to run, or requires stuff your readers don't have.
- `#| include:` - Controls whether any output at all from the chunk appears in the document.
  - `true` (default): Shows whatever the chunk produces in your document.
  - `false`: Runs the code silently in the background but doesn't put it anywhere in your fiinal document. This is actually quite useful for proper papers to hide all the data wrangling nobody wants to see, loading packages, and behind-the-scenes calculations.

4.4.3 Output Options (How results appear)

- `#| message:` - Controls whether informational messages show up in your document.
  - `true` (default): Shows all messages from your code. You'll see package loading notifications and other information R wants to tell you.
  - `false`: Hides these messages. Great for keeping your document clean, especially when loading packages that like to announce themselves-`tidyverse` (Wickham et al. 2019), I'm looking at you!

- `#| warning:` - Controls whether warnings appear in your final document.
  - `true` (default): Shows any warnings your code generates. Useful when debugging or when warnings contain important information.
  - `false`: Suppresses warnings in the output. Helpful for final documents when you're aware of the warnings but don't need to share them with readers.
- `#| output:` - Tells Quarto how to handle what your code produces. A couple examples:
  - `#| output: asis` - Passes the output through without Quarto's usual formatting. This is essential when using packages like `stargazer` that create their own formatted tables or when you want raw HTML/LaTeX to be interpreted.
  - `#| output: fold` - Similar to code-fold but for output - shows a clickable button to expand/collapse the results. Great for extensive output that might otherwise dominate your document.

4.4.4 Figure and Table Options (Making your visuals presentable)

- `#| fig-width:` and `#| fig-height:` - Control the dimensions of plots in inches.
  - Example: `#| fig-width: 8` and `#| fig-height: 6`
  - These options help you avoid awkwardly sized graphics - no more squinting at tiny graphs or dealing with images that overwhelm your page.
- `#| fig-cap:` - Adds a numbered caption to your figure.
  - Example: `#| fig-cap: "Distribution of Residuals"`
  - Automatically creates "Figure 1: Distribution of Residuals" and correctly numbers all your figures in sequence. This is expected in formal papers and makes your document look properly structured.
- `#| tbl-cap:` - Adds a numbered caption to tables.
  - Example: `#| tbl-cap: "Regression Results"`
  - Works just like figure captions, creating "Table 1: Regression Results" with automatic numbering. Particularly useful when you have multiple statistical tables and need to reference them in your text.

4.4.5 Organization Options (For keeping things manageable)

- `#| label:` - Gives your chunk a meaningful name.
  - Example: `#| label: import_data`

- No default. While it seems trivial when you start, labels become incredibly helpful for navigating larger documents and tracking down errors. For example, if you have an error in one of your code chunks, Quarto will give you an error telling you which code chunk was problematic. And the error message "in chunk 'import_data' " is much more useful than "in unnamed chunk #69".

May the Format Be With You: Advanced Tip: Cross Referencing

There is another cool benefit of using labels in your chunk options; if you can learn a few naming conventions, you can use them to generate clickable cross references within your final paper! Here's how it works.
Let's say you have 4 graphs in your paper. And you give all of those graphs labels that begin with the prefix `fig-`, so maybe you have `#| label:fig-boxplot`, `#| label:fig-histogram`, `#| label:fig-bargraph`, and `#| label:fig-scatterplot`. You also named them with `#| fig-cap:`, so your boxplot is labeled with "Figure 1:", your histogram with "Figure 2:", and so forth.
The fact that you gave the code chunks that create graphs labels that begin with the prefix `fig-` tells Quarto that these code chunks create `Figures` and creates anchors in your document by those figures, so if you want to refer to one of the figures in your paper, you can do so with `@` followed by the label name. So your markdown code might look like:
A visual inspection of the histogram in @fig-histogram shows that the data is right-skewed.
But when this renders into a document, the @fig-histogram will be replaced by the words Figure 2, *and* Figure 2 will be a clickable link and by clicking it, the reader will be taken directly to Figure 2 in your document. You can even link to it multiple times, so maybe you have a reference to @fig-histogram not only in your discussion of the data, but also reminding readers about @fig-histogram in the conclusion.
Pretty slick! The other really useful label prefixes to know early on are `tbl` for tables, `lst` for lists, and `eq` for equations.
Early on in the process of learning how to use Quarto, I wouldn't worry too much about this sort of functionality. But once you start to master the basics of Quarto, the jump to really slick interactive work is pretty easy to make!

4.4.6 Document Evolution and YAML Settings

One of the best things about Quarto is how easily a document can transform from one purpose to another. A document that starts as your personal exploration can evolve into a team collaboration tool, then into a formal paper-often with just a few changes to the YAML header.

Let's look at how you might evolve a document through its lifecycle by changing

your YAML settings:

4.4.6.1 Stage 1: Data Exploration (Just for You)

When you're first digging into your data, you want to see everything—code, outputs, messages, warnings. So you might want to set `echo`, `warning`, `message`, etc all to true. Those are the defaults, so you can usually have a pretty lean YAML and code options. At this stage, what is most important is to ensure that you are making use of the `#| label:` tag so you can easily find stuff later.

4.4.6.2 Stage 2: Collaboration

By collaboration, I mean either working with others on a project, or perhaps turning in homework. Here, you probably want to maintain the visibility of code, but you want a bit of a cleaner document. You might consider adding the following to the YAML:

```
format:
  html:
    code-fold: true  # Keeps code available but tucked away for cleaner viewing

execute:
  message: false     # Hide package loading messages that distract from content
```

4.4.6.3 Stage 3: Shareable Paper

As your analysis solidifies and you prepare to share with supervisors or reviewers, you want to have a cleaner output:

```
format:
  html:
    toc: true              # Add a table of contents for navigation
    embed-resources: true  # Ensures reviewers see everything in one file
    theme: journal         # Optional: Choose a professional theme you like

execute:
  echo: false              # Hide code - focus is now on results
  message: false           # Hide messages
  warning: false           # Hide warnings
```

This is the stage were you want to be concentrating on formatting and labeling your tables and figures by using `#| fig-cap:`, `#| fig-width:`, etc in particular code chunks. You also want to ensure that you set `#| include: false` on code chunks that just load libraries and data wrangle.

4.5 Rendering Your Document

Ultimately, the document is created by **rendering** your document. When you click the **Render** button a document will be generated that includes both content and the output of embedded code.

It may be a mistake to leave the most important note until the last section in the chapter, but it is essential to understand a little bit about how how Quarto operates. Quarto takes the .qmd file, and using a package called Knitr (**knitr?**) it creates a new markdown file in the background. To do its thing, Knitr opens a fresh version of R in the background to execute all of the code chunks in. From there, this file is passed into Pandoc/Rmarkdown (Allaire et al. 2024), which converts this background markdown file into something like HTML, PDF, etc.

There is something very important that you need to understand about `knitr` opening the fresh version of R. This means that, even if libraries like `tidyverse` are loaded into memory in the instance of R you are coding in, **they are not loaded in the instance of R that Quarto is spawning**. This is also true of data. Data in your current R environment are not loaded into the fresh R instance generated by Quarto. Additionally, it executes the code chunks sequentially. The first chunk gets executed first, the second chunk second, and so forth in order until all the chunks have been executed. Therefore, if your 5th chunk is where you put `library(tidyverse)`, but the third chunk tries to use `mutate()` or `group_by()` or `case_when()`, R will not be able to execute your code.

This means that your Quarto document **needs** to be, in a sense, self contained. Every library that the document relies upon must be loaded with a `library()` call in a code chunk. Every data set that the Quarto document uses needs to be loaded into R via the .qmd file in a code chunk. Every object the Quarto document manipulates needs to be created within the .qmd file. And the order in which the code appears is of paramount importance.

> Tip from the Helpdesk: Load Required Libraries Early!
>
> Occasionally you will find yourself in a situation where your Quarto document won't render, even though you can execute every single line of code in the Quarto document and get the results you expect. If this is going on, the culprit is typically one of the following three things:
>
> - you are using a library that you have loaded into R but did not include in your .qmd file,
> - you are referring to an object that you created in your current R environment but did not create in your .qmd file, or
> - your R code is not in the right order, and a line of code is relying on something you create in a subsequent code chunk in your .qmd file.
>
> The key to fixing this situation is typically to identify the code chunk

where Quarto is having issues (yet another good reason to label your code chunks!), and ensure that everything used in that code chunk (libraries, data, objects, etc) is loaded into/created in your Quarto document in a previous code chunk.

Typically, when I make a Quarto document, I include as the **very first first** thing after the YAML a code chunk with the `#| include: false` option in it that loads libraries. My next chunk (or chunks) do the task of loading in the data, data wrangling, and so forth, again with `#| include: false` specified, that loads and prepares all the data for analysis. All this happens in the .qmd file before any writing appears.

4.6 End of Chapter Exercise

Literate Programming Project: Create a data-driven report using Quarto that analyzes some aspect of a dataset of your choice. Your report should demonstrate your ability to combine narrative text with R code to create a professional-looking document.

Project Requirements:

- Choose a dataset from the book's previous chapters (such as cars04, gapminder, or iris) or any built-in R dataset that interests you
- Create a Quarto document with a clear title and your name as author Write a brief introduction explaining what aspect of the data you plan to examine (e.g., "This report examines fuel efficiency patterns in the 2004 car data" or "This analysis explores population trends across continents")
- Include at least 3 sections using proper header formatting Demonstrate data analysis using code from previous chapters or similar techniques - feel free to adapt existing code examples to your chosen dataset
- Create at least one summary table using functions like summarize(), group_by(), or summary()
- Use chunk options strategically: show code in some chunks to demonstrate your analysis process, but hide code in others to maintain document flow
- Include some mathematical notation using LaTeX (could be as simple as showing the formula for calculating a percentage or describing a statistical concept)
- Conclude with a brief discussion of what your analysis revealed about the data

Chapter 5

Descriptive Statistics with R

This chapter builds upon the previous chapters, using the tools of R for descriptive statistics. This chapter is written on the assumption that you already have a reasonable understanding of basic descriptive statistics, and as such will not go into detail on the "what" or "why" of descriptive statistics, but rather will focus on the how. The data wrangling skills from Chapter **??** will come in handy here. Often, creating meaningful visualizations and summaries requires filtering, grouping, or transforming your data first.

In many ways, my focus in this chapter is twofold. First, it serves as a review of the first half of a standard intro to applied statistics course. Second, it helps build your R skills by showing you how to do stuff you probably already know how to do in a program like Excel in R.

This chapter is broken into two parts; the first half focuses on graphs, the building blocks of data visualization. The second half focuses on numerical descriptive measures, the building blocks of statistics, econometrics, and the like.

Finally, before we get going, let me follow my own advice and load the required packages and data in my first code chunk!

```
# Libraries used in this chapter:

library(tidyverse)
library(stargazer)
library(AER)
library(wooldridge)

# Data used in this chapter
```

```
data(CPS1985)
data(vote1)

# attach(CPS1985)
# attach(vote1)
```

5.1 Graphs and Visualizations

One of the functionalities R is best known for is for building graphs that are sleek and elegant. In fact, you've probably seen graphs in the wild that were made using R; the New York Times and British Broadcasting Corporation (BBC) typically use R to create graphics for their news stories. The BBC even published a "cookbook" and package ("BBC Visual and Data Journalism Cookbook for r Graphics" 2019) so that you too can make your graphs have the same style as theirs!

When compared to making graphs in a program like Excel or Numbers, what you will probably find is that graphs in R require a little more effort, but the payoff is a graph that looks **many** times better than one from a spreadsheet program. Personally, one of the things that first got me hooked on R was seeing just how much more polished and and professional the graphs were, even with just a few lines of code.

There are two major graphing engines used in R: Base R graphics and `ggplot()` graphics. Base R graphics describe the graph engine that is built directly into R, whereas `ggplot()` is part of the `tidyverse`. I will only be touching on Base R graphics, as in my opinion, `ggplot()` makes better looking graphics and is more intuitive when you want to customize your graphs than Base R. But sometimes, you just want a quick and dirty graph, and Base R is just fine for that.

Making graphs in R-at least the really pretty ones using `ggplot()`-is easier once you understand the underlying philosophy of what graphing is all about. Let's dive into the basics of the Grammar of Graphics.

5.1.1 Grammar of Graphics

The "gg" in `ggplot()` stands for **Grammar of Graphics**, which is the graphing philosophy embedded in the program. The `ggplot()` function is in the `ggplot2` package, which is part of the `tidyverse`. The Grammar of Graphics is based on the idea that a graphic has a set of indispensable elements:

- **Aesthetic**: The dimensions and visual attributes of the graph. Think of this like the canvas upon which you will be creating your graph. This can be the x/y dimension, but also you can graph using color, size, and so forth. These are graphical elements used to display dimensions of the data, rather than being purely cosmetic. For example, perhaps I am making a

scatterplot of individual salaries with years of education on the x-axis and income on the y-axis. If I make all of the dots blue because it looks good, or matches the overall style of my paper, this is not an aesthetic. If I make the dots representing men blue and the dots representing women pink, this is an aesthetic, because I am using color to represent dimensions of the data.

- **Layers**: If the aesthetics are your canvas, the layers are what you paint on top. Each layer has several components, and while there are lots of technical terms-data, mappings, geometries, etc.-don't let the fancy terminology scare you off. They're mostly intuitive concepts! Continuing with the example from above, the data will be all the individual observations in your data set, the geometry of a scatterplot just means we are using dots, and the mapping is simply to put each dot at the right spot on the x/y space.
- **Theme**: The overall styling and presentation of your graph - titles, axis labels, background colors, fonts, legend placement, and so forth. This covers both what your labels should say and how everything should look. Going back to our salary example, you want readers to understand what they're looking at: Is the y-axis income per year or per hour? What do the blue and pink dots represent? A good theme ensures your audience can interpret your graph without having to guess what anything means.

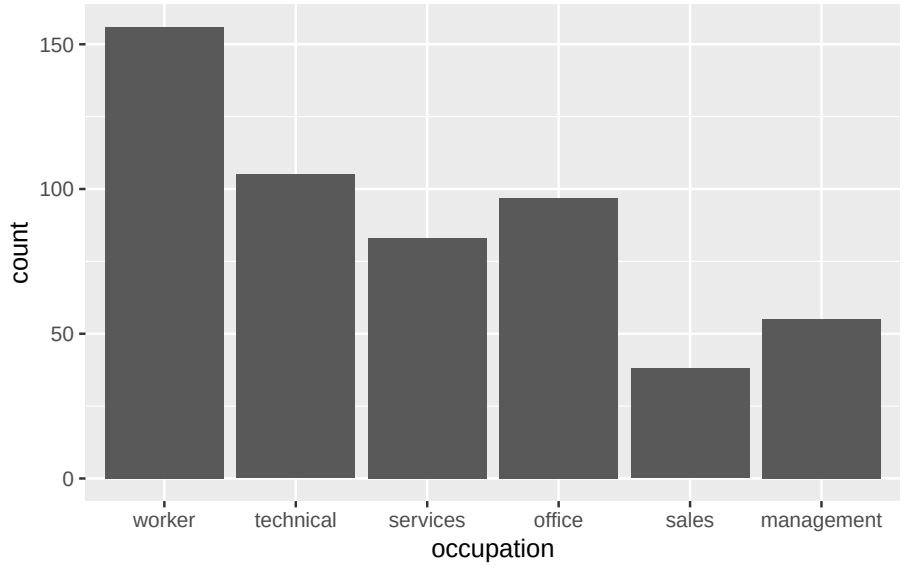
There are a few more technical elements (scales, coordinates, facets) that we'll encounter as we build more complex graphs, but these three concepts will get us started, and honestly, they'll take you pretty far! That said, this chapter is really just scratching the surface of what's possible with `ggplot2`. If you find yourself getting excited about data visualization (and trust me, it's addictive), I highly recommend checking out Hadley Wickham's *ggplot2: Elegant Graphics for Data Analysis* (Wickham 2016). It's the definitive guide to the package, written by its creator, and like many R resources, it's freely available online at https://ggplot2-book.org/. Now let's jump into making some graphs, starting with bar charts and the most evil chart of them all - the pie chart!

5.1.2 Bar and Pie Charts

Let's start with bar charts. The typical use of a bar chart is to plot some numeric variable by category. This might be the number of people in different groups, average income by race or gender, or something similar. We will be using the `CPS1985` data from the `AER` package (Kleiber and Zeileis 2008) which includes 534 observations from the May 1985 Current Population Survey. For full details on the `CPS1985` dataset, check out the `AER` package documentation or run `?CPS1985` in your console.

For basic bar charts, `ggplot()` is considerably easier than base R. Let's jump in with the basic code that will generate a bar chart that counts the number of individuals in the CPS data by occupation:

```
CPS1985 |>
  ggplot(aes(x = occupation)) +
  geom_bar()
```



In three lines of code, we have created a very barebones bar chart for our data. How does this code work?

- `CPS1985 |>` pipes the CPS data into the `ggplot()` function.
- The `ggplot()` sets the *aesthetic* for the overall graph with the `aes()` option. I want the x dimension to be the occupation variable, hence `aes(x = occupation)`. Because the default statistic of a bar chart is to give me counts of each group, I don't need to define any other aesthetics.
- The `+` at the end of the line allows me to add a layer to my graph.
  - This is also a good place to remind you of the "one line, one thing" rule from Chapter **??**!
- `geom_bar()` creates a basic bar chart. The prefix `geom_` tells us that we are adding a *geometry* layer, and our *geometry* is that of a *bar chart*.

Data Storytelling: The Three Stages of Graph Evolution

In Chapter **??** Literate Programming, I suggested that documents evolve through stages. So too do graphs.
There's a very similar three-stage approach you might think of with graphs too:

- Stage 1: Data to Plot - Here you ask the question, "does this show what I think it shows?" There is no need to mess around with

making everything look pretty, this is just you exploring the data, trying to find the graph that does the best job of telling the story you want to tell with the data. The focus here is typically on getting the aesthetics and layers right.
- Stage 2: Basic Communication - The next stage is to ask "can someone else understand this?" In this stage, you need to ensure that the data is labeled, titles are on your graph, and so on. This is when you start adding theme elements, particularly label options.
- Stage 3: Publication Ready - The final stage is to ask the question of "would I put this in a presentation/report?" This step is putting the final polish on your graph, ensuring your colors are right (are you on brand? Do you need to choose colorblind friendly colors?), etc.

Throughout this chapter, I'll usually show Stage 1 and Stage 2 examples, with occasional Stage 3 examples to show what's possible.

You might be wondering why we use `|>` to connect `CPS1985` to `ggplot()` but then use `+` to add layers to our graph. Here's the logic:

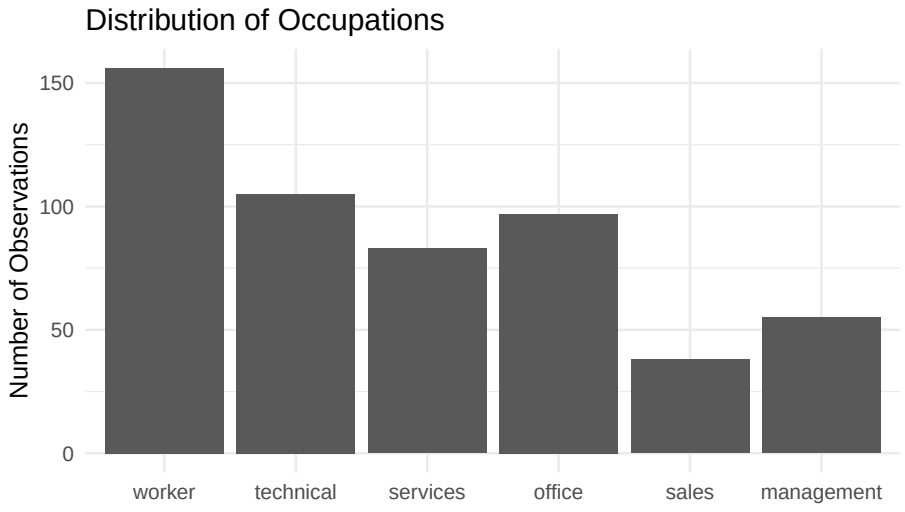
- Use `|>` (pipe) when you're passing data from one step to the next. Think "take this data **and then** do something with it."

- Use `+` (plus) when you're building up layers within ggplot. Think "add this layer **on top of** what I already have."

So `CPS1985 |> ggplot()` means "take the CPS1985 data and pipe it into ggplot," while `ggplot() + geom_bar()` means "start with a ggplot and add a bar chart layer on top." As a general rule, you always use `|>` for `dplyr` data wrangling tasks, but use `+` for `ggplot()` visualization building.

While we have a perfectly functional graph, it is not that pretty. I don't love the default gray background that `ggplot()` gives, nothing is labeled, and so forth. Let's fix this up with a few more layers and theme options:

```
CPS1985 |>
  ggplot(aes(x = occupation)) +
  geom_bar() +
  theme_minimal() +
  labs(title = "Distribution of Occupations",
       caption = "Random sample of 534 observations from 1985 CPS",
       x = "",
       y = "Number of Observations")
```



This code starts with the barebones code from above and augments it to add some additional elements and layers, and with just a few more lines the graph is a lot more readable. Here is what got added:

- `theme_minimal()`: There are a few "all-in-one" themes built into `ggplot()` that alter the appearance of the graph considerably, and `theme_minimal()` is probably my favorite general-purpose theme. The other themes are `theme_gray()` (technically the default theme), `theme_bw()`, `theme_linedraw()`, `theme_light()`, `theme_dark()`, `theme_classic()`, and `theme_void()`. It's worth taking a look at these to see which you prefer.
- `labs()`: Short for **labels**, specifying labels and titles are essential for readability. By default, `ggplot()` labels axes with the variable names from your dataset, but these aren't always reader-friendly. Here we've added a title, caption, and changed the y-axis from "count" to the more intuitive "Number of Observations." For the x-axis, since our title already specifies we're looking at occupations, the default "occupation" label is redundant, so `x = ""` removes it entirely.

Data Storytelling: The Self-Contained Graph Test

A well-designed graph should tell a complete story on its own. If you handed your graph to a friend or colleague or teacher with no context, could they understand what they're looking at and what point you're making? Good graphs don't require you to be standing there explaining what everything means.

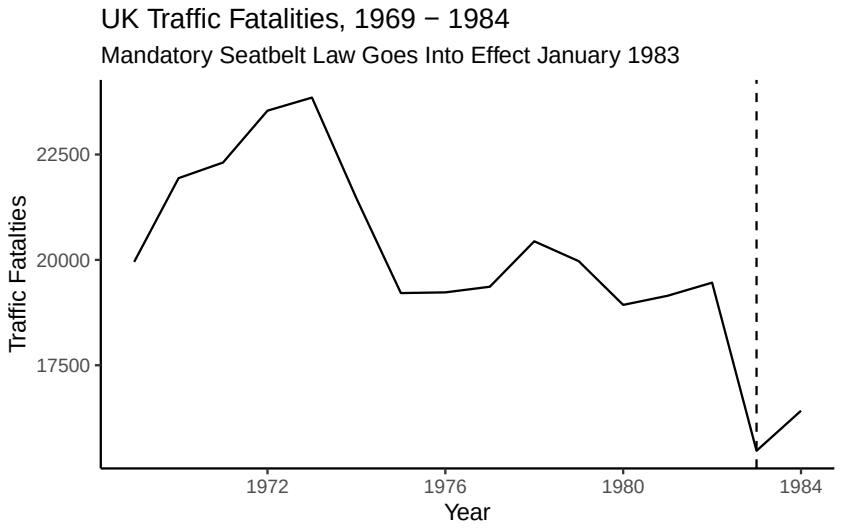
To illustrate this point, I'm going to do things a bit backwards for a moment. I'll start by showing you a graph without any of the code that wrangles the data so you can't cheat and look up the data yourself (I mean you can still cheat by scrolling down a bit but whatever). Look a look at this graph; what does it show?

```
sb |>
  ggplot(aes(y = drivers, x = year)) +
  geom_line()
```

Based on the default labeling, it looks like a time trend of some sort because the x-axis is labeled "year", but drivers could be anything. Something about automobiles, maybe? Traffic safety or number of people getting their licenses? Number of taxi drivers? Or maybe some other kind of drivers. Perhaps we are looking at the data for sales on golf clubs or screwdrivers? If screwdrivers, the tool or the cocktail? And for all of these, where? US? Worldwide? Who knows?

This graph cannot stand alone. And generally speaking, it's not good enough to have a thorough description of the graph in the text around your graph (although you do need that!). A graph needs to tell a self-contained story, so let's put some labels in here:

```
sb |>
  ggplot(aes(y = drivers, x = year)) +
  geom_line() +
  geom_vline(xintercept = 1983, linetype = "dashed") +
  theme_classic() +
  labs(title = "UK Traffic Fatalities, 1969 - 1984",
       subtitle = "Mandatory Seatbelt Law Goes Into Effect January 1983",
       y = "Traffic Fatalties",
       x = "Year")
```



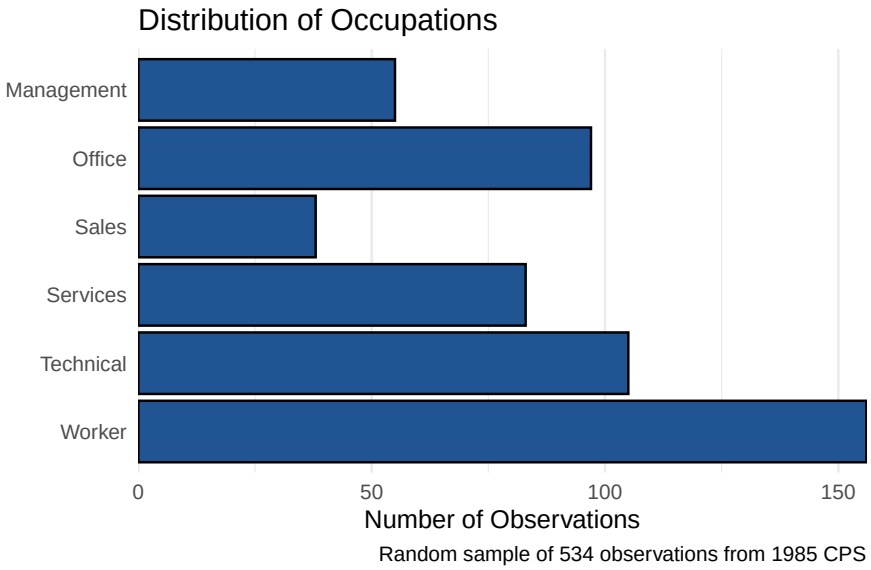
With just a little more effort, we can see what the graph is showing. All I really added was a title/subtitle, axis labels, and the dashed line to show the date when compulsory wearing of seatbelts went into effect. And now, the graph now tells a story.

As promised, here is the code that generates the data from the `Seatbelts` dataset built into R. The original data is monthly and not in a format that the `tidyverse` likes (`dplyr` and `ggplot()` don't play nicely with data configured for the sorts of Time Series stuff we will see in Chapter **??**); most of what you see here is just an application of the data wrangling techniques from Chapter **??**. It changes the data type (that's the `as.tibble()` line), creates a year variable (that's the `mutate()`), and then converts the monthly data into annual data:

```
data(Seatbelts)
sb <- Seatbelts |>
  as.tibble() |>
  mutate(year = floor((1969 + (row_number()-1)/12))) |>
  group_by(year) |>
  mutate(drivers = sum(drivers)) |>
  select(year, drivers) |>
  distinct()
```

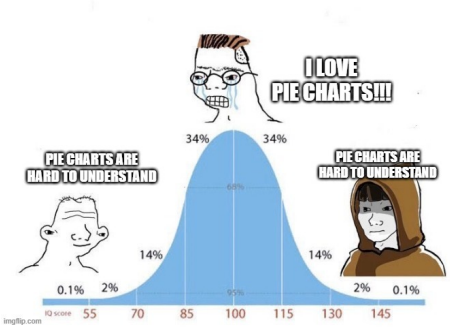
Our occupation graph is fine, but suppose we really want to elevate this graph. Let's think about what we might need to do to make this publication worthy. First, we might think about color. Since this is Data from the US Census, let's imagine that we want to use the official colors of the US Census. I went online and grabbed the official version of the color blue they list in their style guide ("US Census Bureau: Corporate Identity and Branding Standards" 2019), which is `#205493`, so we can include that as a color in the graph. I'm going to add a black outline to the bars to make them pop as well; since black is one of the many "named colors" in R, we'll just refer to it as `"black"` (If you want to learn more about named colors, try running `demo("colors")`!). We might also think to flip the orientation (horizontal bars instead of vertical bars), clean up the gridline situation, remove some empty space between the axis and the beginning of the bars, and alphabetize and capitalize the occupations:

```
CPS1985 |>
  mutate(occupation = fct_rev(fct_relevel(occupation, sort(levels(occupation))))) |>
  mutate(occupation = fct_relabel(occupation, str_to_title)) |>
  ggplot(aes(y = occupation)) +
  geom_bar(fill = "#205493",
           color = "black") +
  scale_x_continuous(expand = c(0,0)) +
  theme_minimal() +
  theme(panel.grid.major.y = element_blank()) +
  labs(title = "Distribution of Occupations",
       caption = "Random sample of 534 observations from 1985 CPS",
       y = "",
       x = "Number of Observations")
```



Real talk: the code complexity just took a bit of a jump here. The `fct_` functions are doing some behind-the-scenes magic to alphabetize and capitalize the occupation labels, but don't stress about understanding them right now. Here's the thing about making graphs in R: it follows the classic 80-20 rule. Getting from ugly to pretty good takes about 20% of the effort and is usually pretty easy, but going from pretty good to magazine-quality? That's the other 80% right there, and it can require some serious code gymnastics. It's why you don't do it until you are sure about the basics of your graph. The payoff is worth it though; this graph is way more polished than our earlier versions!

The other primary way to display categorical data is via a pie chart. Fun fact: many people who study data visualization believe that a pie chart is a really awful way to graph data. Case in point: the folks behind `ggplot2` did not provide an easy way of making a pie chart, they know better!



Base R, however, has no such qualms about letting you make questionable life

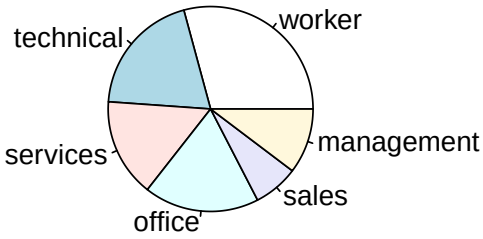
choices. Making a pie chart is quite simple and requires two steps. First, you need to use the `table()` function to create the actual data to be graphed:

```
jobtype <- table(CPS1985$occupation)
jobtype
```

```
    worker  technical   services     office      sales management
       156        105         83         97         38         55
```

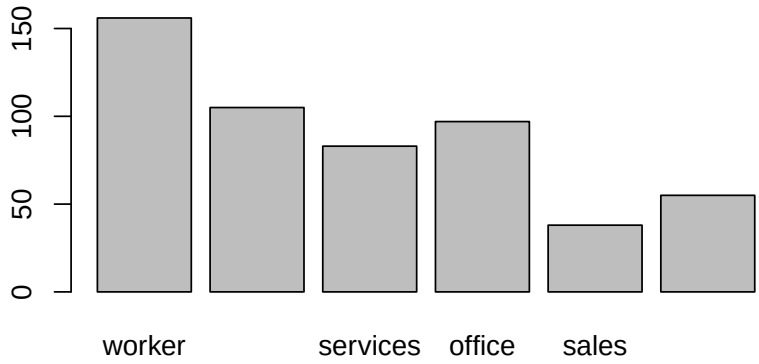
As you can see, `table()` creates a cross-tabulation table of the variable you feed it. From there, the base R pie chart is generated with the `pie()` function:

```
pie(jobtype)
```



In base R, it is easy as pie. Ugly, but easy. For what it's worth, the base R version of the bar chart requires the same initial step of making the cross-tab and then plugging it into the `barplot()` function.

```
barplot(jobtype)
```



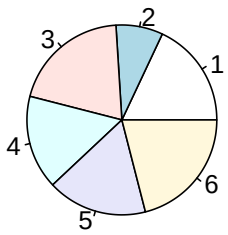
Data Storytelling: Why Pie Charts Are Evil (A Scientific Demonstration)

I am firmly on team "Pie Charts are Evil," and I'm about to demonstrate why with a simple experiment. Here's the thing: our brains just aren't wired to accurately compare angles and pie slices, which is exactly what pie charts force us to do.
Let me show you what I mean. I'll simulate rolling 100 dice and then compare how pie charts vs. bar charts handle the results:

```
set.seed(8675309) # Jenny!
dice_rolls <- sample(1:6, 100, replace = TRUE)
dice_counts <- table(dice_rolls)
```

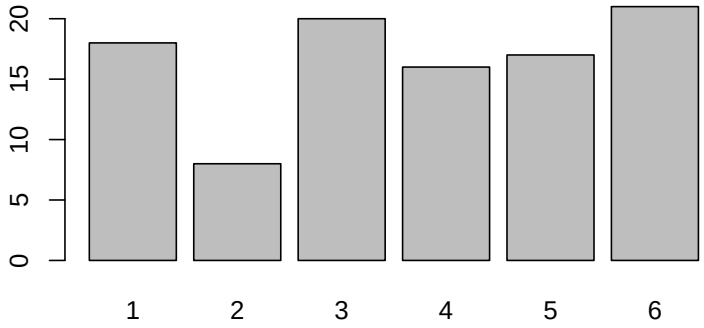
Now, which visualization actually lets you see which number came up most often?
Starting with the pie chart:

```
pie(dice_counts)
```



Seriously, can you tell the difference between the slices for 3 and 6? What about 1, 4, and 5? Your brain is working overtime trying to compare these curved slices, and it's probably getting it wrong.
Now, compare this to the bar chart:

```
barplot(dice_counts)
```



The differences are immediately obvious! Your brain is excellent at comparing heights - that's what bar charts give you to work with.
Just to confirm what the bar chart made obvious (but the pie chart made non-obvious!), here are the actual counts:

Table 5.1: Simulated Dice Roll Results

Die Face	Count
1	18
2	8
3	20
4	16
5	17
6	21

BTW, this isn't just me being a data visualization snob. There's actual research (e.g. Cleveland and McGill 1984) showing that people make more errors reading pie charts than bar charts. When you use a pie chart, you're literally making your audience work harder to understand your data. If you get to choose the presentation type, Why would you do that to them?

With some work, pie charts can also be made using `ggplot()`. However, I feel it is my responsibility to discourage you from making poor life decisions, and making pie charts is among the worst decisions one can make in life. Seriously, if you are making a pie chart, you really ought to be asking yourself what set of awful decisions did you make in your life that led up to a point where you are currently making a pie chart.

But, I get it. Maybe your boss or teacher or whomever really wants a pie chart. Definitely not you, you are better than that.

If it can be done, then I suppose it will be done! Making pie charts in `ggplot()` requires us to make a special version of a stacked barchart, which we will discuss later, so the `ggplot()` based instructions on making Pie Charts will have to wait until a little later in the chapter.

5.1.3 Boxplots

A **boxplot** (sometimes called a **Box and Whiskers Plot**) is a very common way of displaying the distribution of numerical data by showing the 0th, 25th, 50th (median), and 75th percentiles as a box, with whiskers extending to show the range of the data (excluding outliers). I usually skip them when I teach intro to stats because they are stupidly hard to make in Excel. But they are a snap in R.

Let's create a boxplot of the `wage` variable in the `CPS1985` dataset we've been using all chapter. If we want our boxplot to be oriented vertically, we can set `y = wage` (for a horizontal orientation we would set `x = wage`) as our aesthetic in `ggplot()`. A boxplot is a type of geometry, hence `geom_boxplot()`.

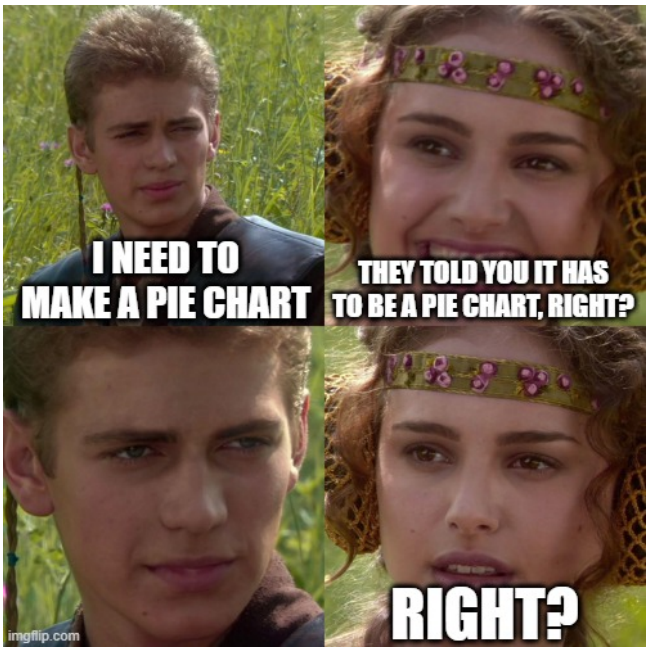
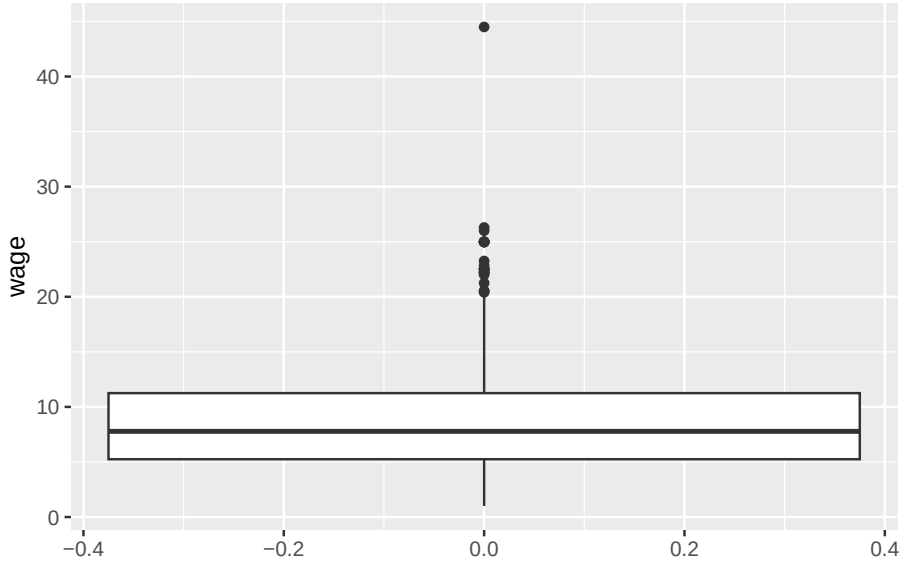


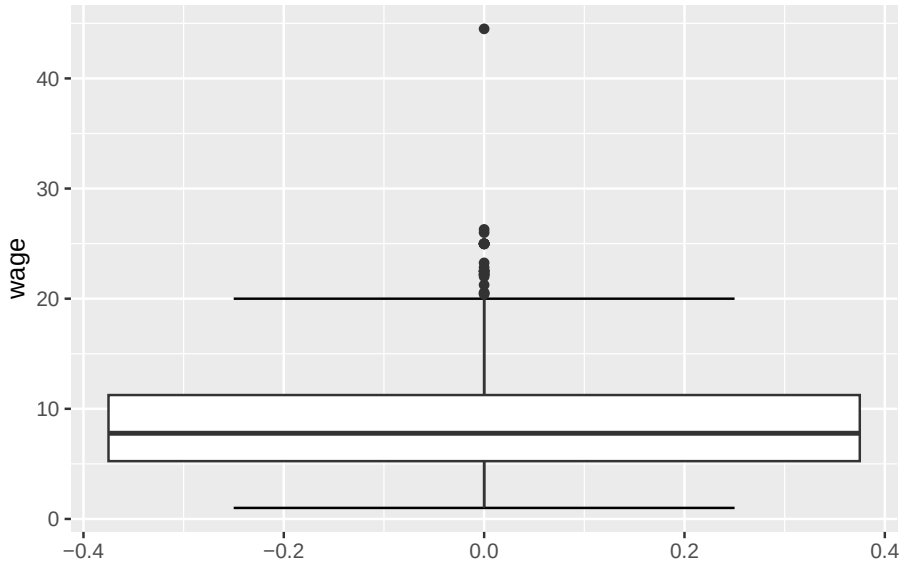
Figure 5.1

```
CPS1985 |>
  ggplot(aes(y = wage)) +
  geom_boxplot()
```



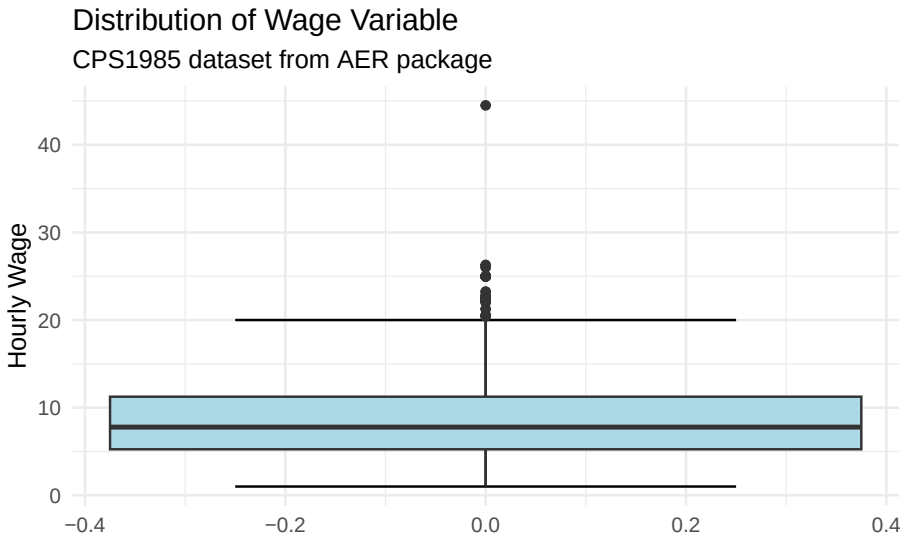
The boxplot shows us the 1st-3rd quartiles of the data, and the dots are the outliers. If we want to turn our boxplot into a box and whiskers plot, we need to add our whiskers with the `stat_boxplot` layer, as seen here:

```
CPS1985 |>
  ggplot(aes(y = wage)) +
  stat_boxplot(geom = "errorbar", width = 0.5) +
  geom_boxplot()
```



Finally, let's clean this up just a little:

```
CPS1985 |>
  ggplot(aes(y = wage)) +
  stat_boxplot(geom = "errorbar", width = 0.5) +
  geom_boxplot(fill = "lightblue") +
  theme_minimal() +
  labs(title = "Distribution of Wage Variable",
       subtitle = "CPS1985 dataset from AER package",
       y = "Hourly Wage",
       x = "")
```

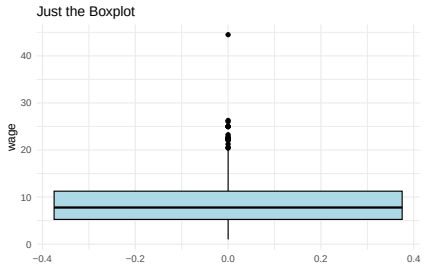


Data Storytelling: Layer Order Matters

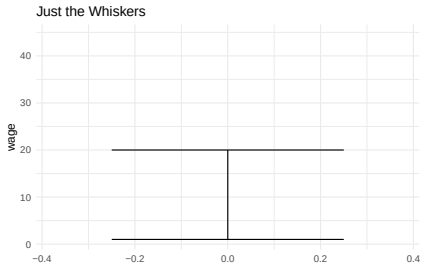
You may have noticed that, when I added the `stat_boxplot()`, I put it *before* the `geom_boxplot()`, and perhaps you wondered if it mattered. Turns out that yes, it did. When building `ggplot()` visualizations, layers are placed on top of each other in order, and I really wanted the `geom_boxplot()` *on top of* the `stat_boxplot()`. Let's take a look at the difference.

First, let's see what the individual parts of the box and whiskers plot are doing; I'm using the `fill = "transparent"` and `color = "transparent"` arguments (`fill` is for areas, `color` is for lines and points) to make different pieces of the graph transparent:

```
# Graph 1 code
CPS1985 |>
  ggplot(aes(y = wage)) +
  stat_boxplot(geom = "errorbar",  color = "transparent", width = 0.5) +
  geom_boxplot(fill = "lightblue", color = "black") +
  theme_minimal() +
  labs(title = "Just the Boxplot")
# Graph 2 code
CPS1985 |>
  ggplot(aes(y = wage)) +
  geom_boxplot(fill = "transparent", color = "transparent") +
  stat_boxplot(geom = "errorbar", width = 0.5) +
  theme_minimal() +
  labs(title = "Just the Whiskers")
```



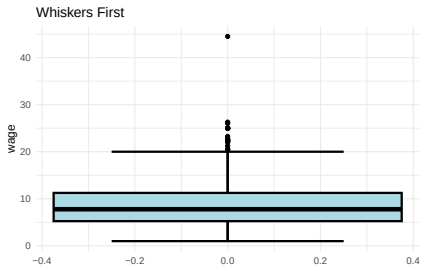
(a) Boxplot



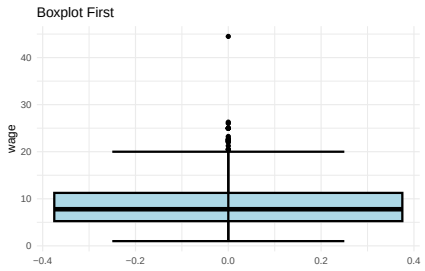
(a) Whiskers

Notice that the whiskers on the `geom_boxplot()` don't go through the box, but the `stat_boxplot()` line is solid. So let's see what happens when we switch the order of those lines.

```
# Graph 1 code
CPS1985 |>
  ggplot(aes(y = wage)) +
  stat_boxplot(geom = "errorbar", linewidth = 1, width = 0.5) +
  geom_boxplot(fill = "lightblue", color = "black", linewidth = 1) +
  theme_minimal() +
  labs(title = "Whiskers First")
# Graph 2 code
CPS1985 |>
  ggplot(aes(y = wage)) +
  geom_boxplot(fill = "lightblue", color = "black", linewidth = 1) +
  stat_boxplot(geom = "errorbar", linewidth = 1, width = 0.5) +
  theme_minimal() +
  labs(title = "Boxplot First")
```



(a) Whiskers First (Preferred)



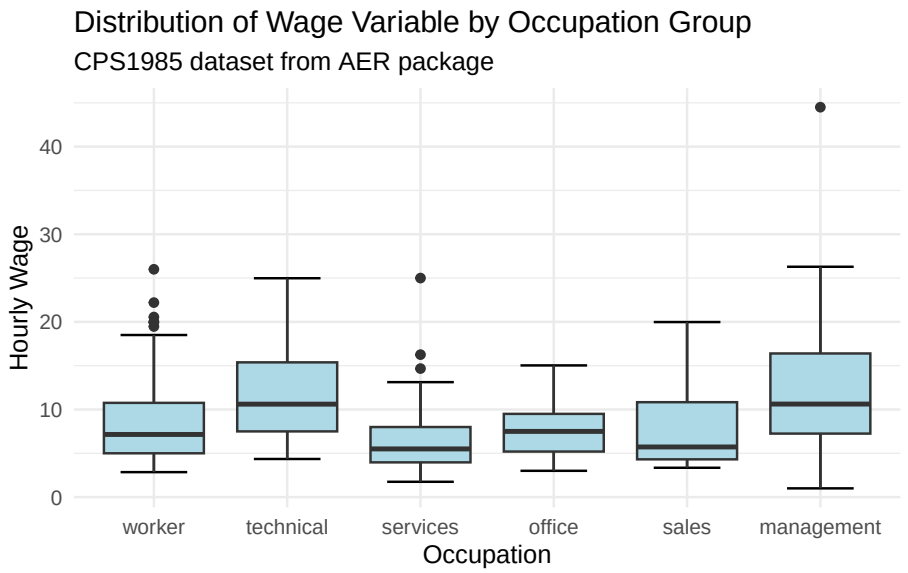
(a) Boxplot First (Not Preferred)

I thickened up the lines a bit to make the difference apparent. In the version where the `stat_boxplot()` comes first, the `geom_boxplot()` is placed on top of it and the vertical line is covered. When `geom_boxplot()` comes first, the full vertical line can be seen because the `stat_boxplot()` is drawn on top of the boxplot itself. The former is the preferred look, which is why I did it in the order I did.
But there is a useful lesson in this little digression. When making graphs with `ggplot()`, all the work is done in order from top to bottom, which means that **layer order matters**. Each new layer gets painted on top of the previous ones, so elements added later can cover up elements added earlier. Always think about what should be in front and what should be in back when building complex visualizations.

Typically, we wouldn't make a boxplot just to look at the distribution of an entire dataset (the super wide box is honestly pretty ugly, and Histograms are more appropriate for that); rather, boxplots are best when used to compare distributions across multiple groups. Let's take a look at the distribution of

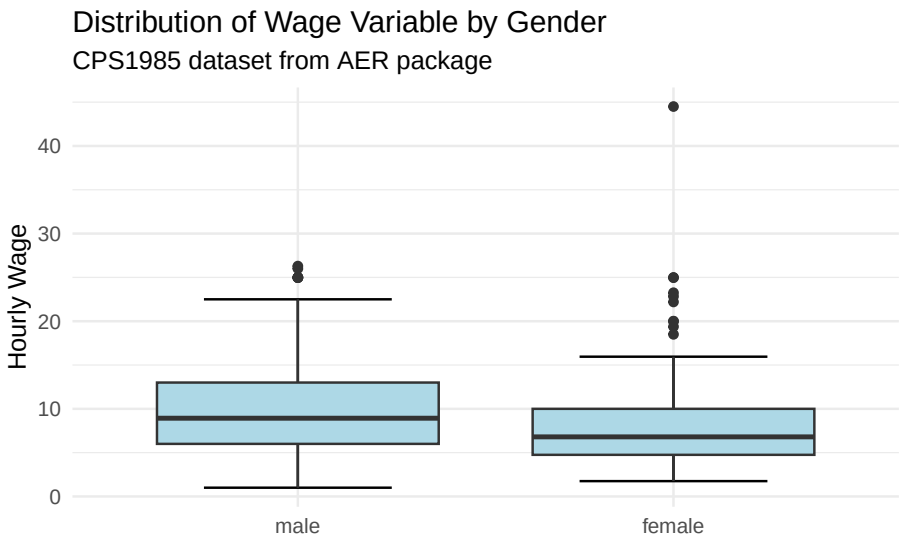
wages by each of the 6 occupation groups in the `CPS1985` dataset:

```
CPS1985 |>
  ggplot(aes(x = occupation, y = wage)) +
  stat_boxplot(geom = "errorbar", width = 0.5) +
  geom_boxplot(fill = "lightblue") +
  theme_minimal() +
  labs(title = "Distribution of Wage Variable by Occupation Group",
       subtitle = "CPS1985 dataset from AER package",
       y = "Hourly Wage",
       x = "Occupation")
```



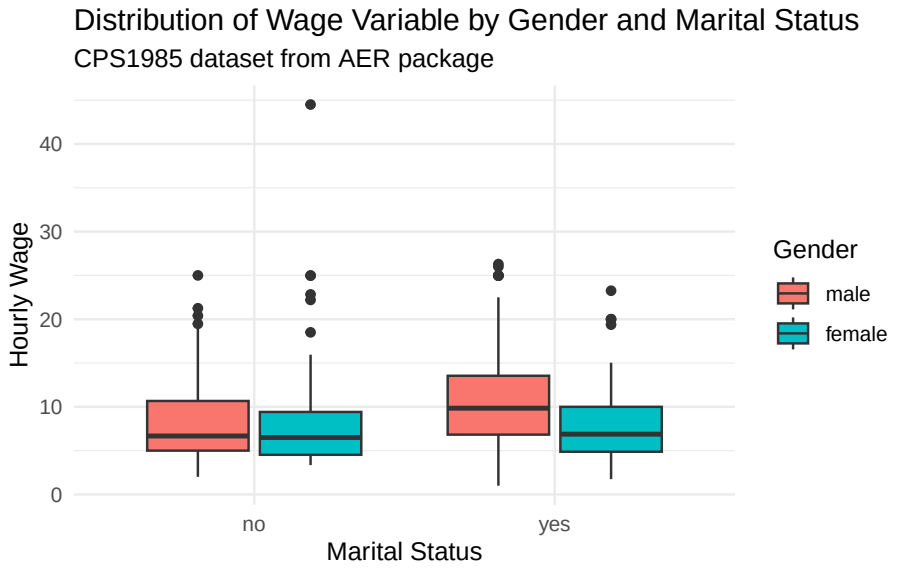
The aesthetic of `aes(x = occupation, y = wage)` tells the `ggplot()` that we are plotting wage on the y (vertical) axis and occupation on the x (horizontal) axis. We can get more complicated with our boxplots. Let's say we want to look at gender differences in the distribution of wage.

```
CPS1985 |>
  ggplot(aes(x = gender, y = wage)) +
  stat_boxplot(geom = "errorbar", width = 0.5) +
  geom_boxplot(fill = "lightblue") +
  theme_minimal() +
  labs(title = "Distribution of Wage Variable by Gender",
       subtitle = "CPS1985 dataset from AER package",
       y = "Hourly Wage",
       x = "")
```



Men seem to make more. Maybe your theory is that this gender wage gap is driven by marital status, so now we have 2 factor variables. We can include one of these factors as a fill aesthetic:

```
CPS1985 |>
  ggplot(aes(x = married, y = wage, fill = gender)) +
  geom_boxplot() +
  theme_minimal() +
  labs(title = "Distribution of Wage Variable by Gender and Marital Status",
       subtitle = "CPS1985 dataset from AER package",
       y = "Hourly Wage",
       x = "Marital Status",
       fill = "Gender")
```

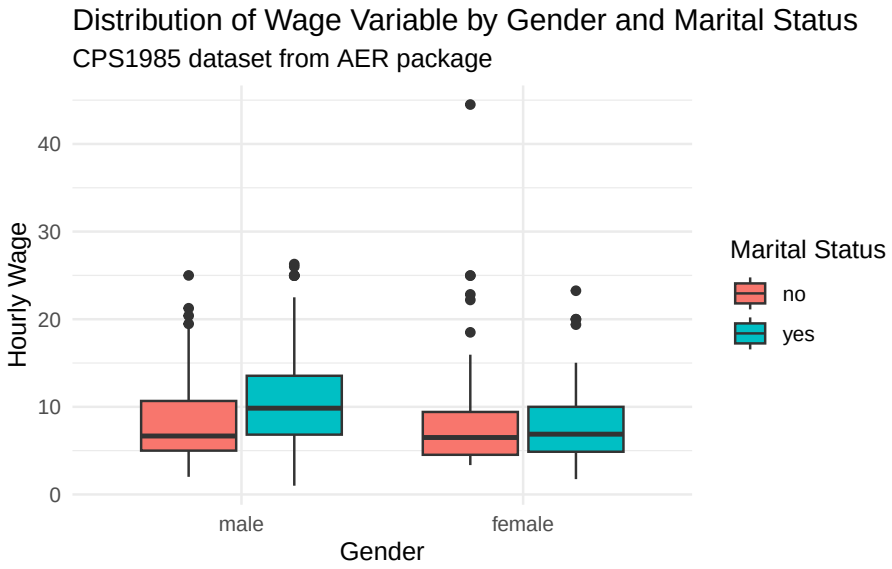


A few things to note about the above code:

- I removed the `stat_boxplot()` because it doesn't do a great job aligning itself with grouped boxes.
- I eliminated the `fill` option in `geom_boxplot`. Why? because I want the fill from the aesthetic to work! Right now, the genders have different colors for their boxplots. If I specified `fill` in the `geom_boxplot()`, because it happens *after* the `aes()`, the color coded genders will be overwritten!
- Variables in the `aes()` that are **not** your x and y variables - variables like `fill`, `color`, `size`, or `shape` - will automatically generate a legend. To name the legend, I added the `fill = "Gender"` line into the `labs()` function.

What happens if we put married as our fill variable and gender as our x variable?

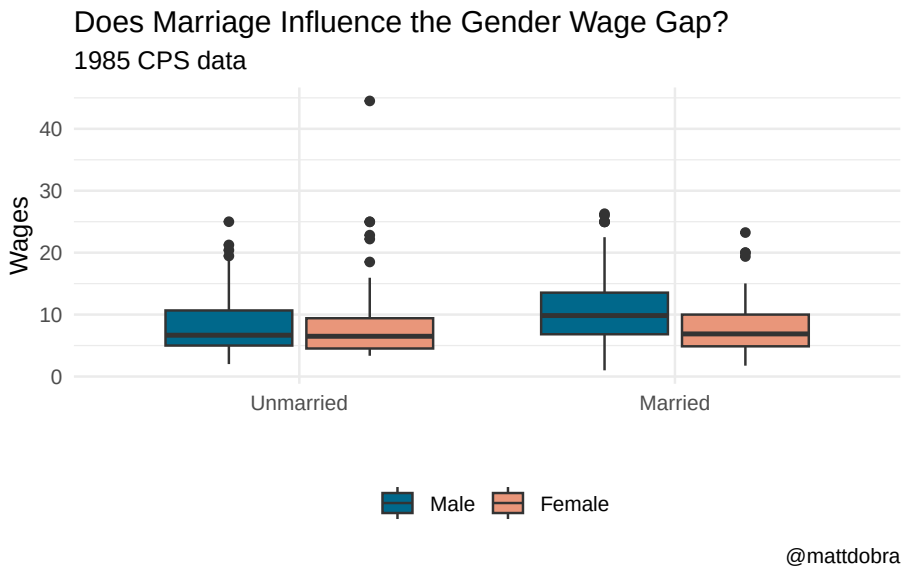
```
CPS1985 |>
  ggplot(aes(x = gender, y = wage, fill = married)) +
  geom_boxplot() +
  theme_minimal() +
  labs(title = "Distribution of Wage Variable by Gender and Marital Status",
       subtitle = "CPS1985 dataset from AER package",
       y = "Hourly Wage",
       fill = "Marital Status",
       x = "Gender")
```



Same data, same boxes, but the arrangement of the boxes tells a different story.

Finally, for fun, let's consider taking this last graph and elevating it a bit:

```
CPS1985 |> ggplot(aes(x = married, y = wage, fill = gender)) +
    geom_boxplot() +
    theme_minimal() +
    labs(title = "Does Marriage Influence the Gender Wage Gap?",
         subtitle = "1985 CPS data",
         x = "",
         y = "Wages",
         fill = "",
         caption = "@mattdobra") +
    scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                      labels = c("male" = "Male", "female" = "Female")) +
    theme(legend.position = "bottom",
          legend.direction = "horizontal") +
    scale_x_discrete(labels = c("no" = "Unmarried", "yes" = "Married"))
```



Here's a quick breakdown of the changes I made:

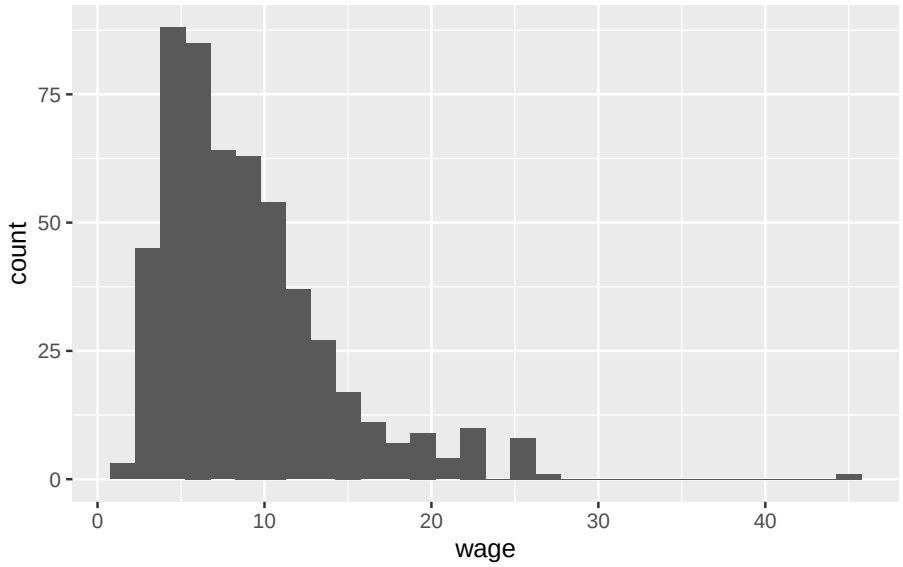
- Visual and Aesthetic Changes:
  - Custom colors: Replaced R's default red/teal with deepskyblue4 and darksalmon - a different take on blue/pink that's less jarring than stereotypical gender colors
  - Legend positioning: Moved legend to bottom with horizontal orientation for cleaner layout
  - Cleaner legend: Removed legend title (fill = " ") since"Male/Female" labels are self-explanatory
- Readability Improvements:
  - More compelling title: Changed from descriptive "Distribution of..." to research question "Does Marriage Influence the Gender Wage Gap?"
  - Better axis labels: "Unmarried/Married" instead of "no/yes" - no guessing required. This also allows me to remove the x-axis title entirely (`x = ""`) because the combination of our compelling graph title and self-explanatory axis labels means readers don't need that extra layer of explanation.
  - Capitalized legend labels: "Male/Female" instead of lowercase variable values
  - Added subtitle and caption: Context and attribution

5.1.4 Histograms

Like Boxplots, histograms are useful for displaying the distribution of continuous data. Also like Boxplots, these are much easier to create in R than in Excel! Since histograms and boxplots have similar vibes, let's continue to look at the distribution of respondent income in the CPS1985 data set so you can think about which works better for this data.

```
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram()
```

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



As you can see, we got a message telling us that `geom_histogram()` defaults to 30 bins, and the implied critism is that perhaps 30 is not the right number of bins for your data! This is generally a true statement, by the way, the right answer is probably not to stick with the defaults. We have options for fixing this:

- Change the width of the bins with `binwidth =`
- Set the number of bins explicitly with `bins =`
- Look the other way, set `#| message: false` in the YAML so the message doesn't pop up anymore, and move on with our lives. YOLO!

Choosing bins is equal parts art and science, and the authoritative "rules" for it aren't always up to the task when it comes to real world data, especially data describing human behavior. I've never particularly liked any treatment I've seen for choosing it in intro stats texts. So I'll share with you my general method for choosing bins:

1. Start with the $\sqrt{n}$. This gives a rough approximation of how many bins I should have.
2. Identify the range of the data, ignoring outliers, you probably want something like the 2nd percentile and the 98th percentile. Divide the range by the $\sqrt{n}$, this gives you a rough idea of what the right binwidth should be.
3. Identify round numbers and natural breakpoints to ensure the data isn't going to be too "lumpy." No, that's not a technical term. I'll explain more when I give an example!
4. Set binwidths based on the first three guidelines.
5. Don't be afraid to ignore the first 4 rules if needed.

So, for this case, we start with step 1 and note that the dataset includes 534 observations, and $\sqrt{534} \approx 23.11$. For step 2, our 2nd and 98th percentile values are \$3.35 and \$22.97, so a `binwidth` of around \$0.84 is a good starting point. But now for step 3: using binwidths of \$0.84 might create problems as real world hourly wages are far more likely to be \$8.00 or \$9.00 than \$8.01 or \$8.99, because humans like round numbers. And if we had bins of \$0.84 or whatever, we might wind up with a bin from \$8.03 to \$8.87 and it would look arbitrarily small compared to the bin that preceded it and the bin that came after it - not because wages are multimodal or something but just because of the lumpiness of the underlying numbers. So we probably want to instead think about using round numbers for our bins, so we are now thinking about intervals of \$1.00. So we will start there, but also consider maybe bigger numbers (binwidths of maybe \$1.50 or \$2.00) if it creates a smoother graph. But not too smooth…like I said, art and science.

To see what this all looks like in action, let's start with a graph that has the `binwidth` set way too low with `binwidth = 0.42`. Here, you can imagine bins

like $7.52 to $7.94 that will contain very few people with those wages, meanwhile the bins in which round numbers fall like $7.00 or $8.00 will look like spikes in the graph.

```
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram(binwidth = 0.42) +
    theme_minimal() +
    labs(title = "Binwidth = 0.42")
```

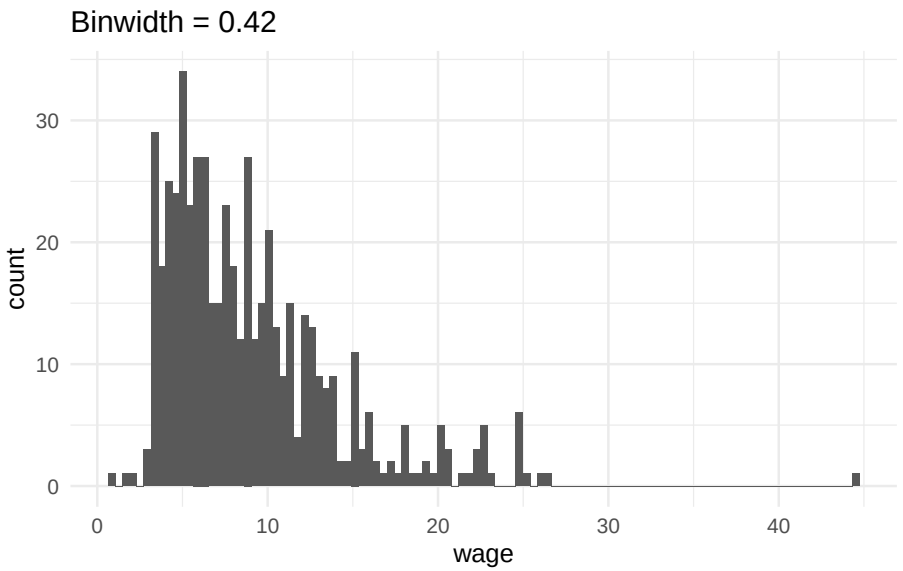


Figure 5.6: Binwidth = 0.42 (Super Lumpy)

The human tendency to like round numbers makes the data lumpy, and choosing too small bins makes the histogram spiky (also not a technical term). By the way, here is a list the wages in the dataset that appear at least 10 times:

Hourly Wage	Count
5.00	18
10.00	18
7.50	14
6.25	12
7.00	12
4.50	12
3.35	12
5.50	12
8.00	11
6.00	11

Hourly Wage	Count
4.00	10
12.00	10
15.00	10
3.50	10

Notice a pattern? The two most common are multiples of $5.00, then you have $7.50 (the halfway point between $5.00 and $10.00), and then $6.25 (the halfway point between $5.00 and $7.50). Over half of the entries are round numbers, and all but one are multiples of 0.25. The outlier, $3.35? That's the minimum wage in 1985!

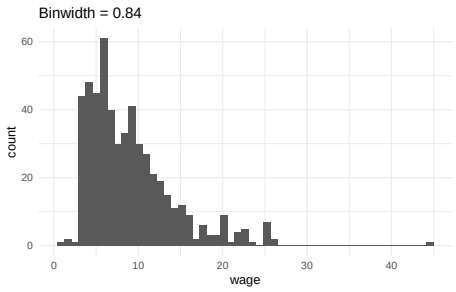
Data generated by human processes or about humans are notorious for this sort of lumpiness. And money is inherently lumpy, I guess, but think about something that is not. Human height. It's probably the case that the number of men in the world who are 72" tall (6 feet) is roughly the same as the number who are 71.9" tall or 72.1" tall, but who the heck says they are 72.1" tall? Nobody! And from what I can tell, most people who are 70" tall round up to 6 foot anyway…look guys, I'm a legit 6'2", I can see what you are doing from up here.

Moving on to the two mentioned above, the next two graphs set binwidths of $0.84 and $1.00, respectively:

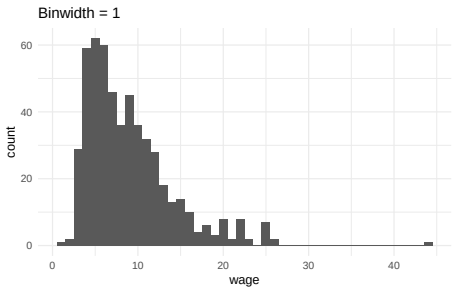
```
# Mathematical approach
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram(binwidth = 0.84) +
    theme_minimal() +
    labs(title = "Binwidth = 0.84")
# Round number approach
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram(binwidth = 1) +
    theme_minimal() +
    labs(title = "Binwidth = 1")
```

While not as pronounced as the `binwidth = 0.42` version, the `binwidth = 0.84` graph is indeed a little spikier than using `binwidth = 1`, I think I'd prefer the latter. We might even consider bigger binwidths to see what they look like; here are binwidths of $1.50 and $2.00:

```
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram(binwidth = 1.5) +
    theme_minimal() +
    labs(title = "Binwidth = 1.5")
CPS1985 |> ggplot(aes(x = wage)) +
    geom_histogram(binwidth = 2) +
```

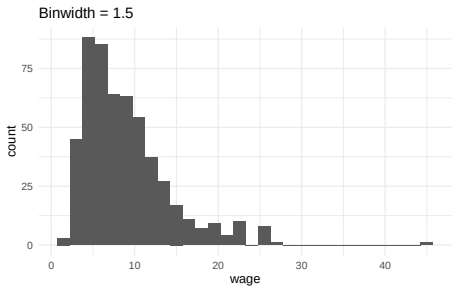


(a) Binwidth = 0.84 (Mathematical)

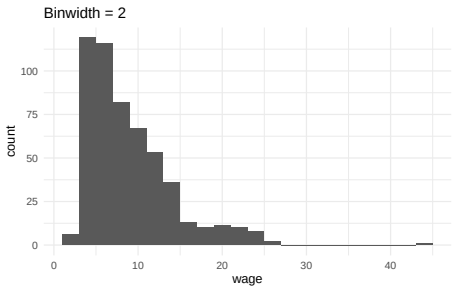


(a) Binwidth = 1 (Practical)

```
    theme_minimal() +
    labs(title = "Binwidth = 2")
```



(a) Binwidth = 1.5 (Smoother)



(a) Binwidth = 2 (Very Smooth)

Now we are getting to the point that we are eliminating a lot of the artifacts that are creating the artificial lumpiness in the data, but at a cost of losing precision. I'll say it again, art and science. I think either the $1.00 or $$2.00 graph work the best for me, so let's go ahead and clean up the $2.00 by setting some common options that are typical when using `geom_histogram()`:

```
CPS1985 |> ggplot(aes(x = wage)) +
  geom_histogram(binwidth = 2,
                 fill = "lightsteelblue",
                 color = "steelblue") +
  theme_minimal() +
  theme(panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank()) +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  labs(title = "Distribution of Wages",
       subtitle = "Data from 1985 CPS",
```

```
        caption = "Sample Size: 534",
        y = "Count",
        x = "Hourly Wage")
```

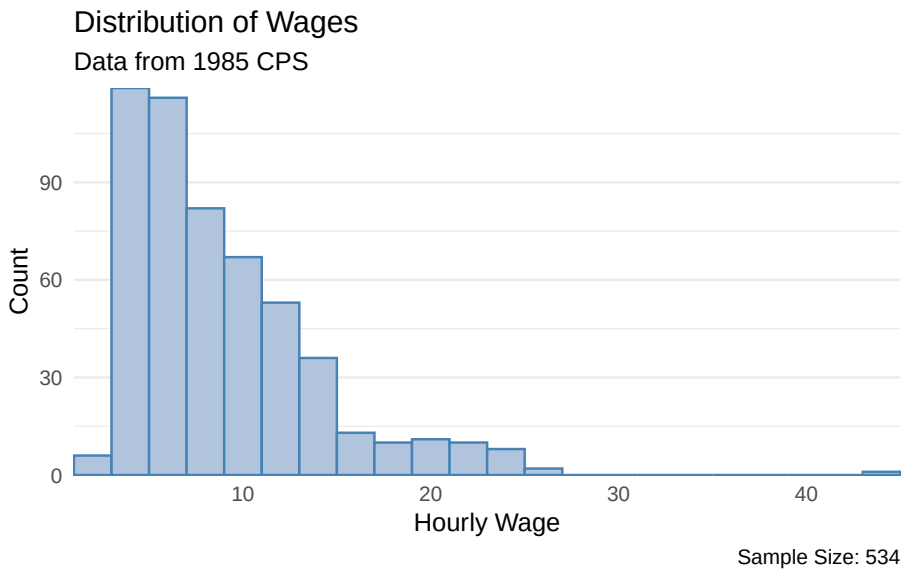


Figure 5.11: Histogram

Just a few finishing touches make this histogram much more polished:

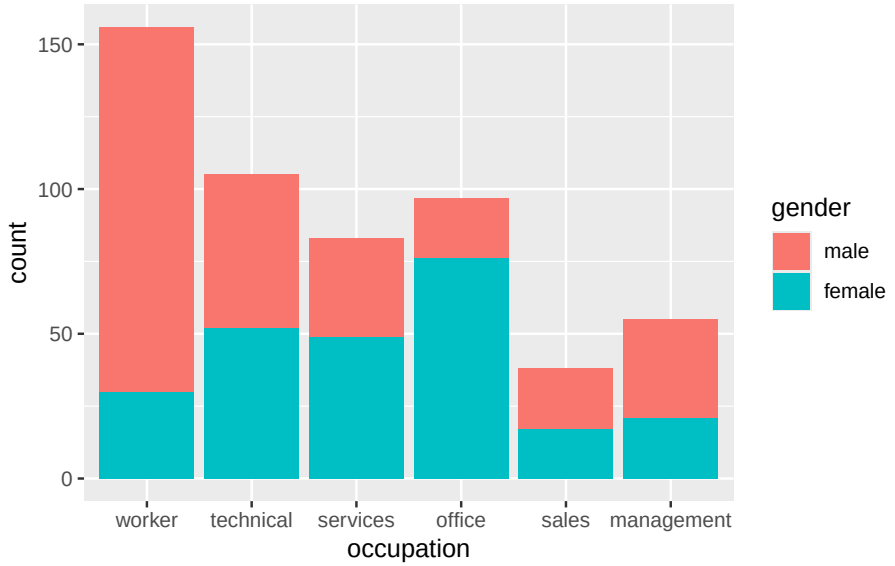
- Custom colors: The `fill` and `color` options give us a cohesive blue theme instead of the default gray. Remember, `color` is for points and lines (0 and 1 dimensional objects) while `fill` is for areas (2 dimensional spaces)
- Cleaner gridlines: Removing vertical gridlines with `theme()` since they're not particularly helpful for histograms
- Tighter spacing: `expand = c(0,0)` eliminates the awkward white space between the axes and the data
- Complete labeling: Title, subtitle, and caption provide context, and capitalizing "Count" is cleaner than the default "count"

5.1.5 Stacked and Grouped Bar Charts (and Pie Charts I Guess)

Stacked and Grouped Bar Charts are typically used to look at compositions of groups...that is to say, groups within groups. First you group, then you subgroup. This is a pretty simple augmentation to the `geom_bar()` from above, where all we need to do is set both an `x` variable for our groups, and a `fill` variable

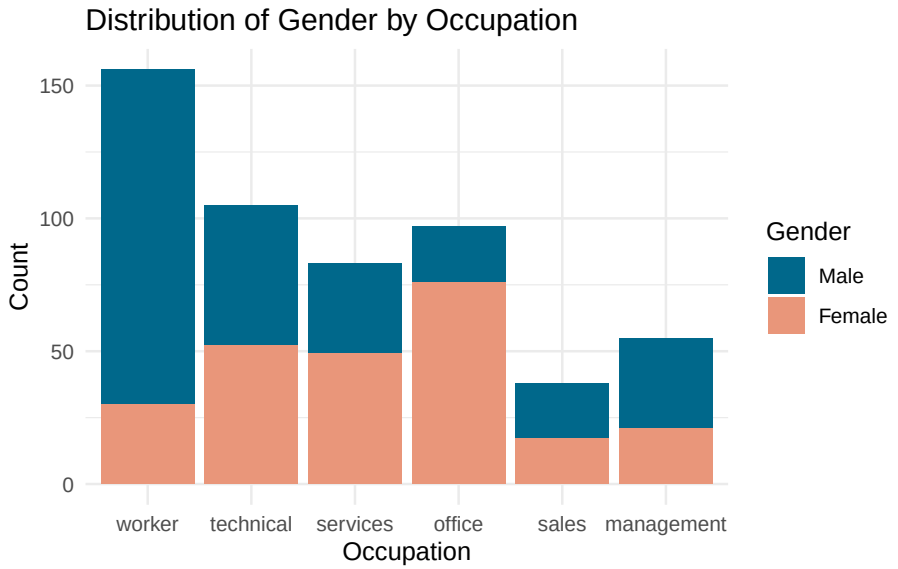
for our subgroups within our groups. This looks at the distribution of genders within each occupation class in the `CPS1985` data:

```
CPS1985 |>
  ggplot(aes(x = occupation, fill = gender)) +
  geom_bar()
```



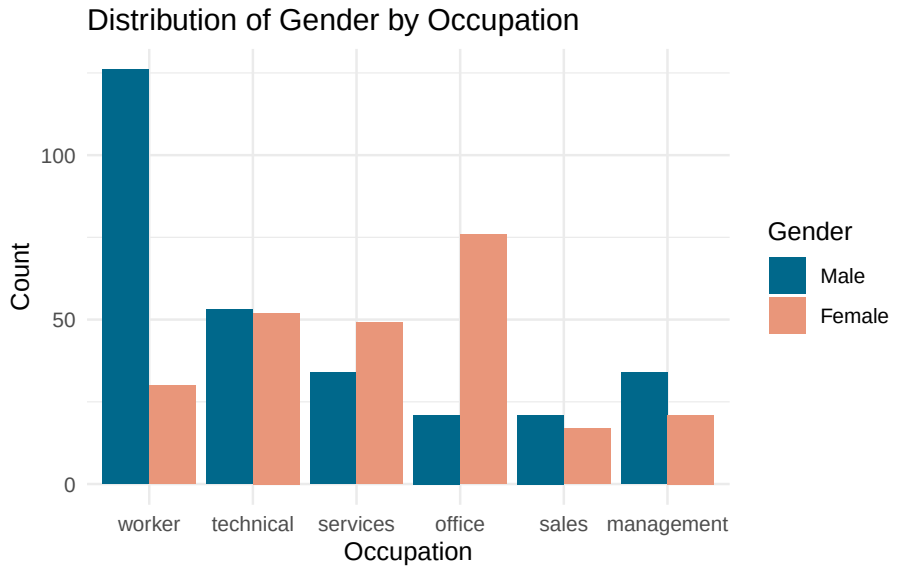
Let's clean this up and definitely edit the colors. Gender stereotype colors (i.e. blue for men, pink for women) probably enhance readability to the viewer, but if you want to avoid them, then you probably don't want to do the exact opposite of what people expect! We'll stick with our blue/salmon combo that's less jarring but still provides good contrast. Plus, nothing is easier than doing a copy/paste on the `scale_fill_manual()` line from a few graphs back that we *know* works, as well as give this chart some labels:

```
CPS1985 |>
  ggplot(aes(x = occupation, fill = gender)) +
  geom_bar() +
  theme_minimal() +
  scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                    labels = c("male" = "Male", "female" = "Female")) +
  labs(x = "Occupation",
       y = "Count",
       fill = "Gender",
       title = "Distribution of Gender by Occupation")
```



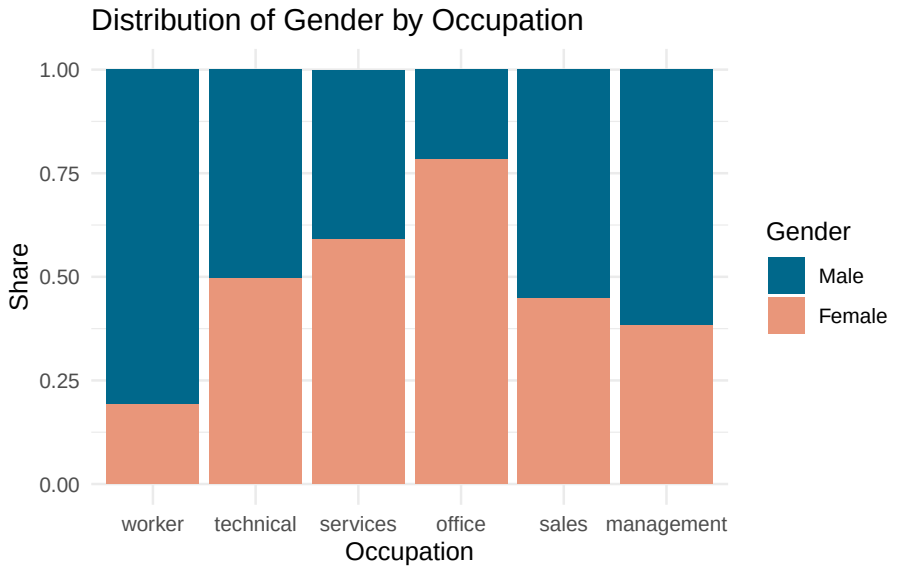
Rather than stacking the bars, we can create side-by-side bars by adding the `position = "dodge"` option in the `geom_bar()` layer.

```
CPS1985 |>
  ggplot(aes(x = occupation, fill = gender)) +
  geom_bar(position = "dodge") +
  theme_minimal() +
  scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                    labels = c("male" = "Male", "female" = "Female")) +
  labs(x = "Occupation",
       y = "Count",
       fill = "Gender",
       title = "Distribution of Gender by Occupation")
```



A common use of the stacked bar chart is to compare subgroup proportions between groups. For example, in the graphs above, we can see that there are more people in the technical category than in sales, but it's harder to compare the relative gender percentages across occupations. Is the percentage of men in technical higher or lower than in sales? And by how much? To accomplish this, we use the `position = "fill"` option in our `geom_bar()` layer.

```
CPS1985 |>
  ggplot(aes(x = occupation, fill = gender)) +
  geom_bar(position = "fill") +
  theme_minimal() +
  scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                    labels = c("male" = "Male", "female" = "Female")) +
  labs(x = "Occupation",
       y = "Share",
       fill = "Gender",
       title = "Distribution of Gender by Occupation")
```



This version of the stacked bar chart makes it easier to compare subgroup proportions across groups.

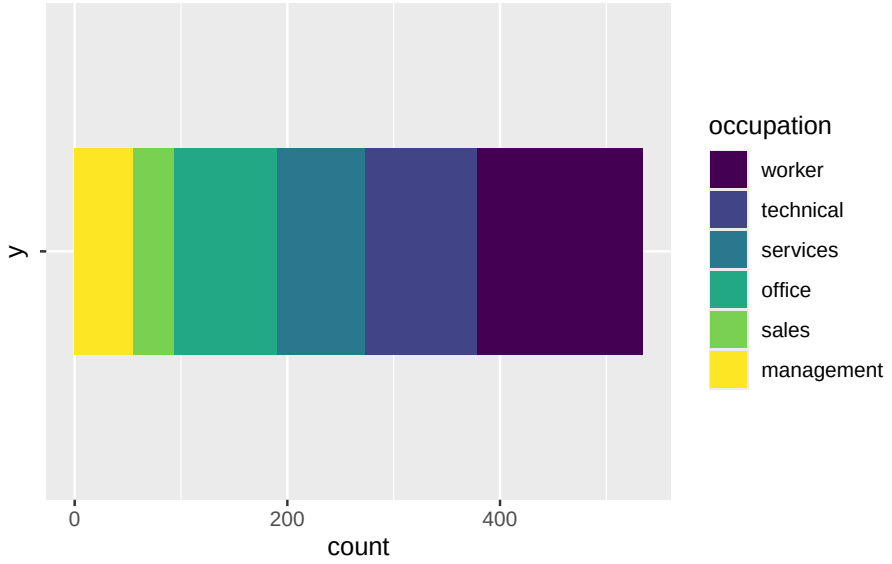
5.1.5.1 Pie Charts

And we are back at pie charts. I kindly ask that you refer back to my Padme/Anakin meme (Figure **??**) from above before continuing to read.

Most people's default understanding of a pie chart is as a circle that is used to represent proportions; the circle is divided into wedges of various sizes, where each wedge represents a group, and the size of each wedge indicates the proportion of the whole that group makes up. This is all well and good, but to make a pie chart in `ggplot()`, we need to develop a different intuition: a pie chart is a stacked bar chart that you have forced to be round.

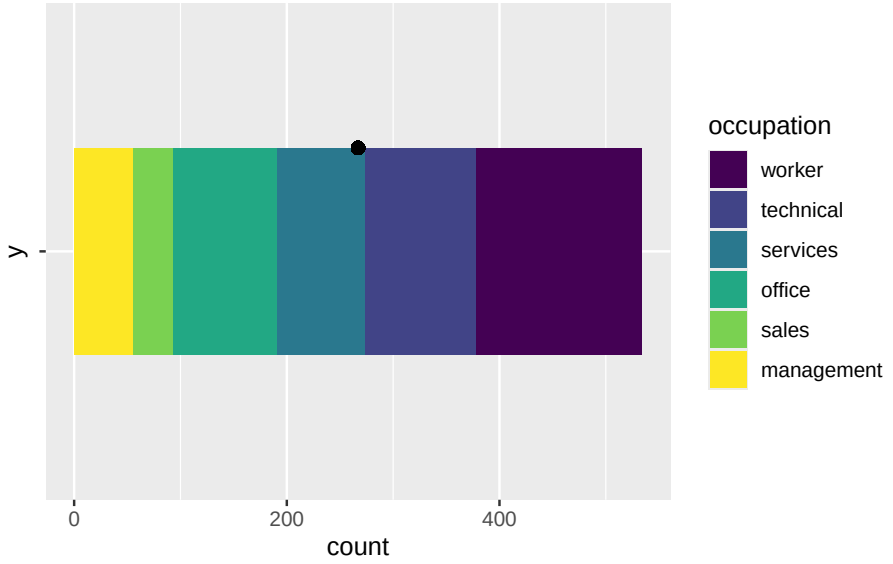
To see what I mean, let's start with a simple stacked bar chart of the occupation data. The code is fairly straightforward, the only oddity here is the use of the `y = ""` in the `aes()` argument. The effect of this is to simply select the entire dataset. Setting the width of the `geom_bar()` is not necessary, it just aids in my being able to place a visual cue in the next graph. I'm also using the `viridis` color scheme to aid with graph visibility.

```
CPS1985 |>
  ggplot(aes(y = "", fill = occupation)) +
  geom_bar(width = 0.5) +
  scale_fill_viridis_d()
```



Behold this horizontal stripe of rainbowy goodness! Next, let's place a reference dot along the top of the bar, exactly in the middle of the x dimension. I've done this below with the `geom_point()` layer. :

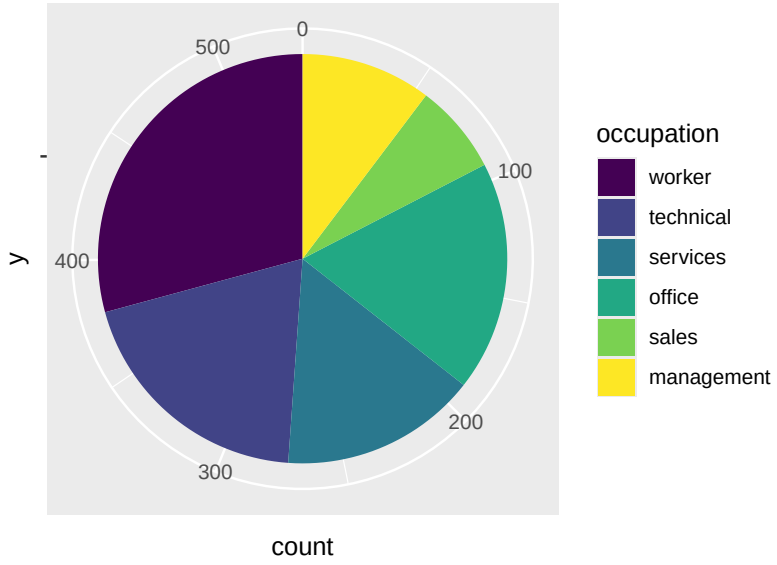
```
CPS1985 |>
  ggplot(aes(y = "", fill = occupation)) +
  geom_bar(width = 0.5) +
  scale_fill_viridis_d() +
  geom_point(y = 1.25, x = 267, size = 2.2, show.legend = FALSE)
```



Now, imagine what would happen if you took this shape and started bending the two ends upward, using the black dot as a pivot point. This will cause the top edge to shrink and the bottom edge to stretch. Keep bending this shape around that pivot point until the top edge has shrunk to a single point, the bottom edge has created a circle that completely surrounds the black dot, and the two ends connect, so the far left yellow edge is now flush with the far right purple edge. Break the spirit of the noble stacked bar chart, and literally bend it to your will! Now you have a pie chart!

More mathily, we can say that we converted your cartesian coordinate system (the standard (x, y) space you learned in high school algebra) into a polar coordinate system. So we take the stacked bar from above, and add one line of code: `coord_polar(theta = "x")`. And that's all it takes, we have a pie chart:

```
CPS1985 |>
  ggplot(aes(y = "", fill = occupation)) +
  geom_bar(width = 0.5) +
  scale_fill_viridis_d() +
  coord_polar(theta = "x")
```

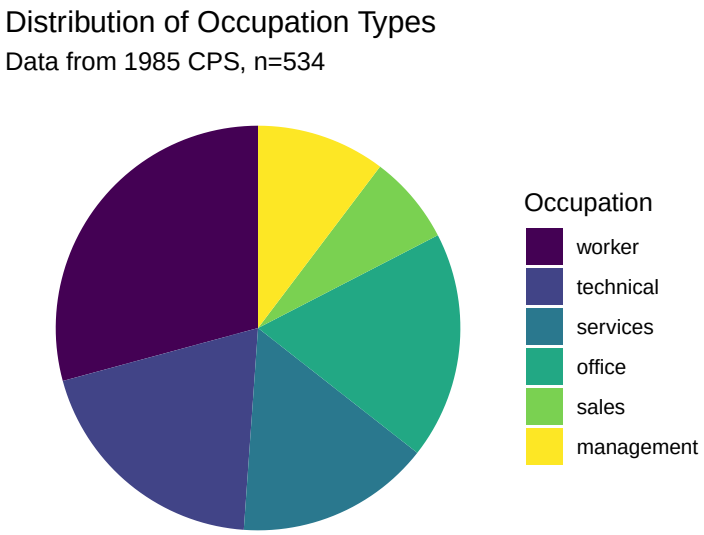


What is going on with the `theta = "x"` argument? In a polar coordinate system, θ (or theta) is the term used to refer to the angle of a line radiating from the center of the coordinate system. In the original stacked bar chart, the x-axis was counting how many people were in each group, so the `theta = "x"` argument is telling `ggplot()` to use the x variable from the stacked chart to determine the angles of the wedges. So the ugly thin white line around your pie chart? That's actually your x-axis!

With pie charts, you often want to use the `theme_void()` argument to get rid of all backgrounds and axes and such, and add the stuff you want in manually:

```
CPS1985 |>
  ggplot(aes(y = "", fill = occupation)) +
  geom_bar() +
  theme_void() +
  scale_fill_viridis_d() +
  coord_polar(theta = "x") +
```

```
  labs(title = "Distribution of Occupation Types",
       subtitle = "Data from 1985 CPS, n=534",
       fill = "Occupation")
```



5.1.5.2 Donut Charts

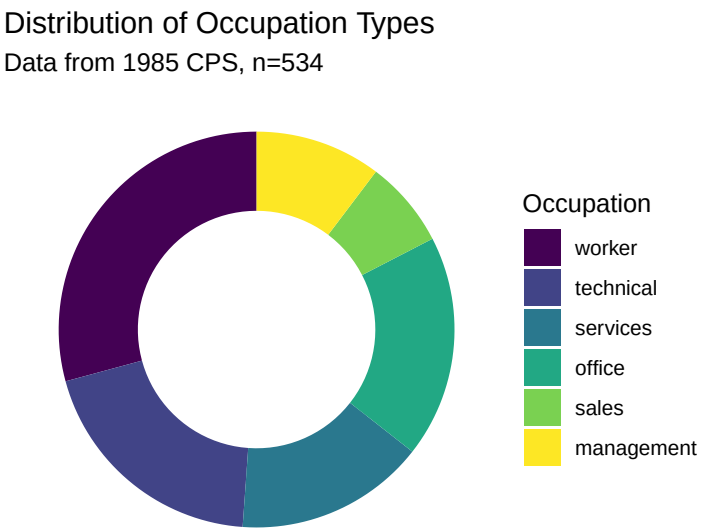
We can use this methodology to make some donut charts too. A donut chart is a pie chart with a hole in it. Because somebody thought "how can we make a pie chart worse?" But…you've come this far. Why not see what embracing evil really looks like?

We start by adjusting our `aes()` call to have `y = 2`. The key insight is that you're setting up a coordinate system where you can then "cut out" the middle.

The specific number matters because it defines your working space, but the exact value (2, 5, etc.) is flexible based on what proportions you want to achieve. As a rule of thumb, the skinnier the donut, the bigger the `y`.

Now that `y` is a number, we can use the `ylim()` argument to cut out our donut hole. The first number in `ylim()` determines how big a hole we cut–it has to be less than `y` but bigger than 0, and smaller numbers mean we cut bigger holes. The second number controls how much empty space you want around your donut; generally you don't want to go more than 0.5 bigger than your `y` value. The code below sets `ylim(0.2, 2.5)`, and the proportions seem reasonably nice.

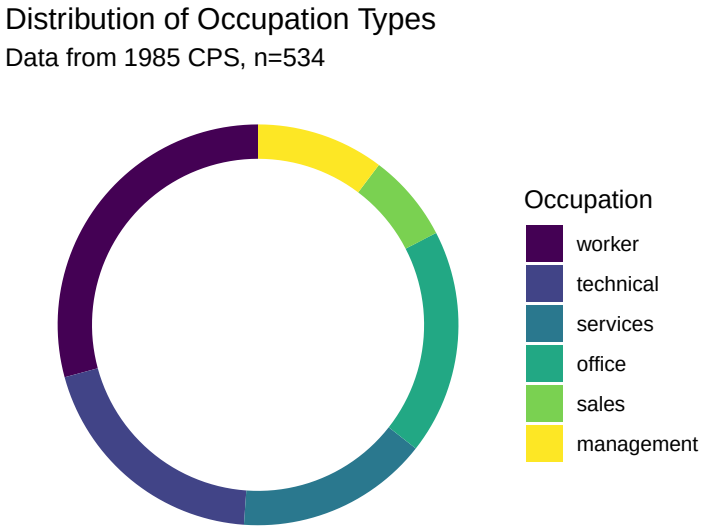
```
CPS1985 |>
  ggplot(aes(y = 2, fill = occupation)) +
  geom_bar() +
  theme_void() +
  scale_fill_viridis_d() +
  coord_polar(theta = "x") +
  labs(title = "Distribution of Occupation Types",
       subtitle = "Data from 1985 CPS, n=534",
       fill = "Occupation") +
  ylim(0.2, 2.5)
```



Feel free to play with the `y = 2` option and the `ylim()` parameters to see how you can control the size and shape of the donut. For example, making `y` (and the corresponding upper limit in `ylim()`) bigger makes your donut skinnier:

```
CPS1985 |>
  ggplot(aes(y = 5, fill = occupation)) +
```

```
  geom_bar() +
  theme_void() +
  scale_fill_viridis_d() +
  coord_polar(theta = "x") +
  labs(title = "Distribution of Occupation Types",
       subtitle = "Data from 1985 CPS, n=534",
       fill = "Occupation") +
  ylim(.2, 5.5)
```



5.1.6 Scatter Plots

Scatter plots are useful for looking at the relationship between 2 numerical (preferably continuous) variables. Because there is only 1 continuous variable in the `CPS1985` dataset (wage, the rest are discrete), let's switch to using the `vote1` dataset from the `wooldridge` package. To see what is in this, type `?vote1` in your console. We might also want to get a quick overview of the data with the `head(vote1)` command:

```
head(vote1)
```

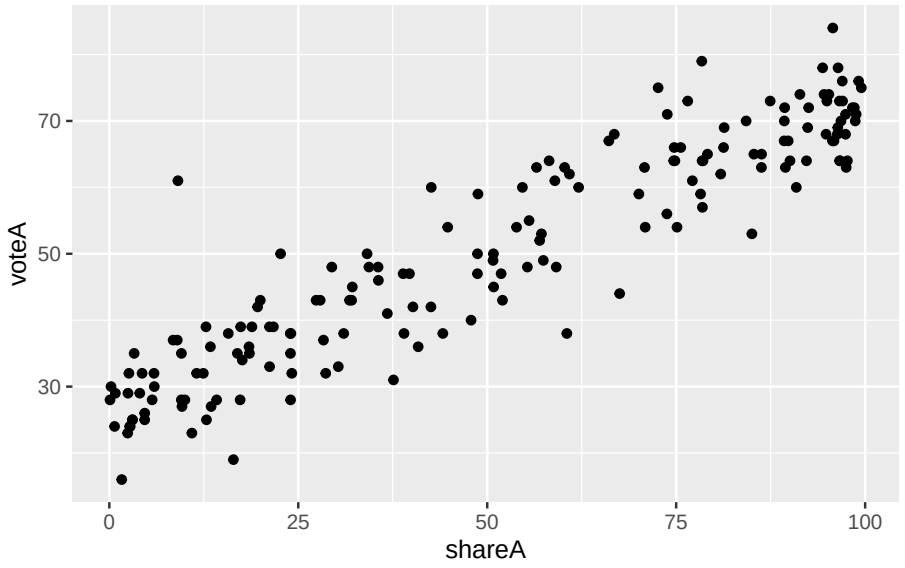
```
  state district democA voteA expendA expendB prtystrA lexpendA lexpendB
1    AL        7      1    68 328.296   8.737       41 5.793916 2.167567
2    AK        1      0    62 626.377 402.477       60 6.439952 5.997638
3    AZ        2      1    73  99.607   3.065       55 4.601233 1.120048
4    AZ        3      0    69 319.690  26.281       64 5.767352 3.268846
5    AR        3      0    75 159.221  60.054       66 5.070293 4.095244
6    AR        4      1    69 570.155  21.393       46 6.345908 3.063064
```

```
    shareA
1 97.40767
2 60.88104
3 97.01476
4 92.40370
5 72.61247
6 96.38355
```

This is a pretty fun dataset because it allows us to look at the extent to which money influences elections in the US.

Let's look at the relationship between the share of campaign expenditures (`shareA`) and vote share received (`voteA`). We need to set `x` and `y` aesthetics, and the geometry of a scatterplot is `geom_point()`:

```
vote1 |>
  ggplot(aes(x = shareA, y = voteA)) +
  geom_point()
```



Why `(x = shareA, y = voteA)` and not `(y = shareA, x = voteA)`? In this case, theory tells that vote share should be the **dependent variable** and campaign expenditures should be the **independent variable**, and when graphing the generally accepted norm is to put the DV (dependent variable) on the Y axis.

Data Storytelling: R Doesn't Know Causality

Here's the thing: R doesn't get causality. You could totally flip this and put `voteA` on the x-axis and `shareA` on the y-axis, and R would happily create a perfectly functional scatterplot.

But you know something about how the world works. This is what separates "I can make graphs in R" from "I can actually analyze data." Sure, anyone can plot variables against each other, but understanding which variable is trying to influence which requires you to think about the real world beyond your computer screen. Does campaign spending influence election outcomes, or do election outcomes magically determine how much money campaigns spent months earlier?

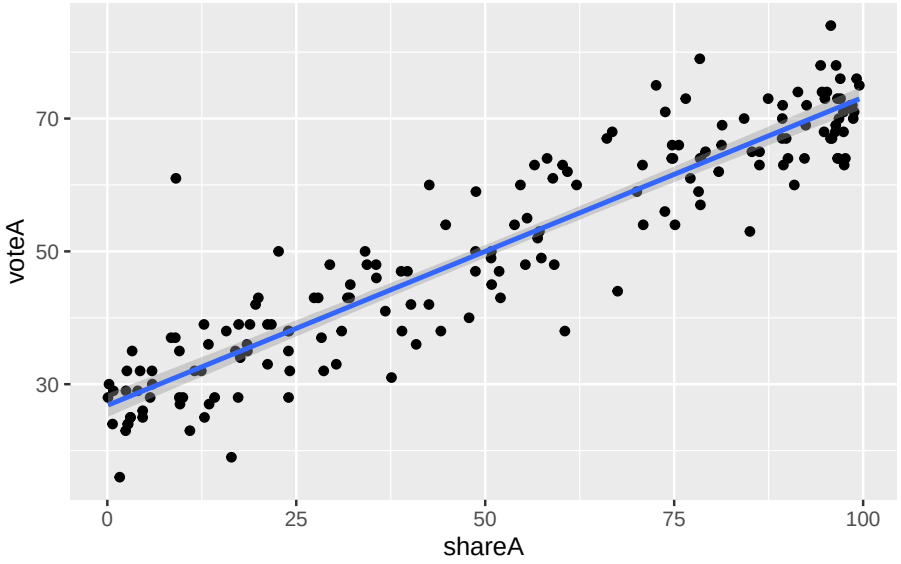
Spoiler alert: time generally moves forward, not backward. R doesn't know this. Maybe R thinks that election results magically travel backward through time to force donors in the past to give money to the winners. It's like a Spider-verse Canon Event for all R knows. R quite literally just sees numbers and does math. It just follows orders.

Whether you're looking at marketing spend vs. sales, study time vs. test scores, or coffee consumption vs. productivity (definitely a causal relationship there), your job is to bring some actual knowledge about how things work to the table. R will crunch the numbers, but it won't tell you what makes sense.

Perhaps you want to see the line of best fit? We can add the `geom_smooth(method = lm)` layer to our `ggplot()`:

```
vote1 |>
  ggplot(aes(x = shareA, y = voteA)) +
  geom_point() +
  geom_smooth(method = lm)
```

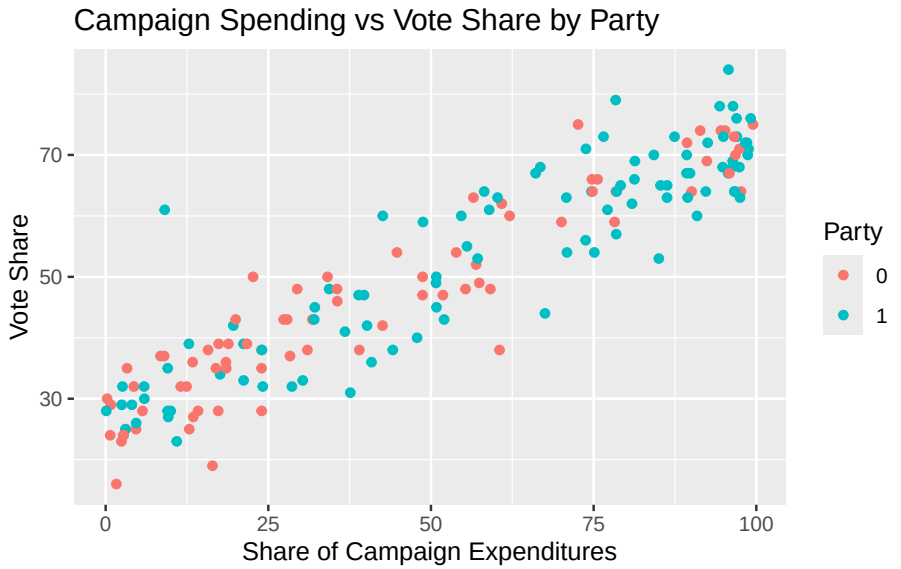
```
`geom_smooth()` using formula = 'y ~ x'
```



> **May the Format be With You: Silencing Chatty Functions**
>
> When you run `geom_smooth()`, R likes to tell you it's `using formula = y ~ x`…basically just confirming it's drawing a straight line. While this transparency is nice, it clutters your output. If you are incorporating lines of best fit into charts in a Quarto document, you almost always want to add `#| message: false` to your code chunk to tell R to zip it.
> For the scatter diagram above, I didn't suppress the message, but I will do so for the rest of them.

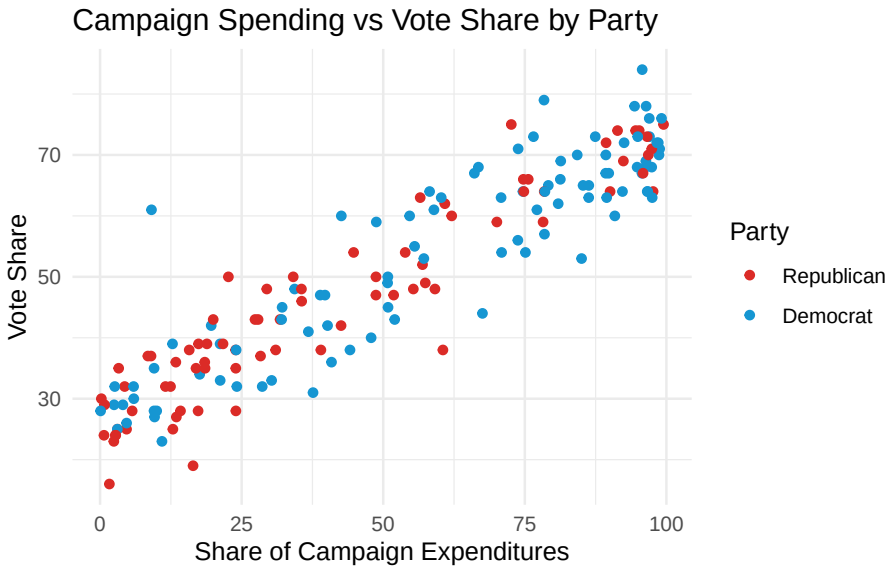
As before, we can add elements if we want here. For example, maybe we want to highlight which party each candidate belongs to. The `democA` variable uses 1 for Democrats and 0 for Republicans, but `ggplot()` wants factors for color coding, so we need to wrap it in `as.factor()`:

```
vote1 |>
  ggplot(aes(x = shareA, y = voteA, color = as.factor(democA))) +
  geom_point() +
  labs(title = "Campaign Spending vs Vote Share by Party",
       x = "Share of Campaign Expenditures",
       y = "Vote Share",
       color = "Party")
```



Let's make this more readable by using proper party colors and labels:

```
vote1 |>
  ggplot(aes(x = shareA, y = voteA, color = as.factor(democA))) +
  geom_point() +
  scale_color_manual(values = c("0" = "#db2b27", "1" = "#1696d2"),
                     labels = c("0" = "Republican", "1" = "Democrat")) +
  labs(title = "Campaign Spending vs Vote Share by Party",
       x = "Share of Campaign Expenditures",
       y = "Vote Share",
       color = "Party") +
  theme_minimal()
```



May the Format Be With You: Brand Consistency (or: How to Look Like You Actually Know What You're Doing)

Want to avoid looking like an amateur who just discovered hex codes? Here's a pro tip from the design world: stop typing the same color codes over and over like some kind of data visualization peasant.
Notice how I just typed `"#db2b27"` and `"#1696d2"` in the graph above? Sure, I could keep doing that throughout the entire document, making typos, forgetting exact codes, and writing code that will produce a result that looks sloppy or lazy. Or I could do what the pros do and set up my colors as variables at the beginning:

```
# Political party colors - define once, use everywhere
dem_blue <- "#1696d2"
rep_red <- "#db2b27"
```

Which is honestly just being lazy in a smart way. Because then, throughout your document, you can simply reference these variables. For example:

```
ggplot(data, aes(x = category, y = value)) +
  geom_bar(fill = dem_blue, color = rep_red) +
  theme_minimal()
```

Why is this approach so much better?

- Consistency: All your graphs will use exactly the same colors - no more "is this the right shade of blue?" moments
- Efficiency: No more hunting for hex codes or copying/pasting from old graphs

- Flexibility: Need to rebrand everything? In my lecture slides for my Principles of Macro classes, I set up my documents with variables named color1 through color5; all I need to do is change the colors in those five lines at the top, and every color in the document gets an instant makeover.
- Professional vibes: Your document gets that polished, branded look that screams "I know what I'm doing"

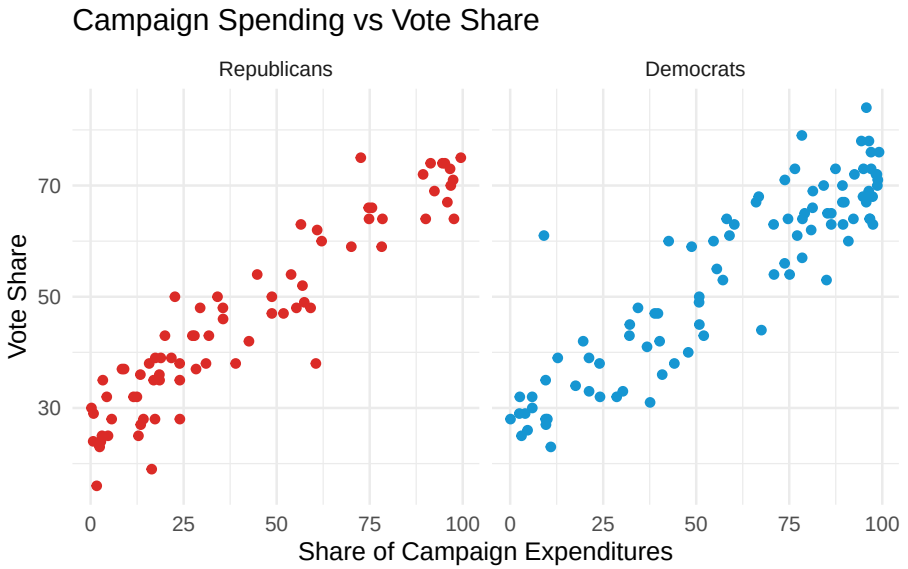
Whether you're trying to impress a boss, avoid getting roasted by your professor, or just want to make something look super professional, this simple technique will level up your work significantly.

Let's put this advice into action. First, we'll set up our color variables:

```
# Political party colors - define once, use everywhere
dem_blue <- "#1696d2"
rep_red <- "#db2b27"
```

Now we can create separate graphs for the Democrats and Republicans using facet_wrap() with our color variables:

```
vote1 |>
  ggplot(aes(x = shareA, y = voteA)) +
  geom_point(aes(color = as.factor(democA))) +
  scale_color_manual(values = c("0" = rep_red, "1" = dem_blue)) +
  facet_wrap(~democA,
             labeller = labeller(democA = c("0" = "Republicans", "1" = "Democrats"))) +
  labs(title = "Campaign Spending vs Vote Share",
       x = "Share of Campaign Expenditures",
       y = "Vote Share") +
  theme_minimal() +
  theme(legend.position = "none")  # Remove legend since facet labels tell the story
```



All in all, R has some extremely powerful graphing capabilities that go well beyond what something like Excel is capable of. These days, when people ask me for help making graphs in Excel, I cringe a little because I know how much better it would look in R. Once you wrap your head around the Grammar of Graphics … aesthetics, layers, themes, etc. … `ggplot()` becomes way more intuitive than clicking through Excel's endless and counterintuitive menu system. Sure, there's a learning curve up front, but the payoff is graphs that actually look professional and really, really, ridiculously good looking.

5.2 Summarizing Data Numerically

Now that we have seen an overview of the graphical capabilities, let's turn to numerical summaries. We return to the `CPS1985` dataset so we can look at both numerical and categorical variables.

5.2.1 Categorical Variables

Let's start with categorical variables and focus on occupation. Categorical variables are summarized with frequencies or proportions. The `table()` command (we saw this above when we made Bar and Pie Charts) works to create these; by default, `table()` summarizes counts.

```
table(CPS1985$occupation)
```

```
    worker  technical   services     office      sales management
       156        105         83         97         38         55
```

Here's the same analysis of the sector variable:

```
table(CPS1985$sector)
```

```
manufacturing  construction         other
           99            24           411
```

OK, typing CPS1985$ over and over is getting pretty cumbersome. There's a shortcut for this called `attach()` that can make your life easier when working with a single dataset.

> Tip from the Helpdesk: To Attach or Not to Attach?
>
> The attach() function lets you refer to variables in a dataset without typing the dataset name every time. So instead of CPS1985$occupation, you could just type occupation. Sounds great, right?
> So, let's start with why this doesn't work (yet):
>
> ```
> table(occupation)
> ```
>
> ```
> Error in eval(expr, envir, enclos): object 'occupation' not found
> ```
>
> Why not? Because R is looking for a variable called `occupation` to make a `table()` with, and it doesn't know where it is. R isn't smart enough to look inside of the `CPS1985` dataset for a variable called `occupation`, so it spits out an error. When I `attach()` the `CPS1985` dataset, I'm simply telling R that it should feel free to look inside the `CPS1985` dataset for variables. In a lot of ways, it's kinda like how libraries work. When I turn on R, it doesn't know what the heck a `ggplot()` is. When I run `library(tidyverse)`, I'm telling R to look inside the `tidyverse` family of functions when I give it a command. If I run `ggplot()` *after* I load the `tidyverse`, R will be able to figure out what `ggplot()` means.
> So, let's attach the dataset and see what happens with that same funcion that broke earlier.
>
> ```
> attach(CPS1985)
> table(occupation)  # No more CPS1985$ needed!
> ```
>
> ```
> occupation
>     worker  technical   services     office      sales management
>        156        105         83         97         38         55
> ```
>
> Ha! Now, you might be wondering why I didn't start with this? Way to bury the lede, right?
> While attach() can be a useful crutch when you're learning R and is fine when you're only working with a single dataset, it can create confusion in larger projects. What happens when you have multiple datasets with variables that have the same name? Which age variable is R using - the one from dataset A or dataset B?

> Honestly, it's probably worth just biting the bullet and learning to use the `$` notation. You'll see both approaches in the wild, but explicit `dataset$variable` notation will save you headaches down the road when your projects get more complex. Your future self will thank you for the clarity!
> For this book, I plan on sticking with the explicit notation, but I reserve the right to switch if the typing gets really tedious or I just get really lazy.

Let's convert those counts to proportions. The easiest way is to divide your totals by the size of the dataset:

```
table(CPS1985$occupation) / nrow(CPS1985)
```

```
    worker  technical   services     office      sales management
0.29213483 0.19662921 0.15543071 0.18164794 0.07116105 0.10299625
```

You can also produce contingency (two-way) tables that look at the intersection of two categorical variables:

```
table(CPS1985$occupation, CPS1985$gender)
```

```
            male female
  worker      126     30
  technical    53     52
  services     34     49
  office       21     76
  sales        21     17
  management   34     21
```

This shows us the breakdown of gender within each occupation category in 1985. Now we can see, for example, that the sales and technical roles were roughly evenly split between the genders, while workers tended to be male and office jobs went to females.

5.2.2 Numerical Variables

While we are limited in how we describe categorical variables, we have lots of options with respect to numerical variables. Let's analyze the wage variable.

Arithmetic means are calculated with the `mean()` function.

```
mean(CPS1985$wage)
```

```
[1] 9.024064
```

A trimmed mean drops the outliers from the top and bottom of the data. For example, if we type:

```
mean(CPS1985$wage, trim = 0.05)
```

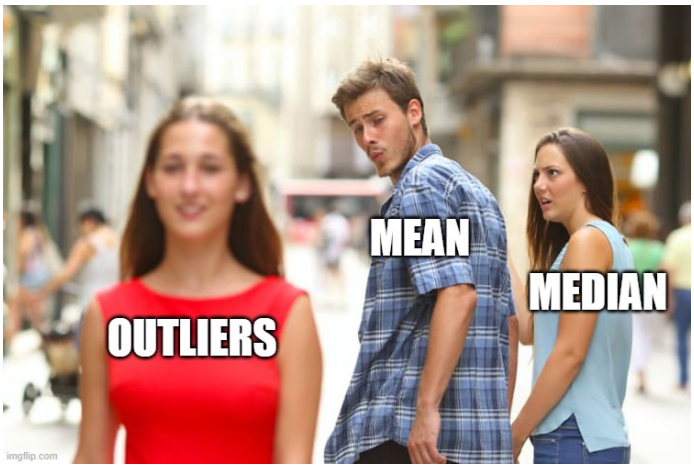
```
[1] 8.544212
```

We tell R to drop the 5% of the lowest wages and 5% of the highest wages from the data and calculate the mean of the middle 90%. Sometimes this makes sense when you are looking at a data set with a lot of skew. Another thing we might do to get a measure of central tendency for skewed data is to calculate a `median()`.

```
median(CPS1985$wage)
```

```
[1] 7.78
```

Generally speaking, medians are the preferred measure of central tendency in cases with skewed data because medians are far less sensitive to the presence of outliers in the data…



Variance and Standard Deviation are calculated with `var()` and `sd()`, respectively.

```
var(CPS1985$wage)
```

```
[1] 26.41032
```

```
sd(CPS1985$wage)
```

```
[1] 5.139097
```

Don't forget, the standard deviation is the square root of the variance!

```
sd(CPS1985$wage)^2
```

```
[1] 26.41032
```

```
sqrt(var(CPS1985$wage))
```

```
[1] 5.139097
```

Minima and maxima can be calculated with the `min()` and `max()` commands.

```
min(CPS1985$wage)
```

```
[1] 1
```

```
max(CPS1985$wage)
```

```
[1] 44.5
```

You can get both easily if you want using `range()`

```
range(CPS1985$wage)
```

```
[1]  1.0 44.5
```

Measures of position (quartiles, percentiles, etc) can be obtained through the `quantile()` function. You need to pass through a `probs` argument to tell it which quantile you want.

If you just want one specific quantile–in this case the first quartile–you type:

```
quantile(CPS1985$wage, .25)
```

```
 25%
5.25
```

You can also use the `c()` language we've seen before to get multiple quantiles at once. For example, if we want the 10th, 25th, 50th, 75th, and 90th percentile, we type:

```
quantile(CPS1985$wage, probs = c(.1, .25, .5, .75, .9))
```

```
   10%    25%    50%    75%    90%
 4.000  5.250  7.780 11.250 15.275
```

The minimum and maximum of the data can also be obtained using the 0th and 100th quartile:

```
quantile(CPS1985$wage, probs = c(0,1))
```

```
  0% 100%
 1.0 44.5
```

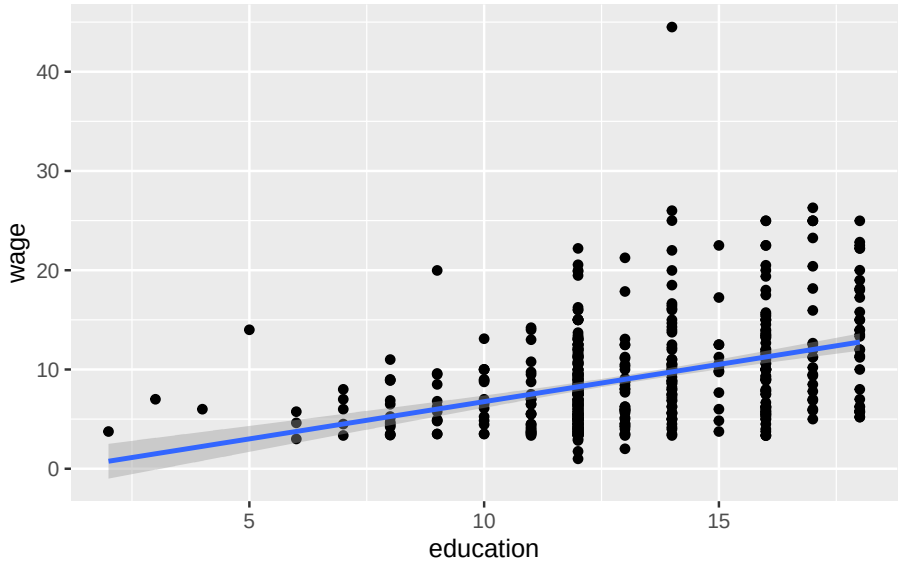
Correlation coefficients can be calculated using `cor()`. Here is the correlation between education and wage:

```
cor(CPS1985$wage, CPS1985$education)
```

```
[1] 0.3819221
```

This indicates a moderate positive correlation between the two variables. We can see this correlation in this graph:

```
CPS1985 |> ggplot(aes(x = education, y = wage)) +
    geom_point() +
    geom_smooth(method = lm)
```



By default, R calculates the Pearson correlation, which is appropriate for interval data. Sometimes you have ordinal data where a Spearman correlation makes more sense, for example, when the data is a ranking but not a measure. In this case, the Pearson is quantifying the linear relationship between education and wage, but a Spearman test simply asks whether or not higher levels of education are linked to higher wages.

```
cor(CPS1985$wage, CPS1985$education, method = "spearman")
```

```
[1] 0.3813425
```

In most economic applications, Pearson is more useful.

Don't forget, the summary() command can be useful here too:

```
summary(CPS1985$wage)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   5.250   7.780   9.024  11.250  44.500
```

The `stargazer()` package/function makes nicely formatted tables of summary statistics, but you have to put in a whole dataframe:

```
stargazer(CPS1985, type = "text")
```

```
==========================================
Statistic   N   Mean  St. Dev.  Min   Max
------------------------------------------
wage       534 9.024   5.139   1.000 44.500
education  534 13.019  2.615     2     18
experience 534 17.822  12.380    0     55
age        534 36.833  11.727   18     64
------------------------------------------
```

It is also capable of making more visually attractive, publication ready tables:

```
stargazer(CPS1985, type = "html")
```

Statistic

N

Mean

St. Dev.

Min

Max

wage

534

9.024

5.139

1.000

44.500

education

534

13.019

2.615

2

18

experience

534

17.822

12.380

0

55

age

534

36.833

11.727

18

64

You can't see it in my code chunk, but to get the `type = "html"` to work right the code chunk needs to have an option of `#| results: asis` or else you will get something funky! We will make extensive use of `stargazer()` later in this book, so file away this tip in your brain for later!

5.3 Wrapping Up

This chapter notebook reviewed what is essentially the first half of an intro to statistics class and introduced how to use R for these calculations. The next step is to tackle inferential statistics using R, which we turn to in Chapter **??**.

5.4 End of Chapter Exercises

Charts and Graphs: Ensure that your charts include axis labels and titles. Use `ggplot()` for all graphics!!

Throughout this book, datasets are given in the form of `package:dataset`. To access them, you first load the relevant package, and then load the dataset you wish to use. For example, to use the `wooldridge:wine` dataset from question 7, you would use the following commands:

```
library(wooldridge)
data(wine)
```

Bar Charts:

1. Look at `fivethirtyeight:hiphop_cand_lyrics` and make bar charts of `sentiment` for both Donald Trump and Hillary Clinton. What does this tell you about how hip-hop artists viewed these candidates? What happens if you focus on specific years or break down by different sentiment themes?

2. Use the `AER:BankWages` data to make bar charts of `job`. Which job type is the most common in the dataset? Create side-by-side or stacked bar charts to examine whether there are different patterns between genders. What does this suggest about gender representation across different banking positions?

3. For the `AER:BankWages` data, calculate summary statistics for `education` by job type, ethnicity, and gender. Create a summary table showing the mean and median years of education for each combination. Do the numerical summaries reveal patterns about educational requirements or ethnicity/gender differences that weren't obvious in your bar charts from question 2? Do different genders/ethnicities require more/less education to get the premium managerial jobs?

4. Create bar charts looking at the distribution of student `gender` and `ethnicity` in the `AER:STAR` data. Consider making both separate charts and grouped/stacked charts. What does this tell you about the student population in this study?

Box Plots and Histograms:

5. Using the `AER:NMES1988` dataset, create a histogram of the number of physician office visits (`visits`) and a box plot examining the number of `visits` by `health` status. What does the distribution of visits look like? How do healthcare utilization patterns differ by health status? Are there outliers, and what might they represent?

6. Using the `dplyr:starwars` dataset, create a histogram of the `height` of Star Wars characters and a box plot looking at `height` by the `gender` of the characters. What does this tell you about the diversity of character sizes in the Star Wars universe? Are there notable differences between genders?

7. Using the `dplyr:starwars` dataset, calculate summary statistics (mean, median, standard deviation, quartiles) for `height` by `gender`. How do these numerical summaries compare to what you observed in the box plots from question 5? Which measure of central tendency (mean vs. median) is more appropriate for this data and why?

Scatter Plots:

8. Use `datasets:USArrests` to look at the relationship between urbanization (`UrbanPop`) and each of the three measures of crime (`Murder`, `Assault`, and `Rape`). Create three separate scatter plots. What patterns do you observe? Does urbanization appear to be related to crime rates? Which crimes show the strongest relationship with urbanization? Including lines of best fit might help with this analysis.

9. Using `datasets:USArrests`, calculate correlation coefficients between `UrbanPop` and each crime variable. How do these correlations compare to

your visual impressions from the scatter plots in question 8? Calculate summary statistics (mean, median, standard deviation) for each crime type - which crimes show the most variability across states?

10. Look at the relationship between alcohol consumption (`alcohol`) and `deaths`, `heart`, and `liver` in `wooldridge:wine`. Create three scatter plots and consider adding trend lines. What do these relationships suggest about the health effects of alcohol consumption? Do the results surprise you?

11. Using `wooldridge:meap01`, look at the relationship between expenditures per student (`expp`) and the various test scores (`math4` and `read4`). Create scatter plots for both relationships. Does more spending appear to be associated with better test scores? Are there outliers that might represent particularly efficient or inefficient school districts?

Thinking about Graphs:

12. Short answer: Compare the histogram and boxplots of `wages` above. Which visualization better represents the distribution of wages, and why? Consider what each type of graph emphasizes and what information might be lost or highlighted in each approach. When would you choose one over the other?
13. Looking at your analysis from question 10 (`wooldridge:wine`), discuss the difference between correlation and causation. Just because alcohol consumption is correlated with health outcomes, can we conclude that alcohol causes these outcomes? What other factors might be involved?
14. Choose one of your analyses from questions 5-11 where you identified outliers. Research or speculate about what might cause these outliers. Should they be removed from analysis, or do they represent important information? How might outliers affect your conclusions?
15. The `AER:STAR` data (question 4) comes from a specific educational study. Based on your analysis of the demographics, discuss how representative this sample might be of the broader U.S. student population. What are the limitations of drawing general conclusions from this specific dataset?
16. You've now created histograms, box plots, bar charts, and scatter plots. For each type of visualization, describe: (a) what type of data it's best suited for, (b) what patterns it reveals well, and (c) what information it might obscure. If you had to choose only one type of graph to explore a new dataset, which would you pick and why?
17. In several of your analyses, you likely encountered missing data, unusual values, or data that required interpretation. Choose one example and discuss how data quality issues might affect your conclusions. What steps could researchers take to improve data quality in future studies?

Chapter 6

Inferential Statistics with R

This chapter provides a brief review of inferential statistics, including both basic univariate tests and some multivariate models, such as the t-test, ANOVA, and chi-square (χ^2). The other major family of multivariate models, regression, will be given a much more detailed treatment starting in Chapter @ref(basicreg), as regression modeling is the cornerstone of econometrics.

As always, let's start by loading the relevant libraries and datasets into memory.

```
library(tidyverse)
library(openintro)
library(Ecdat)
library(wooldridge)
library(AER)

data(fact_opinion)
data(k401k)
data(CPS1985)
data(Fair)

# Setup Colors!
colorlight <- "lightsteelblue"
colordark <- "steelblue"
```

6.1 One-Sample t-test

Let's start simple, with the humble 1-sample t-test. A 1-sample t-test is used to test a hypothesis about the mean of a specific variable.

Fun fact about the t-distribution: it was developed by a dude named William Sealy Gosset who was just trying to use statistics to make better beer at Guin-

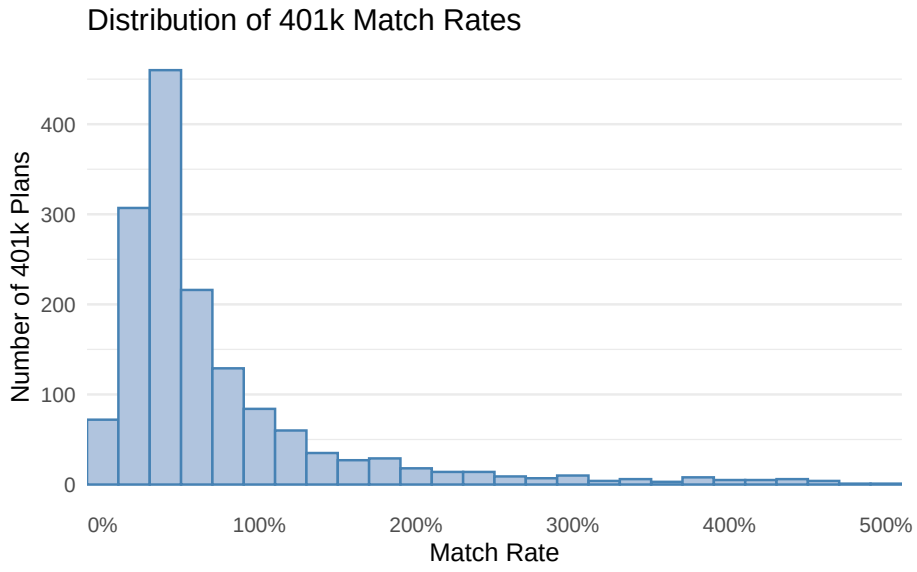
ness. But Guinness didn't let him publish under his own name; I think it's because they didn't want the world to know they were using statistics to make better beer, and they thought of it as a trade secret. Or maybe that they didn't want people to think that Guinness was beer for math nerds, who knows? Either way, he had to publish the thing under a pseudonym: Student. Hence, the t-distribution is often referred to as the Student t-distribution. So, let's not forget that statistics comes from a good place!

We can illustrate the 1-sample t-test using the `k401k` data set from the `wooldridge` package. As always, `?k401k` is a good way to get information about a data set.

```
?k401k
```

This data includes features about 1534 401k plans in the US. Suppose we are interested in the 401k match rate (i.e. for every dollar the 401k owner puts in, how much will the employer match with?), which is found in the `mrate` variable. To get a general overview of the data, we can look at the distribution of the contribution match rate with a simple histogram:

```
k401k|> ggplot(aes(x = mrate)) +
  geom_histogram(binwidth = .2,
                 fill = colorlight,
                 color = colordark) +
  theme_minimal() +
  theme(panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank()) +
  scale_x_continuous(expand = c(0, 0),
                     labels = scales::percent_format()) +
  labs(y = "Number of 401k Plans",
       x = "Match Rate",
       title = "Distribution of 401k Match Rates")
```



Just eyeballing the graph, it's hard to get a handle on what the mean is. It looks like the median and modal match rate are probably 50%, but it seems like there are some plans with some pretty generous match rates out there (not anywhere I've ever worked, sadly) as this data is heavily right-skewed.

Suppose somebody tells you that the average (mean) match rate among 401k plans is 75%. Maybe it's the HR guy at a company trying to hire you, trying to convince you that their company's 75% match rate is competitive. Or maybe it's the union rep where you work, trying to encourage workers to strike until the company raises their 50% match rate which is way lower than the industry standard. Or maybe it's a line in an article in the Wall Street Journal or Barron's or the like talking about the current state of 401k programs in the USA. In any event, now I'm curious, and testing whether or not this is true is where the one-sample t-test comes in.

We can summarize the data and see that the mean in the data is 73.15%:

```
summary(k401k$mrate)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.0100  0.3000  0.4600  0.7315  0.8300  4.9100 
```

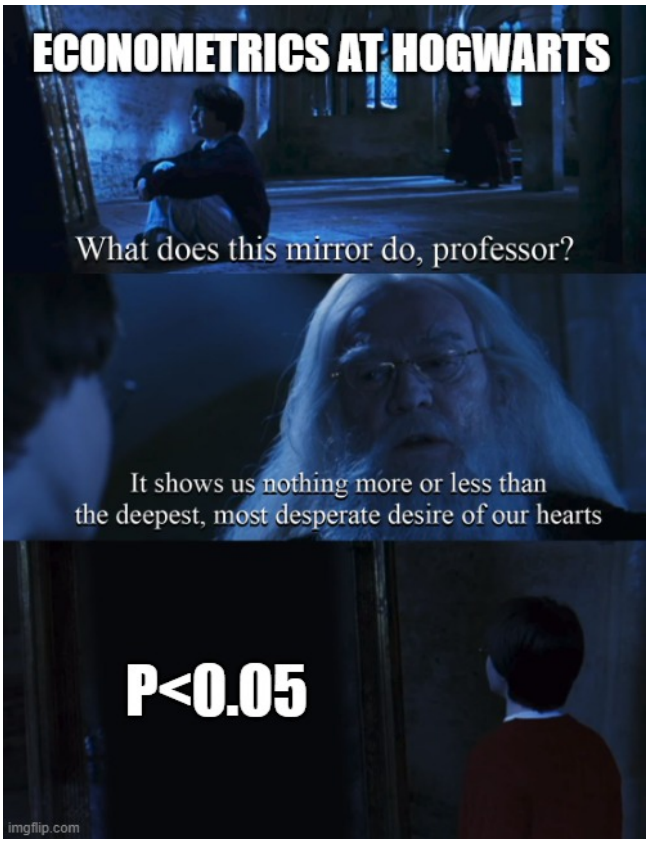
It's nearly 75%, but not quite. But is that far enough below 75% to dismiss the 75% claim? In other words, is it possible that the true mean actually is 75%, but by random chance our sample is not truly representative of the universe of 401k plans? Maybe our sample has disproportionately fewer 401ks with high match rates and more 401ks with low match rates than is true of the overall population? Sorting out this question is the intuition behind the one-sample t-test.

As a reminder, all inferential statistics relies on *Null* and *Alternative* Hypotheses. While most textbooks start with the Null Hypothesis, I prefer to work backward from the Alternative Hypothesis (H_1); this is typically the statement that the researcher is trying to prove is true. But the paradigm of science says that things cannot be proven to be true, they can only be proven to be false. So how do you prove something is true if science doesn't allow it? Proof by negation. You prove everything **but** H_1 is false, and if you can do that, then by default H_1 must be true. This is where the Null Hypothesis (H_0) comes in. This is the negation of H_1, and to prove that H_1 is true, then H_0 must be proven false.

In our 401k case, our hypothesis is about our population mean (μ), so we have:

- H_0: $\mu = .75$
- H_1: $\mu \neq .75$

In economics, it is common to use the confidence level of 0.95 (or $p < 0.05$).



The t-test is executed with the `t.test()` function.

```
t.test(k401k$mrate,
       alternative = "two.sided",
       mu = .75,
       conf.level = 0.95)
```

```
    One Sample t-test

data:  k401k$mrate
t = -0.92887, df = 1533, p-value = 0.3531
alternative hypothesis: true mean is not equal to 0.75
95 percent confidence interval:
 0.6924718 0.7705530
sample estimates:
mean of x
0.7315124
```

Here, we see the results of the t-test. The results state that we do not have

enough evidence to reject H_0, the expert's claim. Why not? There are quite a few things in the output that indicate that we should not reject H_0:

- the p-value ($p = 0.3531$) is not less than 0.05
- the confidence interval (.69 to 0.77) includes our hypothesized value of 0.75
- the t value (-0.929) is not in the rejection region (it would have to be less than -1.96...you can use the `qt()` function to look up t-values)

Let's say that this same source tells you that the average participation rate, `prate`, in 401k programs is over 90%. Notice the directional claim here - they're not just saying it's different from 90%, they're saying it's over 90%. This indicates a one-sided test, where:

- H_0: $\mu \geq .9$
- H_1: $\mu < .9$

We can calculate a one-sided t-test for `prate` as follows:

```
t.test(k401k$prate,
       alternative = "less",
       mu = 90,
       conf.level = .95)
```

```
    One Sample t-test

data:  k401k$prate
t = -6.1786, df = 1533, p-value = 4.135e-10
alternative hypothesis: true mean is less than 90
95 percent confidence interval:
     -Inf 88.06537
sample estimates:
mean of x
 87.36291
```

This suggests that we reject the null hypothesis, that the mean in the data set of 87.4 is far enough below 90 to say that the true mean is probably below 90. We see this in the results because:

- the p-value ($p = 4.135e-10$) is definitely less than 0.05
- the confidence interval ($-\infty$ to 88.1) does not include our hypothesized value of 90
- the t value (-6.18) is in the rejection region (for this test it starts around -1.65)

Tip from the Helpdesk: Scientific Notation - Don't Miss the Forest for the Trees

When you see something like p=4.135e−10, that's scientific notation meaning 4.135×10^{-10} which equals 0.0000000004135. The key thing to remember: this is an *incredibly* small number … way smaller than our 0.05 threshold!
People new to R mess this up all the time! They see "4.135" and think "that's bigger than 0.05, so it's not significant." Nope! Always look at the whole number, including that "e-10" part. The scientific notation is just R's way of avoiding writing out a bunch of zeros!

It is often useful to store the results of a test in an object, for example:

```
test1 <- t.test(k401k$prate,
                alternative = "less",
                mu = 90,
                conf.level = .95)
```

When you store your test as an object, you don't get any output in your console, instead, R creates an object in your environment window based on the name you assigned your object. This code created an object for me called `test1`.

To see what is in this object, we can see what its `attributes()` are:

```
attributes(test1)
```

```
$names
 [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
 [6] "null.value"  "stderr"      "alternative" "method"      "data.name"

$class
[1] "htest"
```

Now, you can refer to these elements using the `$` notation, for example:

```
test1$statistic
```

```
        t
-6.178623
```

```
test1$estimate
```

```
mean of x
 87.36291
```

> May the Format Be With You: Let R Write Your Paper
>
> When writing reports, you'll often want to reference specific test statistics or p-values in your text. Rather than running the test, noting the number, and typing it into your document (and risking typos!), you can use inline code to let R do the work.
> For example, instead of writing:
> - The t-statistic was -6.18.
>
> you can write:
> - The t-statistic was ‘r round(test1$statistic, 2)‘.
>
> When your document renders, R executes everything inside the backticks and includes the results **inline**-that is, as part of the text! The `round()` function ensures your text shows a clean "-6.18" instead of the full "-6.178623" that R calculates. The code calculates and automatically inserts the correct value, and in the rendered version you will see this:
> - The t-statistic was -6.18.
>
> Which is identical to if you had just typed it out. So you might wonder, why bother if it looks the same? For starters, if your data changes, your text updates automatically - no more hunting through your paper to fix outdated numbers. Secondly, no typos! Finally, once you get used to the method, it's actually *faster* to just write the inline code than it would be to do the calculations and put them into your text!

6.2 Two-Sample t-test

The two-sample t-test is used to compare the means of two populations. Typically, you are looking at a situation where you have a numeric dependent variable that you think varies based on the value of some categorical independent variable that has two possible levels.

To see this in action, let's check out a dataset in the `openintro` library called `fact_opinion`. As usual, you should take a look at the documentation with `?fact_opinion`. The dataset is pretty straightforward and comes from Pew Research, where the researchers asked participants to identify whether a particular statement was stating a fact or stating an opinion. There were 10 total statements, 5 of each type, and we have data from 5,035 respondents. So, we have data on how many of each type of statement each respondent classified correctly, and we also have a variable that states what `age_group` they are in:

19-49 or 50+.

This is the sort of data that is amenable to a two sample t-test. We have a natural research question: which age group is better at identifying fact vs opinion? Our independent variable, `age_group`, is categorical with two possible values, and there are two possible dependent variables, `fact_correct` and `opinion_correct`, that are numeric. Ok, technically they are discrete, and you'd ideally prefer to do a t-test on continuous data, but the sample size is big enough that this is a minor quibble…but I digress. Back to the t-test!

Before jumping into the statistical test, let's look at some basic summary statistics to get a sense of the differences between age groups:

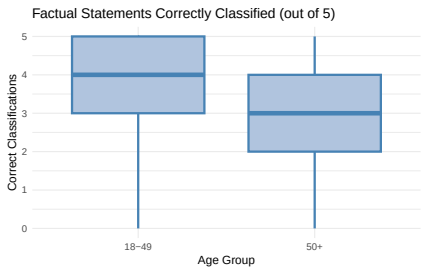
```
facttable <- fact_opinion |>
  group_by(age_group) |>
  summarize(fact_correct = mean(fact_correct),
            opinion_correct = mean(opinion_correct),
            count = n())

knitr::kable(facttable,
             col.names = c("Age Group", "Mean Fact Score", "Mean Opinion Score", "Count"))
```

Age Group	Mean Fact Score	Mean Opinion Score	Count
18-49	3.571977	3.930422	2084
50+	3.220264	3.456455	2951

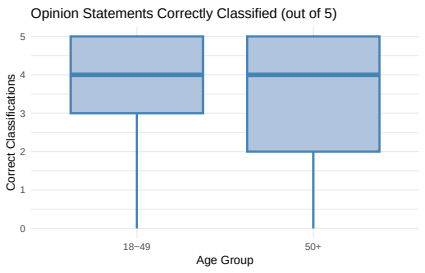
From the table, it seems that the younger group does better on both, and the difference is more pronounced when classifying opinion statements than factual statements. But just because the means are different doesn't mean the differences are statistically significant? Time for our t-tests!

> **Data Storytelling: When Good Graphs Go Bad**
>
> Ordinarily, one might begin this sort of analysis with a visualization. Let's see what happens if we try our usual approach of creating boxplots to compare groups:

```
# Graph 1 code
fact_opinion |>
  ggplot(aes(y = fact_correct, x = age_group)) +
  geom_boxplot(fill = colorlight, color = colordark, linewidth = 1) +
  theme_minimal() +
  labs(title = "Factual Statements Correctly Classified (out of 5)",
       x = "Age Group",
       y = "Correct Classifications")
# Graph 2 code
fact_opinion |>
  ggplot(aes(y = opinion_correct, x = age_group)) +
  geom_boxplot(fill = colorlight, color = colordark, linewidth = 1) +
  theme_minimal() +
  labs(title = "Opinion Statements Correctly Classified (out of 5)",
       x = "Age Group",
       y = "Correct Classifications")
```



(a) Factual Statements



(a) Opinion Statements

Something looks off about these boxplots, right? Notice how the factual statements graph has no upper whiskers for either age group, and the opinion statements graph shows that the measure of central tendency for both groups is *identical*.

Here's the deal: this is not a good graph and I know it. I actually put in a little extra effort to make these graphs look good to illustrate a point. Just because a graph **looks** good doesn't make it a good graph. One of the easiest ways to lie with statistics is quite simply to have good-looking graphs that tell an inaccurate story. The quality of the appearance creates a halo effect around the visualization and lends credence to the analysis, even when the graph is fundamentally flawed, or worse, intentionally misleading.

This is a perfect example of why choosing the right visualization matters! When your data only has a few discrete values (0, 1, 2, 3, 4, 5 in our case), boxplots can be misleading. The "quartiles" don't represent meaningful

> breakpoints when most people scored 4 or 5 out of 5. Those missing upper whiskers? That's because the third quartile equals the maximum value—everyone who didn't get a perfect score clustered at 4 out of 5. Sometimes a graph just isn't the right tool for the job and we need to use numbers. For discrete data like this, summary statistics often tell a clearer story than boxplots.

We should begin our t-tests by stating our Null and Alternative Hypotheses for the two t-tests we are about to run. Our hypothesis is about our population mean (mu), so we have:

- H_0: $\mu_{18-49} = \mu_{50+}$
- H_1: $\mu_{18-49} \neq \mu_{50+}$

In other words, the Null Hypothesis is that both age groups are equally good at discerning fact from opinion, whereas the Alternative Hypothesis is that the groups are in fact different in their capacity to differentiate fact from opinion. Based on the summary statistics from above, it looks like there might be a difference, but we should do the statistical test to be certain. We'll use the same `t.test()` command as before, start with the factual statements.

```
t.test(fact_opinion$fact_correct ~ fact_opinion$age_group,
        mu = 0,
        alt = "two.sided",
        conf.level = .95)
```

```
    Welch Two Sample t-test

data:  fact_opinion$fact_correct by fact_opinion$age_group
t = 8.7566, df = 4380, p-value < 2.2e-16
alternative hypothesis: true difference in means between group 18-49 and group 50+ is not equal t
95 percent confidence interval:
 0.2729682 0.4304571
sample estimates:
mean in group 18-49   mean in group 50+
           3.571977            3.220264
```

The first argument in this code is a little tricky, so let's take a second to understand it, in particular, the squiggly thing in the middle of `fact_opinion$fact_correct ~ fact_opinion$age_group`. This guy is called a **tilde**, and on computers with the standard US keyboard it shares the key with the backtick. In R, the tilde `~` is the formula operator; when I see I a tilde, in my head I say *is a function of* so I read `fact_opinion$fact_correct ~ fact_opinion$age_group` to say that the ability to identify something as a fact *is a function of* age. The general form, then, is: Dependent Variable $\sim$ Independent Variable. This is an important concept to get, because we will be

using the tilde a lot!

The other thing in this code that is, at first glance, a bit confusing is the option `mu = 0`. This is here because, technically, we are testing the difference between the group means and whether or not that is zero. In other words, actual null hypothesis that R is testing is H_0: $\mu_{18-49} - \mu_{50+} = 0$.

But that having been said, most of the arguments in this code are the default parameters of the `t.test()` function anyhow. When you look at the help for `t.test` using `?t.test`, it tells you what the default values are. The options `mu = 0, alt = "two.sided", conf.level = .95` are all defaults, so this code could have been simplified to just `t.test(fact_opinion$fact_correct ~ fact_opinion$age_group)` and we would have been fine.

So what do these results tell us? That we reject the null hypothesis!

- the p-value (p = 2.2e-16) is definitely less than 0.05. In fact, 2.2e-16 is quite literally the smallest number that R can display without going to zero!
- the confidence interval (.27 to .43) does not include our hypothesized difference in means of 0
- the t test statistic (8.76) is in the rejection region; for this test our critical t value will be roughly $\pm$ 1.96

Let's go ahead and take a look at the other t-test, the one that looks at opinion statements.

```
t.test(fact_opinion$opinion_correct ~ fact_opinion$age_group,
        mu = 0,
        alt = "two.sided",
        conf.level = .95)
```

```
    Welch Two Sample t-test

data:  fact_opinion$opinion_correct by fact_opinion$age_group
t = 12.638, df = 4535.7, p-value < 2.2e-16
alternative hypothesis: true difference in means between group 18-49 and group 50+ is n
95 percent confidence interval:
 0.4004395 0.5474941
sample estimates:
mean in group 18-49   mean in group 50+
           3.930422            3.456455
```

Again, we reject the null hypothesis!

- the p-value (p = 2.2e-16) is miniscule
- the confidence interval (.40 to .55) does not include 0
- the t test statistic (12.64) is in the rejection region; because the sample size is the same as in the last test, our critical t value is still right around

$\pm$ 1.96

Tip

Data Storytelling: Don't Skip Tests (Even If You Think the Answer is Obvious)
Did we actually have to run that second t-test? The difference in means was bigger for opinion statements (0.47) than for factual statements (0.35), and it's the same dataset, so surely if the first one was statistically significant, this one would be too, right?
Not necessarily! Statistical significance depends on more than just the size of the difference...it also depends on how much variability there is within each group. If people's scores on opinion statements were all over the map (high standard deviation), even a big difference in means might not be statistically significant. But if everyone scored pretty similarly within their age group (low standard deviation), even a small difference might be highly significant.
The lesson? When it's easy to run the test in R, just run it. Don't try to predict statistical significance by eyeballing means alone.
BTW: In this case, the standard deviations for opinion statements were actually lower than for factual statements, which helps explain why we got such a strong result. But I'm still glad I ran the test rather than assuming!

6.3 ANOVA

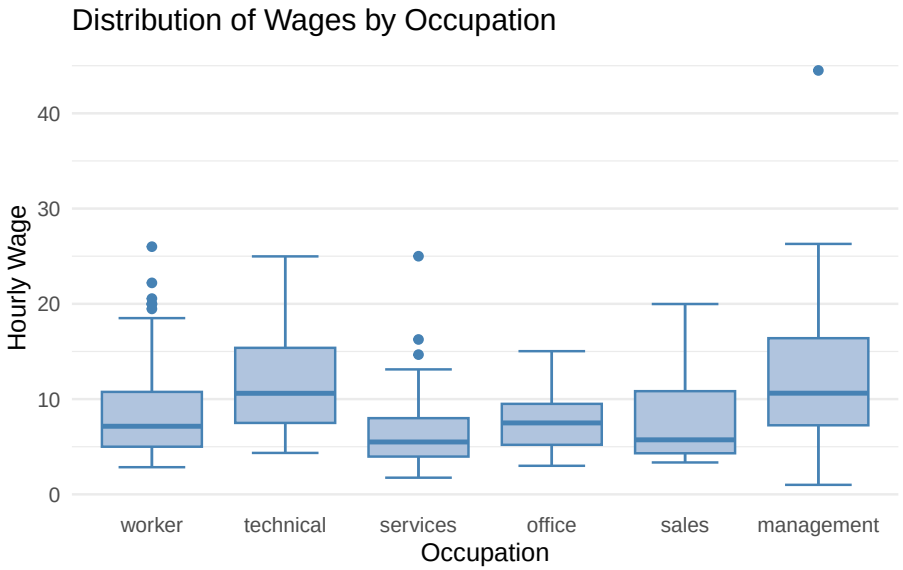
ANOVA stands for **Analysis of Variance** and is a very common statistical technique in experimental sciences where one can run an experiment with a control group and multiple treatment groups. Though there are some notable exceptions, economics is not usually viewed as an experimental science. However, ANOVA is a special case of regression analysis, which is at the core of econometrics.

In many ways, ANOVA is just an amped-up 2-sample t-test; amped-up because you can have your dependent variable explained by more than one independent variable, and those independent variables can have more than two **levels**. The concept of **levels** refers to how many different values a categorical variable has; our `age_group` variable from the `fact_opinion` dataset has two levels. But this means that, while the `fact_opinion` dataset was perfect for t-tests, ANOVA requires categorical variables with more than two levels. So let's return to the `CPS1985` dataset we used in the Descriptive Statistics chapter (Chapter **??**), because the `occupation` variable in `CPS1985` has 6 levels.

6.3.1 One-Way ANOVA

Let's calculate an ANOVA looking at the relationship between `wage` and `occupation` in he `CPS1985` data. We can start by looking at the data visually in a boxplot:

```
CPS1985 |> ggplot(aes(x = occupation, y = wage)) +
  stat_boxplot(geom = "errorbar",
               color = colordark,
               width = 0.5) +
  geom_boxplot(color = colordark,
               fill = colorlight) +
  theme_minimal() +
  theme(panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank()) +
  labs(y = "Hourly Wage",
       x = "Occupation",
       title = "Distribution of Wages by Occupation")
```



The null hypothesis in ANOVA is that ALL of the means are the same...if your categorical variable has k different levels, then your null hypothesis is $\mu_1 = \mu_2 = ... = \mu_k$. The alternative is that at least one is different from at least one other. In this case, the null is then $\mu_1 \neq \mu_2$ OR $\mu_1 \neq \mu_3$ OR $\mu_1 \neq \mu_4$ OR ... OR $\mu_5 \neq \mu_6$. If you're keeping score, that expands out to 15 ($\frac{k(k-1)}{2}$) different ways we might reject the null! From the graph, It certainly looks like at least some of the means are different, but we can execute an ANOVA via the `aov()` command to test this.

```
aov(wage ~ occupation, data = CPS1985)
```

```
Call:
   aov(formula = wage ~ occupation, data = CPS1985)

Terms:
                occupation Residuals
Sum of Squares    2537.697 11539.001
Deg. of Freedom          5       528

Residual standard error: 4.674844
Estimated effects may be unbalanced
```

The outupt of the `aov()` command doesn't actually tell us much. We need to save this test result as an object and use the `summary()` command on that object.

```
anova1 <- aov(wage ~ occupation, data = CPS1985)
summary(anova1)
```

```
             Df Sum Sq Mean Sq F value Pr(>F)
occupation    5   2538   507.5   23.22 <2e-16 ***
Residuals   528  11539    21.9
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This table tells us whether or not we reject the null hypothesis…the Pr(>F) is our p-value, and that is way less than 0.05. so we reject the null hypothesis that all the group means are the same and accept the alternative hypothesis that at least one is different from the rest.

All this tells us is that at least one of the occupational means is different from at least one other one. But remember, since there are 6 levels of the occupation variable, there are $\frac{6*5}{2} = 15$ different possibilities here! If we want to know **which** means are different, we can run the Tukey-Kremer test. To do this, we simply put our anova object into the `TukeyHSD()` function.

```
TukeyHSD(anova1)
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = wage ~ occupation, data = CPS1985)

$occupation
                          diff       lwr         upr     p adj
technical-worker     3.5209542  1.833122  5.20878674 0.0000001
services-worker     -1.8890045 -3.705623 -0.07238631 0.0361288
```

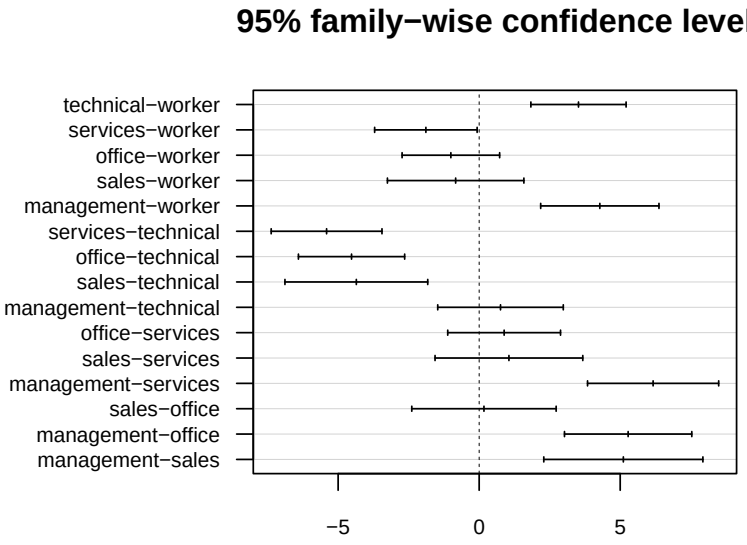
```
office-worker        -1.0038970 -2.732830  0.72503588 0.5584749
sales-worker         -0.8338428 -3.252714  1.58502882 0.9223496
management-worker     4.2775256  2.180694  6.37435762 0.0000001
services-technical   -5.4099587 -7.373822 -3.44609529 0.0000000
office-technical     -4.5248513 -6.407898 -2.64180401 0.0000000
sales-technical      -4.3547970 -6.886120 -1.82347368 0.0000170
management-technical  0.7565714 -1.469044  2.98218646 0.9265714
office-services       0.8851074 -1.114190  2.88440478 0.8032931
sales-services        1.0551617 -1.563792  3.67411581 0.8589942
management-services   6.1665301  3.841732  8.49132800 0.0000000
sales-office          0.1700543 -2.388857  2.72896576 0.9999657
management-office     5.2814227  3.024479  7.53836588 0.0000000
management-sales      5.1113684  2.290815  7.93192217 0.0000046
```

This shows the confidence intervals and p-values for all of the pairwise comparisons. We can display this visually by plotting the Tukey results:

```
plot(TukeyHSD(anova1))
```

Ugh this is unreadable. I really don't love using Base R graphics. We need to change the orientation of the labels on the y axis (switch them to horizontal) so we can actually read them. Here it is again with a couple formatting tweaks:

```
par(mar = c(3,10,3,3))
plot(TukeyHSD(anova1), las = 1, cex.axis = .75)
```



Better, but still not pretty. This is exactly why `ggplot2` exists - but unfortunately, there's no easy `ggplot2` equivalent for Tukey plots. That said, the R community is amazing at solving problems like this - there are custom functions floating around (e.g. (Moura Silva-Júnior 2022) looks like a possible solution to this quandary) that can convert Tukey results into prettier `ggplot2` charts. But for our purposes, the numerical output tells us what we need to know anyway, so we'll just work with what we've got here!

When evaluating the Tukey-Kremer results, any combination where 0 is not included in the confidence interval is considered to be a significant difference. It looks like there are quite a few of them! To summarize, we are rejecting the null hypothesis because it seems like there are two tiers of wage categories that are statistically different from each other; technical and management seem to have similar wage structures, which is higher than that of office, sales, worker, and services, who also have similar wage structures (though technically the difference between worker and services is also statistically significant).

6.3.2 Two-Way ANOVA

The previous example is what is referred to as a one-way ANOVA; one-way refers to the fact that there is exactly one independent variable. In this case, that variable is occupation. ANOVA can include more than 1 independent variable in a version called two-way ANOVA.

Two-way ANOVA is more complex and allows us to have multiple independent categorical variables. Here, let's look at wages as our dependent variable but have 2 categorical independent variables: gender and marital status.

This particular ANOVA has 3 null hypotheses:

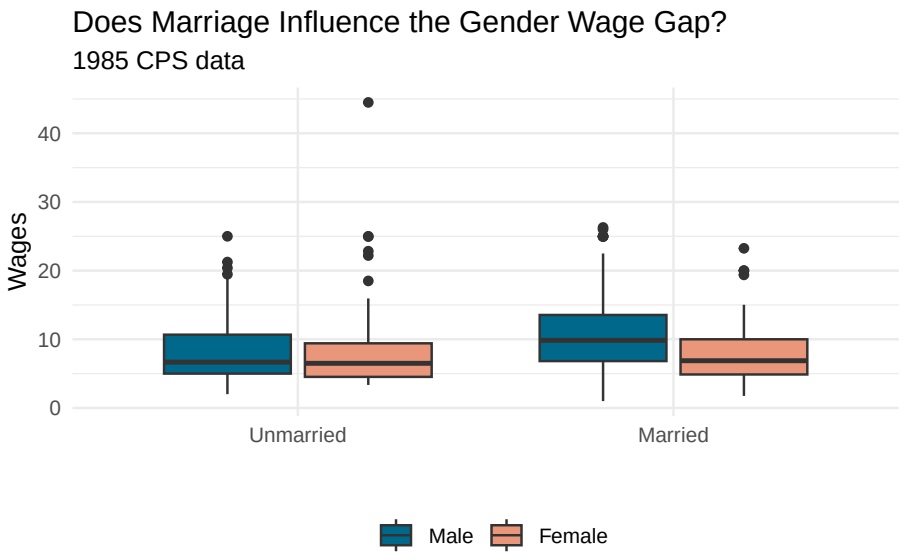
- There is no difference in wages between men and women
- There is no difference in wages between married and unmarried individuals
- There is no interaction between gender and marital status.

Interaction is a complex idea. It is related to the idea in probability regarding independence. Here, an interaction effect would imply that either:

- The effect of gender on wages varies depends on whether or not the person is married or unmarried, or equivalently,
- The effect of marital status on wages depends on whether or not the person is a man or a woman.

Let's take a look at this data graphically using a graph from the previous chapter:

```
CPS1985 |> ggplot(aes(x = married, y = wage, fill = gender)) +
    geom_boxplot() +
    theme_minimal() +
    labs(title = "Does Marriage Influence the Gender Wage Gap?",
         subtitle = "1985 CPS data",
         x = "",
         y = "Wages",
         fill = "") +
    scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                      labels = c("male" = "Male", "female" = "Female")) +
    theme(legend.position = "bottom",
          legend.direction = "horizontal") +
    scale_x_discrete(labels = c("no" = "Unmarried", "yes" = "Married"))
```



From the graph, it sure looks like the married men make more money than married women, but there is no difference in wages between unmarried men and unmarried women.

Let's run this ANOVA and see what the results are:

```
anova2<-aov(wage ~ gender*married, data = CPS1985)
summary(anova2)
```

```
                Df Sum Sq Mean Sq F value   Pr(>F)
gender           1    594   593.7  24.118 1.21e-06 ***
married          1    149   149.0   6.054  0.01420 *
gender:married   1    287   286.8  11.652  0.00069 ***
Residuals      530  13047    24.6
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can reject all 3 null hypotheses at the 5% level. We can dig deeper into the results by looking at the Tukey-Kremer test:

```
TukeyHSD(anova2)
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = wage ~ gender * married, data = CPS1985)

$gender
                 diff       lwr       upr   p adj
```

```
female-male -2.116056 -2.962502 -1.269611 1.2e-06

$married
           diff       lwr      upr     p adj
yes-no 1.111544 0.2240042 1.999084 0.0142019

$`gender:married`
                             diff        lwr        upr     p adj
female:no-male:no    -0.09511392 -1.9895353  1.7993075 0.9992257
male:yes-male:no      2.52131135  0.9437846  4.0988381 0.0002573
female:yes-male:no   -0.67098704 -2.2921516  0.9501775 0.7099932
male:yes-female:no    2.61642528  0.9312927  4.3015579 0.0004171
female:yes-female:no -0.57587312 -2.3019252  1.1501790 0.8255032
female:yes-male:yes  -3.19229840 -4.5630697 -1.8215271 0.0000000
```

The bottom panel examines the interaction effects, and is pretty interesting.

- The difference in wages between unmarried men and unmarried women is insignificant.
- The difference in wages between married females and unmarried females is insignificant.

But:

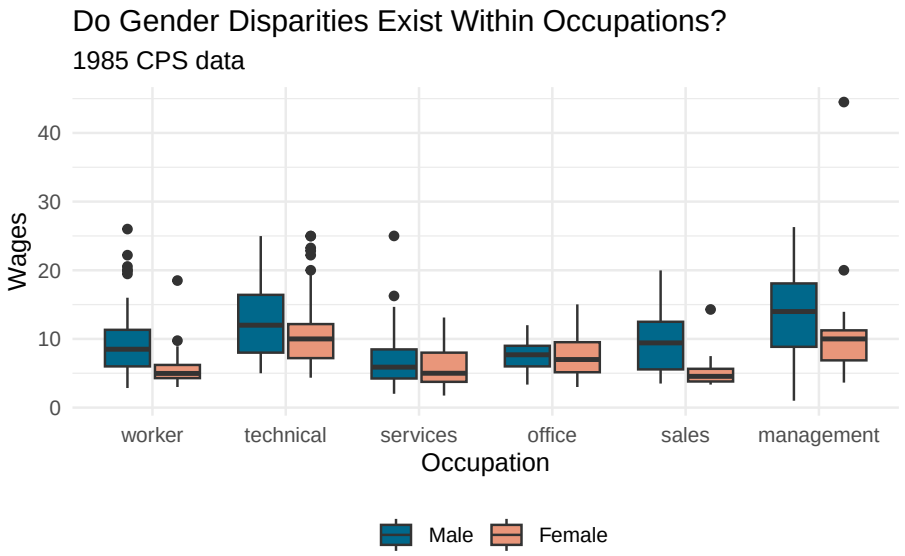
- The difference between married women and married men is significant
- The difference between married men and unmarried men is significant

These results reveal that the gender wage gap (at least in 1985) may not be just a simple story of "men earn more than women therefore gender discrimination." Instead, the data suggests the gap is concentrated among married workers, with unmarried men and women earning similar wages. This pattern points to marriage-specific economic factors rather than uniform discrimination - perhaps reflecting differences in job mobility, career prioritization, or employer perceptions of married workers' productivity and commitment.

Let's look at one more two-way ANOVA, this time looking at gender and occupation. One common pushback you hear against the gender discrimination thesis is that gender pay inequality is actually just a function of occupational choice; the job market doesn't pay women less than men, rather, men tend to choose higher paying occupations.

```
CPS1985 |> ggplot(aes(x = occupation, y = wage, fill = gender)) +
    geom_boxplot() +
    theme_minimal() +
    labs(title = "Do Gender Disparities Exist Within Occupations?",
         subtitle = "1985 CPS data",
         x = "Occupation",
         y = "Wages",
         fill = "") +
```

```
    scale_fill_manual(values= c("deepskyblue4", "darksalmon"),
                      labels = c("male" = "Male", "female" = "Female")) +
    theme(legend.position = "bottom",
          legend.direction = "horizontal",
          axis.ticks.x = element_blank())
```




```
anova3<-aov(wage ~ gender*occupation, data = CPS1985)
summary(anova3)
```

```
                  Df Sum Sq Mean Sq F value   Pr(>F)
gender             1    594   593.7  28.416 1.46e-07 ***
occupation         5   2403   480.6  23.002  < 2e-16 ***
gender:occupation  5    174    34.7   1.663    0.142
Residuals        522  10906    20.9
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Should we do a Tukey-Kremer test? we could, but we would have a bazillion lines of Tukey outupt (ok, technically it's just 82) that we'd get lost wading through. Instead, we need to work through this numerically. And the results are actually pretty interesting!

This says that the `gender` and `occupation` differences are statistically significant, but the interaction effect is not. The insignificant interaction ($p = 0.142$) tells us that the gender pay gap is roughly the same across all occupations. In other words, the effect of being female on wages doesn't depend on which occupation you're in - women earn less than men by roughly the same amount whether they're in management, technical roles, office work, etc. This is actually

quite telling! If the interaction were significant, it might suggest that the gender pay gap varies by occupation - maybe women in technical roles face bigger pay gaps than women in office roles, or vice versa. But since it's not significant, we're seeing consistent gender differences across all job types.

This finding challenges the simple explanations for gender pay gaps you often hear about. The significant gender effect alongside occupation effects suggests it's not just occupational sorting - women choosing lower-paying jobs. At the same time, the consistent gap across occupations (non-significant interaction) pushes back against the glass ceiling narrative, where women supposedly hit barriers only in high-status fields. Of course, this is still data from 1985, so whether these patterns hold today is an open question.

Anyhow, this is enough ANOVA for me. Econometrics typically focuses on regression analysis, not ANOVA. And, for what its worth, ANOVA is just a very special case of regression analysis which we will see in Chapter **??** Categorical Independent Variables.

6.4 Chi-Square

The Chi-Square test (sometimes stylized as χ^2) looks at relationships between 2 (or more) categorical variables. Whereas ANOVA had categorical independent variables with numeric dependent variables, Chi-Square has both categorial independent *and* dependent variables. Let's look at the `Fair` dataset from the `Ecdat` package. This data is from a 1977 paper that looks into the prevalence of extramarital affairs. We might ask the question whether or not there is a relationship between extramarital affairs and whether or not a married couple has kids.

Let's start by making a cross tabulation of couples with kids and couples where an affair is occurring. First, we need to do a bit of data wrangling. The original dataset has a variable called `nbaffairs` that counts the number of affairs, but for our chi-square test we need a simple yes/no variable. So we'll create a new variable called `affair` that equals "yes" if someone had any affairs and "no" if they had zero affairs. Then we'll select just the two variables we need for our analysis.

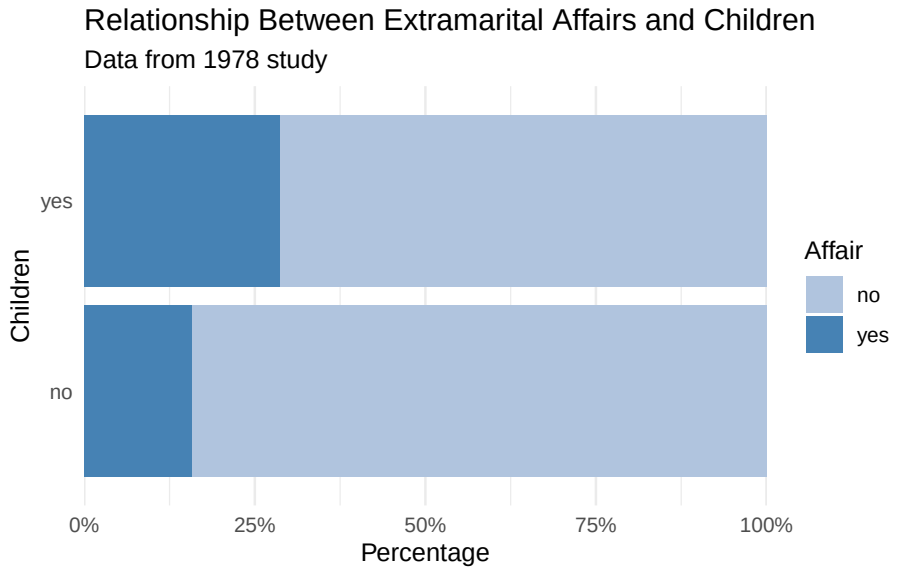
```
affair <- Fair |>
  mutate(affair = ifelse(nbaffairs==0, "no", "yes")) |>
  select(c(affair, child))
table(affair)
```

```
      child
affair  no yes
   no  144 307
   yes  27 123
```

On a percentage basis, it looks like affairs are more prevalent among couples

with kids. Maybe this is more clear with a graph:

```
affair |>
  ggplot(aes(y = child, fill = affair)) +
  geom_bar(position = "fill") +
  scale_fill_manual(values = c("no" = colorlight, "yes" = colordark)) +
  scale_x_continuous(labels = scales::percent_format(),
                     expand = c(0, 0)) +
  theme_minimal() +
  theme(panel.grid.major.y = element_blank(),
        panel.grid.minor.y = element_blank()) +
  labs(title = "Relationship Between Extramarital Affairs and Children",
       subtitle = "Data from 1978 study",
       x = "Percentage",
       fill = "Affair",
       y = "Children")
```



Is this difference big enough to be statistically significant? This is where we call in the Chi-Square test:

```
chisq.test(affair$child, affair$affair)
```

```
    Pearson's Chi-squared test with Yates' continuity correction

data:  affair$child and affair$affair
X-squared = 10.055, df = 1, p-value = 0.00152
```

The p-value is less than .05, so it appears that there is a statistically significant

difference in the rate of affairs between couples with kids from those without kids.

6.5 Wrapping Up

The metnods seen in this chapter are not frequently used in either economics or data science. Because these methods require that all of the independent variables used be categorical, their primary value lies in the world of experimental research.

It is very frequently the case that we want to examine independent variables that are numeric. This poses a problem for models like ANOVA and Chi-Square, because in order to use a numeric independent variable in one of these models, we need to create a set of *ad hoc* categories to fit our numeric data into, and in the process losing much of the variation (and thus information) in the data. None of this is ideal.

There is a modeling technique that allows us to use both categorical and numeric independent variables: regression analysis. Regression forms the basic building block of econometrics and of data science, and is the focus of the irest of this text.

6.6 End of Chapter Exercises

For each exercise, state your hypotheses, create appropriate visualizations, run the statistical tests, and interpret your results in context.

T-Tests:

1. Using the `openintro:fact_opinion` dataset, create a new variable that represents each person's total score (combining their fact_correct and opinion_correct scores). Hint: use `mutate()` to add the two variables together. Then test whether younger adults (18-49) score significantly different from older adults (50+) on this total score. State your hypotheses, run the appropriate test, and interpret the results. Based on what we found in the text, are you surprised?

2. Using the `AER:CPS1985` dataset, test whether there's a significant difference in wages between men and women. Create a boxplot to visualize the data first, then run the appropriate t-test. What can you conclude about the gender wage gap in 1985?

ANOVA:

3. Use the famous `datasets:iris` data to see if any of the 4 measures of iris size (`Sepal.Length`,`Sepal.Width`,`Petal.Length`,`Petal.Width`) varies by iris `Species`. Create boxplots and analyze each of the flower characteristics with one-way ANOVA, and if any of your ANOVAs indicate that you

should reject the null hypothesis, follow up with Tukey-Kramer tests to identify which pairs are statistically different.

4. Install the `palmerpenguins` package using `install.packages("palmerpenguins")`, then load it with `library(palmerpenguins)` and access the data with `data(penguins)`. Examine whether penguin physical characteristics (`bill_length_mm`, `bill_depth_mm`, `flipper_length_mm`, `body_mass_g`) differ across the three penguin `species`. Start by creating boxplots to visualize the data, then run one-way ANOVAs for each measurement. For any significant results, conduct Tukey-Kramer post-hoc tests to determine which species pairs differ significantly.

5. Use the `datasets:mtcars` data to see if size of an engine as measured in number of cylinders (*cyl*) influences fuel efficiency (*mpg*), power (*hp*), or speed (*qsec*). Important: Make sure you force R to treat the cylinder variable as a factor, not a number! Create boxplots and analyze each of the performance characteristics with one-way ANOVA, and if any of your ANOVAs indicate that you should reject the null hypothesis, follow up with Tukey-Kramer tests to identify which pairs are statistically different.

6. Using `CPS1985`, run a two-way ANOVA examining wages by both gender and sector. Test for main effects and interactions. How do your results compare to the gender/occupation analysis from the chapter?

Chi-Square:

7. Using the `AER:CPS1985` dataset, test whether there's a relationship between gender and occupation. Create a cross-tabulation table and run a chi-square test. What does this tell you about occupational segregation in 1985?

8. Create a two-way table examining the relationship between marital status and sector (manufacturing vs service) in the `CPS1985` data. Test for independence and interpret your results.

Chapter 7

Ordinary Least Squares

Regression modeling is at the center of econometric methods. Starting here, the majority of this text will focus on either regression models or extensions of the regression model. We begin with the simplest of regressions; this chapter will focus heavily on bivariate ordinary least squares (OLS) modeling and introduce multivariate OLS.

he t-tests and ANOVA we just covered work great when you're dealing with clean experimental data or when you can neatly categorize your variables. But here's the thing about regression: it's what you use when the world is messy and you can't control it. Unlike experimental scientists who can isolate variables in a lab, economists and analytics folks usually have to work with whatever data the world throws at them. People make decisions, markets fluctuate, policies change, and somehow you're supposed to figure out what's driving what from the chaos that results. Regression is the tool that lets you disentangle these relationships when you can't run controlled experiments.

The good news is that once you wrap your mind around the basic intuition of regression, you'll have unlocked not just what economists do, but also the foundation for understanding every fancy variation that comes later in this book. Sure, the math gets mathier when we move to logit models or time seris analysis, but the core logic remains the same. Nail down OLS and everything else is just a remix. The bad news is that there's a lot of new terminology, some math notation that might look intimidating, and concepts that probably won't click immediately. That's normal.

We'll start with simple bivariate regression where you have one variable trying to explain another, then work our way up to multivariate models where multiple factors are all fighting for credit. By the end of this chapter, you should be able to run and interpret basic regression models, which honestly puts you ahead of a shocking number of people with advanced degrees who make confident claims about complex issues without understanding how to control for confounding

variables.

As usual, let's start by loading the data we'll be using:

```
library(tidyverse)
library(stargazer)
library(openintro)
library(wooldridge)
library(tidyquant)
library(quantmod)
library(strucchange)

data(cars04)
data(airfare)
data(gpa2)



library(Ecdat)

library(AER)



data(fact_opinion)
data(k401k)
data(CPS1985)
data(Fair)

data(vote1)
data(sleep)


# Setup Colors!
colorlight <- "lightsteelblue"
colordark <- "steelblue"

colorlight2 <- "indianred1"
colordark2 <- "indianred4"
```

7.1 Prelude to Regression: The Correlation Test

We saw how to calculate a correlation (Pearson's r) in Chapter **??**. Before we dive into the full power of regression, let's warm up with a simpler question: how do we know if two variables are actually related to each other, or if any pattern we see is just random noise?