Urban Informatics Toolkit

Josiah Parry

2020-04-20

Contents

Chapter 1

Welcome!

Welcome to the Urban Informatics Toolkit! This is an online book that is intended to jumpstart your work of analyzing and developing understanding of the urban commons. In it you will find material on the theoretical underpinnings of Urban Informatics and a (mostly) thorough introduction to the statistical programming language R to get you hands on and working with data that you will encounter in your research, coursework, and in the wild west of open data.

This book represent to me many things. Of which are the two years of study in the Urban Informatics program at Northeastern, nearly five years of self-directed learning of the R programming language, two years of teach R, and many, many, many hours of frustration trying to understand and grasp concepts that could have been distilled into simple and easy to understand language.

In the following chapters I will do my best to avoid overly technical and verbose descriptions of theory and technical concepts. I will attempt to explain everything in a manner that I would to my friends over a beer, coffee, or, as these strange times would have it, a zoom call.

By the end of these pages I hope that you have become self-sufficient in your analyses. My intention is not to teach you everything that you will need to know because this is *impossible*. Data analytics are always changing and what is current will be dated—perhaps by tomorrow. In fact, whilst in the middle of writing this one of the most commonly used tools was changed in such a way that much of what I wrote became bad practice. This is all to say that the purpose of this book is to make you **self-sufficient**. The goal is to ensure that *you* are able to understand the fundamental concepts of scientific inquiry in the big data era, how to think about data analysis, and be equipped with the fundamental knowledge of R to understand what is happening—to some degree—under the hood and what and why.

1.1 Structure of the book

This book is partitioned into four sections each with it's own theme. In the first section *Foundations of Urban Informatics* we will first begin by exploring the nature of big data and its role in the field. Then, we review different approaches to scientific inquiry and seek to understand how big data has further enabled a newer approach. And finally, we review Broken Windows theory as well as the development of **ecometrics**.

The second section is dediated to introducing you to R as a programming language for data analysis. This is where we begin our technical work. We will ensure that you have all of the software and data that you will need to follow along with the exercises in this book. This section will be the most difficult to overcome. This is because you must learn an entirely new mental framework—even if you know how to program in another language. In this section we will work from reading in data to manipulating it while along the way learning some fundamental theory.

The third section is the largest and is dedicated entirely to information visualization. In this you will learn how to craft visualizations and understand when to make what kind of graphic. Furthermore we will dive into expanding upon traditional graphics by incorporating many different aesthetics. This section is expansive due to the importance of visualization. Visualization is our method of communication. While the written word is powerful, it requires more work to get someone to read than to look. If we can improve upon our visualization, we can make the work that we do more accessible to the public.

Finally, we will close by learning how to work with multiple datasets and spatial data. These are two of the more advanced topics and as such will be reserved for when the fundamentals have been reinforced. With regards to spatial analysis, many of you who may come from a geography or GIS background may find this section lacking. You would be right. Given the immense depth of the geospatial sciences, that is a topic that is deserving of its own book.

1.2 Considerations

Before we continue, I want to reitaterate that this book will not introduce you to everything that you will need to know. As the field continues to grow and as the number of tools available in R increase I will work to continually add to and improve this writing. If there are topics that you would like to see included or expanded upon, please submit an issue on GitHub https://github.com/JosiahParry/urban-informatics-toolkit/issues or reach out to me directly.

In this next chapter we will discuss big data.

Without further ado, let's get on with it!

Part I

Foundations of Urban Informatics

Chapter 2

The Utility and Danger of Big Data

Urban Informatics is in a way a byproduct of the "deluge" or "proliferation" of data. In essence, as we as a society have progressed technologically, we have been able to capture and store data on a rather unprecedented scale. This has led to massive stores of data that are used primarily for record keeping that are updated at near-real-time. For example, think of every time you make a Facebook post or send a tweet. That post or tweet is subsequently recorded in a remote database to ensure that it can be accessed at a later time. Dan O'Brien notes that this characteristic of big data is of the utmost consequence for "the advancement of science and policy in the digital age"[1]. These naturally occurring data are so useful because they are essentially a track record of individuals' behavior over time. It is in a way as close as we can get to measuring behavior at real time.

If we turn to the urban context, the importance of big data may become ever more apparent. For example, Boston local government has been keeping detailed records of property assessments, taxes owed and paid, by whom, their demographic characteristics, when and even where down to the ward level (Boston Public Library, recently digitized records). These administrative records have been kept on ink and paper until just a few decades ago. Through digitization efforts, these data are now accessible to historians, urban scholars, local government, and the general public. Having such data accessible provides a way to quantitatively inspect the development of the city from its geography, its policies, its demography, and much more that we have yet to see uncovered.

There are a number of benefits that naturally occurring data provide. The first is that these are, in theory, comprehensive and contains information about

[1] Page 59 of The Urban Commons: How data and technology can rebuild our communities. by Daniel T. O'Brien.

all residents. Through administrative data we should, for example, be able to determine the number of employed tax paying citizens as well as the under-employed who receive government benefits. Additionally, since these data are already being collected, the associated costs are minimal. In contrast to empirically collected data, administrative data are not just a representation of a single moment in time, but rather continually changing and updating. And due to the fact that administrative data are collected at the municipal level we are inherently dealing with geospatial data—data that have a location associated it.

While there are many benefits to administrative big data, there are a couple of dangers we ought to be aware of. The first is that even though big data are comprehensive in theory, we cannot always take them as objective observations of the natural world. We must be cognizant of the fact that the biases that humans have are also represented in data. We cannot and should not separate theory from data. To take from Dan O'Brien

> ". . . the very point of science is to explain why things work the wsay they do. . .If we limit our inquiries only to correlation and eschew explanation, we are no longer conducting science."[2] — Daniel T. O'Brien

Furthermore, we should always be wary of the data that we use. Investigate its integrity, its source, its measurment constructs for what we may be using to understand one thing may be something else which we may not anticipate or expect. And laslty, always be aware of the ways in which you may be bringing in your own world views into your work as what we are after is not confirmation but understanding.

[2]Page 61 of The Urban Commons.

Chapter 3

Data in the municipal context

The so called "proliferation of data" has created vast troves of data asking to be explored. We are, in essence, in the beginning of a new Gold Rush. But rather than discovering gold, today the gold is both being created an discovered. This explosion of data is the product of improved technology in both the collection and storage of data.

If we focus our gaze towards the municipal government, the story is similar, progress is slower, and the data are more familiar. Local governments have been collecting data for centuries but until recently it was not always accessible, or even considered "data". Take the city of Boston as an example. Since the 19th century boston has been issuing and recording building permits. Through a massive digitization effort these permits are now accessible in an online database [1]. Not only are governments slowly turning to modern methods of data storage, but they are also creating applications to encourage citizens to engage with their local governments. Mobile and web applications will hopefully facilitate greater interaction between citizen and government [2]. Each and every one of these citizen to government interactions are recorded and stored in database—though not all are open and accessible to the citizen scientist.

Boston has built a few mobile applications for its residents. Notable among these apps are the BOS:311[3], ParkBoston[4], the city's least favorite Boston PayTix[5], and the new Blue Bikes[6]. Through BOS:311 residents can communicate directly

[1] Boston Building Permits: https://www.boston.gov/departments/inspectional-services/how-find-historical-permit-records

[2] Some note about co-production.

[3] BOS:311: https://itunes.apple.com/us/app/boston-citizens-connect/id330894558?mt=8

[4] ParkBoston: https://apps.apple.com/us/app/parkboston/id953579075

[5] Boston PayTix: https://apps.apple.com/us/app/boston-paytix/id1068651854

[6] BlueBikes

to the Department of Public Works by recording an issue, it's location, and even an image of the issue. Blue Bikes trips, 311 requests, and much more are provided to the public via Analyze Boston, Boston's data portal[7].

This new availability of data has unintentionally altered the way in which scientists interact with data. For the purposes of scientific inquiry, scientists and analysts have historically been rather close to the data generation process. While we as residents and citizens interact with governmental agencies, it is not in the name of science. And the governmental agencies are engaing with residents in for the purpose of governance, not science. As such, much—if not all—of the open and public data that we interact within the urban informatics—and greater digital humanities—fields was not generated with the express purpose of being analyzed. This inherently changes the way in which analyses are approached.

In approaching data of this nature, researchers have began embracing a paradigm of *exploratory data analysis* (EDA). EDA is extremely useful for developing insights from data in which there were no a priori[8] hypotheses. In their influential book *R for Data Science*[9], Garret Grolemund and Hadley Wickham describe this inductive approach of exploratory data analysis.

> "Data exploration is the art of looking at your data, rapidly generating hypotheses, quickly testing them, then repeating again and again and again." [10]

When researchers set out to test a hypothesis they often will become closely involved with the data generation process. In this scenario, researchers are more likely to have preconceived hypotheses and expectiation of what they may find hidden in their data.

This condition is often not the case when working with open data. We do not always know at the outset of what we are looking for. With open data—and any data really—you never know what you may find if you begin to dig. Whip out your hand shovel and prepare to upturn the soil. You might find seedlings that may sprout into your next study.

[7] https://data.boston.gov/

[8] "Relating to or denoting reasoning or knowledge which proceeds from theoretical deduction rather than from observation or experience." Oxford Dictionary

[9] R for Data Science

[10] Introduction - R for Data Science

Chapter 4

Approaches to and Schools of Urban Informatics

4.1 Scientific approaches

In the sciences generally, there are two approaches to scientific inquiry: inductive and deductive. These two approaches can be best characterized as "bottom up" and "top down" respectively. Each has their own origins, strengths, and weaknesses. An argument has been raging—in the scientific sense of raging—since the 17th century about which approach is the best one. My answer to this? Both, and neither. And you'll see why shortly.

Let's start by getting a grasp on deductive approaches (also referred to as deduction). With deduction we start with a theory about the workings of some observed phenomenon. From this theory, we create a hypothesis, then observe (or collect data on) the phenomenon. With this data we then confirm or refute our theory. This is how we all have, most likely, been taught about the scientific method.

Induction works in a reverse order. It works by looking at the natural world and doing just that, looking. It works by noticing something—a pattern, a unique occurrence—then noticing it again, and then again, and then under slightly different conditions. From those observations, we draw hypothesis. We then observe yet again and see if we can refute or add a little bit more credibility to our findings. Then from doing this time and again, we can build a theory. It's somewhat like Sherlock and Watson finding clues and then coming to (frighteningly specific) conclusions. These theories, no matter how we get there, are the frameworks that we use to try and explain phenomena that we see.

4.2 The Chicago School

Much of what is known as the urban sciences today can be traced back to the late 19th and early 20th centuries at the University of Chicago. The University of Chicago was at the time the epicenter of the new field of American Sociology which came to be known as the Chicago School. In that era, the social sciences were seeking to create grand theories of the world. Take, for example, the new field of Anthropology that crafted theories about the origins of the human race. The Chicago School "fostered a very different view of sociology: first-hand data collection was to be emphasized; research on the particular case or setting was to be stressed; induction over deduction was to be promoted"[1].

Scholars such as Robert Park, Ernest Burgess, and Louis Wirth developed a number of micro-theories to understand the city. Most notably is the culminating work The City by Park and Burgess, a collection of essays that encapsulate decades of careful observation that led to a number of theories that still have influence today. Their body of work, important in so many ways, is an early paragon of an inductive approach to social research.

4.3 Complexity and Santa Fe Institute

On the other end of the spectrum rest the Santa Fe Institute (SFI) and their complexity science. The SFI's mission is to "endeavor to understand and unify the underlying, shared patterns in complex physical, biological, social, cultural, technological, and even possible astrobiological worlds."[2] Crudely, it is their goal to unify theory into one general master theory. Central to their theoretical focus is the the view that "valuable ideas are often confined to the echo chambers of academia."[3] In this view, they are not wrong.

Their work is important for bridging many so called gaps between disciplines. They apply biological theories of scaling to those of human development. And their findings have been fruitful! Their focus on the interconnectivity of theory of both the natural and social worlds is in some ways the messy work that must be done. This too is in the spirit of the Chicago School as illustrated by it's view of the City as an organism.

4.4 A Hybrid Approach

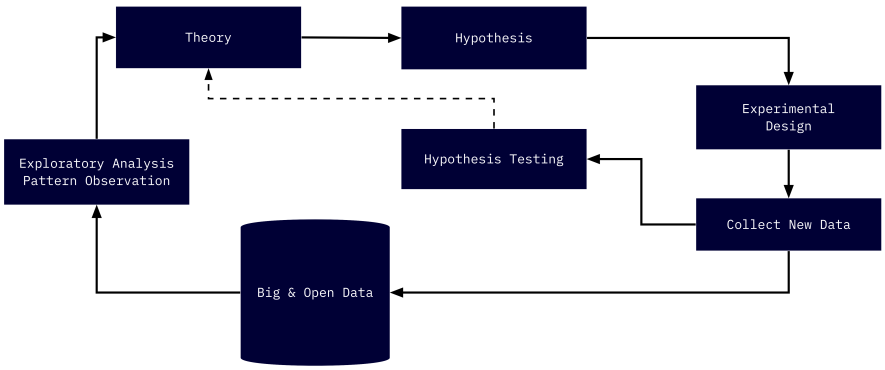
As part of this proliferation of big data we have more and better data within reach. As such we are able to, perhaps even encouraged to, take a much more

[1]Turner, Jonathan H. "The mixed legacy of the Chicago school of sociology." Sociological perspectives 31, no. 3 (1988): 325-338.

[2]Santa Fe Institute

[3]Santa Fe Institute

hybrid approach. Within the these data are a multitude of opportunities to explore and glean patterns that we may have been so subtle that it hasn't been observed before. Or, alternatively, the data had not been collected before. This allows us to take a much more inductive approach where we can craft theories from the patterns that we observe, then we can test those theories in creative ways. It is here where, I believe, that the Boston Area Research Initiative (BARI) rests on the spectrum. This hybrid methodology incorporates both inductive and deductive approaches.



The above graphic is m best attempt to illustrate this hybrid model. We start with data. We use publicly available (open) administrative data to explore. We get our metaphorical hands dirty with the data. After we munge it, transform it, and rearrange it, we will walk away with some tidbit of information. From that we discover more. And at the end of the day we develop a theory—a framework for understanding what we have observed. Next, we then develop a hypothesis using that new theory and apply it to some other set of data or some new circumstance to test and refine the theories we have developed. In this we both create theory from observation, and test those theories on new and unexplored observations.

Dan O'Brien, Director of BARI, claims to track more with the Chicago School in their approach—and this is correct. But, BARI also actively seeks to evaluate existing theory. There are no better examples of the BARI approach than the development and use of **ecometrics** to understand the City and test existing theory.

In the next chapter we will learn about ecometrics, their origins, and their use in evaluating the prominent crimoinological *Broken Windows* theory.

Chapter 5

Ecometrics

Central to the work that is done at BARI are the development and utilization of **ecometrics**. Ecometrics represent a quantitative representation of physical and social environment. In the Urban Informatics context, ecometrics are created to extract **latent constructs**—or variables that can only be inferred from a dataset—that illustrate some physical or social phenomenon.

To understand this, we need to again contextualize these datasets. They *are not* created with the intention of beng analyzed or to measure the blight of a neighborhood or the social unrest of a city. The data may tell a story of an underserved neighborhood or of a gilded community with a beautiful brick façade with next to no collective efficacy efforts. These datasets contain gems—beautifully inelegant snapshots of the societal quotidian. But measuring that? That's the tough part and that is why we create ecometrics. They provide us with a way to adapt existing data to address new problems.

Of the work that BARI conducts, the production of city-wide ecometrics of social and physical disorder are most emblematic of this hybrid approach. To understand this work we need to venture back to 1982 and an article from the Atlantic called *Broken Windows*[1].

5.1 Broken Windows Theory

During the beginning of the crack-cocaine epidemic George Kelling and James Wilson wrote a now [in]famous article titled *Broken Windows* which outlined a new theory to explain the occurrence of crimes. The premise of this article is that the *presence* of disorder is more concerning for a neighborhoods residents

[1] Wilson, James Q., and George L. Kelling. "Broken windows." Atlantic monthly 249, no. 3 (1982): 29-38. https://www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/.

than the actual crime that occurs. Further, the "visual cues of disorder … begets predatory crime and neighborhood decline"[2].

Broken Windows captured the eyes of pundits and policy makers. The simplicity of the theory makes it easy to Broken windows has historically captured the attention of policy makers. The vast public support has led to a large body of work largely disputing the merits of this theory. In the process of doing so, much work has gone into actually quantifying disorder in a city. In a seminal article by Sampson and Raudenbush (1999), the practice of systematic social observation was created[3]. This is a process in which imagery of public spaces is taken and coded to identify disorder—i.e. the presence of empty beer cans—which can then be quantitatively analyzed. This is an early example of an ecometric.

5.2 Quantifying Disorder

In 2015, O'Brien and Sampson published the article *Public and Private Spheres of Neighborhood Disorder: Assessing Pathways to Violence Using Large-scale Digital Records*[4]. This article epitomizes the hybrid approach to urban studies. In it, O'Brien and Sampson utilize 911 dispatches and 311 call data to create measures of both social and physical disorder. These measures were then used to put Broken Windows theory to the test. The process of using existing administrative datasets as a method of estimating social and physical phenomena illustrates the inductive approach. Whereas testing testing the efficacy of Broken Windows is indicative of the more traditional deductive process.

Quantifying disorder is no small task. In their 2015 paper the authors write

> Taking up this challenge, O'Brien, Sampson, and Winship (2015) have proposed a methodology for ecometrics in the age of digital data, identifying three main issues with such data and articulating steps for addressing each. These are (1) identifying relevant content, (2) assessing validity, and (3) establishing criteria for reliability. [5] [6]

[2] O'Brien, Daniel Tumminelli, and Robert J. Sampson. "Public and private spheres of neighborhood disorder: Assessing pathways to violence using large-scale digital records." Journal of research in crime and delinquency 52, no. 4 (2015): 486-510. https://journals.sagepub.com/doi/abs/10.1177/0022427815577835.

[3] Sampson, Robert J., and Stephen W. Raudenbush. "Systematic social observation of public spaces: A new look at disorder in urban neighborhoods." American journal of sociology 105, no. 3 (1999): 603-651. https://www.journals.uchicago.edu/doi/abs/10.1086/210356.

[4] O'Brien, Daniel Tumminelli, and Robert J. Sampson. "Public and private spheres of neighborhood disorder: Assessing pathways to violence using large-scale digital records." Journal of research in crime and delinquency 52, no. 4 (2015): 486-510. https://journals.sagepub.com/doi/abs/10.1177/0022427815577835.

[5] O'Brien, Daniel T., Chelsea Farrell, and Brandon C. Welsh. "Looking through broken windows: the impact of neighborhood disorder on aggression and fear of crime is an artifact of research design." Annual Review of Criminology 2 (2019): 53-71. https://www.annualreviews.org/doi/abs/10.1146/annurev-criminol-011518-024638?journalCode=criminol.

[6] O'Brien, Daniel Tumminelli, and Robert J. Sampson. "Public and private spheres of neigh-

The above is an astute summation of the problems that arise with big data and how they can be overcome. The biggest of concerns, as mentioned in the opening of this section, is the validity of the data we are using.

5.2.1 Defining the phenomenon

The method that they propose requires us to do three main The first is to clearly define the phenomenon that we are hoping to measure. Following, we must idenify the **relevant data**. For example, in O'Brien and Sampson (2015), they define five ecometrics as

- Public social disorder, such as panhandlers, drunks, and loud disturbances;
- Public violence that did not involve a gun (e.g., fight);
- Private conflict arising from personal relationships (e.g., domestic violence);
- Prevalence of guns violence, as indicated by shootings or other inci- dents involving guns; and
- Alcohol, including public drunkenness or public consumption of alcohol.[7]

These definitions provide clear pictures as to what is being measured. The next step is to surf through your data and do your best to match variables or observations to these measures. Then, through some process—usually factor analysis—ensure that these measures are truly relevant.

5.2.2 Validating the measure

Once an ecometric has been defined and properly measured, the next step is to validate it. I think of this process similar to ground truthing in the geospatial sciences. Often when geographic coordinates are recorded an individual will go to that physical location and ensure that whatever that was recorded to be there actually is. This is what we are doing with our ecometrics. We have developed our measures, but we need to compare that to some objective truth so to say.

In Sampson & Raudenbush (1999), they develop measures of physical disorder

borhood disorder: Assessing pathways to violence using large-scale digital records." Journal of research in crime and delinquency 52, no. 4 (2015): 486-510. https://journals.sagepub.com/doi/abs/10.1177/0022427815577835.

[7] O'Brien, Daniel Tumminelli, and Robert J. Sampson. "Public and private spheres of neighborhood disorder: Assessing pathways to violence using large-scale digital records." Journal of research in crime and delinquency 52, no. 4 (2015): 486-510. https://journals.sagepub.com/doi/abs/10.1177/0022427815577835.

through their systematic social observation[8]. But in order to validate their measures, they compared their results to those of a neighborhood audit. This audit served as their ground truth and was used to make any adjustments if needed.

5.2.3 Addressing reliability

This ecometric, like most others, are naturually time bound snapshots of the social and physical world. These measures will naturally change over time. Because of this it is useful to know both how reliable the measure will be for different periods of time[9]. The authors do with with a bit of statistical finesse that is best left to them to explain. But what we are to take away is that ecometrics are often time variant and it is important for us to know at what time scale the ecometrics are intended for.

[8]Sampson, Robert J., and Stephen W. Raudenbush. "Systematic social observation of public spaces: A new look at disorder in urban neighborhoods." American journal of sociology 105, no. 3 (1999): 603-651. https://www.journals.uchicago.edu/doi/abs/10.1086/210356.

[9]O'Brien, Daniel T., Chelsea Farrell, and Brandon C. Welsh. "Looking through broken windows: the impact of neighborhood disorder on aggression and fear of crime is an artifact of research design." Annual Review of Criminology 2 (2019): 53-71. https://www.annualreviews.org/doi/abs/10.1146/annurev-criminol-011518-024638?journalCode=criminol.

Part II

Toolkit Foundations

Chapter 6

The basics

6.1 What is R and why do I care?

What is R? R is the 18th letter of the alphabet, the fourth letter in *QWERTY*—like the keyboard—and, most importantly, R is a software package for statistical computing.

First, a brief history lesson. R is a descendant of the S statistical programming language whose naissance can be traced back to 1976 in Bell Laboratories[1]. As S developed, people sought to commercialize the language. In 1993, the license as well as development and selling rights were given to a private company. From then on, and what is still the case today, S became available only as the commercialized S-PLUS[2].

Later, seeing a need for an improved statistical software environment, two researchers from the University of Auckland created a new statistical programming language, this became known as R. R was developed in the image of S. However, one important early decision to make the R-project free and open source changed its fate dramatically.

Today, the R-project is developed and maintained by a group known as the R Core who "represent multiple statistical disciplines and are based at academic, not-for-profit and industry-affiliated institutions on multiple continents"[3]. They define R as below.

[1]The S System. John Chambers. https://web.archive.org/web/20181014111802/http://ect.bell-labs.com/sl/S/.

[2]Statistical Sciences, Inc. Douglas Martin. 1996. https://github.com/JosiahParry/r-history/raw/master/lit/S/statsci-splus-death.pdf.

[3]R: Software Development Life Cycle. A Description of R's Development, Testing, Release and Maintenance Processes. March 25, 2018. The R Foundation for Statistical Computing c/o Institute for Statistics and Mathematics https://www.r-project.org/doc/R-SDLC.pdf.

> R is an integrated suite of software facilities for data manipulation, calculation and graphical display.[4]

To simplify it, R can be thought of as a fancy calculator. R was designed to do math, specifically statistics. R was designed to be extended to incude further capabilities than just statistics. Indeed it has been. While R is for all intents and purposes a programming language, one should, in theory, feel like they are doing data analysis and not programming[5].

R is unique from other commercial statistical software such as stata and SPSS. Very fundamentally, R is a free project. While it is monetarily free, free refers to "liberty, not price"[6]. In order to truly understand the adventure you will be embarking on shortly, I think it is important you familiarize yourself with the four freedoms of free software. These are:

1. The freedom to run the program as you wish, for any purpose (freedom 0).

2. The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.

3. The freedom to redistribute copies so you can help others (freedom 2).

4. The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this. [7]

These freedoms are a large part of the success of R as a language. Because of the free nature of R, academics and industry experts from around the globe are contributing to the language. This means that many new statistical techniques are first implemented in R.

The contributions that people make to R are changing the ways in which people perform data analysis. Because of this, we need to start contextualizing the tooling we use as **part of** the scientific process—not apart from it. When you engage in your analyses and work on contribute to the vast body of scientific literature, remember that without the tools you are using, much of it would not

[4]The R Project. https://www.r-project.org/about.html.

[5]The S System. John Chambers. https://web.archive.org/web/20181014111802/http://ect.bell-labs.com/sl/S/.

[6]GNU. https://gnu.org.

[7]GNU. https://gnu.org.

be possible. When you engage in science, think to yourself how you are adhering to the four essential freedoms. Are you enabling others to do with your findings as they wish? Will your research be accessible to the greater community? What will you do to "give the whole community a chance to benefit from your [work]?

6.2 The RStudio IDE

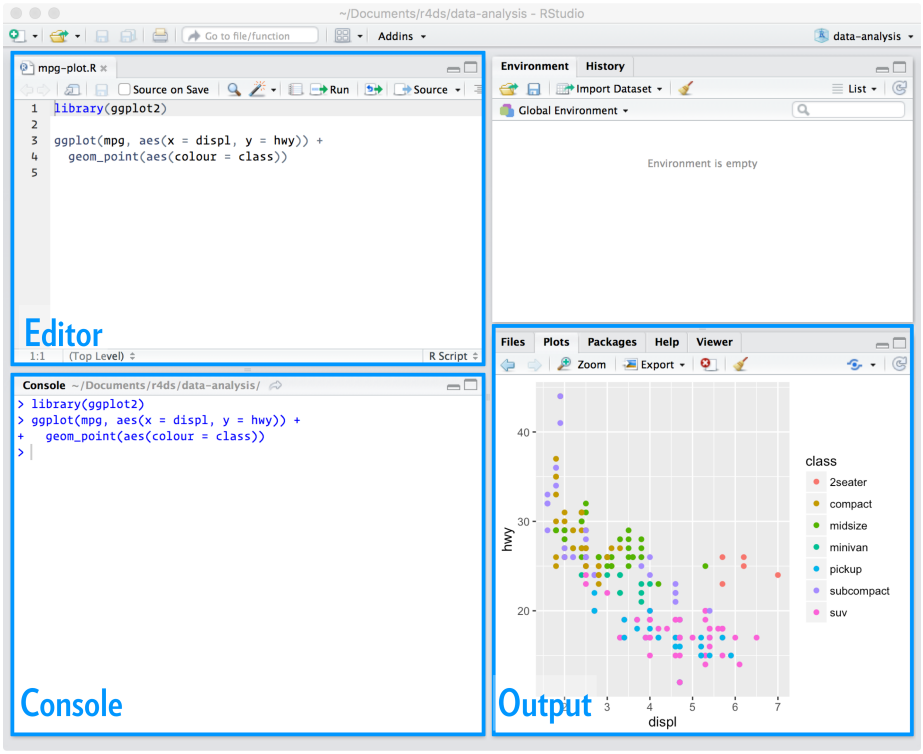
When R is downloaded, interacting with it is somewhat of a cumbersome process. While some people love it, it can feel like programming in the matrix.

For this reason, we will use RStudio to program in R. RStudio is an integrated development environment (IDE). This means that most of the features that you will need to develop in R will all be in one place. RStudio gives you a place to write your R code, execute it, view the awesome graphics you produce, and much more.

I like to think of R as typesetting a printing press and using RStudio like using Mircrosoft word. Chester Ismay and Albert Kim's Modern Dive, provide another excellent analogy of R and RStudio. They describe R as the engine of a car, and RStudio as the dashboard[8].

Let's get familiar with RStudio. You need to know where you are when working within RStudio. There are 4 quadrants that we work with (called panes).

[8]Modern Dive. https://moderndive.com/1-getting-started.html.



The above graphic is borrowed from RStudio, PBC's Thomas Mock's Introduction to the Tidyvers[9]

6.2.1 The Editor

The editor. The top left pane. This is where you will actually write your code. You will see in the image above that there is tab with the name of the R file being edited, `mpg-plot.R`. The simplest way in which R code is written, is in documents with the `.R` extention. Think of the R script as your word document. This is where you put the writing that you want to keep.

There is also a second type of R file called an RMarkdown document, `.Rmd`s. These are a special type of file that lets us intersperse regular prose with code chunks. Rmd is extremely flexibile and enabled the user to render their content in many different formats such as a pdf, powerpoint, html, and others. For example, this book is written with RMarkdown. But, to keep things simple, we will use R scripts for the vast majority of this book, plus they're my favorite way to interact with R.

[9]Intro to RMarkdown. Thomas Mock. https://github.com/jthomasmock/intro-tidyverse/blob/master/intro-to-tidyverse.rmd.

6.2.2 The console

Let us now avert our attention to the bottom left pane. This is known as the console. The console is where your R code is actually execute. When you run a line or chunk of code from your editor, you will see it processed in the console. I often treat my console as my scratch paper. This is a place where I can explore R objects and code without affecting the primary R file. You should become comfortable typing your R code in the editor and see it executed in the console.

6.2.3 Output

Now moving over to the right. This is the most versatile quadrant of RStudio. You will primarily use this quadrant to look at things. There is a pane for navigating your files, looking at help documentation, viewing the charts that you produce, and any interactive applications you may develop.

6.3 Installing R & RStudio

Now that you are somewhat familiar with R and RStudio, it is time to install them. I recommend installing R *and then* RStudio.

R can be downloaded from the **C**entral **R** **A**rchival **N**etwork (CRAN). CRAN is the official location for all things R. CRAN provides access to the R software, license, copyright, and software extensions (called R packages). Go to CRAN to download R[10].

RStudio is provided by RStudio, Public Benefit Corporation (RStudio, PBC). To install RStudio navigate to the download page[11]. Once both have been installed you can open RStudio to get started. Look for the circular R logo. If you get lost navigating the RStudio IDE, be sure to refer to the cheat sheet.

6.4 Preventative Care

6.4.1 R Projects

Once you open up RStudio you will be able to get rocking and rolling. Though, I want to instill some best practices from the get go. This following section will save you and anyone who you collaborate with an undescribable amount of headaches. Folks are tempted to open up RStudio and begin doing analysis. That is all well and good though this leads to many problems. We need to

[10] CRAN. https://cran.r-project.org/.
[11] RStudio IDE Download. https://rstudio.com/products/rstudio/download/#download.

contextualize each and every analysis as it's own **project**. Currently I mean project in the conceptual manner. If there is a common overarching theme, intent, or purpose, that analysis should be delineated as its own project and should be identifiable from others.

You probably already have a notion of projects implemented in your life. Consider your school work. I suspect, and frankly hope, that you have a somewhat organized folder structure where each semester is its own folder, and each course is its own folder within that. An example of what some of my folder organization looks like is below.

```
fall/
  big-data-for-cities/
    projects/
  urban-theory/
    literature/

spring/
  info-design/
    data/
  intro-data-mining/
```

In the above case we would consider each course as its own project. The important thing to keep in mind here is that each project is self-contained. By working inside of self-contained folders we can ensure that there are no problems with accessing files. R uses the concept of a working directory. Think of the working directory as "where do I start when I am looking for things?"

Imagine R is using the working directory `spring` but you are working on your `info-design` work. Conceptually, you feel as if you're working from the `info-design` folder, but R is actually under the impression you are working from `spring`. So when you try to load some data from the `data` folder inside of `info-design` (which looks like `info-design/data`), you have to tell R how to get there. And the way R would get there is probably different than you think at the time.

To prevent this, we can essentially level set with R by creating a project. An R project gently imposes the standalone structure that we will need to prevent most of the headaches described above.

The way to create a new project is by navigating to `File > New Project`.

Click `New Directory`. This will create a new folder for you. Next, RStudio will prompt you to specify what kind of project to create. Today, that will be a generic `New Project`. In the future, I suspect you will end up creating Shiny applications and much more.

The final step is to specify what the project will be called and where to put it. In the below image I name the new directory `uitk`, short for *Urban Informatics*

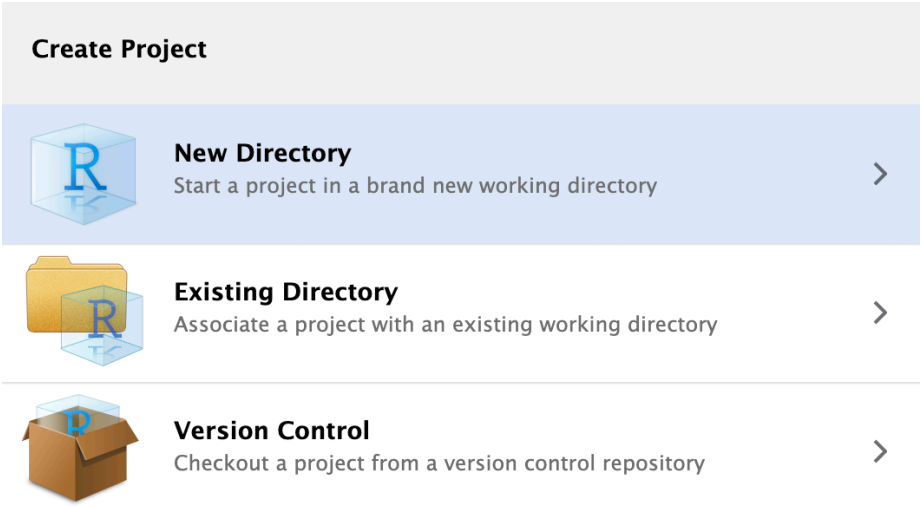


Figure 6.1: Step 1: new Project

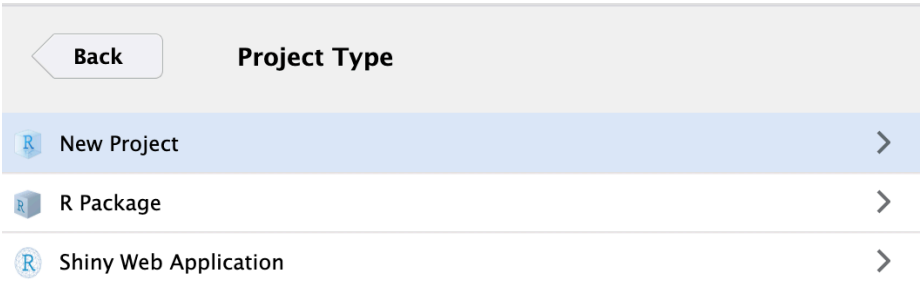


Figure 6.2: Step 2: New Project

Toolkit, and place it in the directory `R`. Be sure to select which folder you want your project to live in.

A few tips:

- Think about where you will be able to find the project again
- Where would it make the most sense for the project to live?
- **Do not** put spaces or periods in the directory name. Use `_` or `-` if you feel the need.

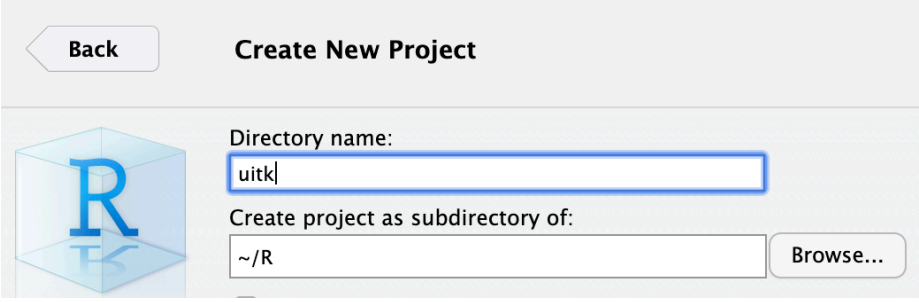


Figure 6.3: Step 3: Naming the project

This will open up a new RStudio session. You will notice in your `Files` pane that there is now a `uitk.Rproj` file there. That file is what tells RStudio about the project, so don't delete it! If you would like to open up an RStudio project you can either open the `.Rproj` file from your file navigator or open it by following `File > Open Project`.

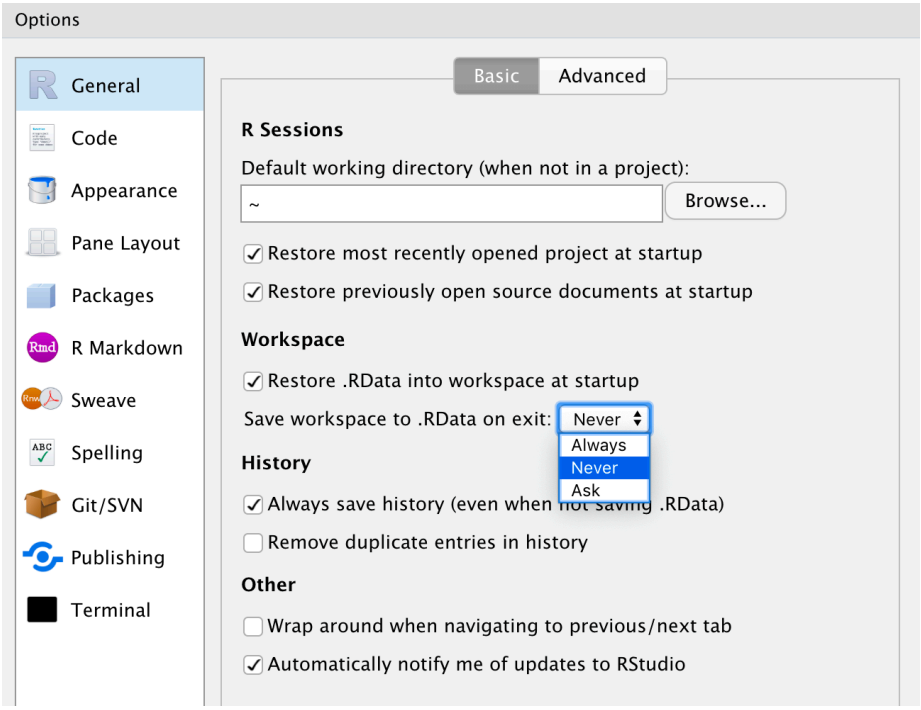
6.4.2 The data

In order to complete the exercises throughout this book you will need to have the data accessible to you. You can download the data here[12]. This link will download a file called `data.zip`. Once downloaded open the file. It will create a folder called `data`. Move that entire folder into your new project. If you created a project called `uitk` in the `R` directory, move the folder to `~/R/uitk`. This will create a folder path of `~/R/uitk/data`.

6.4.3 Your Workspace

In another effort to impose good habits and reproducibility standards I will suggest you change one setting in RStudio. Navigate to `Tools > Global Options` now change the below setting.

[12] Urban Informatics Toolkit Data. https://github.com/JosiahParry/urban-informatics-toolkit/raw/master/data.zip

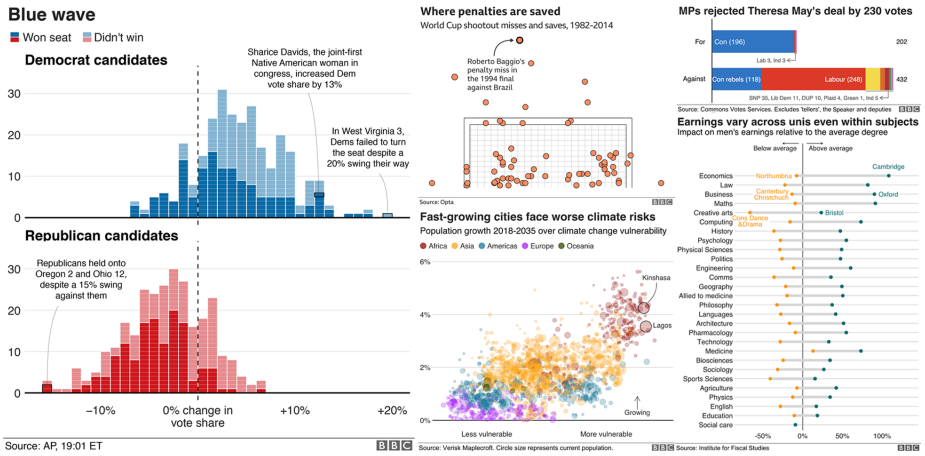


This setting makes it so that your analysis is dependent upon the code you write, not the things you create while interactively programming. A general rule of thumb is that your R script should be able to run from top to buttom successfully.

6.5 Before we embark

Lastly, I want to emphasize that R and RStudio can be used for so much more than statistical analysis. It can be used to make aRt.

It can be used to make beautiful graphics for the BBC.



R can be found in the infrastructure of our modern world. R is utilized in our global financial institutions, civil rights groups such as the ACLU, investigative journalism, national defense, and so much more. Do not feel that the *only* thing you will get from learning R is how to do some simple statistics.

6.6 Getting Help

Undoubtedly you will run into problems when programming and you're going to want or need help. If you don't know where to go, getting help may feel impossible. There are two main places that I recommend you go to for help when you encounter problems: Stack Overflow and RStudio community.

Stack Overflow is one piece of the large Stack Exchange network that is dedicated strictly to programming issues[13]. There is a good chance that you've encountered Stack Exchange at some point in your various google searches. Stack Exchange is a network that is strictly dedicated to a question and answer format. Each topic has their own subdomain. For those who are unfamiliar, think of Stack Exchange like Reddit where each subreddit is their own own Q&A page. And for those who are unfamiliar with Reddit, think of Stack Overflow like Yahoo! Answers or Quora. Stack Overflow has a huge database of questions and answers for all of the programming problems that folks have encountered. It is very unlikely to have a question that has not been answered on Stack Overflow before.

The RStudio Community page is a community forum created by RStudio[14]. This is a location for members of the R community—which you can now count yourself a part of—to ask questions, engage in thoughtful dialogue, and much more. While Stack Overflow is committed to all programming languages, of

[13] Stack Overflow. https://stackoverflow.com.

[14] RStudio Community. https://community.rstudio.com/.

which R is just one of them, the RStudio Community is maintained entirely by R users.

If you have never asked a technical question before I would recommend spending doing so on RStudio Community as Stack Overflow has a history of somewhat rude community[15]. Before you do so, be sure to create a **rep**roducible **ex**ample so that the community can best help you[16].

6.7 Reminders

Learning to program can be exceptionally difficult and frustrating at times. It can be a roller coaster of emotions. It is expected that you will not understand everything the first go around. Do not get down on yourself. I encourage you to take breaks and not push yourself too hard or even be self-critical. I understand you have deadlines, but sometimes it is better if you take a break, eat a healthy snack, go exercise, sleep, be social, or do whatever makes you happy and then come back. You will be much happier and your work will be even better and that I promise you!

If you ever find yourself in a bout of programming induced frustration try one of the below:

- Drink water! Just do this even if you don't feel like it. Water is always good.
- Get some sleep. Without sleep you wil be running at 60% or less.
- Eat your greens. You are what you eat!
- Shower. Feeling clean can change your perspective and approach.
- Get your blood flowing! Go for a walk. Do some squats or pushups. Making sure your blood is moving is important.
- And take care of yourself!

Now, let's get going.

[15]The Rudeness on Stack Overflow is Too Damn High. https://meta.stackoverflow.com/questions/262791/the-rudeness-on-stack-overflow-is-too-damn-high.

[16]Reproducible Examples. https://www.tidyverse.org/help/#reprex.

Chapter 7

R as a calculator

Before we get going, let's find our footing. R is a statistical programming language. That means that R does math and pretty well too. In this chapter you'll learn the basics of using R including:

- arithmetic operators
- creating and assigning variables
- using functions

7.1 Arithmetic Operators

Do you remember PEMDAS? If not, a quick refresher that PEMDAS specifies the order of operations for a math equation. Do the math inside of the parentheses, then the exponents, and then the multiplication or the division before addition or subtraction. We can't write the math out, so we need to type it out. Below are the basic arithmetic operators

- `^` : exponentiation (exponents) [E]
- `*` : multiplication [M]
- `/` : division [D]
- `+` : addition [A]
- `-` : subtraction [S]

These can be used together in parentheses [P] `( )` to determine the order of operations (`PEMDAS`)

There are three different ways to execute code inside of RStudio[1]. The easiest way is to have your cursor on the line of code that you would like to execute. To

[1] Executing Code. https://support.rstudio.com/hc/en-us/articles/200484448-Editing-and-Executing-Code#executing.

execute hold `command + enter` (Mac) or `control + enter` (PC). Alternatively you can press the `Run` button at the top of the source page.

Now, try out some mathematic expressions in the console.

7.2 Variable assignment

I'm sure you recall some basic algebra questions like $y = 3x - 24$. In this equation, x and y are variables that represents some value. We will often need to create variables to represent some value or set of values. In R, we refer to variables as **objects**. Objects can be a single number, a set of words, matrixes, and so many other things.

To create an object we need to assign it to some value. Object assignment is done with the assignment operator which looks like `<-`. You can automagically insert the assignment operator with `opt + -`.

Let's work with the above example. We will solve for `y` when `x` is equal to 5.

First, we need to assign 5 to the variable x.

```
x <- 5
```

If you want to see the contents of an object, you can print it. To print an object you can type the name of it.

```
x
```

```
## [1] 5
```

We can reference the value that `x` stores in other mathematic expressions. Now what does `y` equal? Now solve for y in the above equation!

```
y <- 3 * x - 24

y
```

```
## [1] -9
```

7.3 Functions

Functions are a special kind of R object. Very simply, a function is an object that performs some action and (usually) produces an output. Functions exist

to simplify a task. You can identify a function by the parentheses that are appended to the function name. A function looks like `function_name()`.

R has many functions that come built in. The collection of functions that come out of the box with R are called *Base R**.

An example of a simple base R function is `sum()`. `sum()` takes any number of inputs and calculates the sum of those inputs.

We can run `sum()` without providing any inputs.

```
sum()
```

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

We can provide more inputs (formally called function arguments) to `sum()`. For example to find the sum of 10 we write

```
sum(10)
```

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

The sum of a single number is the number itself. We can provide more arguments to `sum()`. Additional arguments are specified by separating them with commas—e.g. `function(argument_1, argument_2)`.

To find the sum of 10, 3, and 2 we write `sum(10, 3, 2)`.

```
sum(10, 3, 2)
```

```
## [1] 15
```

Much of the analysis we will do is done with functions. You will become much more comfortable with them rather quickly.

If you ever need to know how a function works, you can look at its help page by typing `?function_name()` in your console. That will bring up the documentation page in the bottom right pane.

7.4 Extensions to R

While R was created as a statistical programming language, it was designed with the intention of being extended to include even more functionality. Extensions to R are called *packages*. R packages often provide a set of functions to accomplish a specific kind of task.

To analyse, manipulate, and visualize our data, we will use a number of different packages to do so. Throughout this book we will become familiar with a set of packages that together are known as the Tidyverse.

R packages do not come installed out of the box. We will need to install them our selves. Base R includes a function called `install.packages()`. `install.packages()` will download a specified package from CRAN and install it for us.

To download packages, we must tell `install.packages()` which package to download. We will provide the name of the package as the only argument to `install.packages()`. The name of the package needs to be put into quotations such as `install.packages("package-name")`.

> Note: By putting text into quotations we are creating what is called a **character string**.

> Reminder: we create objects with the assignment operator `<-`.

When we don't use quotes (create a character string), R thinks we are referring to an object we have created.

7.4.1 Exercise

Use your new knowlege of functions and installing packages to install the tidyverse.

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

7.5 Loading Packages

Now that you have installed the tidyverse, you are going to need to know how to make it available to you for use. To load a package, we use the function `library()`. Oddly, though, when specifying which package to load, we do not put that name in quotations.

> Note: It is best practice to load all of your packages at the top of your R script.

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------
```

```
## v ggplot2 3.3.0     v purrr   0.3.3
## v tibble  3.0.0     v dplyr   0.8.5
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts ------------------------------------------------------------------------------------ tidyverse_co
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Notice the message above. When we load the tidyverse, we are actually loading eight packages at once! These are the packages listed under "**Attaching packages**."

You have now successfully installed and loaded the tidyverse. Next, we will begin to learn how to visually analyze data!

7.6 What's next?

Now with this very rudimentary foundation underneath you we will embark upon a very long journey—the journey to learn how to analyze data with R and RStudio. In the following chapters you will learn how to program in R from creating compelling graphics, manipulating data, and performing statistical tests.

In the next chapter we will dive right into working with data. We will begin by learning how to read data into R. Then, we will work through a first visual analysis after which we will focus on skills used in general data manipulation.

Chapter 8

Reading data

Requisite to any data analysis is the data. Making those data available for you to analyse is not always the easiest of tasks. In this chapter we will review how data are imported and some of the formats they may take. Once we complete this chapter we will get going on our very first analaysis!

8.1 Background

There are three general sources where we as social scientists will receive or access data: 1) text files, 2) databases, and 3) application programming interfaces (APIs). Frankly, though this is the age of "big data," we are not always able to interface directly with these sources. But through partnership efforts between the public and private we able to share data. For example, BARI's work with the Boston Police Department provides them with annual access to crime data. But BARIs access is limited. They do not have credentials to log in to the database and perform their own queries. What they are usually presented with is a flat text file(s) that contains the data requisite for analysis. And this is what we will focus in this chapter.

Flat text files will be sufficient for 85% of all of your data needs Now, what do I mean by *flat text file*? A flat text file is a file that stores data in plain text—I know, this seems somewhat confusing. In otherwords, you can open up a text file and actually read the data with your own eyes or a screenreader. For a long while tech pundits believed—and some still do—that text data will be a thing of the past. Perhaps this may be true in the future, but plain text still persists and there are some good reasons for that. Since plain text is extremely simple it is lightweight and usually does not take up that much memory. Also, because there is no fancy embelishing of the data in a plain text file, they can be easily shared from machine to machine without concern of dependent tools

and software. Not to mention that we humans can actually be rather hands on and inspect the source of the data ourselves.

8.2 Actually Reading Data

Within the `tidyverse` there is a package called `readr` (pronounced *read-r*) which we use for reading in rectangular data from text files.

I just threw the phrase *rectangular data* at you. It is only fair to actually describe what that means. If you were to look at rectangular data in something like excel it would resemble a rectangle. In fancy speak, rectangular data is a *two-dimensional* data structure with rows and columns. We will learn more about the "proper" way to shape rectangular data in the "tidying data" chapter. For now, all you need to know is that there are rows and columns in rectangular data.

To get started, let us load the tidyverse. This will load readr for us.

```
library(tidyverse)
```

You most likely have seen and encountered flat text files in the wild inthe form of a `csv`. It is important to know what *csv* stands for because it will help you understand what it actually is. it stands for **c**omma **s**eparated **v**alues. _csv_s are a flat text data file where the data is rectangular! Each new line of the file indicates that there is a new row. Within each row, each comma indicates a new column. If you opened one up in a text editor like text edit or notepad a csv would look something like below.

```
column_a, column_b, column_c,
10, "these are words", .432,
1, "and more words", 1.11
```

To read a csv we use the `readr::read_csv()` function. `read_csv()` will read in the csv file and create a `tibble`. A tibble is type of a data structure that we will be interacting with the most throughout this book. A tibble is a rectangular data structure with rows and columns. Since a csv contains rectangular data, it is natural for it to be stored in a tibble.

> Note: the syntax above is used for referencing a function from a namespace (package name). The syntax is pkgname::function(). This means the `read_csv()` function from the package `readr`. This is something you will see frequently on websites like StackOverflow.

Have a look at the arguments of `read_csv()` by entering `?read_csv()` into the console. You will notice that there are many arguments that you can set. These are there to give you a lot of control over how R will read your data. For now, and most of the time, we do not need to be concerned about these extra arguments. All we need to do is tell R *where* our data file lives. If you haven't deduced from the help page yet, we will supply only the first argument `file`. This argument is *either a path to a file, a connection, or literal data (either a single string or a raw vector).*

> Note: When you see the word *string*, that means values inside of quotations—i.e. *"this is a string"*.

We will read in the dataset we will use in the next chapter. These data are stored in the file named `acs_edu.csv`. We can try reading this as the file path.

```
read_csv("acs_edu.csv")

## Error: 'acs_edu.csv' does not exist in current working directory
##   ('/Users/Josiah/GitHub/urban-commons-toolkit').
```

Oops. We've got red text and that is never fun. Except, this is a very important error message that, frankly, you will get **a lot**.

Again it says:

> *Error: 'acs_edu.csv'* ***does not exist*** *in current* ***working directory***

I've bolded two portions of this error message. Take a moment to think through what this error is telling you.

For those of you who weren't able to figure it out or just too impatient (like myself): this error is telling us that R looked for the file we provided `acs_edu.csv` but it could not find it. This usually means to me that I've either misspelled the file name, or I have not told R to look in the appropriate folder (a.k.a. directory).

`acs_edu.csv` actually lives in a directory called `data`. To tell R—or any computer system, really—where that file is we write `data/acs_edu.csv`. This tells R to first enter the `data` directory and then look for the `acs_edu.csv` file.

Now, read the `acs_edu.csv` file!

```
read_csv(file = "data/acs_edu.csv")
#> Parsed with column specification:
#> cols(
#>   med_house_income = col_double(),
```

```
#>   less_than_hs = col_double(),
#>   hs_grad = col_double(),
#>   some_coll = col_double(),
#>   bach = col_double(),
#>   white = col_double(),
#>   black = col_double()
#> )
#> # A tibble: 1,456 x 7
#>    med_house_income less_than_hs hs_grad some_coll  bach white   black
#>               <dbl>        <dbl>   <dbl>     <dbl> <dbl> <dbl>   <dbl>
#>  1           105735       0.0252   0.196     0.221 0.325 0.897 0.0122
#>  2            69625       0.0577   0.253     0.316 0.262 0.885 0.0171
#>  3            70679       0.0936   0.173     0.273 0.267 0.733 0.0795
#>  4            74528       0.0843   0.253     0.353 0.231 0.824 0.0306
#>  5            52885       0.145    0.310     0.283 0.168 0.737 0.0605
#>  6            64100       0.0946   0.294     0.317 0.192 0.966 0.00256
#>  7            37093       0.253    0.394     0.235 0.101 0.711 0.0770
#>  8            87750       0.0768   0.187     0.185 0.272 0.759 0.0310
#>  9            97417       0.0625   0.254     0.227 0.284 0.969 0.00710
#> 10            43384       0.207    0.362     0.262 0.124 0.460 0.105
#> # ... with 1,446 more rows
```

This is really good! Except, all that happened was that the function was ran. The data it imported was not saved anywhere which means we will not be able to interact with it. What we saw was the output of the data. In order to interact with the data we need to assign it to an object.

> Reminder: we assign object with the assignment operator `<-`— i.e. `new_obj <- read_csv("file-path.csv")`. Objects are things that we interact with such as a tibble. Functions such as `read_csv()` *usually*, but not always, modify or create objects.

In order to interact with the data, let us store the output into a tibble object called `acs`.

```
acs <- read_csv(file = "data/acs_edu.csv")
```

Notice how now there was no data printed in the console. This is a good sign! It means that R read the data and stored it properly into the `acs` object. When we don't store the function results, the results are (usually) printed out. To print an object, we can just type it's name into the console.

```
acs
#> # A tibble: 1,456 x 7
#>    med_house_income less_than_hs hs_grad some_coll  bach white   black
#>               <dbl>        <dbl>   <dbl>     <dbl> <dbl> <dbl>   <dbl>
#>  1           105735       0.0252   0.196     0.221 0.325 0.897 0.0122
#>  2            69625       0.0577   0.253     0.316 0.262 0.885 0.0171
#>  3            70679       0.0936   0.173     0.273 0.267 0.733 0.0795
#>  4            74528       0.0843   0.253     0.353 0.231 0.824 0.0306
#>  5            52885       0.145    0.310     0.283 0.168 0.737 0.0605
#>  6            64100       0.0946   0.294     0.317 0.192 0.966 0.00256
#>  7            37093       0.253    0.394     0.235 0.101 0.711 0.0770
#>  8            87750       0.0768   0.187     0.185 0.272 0.759 0.0310
#>  9            97417       0.0625   0.254     0.227 0.284 0.969 0.00710
#> 10            43384       0.207    0.362     0.262 0.124 0.460 0.105
#> # ... with 1,446 more rows
```

This is sometimes a little overwhelming of a view. For previewing data, the function `dplyr::glimpse()` (there is the namespace notation again) is a great option. Try using the function `glimpse()` with the first argument being the `acs` object.

```
glimpse(acs)
#> Rows: 1,456
#> Columns: 7
#> $ med_house_income <dbl> 105735, 69625, 70679, 74528, 52885, 64100, 37093, ...
#> $ less_than_hs     <dbl> 0.02515518, 0.05773956, 0.09364548, 0.08426318, 0....
#> $ hs_grad          <dbl> 0.19568768, 0.25307125, 0.17332284, 0.25298192, 0....
#> $ some_coll        <dbl> 0.2211696, 0.3157248, 0.2726736, 0.3534052, 0.2830...
#> $ bach             <dbl> 0.32473048, 0.26167076, 0.26677159, 0.23124279, 0....
#> $ white            <dbl> 0.8972737, 0.8849885, 0.7328322, 0.8235779, 0.7371...
#> $ black            <dbl> 0.012213740, 0.017090069, 0.079514240, 0.030640286...
```

8.3 Other common data formats

While csv files are going to be the most ubiquitous, you will invariably run into other data formats. The workflow is almost always the same. If you want to read excel files, you can use the function `readxl::read_excel()` from the `readxl` package.

```
acs_xl <- readxl::read_excel("data/acs_edu.xlsx")

glimpse(acs_xl)
#> Rows: 1,456
```

```
#> Columns: 7
#> $ med_house_income <dbl> 105735, 69625, 70679, 74528, 52885, 64100, 37093, ...
#> $ less_than_hs     <dbl> 0.02515518, 0.05773956, 0.09364548, 0.08426318, 0....
#> $ hs_grad          <dbl> 0.19568768, 0.25307125, 0.17332284, 0.25298192, 0....
#> $ some_coll        <dbl> 0.2211696, 0.3157248, 0.2726736, 0.3534052, 0.2830...
#> $ bach             <dbl> 0.32473048, 0.26167076, 0.26677159, 0.23124279, 0....
#> $ white            <dbl> 0.8972737, 0.8849885, 0.7328322, 0.8235779, 0.7371...
#> $ black            <dbl> 0.012213740, 0.017090069, 0.079514240, 0.030640286...
```

Another common format is a **tsv** which stands for **t**ab **s**eparated **f**ormat. `readr::read_tsv()` will be able to assist you here.

If for some reason there are special delimiters like `|`, the `readr::read_delim()` function will work best. For example `readr::read_delim("file-path", delim = "|")` would do the trick!

Additionally, another extremely common data type is *json* which is short for javascript object notation. json is a data type that you will usually not read directly from a text file but interact with from an API. If you do happen to encounter a json flat text file, use the `jsonlite` package. `jsonlite::read_json()`.

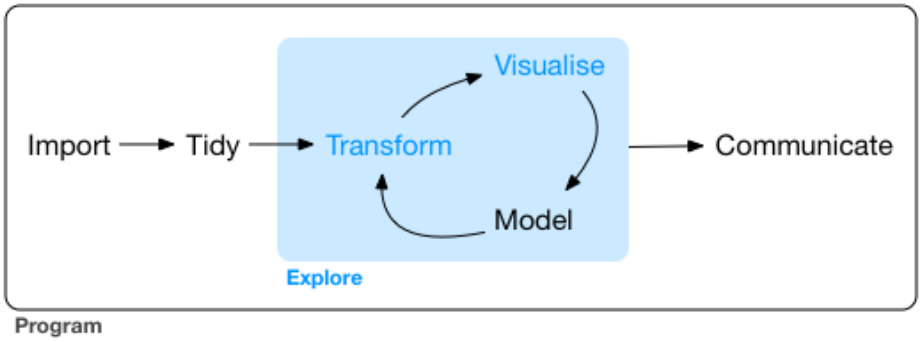
With this new skill we are ready for our first analysis. In the next chapter we will perform our very first graphical analysis using the package `ggplot2` from the tidyverse.

Chapter 9

Exploratory Visual Analysis

In the next part of this book we will introduce to you visual data exploration through the use of the R package `ggplot2`. You will ask questions of your data, visualize relationships, and draw inferences from the graphics you develop.

The below image from R for Data Science is renowned for its representation of the data analysis workflow. The concept map encompasses the need to get data (import), clean it up (tidy), explore, and finally communicate insights. The box in blue is a representation of EDA. Within EDA we will find ourselves transforming our data—creating new variables, aggregating, etc.—visualizing it, and creating statistical models.



This chapter will focus on the visualization step of EDA. We have all heard the trope that "an image is worth a thousand words." I'd take a leap and say that a good visualization is worth ten thousand words. An older statistical manual from the National Institute of Standards and Technology (NIST) beautifully lays out the role that visualization plays in EDA.

> The reason for the heavy reliance on graphics is that by its very nature the main role of EDA is to open-mindedly explore, and

> graphics gives the analysts unparalleled power to do so, enticing the data to reveal its structural secrets, and being always ready to gain some new, often unsuspected, insight into the data. In combination with the natural pattern-recognition capabilities that we all possess, graphics provides, of course, unparalleled power to carry this out.[1]

source

In the following section, you will become acquainted with the graphical R package `ggplot2` for visual analysis and the American Community Survey. We will walk through the process building a chart from the ground up and drawing inferences from it along the way.

9.1 The American Community Survey

For this first data exploration we will work with data from the American Community Survey (ACS). While the ACS is central to Urban Informatics (UI), it does not exhibit the primary characteristic of that we rely upon in UI—namely being naturally occurring. This is a topic we will explore in more depth later. In order to use the ACS data, we must understand what data we are actually working with. The ACS is one of the most fundamental data sets in American social sciences. The ACS is administered by the US Census Bureau but is done so for much different purposes. Article I Section 2 of the US Constitution legislates a decennial census.

> . . . [an] enumeration shall be made within three Years after the first Meeting of the Congress of the United States, and within every subsequent Term of ten Years, in such Manner as they shall by Law direct.

The above clause requires the US government to conduct a complete counting of every single individual in the United States for the purposes of determining how many congressional representatives each state will have. These censuses provided a detailed image of *how many* people there were in the US, but lacked much information beyond that. The first census asked each household for "the number of free white males under 16 years" and of "16 years and upward", the "number of free White females", "number of other free persons, and the"number of slaves"[2] Since then the breadth of questions asked during the census has increased as well as other supplementary sources of information.

The ACS was developed in response to two shortcomings of the decennial census. The first being that the census only occurrs every ten years. There was, and

[1] NIST Handbook. https://www.itl.nist.gov/div898/handbook/eda/section1/eda11.htm.

[2] https://www.census.gov/history/www/through_the_decades/index_of_questions/1790_1.html

still is, a need for more consistent and current data. Not only are the censuses too infrequent, but they also do not provide the most colorful picture of who it is that lives within the US. Local, state, and federal governments desired more context about who their constituents are.

The ACS was developed and first officially released in 2005[3]. The ACS uses a "series of monthly samples" to "produce annual estimates for the same small areas (census tracts and block groups) formerly surveyed via the decennial census long-form sample"[4]. As Catherine Rampell wrote in the New York times

> "It tells Americans how poor we are, how rich we are, who is suffering, who is thriving, where people work, what kind of training people need to get jobs, what languages people speak, who uses food stamps, who has access to health care, and so on."[5]

The impact of the ACS are wide stretching from funding to social research.

9.1.1 Understanding ACS Estimates

Continuous sampling done by the US Census Bureau occurs at a monthly basis and are used to produce annual estimates[6]. There are two different types of estimates one can retrieve from the ACS. These are the 1-year and 5-year estimates. Each kind of estimate serves a different purpose.

When choosing between 1-year and 5-year estimates we are making a tradeoff. 1-year estimates provide us with the most current data possible at the expense of a smaller sample size. This means that the estimates are not as reliable as the 5-year estimates which are collected over a period of 60 months. On the other hand, when we consider 5-year estimates, we benefit from a large sample size and increased reliability, but we lose the ability to make statements about a single year.

In the cases where 5-year estimates are used researchers are analyzing populations and rates derived from five years of data collection. This requires you, the researcher, to qualify this with a statement to the effect of *"the rate of unemployment in 2014-2018 was 4%"*[7]. Because of this, you are unable to use consecutive 5-year estimates to analyze annual trends. In the case that you need to analyse annual trends 1-year estimates are recommended.

[3] https://www2.census.gov/programs-surveys/acs/methodology/design_and_methodology/acs_design_methodology_report_2014.pdf

[4] https://www.census.gov/programs-surveys/acs/methodology.html

[5] https://www.nytimes.com/2012/05/20/sunday-review/the-debate-over-the-american-community-survey.html

[6] https://www2.census.gov/programs-surveys/acs/methodology/design_and_methodology/acs_design_methodology_report_2014.pdf

[7] https://www.census.gov/content/dam/Census/library/publications/2018/acs/acs_general_handbook_2018_ch03.pdf

There is also another important tradeoff one must consider when using ACS data and that is of unit of analysis. The census and ACS are conducted at the household level. However, estimates are provided for geographic areas. These geographic areas have a hierarchy going from block groups at the smallest level geography, to census tracts, and to counties. Beyond counties are geographies at the state level and even larger. Urban informatics is inherently focuses on a more micro—arguably meso—unit of analysis.

The following analysis is done using the Massachussetts census indicators published from BARI. The dataset is based on 5-year estimates from the ACS at the tract level.

9.2 A first visualization

For your first introduction to R, we will explore the relationship between education and income in Massachusetts.

NOTE: this needs to be updated once I make `uitk` a package with R objects

9.2.1 Familiarize yourself

There is no one best way to begin an exploratory analysis to guarantee interesting outcomes. But before one begins their EDA, they must know what their data actually contain. We will use the `read_csv()` function from the last chapter to read in the `acs_edu` dataset. Save it to a variable called `acs_edu`.

```
library(tidyverse)
acs_edu <- read_csv("data/acs_edu.csv")
```

`acs_edu` contains data demographic information about every census tract in Massachusetts. Print `acs_edu` to the console. What do you see?

```
acs_edu
#> # A tibble: 1,456 x 7
#>    med_house_income less_than_hs hs_grad some_coll  bach white   black
#>               <dbl>        <dbl>   <dbl>     <dbl> <dbl> <dbl>   <dbl>
#>  1           105735       0.0252   0.196     0.221 0.325 0.897 0.0122
#>  2            69625       0.0577   0.253     0.316 0.262 0.885 0.0171
#>  3            70679       0.0936   0.173     0.273 0.267 0.733 0.0795
#>  4            74528       0.0843   0.253     0.353 0.231 0.824 0.0306
#>  5            52885       0.145    0.310     0.283 0.168 0.737 0.0605
#>  6            64100       0.0946   0.294     0.317 0.192 0.966 0.00256
#>  7            37093       0.253    0.394     0.235 0.101 0.711 0.0770
#>  8            87750       0.0768   0.187     0.185 0.272 0.759 0.0310
```

```
#>  9            97417       0.0625   0.254     0.227 0.284 0.969 0.00710
#> 10            43384       0.207    0.362     0.262 0.124 0.460 0.105
#> # ... with 1,446 more rows
```

`## # A tibble: 1,456 x 7` is printed out at the top followed by column names, their types—e.g. `<dbl>`—their respective values and, to the far left we see the numbers 1 through 10 before each row of values.

Let us dissect `# A tibble: 1,456 x 7` a little bit more. This alone is quite informative. It tells us that the type of object we are working with is a `tibble` with 1,456 rows and 7 columns.

A tibble is a method of representing rectangular data and is very similar to a table one may create within Excel with rows an columns. When working with tibbles we try to adhere to what are called the principles of tidy data[8]. There are three key principles that we ought to keep in mind when working with rectangular data.

1. Each variable forms a column.
2. Each observation forms a row.
3. Each value represents a combination of an observation and a variable.

There can often be confusion about what should be a variable and what is to be an observation. In *Tidy Data* Hadley Wickham write that

> "A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes."[9]

Say we have a tibble of survey respondents. In this case each row should be a respondent and each column should be a variable that is associated with that respondent. This could be something such as age, birth date, or the respondents response to a survey question.

In the case of our `acs_edu` tibble, our unit of observation, aka row, is a census tract. Each variable measures a different characteristic of a census tract. For example, the column `med_house_income` is an estimate of the median household income of a given census tract. The other columns indicate what proportion of a population meets some criteria.

How does one know what criteria their columns represent? This brings us to the importance of column names. Column names ought to be descriptors of

[8] https://vita.had.co.nz/papers/tidy-data.pdf
[9] https://vita.had.co.nz/papers/tidy-data.pdf

their corresponding variables. This is a surprsingly difficult task! In `acs_edu` we can infer—though we should always have documentation to supplement the data—that the variables measure income, educational attainment rates, and race.

9.2.2 Form a question

Once you have familiarized yourself with the data that you will be working with, you can begin to form a question that can be feasibly be explored or answered with the present data. The importance of domain expertise in EDA cannot be understated. Without an understanding of what underlying phenomena your data are measuring it will be extremely difficult to come to meaningful insights.

My background is in sociology. Within sociology, and specifically social stratification, it is believed that more education leads to more social prestiege, economic stability, and is more readily accessible by the white population. Given this background and the data available in `acs_edu`, we will explore the relationship between education and income. We will try to answer the question *what is the relationship between education and income?* We will first look at each variable in isolation and then try and identify any relationship that may exist between the two variables.

9.2.3 Building a graph

To create our visualizations we will use the package `ggplot2` from the tidyverse. Before we can begin, we need to make sure that the collection of functions from ggplot2 are available to us with the `library()` function.

> Reminder: `library(pkg_name)` loads a package into your workspace and makes the functions and objects it exports available to you.

```
library(ggplot2)
```

To begin building a ggplot, we use the function `ggplot()`. There are two function arguments these being `data` and the aesthetics `mapping`. The `data` is the tibble that we wish to visualize. In this case we want to visualize the data from `acs_edu`.

We will begin constructing our first visualization with the `ggplot()` **function** using the `acs_edu` **object**.

> Reminder: Functions are characterised by the parentheses at the end of them. Functions do things whereas objects hold information.

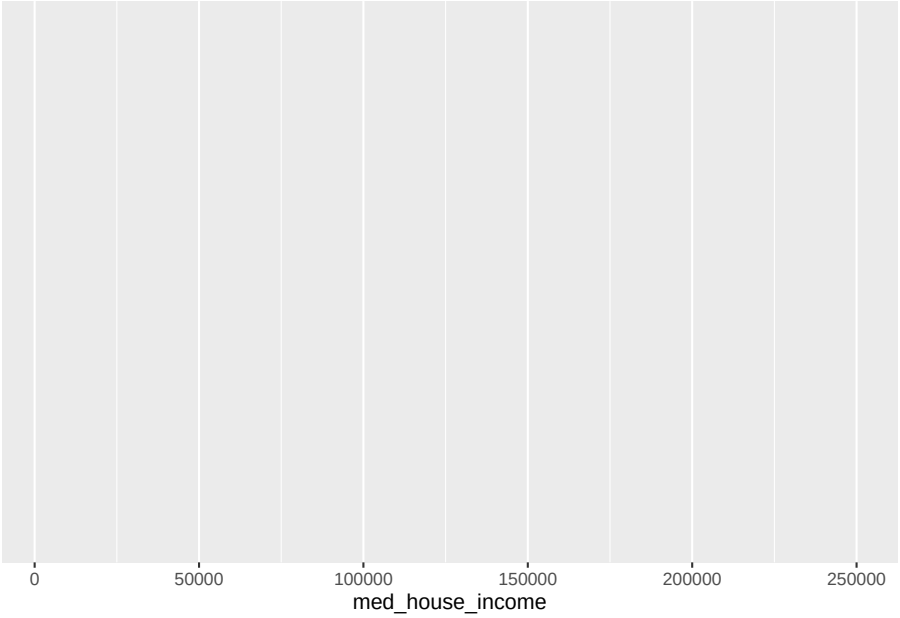
```
ggplot(acs_edu)
```


Notice that this plot is entirely empty. This is because we have not defined what it is that we want to visualize. ggplot uses what is called a grammar of graphics (this is expanded upon in depth in the *Visualizing Trends and Relationships* chapter) which requires us to sequentially build our graphs by first defining what data and variables will be visualized and then adding layers to the plot.

The next step we need to take is to define which columns we want to visualize. These are called the *aesthetics* and they are defined using the `aes()` function which is supplied to the `mapping` argument. The purpose of `aes()` is to tell ggplot which columns are mapped to what. The most important and fundamental of these are the `x` and `y` arguments. These refer to the x and y axes in the chart that we will begin to make.

Before we begin to analyze the relationship between `med_house_income` and `bach` (bachelor's degree attainment rate), we ought to do our due diligence of looking at the distribution of each of these first. Let us start with the `med_house_income` column. When exploring only a single variable, we want to supply that to the `x` argument of `aes()`.

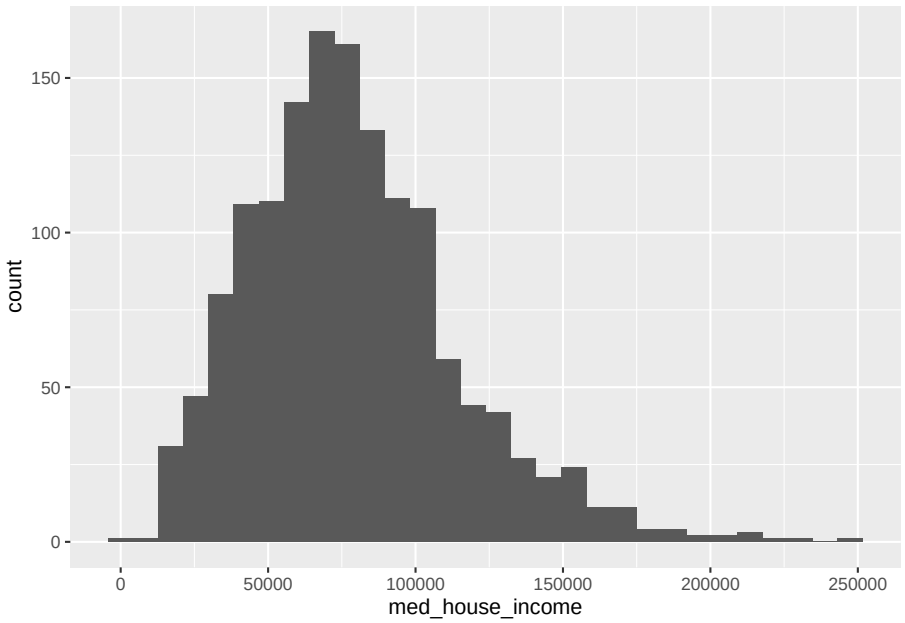
```
ggplot(acs_edu, aes(x = med_house_income))
```



Alright, we are making progress. We can see that the x axis is now filled out a bit more. The axis breaks have been labeled as has the axis itself. In order to see the data in a graphical representation, we need to determine how we want to see the data and what sort of geometry will be used to visualize it.

To **add** geometry to our ggplot, we use the plus sign `+` which signifies that we are adding a layer on top of the basic graph. There are many ways we can visualize univariate data but the histogram has stood the test of time. To create a histogram we **add** the `geom_histogram()` layer to our existing ggplot code.

```
ggplot(acs_edu, aes(x = med_house_income)) +
  geom_histogram()
```



> Note: To ensure that our code is legible we add each new layer on a line. R will manage the indentation for you. Code readibility is very important and you will thank yourself later for instilling good practices from the start.

This histogram illustrates the distribution of median household income in the state of Massachusetts. The median value seems to sit somewhere around $75k with a few outliers near $250k as well demonstrating a right skew.

> Reminder: The skew is where there are few [observations].

Usually when we look at distributions of wealth they are extremely right skewed meaning there are a few people who make an outrageous amount of money. What is interesting is that this histogram is rather normally distributed almost challenging intuition. This is because the ACS does something called top-coding. Top-coding is the practice of creating a ceiling value. For example, if there is a tract has a median household income of $1m, that will be reduced to the top-coded value—what appears to be $250k. This creates what are called censored data.

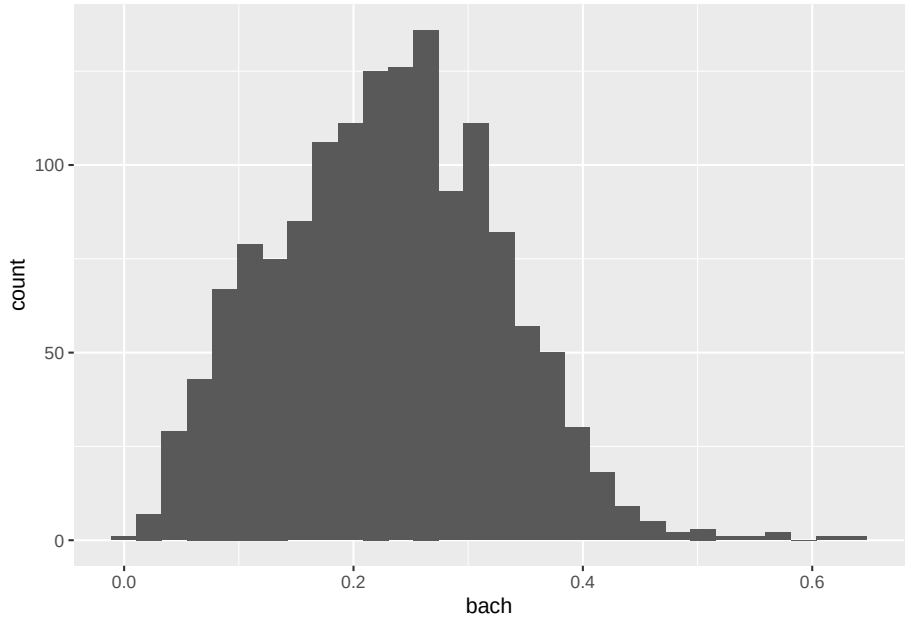
> **Censored data**: data "in which the value of a measurement or observation is only partially known." [10][11]

[10] https://en.wikipedia.org/wiki/Censoring_(statistics)

[11] https://stats.stackexchange.com/questions/49443/how-to-model-this-odd-shaped-distribution-almost-a-reverse-j

Let us now create a histogram of our second variable of interest, `bach`.

```
ggplot(acs_edu, aes(x = bach)) +
  geom_histogram()
```



This histogram illustrates the distribution of the bachelor degree attainment rate (the proportion of people with a bachelor's degree) across census tracts in Massachusetts. Because we did our homework ahead of time, we know that the national attainment rate in 2018 for people over 25 was ~35%[12]. Our histogram shows that within MA there is a lot of variation in the attainment rate from a low of about 0% to a high of over 60%. There is not a steep peak in the distribution which tells us that there is a fair amount of variation in the distribution.

Now that there is an intuition of the distribution and characteristics of both `med_house_income` and `bach`, we can begin to try and answer the question *what is the effect of education on median household income?* The phrasing of our question will determine how we visualize our data.

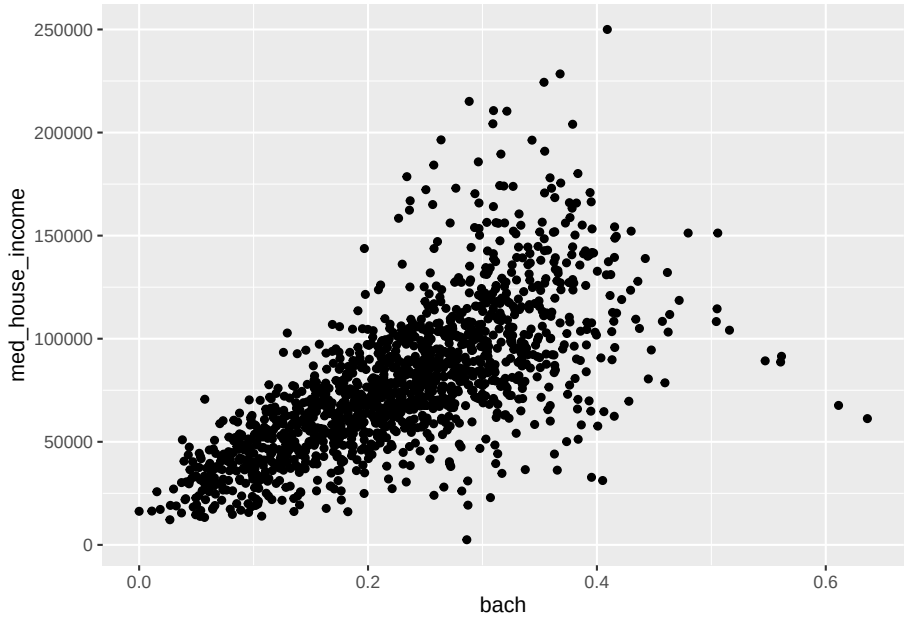
When stating research questions we often phrase it as *what is the effect of x* ***on*** *y*? In this formulation we are determining that `bach`, our independent variable, will be plotted on the x axis and `med_house_income` will be plotted on the y axis. To visualize this bivariate relationship we will create a scatter plot.

[12] https://www.census.gov/data/tables/2018/demo/education-attainment/cps-detailed-tables.html

> This structure and phrasing is useful for continuity in verbal communication, graphical representation, and hypothesis testing.

We can visualize this relationship by adding additional mapped aesthetics. In this case, we will map both the x and y arguments of the `aes()` function. Rather than adding histogram layer, we will need to create a scatter plot. Scatter plots are created by plotting points for each (x, y) pair. To get such an effect we will use the `geom_point()` layer.

```
ggplot(acs_edu, aes(x = bach, y = med_house_income)) +
  geom_point()
```

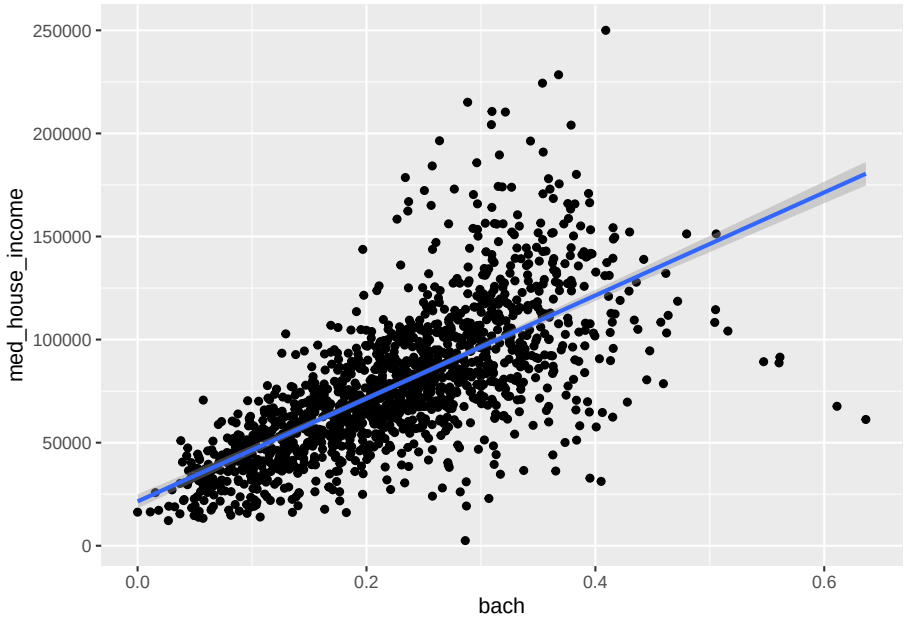


The above scatter plot provides a lot of information. We see that there is a positive linear trend—that is that when the `bach` value increases so does the `med_house_income` variable. When looking at a scatter plot we are looking to see if there is a consistent pattern that can be sussed out.

In this scatterplot we can see that there is a linear pattern. When the points on the scatter plot are closer to eachother and demonstrate less spread, that means there is a stronger relationship between the two variables. Imagine if we drew a line through the middle of the points, we would want each point to be as close to that line as possible. The further the point is away from that line, the more variation there is. In these cases we often create linear regression models to estimate the relationship.

Using ggplot, we can plot the estimated linear regression line on top of our scatterplot. This is done with a `geom_smooth()` layer. By default, `geom_smooth()` does not plot the linear relationship. To do that, we need to specify what kind of smoothing we would like. To plot the estimated linear model, we set `method = "lm"`.

```
ggplot(acs_edu, aes(x = bach, y = med_house_income)) +
  geom_point() +
  geom_smooth(method = "lm")
#> `geom_smooth()` using formula 'y ~ x'
```

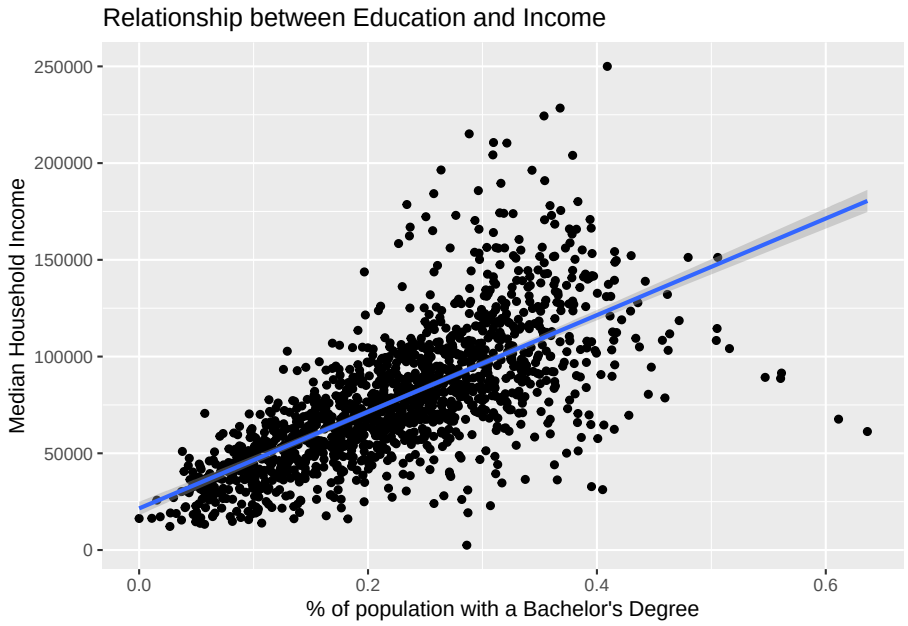


Wonderful! To finish up this graphic, we should add informative labels. Labels live in their own layer which is created with `labs()`. Each argument maps to an aesthetic—e.g. `x` and `y`. By default ggplot uses the column names for axis labels, but these labels are usually uninformative.

Let's give the plot a title and better labels for its axes. We will set the following arguments to `labs()`

- `x = "% of population with a Bachelor's Degree"`
- `y = "Median Household Income"`
- `title = "Relationship between Education and Income"`

```
ggplot(acs_edu, aes(x = bach, y = med_house_income)) +
  geom_point() +
  geom_smooth(method = "lm") +
  labs(x = "% of population with a Bachelor's Degree",
       y = "Median Household Income",
       title = "Relationship between Education and Income")
#> `geom_smooth()` using formula 'y ~ x'
```



Note that each argument is placed on a new line. Again, this is to improve readability.

What can we determine from this graph? Take a few minutes and write down what you see and what you can infer from that.

Consider asking these questions:

- Is there a relationship between our variables?
- Is the relationship linear?
- Which direction does the trend go?
- How spread out are the value pairs?
- Are there any outliers?

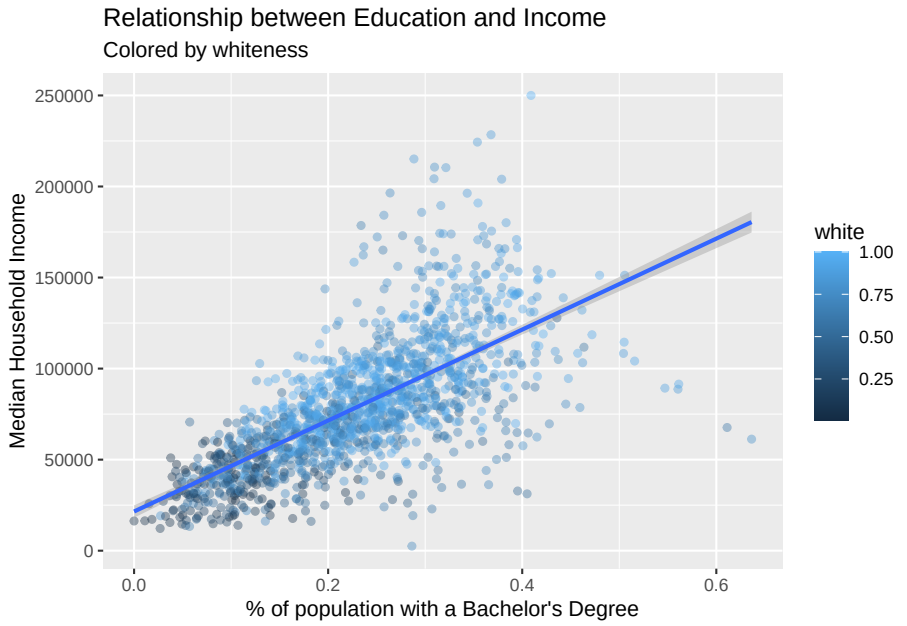
This chart ilicits further lines of inquiry. For example, in the sociological literature there is a well documented achievement gap. The achievement gap can be

traced along racial lines—though it is not inherently *caused* by race but rather intertwined with it. Can this be seen in the tracts of Massachusetts?

We can visualize a third variable on our chart by mapping another **aes**thetic. When we add a third variable to the visualization we are generally trying to illustrate group membership or size / magnitude of the third variable. Due to the large number of points on our chart already, we may benefit more from mapping color rather than size—imagine 1,000 points overlapping even more than they already do.

We can map the proportion of the population that is white to the color of our points. We do this by setting the `color` aesthetic to `white`. While we're at it, let us include a subtitle which is informative to the viewer.

```
ggplot(acs_edu, aes(x = bach, y = med_house_income, color = white)) +
  geom_point(alpha = .4) +
  geom_smooth(method = "lm") +
  labs(x = "% of population with a Bachelor's Degree",
       y = "Median Household Income",
       title = "Relationship between Education and Income",
       subtitle = "Colored by whiteness")
#> `geom_smooth()` using formula 'y ~ x'
```



What can we conclude now? Does the addition of the third variable increase or decrease the utility of our scatter plot? Does the trend seem to mediated by race? I'll leave those questions to you to answer.

You've now completed your first visual analysis. You've learned how to create publication ready histograms and scatter plots using ggplot2. This is no small feat!

This chapter provided you with data that was used in our visualization exercises. You're going to want to be able to visualize and analyze your own data. The next chapter introduces you reading data and some of the most common file formats you may encounter.

Chapter 10

General data manipulation

We spent the last chapter performing our first exploratory visual analysis. From our visualizations we were able to inductively conlcude that as both median household income and the proportion of the population with a bachelors degree increases, so does the share of the population is white.

While we were able to make wonderful visualizations, we did skip a number of steps in the exploratory analysis process! Arguably the most important is that we skipped the curating of our dataset. The ACS dataset was already cleaned and curated for you. This almost always will not be the case. As such, we're going to spend this chapter learning about ways of selecting subsets of our data.

Now that you have the ability to read in data, it is important that you get comfortable handling it. Some people call the process of rearranging, cleaning, and reshaping data massaging, plumbing, engineering, and myriad other names. Here, we will refer to this as data manipulation. This is a preferable catch-all term that does not illicit images of toilets or Phoebe Buffay (she was a masseuse!).

You may have heard of the 80/20 rule, or at least one of the many 80/20 rules. The 80/20 rule I'm referring to, is the idea that data scientists will spend 80% or more of their time cleaning and manipulating their data. The other 20% is the analysis part—creating statistics and models. I mention this because working with data is *mostly* data manipulation and only *some* statistics. Be prepared to get your hands dirty with data—euphemisms like this is probably why the phrase *data plumbing* ever came about.

In this chapter we will learn how to preview data, select columns, arrange rows, and filter rows. This is where we get hands on with our data. This is just about as physical as we will be able to get unless you want to open up the raw csv and edit the data there!

> Note: I do not recommend editing any csvs directly. That statement was in jest.

This is all to say that you will find yourself with messy, unsanitized, gross, not fun to look at data most of the time. Because of this, it is really important that we have the skills to clean our data. Right now we're going to go over the foundational skills we will learn how to select columns and rows, filter data, and create new columns, and arrange our data. To do this, we will be using the `dplyr` package from the tidyverse.

The data we read in in the last chapter was only a select few variables from the annual census data release that the team at the Boston Area Research Initiative (BARI) provides. These census indicators are used to provide a picture into the changing landscape of Boston and Massachusetts more generally. In this chapter we will work through a rather real life scenario that you may very well encounter using the BARI data.

10.1 Scenario

A local non-profit is interested in the commuting behavior of Greater Boston residents. Your adviser suggested that you assist the non-profit in their work. You've just had coffee with the project manager to learn more about what their specific research question is. It seems that the extension of the Green Line is of great interest to them. She spoke at length about the history of the Big Dig and its impact on commuters working in the city. This poured over into a conversation about the spatial and social stratification of a city. She looks at her watch and realizes she's about to miss the commuter train home. You shake their hand, thank them for their time (and the free coffee because you're a grad student), and affirm that you will email them in a week or so with some data for them to work with.

You're back at your apartment, french press next your laptop—though not too close—notes open, and ready to begin. You pore over your notes and realize while you now have a rather good understanding of *what* the Green Line Extensions is and the impact that the Big Dig had, you really have no idea what about commuting behavior in Greater Boston they are interested in. You realize you did not even confirm what constitutes the Greater Boston area. You push down the coffee grinds and pour your first cup of coffee. This will take at least two cups of coffee.

The above scenario sounds like something out of a stress dream. This is scenario that I have found myself in many times and I am sure that you will find yourself in at one point as well. The more comfortable you get with data analysis and asking good questions, the more guided and directed you can make these seemingly vague objectives.

> At the end of this chapter, we will expound upon ways to prevent this in the future.

10.2 Getting physical with the data

The data we used in both chapters one and two were curated from the annual census indicator release from BARI[^indicators]. This is the dataset from which `acs_edu` was created. We will use these data to provide relevant data relating to commuting in the Greater Boston Area. The first thing you'll notice is that these data are large and somewhat unforgiving to work with. What a better way to get started than with big data?

We will be using the tidyverse to read in and manipulate our data (as we did last chapter). Recall that we will load the tidyverse using `library(tidyverse)`.

> Refresher: the tidyverse is a collection of packages used for data analysis. When you load the tidyverse it loads `readr`, `ggplot2`, and `dplyr` for us, among other packages. For now, though, these are the only relevant packages.

Try it:

- Load the tidyverse
- Read in the file `ACS_1317_TRACT.csv` located in the `data` directory, store it in an object called `acs_raw`

```
library(tidyverse)

acs_raw <- read_csv("data/ACS_1317_TRACT.csv")
```

Wonderful! You've got the hang of reading data in which is truly no small feat. Once we have the data accessible from R, it is important to get familiar with what the data are. This means we need to know *which* variables are available to us and get a feel for what the values in those variables represent.

Try printing out the `acs_raw` object in your console.

```
acs_raw
```

Oof, yikes. It's a bit messy! Not to mention that R did not even print out all of the columns. That's because it ran out of room. When we're working with wide data (many columns), it's generally best to view only a preview of the data. The function `dplyr::glimpse()` can help us do just that. Provide `acs_raw` as the only argument to `glimpse()`.

```
glimpse(acs_raw)
```

Much better, right? It is frankly still a lot of text, but the way it is presented is rather useful. Each variable is written followed by its data type, i.e. `<dbl>`, and then a preview of values in that column. If the `<dbl>` does not make sense yet, do not worry. We will go over data types in depth later. Data types are not the most fun and I think it is important we have fun!

`acs_raw` is the dataset from which `acs_edu` was created. As you can see, there are many, many, *many* different variables that the ACS data provide us with. These are only the tip of the iceberg. **Pause**

Now have a think.

Looking at the preview of these data, which columns do you think will be most useful to the non-profit for understanding commuter behavior?

> Note: "all of them" is not always the best answer. By providing too much data one may be moved to inaction because they now must determine what variables are the most useful and how to use them.

If you spotted the columns commute_less10, commute1030, commute3060, commute6090, and commute_over90, your eyes and intuition have served you well! These variables tell us about what proportion of the sampled population in a given census tract have commute times that fall within the indicated duration range, i.e. 30-60.

10.2.1 `select()`ing

So now we have an intuition of the most important variables, but the next problem soon arises: how do we isolate just these variables? Whenever you find yourself needing to select or deselect columns from a tibble `dplyr::select()` will be the main function that you will go to.

What does it do?: `select()` selects variables from a tibble and returns another tibble.

Before we work through how to use `select()`, refer to the help documentation and see if you can get somewhat of an intuition by typing `?select()` into the console. Once you press enter the documentation page should pop up in RStudio.

There are a few reasons why I am directing you towards the function documentation.

1. To get you comfortable with navigating the RStudio IDE

2. Expose you to the R vocabulary
3. Soon you'll be too advanced for this book and will have to figure out the way functions work on your own!

Perhaps the help documentation was a little overwhelming and absolutely confusing. That's okay. It's just an exposure! With each exposure things will make more sense. Let's tackle these arguments one by one.

> `.data`: A tbl. All main verbs are S3 generics and provide methods for tbl_df(), dtplyr::tbl_dt() and dbplyr::tbl_dbi().

What I want you to take away from this argument definition is `a tbl`. Whenever you read `tbl` think to your self "oh, that is just a `tibble`." If you recall, when we read rectangular data with `readr::read_csv()` or any other `readr::read_*()` function we will end up with a tibble. To verify that this is the case, we can double check our objects using the function `is.tbl()`. This function takes an object and returns a logical value (`TRUE` or `FALSE`) if the statement is true. Let's double check that `acs_raw` is in fact a tbl.

```
is.tbl(acs_raw)
#> [1] TRUE
```

> Aside: Each object type usually has a function that let's you test if objects are that type that follow the structure `is.obj_type()` or the occasional `is_obj_type()`. We will go over object types more later.

We can read the above as if we are asking R the question "is this object a `tbl`?" The resultant output of `is.tbl(acs_raw)` is `TRUE`. Now we can be doubly confident that this object can be used with `select()`.

The second argument to `select()` is a little bit more difficult to grasp, so don't feel discouraged if this isn't clicking right away. There is *a lot* written in this argument definition and I feel that not all of it is necessary to understand from the get go.

> `...`: One or more unquoted expressions separated by commas. You can treat variable names like they are positions, so you can use expressions like x:y to select ranges of variables.

`...`, referred to as "dots" means that we can pass any number of arguments to the function. Translating "one or more unquoted expressions separated by commas" into regular person speak reiterates that there can be multiple other

arguments passed into `select()`. "Unquoted expressions" means that if we want to select a column we do not put that column name in quotes.

"You can treat variable names like they are positions" translates to "if you want the first column you can write the number `1` etc." and because of this, if you want the first through tenth variable you can pass `1:10` as an argument to `...`.

The most important thing about `...` is that we **do not** assign `...` as an argument, for example `. ... = column_a` is **not** the correct notation. We provide `column_a` alone.

As always, this makes more sense once we see it in practice. We will now go over the many ways in which we can select columns using `select()`. Once we have gotten the hang of selecting columns we will return back to assisting our non-profit.

We will go over:

- selecting by name
- selecting by position
- select helpers

10.2.2 `select()`ing exercises

`select()` enables us to choose columns from a tibble based on their names. But remember that these will be unquoted column names.

Try it:

- select the column `name` from `acs_raw`

```
select(acs_raw, name)
#> # A tibble: 1,478 x 1
#>    name
#>    <chr>
#>  1 Census Tract 7281, Worcester County, Massachusetts
#>  2 Census Tract 7292, Worcester County, Massachusetts
#>  3 Census Tract 7307, Worcester County, Massachusetts
#>  4 Census Tract 7442, Worcester County, Massachusetts
#>  5 Census Tract 7097.01, Worcester County, Massachusetts
#>  6 Census Tract 7351, Worcester County, Massachusetts
#>  7 Census Tract 7543, Worcester County, Massachusetts
#>  8 Census Tract 7308.02, Worcester County, Massachusetts
#>  9 Census Tract 7171, Worcester County, Massachusetts
#> 10 Census Tract 7326, Worcester County, Massachusetts
#> # ... with 1,468 more rows
```

The column `name` was passed to `....` Recall that dots allows us to pass "one ore more unquoted expressions separated by commas." To test this statement out, select `town` in addition to `name` from `acs_raw`

Try it:

- select `name` and `town` from `acs_raw`

```
select(acs_raw, name, town)
#> # A tibble: 1,478 x 2
#>    name                                                  town
#>    <chr>                                                 <chr>
#>  1 Census Tract 7281, Worcester County, Massachusetts    HOLDEN
#>  2 Census Tract 7292, Worcester County, Massachusetts    WEST BOYLSTON
#>  3 Census Tract 7307, Worcester County, Massachusetts    WORCESTER
#>  4 Census Tract 7442, Worcester County, Massachusetts    MILFORD
#>  5 Census Tract 7097.01, Worcester County, Massachusetts LEOMINSTER
#>  6 Census Tract 7351, Worcester County, Massachusetts    LEICESTER
#>  7 Census Tract 7543, Worcester County, Massachusetts    WEBSTER
#>  8 Census Tract 7308.02, Worcester County, Massachusetts WORCESTER
#>  9 Census Tract 7171, Worcester County, Massachusetts    BERLIN
#> 10 Census Tract 7326, Worcester County, Massachusetts    WORCESTER
#> # ... with 1,468 more rows
```

Great, you're getting the hang of it.

Now, in addition to to selecting columns solely based on their names, we can also select a range of columns using the format `col_from:col_to`. In writing this `select()` will register that you want every column from and including `col_from` up until and including `col_to`.

Let's refresh ourselves with what our data look like:

```
glimpse(acs_raw)
```

Try it:

- select the columns `age_u18` through `age_o65`.

```
select(acs_raw, age_u18:age_o65)
#> # A tibble: 1,478 x 4
#>    age_u18 age1834 age3564 age_o65
#>      <dbl>   <dbl>   <dbl>   <dbl>
#>  1   0.234   0.202   0.398   0.166
#>  2   0.181   0.151   0.461   0.207
```

```
#>  3   0.171   0.214   0.437   0.178
#>  4   0.203   0.227   0.436   0.133
#>  5   0.177   0.203   0.430   0.190
#>  6   0.163   0.237   0.439   0.162
#>  7   0.191   0.326   0.380   0.102
#>  8   0.202   0.183   0.466   0.148
#>  9   0.188   0.150   0.462   0.200
#> 10   0.244   0.286   0.342   0.128
#> # ... with 1,468 more rows
```

Now to really throw you off! You can even reverse the order of these ranges.

Try it:

- select columns from `age_o65` to `age_u18`.

```
select(acs_raw, age_o65:age_u18)
#> # A tibble: 1,478 x 4
#>    age_o65 age3564 age1834 age_u18
#>      <dbl>   <dbl>   <dbl>   <dbl>
#>  1   0.166   0.398   0.202   0.234
#>  2   0.207   0.461   0.151   0.181
#>  3   0.178   0.437   0.214   0.171
#>  4   0.133   0.436   0.227   0.203
#>  5   0.190   0.430   0.203   0.177
#>  6   0.162   0.439   0.237   0.163
#>  7   0.102   0.380   0.326   0.191
#>  8   0.148   0.466   0.183   0.202
#>  9   0.200   0.462   0.150   0.188
#> 10   0.128   0.342   0.286   0.244
#> # ... with 1,468 more rows
```

Alright, so now we have gotten the hang of selecting columns based on their names. But equally important is the ability to select columns based on their position. Consider the situation in which you regularly receive georeferenced data from a research partner and the structure of the dataset is rather consistent *except* that they frequently change the name of the coordinate columns. Sometimes the columns are `x` and `y`. Sometimes they are capitalized `X` and `Y`, `lon` and `lat`, or even `long` and `lat`. It eats you up inside! But you know that while the names may change, their positiions never do—they're always the last two columns. You decide to program a solution rather than having a conversation with your research partner—though, I recommend you both level set on reproducibility standards.

> "…You can treat variable names like they are positions…"

The above was taken from the argument definition of dots `...`. Like providing the name of the column, we can also provide their positions (also referred to as an index). In our previous example, we selected the `name` column. We can select this column by it's position too. `name` is the second column in our tibble. We select it by position like so:

```
select(acs_raw, 2)
#> # A tibble: 1,478 x 1
#>    name
#>    <chr>
#>  1 Census Tract 7281, Worcester County, Massachusetts
#>  2 Census Tract 7292, Worcester County, Massachusetts
#>  3 Census Tract 7307, Worcester County, Massachusetts
#>  4 Census Tract 7442, Worcester County, Massachusetts
#>  5 Census Tract 7097.01, Worcester County, Massachusetts
#>  6 Census Tract 7351, Worcester County, Massachusetts
#>  7 Census Tract 7543, Worcester County, Massachusetts
#>  8 Census Tract 7308.02, Worcester County, Massachusetts
#>  9 Census Tract 7171, Worcester County, Massachusetts
#> 10 Census Tract 7326, Worcester County, Massachusetts
#> # ... with 1,468 more rows
```

Try it:

- select `age_u18` and `age_o65` by their position

```
select(acs_raw, 6, 9)
#> # A tibble: 1,478 x 2
#>    age_u18 age_o65
#>      <dbl>   <dbl>
#>  1   0.234   0.166
#>  2   0.181   0.207
#>  3   0.171   0.178
#>  4   0.203   0.133
#>  5   0.177   0.190
#>  6   0.163   0.162
#>  7   0.191   0.102
#>  8   0.202   0.148
#>  9   0.188   0.200
#> 10   0.244   0.128
#> # ... with 1,468 more rows
```

You may see where I am going with this. Just like column names, we can select a range of columns using the same method `index_from:index_to`.

Try it:

- select the columns from `age_u18` to `age_o65` using `:` and the column position
- select the columns in reverse order by their indexes

```
select(acs_raw, 6:9)
#> # A tibble: 1,478 x 4
#>    age_u18 age1834 age3564 age_o65
#>      <dbl>   <dbl>   <dbl>   <dbl>
#>  1   0.234   0.202   0.398   0.166
#>  2   0.181   0.151   0.461   0.207
#>  3   0.171   0.214   0.437   0.178
#>  4   0.203   0.227   0.436   0.133
#>  5   0.177   0.203   0.430   0.190
#>  6   0.163   0.237   0.439   0.162
#>  7   0.191   0.326   0.380   0.102
#>  8   0.202   0.183   0.466   0.148
#>  9   0.188   0.150   0.462   0.200
#> 10   0.244   0.286   0.342   0.128
#> # ... with 1,468 more rows
```

```
select(acs_raw, 9:6)
#> # A tibble: 1,478 x 4
#>    age_o65 age3564 age1834 age_u18
#>      <dbl>   <dbl>   <dbl>   <dbl>
#>  1   0.166   0.398   0.202   0.234
#>  2   0.207   0.461   0.151   0.181
#>  3   0.178   0.437   0.214   0.171
#>  4   0.133   0.436   0.227   0.203
#>  5   0.190   0.430   0.203   0.177
#>  6   0.162   0.439   0.237   0.163
#>  7   0.102   0.380   0.326   0.191
#>  8   0.148   0.466   0.183   0.202
#>  9   0.200   0.462   0.150   0.188
#> 10   0.128   0.342   0.286   0.244
#> # ... with 1,468 more rows
```

> Base R Side Bar: To help build your intuition, I want to point out some base R functionality. Using the colon `:` with integers (whole numbers) is actually not a `select()` specific functionality. This is something that is rather handy and built directly into R. Using the colon operator, we can create ranges of numbers in the same exact way as we did above. If we want create the range of numbers from 1 to 10, we write `1:10`. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

In our scenario, we want to select the last two columns. We may not know their names *or* their position. Luckily, there's a function for that.

`last_col()` is a handy function that enables us to select the last column. There is also an option to get an offset from the last column. An offset would allow us to grab the second to last column by setting the offset to 1. By setting the offset, `last_col()` will from the `offset + 1` from the last column. So if the offset is set to 1, we would be grabbing the second to last column.

Let's give it a shot:

```
select(acs_raw, last_col())
#> # A tibble: 1,478 x 1
#>    m_atown
#>    <chr>
#>  1 HOLDEN
#>  2 WEST BOYLSTON
#>  3 WORCESTER
#>  4 MILFORD
#>  5 LEOMINSTER
#>  6 LEICESTER
#>  7 WEBSTER
#>  8 WORCESTER
#>  9 BERLIN
#> 10 WORCESTER
#> # ... with 1,468 more rows
select(acs_raw, last_col(offset = 1))
#> # A tibble: 1,478 x 1
#>    area_acr_1
#>         <dbl>
#>  1     23242.
#>  2      8868.
#>  3     24610.
#>  4      9616.
#>  5     18993.
#>  6     15763.
#>  7      9347.
#>  8     24610.
#>  9      8431.
#> 10     24610.
#> # ... with 1,468 more rows
select(acs_raw, last_col(offset = 1):last_col())
#> # A tibble: 1,478 x 2
#>    area_acr_1 m_atown
#>         <dbl> <chr>
#>  1     23242. HOLDEN
#>  2      8868. WEST BOYLSTON
#>  3     24610. WORCESTER
#>  4      9616. MILFORD
```

```
#>  5     18993. LEOMINSTER
#>  6     15763. LEICESTER
#>  7      9347. WEBSTER
#>  8     24610. WORCESTER
#>  9      8431. BERLIN
#> 10     24610. WORCESTER
#> # ... with 1,468 more rows
```

`last_col()` comes from another packages called `tidyselect` which is imported with `dplyr`. This package contains a number of helper functions. There are 9 total helpers and you've already learned one of them. We will briefly review four more of these. I'm sure you are able to deduce how the functions work solely based on their names. The functions are:

- `starts_with()`: a string to search that columns start with
- `ends_with()`: a string to search that columns end with
- `contains()`: a string to search for in the column names at any position
- `everything()`: selects the remaining columns

Each of these function take a character string and searches the column headers for them.

Try it out:

- find all columns that start with `"med"`

```
select(acs_raw, starts_with("med"))
#> # A tibble: 1,478 x 7
#>    med_house_income med_gross_rent med_home_val med_yr_built_raw med_yr_built
#>               <dbl>          <dbl>        <dbl>             <dbl> <chr>
#>  1           105735           1640       349000              1988 1980 to 1989
#>  2            69625            894       230200              1955 1950 to 1959
#>  3            70679           1454       207200              1959 1950 to 1959
#>  4            74528            954       268400              1973 1970 to 1979
#>  5            52885           1018       223200              1964 1960 to 1969
#>  6            64100            867       232700              1966 1960 to 1969
#>  7            37093            910       170900              1939 Prior to 19~
#>  8            87750           1088       270100              1939 Prior to 19~
#>  9            97417           1037       379600              1981 1980 to 1989
#> 10            43384           1017       156500              1939 Prior to 19~
#> # ... with 1,468 more rows, and 2 more variables: med_yr_moved_inraw <dbl>,
#> #   med_yr_rent_moved_in <dbl>
```

- select columns that end with `"per"`

```
select(acs_raw, ends_with("per"))
#> # A tibble: 1,478 x 8
#>    fam_pov_per fam_house_per fem_head_per same_sex_coup_p~ grand_head_per
#>          <dbl>         <dbl>        <dbl>            <dbl>          <dbl>
#>  1      0.0475         0.797       0.0899            0            0
#>  2      0.0652         0.698       0.120             0            0.00583
#>  3      0.0584         0.659       0.114             0            0
#>  4      0.0249         0.657       0.121             0            0
#>  5      0.198          0.531       0.158             0            0.00946
#>  6      0.0428         0.665       0.0603            0            0.0353
#>  7      0.0762         0.632       0.227             0            0.00643
#>  8      0.101          0.636       0.0582            0.297        0.0260
#>  9      0.0149         0.758       0.0721            0            0.00434
#> 10      0.0954         0.460       0.225             0            0.0279
#> # ... with 1,468 more rows, and 3 more variables: vacant_unit_per <dbl>,
#> #   renters_per <dbl>, home_own_per <dbl>
```

- find any column that contains the string `"yr"`

```
select(acs_raw, contains("yr"))
#> # A tibble: 1,478 x 4
#>    med_yr_built_raw med_yr_built  med_yr_moved_inraw med_yr_rent_moved_in
#>               <dbl> <chr>                      <dbl>                <dbl>
#>  1             1988 1980 to 1989                2004                 2012
#>  2             1955 1950 to 1959                2003                 2010
#>  3             1959 1950 to 1959                2007                 2012
#>  4             1973 1970 to 1979                2006                 2011
#>  5             1964 1960 to 1969                2006                 2011
#>  6             1966 1960 to 1969                2000                 2009
#>  7             1939 Prior to 1940               2011                 2012
#>  8             1939 Prior to 1940               2006                 2012
#>  9             1981 1980 to 1989                2004                 2012
#> 10             1939 Prior to 1940               2011                   NA
#> # ... with 1,468 more rows
```

- select columns that start with `med` then select everything else

```
select(acs_raw, contains("yr"), everything())
#> # A tibble: 1,478 x 59
#>    med_yr_built_raw med_yr_built med_yr_moved_in~ med_yr_rent_mov~ ct_id_10
#>               <dbl> <chr>                  <dbl>            <dbl>    <dbl>
#>  1             1988 1980 to 1989            2004             2012  2.50e10
#>  2             1955 1950 to 1959            2003             2010  2.50e10
#>  3             1959 1950 to 1959            2007             2012  2.50e10
```

```
#>  4             1973 1970 to 1979            2006             2011  2.50e10
#>  5             1964 1960 to 1969            2006             2011  2.50e10
#>  6             1966 1960 to 1969            2000             2009  2.50e10
#>  7             1939 Prior to 19~            2011             2012  2.50e10
#>  8             1939 Prior to 19~            2006             2012  2.50e10
#>  9             1981 1980 to 1989            2004             2012  2.50e10
#> 10             1939 Prior to 19~            2011               NA  2.50e10
#> # ... with 1,468 more rows, and 54 more variables: name <chr>, total_pop <dbl>,
#> #   pop_den <dbl>, sex_ratio <dbl>, age_u18 <dbl>, age1834 <dbl>,
#> #   age3564 <dbl>, age_o65 <dbl>, for_born <dbl>, white <dbl>, black <dbl>,
#> #   asian <dbl>, hispanic <dbl>, two_or_more <dbl>, eth_het <dbl>,
#> #   med_house_income <dbl>, pub_assist <dbl>, gini <dbl>, fam_pov_per <dbl>,
#> #   unemp_rate <dbl>, total_house_h <dbl>, fam_house_per <dbl>,
#> #   fem_head_per <dbl>, same_sex_coup_per <dbl>, grand_head_per <dbl>,
#> #   less_than_hs <dbl>, hs_grad <dbl>, some_coll <dbl>, bach <dbl>,
#> #   master <dbl>, prof <dbl>, doc <dbl>, commute_less10 <dbl>,
#> #   commute1030 <dbl>, commute3060 <dbl>, commute6090 <dbl>,
#> #   commute_over90 <dbl>, by_auto <dbl>, by_pub_trans <dbl>, by_bike <dbl>,
#> #   by_walk <dbl>, total_house_units <dbl>, vacant_unit_per <dbl>,
#> #   renters_per <dbl>, home_own_per <dbl>, med_gross_rent <dbl>,
#> #   med_home_val <dbl>, area_acres <dbl>, town_id <dbl>, town <chr>,
#> #   fips_stco <dbl>, county <chr>, area_acr_1 <dbl>, m_atown <chr>
```

10.3 Selecting Rows

Though a somewhat infrequent event, it will be handy to know how to select rows. There are two ways in which we can select our rows. The first is by specifying exactly which rows by their position. The other way is to filter down our data based on a condition—i.e. median household income within a range. The functions to do this are `slice()` and `filter()` respectively. The remainder of this chapter will introduce you to `slice()`. We will learn how to filter in the next chapter.

Like `select()` we can also select rows. But rows do not have names, so we must select the rows based on their position. Given your familiarity with selecting by column position this should be a cake walk for you.

Similar to `last_col()` we have the function `n()`. `n()` is a rather handy little function which tells us how many observations there are in a tibble. This allows to specify the last row of a tibble.

```
slice(acs_raw, n())
#> # A tibble: 1 x 59
#>   ct_id_10 name  total_pop pop_den sex_ratio age_u18 age1834 age3564 age_o65
```

```
#>      <dbl> <chr>     <dbl>   <dbl>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1  2.50e10 Cens~      5821   2760.     0.885   0.181   0.204   0.435   0.180
#> # ... with 50 more variables: for_born <dbl>, white <dbl>, black <dbl>,
#> #   asian <dbl>, hispanic <dbl>, two_or_more <dbl>, eth_het <dbl>,
#> #   med_house_income <dbl>, pub_assist <dbl>, gini <dbl>, fam_pov_per <dbl>,
#> #   unemp_rate <dbl>, total_house_h <dbl>, fam_house_per <dbl>,
#> #   fem_head_per <dbl>, same_sex_coup_per <dbl>, grand_head_per <dbl>,
#> #   less_than_hs <dbl>, hs_grad <dbl>, some_coll <dbl>, bach <dbl>,
#> #   master <dbl>, prof <dbl>, doc <dbl>, commute_less10 <dbl>,
#> #   commute1030 <dbl>, commute3060 <dbl>, commute6090 <dbl>,
#> #   commute_over90 <dbl>, by_auto <dbl>, by_pub_trans <dbl>, by_bike <dbl>,
#> #   by_walk <dbl>, total_house_units <dbl>, vacant_unit_per <dbl>,
#> #   renters_per <dbl>, home_own_per <dbl>, med_gross_rent <dbl>,
#> #   med_home_val <dbl>, med_yr_built_raw <dbl>, med_yr_built <chr>,
#> #   med_yr_moved_inraw <dbl>, med_yr_rent_moved_in <dbl>, area_acres <dbl>,
#> #   town_id <dbl>, town <chr>, fips_stco <dbl>, county <chr>, area_acr_1 <dbl>,
#> #   m_atown <chr>
```

Unlike `last_col()`, `n()` provides us with a number. Instead of specifying an offset we can instead subtract directly from the output of `n()`. To grab the last three rows we can write `(n() - 3):n()`. We put `n()-3` inside of parantheses so R knows to process `n() - 3` first.

```
slice(acs_raw, (n() - 3):n())
#> # A tibble: 4 x 59
#>   ct_id_10 name  total_pop pop_den sex_ratio age_u18 age1834 age3564 age_o65
#>      <dbl> <chr>     <dbl>   <dbl>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1  2.50e10 Cens~      2519   3083.     0.806   0.202   0.268   0.397   0.132
#> 2  2.50e10 Cens~      3500   5392.     1.05    0.205   0.277   0.395   0.122
#> 3  2.50e10 Cens~      5816   2677.     1.20    0.191   0.233   0.458   0.118
#> 4  2.50e10 Cens~      5821   2760.     0.885   0.181   0.204   0.435   0.180
#> # ... with 50 more variables: for_born <dbl>, white <dbl>, black <dbl>,
#> #   asian <dbl>, hispanic <dbl>, two_or_more <dbl>, eth_het <dbl>,
#> #   med_house_income <dbl>, pub_assist <dbl>, gini <dbl>, fam_pov_per <dbl>,
#> #   unemp_rate <dbl>, total_house_h <dbl>, fam_house_per <dbl>,
#> #   fem_head_per <dbl>, same_sex_coup_per <dbl>, grand_head_per <dbl>,
#> #   less_than_hs <dbl>, hs_grad <dbl>, some_coll <dbl>, bach <dbl>,
#> #   master <dbl>, prof <dbl>, doc <dbl>, commute_less10 <dbl>,
#> #   commute1030 <dbl>, commute3060 <dbl>, commute6090 <dbl>,
#> #   commute_over90 <dbl>, by_auto <dbl>, by_pub_trans <dbl>, by_bike <dbl>,
#> #   by_walk <dbl>, total_house_units <dbl>, vacant_unit_per <dbl>,
#> #   renters_per <dbl>, home_own_per <dbl>, med_gross_rent <dbl>,
#> #   med_home_val <dbl>, med_yr_built_raw <dbl>, med_yr_built <chr>,
#> #   med_yr_moved_inraw <dbl>, med_yr_rent_moved_in <dbl>, area_acres <dbl>,
#> #   town_id <dbl>, town <chr>, fips_stco <dbl>, county <chr>, area_acr_1 <dbl>,
```

```
#> #   m_atown <chr>
```

Try it:

- select the first row, rows 100-105, and the last row

```
slice(acs_raw, 1, 100:105, n())
#> # A tibble: 8 x 59
#>   ct_id_10 name  total_pop pop_den sex_ratio age_u18 age1834 age3564 age_o65
#>      <dbl> <chr>     <dbl>   <dbl>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1  2.50e10 Cens~      4585    333.      1.13   0.234   0.202   0.398   0.166
#> 2  2.50e10 Cens~      5223   2402.      1.21   0.183   0.171   0.450   0.197
#> 3  2.50e10 Cens~      5586    592.      1.09   0.278   0.116   0.413   0.193
#> 4  2.50e10 Cens~      4474   1119.      0.962  0.282   0.0847  0.427   0.206
#> 5  2.50e10 Cens~      6713    674.      0.928  0.223   0.216   0.423   0.139
#> 6  2.50e10 Cens~      6676   3541.      0.999  0.249   0.266   0.395   0.0902
#> 7  2.50e10 Cens~      8141    820.      1.25   0.258   0.169   0.410   0.164
#> 8  2.50e10 Cens~      5821   2760.      0.885  0.181   0.204   0.435   0.180
#> # ... with 50 more variables: for_born <dbl>, white <dbl>, black <dbl>,
#> #   asian <dbl>, hispanic <dbl>, two_or_more <dbl>, eth_het <dbl>,
#> #   med_house_income <dbl>, pub_assist <dbl>, gini <dbl>, fam_pov_per <dbl>,
#> #   unemp_rate <dbl>, total_house_h <dbl>, fam_house_per <dbl>,
#> #   fem_head_per <dbl>, same_sex_coup_per <dbl>, grand_head_per <dbl>,
#> #   less_than_hs <dbl>, hs_grad <dbl>, some_coll <dbl>, bach <dbl>,
#> #   master <dbl>, prof <dbl>, doc <dbl>, commute_less10 <dbl>,
#> #   commute1030 <dbl>, commute3060 <dbl>, commute6090 <dbl>,
#> #   commute_over90 <dbl>, by_auto <dbl>, by_pub_trans <dbl>, by_bike <dbl>,
#> #   by_walk <dbl>, total_house_units <dbl>, vacant_unit_per <dbl>,
#> #   renters_per <dbl>, home_own_per <dbl>, med_gross_rent <dbl>,
#> #   med_home_val <dbl>, med_yr_built_raw <dbl>, med_yr_built <chr>,
#> #   med_yr_moved_inraw <dbl>, med_yr_rent_moved_in <dbl>, area_acres <dbl>,
#> #   town_id <dbl>, town <chr>, fips_stco <dbl>, county <chr>, area_acr_1 <dbl>,
#> #   m_atown <chr>
```

10.4 Revisiting commmuting

We've just spent a fair amount of time learning how to work with our data. It's now time to return to the problem at hand. We still haven't addressed what data will be of use to our partner at the non-profit. While urban informatics is largely technical, it is still mostly intellectual. We have to think through problems and be methodical with our data selection and curation. We have to think about what our data tells us and why it is important.

During these exercises, I hope you were looking at the data and thinking about what may be helpful to the non-profit. Again, the goal is to provide them with what is useful, but not more than they need.

10.4.1 Exercise

It is now incumbent upon you to curate the data BARI Census Indicator dataset for the non-profit. Refamiliarize yourself with the data. Select a subset of columns that you believe will provide the best insight into commuting behavior in the Greater Boston Area while also providing demographic insight into the the area.

When making decisions like this, I like to think of a quote from The Master of Disguise:

> "Answer these questions for yourself: who? Why? Where? How?"

Save the resultant tibble to an object named `commute` or something else informative.

Below is one approach to this question. For this, I have selected all columns pertaining to commute time (columns that start with `commute`), the method by which people commute (begin with `by`), medisan household income, and the name of the census tract. The name of the census tract will be helpful for identifying "where".

```
commute <- select(acs_raw,
                  county,
                  hs_grad, bach, master,
                  starts_with("commute"),
                  starts_with("by"),
                  med_house_income)
```

[^indicators]

Chapter 11

That's *too much* data

Let us continue with the scenario developed in the last chapter. There is a non-profit who is seeking graduate student assistance to provide a curated dataset that provides insight into the commuting behavior of the Greater Boston Area. Using BARI's Massachussett's Census Indicators dataset, we were able to reduce the 52 initial columns down to 11. However these data are for the entire state not just the Greater Boston Area. This leaves us with two tasks: 1) definine the Greater Boston Area and 2) create a subset of our data that fit our criteria defined in 1.

Execute the below code to recreate to `commute` tibble. This will be the *second to last* time we write this chunk of code.

```
library(tidyverse)

acs_raw <- read_csv("data/ACS_1317_TRACT.csv")

commute <- select(acs_raw,
                  county,
                  hs_grad, bach, master,
                  starts_with("commute"),
                  starts_with("by"),
                  med_house_income)
```

11.1 `filter()`ing

Previously we looked at ways of selecting columns. Here, we will focus on creating subsets of our data. We will rely on the function `dplyr::filter()` for this. `filter()` differs from `slice()` in that filter will check to see if data fit a specified criteria whereas slice is only concerned with the position of a row.

> Explore the help documentation of `filter()` by running the command `?filter()` in your console.

The first argument of `filter()` is of course the tibble which you would like to subset. Secondly, we again see `....` In the case of `filter()`, we provide what are called logical expressions to `....` `filter()` then only returns the observations when that logical expression, or condition, is true.

Almost every time you purchase something online whether that is from Amazon or Etsy you are filtering your data using some logic. Whether that is checking the Prime tick box, specifying a price range on Etsy, or a restaurant rating on Yelp. These are all *conditions* that you are providing to your search.

We can create these types of filters on our own data and to do so, we will need to understand how logical expressions work. A logical expression is any expression that can be boiled down to true or false.

For example, using our `commute` dataset, we can check to see which Census Tracts have more than 75% of commuters traveling by auto.

```
auto_commuters <- filter(commute, by_auto > 0.75)

select(auto_commuters, by_auto)
```

```
## # A tibble: 1,012 x 1
##    by_auto
##      <dbl>
##  1   0.927
##  2   0.970
##  3   0.925
##  4   0.904
##  5   0.899
##  6   0.911
##  7   0.810
##  8   0.851
##  9   0.900
## 10   0.787
## # ... with 1,002 more rows
```

This above line checks every single value of `by_auto` and asks "is this value above 0.75?" and when it is filter will include that row in the output. Another way to say this is that when `by_auto` is above 0.75 *the condition is met.*

In R, as with most other programming languages, there are a number of *logical operators* that are used to check for conditions. We will learn these in the following section.

11.2 Logical operators

R has myriad ways to create logical expressions. Here we will focus on the six main *relational* logical operators. These are called relational because we are checking to see how one value relates to another. In general we tend to ask questions like "are these the same?", "are they not the same?", and "is one value larger or smaller than another?". While you are getting the hang of it, I encourage you to try and verbalize logical expressions.

- `<` : less than
- `>` : greater than
- `<=` : less than or equal to
- `>=` : greater than or equal to
- `==` : exactly equal (I like to think of it as *"are these the same thing?"*)
- `!=` : not equal (*"are these things not the same"*)
- `!`: Negation. This returns the opposite value.

Let's bring it back to early algebra and work with the variable x and y. Let's say `x = 3` and `y = 5`.

Is `x` less than `y`?

```
# set variables.
x <- 3
y <- 5

x < y
```

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

Yes it is. Here R returns a logical value which are represented as `TRUE` and `FALSE`.

Is `x` greater than `y`?

```
# greater than?
x > y
```

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

And for the sake of illustration:

```
# less than or equal
x <= y
```

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

```
# greater or equal
x >= y
```

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

```
# exactly equal?
x == y
```

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

```
# not equal
x != y
```

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

The power of logical operators isn't necessarily in the ability to compare one value against another, but the ability to compare many values to one value or even many values to multiple other values. This is what `filter()` helps us do.

Using `filter()` we can compare one column to another. When we do this, the values of those columns at each row are compared. For example, we can identify all of the Census Tracts where people walk more than they drive.

> Note: I am saving the results to an object called `walk` to then select only a few columns.

```
walking <- filter(commute, by_walk > by_auto)

select(walking, county, by_walk, by_auto)
```

```
## # A tibble: 54 x 3
##    county    by_walk by_auto
##    <chr>       <dbl>   <dbl>
##  1 HAMPSHIRE   0.652  0.0993
##  2 HAMPSHIRE   0.632  0.0985
##  3 HAMPSHIRE   0.580  0.189
##  4 HAMPSHIRE   0.613  0.143
```

```
##  5 HAMPSHIRE   0.625  0.0916
##  6 MIDDLESEX   0.427  0.213
##  7 MIDDLESEX   0.545  0.142
##  8 MIDDLESEX   0.372  0.287
##  9 MIDDLESEX   0.299  0.240
## 10 SUFFOLK     0.548  0.105
## # ... with 44 more rows
```

We can also use filter to see where the walking rates and the driving rates are the same. As shown above we use `==` to test if things are the same.

> **An important distinction**: `<-` is different from `=` and `=` is different from `==`. If you are ever confused about which operator to use ask yourself what your goal is. If your goal is to assign an object use `<-`. If your goal is to assign an argument value use `=`. And if you are trying to compare two things use `==`.

```
walk_auto <- filter(commute, by_walk == by_auto)

select(walk_auto, county, by_walk, by_auto)
```

```
## # A tibble: 2 x 3
##   county  by_walk by_auto
##   <chr>     <dbl>   <dbl>
## 1 <NA>          0       0
## 2 SUFFOLK       0       0
```

So far we have only been checking one condition and in most cases this actually will not suffice. You may want to check multiple conditions at one time. When we use filter we can add a logical expression as another argument. In doing so, filter will check to see if both conditions are met and, when they are, that row is returned. This is called an "and" statement. Meaning condition one **and** condition two need to be `TRUE`.

Building upon the `walking` example, we can further narrow down the observations by adding a second condition which returns only the observations that have median household incomes below $40,000.

```
low_inc_walk <- filter(commute,
                       by_walk > by_auto,
                       med_house_income < 40000)

select(low_inc_walk, by_walk, by_auto, med_house_income)
```

```
## # A tibble: 11 x 3
##    by_walk by_auto med_house_income
##      <dbl>   <dbl>            <dbl>
##  1   0.580   0.189             2499
##  2   0.581   0.214            21773
##  3   0.647   0.108            36250
##  4   0.407   0.170            34677
##  5   0.381   0.159            30500
##  6   0.465   0.192            28618
##  7   0.340   0.243            16094
##  8   0.536   0.112            19267
##  9   0.436   0.161            22930
## 10   0.451   0.170            36500
## 11   0.677   0.107            31218
```

11.2.1 & statements:

Now I want to introduce two more logical operators, **and** (`&`), which was implicitly used in the above filter statement, and **or** (`|`). `&` compares two conditions and will return `TRUE` only if they are both `TRUE`. `|` will return `TRUE` when one of the conditions is `TRUE`.

Now for an illustrative example of `&` statements :

```
# We have TRUE and TRUE, this should be false because they aren't both TRUE
TRUE & FALSE
```

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

```
# both a TRUE, we expect TRUE
TRUE & TRUE
```

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

```
# The first statement is TRUE, but the second is not TRUE, expect FALSE
(1 == 1) & (1 < 1)
```

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

```
# The first statement is TRUE and the second is TRUE, expect TRUE
(1 == 1) & (1 <= 1)
```

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

11.2.2 | statements:

Or statements are used to evaluate whether or not something meets **at least** one of the two conditions. This means that the only time that an *or* statement evaluates to `FALSE` is when both expressions result in `FALSE`.

```
# True is present, so we expect TRUE
TRUE | TRUE
```

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

```
# True is present, so we expect TRUE
TRUE | FALSE
```

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

```
#
FALSE | FALSE
```

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

We can alter the previous `filter()` statement to show us places that walk more than they drive **or** have a low median household income.

```
low_inc_or_walk <- filter(commute,
                          by_walk > by_auto | med_house_income < 40000)

select(low_inc_or_walk, by_walk, by_auto, med_house_income)
```

```
## # A tibble: 224 x 3
##    by_walk by_auto med_house_income
##      <dbl>   <dbl>            <dbl>
##  1 0.0959    0.810            37093
##  2 0.102     0.800            31465
##  3 0.0478    0.813            14604
##  4 0.144     0.647            34940
##  5 0.0541    0.831            26615
##  6 0.0345    0.931            37935
##  7 0.0430    0.750            16400
##  8 0.0543    0.785            19548
##  9 0.00469   0.944            34821
## 10 0         0.955            34697
## # ... with 214 more rows
```

11.2.3 Negation

Many times it will be easier to create a logical statement and say you want the *opposite* of those results. In this case we will use the bang operator or the exclamation mark, `!`. To negate a logical value or logical statement put the bang **in front** of the statement or value.

For example we can make `FALSE` true by negating it.

```
!FALSE
```

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

We can take a previous example

```
# The first statement is TRUE and the second is TRUE, expect TRUE
(1 == 1) & (1 <= 1)
```

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

```
# negate it
!(1 == 1) & (1 <= 1)
```

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

Keep this in your pocket for later.

11.3 Defining the Greater Boston Area

You now have developed the requisite skills to subset the commuting data to just the Greater Boston Area. But we still haven't completely decided what constitutes it. We will take the naïve approach and say that Suffolk, Norfolk, and Middlesex counties are the Greater Boston Area. We can now filter our data to just these counties!

```
gba_commute <- filter(commute, county == "SUFFOLK" | county == "NORFOLK" | county == "M
```

The above code is actually rather redundant as we have written `county ==` three different times. When we using the same equality comparison we can actually use the sepcial `%in%` operator. This lets us look for a value **in** a vector of values (we'll learn more about vectors very shortly).

For example:

```
1 %in% c(1, 2, 3)
```

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

This looks to see if the value on the left hand side is any of the three values in the vector—the thing that looks like `c(val1, val2, ...)`. Using this we can rewrite `gba_commute` as:

```
gba_commute <- filter(commute, county %in% c("SUFFOLK", "NORFOLK", "MIDDLESEX"))
```

11.3.1 Writing Data

You have created the proper subset of data that is needed. However, there is one more hurdle of jump—sending the data. To do this we need to get the tibble out of R and into a data format that can be used—probably a csv. `readr` provides funcitonality to do this as well.

While we used `read_csv()` earlier, to write a csv we will use `write_csv()`. The functionality is beautifully simple. The first argument here will be the tibble that you're going to write, `gba_commute` in this case. And the second is the path to where you will write the data.

In general I recommend that your project has two folders. One titled `data-raw` where you will keep the scripts and raw data that you used to process the data. Then I suggest having a `data` folder as well. This is where you will keep your tidy, or finalized, data files.

```
write_csv(gba_commute, "data/gba_commute.csv")
```

Now you have a csv file that can be shared!

Chapter 12

The pipe %>%

Until now we have been using one function at a time. This can feel like it is rather limiting at times. The approach that we have been taking has been to perform some action, save the resultant object, and then perform another action. This leads to either overwriting the same existing object multiple times with an assignment `<-` or creating multiple other objects. The former solution does not have a great story for reproducibility. At any point within a script may refr to many different objects with the same name. The second solution can clutter your working environment and lead to an excess usage of memory.

Well then, "what do we do instead?" you may be asking. And my answer is "use the forward pipe operator, of course." The forward pipe operator looks like `%>%`. This is a special function which allows the user to "pipe an object forward into a function or . . . expression" (Milton and Wickham, 2019). This is where the true power of the tidyverse comes from from.

What it does: The pipe operator takes the object or the output an expression on it's left hand side `lhs` and provides that as the first argument in the function of the right hand side. Additionally, it exposes the `lhs` as a temporary variable `..` It is documented as `lhs %>% rhs`.

The creator of #TidyTuesday and RStudio employee, Thomas Mock has created a very illustrative example of how the pipe can simplify complex R function calls[1].

The first example illustrates the creation of intermediate variables.

```
did_something <- do_something(data)
```

[1]Tidy Tuesday is a weekly online community event in which useRs across the world analyse the same dataset and share their visualizations online. Get involved on twitter with `#TidyTuesday`.

```
did_another_thing <- do_another_thing(did_something)

final_thing <- do_last_thing(did_another_thing)
```

The second demonstrates nesting function calls inside of other function calls.

```
final_thing <- do_last_thing(
  do_another_thing(
    do_something(
      data
    )
  )
)
```

Nested function calls are often difficult to debug and the user may get caught up in the mintuae of properly places parentheses.

> Note: Debugging is the process of taking misbehaving code and fixing it.

Using the pipe this chain of functions can be rewritten in the order that it happens.

```
final_thing <- data %>%
  do_something() %>%
  do_another_thing() %>%
  do_last_thing()
```

By using the pipe we are able to align our thinking and code writing. Additionally, each function call is separated on its own line which makes debugging a less daunting task.

12.1 Applying the pipe

Remember how I pointed out that the first argument for almost every function is the data? This is where that comes in handy. This allows us to use the pipe to chain together functions and "makes it more intuitive to both read and write" (magrittr vignette).

The tidyverse was designed with this in mind. This is why `select()`, `filter()`, and `mutate()` among many others are data first functions. Moreover, the output of each function is always a data frame which allows the user to provide that output as input into the next function.

As always, the most helpful way to wrap your head around this is to see it in action. Let's take one of the lines of code we used above and adapt it to use a pipe. We will select the name column of our data again. Previously we may have written

```
my_tbl <- select(data_frame, ...)
smaller_tbl <- filter(my_tbl, ...)
new_col_tbl <- mutate(smaller_tbl, ...)
```

But now we are able to write more complex function chains such as

```
data %>%
  filter() %>%
  mutate() %>%
  select()
```

In the chapter on filtering data we began by reading in the data, selecting columns, and then filtered data. Here we will recreate the `low_inc_or_walk` object which identified Census Tracts that have a higher rate of commuters who walk than drive or have a median household income below $40,000.

```
library(tidyverse)

low_inc_or_walk <- read_csv("data/ACS_1317_TRACT.csv") %>%
  select(
    county,
    starts_with("commute"),
    starts_with("by"),
    med_house_income
  ) %>%
  filter(
    by_walk > by_auto,
    med_house_income < 40000
  )
glimpse(low_inc_or_walk)
#> Rows: 11
#> Columns: 11
#> $ county           <chr> "HAMPSHIRE", "SUFFOLK", "SUFFOLK", "SUFFOLK", "SUF...
#> $ commute_less10   <dbl> 0.40234261, 0.34490741, 0.41918528, 0.09421755, 0....
#> $ commute1030      <dbl> 0.5077599, 0.3726852, 0.4042926, 0.5200162, 0.5528...
#> $ commute3060      <dbl> 0.07027818, 0.227777778, 0.14673675, 0.35624747, 0....
#> $ commute6090      <dbl> 0.01288433, 0.04953704, 0.02058695, 0.02951880, 0....
#> $ commute_over90   <dbl> 0.006734993, 0.005092593, 0.009198423, 0.000000000...
#> $ by_auto          <dbl> 0.1889132, 0.2140026, 0.1082621, 0.1704500, 0.1592...
```

```
#> $ by_pub_trans     <dbl> 0.04570873, 0.11933069, 0.15384615, 0.29191557, 0....
#> $ by_bike          <dbl> 0.014344761, 0.019815059, 0.008547009, 0.071286340...
#> $ by_walk          <dbl> 0.5801118, 0.5808014, 0.6471306, 0.4066109, 0.3805...
#> $ med_house_income <dbl> 2499, 21773, 36250, 34677, 30500, 28618, 16094, 19...
```

The reason this pipe works is because the output of each function call is yet another tibble and the pipe operator is passing that resultant tibble as the first argument to the next function.

Not only does the pipe aid in the manipulation of data, it also has a lot of utility in crafting ggplots. By piping your tibble into a ggplot call, this allows you to quickly iterate on the input data from either filtering down your data to creating new variables for visualization purposes.

The following sections will use the pipe operator in favor of the above listed alternatives.

12.2 Revisiting our scenario

Now that we have the pipe operator at our fingertips, we ought to think about how we can incorporate it into our previous work. In our earlier scenario we

```
acs_raw <- read_csv("data/ACS_1317_TRACT.csv")

commute <- select(acs_raw,
       county,
       hs_grad, bach, master,
       starts_with("commute"),
       starts_with("by"),
       med_house_income) %>%
  filter(county %in% c("SUFFOLK", "NORFOLK", "MIDDLESEX"))
```

Chapter 13

Creating new measures

It's been a week now and the non-profit has finally emailed you back. They were ecstatic with what you provided but it begat even more questions for them. They indicated that while the median household income data was very intruiging, that would be difficult for them to report on. As such, they would like you to report on the income quintiles as well. Moreover, they also would like to see the rate of Bachelor's and Master's degrees combined into one general educational attainment variable.

This poses some challenges for you. You know *what* is being asked, just not necessarily *how* to achieve that from R. To accomplish this we're going to have to learn how to use the `dplyr::mutate()` function. For the sake of example, let's select only the columns that we're going to need and make a tibble called `df` just to work with.

```
library(tidyverse)

commute <- read_csv("data/gba_commute.csv")
df <- select(commute, med_house_income, bach, master)

df
#> # A tibble: 648 x 3
#>    med_house_income  bach master
#>               <dbl> <dbl>  <dbl>
#>  1            75085 0.188 0.100
#>  2           132727 0.400 0.130
#>  3           110694 0.317 0.139
#>  4           109125 0.322 0.144
#>  5            76746 0.177 0.0742
#>  6           138700 0.310 0.207
#>  7           104673 0.247 0.149
```

```
#>  8            73191 0.300 0.126
#>  9           121488 0.198 0.140
#> 10            99358 0.348 0.151
#> # ... with 638 more rows
```

`mutate()` is a function that let's us create or modify variables. The arguments for `mutate()` are the same as those for `select()`—`.data` and `....` In the case of `mutate()` dots works a littble bit differently. After indicating our data, we create columns by specifying a name-value pair. More simply the names of our arguments will be the name of the columns that we are creating. The value is any expression. For example we could use `mutate(df, one = 1)` to create a column called `one` with the value of 1. When using mutate, however, the result from the expression needs to be either only *one* value, or as many values as there are rows.

If we take our `df`, we can add the columns `bach` and `master` together to create a new column called `edu_attain`.

```
mutate(df, edu_attain = bach + master)
#> # A tibble: 648 x 4
#>    med_house_income  bach master edu_attain
#>               <dbl> <dbl>  <dbl>      <dbl>
#>  1            75085 0.188 0.100       0.288
#>  2           132727 0.400 0.130       0.531
#>  3           110694 0.317 0.139       0.456
#>  4           109125 0.322 0.144       0.466
#>  5            76746 0.177 0.0742      0.251
#>  6           138700 0.310 0.207       0.516
#>  7           104673 0.247 0.149       0.396
#>  8            73191 0.300 0.126       0.426
#>  9           121488 0.198 0.140       0.338
#> 10            99358 0.348 0.151       0.499
#> # ... with 638 more rows
```

We could even think about ways that we can check if some observations are above some specified income threshold.

```
mutate(df, above_70k_inc = med_house_income > 80000)
#> # A tibble: 648 x 4
#>    med_house_income  bach master above_70k_inc
#>               <dbl> <dbl>  <dbl> <lgl>
#>  1            75085 0.188 0.100  FALSE
#>  2           132727 0.400 0.130  TRUE
#>  3           110694 0.317 0.139  TRUE
#>  4           109125 0.322 0.144  TRUE
```

```
#>  5            76746 0.177 0.0742 FALSE
#>  6           138700 0.310 0.207  TRUE
#>  7           104673 0.247 0.149  TRUE
#>  8            73191 0.300 0.126  FALSE
#>  9           121488 0.198 0.140  TRUE
#> 10            99358 0.348 0.151  TRUE
#> # ... with 638 more rows
```

This function is immensly useful and can be combined with almost any expression to create new data for us. Furthermore there are a number of handy functions built into dplyr that help us create new columns. Some of these are `case_when()`, `min_rank()`, and `ntile()` among others. You can always explore these with `?function_name()`. For our purposes, we will look at the use of `ntile()`.

`ntile()` is a function that will calculate percentiles for us. Given a column of data, `x`, and a number of buckets, `n`, we can create a new column of ranks. In our case, we are interested in calculating the quintile of `med_house_income`. This means we can provide `med_house_income` and `n = 5` as arguments to `ntile()` to group our observations by quintile.

```
mutate(df, inc_quintile = ntile(med_house_income, 5))
#> # A tibble: 648 x 4
#>    med_house_income  bach master inc_quintile
#>               <dbl> <dbl>  <dbl>        <int>
#>  1            75085 0.188 0.100            2
#>  2           132727 0.400 0.130            5
#>  3           110694 0.317 0.139            4
#>  4           109125 0.322 0.144            4
#>  5            76746 0.177 0.0742           2
#>  6           138700 0.310 0.207            5
#>  7           104673 0.247 0.149            4
#>  8            73191 0.300 0.126            2
#>  9           121488 0.198 0.140            5
#> 10            99358 0.348 0.151            4
#> # ... with 638 more rows
```

Now we can put everything together into one mutate call to create the new variables that were requested!

```
updated_commute <- commute %>%
  mutate(edu_attain = bach + master,
         inc_quintile = ntile(med_house_income, 5))
```

As you have made these changes you can now write the data again to csv and share it. As this process becomes more and more iterative, it's good to put *some*

structure to the data so you have an idea of the history. One general practice that is good to get into is dating your files. So in this case I would label the file `yyyy-mm-dd-commute.csv`.

Chapter 14

Data Structures

There is a topic I have been skirting around for some time now and I think it is time that we have to have a rather important conversation. It's one that is almost never fun but is quite necessary because without it, there may be many painful lessons learned in the future. We're going to spend this next chapter talking about data structures—but not all of them! We'll only cover the three most common and, by the end of this, it is my hope you will have a much stronger idea of *what* you are working with and *why* it behaves the way it does.

We will cover vectors, data frames rather briefly, and lists. We'll talk about some of their defining characteristics and how we can interact with them. Often the theory behind these object types are omitted, but I am of the mind that learning this early on will pay off in dividends. Take a deep breath before we dive in and remind yourself that it ain't nothin' but a thing.

This section is undoubtedly the most theoretically dense from a software perspective of this entire book. These concepts may be a little bit difficult to grasp at the first go around particularly if you do not have a programming background. But do not be discouraged! This is tough and there is no way to around it, so might as well go through it. If you can grasp this chapter programming in R will become so much easier. You will develop an intuition of why certain things happen to your data and how to interact with other data structures.

14.1 Atomic Vectors

I like to think of the atomic vector much like the atom—that is as the building block of any R object. You've actually been working with atomic vectors this entire time. But we haven't been very explicit about this yet. Up until this point we have been working mainly with tibbles. And here is the secret: each column of a tibble is *actually* an atomic vector.

What makes a vector a atomic is that it can only be **a single data type** and that they are *one-dimensional*—opposed to tibbles which are two-dimensional[1]. You may have noticed that every value of a column is of the same data type. This means that they are rather strict to work with and for good reason. Imagine you wanted to multiple a column by 10, what would happen if a few of the values in the column were actually written out as text? Let's try exploring this idea.

The most common way to create a vector in R is to use the `c()` function. This stands for *combine*. We can combine as many elements as we want into a single vector using `c()`. Each element of the vector is it's own argument (separated by a comma).

For example if we wanted to create a vector of Boston's unemployment rate rate for each month in 2019 that we have data for (until October as of this writing on Dec. 18th, 2019) we could write the below. We will save it in a a vector called `unemp`[2].

```
unemp <- c(3.2, 2.8, 2.8, 2.4, 2.8, 2.9, 2.7, 2.6, 2.7, 2.3)
unemp
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3
```

What is really great about vectors is that we can perform any number of operations on them—i.e. find the sum of all the values, the average, add a value to each element, etc.

If we wanted to find the average unemployment rate for Boston for Jan - Oct. 2019, we can supply the vector to the function `mean()`.

```
mean(unemp)
#> [1] 2.72
```

However, you may be thinking "there are 12 months in a year not 10 and that should be represented" and if you are, I totally agree with you. Since the data for November and December are missing, we should denote that and update `unemp` accordingly. R uses `NA` to represent missing data. To represent this we can append two `NA`s to the vector we have. There are two ways we can do this. We can either combine `unemp` with two `NA`s, or rewrite the above vector.

```
# combining existing with 2 NAs
c(unemp, NA, NA)
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3  NA  NA
```

[1] Data Types and Structures. Software Carpentries. https://swcarpentry.github.io/r-novice-inflammation/13-supp-data-structures/

[2] Boston Unemployment https://data.bls.gov/timeseries/LAUMT257165000000003?amp%253bdata_tool=XGtable&output_view=data&include_graphs=true.

This works, but since we will be saving this to `unemp` again it is not best practices to use the variable you are changing in that objects assignment.

```
# for example
unemp <- c(unemp, NA, NA)
```

The above is rather unclear and might confuse someone that will have to read your code at a later time—that person may even be you. For this reason we will redefine it.

```
unemp <- c(3.2, 2.8, 2.8, 2.4, 2.8, 2.9, 2.7, 2.6, 2.7, 2.3, NA, NA)
unemp
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3  NA  NA
```

We know that there are 12 elements in this vector, but sometimes it is quite nice to sanity check oneself. We can always find out how long (or how many elements are in) a vector is by supplying the vector to the `length()` function.

```
# how many observations are in `unemp`?
length(unemp)
#> [1] 12
```

There are a total of six types of vectors. Fortunately, only four of these really matter to us. These are `integer`, `double`, `character`, `logical`.

Integers represent whole numbers. To specify an integer we append an `L` after the number such as `20L`. Doubles are any number that requires any precision aka decimal places. You can specify doubles in a number of formats such as scientific notation. Generally the easiest way to do this, though, is using a decimal. Together integers and doubles are lumped into the category of numeric. This is because, well, they are numbers.

As you learned previously, character vectors are created with the use of quotation marks; either `"` or `'`.

We've already created a vector of type double, `unemp`. You can check what type of vector `unemp` is with `typeof()`

> Note: `typeof()` is used only for internal R object such as lists and vectors. In most cases you will want to use `class()` to return the class of an object.

```
typeof(unemp)
#> [1] "double"
```

Say we create another vector called `month` with the numbers 1 through 12.

```
month <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)

typeof(month)
#> [1] "double"
```

Notice that since we didn't specify the `L` after the numbers R defaulted to treating `month` as a double. When possible it is good to make the distinction between integer and numeric.

```
month <- c(1L, 2L, 3L, 4L, 5L, 6L, 7L, 8L, 9L, 10L, 11L, 12L)

typeof(month)
#> [1] "integer"
```

R has a number of vectors that are built in these being the letters of the alphabet (`letters` and `LETTERS` respectively), as well as `month.abb`, `month.name`, and `pi`. `month.name` is already available to us so let's not recreate it.

```
month.name
#>  [1] "January"   "February"  "March"     "April"     "May"       "June"
#>  [7] "July"      "August"    "September" "October"   "November"  "December"

typeof(month.name)
#> [1] "character"
```

Notice the quotes around each vector element. This is how we identify character vectors.

Logical vectors are the last kind of vector we need to go over. Logical vectors are represented as the values `TRUE` and `FALSE`. Simple enough. Onward!

Recall that vectors are atomic meaning that there can only be one type per vector and we cannot mix and match. When a character is in the presence of another element of a different type, that value is ***coerced*** into a character. Coersion is the process of implicitly or contextually changing an object from one type to another. For example:

```
x <- c("a", 1)

typeof(x)
#> [1] "character"
```

Something similar happens when a logical value is in the presence of a numeric value

```
c(TRUE, 1, FALSE)
#> [1] 1 1 0
```

In the presence of a numeric value `TRUE` becomes equal to `1L` and `FALSE` equal to `0L`.This behavior exists whenever a logical value is presented where a numeric is expected such as the function call below.

```
sum(TRUE, FALSE, FALSE)
#> [1] 1
```

While coersion occurs from other processes like combining values in a vector, ***casting*** is the process of intentionally changing an object's class. There are a number of casting functions whice generaly take the shape of `as.class()` or `as_class()`. Each of the vector types covered have their own casting functions.

```
as.integer(TRUE)
#> [1] 1
as.character(123)
#> [1] "123"
as.double("2.331")
#> [1] 2.331
as.logical(0)
#> [1] FALSE
```

As you progress in your R journey you will find scenarios in which you need to cast objects from one class to another and these functions are the trick.

You now have a strong understanding of the underbellies of R vectors. One thing that is missing is an understanding of how we can select subsets from vectors. To extract a value from vectors we append square brackets at the end of the vector `vec[]`. We supply an index value to the square brackets to receive the value at that position

To select the month of January from the `unemp` vector, the first element, we provide the value of `1` to the brackets.

```
unemp[1]
#> [1] 3.2
```

To extract more than one value, we provide a vector of the row indexes we desire.

```
unemp[c(1, 3)]
#> [1] 3.2 2.8
```

There is yet another way to extract values from these vectors. We can provide a logical vector to our square brackets. For example, we can identify every value of `unemp` that is above the average rate.

```
# find average removing missing values
avg_unemp <- mean(unemp, na.rm = TRUE)

# identify which values are above average
index <- unemp > avg_unemp

index
#>  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE    NA    NA
```

Notice that the `NA`s stayed `NA`? They can be pesky. Hadley writes in Advanced R "missing values tend to be infectious: most computations involving a missing value will return another missing value."[3]

```
unemp[index]
#> [1] 3.2 2.8 2.8 2.8 2.9  NA  NA
```

How annoying those NAs can be! To prevent these NAs from showing upwe can add another condition to our `index` line to remove NAs. Like there are `as.*()` functions for casting, there are also `is.*()` functions for testing. `is.*()` returns a logical vector of the same length as the provided vector.

> Note: the `*` is called a wildcard. The wildcard character comes from SQL and when present means that any string can follow. `is.*()` is intended to indicate any possible testing function such as `is.numeric()`, `is_tibble()`, etc.

```
is.na(unemp)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

As we learned, we can negate logical vectors with an `!`. We can negate the test results and include an an `&` condition to only identify unemployment values above average *and* aren't missing.

```
index <- unemp > avg_unemp & !is.na(unemp)

unemp[index]
#> [1] 3.2 2.8 2.8 2.8 2.9
```

[3] Vectors. Advanced R. https://adv-r.hadley.nz/vectors-chap.html.

There is one last thing to keep in mind and with subsetting vectors using a logical vector that is of a different length. When you use a logical vector to subset and they are of differing length, the logical vector will be recycled for the remaining values of the vector being subset. As always, an example will be the best.

Say we have an object called `x` which are the values from 0 to 10 and an `index` to subset with. If we subset it with `index` and `index` is a logical vector of length two with the values of `TRUE` and `FALSE`, every other observation will be returned. This is because come the third value in `x`, R has ran out of values in `index` to use so it goes back to the beginning

```
x <- 0:10
x
#>  [1]  0  1  2  3  4  5  6  7  8  9 10
```

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

x[index]
#> [1]  0  2  4  6  8 10
```

And what happens when the only value is a single logical value?

```
x[TRUE]
#>  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```
x[FALSE]
#> integer(0)
```

In this latter case see how the output says `integer(0)`. This is informing you that the vector contains 0 elements.

14.2 Data Frames

The entirety of the work in this book so far has been with `tibbles`. Tibbles are actually a special type of data frame. Data frames are R's native way for storing rectangular data. Rectangles are two-dimensional, so are data frames.

Data frames are secretly just a bunch of vectors squished together. The important thing is that all vectors are of the same length. This ensures that each observation (row) has one value from each vector. Because of the nature of a data frame, each column must adhere to the rules of vectors.

Let's create a tibble using the `unemp` vector and the `tibble()` function. `tibble()` works in a somewhat similar manner as `mutate()` where the arguments we provide are name value pairs. In the case of tibble, the argument take the form of `col_name = vector`.

We create a tibble with the unemployment rate below.

```
library(dplyr)

tibble(
  unemp_rate = unemp
)
#> # A tibble: 12 x 1
#>    unemp_rate
#>         <dbl>
#>  1        3.2
#>  2        2.8
#>  3        2.8
#>  4        2.4
#>  5        2.8
#>  6        2.9
#>  7        2.7
#>  8        2.6
#>  9        2.7
#> 10        2.3
#> 11         NA
#> 12         NA
```

We can add the month name and create a new column to indicate if that month has a higher than average unemployment rate.

```
unemp_tbl <- tibble(
  unemp_rate = unemp,
  month = month.name
) %>%
  mutate(above_avg = unemp_rate > avg_unemp)

unemp_tbl
#> # A tibble: 12 x 3
#>    unemp_rate month     above_avg
#>         <dbl> <chr>     <lgl>
#>  1        3.2 January   TRUE
#>  2        2.8 February  TRUE
#>  3        2.8 March     TRUE
#>  4        2.4 April     FALSE
#>  5        2.8 May       TRUE
```

```
#>  6        2.9 June      TRUE
#>  7        2.7 July      FALSE
#>  8        2.6 August    FALSE
#>  9        2.7 September FALSE
#> 10        2.3 October   FALSE
#> 11       NA   November  NA
#> 12       NA   December  NA
```

To interact with the underlying vector of a data frame we can use the dollar sign `$` operator. This takes the form of `tbl$col_name`.

For example, extracting the `unemp_rate` column looks like:

```
unemp_tbl$unemp_rate
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3  NA  NA
```

Note the difference between `select(tbl, col)` and `tbl$col`.

```
select(unemp_tbl, unemp_rate)
#> # A tibble: 12 x 1
#>    unemp_rate
#>         <dbl>
#>  1        3.2
#>  2        2.8
#>  3        2.8
#>  4        2.4
#>  5        2.8
#>  6        2.9
#>  7        2.7
#>  8        2.6
#>  9        2.7
#> 10        2.3
#> 11       NA
#> 12       NA
```

The difference is that `$` returns the underlying vector whereas `select()` will always return another data frame. You now have the ability to both filter data and grab a subset of a vector. But we have yet to visit how to grab a single value from a data frame.

You could try something like

```
unemp_tbl %>%
  select(1) %>%
  slice(10)
```

```
#> # A tibble: 1 x 1
#>   unemp_rate
#>        <dbl>
#> 1        2.3
```

To grab the 10th value of the first column. But again, you still have a tibble and you are not able to use that directly like a standalone number.

We can again use brackets to subset the our R object. But data frames are two dimensional, so we need to specify the indexes in two dimensions. If you have made a hand drawn graph used a cartesian plane, which I assume you all have, this will is the same idea. With a cartesian plane we can identify any point with a combination of two values: x and y. x refers to the horizontal axis and y the vertical axis. When we put the cartesian plane in the same frame of reference as the rectangular data frame we envision our rows as the x and our columns as the y.

In specifying our index, we are able to select all rows or all columns by leaving the x or y spot empty respectively.

```
unemp_tbl[,1]
#> # A tibble: 12 x 1
#>    unemp_rate
#>         <dbl>
#>  1        3.2
#>  2        2.8
#>  3        2.8
#>  4        2.4
#>  5        2.8
#>  6        2.9
#>  7        2.7
#>  8        2.6
#>  9        2.7
#> 10        2.3
#> 11         NA
#> 12         NA
unemp_tbl[10,]
#> # A tibble: 1 x 3
#>   unemp_rate month   above_avg
#>        <dbl> <chr>   <lgl>
#> 1        2.3 October FALSE
```

To replicate the above tidyverse example we would provide the indexes 10 and 1 respectively.

```
unemp_tbl[10,1]
#> # A tibble: 1 x 1
#>   unemp_rate
#>        <dbl>
#> 1        2.3
```

This is great, we've rewritten our tidyverse code in base R. But, just like the tidyverse code, we maintain the tibble data structure. This is because when we use a single bracket, it maintains the data structure of the object we are selecting from. If we wrap our brackets in another set of bracket, we are returned the an object of the same class as the underlying object.

```
unemp_tbl[[10,1]]
#> [1] 2.3
```

What that code is doing is narrowing the tibble down to a single column with a single row index and then extracting the underlying vector (the second bracket). To extract the underlying vector using the tidyverse, we can use the function `dplyr::pull()`.

```
unemp_tbl %>%
  select(1) %>%
  slice(10) %>%
  pull()
#> [1] 2.3
```

Now this brings us to the second-most fundamental structure in R: the list. Yes, second-most fundamental. I've been keeping a secret from you. Data frames are actually just lists in disguise. To prove it, I will remove the class from `unemp_tbl` and return the class of that unclassed object.

```
unclass(unemp_tbl) %>%
  class()
#> [1] "list"
```

That is right, data frames are actually just lists disguised as rectangles.

14.3 Lists

There is a good chance that you will not have to interact with them too often That doesn't mean you shouldn't know how to when that time comes.

Lists are generally the most flexible object type in R. Unlike vectors and data frames lists do not impose any structure on the storage of our data.

The most simple lists may resemble something like a vector.

```
list("Jan", "Feb", "Mar")
#> [[1]]
#> [1] "Jan"
#>
#> [[2]]
#> [1] "Feb"
#>
#> [[3]]
#> [1] "Mar"
```

Notice how this prints differently than

```
c("Jan", "Feb", "Mar")
#> [1] "Jan" "Feb" "Mar"
```

Each element of a list is self-contained. I think of lists somewhat like shipping containers where each element is its own container and all components of each element are together. We can include any type of R object in a list. For example, we can include the `unemp_tbl` and associated vectors.

```
l <- list(unemp_tbl, unemp, month.name)
```

We can view the structure of the list to get an idea of what is actually contained by that list.

```
str(l)
#> List of 3
#>  $ : tibble [12 x 3] (S3: tbl_df/tbl/data.frame)
#>   ..$ unemp_rate: num [1:12] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3 ...
#>   ..$ month     : chr [1:12] "January" "February" "March" "April" ...
#>   ..$ above_avg : logi [1:12] TRUE TRUE TRUE FALSE TRUE TRUE ...
#>  $ : num [1:12] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3 ...
#>  $ : chr [1:12] "January" "February" "March" "April" ...
```

The structure of `l` shows us that the first element is a tibble (has class `tbl_df`), and the other elements are numeric and character vectors respectively.

Because of this flexibility there are not predetermined dimensions that we can specify to our brackets. Like extracting the underlying vector value from a data frame we have to use `[[` for indexing. I like to think of `[` as walking up to the storage container and `[[` as actually opening it up and going inside. To get a sense of the difference lets look at the `unemp` vector.

```
l[2]
#> [[1]]
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3  NA  NA
class(l[2])
#> [1] "list"
```

When using the single bracket we are just selecting the first element of the list which is why we are returned another list.

```
l[[2]]
#>  [1] 3.2 2.8 2.8 2.4 2.8 2.9 2.7 2.6 2.7 2.3  NA  NA
class(l[[2]])
#> [1] "numeric"
```

When we use the double bracket we are going inside of the container and actually plucking that element out of the list. Once you have plucked out that element, we can again use another set of brackets to subset that item. To grab the tenth row and first column of the `unemp_tbl` inside of `l` we can write.

```
l[[1]][[10,1]]
#> [1] 2.3
```

Now that we know that data frames are lists we can actually extract the underlying vectors using `[[` as well as `$`. We can get the tenth row and first column a number of ways.

```
# subsetting the data frame
l[[1]][[10,1]]
#> [1] 2.3

# grabbing the first vector then position
l[[1]][[1]][10]
#> [1] 2.3

# grabbing the vector by name then position
l[[1]]$unemp_rate[10]
#> [1] 2.3
```

Frankly all of these brackets can get a little messy. The tidyverse package `purrr` has a super handy function called `pluck()` which handles all of these brackets for us. `purrr::pluck()` is meant for flexible indexing into data structures (documentation).

`pluck()` works by first providing the object that you'd like to index—again, notice the data first emphasis—and then providing the position of the element

you would like to pluck out of the object. Generally, I will use `pluck()` when possible. By doing so the code becomes more readable and adheres to a single style more thoroughly.

```
purrr::pluck(l, 1, 1, 10)
#> [1] 2.3
```

Congratulations! You made it to the end of this exceptionally dense chapter. You may feel a little overwhlemed and that is to be expected. Nonetheless you should be proud! I have a few more asks of you before you move on.

14.3.1 Exercises

1. Drink some water
2. Move around a bit and shake it out
3. Create a list with the vectors `unemp`, `month.name`, and `avg_unemp`.
4. Recreate the `unemp_tbl` but referencing the list elements

```
library(purrr)

unemp_l <- list(unemp, month.name, avg_unemp)

tibble(
  unemp_rate = pluck(unemp_l, 1),
  month = pluck(unemp_l, 2)
) %>%
  mutate(above_avg = unemp_rate > pluck(unemp_l, 3))
#> # A tibble: 12 x 3
#>    unemp_rate month     above_avg
#>         <dbl> <chr>     <lgl>
#>  1        3.2 January   TRUE
#>  2        2.8 February  TRUE
#>  3        2.8 March     TRUE
#>  4        2.4 April     FALSE
#>  5        2.8 May       TRUE
#>  6        2.9 June      TRUE
#>  7        2.7 July      FALSE
#>  8        2.6 August    FALSE
#>  9        2.7 September FALSE
#> 10        2.3 October   FALSE
#> 11         NA November  NA
#> 12         NA December  NA
```

Chapter 15

Summary statistics

The last chapter we focused on the underlying data structures that we interact with in R. Most importantly we covered the atomic vector data structure and learned that the columns of a tibble are vectors. When we have been using `mutate()` to create new columns, we have actually been creating other vectors. When we have filtered data, we have checked the values of the underlying vectors to see if they have matched our specified conditions. Moving forward, we will begin to think about ways of summarizing our data. To do so we will be working with vectors more often. Being able to understand the role a vector plays in data frame operations will make this learning process even easier.

Let us start by asking the question *"what is a statistic?"* Very simply a statistic is a single number that is used to characterize a sample of data. Most often we see statistics used to describe central tendancy and spread—measures like the mean and standard deviation. However, the ways in which we can characterize a sample of data are not restricted to traditional frequentists statistics. We want to be more creative in the ways that we look at our data. In addition to evaluating central tendancy and spread we may find ourselves looking at the average word counts of tweets, or distances from the Boston Common, and so much more.

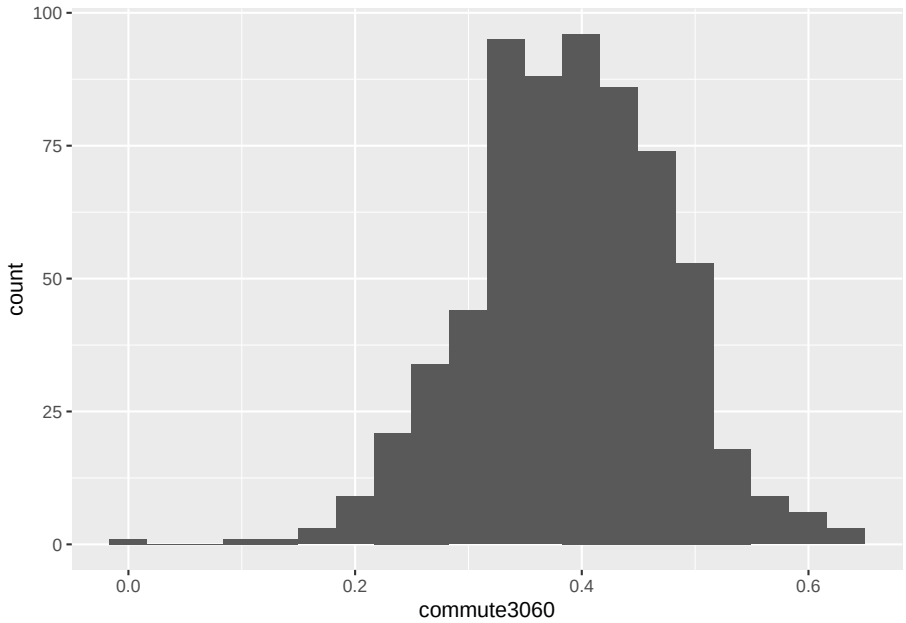
When we begin to summarize data, we are taking all observations or maybe a subset of observations and calculating one value to represent that sample. This is very important framing to have. Whenever we want to create an aggregate measure of our data there must be only one observation per-subset. Meaning, if you have a data frame with 150 rows and 3 groups within that, there ought to be only three resultant observations—though there may be many more variables.

Let us revisit the `commute`, and specifically the rate of commuters who travel between 30 and 60 minutes.

```
library(tidyverse)

commute <- read_csv("data/gba_commute.csv")
```

It is of course of interest to identify the average rate of 30-60 minute commuting, as well as the standard deviation, and median. What does this look like and how is it done? Prior to measuring central tendency and spread, we begin by visualizing our data. The purpose of visualizing your data before hand is to give you an intuition of how it may behave and its shape.

```
ggplot(commute, aes(commute3060)) +
  geom_histogram(bins = 20)
#> Warning: Removed 6 rows containing non-finite values (stat_bin).
```



In the above histogram we can intuit a number of things. The distribution looks fairly normally distributed meaning that both the mean and median are likely to be close to eachother and are equally sound measures of center, most likely somewhere around 0.3. Secondly, due to the distribution's rather round shape (or "fat tails"), it can be expected to have a rather large standard deviation. Once the intuition has been developed, one should calculate the relevant statistics to quantify these characteristics.

Let's calculate the mean, median, standard deviation, and range of this single variable. When calculating statistics like the mean and standard deviation, we

are calculating univariate statistics and as such, working with only one column (variable) at a time—this tends to be the case almost always.

We will first extract the `commute3060` column as a vector using `dplyr::pull()`.

```
commute_rate <- pull(commute, commute3060)
```

> Exercise: Read the help documentation for the functions `mean()`, `median()`, and `sd()` to get a sense of how these functions work. Calculate the mean, median, and standard deviation of `commute_rate`.

```
mean(commute_rate)
#> [1] NA

median(commute_rate)
#> [1] NA

sd(commute_rate)
#> [1] NA
```

The results of these functions bring good and bad news. The good news is that their output is a single value. The bad news is that the output is `NA`. To reiterate a previous point, `NA` will infect your analyses. The only way to get around this is to perform these calculations without them.

Note that each of the functions used above have an argument called `na.rm`. `na.rm` tells R to remove the NAs prior to calculation. We can recalculate these statistics with the `na.rm` argument set to `TRUE`.

```
mean(commute_rate, na.rm = TRUE)
#> [1] 0.3896481

median(commute_rate, na.rm = TRUE)
#> [1] 0.3921988

sd(commute_rate, na.rm = TRUE)
#> [1] 0.08624757
```

Let us look at one last example: identifying the range. The `range()` function returns the minimum and the maximum values.

```
(commute_range <- range(commute_rate, na.rm = TRUE))
#> [1] 0.0000000 0.6327961
```

> Note: by wrapping an assignment in parentheses, the resultant object will be printed to the console.

`range()` returned two values. This can be verified with the `length()` function.

```
length(commute_range)
#> [1] 2
```

A length two vector does not adhere to providing just out value. We will see why this is a problem illustrated in the next chapter. To recreate this, use the `min()` and `max()` functions.

```
min(commute_rate, na.rm = TRUE)
#> [1] 0
max(commute_rate, na.rm = TRUE)
#> [1] 0.6327961
```

Each of these statistics—mean, median, standard deviation, etc—are a way to characterize a larger sample of data. The lesson to take away here is that we will always need a single value when summarising data. Often we will be taking a column (vector) and calculating a single metric from that.

In the following chapter we will learn how to calculate summary statistics using the tidyverse.

Chapter 16

Summarizing the tidy way

Now that we have a basic understanding of how to manipulate our dataset, summarising the dataset into a few useful metrics is important. When we have massive datasets with many subgroups, summary statistics will be very important for distilling all of that information into something consumable. Aggregation will also be very important for visualization purposes.

We have already reviewed what constitutes a summary statistic and how to create them working with vectors. But we have not done so within the context of the tidyverse. We have figured out how to select, filter, mutate and all within a chain of functions. But we have not followed this to its natural next step, the `group_by()` and `summarise()` functions.

dplyr incluides a wonderful helper function called `count()`. It does just what it says it does. It counts the number of observations in a tibble. Let's recreate the `commute` tibble and see it for ourselves.

```
library(tidyverse)

commute <- read_csv("data/gba_commute.csv")

count(commute)
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   648
```

We can also count by groups in a data set. For example, we can count how many observations there are per county.

```
count(commute, county)
#> # A tibble: 3 x 2
#>   county        n
#>   <chr>     <int>
#> 1 MIDDLESEX   318
#> 2 NORFOLK     130
#> 3 SUFFOLK     200
```

`count()` is actually a wrapper around the function `summarise()` which is a much more flexible function. `summarise()` is the aggregate analog to `mutate()`. The difference between `mutate()` and `summarise()` is that the result of an expression in `mutate()` must have the same number of values as there are rows—unless of course you are specifying a scalar value like `TRUE`—whereas `summarise()` requires the result to be one an element of length one.

> Notes: - A wrapper is function that executes another function. - A scalar is a vector of length one.

We can recreate the first above `count()` call with `summarise()` and the handy `n()` function we learned a while ago. Here we follow the same pattern of assigning column names to expressions as we do with.

```
summarise(commute, n = n())
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1   648
```

Like `mutate()` there is no restriction on the number of new columns we can create. Previously we calculated the min, max, mean, and standard deviation of the `commute3060` variable. This is done rather neatly with `summarise()`.

```
commute %>%
  summarise(
    min_commute = min(commute3060, na.rm = TRUE),
    max_commute = max(commute3060, na.rm = TRUE),
    avg_commute = mean(commute3060, na.rm = TRUE),
    sd_commute  = sd(commute3060, na.rm = TRUE)
    )
#> # A tibble: 1 x 4
#>   min_commute max_commute avg_commute sd_commute
#>         <dbl>       <dbl>       <dbl>      <dbl>
#> 1           0       0.633       0.390     0.0862
```

Frankly this alone is somewhat unimpressive. The power of `summarise()` comes from incorporating `group_by()` into the function chain. `group_by()` allows us to explicitly identify groups within a tibble as defined by a given variable. The resulting tibble from a `group_by()` call is seemingly unchanged.

```
commute %>%
  group_by(county)
#> # A tibble: 648 x 14
#> # Groups:   county [3]
#>    county hs_grad  bach master commute_less10 commute1030 commute3060
#>    <chr>    <dbl> <dbl>  <dbl>          <dbl>       <dbl>       <dbl>
#>  1 MIDDL~   0.389 0.188 0.100          0.0916       0.357       0.375
#>  2 MIDDL~   0.167 0.400 0.130          0.0948       0.445       0.344
#>  3 MIDDL~   0.184 0.317 0.139          0.0720       0.404       0.382
#>  4 MIDDL~   0.258 0.322 0.144          0.0983       0.390       0.379
#>  5 MIDDL~   0.301 0.177 0.0742         0.0670       0.379       0.365
#>  6 MIDDL~   0.159 0.310 0.207          0.0573       0.453       0.352
#>  7 MIDDL~   0.268 0.247 0.149          0.0791       0.475       0.368
#>  8 MIDDL~   0.261 0.300 0.126          0.137        0.450       0.337
#>  9 MIDDL~   0.315 0.198 0.140          0.0752       0.478       0.329
#> 10 MIDDL~   0.151 0.348 0.151          0.0830       0.474       0.322
#> # ... with 638 more rows, and 7 more variables: commute6090 <dbl>,
#> #   commute_over90 <dbl>, by_auto <dbl>, by_pub_trans <dbl>, by_bike <dbl>,
#> #   by_walk <dbl>, med_house_income <dbl>
```

However, if we look at comments above the tibble, we see something new: `# Groups: county [3]`. This tells us a couple of things. First that the groups were created using the `county` column, that there are fifteen groups, and that the data frame is now grouped implying that any future `mutate()` or `summarise()` calls will be performed on the specified groups. If we then look at the class of that grouped tibble we see that there is a new class introduced which is `grouped_df`.

```
commute %>%
  group_by(county) %>%
  class()
#> [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

> Note: a tibble has the classes `tbl` and `tbl_df` on top of the Base R class `data.frame`.

When a tibble has this object class, dplyr knows that operations should be grouped. For example if you were to calculate the mean, this would be the mean for the specified groups rather than the mean for the entire dataset. One

function that is extremely useful is the `n()` function to identify how many observations there are per group inside of a mutate call.

> I am including the `commute3060` column to illustrate that the new `n` column will be the same for each group value.

```
commute %>%
  group_by(county) %>%
  mutate(n = n()) %>%
  select(county, commute3060, n)
#> # A tibble: 648 x 3
#> # Groups:   county [3]
#>    county    commute3060     n
#>    <chr>           <dbl> <int>
#>  1 MIDDLESEX       0.375   318
#>  2 MIDDLESEX       0.344   318
#>  3 MIDDLESEX       0.382   318
#>  4 MIDDLESEX       0.379   318
#>  5 MIDDLESEX       0.365   318
#>  6 MIDDLESEX       0.352   318
#>  7 MIDDLESEX       0.368   318
#>  8 MIDDLESEX       0.337   318
#>  9 MIDDLESEX       0.329   318
#> 10 MIDDLESEX       0.322   318
#> # ... with 638 more rows
```

Here each group only has one unique value for `n`. As discussed previously, when we want to calculate aggregate measures, there ought to only value per-group. This ability to perform grouped calculation within `mutate()`can be extremely powerful, but does not create a proper aggregated dataset. For this, we can again use `summarise()`

Let's recreate the grouped count from before.

```
commute %>%
  group_by(county) %>%
  summarise(n = n())
#> # A tibble: 3 x 2
#>   county        n
#>   <chr>     <int>
#> 1 MIDDLESEX   318
#> 2 NORFOLK     130
#> 3 SUFFOLK     200
```

We can also include the summary statistic calculations from before.

```
commute %>%
  group_by(county) %>%
  summarise(
    n = n(),
    min_commute = min(commute3060, na.rm = TRUE),
    max_commute = max(commute3060, na.rm = TRUE),
    avg_commute = mean(commute3060, na.rm = TRUE),
    sd_commute  = sd(commute3060, na.rm = TRUE)
    )
#> # A tibble: 3 x 6
#>   county        n min_commute max_commute avg_commute sd_commute
#>   <chr>     <int>       <dbl>       <dbl>       <dbl>      <dbl>
#> 1 MIDDLESEX   318       0.104       0.612       0.383     0.0825
#> 2 NORFOLK     130       0.186       0.613       0.392     0.0761
#> 3 SUFFOLK     200       0           0.633       0.400     0.0972
```

Chapter 17

Toolkit review

We've now come to the end of the first technical section of the Urban Informatics Toolkit. And you have officially covered ***a lot*** of ground. You've installed both R and RStudio. You've learned the basics of R operations and data structures. You've read and manipulated a large dataset, selected columns, created new ones, and even created a few visualization. You've learned to chain multiple functions together and have even created your own sets of summary statistics. These are all very important useful skills which will serve the foundation of everything else you will do in R.

The next section of this book will focus entirely on data visualization. We will begin by learning about the Grammar of Graphics. Next we will learn how to apply that grammar in R with ggplot2. Following, we will create a *ton* of graphics and build intuition about when and how to create different types of graphics.

Are you ready?

Are you hydrated?

Take a deep breath and let's get after it!

Part III

Vizualization Strategies

Chapter 18

Grammar of layered graphics I

You've made it quite far through this book. Now, I want to bring us back to the very beginning. In the first chapter we created a few visualizations with `ggplot2`. I want to unpack ggplot2 a bit more and also address some of the more philosophical underpinnings of visualization.

This chapter introduces you to the idea of the grammar of graphics, discusses when which visualizations are appropriate, and some fundamental design principles follow.

18.1 The Grammar of Layered Graphics

The `gg` in `ggplot2` refers to the grammar of graphics (and the `2` is because it's the second iteration). *The Grammar of Graphics* (Wilkinson, 1999) is a seminal book in data visualization for the sciences in which, Wilkinson defines a complete system (grammar) for creating visualizations that go beyond the standard domain of "named graphics"—e.g. histogram, barchart, etc. [1][2]

ggplot2 is "an open source implementation of the grammar of graphics for **R**."[3] Once we can internalize the grammar of graphics, creating plots will be an intuitive and artistic process rather than a mechanical one.

[1] A Layered Grammar of Graphics. Hadley Wickham. https://vita.had.co.nz/papers/layered-grammar.pdf

[2] The Grammar of Graphics. https://www.springer.com/gp/book/9780387245447.

[3] A Layered Grammar of Graphics. Hadley Wickham. https://vita.had.co.nz/papers/layered-grammar.pdf

There are five high level components of the layered grammar[4]. We will only cover the first two in this chapter.

1. **Defaults**:
    - Data
    - Mapping
2. **Layers**:
    - Data*
    - Mapping*
    - Geom
    - Stat
    - Position
3. Scales
4. Coordinates
5. Facets

18.2 Layers and defaults

Let's first get some data into our environment. We will use the `commute` dataset again.

```
library(tidyverse)

commute <- read_csv("data/gba_commute.csv")
```

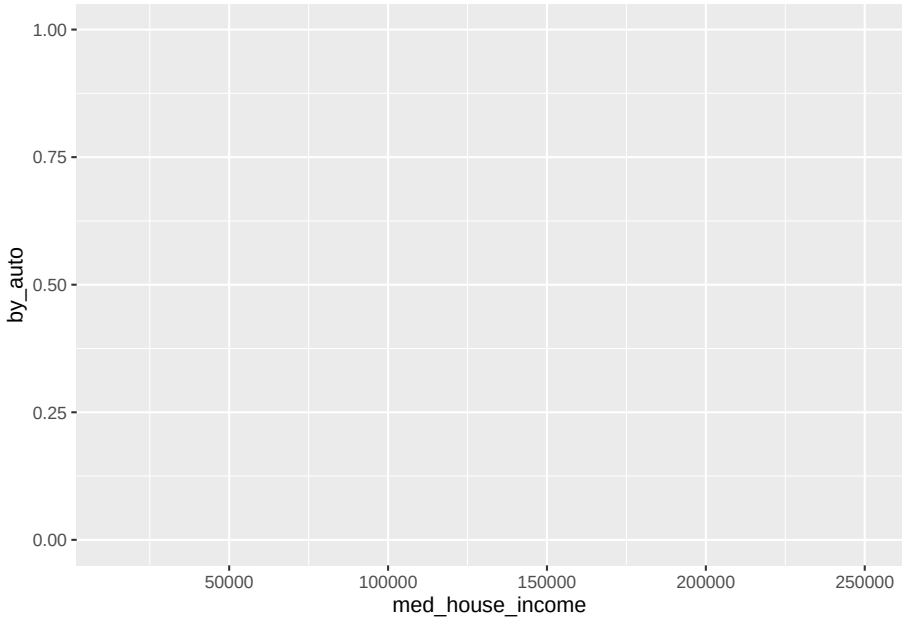
In the first chapter of this section we explored these principles but did not put a name to them. Recall that we can use `ggplot()` by itself and it returns a chart of nothing.

```
ggplot()
```

[4] A Layered Grammar of Graphics. Hadley Wickham. https://vita.had.co.nz/papers/layered-grammar.pdf

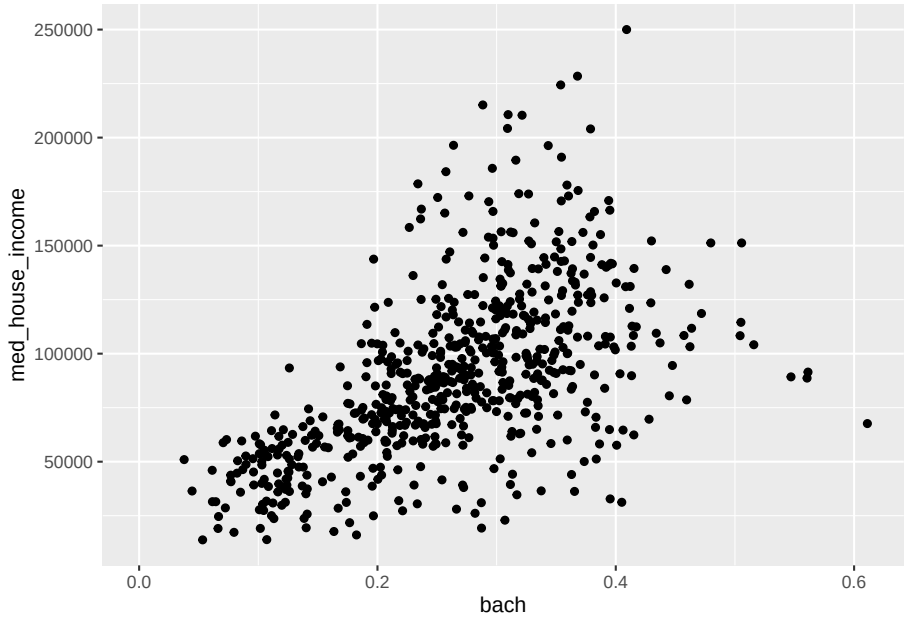
This is because we have not specified any of the defaults. In order for us to plot anything at all, we need to specify what (the data object) will be visualized, which features (the aesthetic mappings), and how (the geoms). When we begin to specify our x and y aesthetics the scales are interpreted.

```
ggplot(commute, aes(med_house_income, by_auto))
```



The final step is to add the geom layer which will inherit the data, aesthetic mappings, scale, and position while the `geom_*()` layer dictates the geometry.

```
ggplot(commute, aes(bach, med_house_income))+
  geom_point()
```



While this is the most common way you might define a ggplot, you should also be aware of the fact that each layer can stand on its own without you defining any of the defaults in the `ggplot()` call. Each geom inherits the defaults from `ggplot()`, but each `geom_*()` also has arguments for `data`, and `mapping`, providing you with increased flexibility.

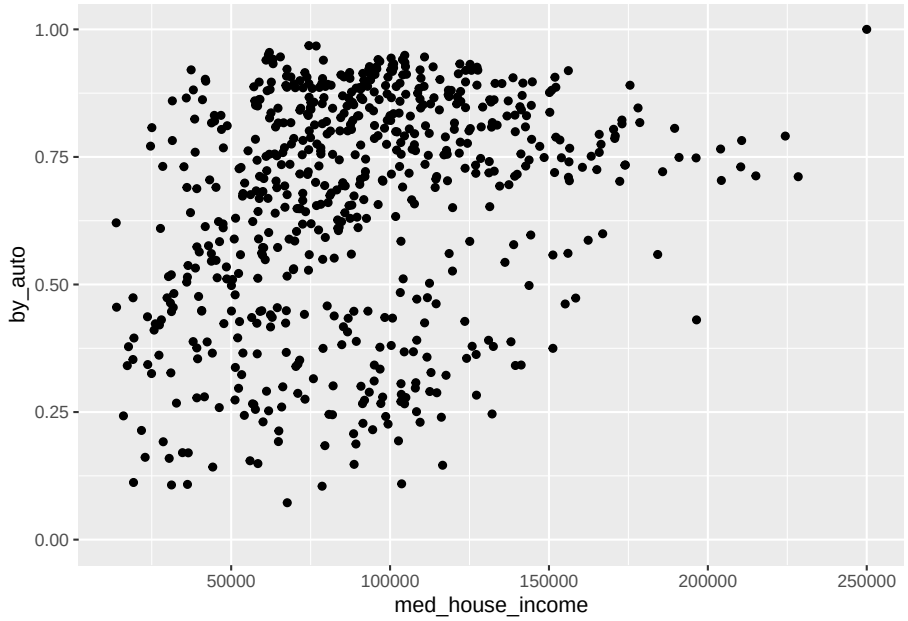
> Note: the `geom_*()`s have the data as the second argument so either put the data there or name the argument explicitly. The choice is yours. Choose wisely!

What happens if we provide all of this information to `geom_point()` and entirely omit `ggplot()`?

```
geom_point(aes(med_house_income, by_auto), commute)
#> mapping: x = ~med_house_income, y = ~by_auto
#> geom_point: na.rm = FALSE
#> stat_identity: na.rm = FALSE
#> position_identity
```

We see that we do not have the plot, but we do have all of the information required of a layer is printed out to the console. If we add an empty ggplot call ahead of the layer, we will be able to create the plot.

```
ggplot() +
  geom_point(aes(med_house_income, by_auto), commute)
```



Being able to specify different data objects within each layer will provie to be extraordinarily helpful when we begin to work with spatial data, or plotting two different data frames with the same axes, or any other creative problem you wish to solve.

Chapter 19

Visualizing Trends and Relationships

We now have a language for creating graphics. Next we must build the intuition of which plots to build and when. We will cover the most basic of visualizations starting with univariate followed by bivariate plots. We will then discuss ways to extend visualizations beyond two variables and some simple principles of design.

In most cases a data analysis will start with a visualization. And that visualization will be dictated by the characteristics of the data available to us. In intro to statistics you probably learned about the four types of data which are: nominal and ordinal, together referred to as *categorical*; interval and ratio, together referred to as *numerical* We're going to contextualize these in R terms where *categorical* is `character` and *numerical* is `numeric`.

Categorical and numeric have different are treated differently and thus lead to different kinds of visualizations. When we refer to categorical or character, we are often thinking of groups or a label. In the case where we don't have a quantifiable numeric value, we often count those variables.

Throughout this chapter we will use another ACS data with variables focused towards housing. This file lives at `data/acs-housing.csv`.

```
library(tidyverse)

acs <- read_csv("data/acs-housing.csv")
```

19.1 Univariate visualizations

There is always a strong urge to begin creating epic graphs with facets, shapes, colors, and hell, maybe even three dimensions. But we must resist that urge! We *must* understand the distributions of our data before we start visualizing them and drawing conlcusions. Who knows, we may find anomalies or errors in the data cleaning process or even collection process. We should always begin by studying the indivual variable characteristics with univariate visualizations.

> Note that univariate visualizations are for numeric variables

There a couple of things that we are looking for in a numeric univariate visualization. In the broadest sense, we're looking at characterizations of central tendency, and spread. When we create these visualizations we're trying to answer the following questions:

- Where is the middle?
- Is there more than one middle?
- How close together are our points?
- Are there any points very far from the middle?
- Is the distribution flat? Is it steep?

In exploring these questions we will rely on three types of plots:

1. Histogram
2. Density
3. Box plot

Each type of plot above serves a different purpose.

19.1.1 Histogram

We have already created a number of histograms already but it is always good to revisit the subject. Histograms puts our data into `n` buckets (or bins, or groups, or whatever your stats professor called them), counts the number of values that fall into each bucket, and use that frequency count as the height of each bar.

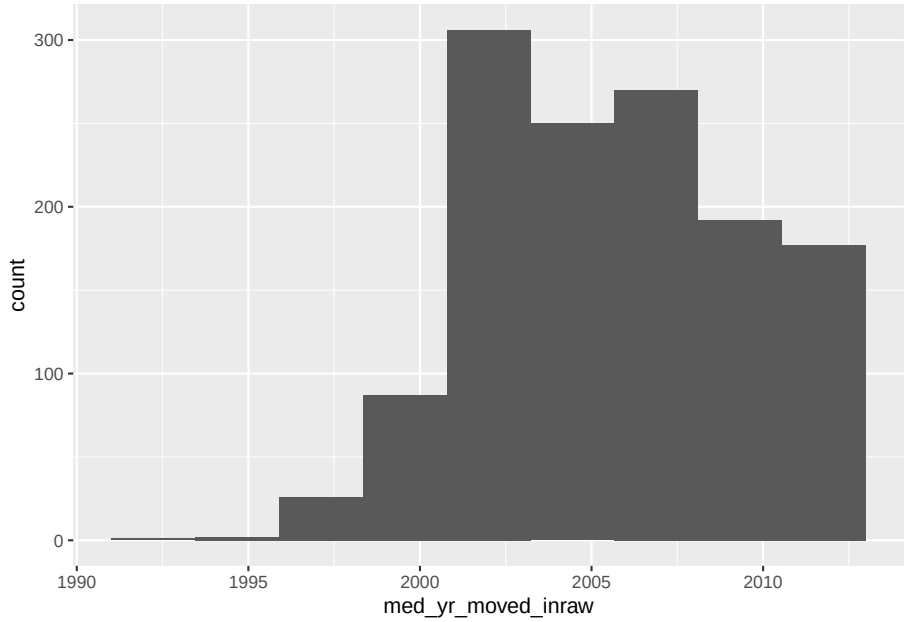
The true benefit of the histogram is that it is the easiest chart to consume by the layperson. But the downside is that merely by changing the number of bins, the distribution can be rather distorted and it is on you, the researcher and analyst, to ensure that there is no miscommunication of data.

When we wish to create a histogram, we use the `geom_histogram(bins = n)` geom layer. Since it is a univariate visualization, we only specify one aesthetic mapping—in this case it is `x`.

Let's look at the distribution of the `med_yr_moved_inraw` column for an example.

- Create a histogram of `med_yr_moved_inraw` with 10 bins.

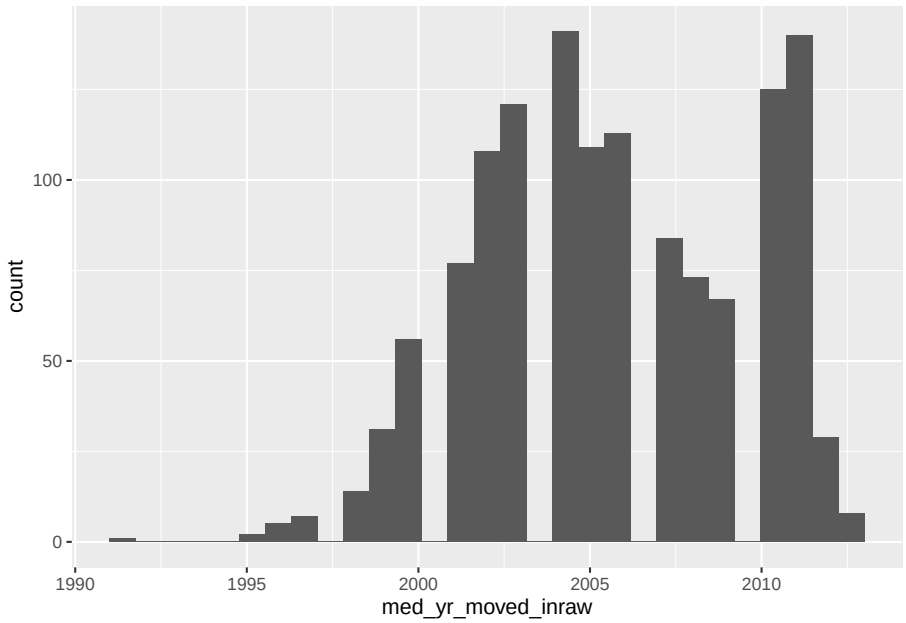
```
ggplot(acs, aes(med_yr_moved_inraw)) +
  geom_histogram(bins = 10)
```



This histogram is rather informative! We can see that shortly after 2000, there was a steep increase in people moving in. Right after 2005 we can see that number tapering off—presumably due to the housing crises that begat the Great Recession.

Now, if we do not specify the number of bins, we get a very different histogram.

```
ggplot(acs, aes(med_yr_moved_inraw)) +
  geom_histogram()
```



The above histogram shows gaps in between buckets of the histogram. On a first glance, we would assume that there may be missing data or some phenemonon in the data recording process that led to some sort of missingness. But that isn't the case! If we count the number of observations per year manually, the story becomes apparent.

> Note: I am using the base R function `table()` to produce counts. This produces a class `table` object which is less friendly to work with. Using `table()` rather than count serves two purposes: 1) you get to learn another function and 2) the printing method is more friendly for a bookdown document.

```
(moved_counts <- table(acs$med_yr_moved_inraw))
#>
#> 1991 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009
#>    1    2    5    7   14   31   56   77  108  121  141  109  113   84   73   67
#> 2010 2011 2012 2013
#>  125  140   29    8

glue::glue("There are {length(moved_counts)} unique values")
#> There are 20 unique values
```

> The glue function provides a way to create strings by combining R expressions and plain text. More in the appendix.

This above tells us something really important and explains why our histogram is all wonky. Our histogram looks the way it does because we have specified more bins than there are unique values! The moral of the story is that when creating a histogram, be thoughtful and considerate of the number of bins your are using—it changes the whole story.

19.1.2 Density Function plot

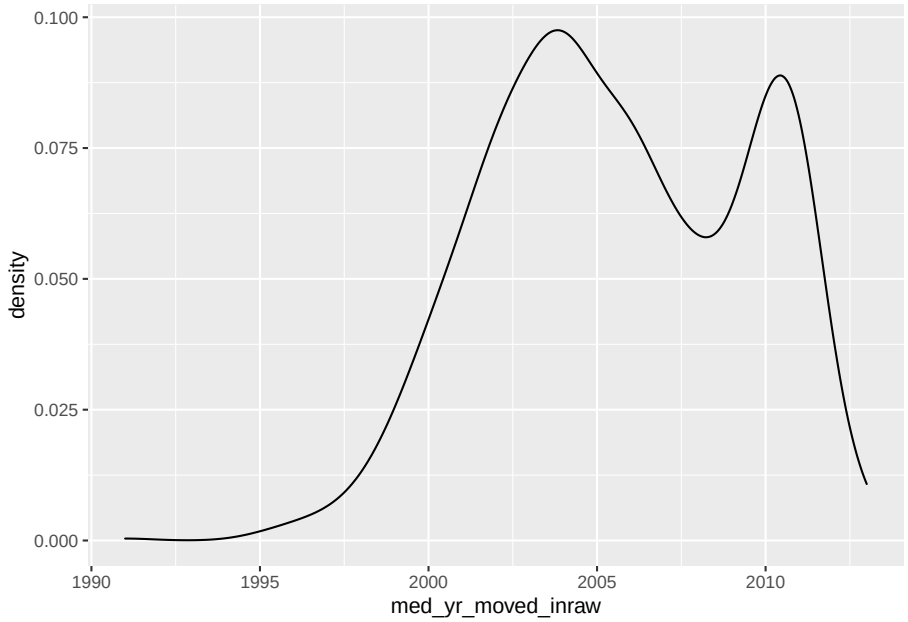
Histograms are a fairly straight-forward chart that provides illustrates the distribution of a sample space. The histogram does not provide a fine grain picture of what the underlying distribution looks like. When we are concerned with understanding the underlying shape of a distribution we should use a **kernel density plot** (aka density plot).

The density plot represents a variable over a continuous space and by doing so creates a much better visual representation of the underlying distribution shape with all of its curves.

Like a histogram, we only provide a single variable to the aesthetic mappings. The geom layer for a density distribution is `geom_density()`.

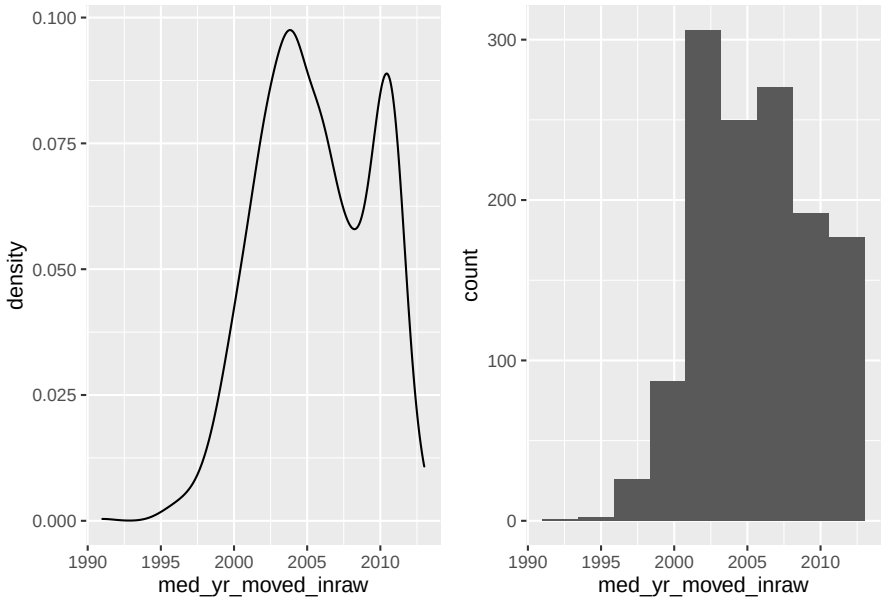
- Create a density plot of `med_yr_moved_inraw`

```
ggplot(acs, aes(med_yr_moved_inraw)) +
  geom_density()
```



Now compare the histogram to the density plot.

- Which do you feel does a better job illustrating the shape of the distribution?
- Which do you think is more interpretable?



19.1.3 Boxplots

The boxplot is the third univariate visualization we will cover. Unlike histograms and density plot, the box plot's power comes from being able to effectively illustrate outliers and the general spread of a variable.

There are five elements that make the boxplot:

1. Minimum
2. First quartile (25th percentile)
3. Median (50th percentile)
4. Third quartile (75th percentile)
5. Maximum

When creating a boxplot, the definition of minimum and maximum change a little bit. We are defining the minimums and maximums *without* the outliers. And in the context of a boxplot the outliers are determined by the **IQR** (inner

quartile range). The IQR is different between the third and first quartile. We then take the IQR and *add* it to the third quartile to find the upper bound and then subtract the IQR from the first quartile to find the lower bound.

$IQR = Q3 - Q1$

$Minimum = Q1 - IQR$

$Maximum = Q3 + IQR$

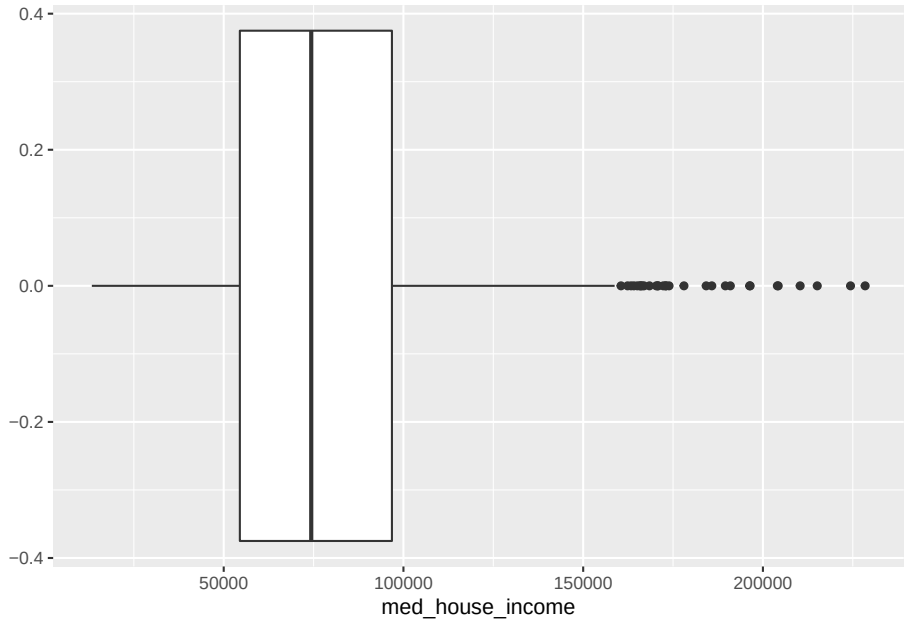
> Note that this is a naive approach to defining an outlier. This is not a hard and fast rule of what is considered an outlier. There are many considerations that should go into defining an outlier other than arbitrary statistical heuristics. Be sure to have a deep think before calling anything an outlier.

Any points that fall outside of that the minimum and maximum are plotted individually to give you an idea of any *potential* outliers.

To create a boxplot we use the `geom_boxplot()` function.

- Create a boxplot of `med_house_income`

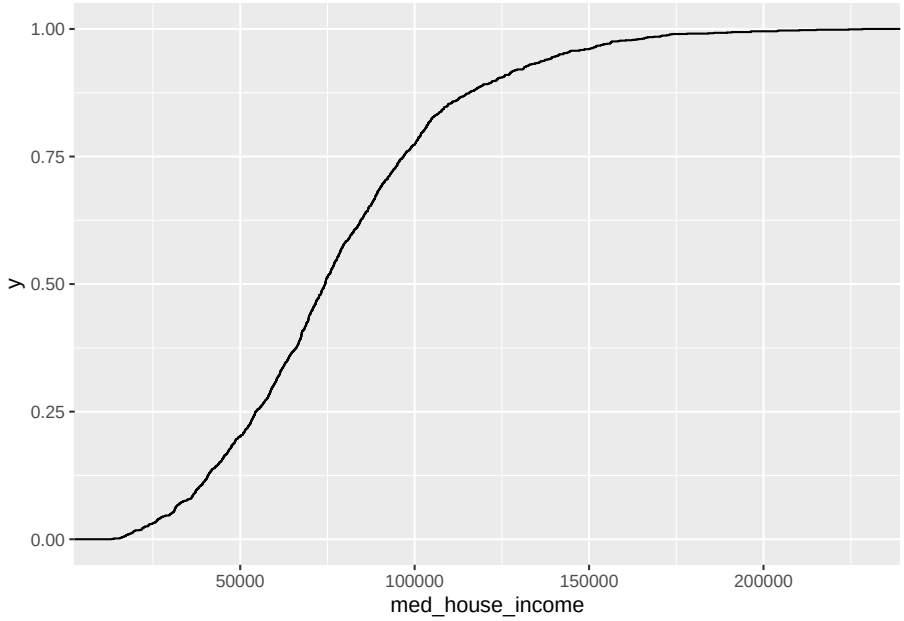
```
ggplot(acs, aes(med_house_income)) +
  geom_boxplot()
```



From this boxplot, what can we tell about median household income in Massachusetts?

19.1.4 Empirical Cumulative Distribution Function (ECDF)

```
ggplot(acs, aes(med_house_income)) +
  geom_step(stat = "ecdf")
```

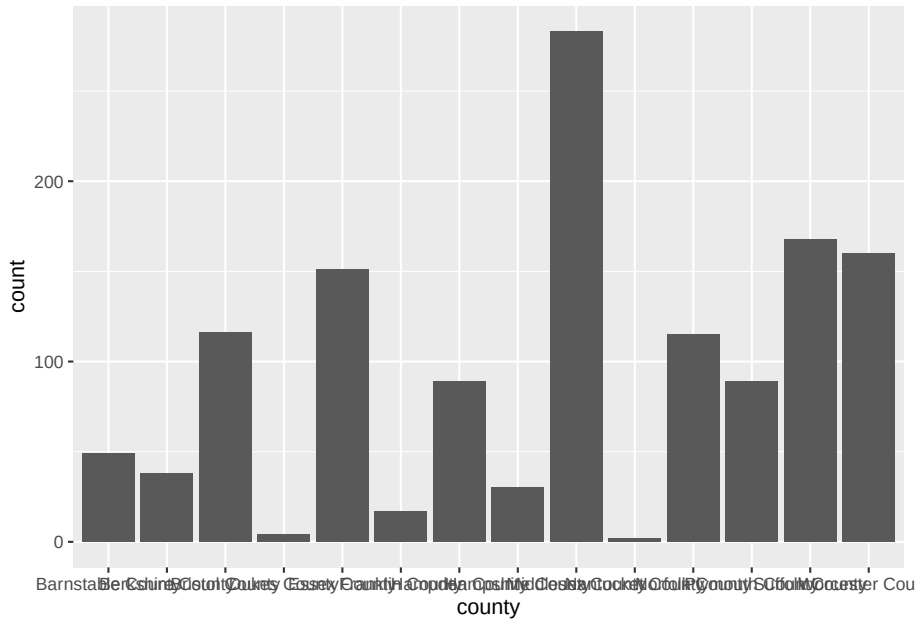


19.1.5 Barchart

The last univariate chart we will touch on is the bar chart. When we are faced with a single categorical variable there is not much that we can do to summarize it. The approach is to identify the frequency or relative frequency with which each level (unqiue value) occurs. This is essentially a histogram but for values which cannot have ranges and where order does not matter—though we may be interested in ordering our values.

To create a bar chart of categorical features we simply provide that feature to our aesthetic mapping and add `geom_bar()` layer

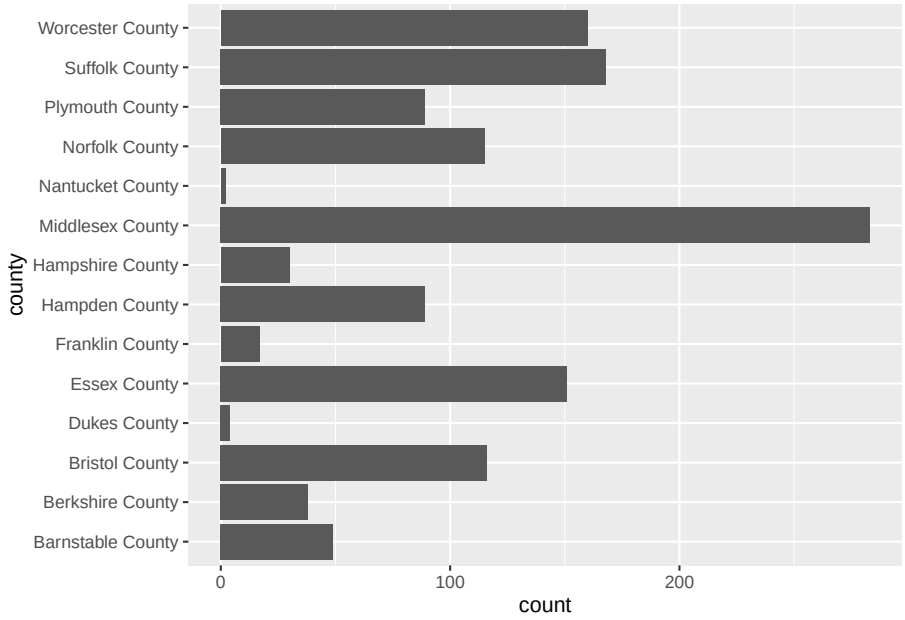
```
ggplot(acs, aes(county)) +
  geom_bar()
```



I'm sure you're looking at this chart and thinking something along the lines of "I can't read a single label, this is awful." And yeah, you're totally right. In general when creating a bar chart it is better to to flip the axes. The main justification for fliping the axes is so that we can read the labels better. In addition to making the labels more legible, by flipping the axes, the comparisons between bars is perceivedly easier.

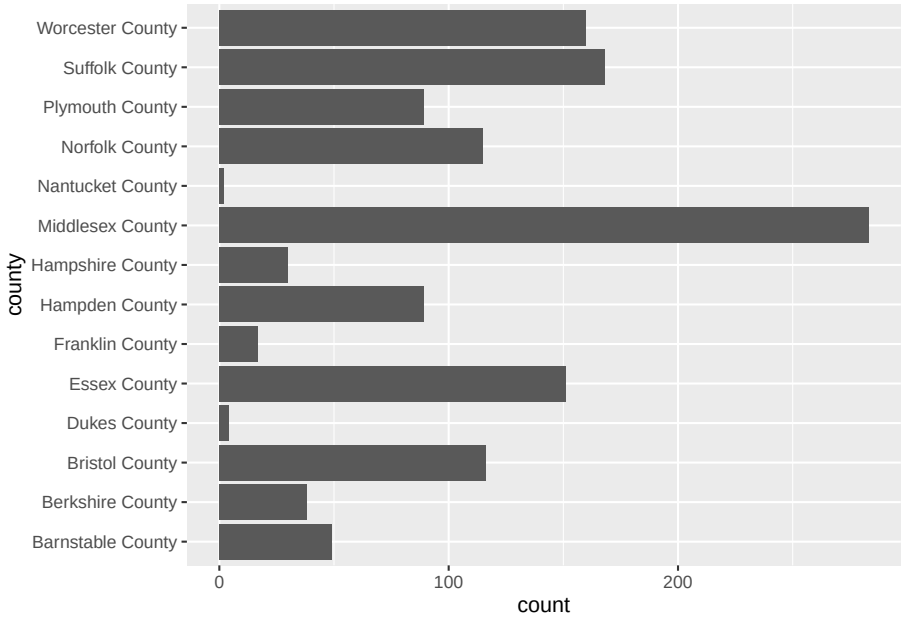
To flip the axes, we can map the `county` column to the `y` axis rather than `x` (which is done positionally).

```
ggplot(acs, aes(y = county)) +
  geom_bar()
```



If you find yourself in the situation where you have variables mapped to both the x and y columns we can add a `coor_flip()` layer to the plot which will handle the flipping for us.

```
ggplot(acs, aes(county)) +
  geom_bar() +
  coord_flip()
```



Be sure to keep `coord_flip()` in your back pocket! It is a rather handy function.

19.2 Bivariate visualizations

We are ready to introduce a second variable into the analysis. With bivariate relationships (two-variables) we are often looking to answer, in general, if one variable changes with another. But the way we approach these relationships is dependent upon the type of variables we have to work with. We can can either be looking at the bivariate relationship of

- 2 numeric variables,
- 1 numeric variable and 1 categorical,
- or 2 categorical variables.

19.2.1 Two Numeric Variables

19.2.1.1 Scatter plot

When confronted with two numeric variables, your first stop should be the scatter plot. A scatter plot positions takes two continuous variables and plots each point at their (x, y) coordinate. This type of plot illustrates how the two variables change with each other—if at all. It is exceptionally useful for

pinpointing linearity, clusters, points that may be disproportionately distorting a relationship, etc.

Scatter plots are useful for asking questions such as "when x increases how does y change?" Because of this natural formulation of statistical questions—i.e. we are always interested in how the x affects the y—we plot the variable of interest vertically along the y axis and the independent variable along the horizontal x axis.

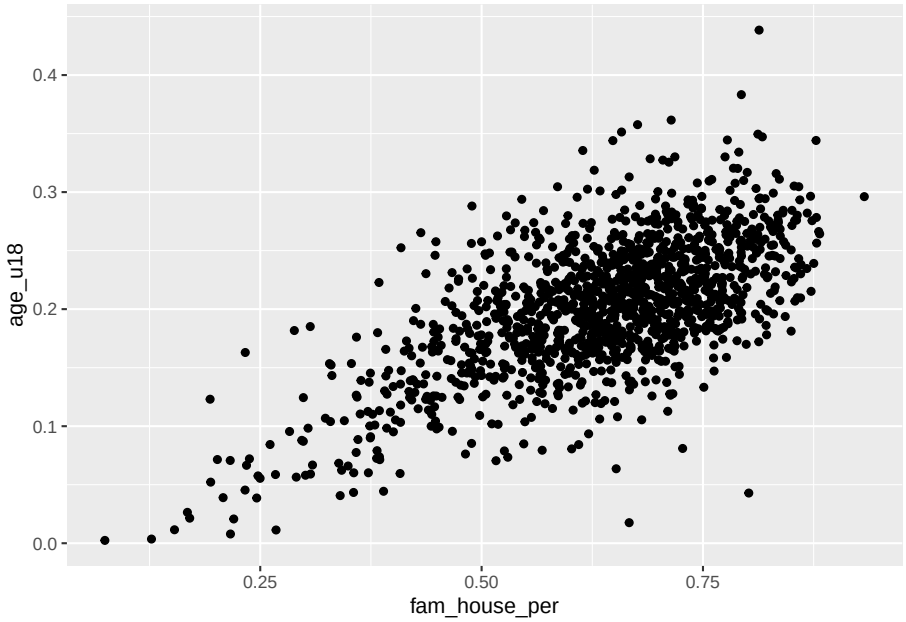
Take for example the question "how does the proportion of individuals under the age of eighteen increase with the number of family households?"

Using a scatter plot, we can begin to answer this question! Notice how in the formulation of our question we are asking how does y change with x. In this formulation we should plot the `fam_house_per` against the `age_u18` column.

> Note: when plotting *against* something. We are plotting x *against* y.

Recall that to plot a scatter plot we use the `geom_point()` layer with an x and y aesthetic mapped.

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_point()
```



The above scatter plot is useful, but there is one downside we should be aware of and that is the number of points that we are plotting. Since there are over

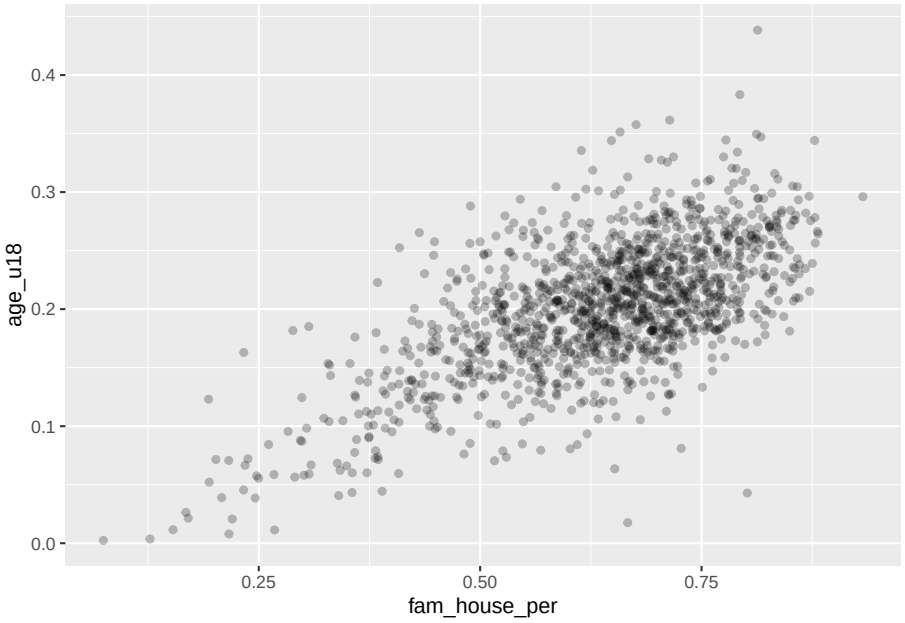
1,400 points—as is often the case with big data—they will likely stack on top of each other hiding other points and leading to a dark uninterpretable mass! We want to be able to decipher the concentration of points as well as the shape.

> When there are too many points to be interpretable this is called overplotting

To improve the visualization we have a few options. We can make each point more transparent so as they stack on top of eachother they become darker. Or, we can make the points very small so that as they cluster they become a bigger and darker mass.

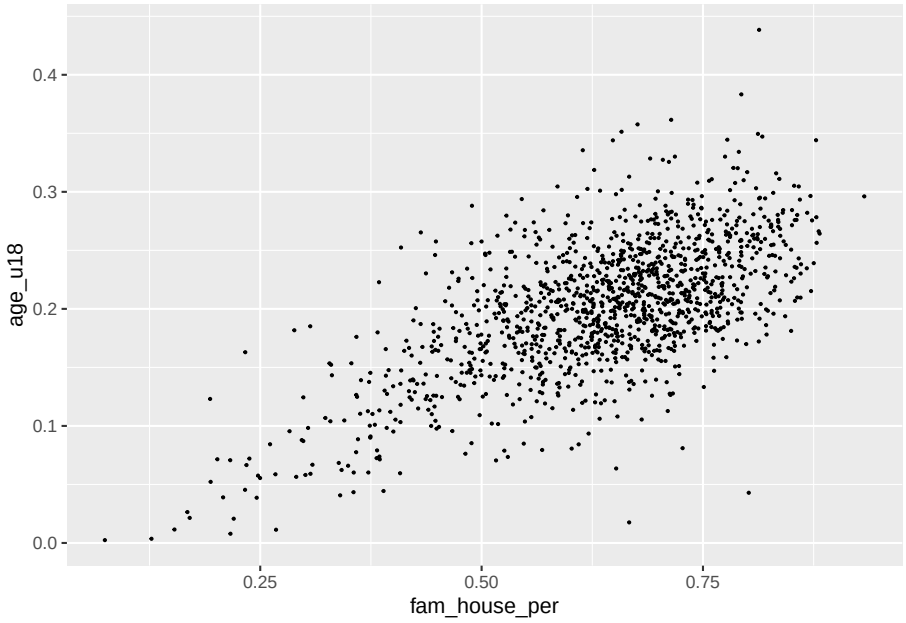
To implement these stylistic enhancements we need to set some aesthetic arguments inside of the geom layer. In order to change the transparency of the layer we will change the `alpha` argument. `alpha` takes a value from 0 to 1 where 0 is entirely transparent and 1 is completely opaque. Try a few values and see what floats your boat!

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_point(alpha = 1/4)
```



Alternatively, we can change the size (or even a combination of both) of our points. To do this, change the `size` argument inside of `geom_point()`. There is not a finite range of values that you can specify so experimentation is encouraged!

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_point(size = 1/3)
```



> Remember when deciding the `alpha` and `size` parameters your are implementing stylistic changes and as such there are no *correct* solution. Only marginally better solutions.

19.2.1.2 Hexagonal heatmap

Scatter plots do not scale very well with hundreds or thousand of points. When the scatter plot becomes a gross mass of points, we need to find a better way to display those data. One solution to this is to create a heatmap of our points. You can think of a heatmap as the two-dimension equivalent to the histogram.

The heatmap "divides the plane into rectangles [of equal size], counts the number of cases in each rectangle", and then that count is then used to color the rectangle[1]. An alternative to the rectangular heatmap is the hexagonal heatmap. The hexagonal heatmap has a few minor visual benefits over the rectangular heatmap. But choosing which one is better suited to the task it up to you!

The geoms to create these heatmaps are

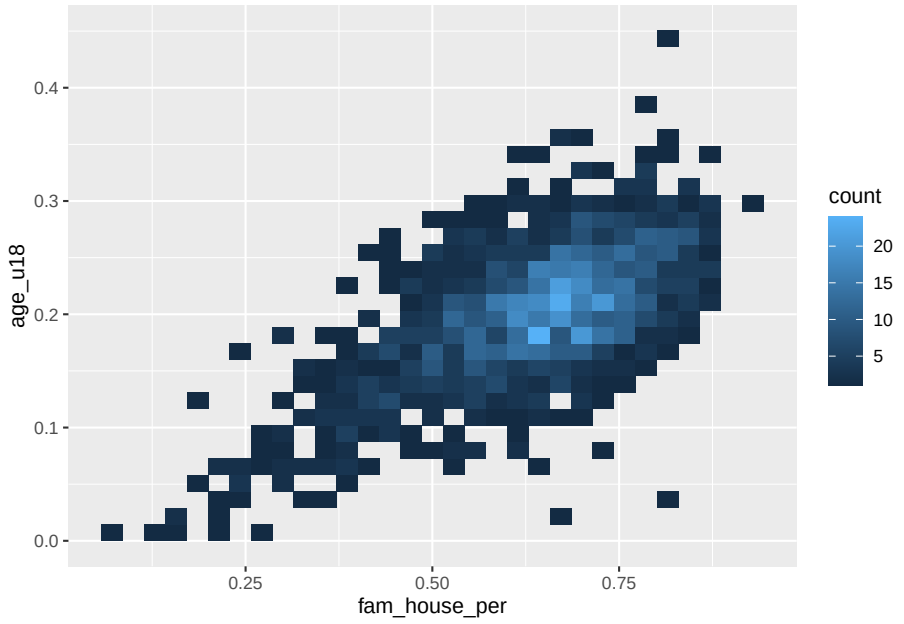
- `geom_bin2d()` for creating the rectangular heatmap and

[1] `geom_bin2d()`. https://ggplot2.tidyverse.org/reference/geom_bin2d.html

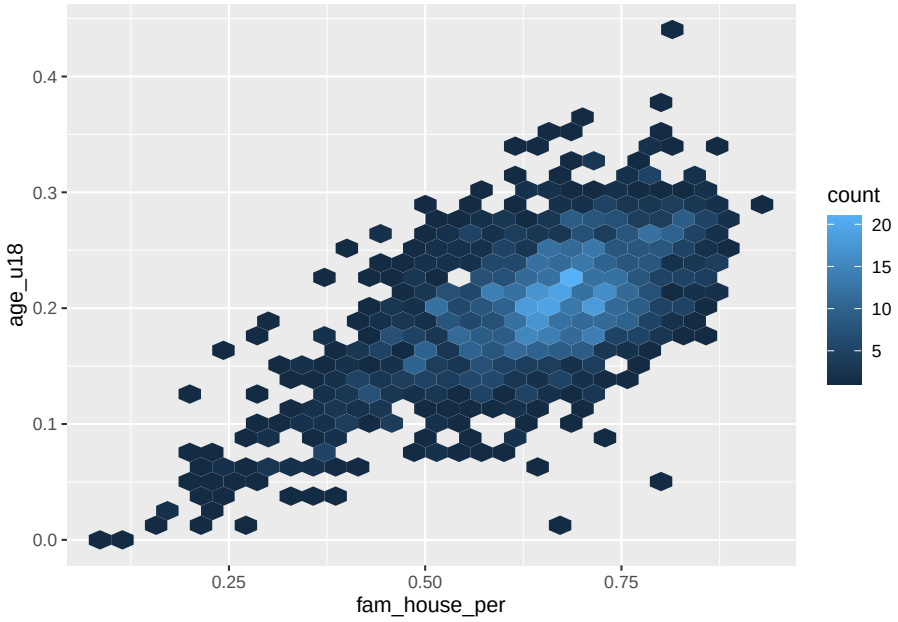
- `geom_hex()` for a hexagonal heatmap.

- Convert the above scatter plot into a heat map using both above geoms.

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_bin2d()
```



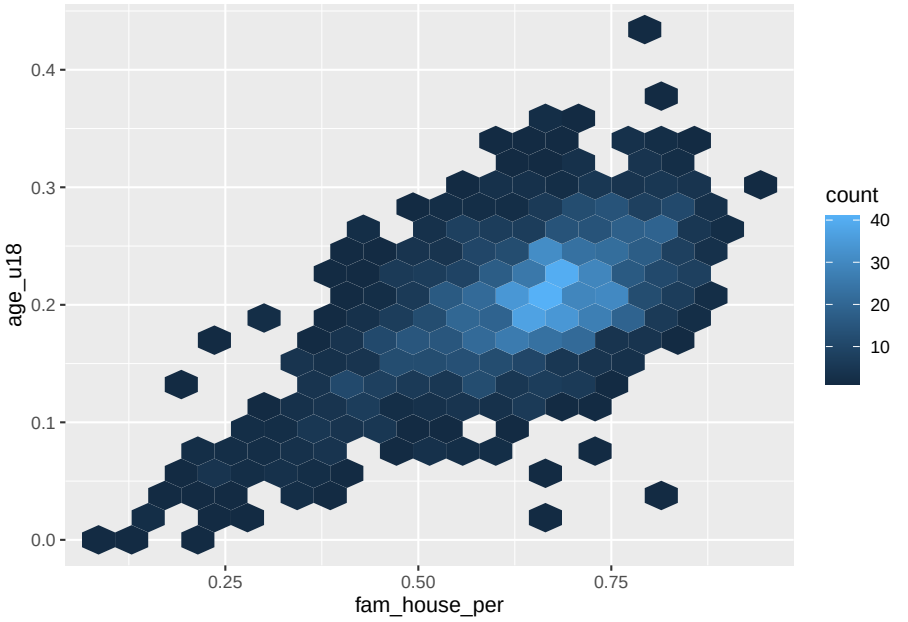

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_hex()
```



Just like a histogram we can determine the number of bins that are used for aggragating the data. By adjusting the `bins` argument to `geom_hex()` or `geom_bin2d()` we can alter the size of each hexagon or rectangle. Again, the decision of how many bins to include is a trade-off between interpretability and accurate representation of the underlying data.

- Set the number of `bins` to 20

```
ggplot(acs, aes(fam_house_per, age_u18)) +
  geom_hex(bins = 20)
```



19.2.2 One numeric and one categorical

The next type of bivariate relationship we will encounter is that between a numeric variable and a categorical variable. In general there are two lines of inquiry we might take. The first is similar to our approach of a single numeric variable where we are interested in measures of centrality and spread but are further interested in how those characteristics change by category (or group membership). The second seeks to compare groups based on some aggregate measure of a numeric variable.

As an example, imagine we are interested in evaluating the educational attainment rate by county in the Greater Boston Area. We can take the approach of ranking the educational attainment rate by the median or average. Or, we could also try and evaluate if the counties differ in the amount of variation.

```
gba_acs <- acs %>%
  filter(toupper(county) %in% c("SUFFOLK COUNTY", "NORFOLK COUNTY", "MIDDLESEX COUNTY"))
```

We will explore different techniques for addressing both lines of inqury.

19.2.2.1 Ridgelines

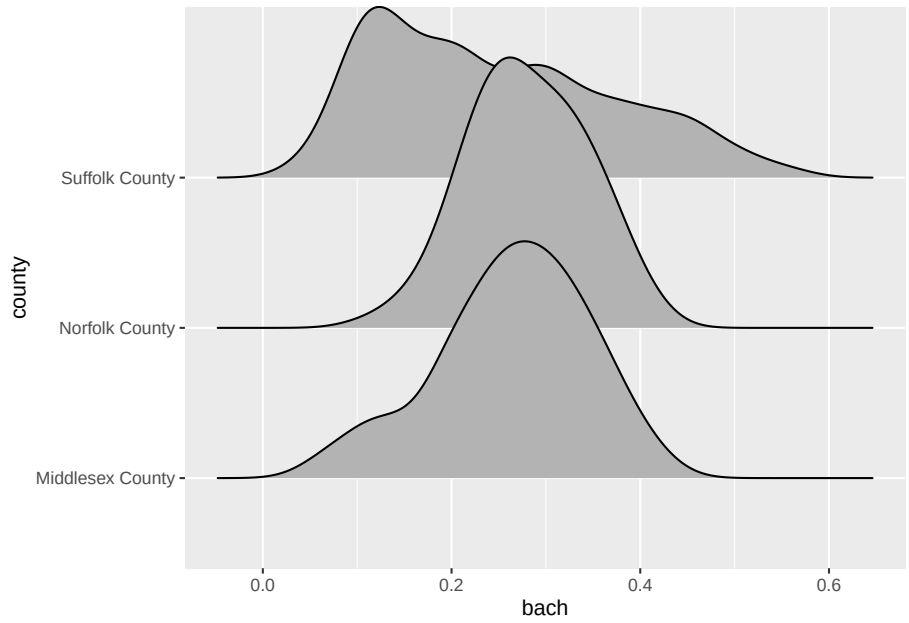
Ridgelines are a wonderful method for visualizing the distribution of a numeric variable for each level in a categorical variable. Ridgeline plot a density curve

for each level in your categorical variable and stacks them vertically. In doing so, we have a rather comfortable way to assess the shape of each level's distribution.

To plot a ridgeline, we need to install the package `ggridges` and use the function `ggridges::geom_density_ridges()`.

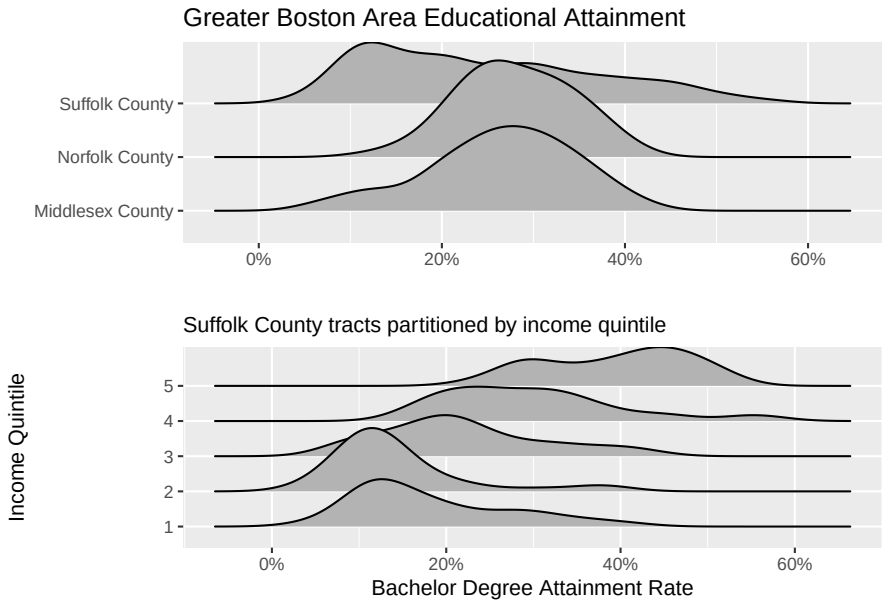
> Reminders: install packages with `install.packages("pkg-name")`. The expression `ggridges::geom_density_ridges()` is used for referencing an exported function from a namespace (package name). The syntax is *`pkgname::function()`*.

```
ggplot(gba_acs, aes(bach, county)) +
  ggridges::geom_density_ridges()
```



The ridgeline plot very clearly illustrates the differences in distributions within the `county` variable. From this plot, we can tell that Suffolk County has rather extreme variation in the Bachelor's degree attainment rate. And when compared to Norfolk and Middlesex counties, it becomes apparent that the median Suffolk County attainment rate falls almost 20% lower.

A plot such as the above may lead one to investigate further. Suffolk County is large and contains every single neighborhood of Boston from Back Bay, to Mission Hill, and Roxbury. We can drill down further into Suffolk County by identifying income percentiles and plotting those as well.



Ridgelines are the perfect tool for exploring changes in variation among different groups. Before you run an ANOVA visualize the variation of your variables with a ridgeline plot first!

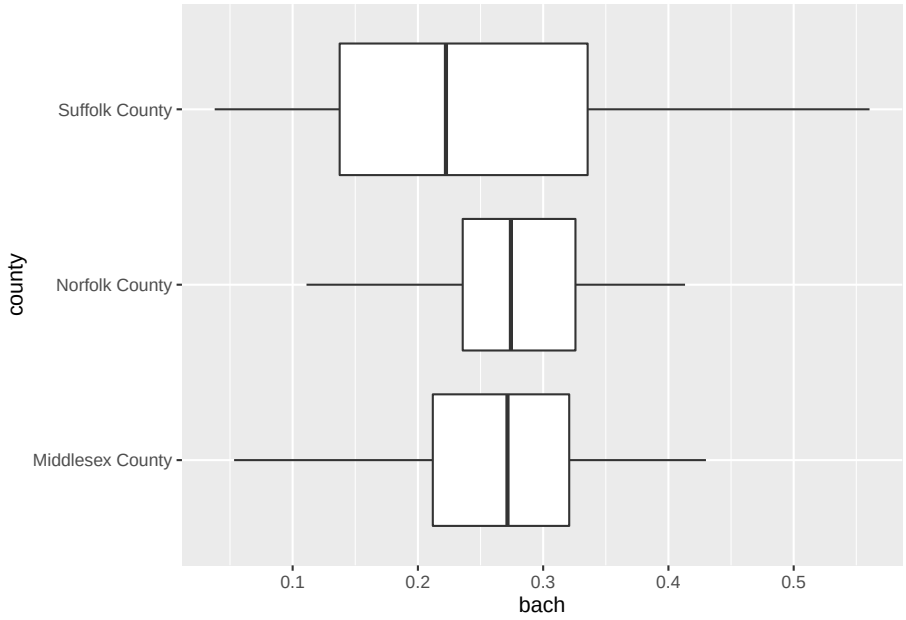
19.2.2.2 Boxplot

It's time to come back to the boxplot. The boxplot is indeed wonderful for a single variable. But much in the same way that multiple density plots is what makes the ridgeline fantastic, so does multiple box plots!

Again, when using the boxplot we are not as concerned about the *shape* of the distribution but rather *where* the data are. The boxplot is extremely useful for identifying skewness and potential outliers.

We can look at the distribution of educational attainment using a boxplot just like above. The only difference is the use of the `geom_boxplot()`.

```
ggplot(gba_acs, aes(bach, county)) +
  geom_boxplot()
```



19.2.2.3 Barchart

We've already used the barchart to plot counts of categorical variables. But they are also useful for visualizing summary values of each categorical variable.

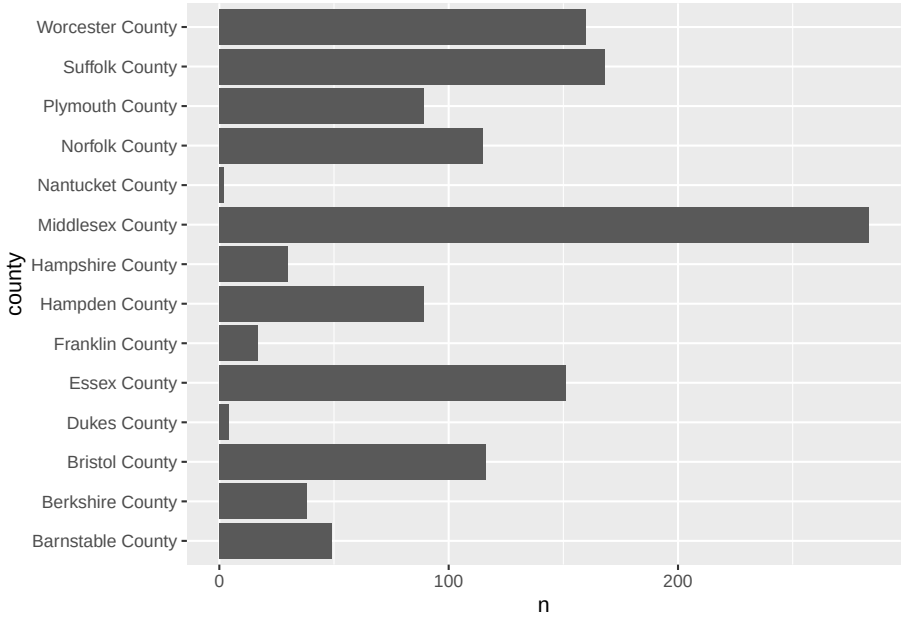
For example, if we were to plot the number of observations per county we can use our knowledge of `summarise()` to recreate the values.

```
county_counts <- acs %>%
  group_by(county) %>%
  summarise(n = n())

county_counts
#> # A tibble: 14 x 2
#>    county               n
#>    <chr>            <int>
#>  1 Barnstable County    49
#>  2 Berkshire County     38
#>  3 Bristol County      116
#>  4 Dukes County          4
#>  5 Essex County        151
#>  6 Franklin County      17
#>  7 Hampden County       89
#>  8 Hampshire County     30
```

```
#>  9 Middlesex County    283
#> 10 Nantucket County      2
#> 11 Norfolk County      115
#> 12 Plymouth County      89
#> 13 Suffolk County      168
#> 14 Worcester County    160
```

Now that we've counted the number of points per value, we can plot that using either `geom_bar()` and setting `stat = "identity"` *or* we can use `geom_col()`. I prefer the latter.

```
ggplot(county_counts, aes(n, county)) +
  geom_col()
```



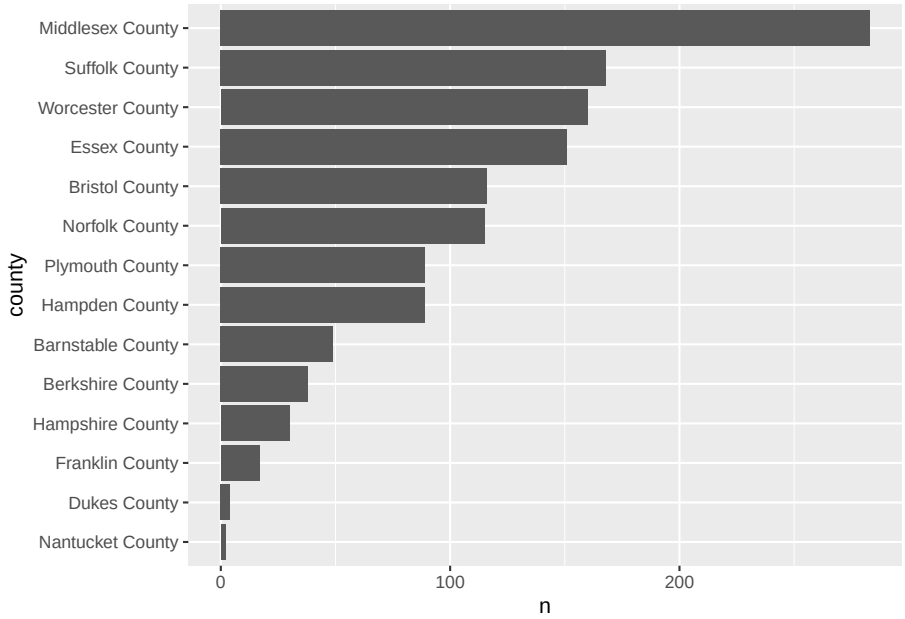
Here's the thing, barcharts, and horizontal barcharts in particular are phenomenal for ranking. But ggplot2 doesn't order the bars for us. We need to do that on our own. To do so, we will use our knowledge of `mutate()` and a new function `forcats::fct_reorder()`.

`fct_reorder()` is a function used for reordering categorical variables by some other numeric variable. In our case, we want to reorder `county` by `n`. So, within a `mutate()` function call we will alter `county` to be the value of the output `fct_reorder(county, n)`.

> If you are confused by `fct_reorder()`, remember to check out the help documentation with `?fct_reorder()`.

The modified `county_counts` tibble can then be piped into our `ggplot()`.

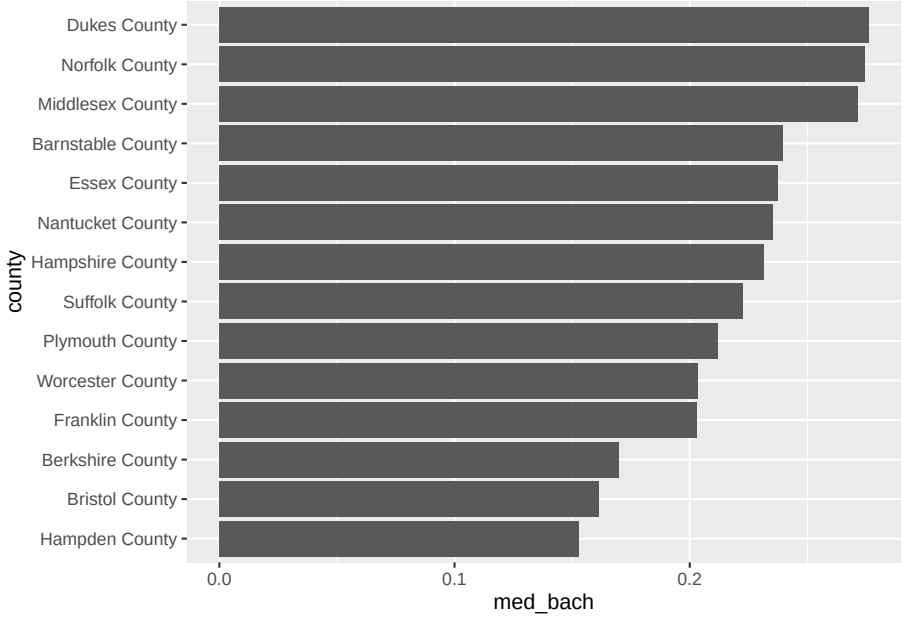
```
county_counts %>%
  mutate(county = fct_reorder(county, n)) %>%
  ggplot(aes(n, county)) +
  geom_col()
```



This is a pattern that you will follow rather frequently—particularly when you need to rank variables. Now knowing the number of observations is useful, but we really want to use the barchart for visualizing some value of importance. Let's continue the example of educational attainment but for the entirety of Massachusetts this time.

- Using `group_by()` and `summarise()`, calculate the median Bachelor degree attainment rate and call that column `med_bach`.
- Reorder `county` by `avg_bach`
- Create an ordered horizontal barchart of `avg_bach` by `county`.

```
acs %>%
  group_by(county) %>%
  summarise(med_bach = median(bach)) %>%
  mutate(county = fct_reorder(county, med_bach)) %>%
  ggplot(aes(med_bach, county)) +
  geom_col()
```



This brings us very naturally to our next type of plot: the lollipop plot.

19.2.2.4 Lollipop plot

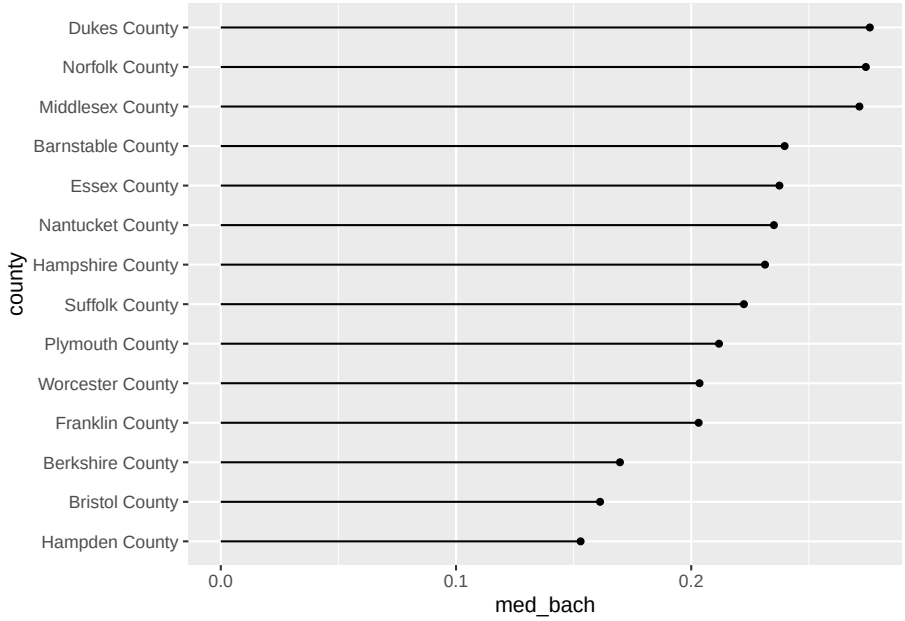
The lollipop plot is the barchart's more fun cousin. Rather than a big thick bar we plot the summary value with a big point and draw a thin line back to it's respective axis' intercept. We can either manually create the lollipop using a creative combination of geoms, or use a geom incorporated in another package. I will almost always recommend that you don't recreate something if you do not have to. As such, we will use the `ggalt::geom_lollipop()` function.

> Remember: *`pkgname::function()`*. If you do not have `pkgname` installed, install it with `install.packages("pkgname")`.

We can copy our previous barplot code and only replace the geom to produce a lollipop plot! Since we are keeping `med_bach` in the x position we will need to specify `horizontal = TRUE` in `geom_lollipop()`. This is a quirk of the geom but an easy one to get past. I recommend setting `horizontal = FALSE` to get a firmer understanding of what is happening. There is nothing quite like purposefully breaking your code to figure out what is happening!

```
acs %>%
  group_by(county) %>%
  summarise(med_bach = median(bach)) %>%
```

```
  mutate(county = fct_reorder(county, med_bach)) %>%
  ggplot(aes(med_bach, county)) +
  ggalt::geom_lollipop(horizontal = TRUE)
```



19.3 Review:

You've now built up a repetoire of different types of visualizations that you can use in your own analyses. You've built an intuition of what types of visualization are suitable given the types of variables at your disposal.

In the next chapter we will explore ways of improving upon the plots that we already know how to build. We will explore further the *Layered Grammar of Graphics* how how to improve upon our charts using scales, and facets, and breifly touch upon coordinates.

Chapter 20

Grammer of layered graphics II

We've developed a strong foundation for building charts from the ground up by specifying our **defaults** (data, and aesthetic mappings), and adding geom **layers**. In order to take our charts to the next level we need to familiarize ourselves with the other components of the *Layered Grammar of Graphics*: scales, coordinates, and facets.

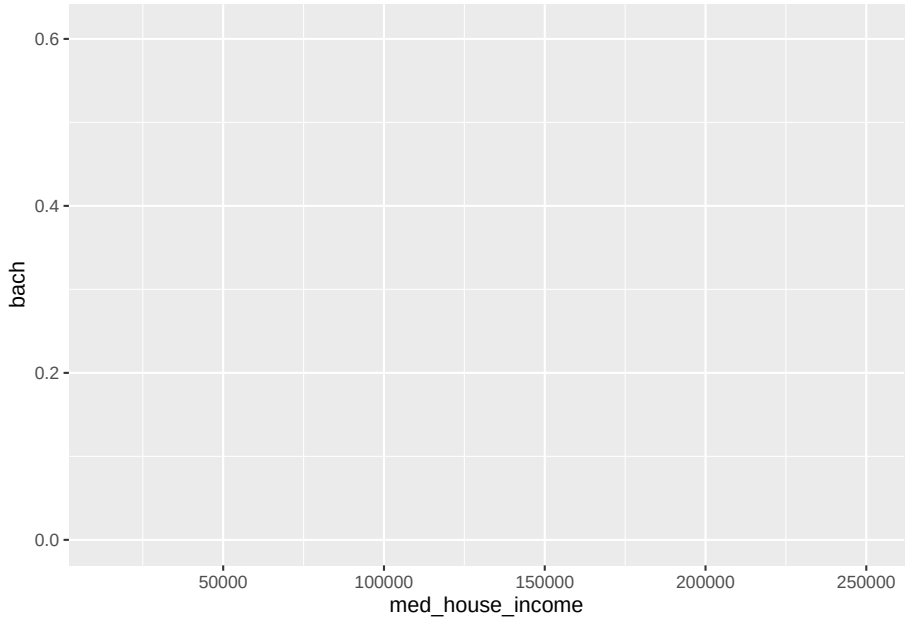
For these examples we will again return to our commute dataset. We will also recreate the two columns `hh_inc_quin` and `edu_attain`.

```
library(tidyverse)

commute <- read_csv("data/gba_commute.csv") %>%
  mutate(hh_inc_quin = ntile(med_house_income, 5),
         edu_attain = bach + master)
```

20.1 Scales

Recall from *Grammar of Layered Graphics I* that when we supply our aesthetic mappings our axes are filled out automatically.

```
(p <- ggplot(commute, aes(med_house_income, bach)))
```

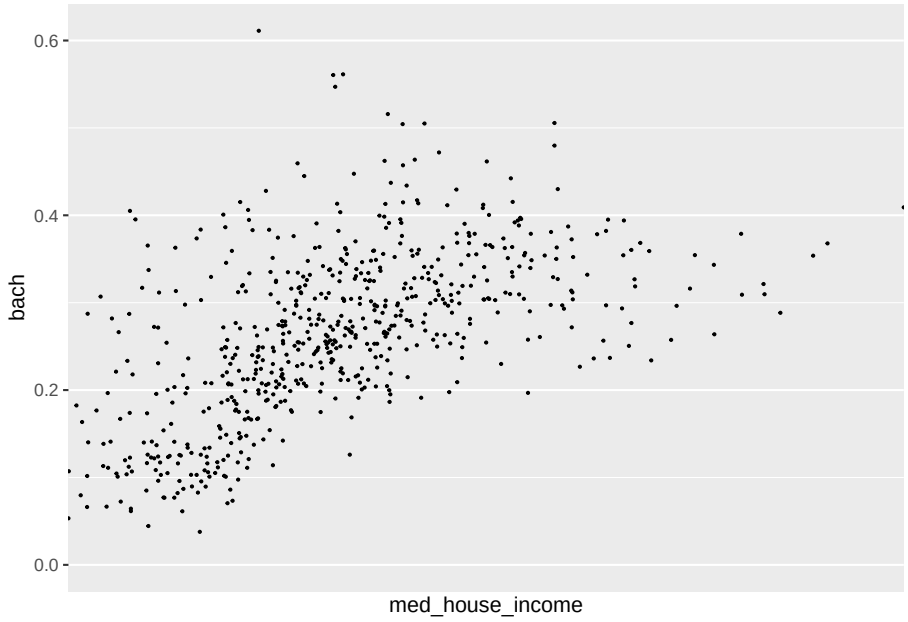


By specifying our defaults in the `ggplot()` call, we implicitly are providing the x and y axes. From those mappings, ggplot2 is able to identify the type of variable mapped to each aesthetic and its values. That inference makes it possible for us to plot without having to explicitly state what our axes are.

```
p <- ggplot(commute, aes(med_house_income, bach)) +
  geom_point(size = 1/3)
```

In the above chart, each column is being mapped as a continuous variable. We are able to manually specify what each scale type is by using the various `scale_*_type()` layers from ggplot2. These layers follow a general format of first specifying `scale` followed by which aesthetic we're scaling, and what data type. For example, to change the `med_house_income` axis to a discrete axis we can apply the layer `scale_x_discrete()`

```
p +
  scale_x_discrete()
#> Warning: Removed 8 rows containing missing values (geom_point).
```



In doing so we have lost the axis labels! That is because ggplot2 considers both integers and floating point (numbers with decimals) as continuous and categorical variables as being discrete.

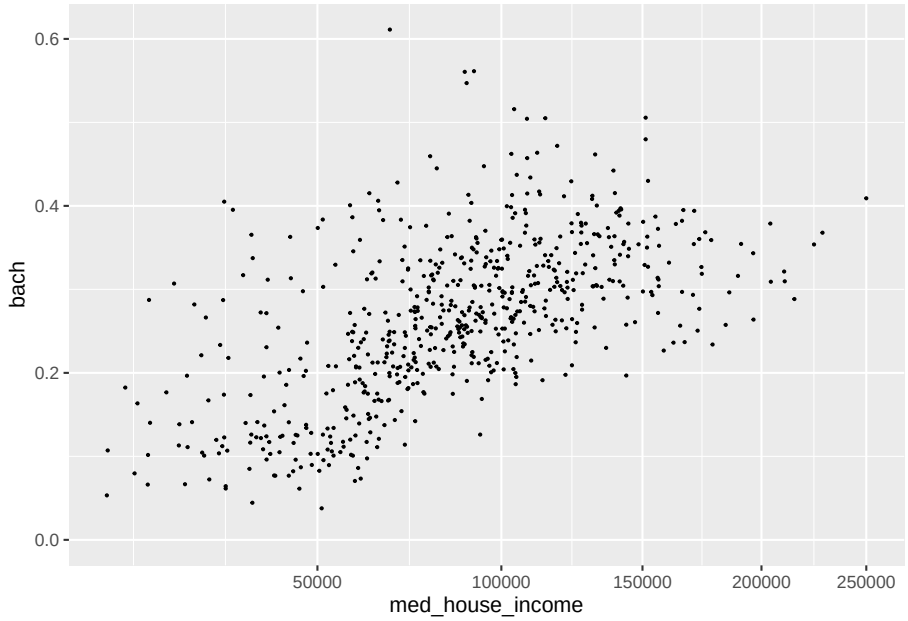
Nonetheless, we have a lot of functions at our disposal to alter the axes to our liking!

20.1.1 Transformations

In our data exploration, we will come across non-normal distributions of data. For example income is almost always right skewed and displays some sort of log-normal-ish behavior. We may not want to actually change to underlying values of that variable, but want to apply transformations for the purposes of visualization. In those cases, we can apply scale transformations.

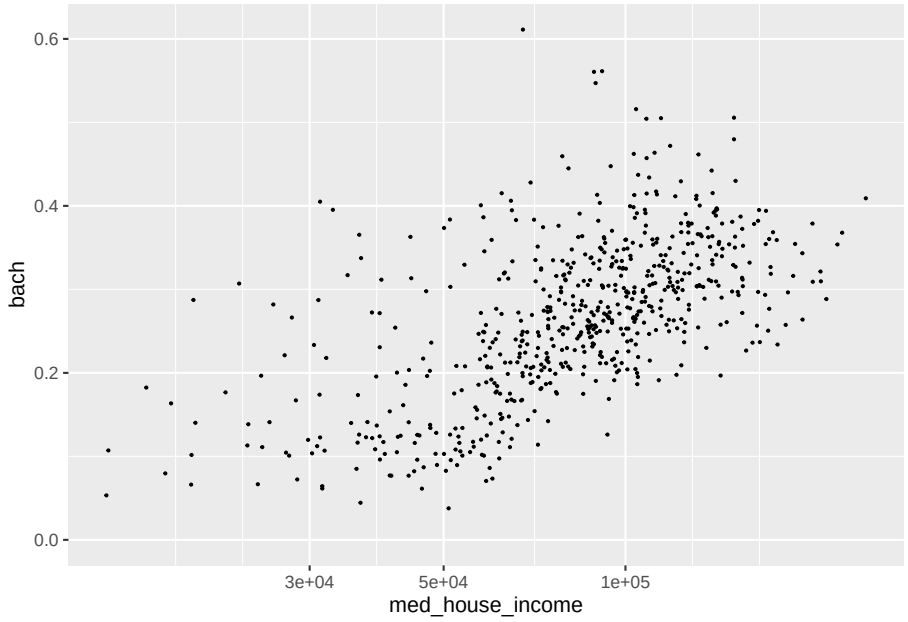
As an example, in our visualization of income and education there is a slight right skew to `med_house_income`. The graphic doesn't justify a logarithmic transformation, but may benefit from a sqrt transformation. We can apply this with `scale_x_sqrt()`.

```
p +
  scale_x_sqrt()
#> Warning: Removed 8 rows containing missing values (geom_point).
```



We can apply a log10 transformation as well with `scale_*_log10()`.

```
p +
  scale_x_log10()
#> Warning: Removed 8 rows containing missing values (geom_point).
```



This is an overcorrection. The slight upward arch in the original plotting is now inverted. Nonetheless, I hope the point is made.

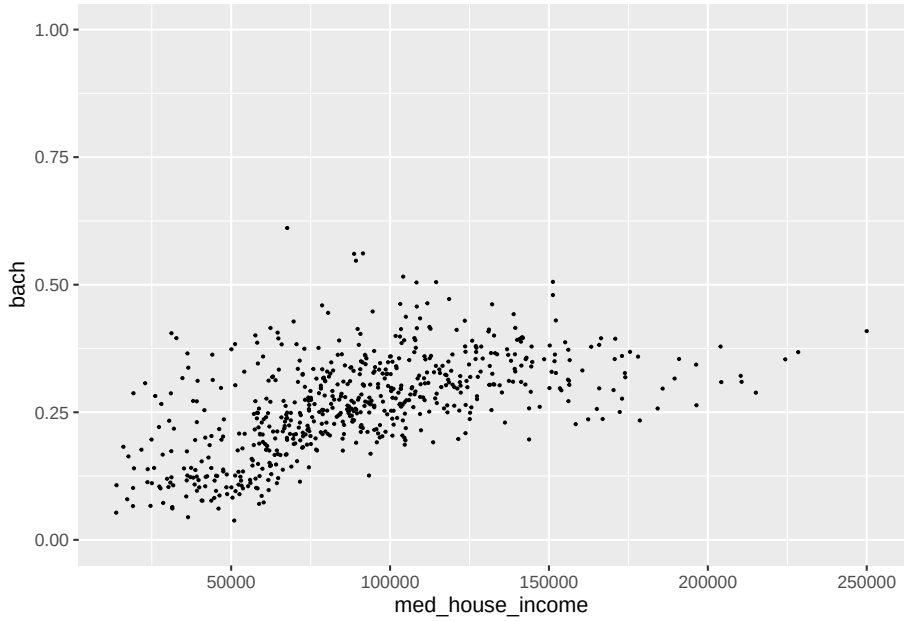
In addition to applying transformations, we will want to have more control over the limits of our graph. For example, say we want to have our y axis include all *possible* values of [0,1]. We can tell ggplot what range of values we want our axes to contain with `lims()`. `lims()` takes a name-value pair where the name is an aesthetic and the value is a numeric vector with two elements—the first being the value at the origin and the second being at the extent of the axis[1].

> Note: [0,1] means from 0 inclusive to 1 inclusive.

We can modify our y axis to have the limits of [0,1] by adding a `lims()` layer where we set the `y` aesthetic to `c(0, 1)`.

```
p +
  lims(y = c(0, 1))
#> Warning: Removed 8 rows containing missing values (geom_point).
```

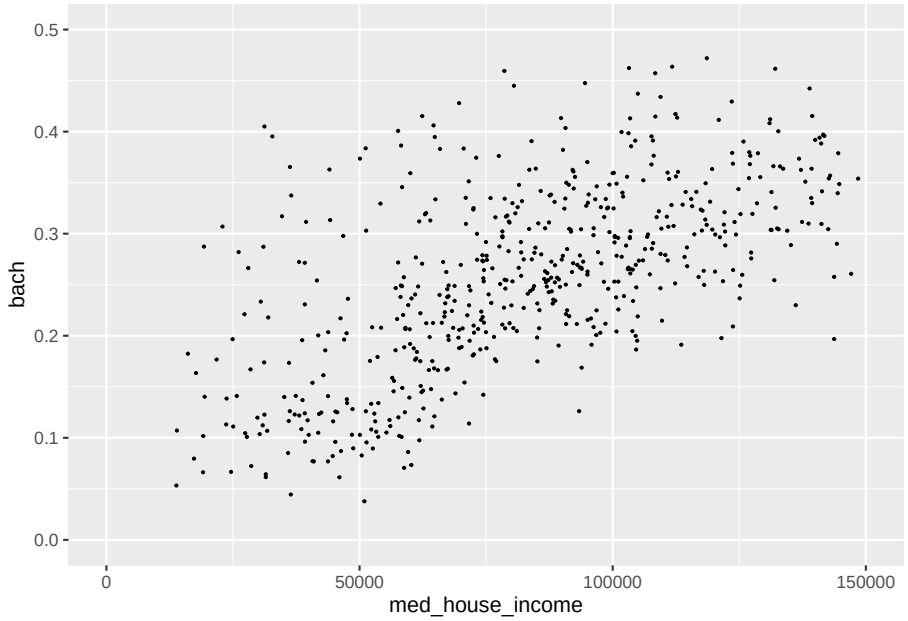
[1] https://ggplot2.tidyverse.org/reference/lims



The graph we get we when expand our y axis limits definitely contains a bit too much white space. But by expanding ths grid, we can see this sort of flattening out of education at around $150,000 while income still continues to increase. Perhaps if we omit tose values, the relationship may seem even stronger. Let's experiment with that.

- Set the x axis limit to be from [0, 150000]
- Set the y axis limits to be from [0, 0.5]

```
p +
  lims(
    x = c(0, 150000),
    y = c(0, .5)
    )
```



By changing the extent of our axes this relationship seems much more robust! Such a visualization could spur further validation of this ACS data.

> Remember, ACS data come from samples and sometimes those samples are small. Because of these small sample sizes, we may very well get values that are not properly representative. It is on you to decide whether or not you should include or exclude the values!

20.1.2 Labeling

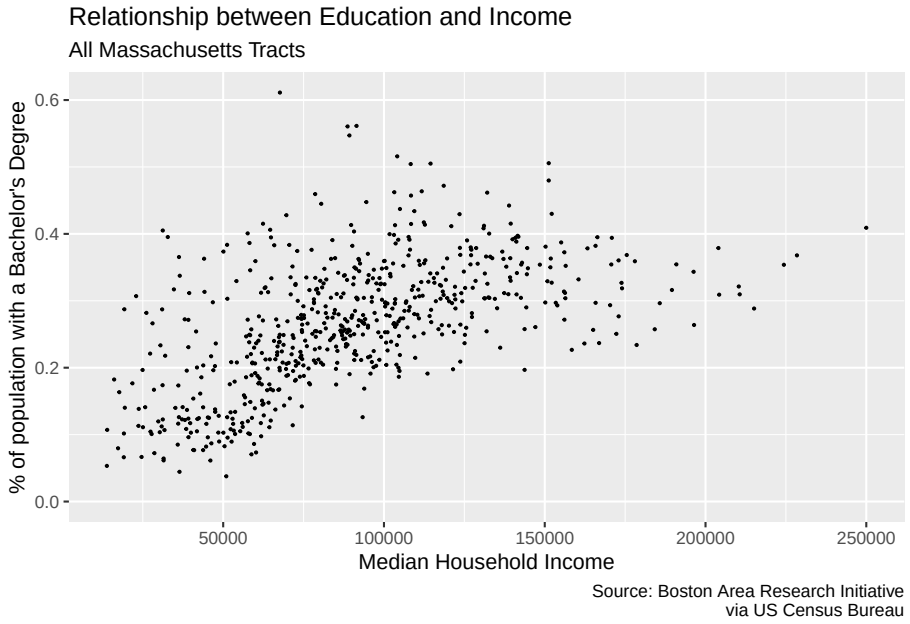
The plots we create, while lovely as they are, are somewhat lacking in the labeling department. I would put money on it that no publication would accept plots with labels such as the ones above for sole reason being that our axes titles and scale labels are hard to interpret.

We've already used it befofe but to be extra clear, to add titles and axis labels to our plots (not adjusting the scale labels) we use a `labs()` layer. With `labs()`, you can label any aesthetic you have mapped as well as adding a `title`, `subtitle`, `caption`, and a `tag`.

Let's add some titles and labels to our plot.

```
p <- ggplot(commute, aes(med_house_income, bach)) +
  geom_point(size = 1/3) +
    labs(
```

```
       y = "% of population with a Bachelor's Degree",
       x = "Median Household Income",
       title = "Relationship between Education and Income",
       subtitle = "All Massachusetts Tracts",
       caption = "Source: Boston Area Research Initiative\nvia US Census Bureau"
      )

p
```



Friends, it's looking pretty good. But there are just two more changes we need to make: our axes labels! The x and y axes labels are meant to illustrate dollar amounts and percentages but respectively. To change the *scale* labels. we will use some helper functions from the package `scales`

{scales} provides handy functions for taking a variable and altering the labeling to match some other format. In our case, we are interested in printing our `med_house_income` as in a dollar format, i.e. `2000` becomes `\$2,000`, and `bach` as a percentage, i.e. `.4` becomes `%40`. To alter our labels we will use `scales::dollar()`, and `scales::percent()` respectively.

> Isn't it nice how well named functions can be sometimes?

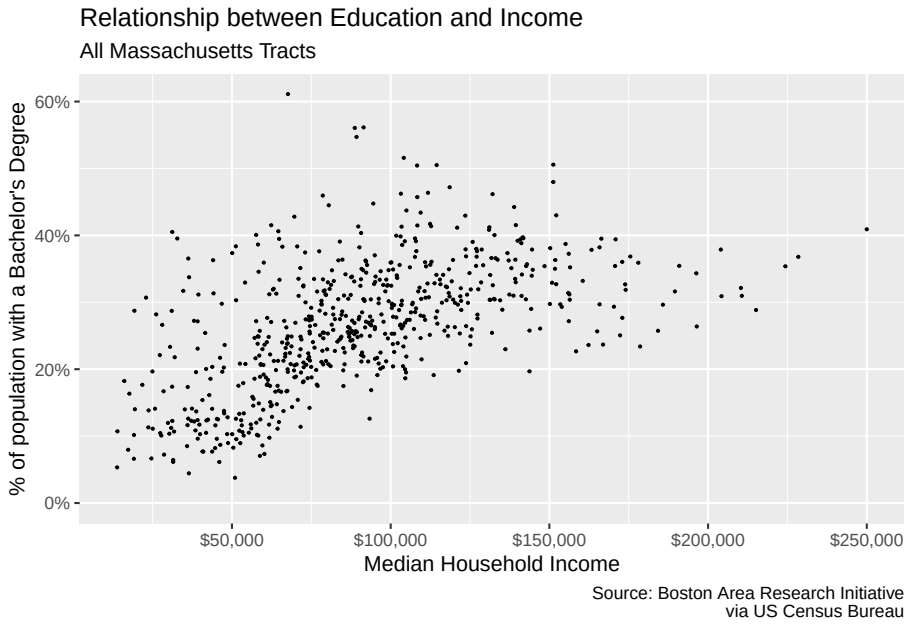
To produce the examples outlined above we would call the function as such:

```
scales::dollar(2000)
#> [1] "$2,000"
scales::percent(.4)
#> [1] "40%"
```

Now we have an understanding of *how* the function behaves, but where do we actually change the labels? This is where we come full cirlce back to our `scale_*_continuous()` layer. As we mentioned earlier, `ggplot()` will handle making the scales for us. But `ggplot()` doesn't know how we want to label our variables or how they should appear on the axes. And now the impetus is on us to make these changes manually. To change the axis labels we will specify which axis we are altering using the proper scale layer—i.e. `scale_y_` or `scale_x_`. Then, in each layer we set the `labels` argument to the respective labelling function we want—e.g. `scales::percent` and `scales::dollar`.

> Note: If you append parentheses like you normally would you will get an error. This case we want to ignore them because when they are present, R will try to evaluate that function. Rather, we are interested in providing the *function object* to the `labels` argument rather than provide it with a vector of output.

```
p +
  scale_x_continuous(labels = scales::dollar) +
  scale_y_continuous(labels = scales::percent)
#> Warning: Removed 8 rows containing missing values (geom_point).
```



In addition to being able to control the **defaults**, the **layers**, and now the **scales** you are well equipped to create and manipulate your own plots.

20.2 Coordinates

While you're likely to create 98% of your visualizations without ever manipulating the coordinates, it is still good knowledge to have!

As we have mentioned and alluded to, we are working in a two-dimensional space—meaning with x and y coordinates. When working in two-dimensions, thee cartesian plane is the natural choice for a coordinate reference system. In all of our plots, this has been the default. Behind the scenes, ggplot is essentially adding a `coord_cartesian()` layer to your plot.

> Think of this much of the same way that your scales are inferred.

If, however, we find the need to alter or manipulate the coordinate system the tools are available to us. We've actually already used one, `coord_flip()`. Like, with scales, all coordinate based functions are prefixed with `coord_()`. If you will need to use these coordinate layers, it will be to essentially change the aspect ratio of your plots.

We will encounter coordinates much more when we talk about spatial data. For now, though, all you need to know is that they exist and are a major underlying part of your plots.

20.3 Facets

The last portion of the grammar to visit is facetting. When we facet a plot we are creating what are called "small multiples", a term coined by the prominent Edward Tufte. A facetting, in other words, creates a graph for each unique level in a categorical variable. Think of this like a `group_by()` for plotting.

There are two types of facetting we can do: wrapped and grid. These are done with `facet_wrap()` and `facet_grid()` respectively. The reference documentation sums up the differences best:

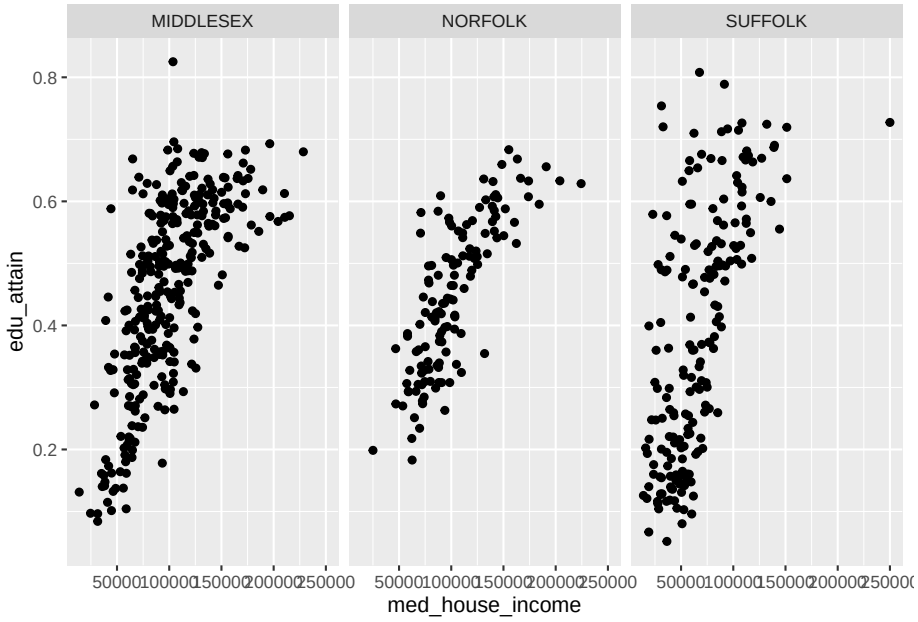
> "`facet_grid()` forms a matrix of panels defined by row and column faceting variables. It is most useful when you have two discrete variables, and all combinations of the variables exist in the data. If you have only one variable with many levels, try `facet_wrap()`."[2]

Let's look at `facet_wrap()` first. To create the facetting, we need to add `facet_wrap()` as a layer to our existing plot. There is only one argument that we are required to fulfill and that is the `facet` argument. `facet` expects a set of variables defined by the `vars()` function. `vars()` is a function used throughout the tidyverse to specify which columns are to be referenced used within the context of a the function it's being used in.

To recreate the above plot but facetting by county, we would add `facet_wrap(vars(county))` as a layer to our plot.

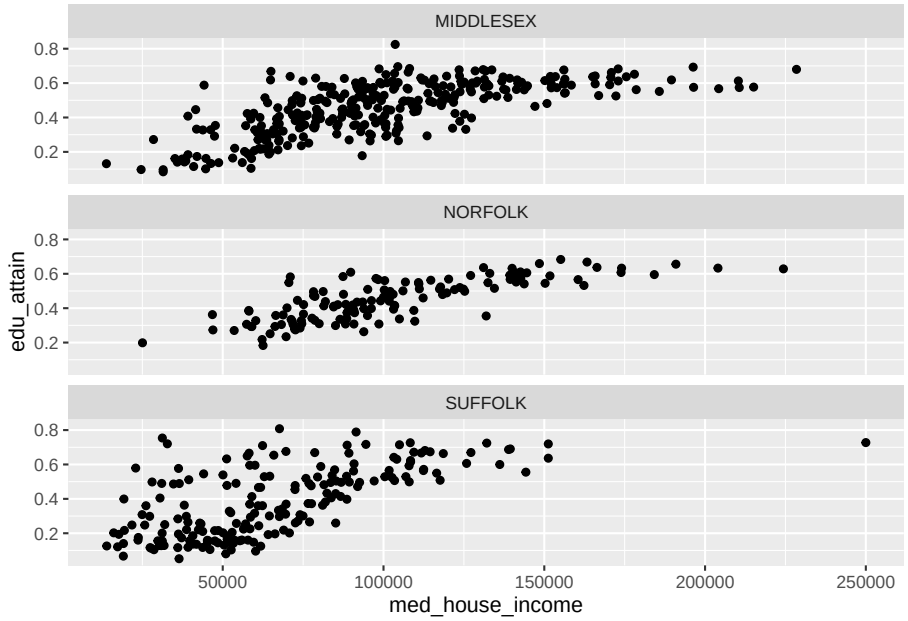
```
p <- commute %>%
  filter(!is.na(hh_inc_quin)) %>%
  ggplot(aes(med_house_income, edu_attain)) +
  geom_point()

p +
  facet_wrap(vars(county))
```

[2] https://ggplot2.tidyverse.org/reference/facet_grid.html



With `facet_wrap()` we are able to explicitly state how many rows or columns of plots there should be. The defaults may be nice, but it's always good to be explicit about our expecvtations! We set the `nrow` or `ncol` argument to do this. Since our above example defaulted to `ncol = 3`, let's try this with three rows.

```
p +
  facet_wrap(vars(county), nrow = 3)
```



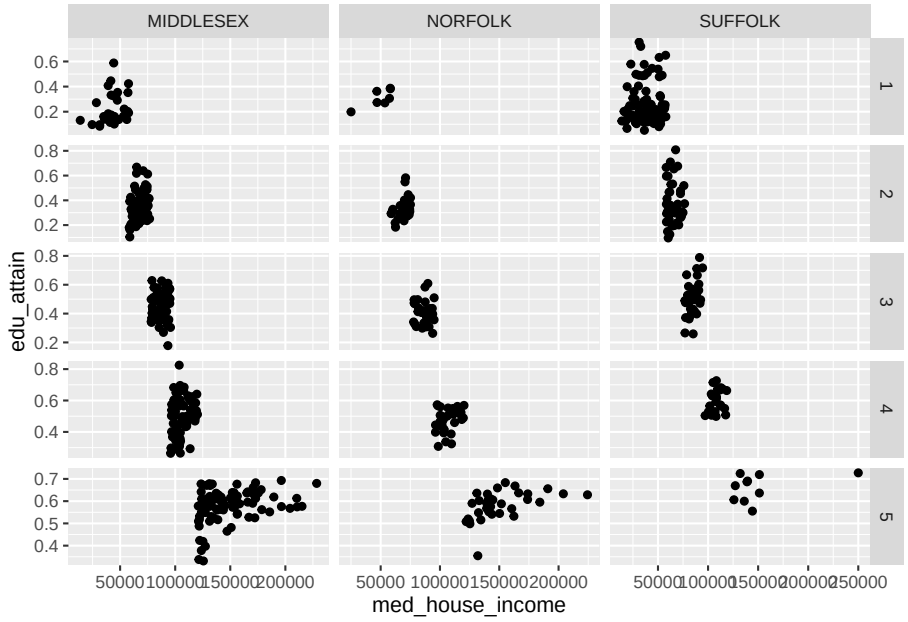
The grid works a little bit differenly. Rather than specifying which columns to facet on and the number of rows or columns, we can create a grid (or matrix) of small multiples. With `facet_grid()` we use the `rows` and `cols`.

We can recreate our above graphs by passing `vars(county)` to either `rows` or `cols`. But where `facet_grid()` shines is when you have data in every pairing of two categorical variables. For example we can we create a facet for each combination of `county` and `hh_inc_quin`.

```
p +
  facet_grid(cols = vars(county),
             rows = vars(hh_inc_quin))
```

When we create facets, each panel has shares the same scales. We can change this by setting the `scales` argument to one of `"free"`, `"free_x"`, or `"free_y"`. These in essence, "free up" the scales for each panel. We can choose to share the scales on the x axis by setting `scales = "free_y"` or vice versa.

```
p +
  facet_grid(cols = vars(county),
             rows = vars(hh_inc_quin),
             scales = "free")
```



Note: the behavior of `scales = "free"` changes behavior when set in the context of `facet_wrap()` vs `facet_grid()`. The former frees the scales for each panel. The latter frees the scales for either a column or row of panels.

It is also important to note that what we have gone through is by no means exhaustive. You should, at minimum, familiarize yourself with both `scale_*_reverse()`, and `scale_*_binned()` in your spare time. There are dozens, if not hundreds, of ggplot2 functions to suit your every whim. And, as you have already briefly seen, the ggplot function universe is not relegated to just ggplot2. There are many other packages which have built custom geoms and other enhancements that may benefit you.

Chapter 21

Visualizing beyond 2-dimensions

Over the duration of the last three chapters we have cultivated a fundamental understanding of the grammar of graphics and have discussed how to craft univariate visualizations and explore bivariate relationships. In those graphics we have not gone beyond two-dimensions. We only utilized two aesthetics to map. Howver, there is many more that we can incorporate into our visualizations which will in turn enable us to explore three or for variables at once.

> Note that these will not be 3D, but rather visualize three variables.

To improve our graphics we will utilize the color, shape, and size aesthetics, as well as faceting. Of course, this begs the question of which aesthetic do I choose? Well, that depends upon what type of data you will be visualizing. Each aesthetic serves different purposes and can be used for a different type of variable.

In general we can use the below mappings:

- color -> continuous or discrete
- shape -> discrete
- size -> continuous

21.1 Color

Let us first take a look at the use of color. Color is, after position, the easiest visual cue for we humans to distinguish (that viz book on my coffee table)

between. It is also a rather versatile visual cue as it can be used to address both continuous and discrete variables. We will first explore the use of color for discrete measurements. In this context, I do not necessarily mean discrete as in integers, but more or less groups. This is where there is not *necessarily* an order or scale implied in the data. It *can* however be indicative of order—think for example age groups. To explore the use of color for groups or discrete data, we will look at Boston ecometrics of social disorder as discussed previously (O'Brien 2015 CITE NEEDED). Ecometrics are stored in a file called `ecometrics.csv` the `data` directory. Read it in as `ecometrics`.

```
library(tidyverse)

ecometrics <- read_csv("data/ecometrics.csv")
```

At this point in your learning, I think it is appropriate to introduce you to a new package that can be used to quickly summarize and visualize your data. That is called `skimr`. Within the package there is a function called `skim()`. This package is really useful for quickly getting an understanding of a dataset as it provides useful summary statistics for each variable as well as a histogram for numeric columns.

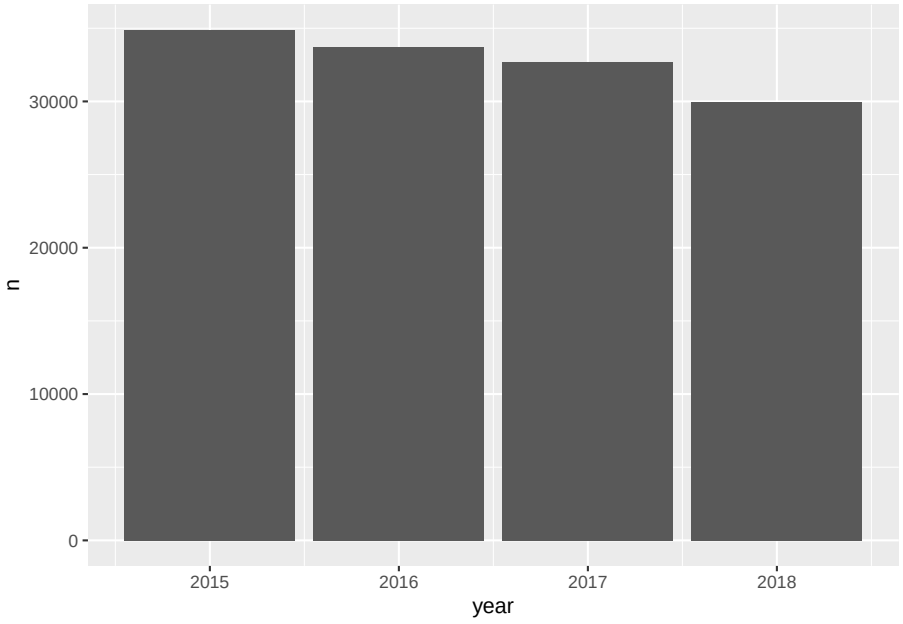
```
skimr::skim(ecometrics)
```

```
#> -- Data Summary ------------------------
#>                            Values
#> Name                       ecometrics
#> Number of rows             68
#> Number of columns          4
#> _________________________
#> Column type frequency:
#>   character                2
#>   numeric                  2
#> _________________________
#> Group variables            None
#>
#> -- Variable type: character -------------------------------------------------------
#>   skim_variable n_missing complete_rate   min   max empty n_unique whitespace
#> 1 type                  0             1     5    34     0       15          0
#> 2 measure               0             1     4    16     0        4          0
#>
#> -- Variable type: numeric ---------------------------------------------------------
#>   skim_variable n_missing complete_rate  mean      sd    p0   p25   p50   p75
#> 1 year                  0             1 2016.    1.13  2015 2016. 2016. 2017.
#> 2 n                     0             1 1929. 1857.      68  786  1230  2690.
#>    p100 hist
```

```
#> 1  2018
#> 2  7392
```

A simple graphic here would be to evaluate the raw counts by year. A simple bar chart would look like this.

```
ggplot(ecometrics, aes(year, n)) +
  geom_col()
```



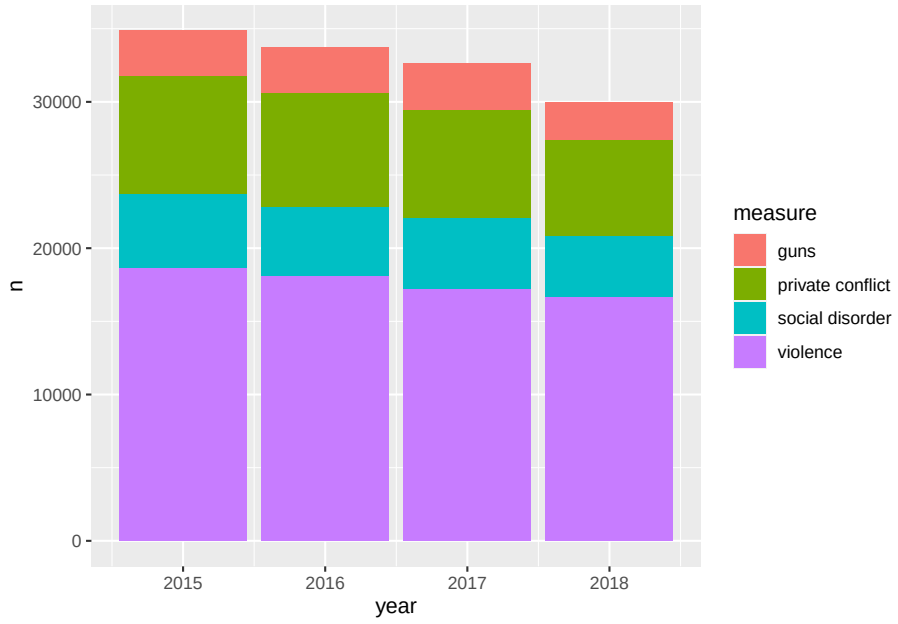
But, we are aware that there are different measurements. These were described previously and can be seen below.

```
distinct(ecometrics, measure)
#> # A tibble: 4 x 1
#>   measure
#>   <chr>
#> 1 violence
#> 2 guns
#> 3 private conflict
#> 4 social disorder
```

How can we partition our visualization to illustrate the number of counts per ecometric per year? We can use color—each measurement will receive it's own color. This will make it easier to determine the frequency of which each ecometric occurs. To do this we setting `fill` rather than `color` this is because we are

working with a polygon shape. `color` is used in working with lines and points. A useful trick is to think of `color` as the border and `fill` as the body fill.

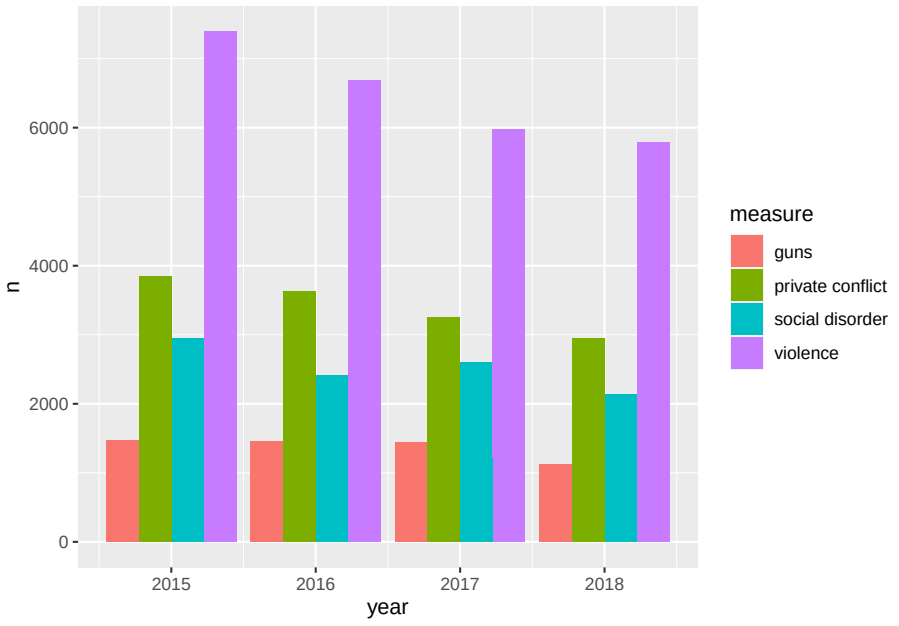
```
ggplot(ecometrics, aes(year, n, fill = measure)) +
  geom_col()
```



By mapping the fill to the `measure` variable we were able to create a stacked bar chart! It is apparent that `violence` is the most frequent of these ecometrics, followed by `private conflict`, `social disorder`, and then `guns`.

One of the downsides about the stacked barchart is that it is difficult to compare the sizes of each group relative to ones that are not adjacent. For example comparing `guns` to `social disorder` is made difficult as `private conflict` is situated between them. We can adjust our chart so that each bar is situated next to eachother. We do this by setting the argument `position = "dodge"` within the `geom_col()` layer.

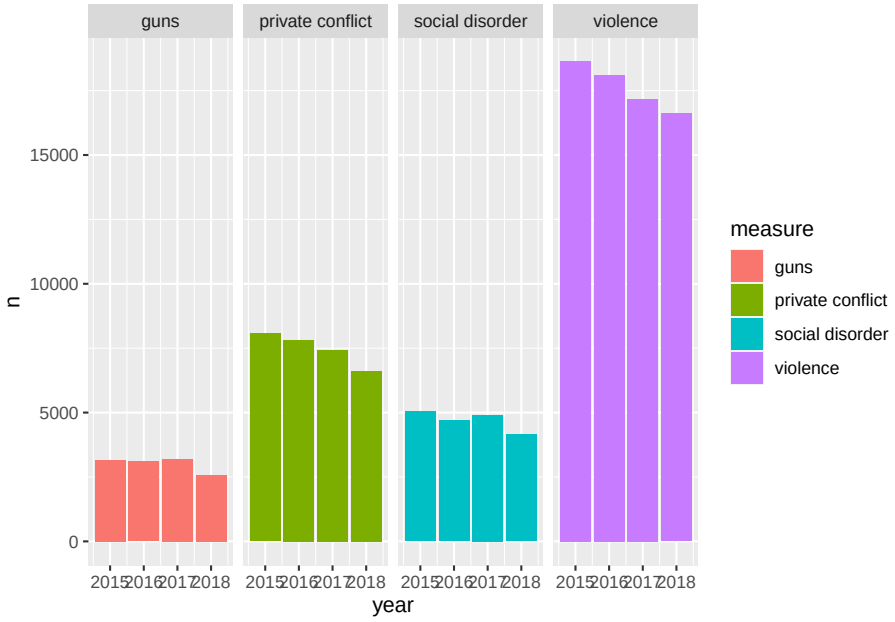
```
ggplot(ecometrics, aes(year, n, fill = measure)) +
  geom_col(position = "dodge")
```



The dodged bar chart makes it much easier to compare the heights of each bar. But now we are creating a somewhat cluttered graphic. In the situation where there are multiple groups and subgroups, it is often preferred to utilize facetting because the most important thing in any graphic is how easy it is to consume. We would rather make four plots than one messy plot.

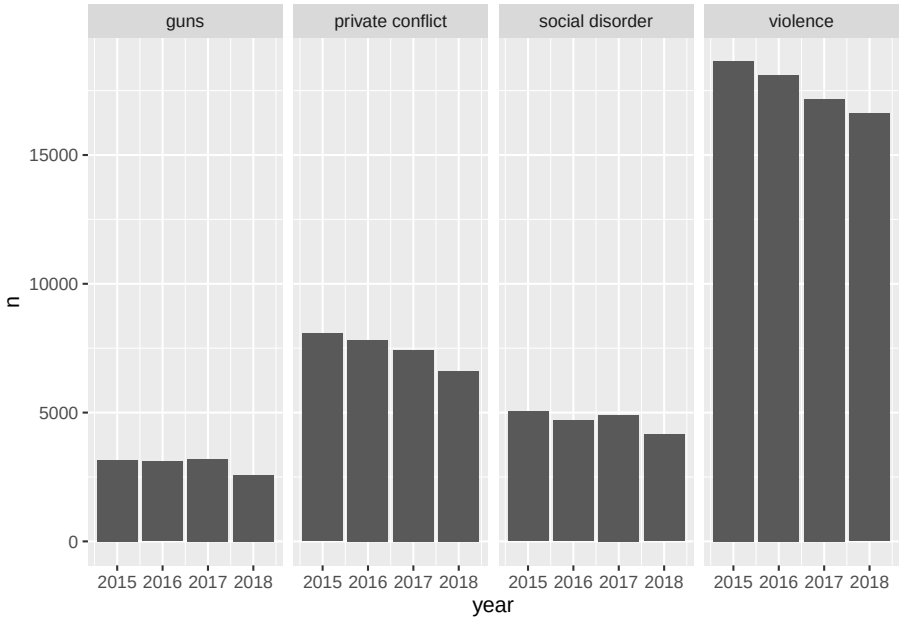
Let's facet by `measure` and tell `facet_wrap()` to create only one row.

```
ggplot(ecometrics, aes(year, n, fill = measure)) +
  geom_col() +
  facet_wrap("measure", nrow = 1)
```



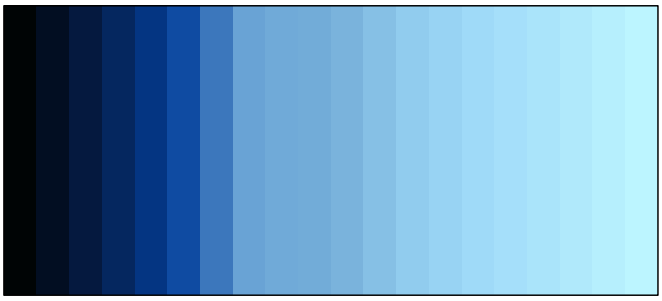
This is awesome! We have four different plots one for each measurement and it is extremely easy to see how each ecoemtrics has trended over the four year period. It seems like there has been a steady decrease! With this plot, however, we are labeling the ecometrics twice: once with the panel label and once with the legend. Since each facet is labeled individually and are not situated next to any other ecometrics, the color becomes redundant. Unless there is an important reason to visualize the color when faceting, it is most likely not needed. As such, a final visualization would look like below.

```
ggplot(ecometrics, aes(year, n)) +
  geom_col() +
  facet_wrap("measure", nrow = 1)
```



21.1.1 Continuous color scales

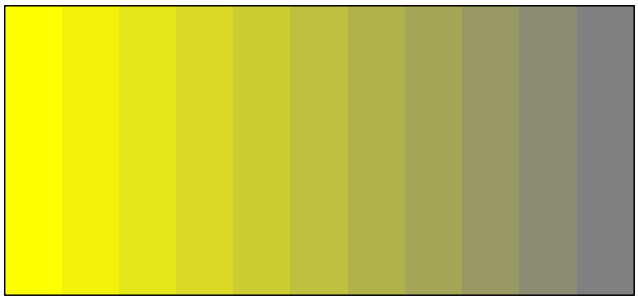
In the cases where our variable of interest is a continuous numeric one, we ought to be using a continuous color scale. These are scales that change from one color to another to illustrate a range of values—for example we could use this to visualize probabilities from 0 - 1. The below is an example of one of these color palettes.



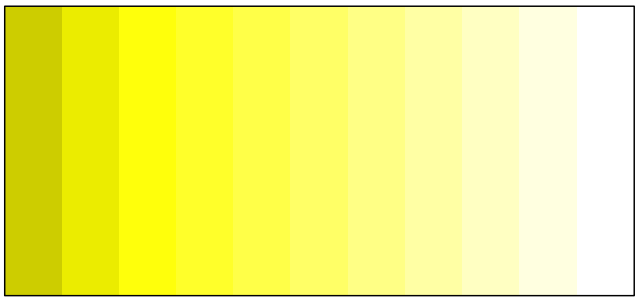
Often you'll encounter visualizations that use a rainbow color palette or some other number of colors. To illustrate a range of values. This is *not* recommended. When we are looking at color, we are best able to detect changes in the luminescence (perceived brightness) and saturation [1]. As such, we should work with color palettes that are easiest to interpret. First we'll visualize what changing saturation and brightness looks like.

Below is is an image of 10 colors. Starting at the left is the color yellow (hex code `#FFFF00FF`). Each step it is desaturated by 10 percent ending with the color grey (hex code `#808080FF`). We can think of saturation as how much color there is.

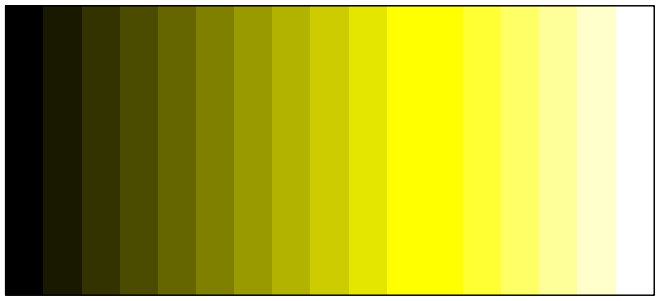
[1] Why should engineers and scientists care about color and design? *Flowing Data*. https://flowingdata.com/2008/04/29/why-should-engineers-and-scientists-care-about-color-and-design/.



Below is an example of what changing the brightness can look like.

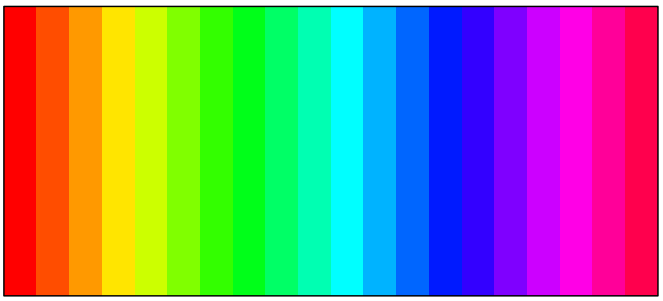


If we expand the usage of brightness on both ends of the spectrum we get the below.

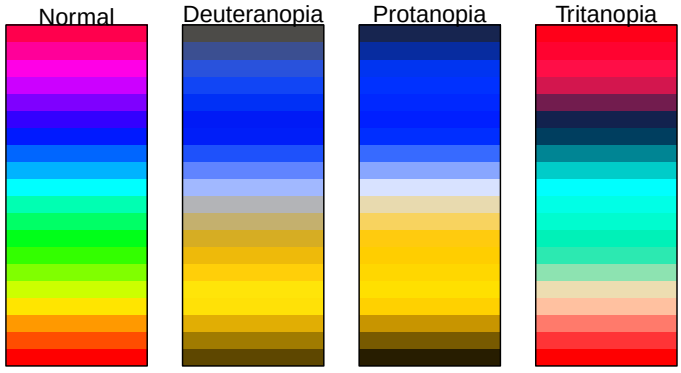


You may find that in the course of your work that individuals will use a color palette like the below.

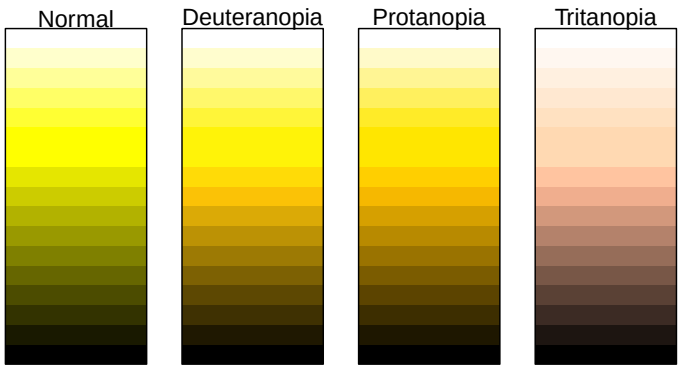
```
prismatic:::plot.colors(rainbow(20))
```



I strongly **advise against** this. These colors make it extremely difficult to tell changing values. Consider for a moment how you would try and tell the difference in numeric value between a magenta and a maroon. It would be rather difficult. Moreover, this color palette is not very accessible to those who are color deficient. The below is an approximation of what those with varying types of color blindness might see.



Compare this with the earlier example of dark to light yellows.



This is much more accessible to those who are color deficient as well as providing a clearer understanding of a *range* of values.

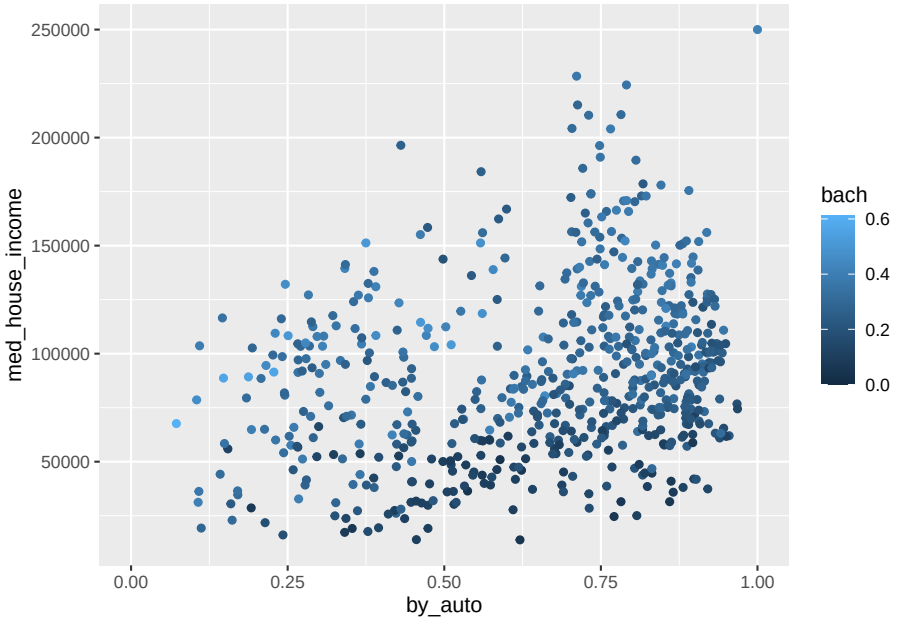
21.1.1.1 Example

To explore this in R, we will return to the commute data set we created from BARI's Census Indicators very early on. We will visualize the relationship of commuting by automobile and median household income. Moreover, we will color each point by the rate of Bachelor's degree attainment. As educational attainment tends to increase with income, we should expect the coloring to somewhat follow household income.

Create the visualization following the below steps.

- Read in `data/gba_commute.csv`
- Plot `med_house_income` against (on the y axis) `by_auto`
- Color the plot by `bach`
- Add the appropriate geometry

```
commute <- read_csv("data/gba_commute.csv")

(p <- ggplot(commute, aes(by_auto, med_house_income, color = bach)) +
  geom_point())
```



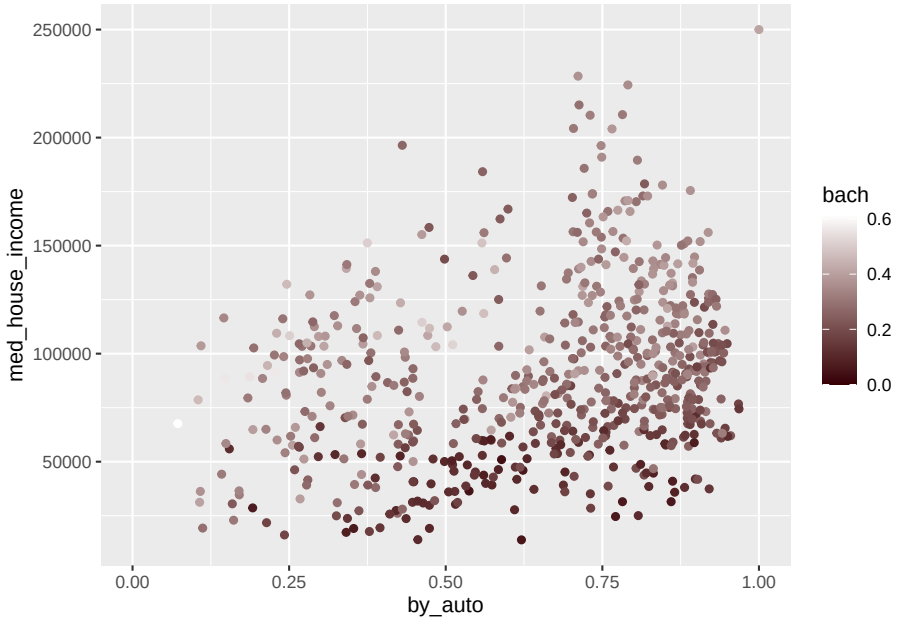
Wonderful! As anticipated, the color of the points are darkest at the bottom where median household income is the lowest. And as income increases, so does the brightness and saturation of our color scale. If we return to the grammar of graphics, we can take further control of the scales. In this case, the color of the scales. One of the ways we can change the color scale is to add the layer `scale_color_gradient()` to manually choose what colors to use in our scale. This provides a lot of flexibility to you as a information designer. Think of how you can use the color to represent what it is thatyou are visualizing. Or, how you can use colors to adhere to a color palette of your own or an organization you are working with.

`scale_color_gradient()` allows us to provide the colors for low values and high values. In general, you should choose a darker color to represent lower values and brighter colors for higher values. Below we add the layer and provide color codes.

> You can find color codes via google by searching "color picker." Or, you can use any number of free color palette generators online. I am personally a fan of coolors.co.

Below we've changed the plot to transition from a dark red to a white as values increase.

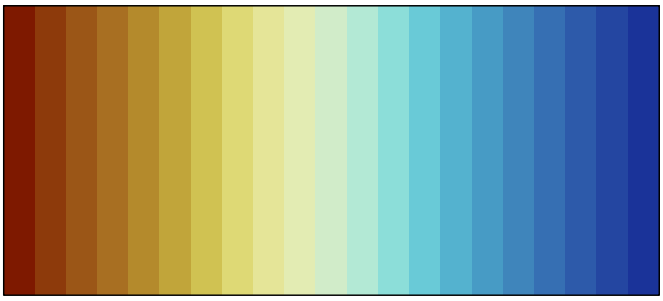
```
p +
  scale_color_gradient(low = "#360002", high = "white")
```



Try modifying the above plot by picking two colors that you think do a good job of visualizing a range of values.

21.1.1.2 Diverging colors

There are many a time when our variable of interest has a middle value and illustrating the center as well as the deviation from that center is important. To visually represent this we use what is called a diverging color scale. Diverging color scales are characterized by a middle color from which both ends of the spectrum originate. The center is to represent some middle value. If we contextualize this as z-scores and the color palette below, the center would be `0` and any negative scores would trend towards red. Whereas if they trended positive they would become more blue.



One thing to be wary of is using a divergent color scale when it is not necessary. This is an easy trap to fall into since they're pretty cool. Remember, *only* use diverging color palettes if there is an existing middle value.

21.1.1.3 Example

Take our ecometrics again. Say we are interested in what the annual deviation is from the sample mean—the average for all years—of each ecometric. This is the perfect use case for a diverging color scale. This will require a bit of computational creativity. So lets work through this.

Let's think about each of the measures we need to calculate. We need to:

- Find the number of counts for each year by ecometric.
- Find the average count for all years for each ecometric.
- Identify the deviation from the mean.

We will work through this sequentially. First and foremost we need to calculate the total number of crime reports by ecometric by year. The dataset has more than one observation per year per ecometric. We can see this by running a quick count.

```
count(ecometrics, measure, year)
#> # A tibble: 16 x 3
```

```
#>    measure           year     n
#>    <chr>            <dbl> <int>
#>  1 guns              2015     5
#>  2 guns              2016     5
#>  3 guns              2017     5
#>  4 guns              2018     5
#>  5 private conflict  2015     4
#>  6 private conflict  2016     4
#>  7 private conflict  2017     4
#>  8 private conflict  2018     4
#>  9 social disorder   2015     3
#> 10 social disorder   2016     3
#> 11 social disorder   2017     3
#> 12 social disorder   2018     3
#> 13 violence          2015     5
#> 14 violence          2016     5
#> 15 violence          2017     5
#> 16 violence          2018     5
```

We need to tidy this up and ensure that each row is only one observation—in this case one ecometric per year—with the total count. The logic to accomplish this is to first `group_by()` *both* `measure` and `year` and then sum the `n` values. As such:

```
ecometrics %>%
  group_by(measure, year) %>%
  summarise(n = sum(n))
#> # A tibble: 16 x 3
#> # Groups:   measure [4]
#>    measure           year     n
#>    <chr>            <dbl> <dbl>
#>  1 guns              2015  3146
#>  2 guns              2016  3111
#>  3 guns              2017  3190
#>  4 guns              2018  2585
#>  5 private conflict  2015  8063
#>  6 private conflict  2016  7807
#>  7 private conflict  2017  7410
#>  8 private conflict  2018  6592
#>  9 social disorder   2015  5043
#> 10 social disorder   2016  4707
#> 11 social disorder   2017  4876
#> 12 social disorder   2018  4168
#> 13 violence          2015 18621
#> 14 violence          2016 18080
```

```
#> 15 violence          2017 17172
#> 16 violence          2018 16630
```

One of the dplyr quirks is that after you `summarise()`, one level of grouping is removed. This is because we have already performed our aggregate measure and the last level of grouping is now unit of analysis (a row). Since we are currently grouped at the `measure` level and each row represent one year we are already ready to calculate the average `n` value by group. Rather than using `summarise()` as we are used to doing with summary statistics, we will use `mutate()` and create a column called `avg` with the average `n` value for each group. This is because we want to be able to perform a column-wise operation to subtract the average from each row's `n` value.

I continue by creating three new columns. The first is `avg` which is set to the mean of `n` by group. Since the resulting value of `mean(n)` is a single value, each observation in the group gets that value. Second, I create a new column called `deviation` which subtracts the new column we created from the `n` column. The lastly I created a new column to contain the name of the ecometric as well as the year. This is done to ensure that in the visualization each row can be plotted.

```
annual_metrics <- ecometrics %>%
  # group by measure and year
  group_by(measure, year) %>%
  # find the total n for each measure and year
  # summarise loses the last level of grouping
  summarise(n = sum(n)) %>%
  mutate(
    # calculate average `n` by ecometric
    avg = mean(n),
    # calculate the deviation from the mean
    deviation = n - avg,
    # creating a new column that combines the name of the ecometric and the year
    name = glue::glue("{measure}: {year}"),
  )
```

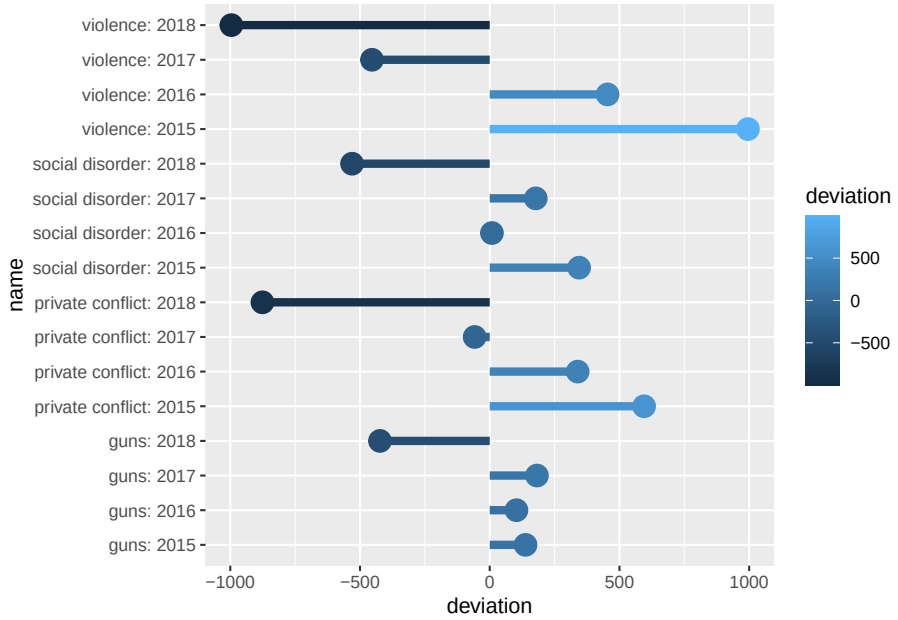
I snuck in a new package right there. `glue` is a package that let's us incorporate R expressions with plain text. Whatever is inside of the brackets `{` will be run as an R expression. So since I referenced the columns `measure` and `year` as `"{measure}: {year}"` each value of `name` will look something like `social disorder: 2017`.

Now that we have a data frame with all of the values we want to plot, let's go ahead and do that! We will use a lollipop chart here to do so.

> Remember that `geom_lollipop()` comes from the `ggalt` package. We can use the function without loading the whole package by referencing the `pkgname::function_name()`.

Within the chart we map `devation` to the color aesthetic.

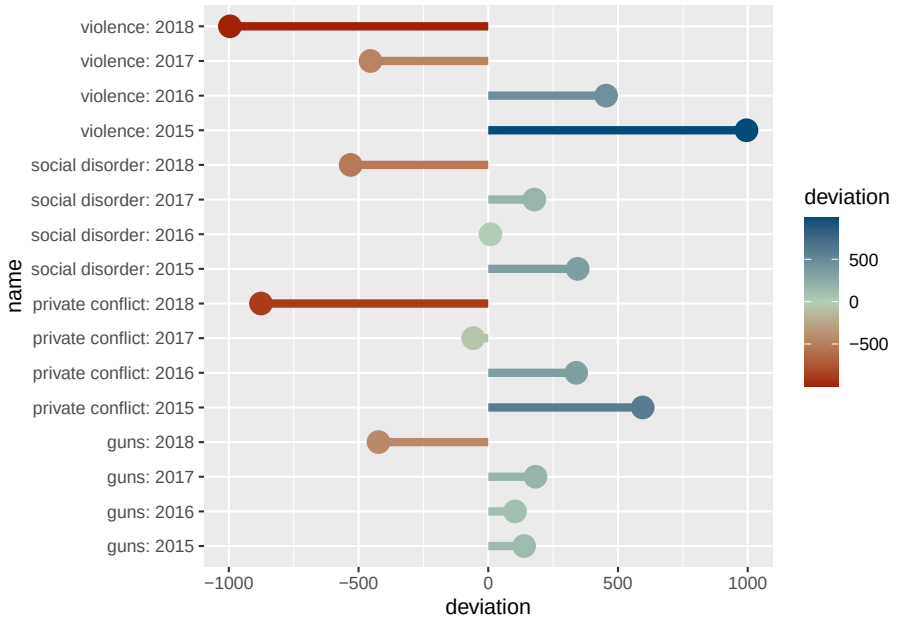
```
(p <- ggplot(annual_metrics, aes(y = deviation, x = name, color = deviation)) +
  ggalt::geom_lollipop(size = 2) +
  coord_flip())
#> Registered S3 methods overwritten by 'ggalt':
#>   method                  from
#>   grid.draw.absoluteGrob  ggplot2
#>   grobHeight.absoluteGrob ggplot2
#>   grobWidth.absoluteGrob  ggplot2
#>   grobX.absoluteGrob      ggplot2
#>   grobY.absoluteGrob      ggplot2
```



The above graphic doesn't take into account the middle value of `0`. Because of this, we need to tell ggplot2 that this is a diverging color scale. When we were working with a normal continuous color scale we used `scale_color_gradient()`. Instead, since we have a middle value, to create the diverging color gradient we use `scale_color_gradient2()`. This adds two more arguments `mid` and `midpoint`. The former is the color of that middle value. The second is what that middle value maps to—defaults to 0, which is fine for our example.

Now if we add the below colors—a red, yellow-ish beige, and a blue—we will have a diverging color scale on the plot!

```
p +
  scale_color_gradient2(low = "#A12106", mid = "#B2CEB7", high = "#004C76")
```



Information design is a seemingly endless field of which we only touched on a very small amount. The R community has put a lot of work into enabling the use of color for visualization purposes. The above images of color palettes were created with the help of the wonderful packages `prismatic` and `paletteer`

To explore color more check out the packages `prismatic` and `paletteer` by Emil Hvitfeld.

21.2 Shape and Size

We spent a fair amount of time looking at a number of the ways that color can be used. Color can be used to visualize such a wide variety and as such was deserving of such a long section. Here we will briefly look at the other two aesthetics: shape and color.

21.2.1 Shape

Shape is another useful way to visualize groups or categories in our data. In general we should use shape only if color is not an option for us. Variations in shapes can be more difficult to discern—particularly when the number of

groups to be visualized reaches beyond ~4. Moreover, shapes, depending on choice and intricacy can become overly distracting and can detract from the visualization as a whole. As a heuristic, do not map both shape and color to the same variable.

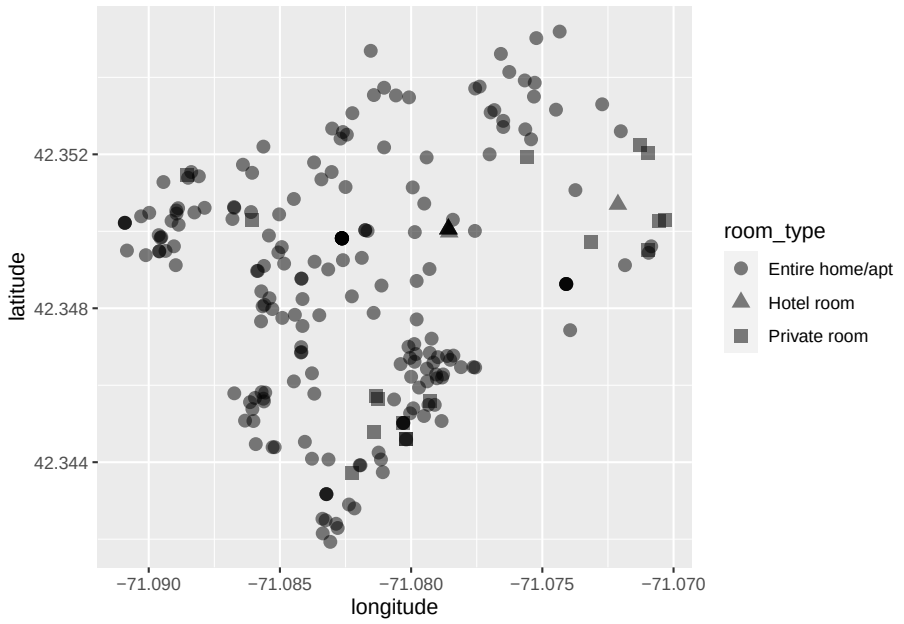
To illustrate the use of shape we will use data from Inside Airbnb [2]. This dataset contains Airbnb listings for Boston as well. The dataset can be found at `data/airbnb/airbnb.csv`. We will go into the dataset in more depth in the chapter on spatial analysis.

For this visualization we will read in the dataset, filter it down to just `"Back Bay"`, and then plot ponts based on their place in space—aka latitude and longitude. We will also map shape to the `room_type`. Doing this will show us the spatial distribution of Airbnbs as well as the different kinds.

> Note that while we normally say "latitude and longitude", that actually is saying "y and x" respectively. So be sure to put the latitude in the y aesthetic position and *not* longitude.

```
# read in data
airbnb <- read_csv("data/airbnb/airbnb.csv")
#> Parsed with column specification:
#> cols(
#>   id = col_double(),
#>   neighborhood = col_character(),
#>   room_type = col_character(),
#>   price = col_double(),
#>   longitude = col_double(),
#>   latitude = col_double()
#> )

# filter to backbay
bb <- airbnb %>%
  filter(neighborhood == "Back Bay")

# plot the points by shape
ggplot(bb,  aes(longitude, latitude, shape = room_type)) +
  geom_point(alpha = .5, size = 3)
```

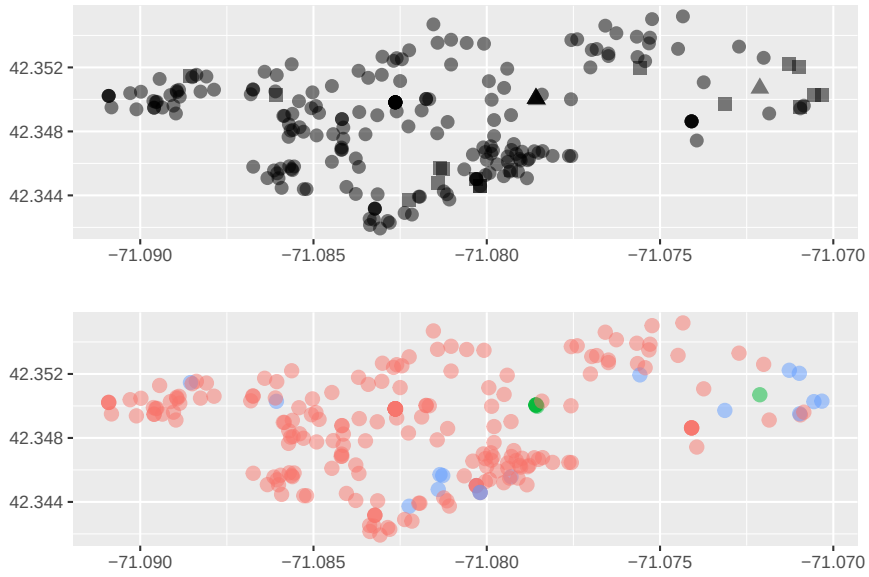
[2]Inside Airbnb. http://insideairbnb.com/. [3]: Big Belly. https://data.boston.gov/dataset/big-belly-alerts-2014/resource/8fb74119-97b9-4114-b773-ea0a82142d3b.



By changing each point's shape by their associated room type we can get a somewhat better idea of the spatial distribution by type.

What can we tell from this visualization? It looks like most Airbnbs are overwhelmingly the entire home or appartment. We can infer from this that most Airbnbs _are not_being used as additional revenue for residents. But rather that each Airbnb might be a unit of housing that is no longer available to Boston residents. Could this be increasing demand for housing? Is the rise in Airbnbs creating a shortage of housing and can it be one of the factors behind the rising Boston rents?

The above visualization is good, but lets compare that with the use of color.



Which do you prefer? Which do you feel does a better job getting this message across? Again, we are delving into the world of design and there is never a *correct* answer. But sometimes there may be a consensus. So just do your best and ask others for their thoughts on your visualizations.

21.2.2 Size

Moving onto size. Size is an aesthetic that is one of the hardest for humans to properly distinguish between. Because of this, size should usually be used when comparing observations with large discrepancies between them. Doing this with ggplot is again rather straight forward as here the only difference is, is that we set the aesthetic `size` to some other column (this must be numeric).

This visualization will use data from Analyze Boston's legacy database. Analyze Boston has previously released all signals from their Big Belly trashcans for the year of 2014[4]. In the words of the data dictionary:

> Bigbelly's contain compactors which crush waste within their waste bins in order to reduce the volume required to store the waste deposited into the Bigbelly. The compactors are able provide an average compaction ratio of 5:1, meaning a Bigbelly can store the equivalent of 150G (gallons) of uncompactedwaste within a standard 30G waste bin. The compactor in a Bigbelly will run when it has detected

[4] Analyze Boston: Big Belly Alerts 2014

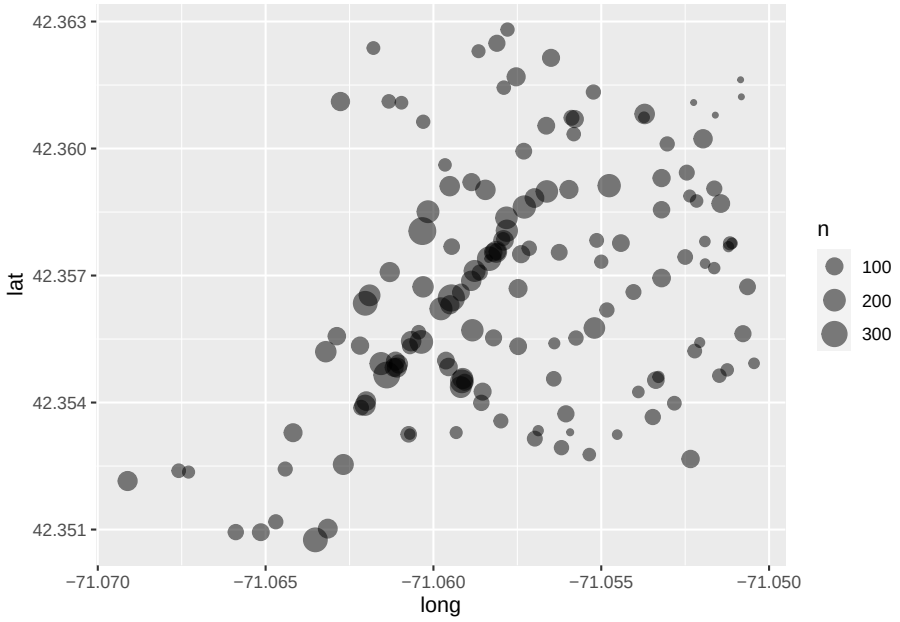
that waste within the bin has reached a certain level. After a compaction cycle the waste will be crushed to occupy a smaller amount of space within the waste bin. These cycles are repeated as more waste is added to the Bigbelly. After approximately 150G of uncompacted waste has been added to the Bigbelly the compactor will not be able to compress the waste further, this condition is detected and thewaste bin is considered to be full.

For the sake of example, I have created an aggregate count of signals from all Big Belly receptacles for 2014. This file can be found at `data/downtown-big-belly.csv`.

```
big_belly <- read_csv("data/downtown-big-belly.csv")

glimpse(big_belly)
#> Rows: 147
#> Columns: 4
#> $ description <chr> "1 North Market Street  (in front of McCormick & Schmic...
#> $ n           <dbl> 47, 179, 257, 108, 171, 93, 27, 173, 55, 64, 126, 190, ...
#> $ lat         <dbl> 42.36034, 42.35653, 42.35634, 42.35557, 42.35520, 42.35...
#> $ long        <dbl> -71.05582, -71.06190, -71.06203, -71.06288, -71.06321, ...
```

Similar to the Airbnb data, we will visualize by latitude and longitude (`lat` and `long` respectively) while mapping the `size` aesthetic to `n`.

```
ggplot(big_belly, aes(long, lat, size = n)) +
  geom_point(alpha = 1/2)
```


42.363
42.360
42.357
42.354
42.351
lat
−71.070
−71.065
−71.060
−71.055
−71.050
long
n
100
200
300

Chapter 22

Visualizing through time

Throughout this section, one type of visualization has been missing from our repetoire—the timeseries plot. This is because time series data is rather cumbersome to work with. Time series are unique because each observation represents some point in time. There is order inherent to the data. Because of this natural ordering of the data it makes the problem a bit trickier. But now that we have the visual tools and principles sorted out, we can apply them to time series data and visualize them as well.

For this section we will work with the 2014 Big Belly data again. Our goal will be to visualize the reports by fullness over the course of the year. In this chapter we will first learn how to work with dates. Then we will use our new skills to aggregate our observations and view them over time. Next, we will discuss traditional methods of time series visualization. And finally we will quickly touch on the use of animation for viewing time-series.

22.1 Working with dates

The file that we will use is located at `data/big-belly-2014.csv`. Read in the dataset and assign it to the tibble `big_belly` and then preview it with `glimpse()`.

```
library(tidyverse)

big_belly <- read_csv("data/big-belly-2014.csv")

glimpse(big_belly)
```

```
## Rows: 51,440
```

```
## Columns: 6
## $ description <chr> "Atlantic & Milk", "1330 Boylston @ Jersey Street", "SE...
## $ timestamp   <dttm> 2014-01-01 00:41:00, 2014-01-01 01:19:00, 2014-01-01 0...
## $ fullness    <chr> "YELLOW", "YELLOW", "YELLOW", "RED", "GREEN", "YELLOW",...
## $ collection  <lgl> FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, TRUE, FA...
## $ lat         <dbl> 42.35870, 42.34457, 42.34818, 42.34818, 42.34933, 42.34...
## $ long        <dbl> -71.05144, -71.09783, -71.09744, -71.09744, -71.07702, ...
```

What we can see is that the data are rather large and contains a column with the type `<dttm>` which is new to us. `<dttm>` is the way that a tibble represents a column of the type date time.

> Note that the formal class for date time data is `POSIXct`. Working with date times with any computer is a tricky process.

Date time objects follow the format of `YYYY-MM-DD HH:MM:SS` where `Y` is year, `M` month, `D` day, `H` hours, `M` minutes, and `S` seconds.

This begs the question "how can we aggregate based on time?" The easiest way to do this is to group our observations by some interval. To do this we will solicit the help of the package `lubridate`. lubridate is part of the tidyverse, but it doesn't loadwith the tidyverse. As such, we will have to load the package ourselves.

```
library(lubridate)
```

The package contains dozens of useful functions for working with dates in R. By no means will we go over each of them.

> To explore all of the functions in a package click on the `Packages` pane and search for the package you are interested in. Click the hyper link with the package name and then all of the exported functions and objects *should* be documented there.

We will, however quickly touch on `ymd()`, `month()`, and `floor_date()`.

The first is not useful in this example since `read_csv()` already parsed the column as a date time for us, but is important nonetheless. `ymd()` will convert a character string into a date object. The letters stand for *year*, *month*, and *date*. This function parses dates in the format of yup, you guessed it, year-month-day. For example:

```
ymd("2020 01 20")
```

```
## [1] "2020-01-20"
```

```
ymd("2020 jan. 20")
```

```
## [1] "2020-01-20"
```

```
ymd("20, january-20")
```

```
## [1] "2020-01-20"
```

There are others such as `mdy()` and `dmy()` which you can use to parse dates as well.

> The next time you are trying to parse a date think about the way it is formatted. Does the year precede or follow the month?

Immediately useful to us is the `month()` function. This will extract the month component of a given date.

```
month("2020-01-01")
```

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

The above returned the value of `1` because January is represented by 1. If we wished to return the full name of the month we set the argument `label = TRUE`.

```
month("2020-01-01", label = TRUE)
```

```
## [1] Jan
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
```

Furthermore, we can tell lubridate to not abbreviate the months by setting `abbr = FALSE`.

```
month("2020-01-01", label = TRUE, abbr = FALSE)
```

```
## [1] January
## 12 Levels: January < February < March < April < May < June < ... < December
```

We can use the above function to extract the months from our Big Belly dataset. This will be extremely useful in aggregation. The one limitation, however, is that if there were another year present we would be grouping observations from both years into the the same month bucket. To avoid this, we can use the function

`floor_date()`. `floor_date()` takes a date object and returns the closest date rounding down to a given unit. This may be best explained through an example. Given the date March 15th, 2020, let us round down to the near units `"week"`, `"month"`, and `"year"`.

```
# create date object
mar_15 <- ymd("2020-03-15")

# round to week
floor_date(mar_15, unit = "week")
```

```
## [1] "2020-03-15"
```

```
# round to month
floor_date(mar_15, unit = "month")
```

```
## [1] "2020-03-01"
```

```
# round to year
floor_date(mar_15, unit = "year")
```

```
## [1] "2020-01-01"
```

With these new tools lets modify our `big_belly` tibble by creating two new columns `month` (with the labels) and `week` using `month()` and `floor_date()` respectively and assign the result to an object called `bb`.

```
bb <- big_belly %>%
  mutate(
    month = month(timestamp, label = TRUE),
    week = floor_date(timestamp, unit = "week")
  )

slice(bb, 1:5)
```

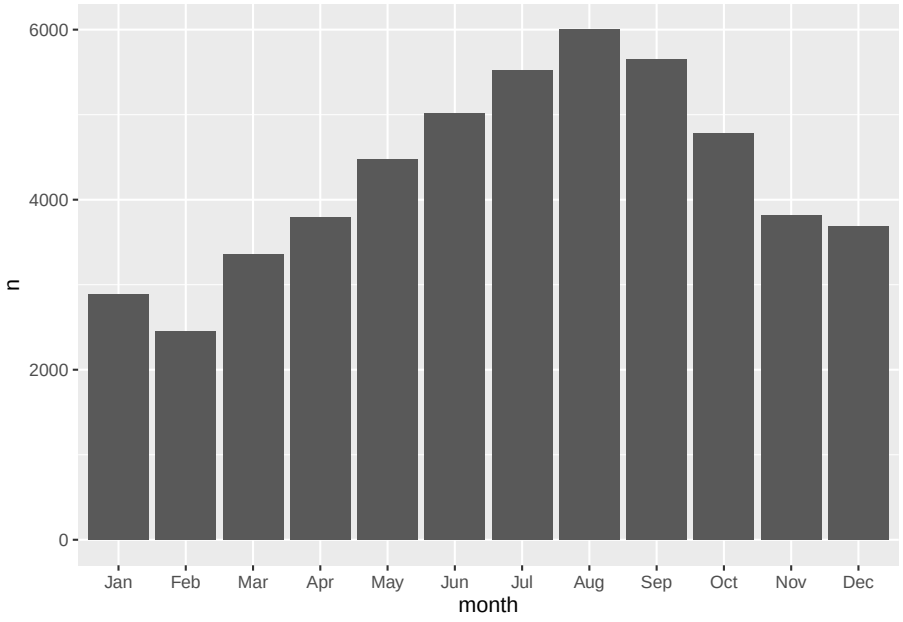
```
## # A tibble: 5 x 8
##   description timestamp           fullness collection   lat  long month
##   <chr>       <dttm>              <chr>    <lgl>      <dbl> <dbl> <ord>
## 1 Atlantic &~ 2014-01-01 00:41:00 YELLOW   FALSE       42.4 -71.1 Jan
## 2 1330 Boyls~ 2014-01-01 01:19:00 YELLOW   FALSE       42.3 -71.1 Jan
## 3 SE Brookli~ 2014-01-01 01:32:00 YELLOW   FALSE       42.3 -71.1 Jan
## 4 SE Brookli~ 2014-01-01 01:34:00 RED      FALSE       42.3 -71.1 Jan
## 5 Huntington~ 2014-01-01 02:10:00 GREEN    TRUE        42.3 -71.1 Jan
## # ... with 1 more variable: week <dttm>
```

22.2 Standard visual approach

When we visualize time we need to consider how we humans cognitively and visually perceive time. Time is linear. It follow a path from start to end. Because of this we want to (almost) always plot our time dimension on the horizontal x axis as time proceeds forwards not up or down. Perhaps as a product of these restrictions there are not many variations on how we can visualize time. Two plots reign in time-series: the line and the bar plot.

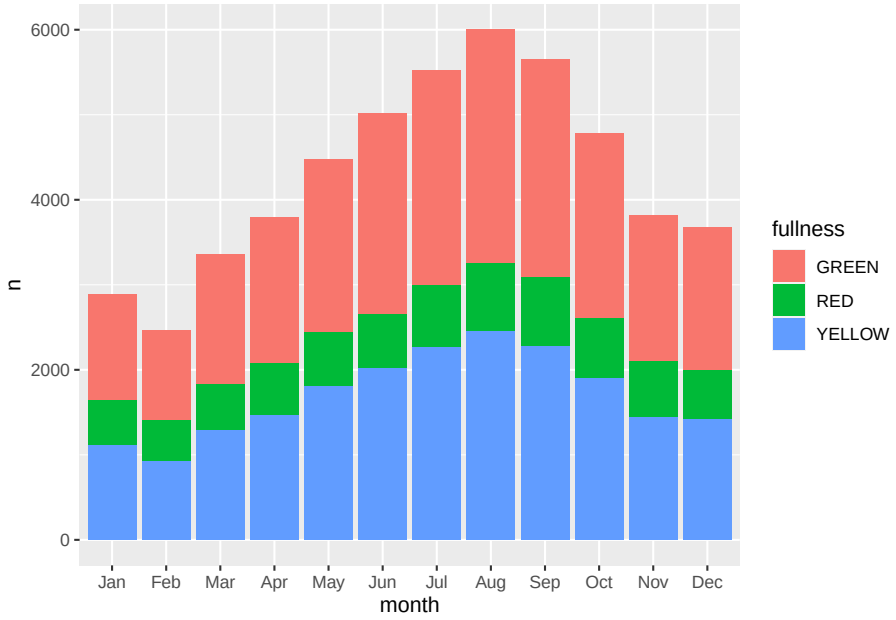
We can take our `bb` tibble and create a barchart of the total number of observations per month.

```
count(bb, month) %>%
  ggplot(aes(month, n)) +
  geom_col()
```



We can add a few more characters to the above code chunk so we can compare fullness over the months as well.

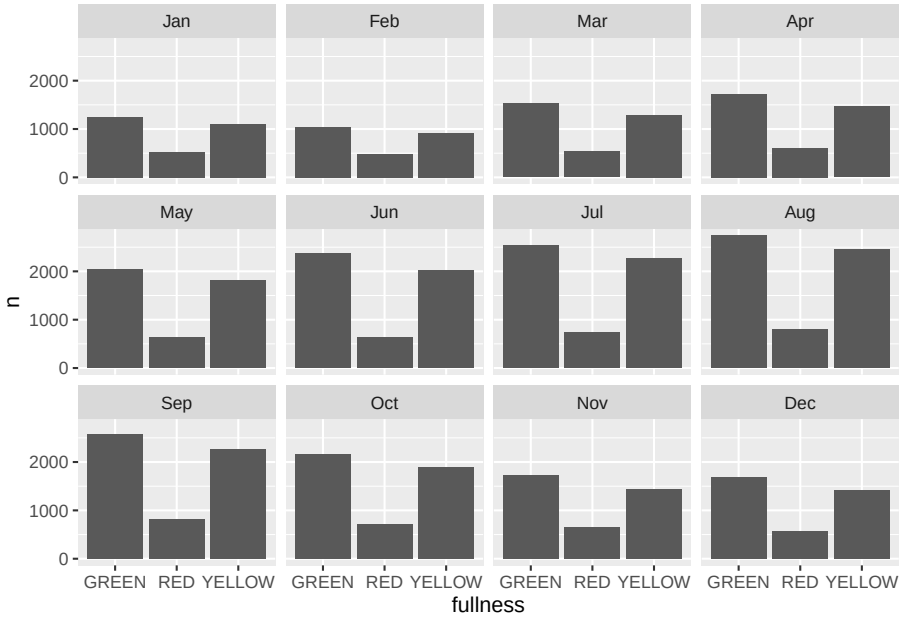
```
count(bb, month, fullness) %>%
  ggplot(aes(month, n, fill = fullness)) +
  geom_col()
```



Before we continue, we have to address the elephant in the room. We just made a plot with a legend that makes no sense. `GREEN` is mapped to red, `RED` is mapped to green, and `YELLOW` is mapped to blue. That's not good. We can fix this by adding manually mapping the color using one of the scale functions `scale_fill_manual(values = c("green", "red", "yellow"))`. Now back to the task at hand—time series.

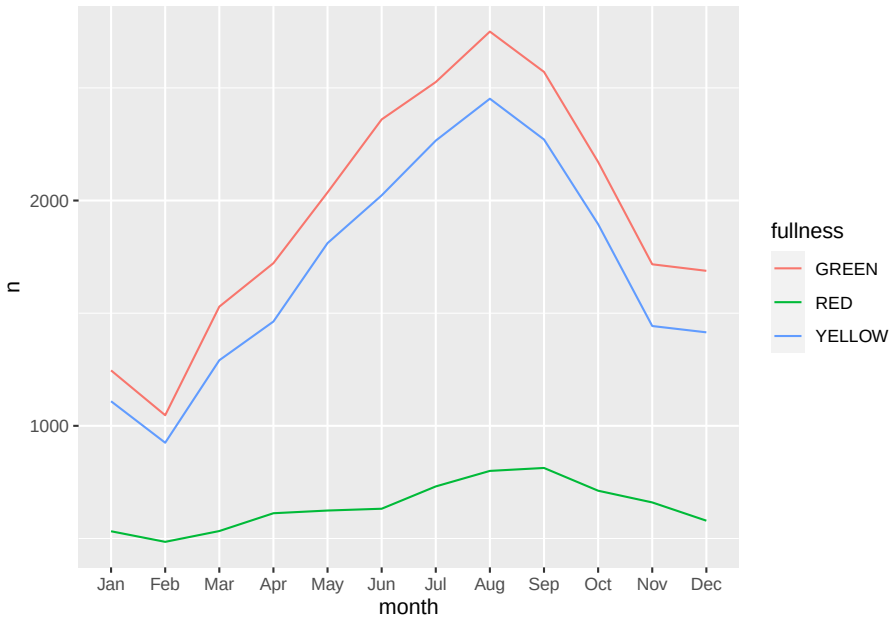
The stacked barchart above runs into the same limitations mentioned previously. And given the number of different time groups creating a dodged bar chart will become to cluttered by putting 36 total bars on there. We could consider faceting.

```
count(bb, month, fullness) %>%
  ggplot(aes(fullness, n, group = month)) +
  geom_col() +
  facet_wrap("month")
```



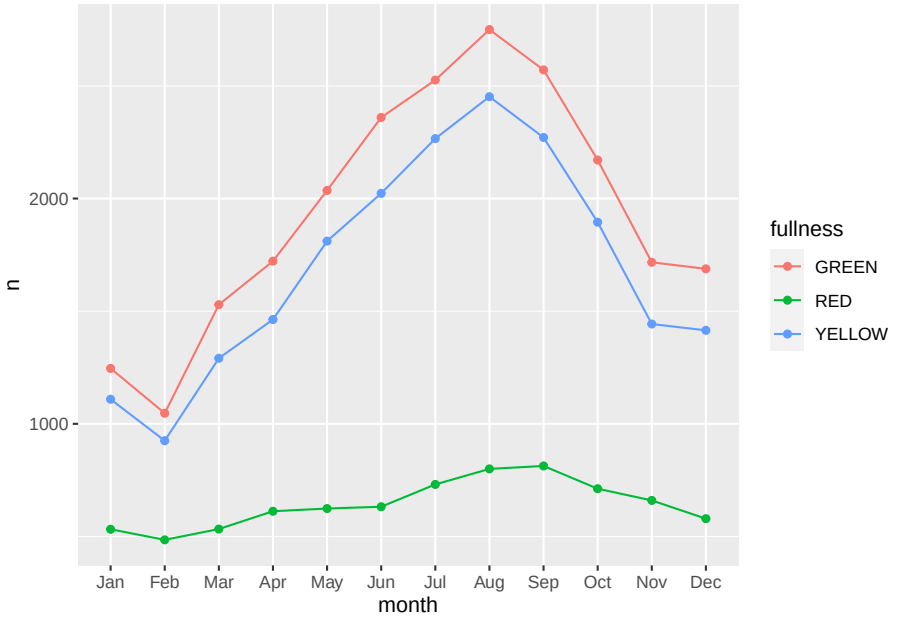
This is okay, but again, still too cluttered. In this scenario we should consider the use of a line graph. The line graph is the de facto time series plot. The lines connect points in time. Achieving this with ggplot is rather straight forward. We will put our time dimension on the x axis and add the `geom_line()` layer. By default `geom_line()` doesn't know what points to connect to each other when there is more than one per x value. So in our case we need to tell ggplot which points to connect by setting the `group` aesthetic. Since we want to group together the lines by `fullness` level set `group` as well as `color` to `fullness`. We can step up the plotting game by also

```
count(bb, month, fullness) %>%
  ggplot(aes(month, n, group = fullness, color = fullness)) +
  geom_line()
```



We can step up our plotting game by also adding a layer of points on top of the line to accentuate our lines.

```
count(bb, month, fullness) %>%
  ggplot(aes(month, n, group = fullness, color = fullness)) +
  geom_line() +
  geom_point()
```



The three types are graphs are all that you may ever truly need when visualizing time. But sometimes it's nice to go over the top and add all of the frills. Often a dynamic visualization may be more persuasive than a static one.For this, we can use animation and the `gganimate` package.

22.3 Animation as time

In 2008 researchers at Microsoft and a scholar from Georgia Institute of Technology published a paper titled *Effectiveness of Animation in Trend Visualization*[1]. This paper explored the reaction to a talk from 2006 which captured the attention of many by using animation to visualize global trends in health. That TED talk by Hans Rosling was a moving utilization of data visualization[2]. But why? This paper presented interactive and static visualizations to discover people's perspective on animation and its alternatives. They found that

> "users really liked the animation view: Study participants described it as"fun","exciting", and even"emotionally touching." At the same time, though, some participants found it confusing: "the dots flew everywhere."[3]

[1]Effectiveness of Animation in Trend Visualization. https://www.cc.gatech.edu/~stasko/papers/infovis08-anim.pdf.

[2]Hans Rosling. https://www.ted.com/talks/hans_rosling_the_best_stats_you_ve_ever_seen/transcript.

[3]Effectiveness of Animation in Trend Visualization. https://www.cc.gatech.edu/~stasko/papers/infovis08-anim.pdf.

We now have scientific backing that animation is pretty good but also we're unsure—which seems to be in line with the rest of science. We will take the above quote as a measure of caution. That we *can* creaate animations but also be careful with our use of them. For the remainder of this chapter we will learn about the the extension to ggplot2, gganimate. gganimate extends ggplot by extending the grammar to include elements pertaining solely to animation.

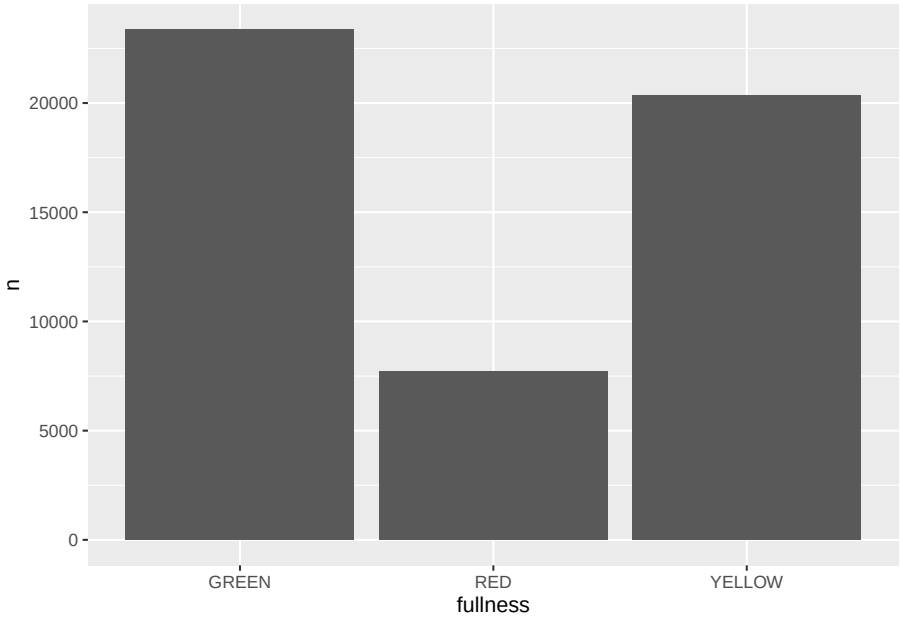
When working with gganimate we need to think in the context of *frames*. Each frame will be a cross-section of our data. Or you can think of each frame a still in our film. In the context of time-series each frame will be our period of time. So if we continue with our Big Belly example above, each frame can be thought of as each month.

To create animations we will need to install four packages: `gganimate`, `gifski` (what a brilliant name), `png`, and `transformr`. You can do so by running `install.packages(c("gganimate", "gifski", "png", "transformr"))`.

For extending ggplot2, gganimate adds three main types of layers to ggplot. There are most importantly the `transition_*()`s and the `enter_*()` and `exit_*()` layers. In this chapter we will only address the `transition_*()` layers. The transitions will be used to move through our time dimension. For working with time data we will use `transition_states()` and `transition_reveal()`. These are best used with bar charts and line plot respectively.

Since we will be changing our frames by our time dimension—`month` in this case—we start building our ggplot without it included. So, to create an animation of the counts by fullness by month we actually just start by visualizing the total counts.

```
(p <- count(bb, month, fullness) %>%
  ggplot(aes(fullness, n)) +
  geom_col())
```



All that is needed to add animation to this plot is a transition layer. For the use of a bar chart we can use `transition_states(states)` where states is the unquoted name of the column that wil be transitioned through.

```
library(gganimate)

p + transition_states(month)
```

This is awesome! We've created an animation . The bars change heigh with each change of state. The only bummer is that we don't know what states the animation is going through! When the animation is made there are four temporary variables made available for labeling. The documentation for `transition_states()` notes that the following variables are available[4].

- **transitioning** is a boolean indicating whether the frame is part of the transitioning phase
- **previous_state** The name of the last state the animation was at
- **closest_state** The name of the state closest to this frame
- **next_state** The name of the next state the animation will be part of

We can add a label layer to make this much more informative. To reference these variables we can use glue like quote strings with the above variables.

[4] `transition_states()`. https://gganimate.com/reference/transition_states.html.

```
p +
  transition_states(month)  +
  labs(title = "{closest_state}")
```

This ts the scaffolding for a great visualization. We can take this existing plot and add or adjust the layers to adjust the styling to be more engaging.

We can also add animation to our previous line plot. We can use the `transition_reveal()` layer to reveal changes over time.

```
count(bb, month, fullness) %>%
  ggplot(aes(month, n, group = fullness, color = fullness)) +
  geom_line() +
  geom_point() +
  transition_reveal(month)
```

```
Error: along data must either be integer, numeric, POSIXct, Date, difftime, orhms
```

Note the above error. This is telling us that since our x, or time dimension is actually being treated as a factor, it cannot move along the axis. The error message is informative enough to let us know that we need to change month to a numeric value. As a way to address this but yet maintain the nice labeling on the x axis we can create another column called `month_integer` which is jsut the integer value of the month—so Jan is 1 etc. We can provide that value to `transition_reveal()`. Moreover `transition_reveal()` creates the `frame_along` temporary variable if you'd so like to use it.

```
count(bb, month, fullness) %>%
  mutate(month_integer = as.integer(month)) %>%
  ggplot(aes(month, n, group = fullness, color = fullness)) +
  geom_line() +
  geom_point() +
  transition_reveal(month_integer)
```

This animation does a great job of illustrating the changing slopes of the fullness measures.

While it may seem a little underwhelming, these two functions can provide you with most of the functionality you will need for creating animations. Explore the functions in the package to explore how you can add to your animation and make it truly yours.

> `help(package = "gganimate")` will bring up all of the help documentation for the package.

And with that, we can conclude this section. Congratulations!

Chapter 23

Visualization Review

So here we are, at the end of the third and longest section of the Urban Informatics Toolkit. We've covered a lot of ground and rather quickly so let's recap.

We started by going over the grammar of graphics. The grammar is used to define the components of a visualization and ggplot2 is the R implementation of the graphics. The grammar has five main components:

1. Defaults:
    - Data
    - Mapping
2. Layers:
    - Data
    - Mapping
    - Geom
    - Stat
    - Position
3. Scales
4. Coordinates
5. Facets

We build plots by providing data either to `ggplot()` which sets the defaults, or the data can be provided directly to the geom layers. Remember that the `geom_*()` layers are what creates the geometry on the plots. Without them we do not populate the graphic. The layers figure out the positions, scales, and coordinates from the data. We can also adjust these to fit our preference by using `scale_*()` and `coord_()` layerss if we would so desire.

Following this, we explored the ways in which univariate and bivariate relationships can be explored visually. In this we explored the use of a number of different plots which can be added to your repertoire. Then we looked at how we can use facetting, color, shape, and size to explore beyond two dimensions. And finally, we briefly looked at the use of animation to explore data through time.

All of the visualization strategies can be used either independently or in combination to create compelling graphics thats tell a story from data.

In the next sectino we will cover a few more advanced and disparate topics which are important to have in your tool kit. It may be good to take a break right now before we continue.

Are you hydrated?

Part IV

More than hammer and nails

Chapter 24

Multiple data sets

Many times working with just one data set will not suffice. More often than not the data that we will be working with be will need to be supplemented by other data sets. These datasets have a shared relation which are what enables us to **join** them together. A join is a way of combining the columns between two sets of data while ensuring the rows are properly aligned.

The relationship that joins these tables together are expressed in the data through what is called a **common identifier**. This a variable exists in both datasets—perhaps, with a different name—and can be used as a reference to associate rows from each table together.

Recall to our definition of tidy data. The definition that we used characterized the data inside of a single table. There is actually another rule: "each observational unit forms a table." So, in our previous definition a row was a single observation. When we extend the definition as such this is still the case but we have to create distinctions between "observational units[s]."

To explore this concept we will use data about Airbnb listings in Boston. These data come from Inside Airbnb. Inside Airbnb collects Airbnb's public listing and makes them available through a "non-commercial set of tools and data that allows you to explore how Airbnb is really being used in cities around the world"[1].

Inside Airbnb data are another example of harnessing naturally occurring data. The listings are not generated with the intent of *being* data, but by the virtue of their existence, they become data. We are able to harness them to learn about the shifting neighborhood dynamic of cities. They can tell us something about short term rentals in a locale and about where people may be visiting and when.

We will first look at two data sets: `listings`, and `hosts`.

[1] Inside Airbnb. http://insideairbnb.com/about.html.

```
listings <- read_csv("data/airbnb/listings.csv")

glimpse(listings)
#> Rows: 3,799
#> Columns: 7
#> $ id               <dbl> 3781, 5506, 6695, 8789, 10730, 10813, 10986, 16384...
#> $ neighborhood     <chr> "East Boston", "Roxbury", "Roxbury", "Downtown", "...
#> $ room_type        <chr> "Entire home/apt", "Entire home/apt", "Entire home...
#> $ price            <dbl> 125, 145, 169, 99, 150, 179, 125, 50, 154, 115, 14...
#> $ minimum_nights   <dbl> 28, 3, 3, 91, 91, 91, 91, 91, 29, 1, 2, 2, 2, 6, 3...
#> $ availability_365 <dbl> 68, 322, 274, 247, 29, 0, 364, 365, 304, 285, 62, ...
#> $ host_id          <dbl> 4804, 8229, 8229, 26988, 26988, 38997, 38997, 2307...
```

```
hosts <- read_csv("data/airbnb/hosts.csv")

glimpse(hosts)
#> Rows: 1,335
#> Columns: 9
#> $ id               <dbl> 4804, 8229, 26988, 38997, 23078, 71783, 85130, 8577...
#> $ name             <chr> "Frank", "Terry", "Anne", "Michelle", "Eric", "Lanc...
#> $ since_year       <dbl> 2008, 2009, 2009, 2009, 2009, 2010, 2010, 2010, 201...
#> $ since_month      <chr> "12", "02", "07", "09", "06", "01", "02", "02", "03...
#> $ since_day        <chr> "03", "19", "22", "16", "24", "19", "24", "26", "23...
#> $ response_rate    <chr> "100%", "100%", "100%", "92%", "50%", "98%", "66%",...
#> $ acceptance_rate <chr> "50%", "100%", "84%", "17%", "N/A", "98%", "97%", "...
#> $ superhost        <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, ...
#> $ n_listings       <dbl> 5, 2, 9, 13, 3, 40, 7, 4, 1, 2, 1, 1, 5, 1, 1, 1, 1...
```

Notice that the each row of the table is an observation of a different phenomenon. Each observation in `listings` is rentable dwelling and each row in `hosts` is a different individual. One host may have multiple listings. This sort of relationship is called *one to many* meaning that for each host there is one or more listings. This is rather evident in the number of observations in each table—3,799 and 1,335 respectively.

To associate the host information with the listings information we need to join the two datasets together. Most often the common identifier will be a type of id (often referred to as keys). Identifiers are meant to be unique at the observational unit. If we look at the variables of our tibbles we notice that `listings` contains `host_id` and `hosts` an `id` column. Fortunately the first two rows of each dataset share identifiers.

Before we get to joining our datasets, we need to understand the different types of joins that are available to us. When doing joins we consider two tables: one on the left hand side and the other on the right hand side referred to in documentation as x and y respectively. There are a plethora of joins that are

possible but I want to keep the focus to four types. These are the left, right, inner, and anti joins.

The left and right join are identical, the only difference is which table we may consider the target of the join. In the case of the left join, it will

> "return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA [missing] values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned."

Simply, in a left join we will never lose rows from x. If there happens to be no match from y there will be missing values. The reverse is true for the right join. A right join ensures that all rows from y will remain at the end. If there are duplicate matches between x and y each of those will be returned.

For the sake of example, let's reduce our data down to the first five rows and only a few columns for each. Hosts data will be stored in the tibble `h` and listings in `l`.

```
h <- slice(hosts, 1:5) %>%
  select(id, name, n_listings)

l <- slice(listings, 1:5) %>%
  select(host_id, price, neighborhood)

h
#> # A tibble: 5 x 3
#>      id name     n_listings
#>   <dbl> <chr>         <dbl>
#> 1  4804 Frank             5
#> 2  8229 Terry             2
#> 3 26988 Anne              9
#> 4 38997 Michelle         13
#> 5 23078 Eric              3
l
#> # A tibble: 5 x 3
#>   host_id price neighborhood
#>     <dbl> <dbl> <chr>
#> 1    4804   125 East Boston
#> 2    8229   145 Roxbury
#> 3    8229   169 Roxbury
#> 4   26988    99 Downtown
#> 5   26988   150 Downtown
```

In `h` we have one observation per host meaning that there are 5 distinct hosts in our dataset. In general when there is only one id per observation for a table, that

is generally referred to as the **primary key** as it is used to identify observations in that table. In this case, `id` is the primary key for `h`. Whereas when we look to `l`, there are multiple observations for the `host_id`.

```
count(l, host_id)
#> # A tibble: 3 x 2
#>   host_id     n
#>     <dbl> <int>
#> 1    4804     1
#> 2    8229     2
#> 3   26988     2
```

When the ID is used to identify observations in another table, that is referred to as the **foreign key**. Here, `host_id` is used to connect `l` to the `h` table and as such is a foreign key.

Say we want to know all of the host information associated with each listing we will need to join `h` to `l`. In this situations we want to keep all records of listings and only keep the host information when it is relevant. A left join with `l` as the left hand table and `h` as the right hand table will provide us with exactly what we need.

To actually perform the join, we will use the function `left_join()` from dplyr. There are three arguments that we need to fulfill: `x`, `y`, and `by`. That is, what table will be on our left and which will be on the right? And, which column(s) will act as the common identifier.

Fulfilling the `x` and `y` arguments is straightforward, we simply supply the relevant tibbles. Marking the common identifier is a little bit tricky. `by` expects a character vector with the name of the common identifier. However, in most cases, the column names will differ by table. To connect them we need to create what is called a named vector.

In general we create a vector with `c()` and each element is separated by a comma. If we were to create a vector with the names of our identifier (from left to right) it would look like

```
c("host_id", "id")
#> [1] "host_id" "id"
```

This is a vector of length 2. We need to create a vector of length 1 where the only element is named. We can name vector elements like `c("element_name" = "element")`. For `left_join()` the name of the element is the identifier in the left hand table and the element is the name of identifier in the right hand table. Putting this all together it looks like

```
c("host_id" = "id")
#> host_id
#>    "id"
```

Now we have all of the pieces that we need and can perform the join.

```
left_join(l, h, by = c("host_id" = "id"))
#> # A tibble: 5 x 5
#>   host_id price neighborhood name  n_listings
#>     <dbl> <dbl> <chr>        <chr>       <dbl>
#> 1    4804   125 East Boston  Frank           5
#> 2    8229   145 Roxbury      Terry           2
#> 3    8229   169 Roxbury      Terry           2
#> 4   26988    99 Downtown     Anne            9
#> 5   26988   150 Downtown     Anne            9
```

There are no missing values in this join because each `host_id` had a counterpart in the host table. Now, say we perform right join but keep our tables exactly where they are. What do you expect this to look like?

```
right_join(l, h, by = c("host_id" = "id"))
#> # A tibble: 7 x 5
#>   host_id price neighborhood name     n_listings
#>     <dbl> <dbl> <chr>        <chr>         <dbl>
#> 1    4804   125 East Boston  Frank             5
#> 2    8229   145 Roxbury      Terry             2
#> 3    8229   169 Roxbury      Terry             2
#> 4   26988    99 Downtown     Anne              9
#> 5   26988   150 Downtown     Anne              9
#> 6   38997    NA <NA>         Michelle         13
#> 7   23078    NA <NA>         Eric              3
```

There are two key differences here. The first is that, since the right hand table has unmatched `id`s, we should anticipate missing values in the columns from `l`—`price` and `neighborhood`. Moreover, since there are multiple matches in the left hand table, we are returned more than observation per match.

This naturally brings us to the inner join. The inner join will *only* return row where there are matching observations in *both* tables.

```
inner_join(l, h, by = c("host_id" = "id"))
#> # A tibble: 5 x 5
#>   host_id price neighborhood name  n_listings
#>     <dbl> <dbl> <chr>        <chr>       <dbl>
#> 1    4804   125 East Boston  Frank           5
```

```
#> 2    8229   145 Roxbury      Terry          2
#> 3    8229   169 Roxbury      Terry          2
#> 4   26988    99 Downtown     Anne           9
#> 5   26988   150 Downtown     Anne           9
```

Oftentimes, as with this time, the inner join behaves much like a left or right join where one of the tables has each observation matched. In general I recommend performing left or right joins so we can better keep track of missingness.

The last join to cover is the anti join. The anti join is rather unique and is used to find where there are *not* matches between two tables. When using an anti join we are returned all observations from x where there are *not* matches in y. From our previous right join, we know that there are not matches for two of the host ids in `h`. To find these using a join we can put `h` in the x position and `l` in the y position. Note that since we are switching the order of our tibbles we need to rearrange the vector we supplied to `by`.

```
anti_join(h, l, by = c("id" = "host_id"))
#> # A tibble: 2 x 3
#>      id name     n_listings
#>   <dbl> <chr>         <dbl>
#> 1 38997 Michelle         13
#> 2 23078 Eric              3
```

If we reverse this, we should receive an empty tibble.

```
anti_join(l, h, by = c("host_id" = "id"))
#> # A tibble: 0 x 3
#> # ... with 3 variables: host_id <dbl>, price <dbl>, neighborhood <chr>
```

Anti joins are often a good way to sanity check your data when looking for completeness or missingness. Be sure to keep this one in your back pocket!

24.1 Exercise

For this exercise your goal is to identify the average price and availability of Airbnb rentals by host. Then plot

- Read in the reviews dataset
- Count the total number of reviews by listing, create column `n_reviews`
- join to listings
- join that to host, name it `airbnb_full`.

```
reviews <- read_csv("data/airbnb/reviews.csv")
#> Parsed with column specification:
#> cols(
#>   listing_id = col_double(),
#>   date = col_date(format = "")
#> )

airbnb_full <- group_by(reviews, listing_id) %>%
  summarise(n_reviews = n()) %>%
  left_join(listings, by = c("listing_id" = "id")) %>%
  left_join(hosts, by = c("host_id" = "id"))

glimpse(airbnb_full)
#> Rows: 2,668
#> Columns: 16
#> $ listing_id       <dbl> 3781, 5506, 6695, 8789, 10730, 10813, 18711, 22195...
#> $ n_reviews        <int> 2, 26, 30, 2, 4, 35, 9, 11, 23, 21, 3, 20, 10, 15,...
#> $ neighborhood     <chr> "East Boston", "Roxbury", "Roxbury", "Downtown", "...
#> $ room_type        <chr> "Entire home/apt", "Entire home/apt", "Entire home...
#> $ price            <dbl> 125, 145, 169, 99, 150, 179, 154, 115, 148, 275, 9...
#> $ minimum_nights   <dbl> 28, 3, 3, 91, 91, 91, 29, 1, 2, 2, 6, 3, 2, 2, 2, ...
#> $ availability_365 <dbl> 68, 322, 274, 247, 29, 0, 304, 285, 62, 247, 197, ...
#> $ host_id          <dbl> 4804, 8229, 8229, 26988, 26988, 38997, 71783, 8513...
#> $ name             <chr> "Frank", "Terry", "Terry", "Anne", "Anne", "Michel...
#> $ since_year       <dbl> 2008, 2009, 2009, 2009, 2009, 2009, 2010, 2010, 20...
#> $ since_month      <chr> "12", "02", "02", "07", "07", "09", "01", "02", "0...
#> $ since_day        <chr> "03", "19", "19", "22", "22", "16", "19", "24", "2...
#> $ response_rate    <chr> "100%", "100%", "100%", "100%", "100%", "92%", "98...
#> $ acceptance_rate  <chr> "50%", "100%", "100%", "84%", "84%", "17%", "98%",...
#> $ superhost        <dbl> 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,...
#> $ n_listings       <dbl> 5, 2, 2, 9, 9, 13, 40, 7, 4, 1, 1, 1, 5, 5, 1, 1, ...
```

```
host_summary <- left_join(listings, hosts, by = c("host_id" = "id")) %>%
  group_by(superhost) %>%
  summarise(avg_avail = mean(availability_365),
            avg_price = mean(price),
            avg_min_nights = mean(minimum_nights))
```

Chapter 25

Statistics

In the course of your analyses you will both want to and need to conduct statistical tests. It is important that you are equipped to perform these tests in R as well. Teaching you statistical concepts is outside of the scope of this book. For an introduction to statistical concepts using R, I recommend reading David Dalpiaz's free and open *R for Statistical Learning*[1]. In this section we will review *how* to implement statistical tests and extract useful information from them as well. We will cover t-tests, ANOVA, and linear regression.

To explore these statistics we will use data from Inside Airbnb. Of interest is the relationship between price and superhosts, price and room type, and finally how both superhosts and room type contribute to price.

25.1 The data

We will use data from both the `hosts` and `listings` datasets. The former contains superhost data while the later has both the price and room type information. First we will read in both of these datasets.

```
library(tidyverse)

listings <- read_csv("data/airbnb/listings.csv")
hosts <- read_csv("data/airbnb/hosts.csv")

glimpse(listings)
```

```
## Rows: 3,799
```

[1] *R for Statistical Learning*. https://daviddalpiaz.github.io/r4sl/.

```
## Columns: 7
## $ id               <dbl> 3781, 5506, 6695, 8789, 10730, 10813, 10986, 16384...
## $ neighborhood     <chr> "East Boston", "Roxbury", "Roxbury", "Downtown", "...
## $ room_type        <chr> "Entire home/apt", "Entire home/apt", "Entire home...
## $ price            <dbl> 125, 145, 169, 99, 150, 179, 125, 50, 154, 115, 14...
## $ minimum_nights   <dbl> 28, 3, 3, 91, 91, 91, 91, 91, 29, 1, 2, 2, 2, 6, 3...
## $ availability_365 <dbl> 68, 322, 274, 247, 29, 0, 364, 365, 304, 285, 62, ...
## $ host_id          <dbl> 4804, 8229, 8229, 26988, 26988, 38997, 38997, 2307...
```

```
glimpse(hosts)
```

```
## Rows: 1,335
## Columns: 9
## $ id              <dbl> 4804, 8229, 26988, 38997, 23078, 71783, 85130, 8577...
## $ name            <chr> "Frank", "Terry", "Anne", "Michelle", "Eric", "Lanc...
## $ since_year      <dbl> 2008, 2009, 2009, 2009, 2009, 2010, 2010, 2010, 201...
## $ since_month     <chr> "12", "02", "07", "09", "06", "01", "02", "02", "03...
## $ since_day       <chr> "03", "19", "22", "16", "24", "19", "24", "26", "23...
## $ response_rate   <chr> "100%", "100%", "100%", "92%", "50%", "98%", "66%",...
## $ acceptance_rate <chr> "50%", "100%", "84%", "17%", "N/A", "98%", "97%", "...
## $ superhost       <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, ...
## $ n_listings      <dbl> 5, 2, 9, 13, 3, 40, 7, 4, 1, 2, 1, 1, 5, 1, 1, 1, 1...
```

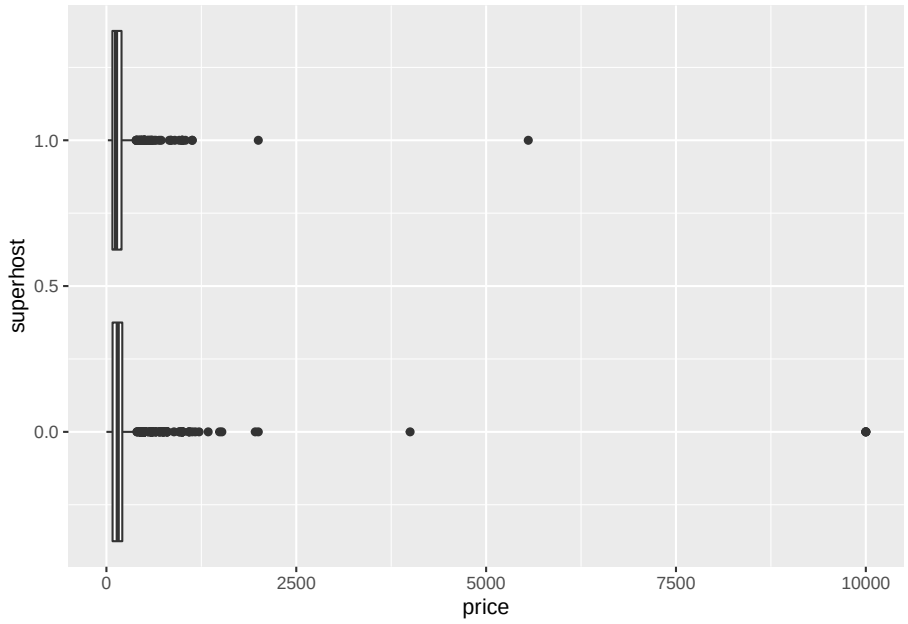
In order to join these two tibbles together we need to figure out what the common identifiers are. In `listings` we can infer that the **primary key** is `id` while the **foreign key** is `host_id`. While the `hosts` tibble only has one `id` column. Clearly the join needs to be between `host_id` and `id` from `listings` and `hosts` respectively. We will perform a left join then select the columns `price`, `room_type` and `superhost` and assign that to the object `airbnb`.

```
airbnb <- left_join(listings, hosts, by = c("host_id" = "id")) %>%
  select(price, room_type, superhost)

glimpse(airbnb)
```

```
## Rows: 3,799
## Columns: 3
## $ price     <dbl> 125, 145, 169, 99, 150, 179, 125, 50, 154, 115, 148, 275,...
## $ room_type <chr> "Entire home/apt", "Entire home/apt", "Entire home/apt", ...
## $ superhost <dbl> 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, ...
```

Before we can engage in any statistical testing we should do our due diligence and first visualize the relationship before we test it. Because we are comparing a group to a continuous variable a boxplot will suffice.

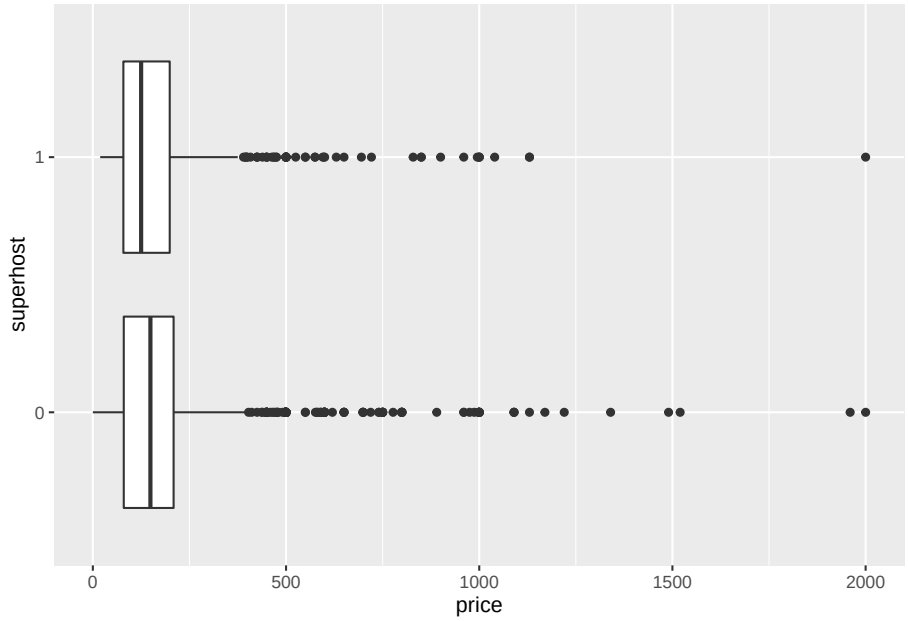
```
ggplot(airbnb, aes(price, superhost, group = superhost)) +
  geom_boxplot()
```



> Note that we are setting the group to `superhost` this is because it is dummy coded as a numeric value. ggplot is attempting to consider it numeric rather than categorical.

Clearly there are some noticeable outliers above the $2,500 mark. Let's filter out these values before we get on to our testing and while we're at it, let's convert `superhost` to a factor with `factor()`. Save the the filtered results to `bnb_filt`. Recreate the above visualization with the new object.

```
bnb_filt <- filter(airbnb, price < 2500) %>%
  mutate(superhost = factor(superhost))

ggplot(bnb_filt, aes(price, superhost)) +
  geom_boxplot()
```



The first thing you may notices is that we no longer had to specify the `group` aesthetic because we converted `superhost` to a non-numeric format. From the above visualization it looks like that being a superhost does not necessarily increase the price of a listing. We can now test these means by using `t.test()`. There are a number of ways in which we can use this function but the most generalizable way is to use what is called the **formula** interface.

25.2 The formula interface

The formula interface is a way of defining *statistical* formulae. Or maybe a bit more clearly it let's us tell R which columns to use when fitting a model[2]. The general format it takes is `y ~ x` which reads *y as a function of x*. In the case of a t-test the `y` is the variable that we will be testing the means of and the `x` is what group to compare. If our data are already in a tidy format—like our Airbnb data—this will be rather easy to adhere to.

25.3 T-tests

To perform a t-test we will use the `t.test()` function with the arguments `formula` and `data`. An example call looks like `t.test(y ~ x, data = df)`. In

[2] *R for Dummies*. https://www.dummies.com/programming/r/how-to-use-the-formula-interface-in-r/.

our case our `y` is `price` because it is our variable of interest or our dependent variable. Since we are curious how `price` changes by `superhost` status, we will put `superhost` in the `x` spot.

Conduct a t-test and store the results in `price_t`. Print it out afterwards.

```
(price_t <- t.test(price ~ superhost, data = bnb_filt))
```

```
##
##  Welch Two Sample t-test
##
## data:  price by superhost
## t = 2.3261, df = 2029.4, p-value = 0.02011
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   1.949967 22.904491
## sample estimates:
## mean in group 0 mean in group 1
##        175.1989        162.7717
```

The above is a somewhat cluttered buch of numbers and words. But in there we can see our t-value (`t = 2.3261`), degrees of freedom (`df = 2029.4`), and our p-value (`p-value = 0.02011`). From this test we can tell that with an alpha level of 0.05 we can reject the null hypothesis.

25.3.1 Tidying up after our models

While this is useful, we're going to, at some point, want to extract these statistics in some usable format. Enter `broom`. From the documentation:

> "`broom` summarizes key information about models in tidy tibble()s. broom provides three verbs to make it convenient to interact with model objects.
>
> - tidy() summarizes information about model components
> - glance() reports information about the entire model
> - augment() adds informations about observations to a dataset[3]

Make sure that broom is installed with `install.packages("broom")`. Once that is installed use the function `tidy()` on the `price_t` object,

[3] {broom} https://broom.tidyverse.org/.

```
broom::tidy(price_t)
```

```
## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>     <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1     12.4      175.      163.      2.33  0.0201     2029.     1.95      22.9
## # ... with 2 more variables: method <chr>, alternative <chr>
```

The result is a tibble that can be easily manipulated and worked with. But naturally we will want to explore beyond just two groups. And in that case we must perform an analysis of variance (ANOVA).

25.4 ANOVA

The ANOVA test is used in the case when t-tests cannot. That is, they are used when we want to know if there is a difference in means between groups when there are two or more groups. To perform an ANOVA we use the `aov()` function—an initialism for **a**nalysis **o**f **v**ariance—with the same arguments that we used in `t.test()`. The only difference here is that the `x` is a column that has more than two groups—`room_type`. We can fit the ANOVA model with `price` as our `y` and `room_type` as our `x`.

```
(price_aov <- aov(price ~ room_type, data = bnb_filt))
```

```
## Call:
##    aov(formula = price ~ room_type, data = bnb_filt)
##
## Terms:
##                 room_type Residuals
## Sum of Squares   15152015  70853604
## Deg. of Freedom         3      3789
##
## Residual standard error: 136.7473
## Estimated effects may be unbalanced
```

When we print out the ANOVA model object we actually don't see the results we were anticipating. To get thos we have to pass the model object to the function `summary()`.

```
summary(price_aov)
```

```
##               Df   Sum Sq Mean Sq F value Pr(>F)
## room_type      3 15152015 5050672   270.1 <2e-16 ***
## Residuals   3789 70853604   18700
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Now here we find the results of our test: $p < 0.001$. You may notice already that there is some inconsistency in which that ways that models are interacted with. That is why we use broom, to have one common way of working with model objects. If we'd like to access the model results in a consistent way we can again use `broom::tidy()`.

```
broom::tidy(price_aov)
```

```
## # A tibble: 2 x 6
##   term         df     sumsq   meansq statistic    p.value
##   <chr>     <dbl>     <dbl>    <dbl>     <dbl>      <dbl>
## 1 room_type     3 15152015. 5050672.      270.  7.32e-159
## 2 Residuals  3789 70853604.   18700.       NA  NA
```

Remember though that ANOVA tests if there is *any* variation between any two groups. The results of the test do not tell us *which* groups are different. And this is when we turn to Tukey's Honestly Significant Difference (HSD). Tukey's HSD creates a set of confidence intervals to compare each unique combination of our variables. To perform the test in R we pass the ANOVA model object to the function `TukeyHSD()`

```
(price_hsd <- TukeyHSD(price_aov))
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = price ~ room_type, data = bnb_filt)
##
## $room_type
##                                       diff        lwr         upr     p adj
## Hotel room-Entire home/apt     -74.062500 -130.09198  -18.033019 0.0038353
## Private room-Entire home/apt  -132.658355 -144.68495 -120.631764 0.0000000
## Shared room-Entire home/apt   -123.687500 -211.84397  -35.531026 0.0017890
## Private room-Hotel room        -58.595855 -115.00228   -2.189435 0.0381673
## Shared room-Hotel room         -49.625000 -153.58946   54.339462 0.6097851
## Shared room-Private room          8.970855  -79.42567   97.367379 0.9937854
```

The results of the above test show that there are rather significant difference between hotel rooms and entire homes or apartments, private rooms and entire

homes, shared rooms and entire homes, as well as private room and hotel room. With the use of broom and ggplot we can begin to visualize these restuls.

```
(price_hsd_tidy <- broom::tidy(price_hsd))
```

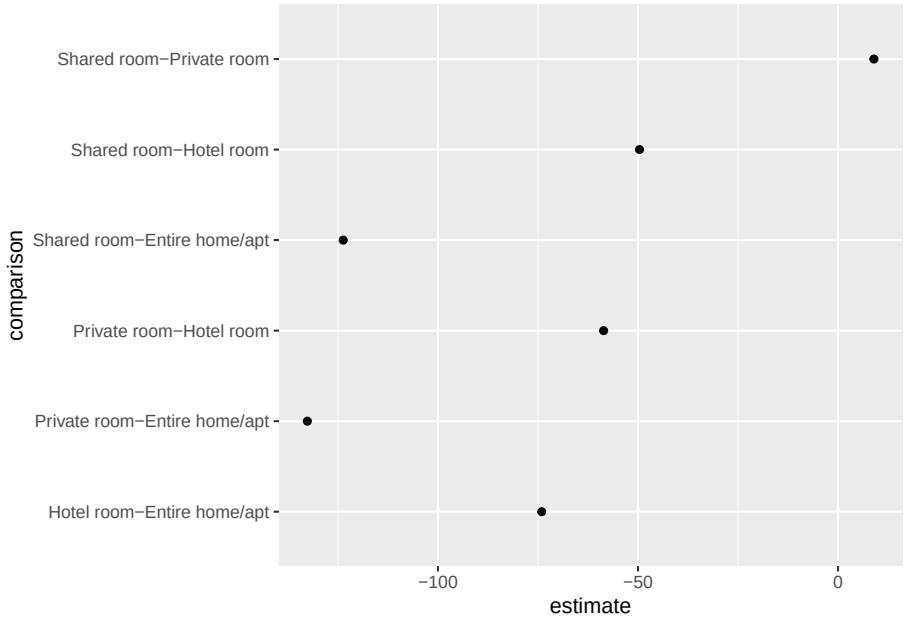
```
## # A tibble: 6 x 6
##   term      comparison                   estimate conf.low conf.high adj.p.value
##   <chr>     <chr>                           <dbl>    <dbl>     <dbl>       <dbl>
## 1 room_type Hotel room-Entire home/apt     -74.1    -130.     -18.0      3.84e-3
## 2 room_type Private room-Entire home/apt  -133.     -145.    -121.       1.38e-8
## 3 room_type Shared room-Entire home/apt   -124.     -212.     -35.5      1.79e-3
## 4 room_type Private room-Hotel room        -58.6    -115.      -2.19     3.82e-2
## 5 room_type Shared room-Hotel room         -49.6    -154.      54.3      6.10e-1
## 6 room_type Shared room-Private room         8.97    -79.4     97.4      9.94e-1
```

With the tidied HSD object we can create a graph of point estimates and error bars.

> I personally like to call these Tie Fighter plots because they resemble the space ships from Star Wars.

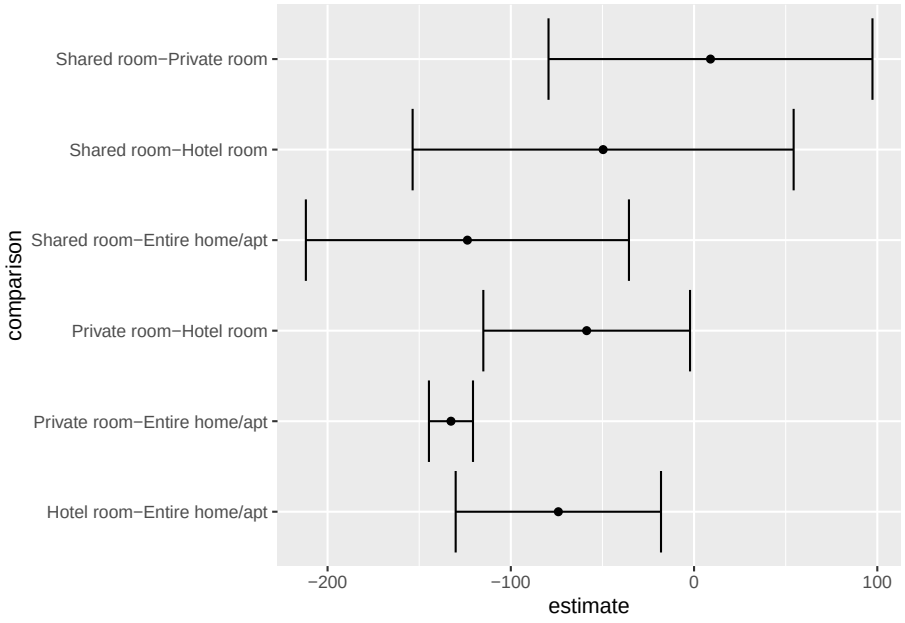
We can being by plotting the point estimates of each comparison.

```
(p <- ggplot(price_hsd_tidy, aes(estimate, comparison)) +
  geom_point())
```



Next we can add a horizontal error bar (`geom_errorbarh()`) layer to the plot. This layer requires some additional aesthetics that we will set in the layer itself. These are `xmin` and `xmax`. Respectively they are used to mark the minimum and maximum extents of the error bars. In the case of the HSD object, the bounds of th confidence intervals have been already calculated for us and can be found in the columns `conf.low` and `conf.high`. We can pass these to the `xmin` and `xmax` aesthetic arguments.

```
p +
  geom_errorbarh(aes(xmin = conf.low, xmax = conf.high))
```



25.5 Linear regression

When we want to move on to inference with linear models, we turn to the `lm()` function. This, like the `t.test()` and `aov()` functions requires both a formula and data. The difference is that the formulas that we will use are a bit more complex because we will often be using many variables. To predict y as some function of multiple inputs we have to declare all of those inputs in our formula which takes form of `y ~ x1 + x2 + ...`. So if we were to create a linear model that predicts price as a function of room type and whether or the host is a superhost, our formula will look like `price ~ room_type + superhost`.

```
price_lm <- lm(price ~ room_type + superhost, data = bnb_filt)

summary(price_lm)
```

```
##
## Call:
## lm(formula = price ~ room_type + superhost, data = bnb_filt)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -219.01  -64.01  -26.01   20.99 1780.99
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)             219.0073     3.0450  71.925  < 2e-16 ***
## room_typeHotel room     -74.3532    21.8938  -3.396 0.000691 ***
## room_typePrivate room -132.7222     4.7004 -28.237  < 2e-16 ***
## room_typeShared room  -123.5526    34.3168  -3.600 0.000322 ***
## superhost1               0.7244     4.9720   0.146 0.884172
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 136.8 on 3788 degrees of freedom
## Multiple R-squared:  0.1762, Adjusted R-squared:  0.1753
## F-statistic: 202.5 on 4 and 3788 DF,  p-value: < 2.2e-16
```

The output of this is very similar to that of the ANOVA model. Perhaps we can visualize it the same way?

```
broom::tidy(price_lm)
```

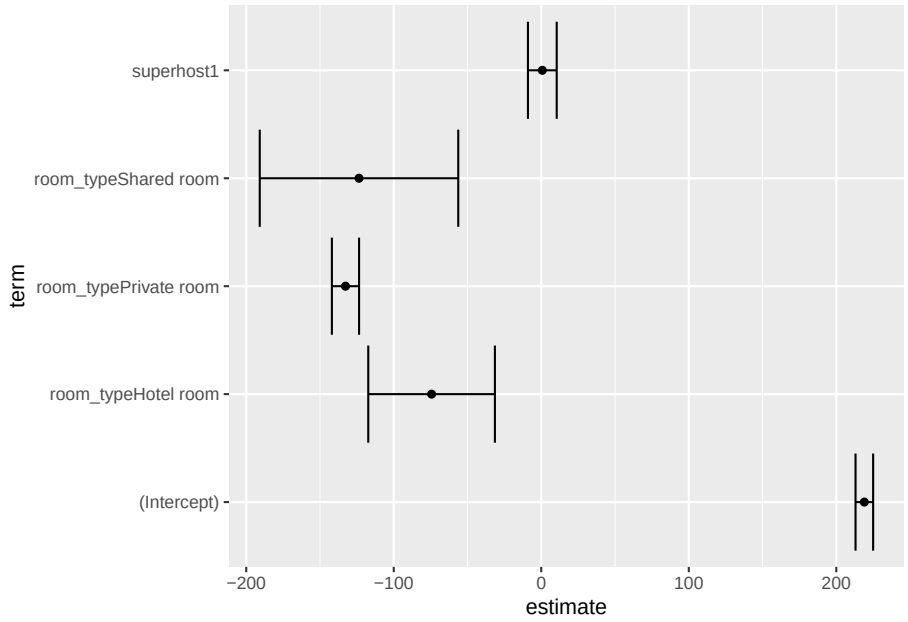
```
## # A tibble: 5 x 5
##   term                   estimate std.error statistic   p.value
##   <chr>                     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)             219.         3.04    71.9   0.
## 2 room_typeHotel room     -74.4       21.9     -3.40  6.91e-  4
## 3 room_typePrivate room -133.         4.70   -28.2   2.34e-159
## 4 room_typeShared room  -124.        34.3     -3.60  3.22e-  4
## 5 superhost1               0.724      4.97     0.146 8.84e-  1
```

Unfortunately when we tidy this up we don't have the same columns. However we can ask for them explicitly by setting the argument `conf.int = TRUE`.

> You can find all possible arguments for `tidy()` in the exported object `broom::argument_glossary`.

Using a similar structure as above, we can create a coefficient plot.

```
broom::tidy(price_lm, conf.int = TRUE) %>%
  ggplot(aes(estimate, term)) +
  geom_point() +
  geom_errorbarh(aes(xmin = conf.low, xmax = conf.high))
```



Unlike the t-test, a linear model provides much more information that will become useful such as goodness of fit measures, residuals, and predicted values. To extract these we can use the functions `glance()` and `augment()` from broom.

```
broom::glance(price_lm)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic   p.value    df  logLik    AIC    BIC
##       <dbl>         <dbl> <dbl>     <dbl>     <dbl> <int>   <dbl>  <dbl>  <dbl>
## 1     0.176         0.175  137.      203. 1.29e-157     5 -24035. 48081. 48118.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

```
broom::augment(price_lm) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 10
```

```
##   price room_type superhost .fitted .se.fit .resid    .hat .sigma .cooksd
##   <dbl> <chr>     <fct>       <dbl>   <dbl>  <dbl>   <dbl>  <dbl>   <dbl>
## 1   125 Entire h~ 1            220.    4.66  -94.7 0.00116   137. 1.12e-4
## 2   145 Entire h~ 1            220.    4.66  -74.7 0.00116   137. 6.94e-5
## 3   169 Entire h~ 1            220.    4.66  -50.7 0.00116   137. 3.20e-5
## 4    99 Entire h~ 1            220.    4.66 -121.  0.00116   137. 1.81e-4
## 5   150 Entire h~ 1            220.    4.66  -69.7 0.00116   137. 6.04e-5
## # ... with 1 more variable: .std.resid <dbl>
```

Chapter 26

Spatial Analysis

In the very beginning of the book we discussed some of the benefits of administrative data. One of the benefits mentioned is that administrative data are inherently **spatial**. Most data tend to be spatial in nature. This is because most events happen at a physical place. In the context of administrative data, we know that all data recorded at that level must fall within the municipal boundaries. Often we know the location within the municipality to a much finer scale—i.e. the block group, the voting ward, or even the exact point location.

Identifying whether or not your data is spatial is easiest when there are geographic components present such as latitude and longitude. However, your data can also be spatial even if it isn't explicitly stated. Some things you can keep an eye out for are things like neighborhood, towns, county, etc. as these are location specific. It is likely that you will be performing analyses rooted in space and accounting for things such as counties but perhaps without using any geospatial techniques.

In this section we will go over the very basics of geospatial analysis.

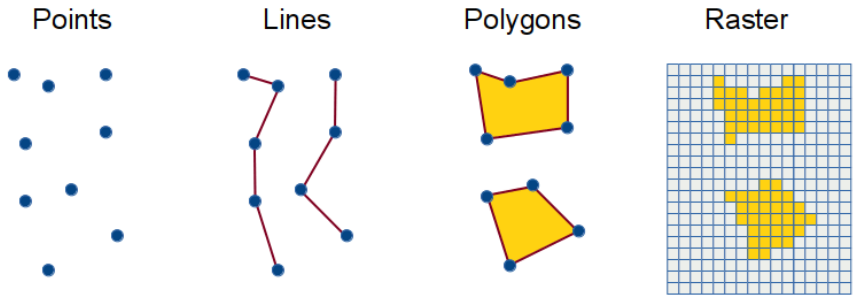
26.1 Types of spatial data

Within the field of Geographic Information Systems (**GIS**) there are two general umbrellas in which data fall under. These are vector and raster data.

Vector data is what you will find yourself working with most frequently. Most simply put they are *"points, lines, and polygons."* The basis of vector data is the coordinate point. Just like the scatter plots we have built, each coordinate point is a combination of an x and y value (longitude and latitude respectively). This combination of x and y will tell us where something is. By combining two or more points we can trace along a path—think of the connect the dots

diagrams you would do at restaurants as a kid—and create line segments. If, however, at any point these lines close, you now have a polygon.

The other umbrella of data is known as **raster** data. Raster data are to deal with more complex data that cannot easily be captured by a single point. Rasters are used to "represent spatially continuous phenomenon" [1]. Raster analysis is done to evaluate things like changing vegetation, elevation and slope modeling, analysing reflective surfaces, among much more. Raster analysis typically relies on satellite imagery or LiDAR laser point cloud data. Raster analysis is an extremely complex topic and requires devoted attention. As such, we will not cover it in this book. But, know that it exists and is out there!



26.2 Working with spatial data

Working with spatial data is made rather straightforward by the `sf` package.

sf is shorthand for *simple features*. sf let's us represent physical objects or phenomena in that occur the real world through data. It is built upon an international standard that "describes how such objects can be stored in and retrieved from databases, and which geometrical operations should be defined for them." (sf vinette, 1).

All simple features are representation of vector data. That is that they are composed of points. These points are usually represented two-dimensionally with longitude and latitude (x and y). We can associate a third dimension of altitude if so desired to extend to three dimensions: longitude, latitude, and altitude (x, y, & z). In our cases, however, this will not be used.

In this section we will working with the `locations` Airbnb dataset. `locations` contains the longitude and latitude of Airbnb listings in Boston. These are an example of point data (which contain two-dimension).

[1] R Spatial. https://rspatial.org/raster/spatial/Spatialdata.pdf.

26.3 Creating simple features from a tibble

Begin by installing the `sf` (simple-features) package and loading the `locations` Airbnb dataset.

```
# install.packages("sf")
library(sf)
library(tidyverse)

locations <- read_csv("data/airbnb/locations.csv")

head(locations)
```

```
## # A tibble: 6 x 3
##      id longitude latitude
##   <dbl>     <dbl>    <dbl>
## 1  3781     -71.0     42.4
## 2  5506     -71.1     42.3
## 3  6695     -71.1     42.3
## 4  8789     -71.1     42.4
## 5 10730     -71.1     42.4
## 6 10813     -71.1     42.3
```

So far everything is the same. We have read in our dataset and created a tibble. The next step is to make this tibble a simple feature. Fortunately, sf keeps this process rather simple for us by representing spatial data in native R data formats—namely, the data frame. To make simple features from an existing tibble, we need to cast the object as an `sf` object. And we do this with `st_as_sf()`.

Generally when we cast objects we use functions like `as.integer()` or `as_tibble()`. Here, there is a prefixed `st_`. This stands for *spatial transformation*. All transformations are prefixed as such—this is in an effort to keep continuity between GIS tools. Most functions cast objects to other classes with no function arguments. `st_as_sf()` unfortunately cannot read your mind and is not aware of what the geometry is in the tibble. As such, we need to use the `coords` argument in `st_as_sf()`. `coords` gives us the ability to tell sf what the columns are that contain our coordinate points. For point data we need to provide a character vector of length two with the x and y dimensions aka longitude and latitude.

> Note that we are likely used to saying *lat, long*, but that actually maps to *y, x*. This is something that trips everyone up! Just make sure you put longitude in the x spot and latitude in the y spot.

To convert the `locations` data frame to a simple feature we will use `st_as_sf()` and set the `coords` argument to `c("longitude", "latitude")`.

```
st_as_sf(locations,
         coords = c("longitude", "latitude"))
```

```
## Simple feature collection with 3799 features and 1 field
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: -71.1728 ymin: 42.23576 xmax: -70.99595 ymax: 42.39549
## CRS:            NA
## # A tibble: 3,799 x 2
##       id             geometry
##    <dbl>              <POINT>
##  1  3781 (-71.02991 42.36413)
##  2  5506 (-71.09559 42.32981)
##  3  6695 (-71.09351 42.32994)
##  4  8789 (-71.06265 42.35919)
##  5 10730  (-71.06185 42.3584)
##  6 10813 (-71.08904 42.34961)
##  7 10986 (-71.05075 42.36352)
##  8 16384  (-71.07132 42.3581)
##  9 18711 (-71.06096 42.32212)
## 10 22195  (-71.0793 42.34558)
## # ... with 3,789 more rows
```

Now that we have successfully created a simple feature we can see that we no longer have the columns `longitude` and `latitude` but a `geometry` column instead. Notice that when printed, the object tells us what type of geometry we are working with, it's dimensions, and the bounding box for these points.

> A bounding box is the furthest extent that our data reaches in both latiude and longitude.

The printed object informs us that there are actually two missing pieces of information the epsg and proj4string. This is because we failed to specify a **coordinate reference system** (CRS). While this book is not intended as an introduction to GIS, this is still worth briefly expanding upon. We use a CRS because we are trying to place points in two-dimensions when the Earth is round! Try peeling and orange and laying the peel flat. It's impossible. There is now way to visualize a circle as a rectangle without introducing *some* error. This is what a CRS accounts for. There are many CRS for each type of map projection and each type of unit. Most, if not all, of the frustration you may encounter when working with spatial data will be due to mismatching CRS.

Fortunately we will *most likely* be working with data that is collected using the **WGS84** reference system. This is a CRS that is used to define a global reference system that is used consistently throughout government agencies, and typically in online data recording. The Airbnb data uses this references system.

Most other online data sources use this reference system as well. For example Google and Twitter provide their data using this CRS. The times when you are most likely to encounter a CRS isn't WGS84 is when working with data from local agencies that need highly accurate and tailored spatial data. This would be agencies like water departments, and forestry groups, etc.

To ensure that our data are properly represented in space, we need to provide the CRS in the creation of our simple features. We do this by specifying the `crs` argument. `crs` will accept a number that indicates what projection you are using. There are too many CRS identifiers to commit to memory. This information is usually recorded in the original data source. Be sure to confirm the spatial dimensions! For WGS84, the CRS identifier is `4326`. This is probably worth committing to memory.

https://confluence.qps.nl/qinsy/latest/en/world-geodetic-system-1984-wgs84-182618391.html#id-.WorldGeodeticSystem1984(WGS84)v9.1-WGS84definitions

We will now create an object called `loc_sf` using `st_as_sf()` and providing both the `coords` and the `crs`.

```
loc_sf <- st_as_sf(locations,
         coords = c("longitude", "latitude"),
         crs = 4326)

loc_sf
```

```
## Simple feature collection with 3799 features and 1 field
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: -71.1728 ymin: 42.23576 xmax: -70.99595 ymax: 42.39549
## CRS:            EPSG:4326
## # A tibble: 3,799 x 2
##       id             geometry
##    <dbl>          <POINT [°]>
##  1  3781 (-71.02991 42.36413)
##  2  5506 (-71.09559 42.32981)
##  3  6695 (-71.09351 42.32994)
##  4  8789 (-71.06265 42.35919)
##  5 10730  (-71.06185 42.3584)
##  6 10813 (-71.08904 42.34961)
##  7 10986 (-71.05075 42.36352)
##  8 16384  (-71.07132 42.3581)
```

```
##  9 18711 (-71.06096 42.32212)
## 10 22195  (-71.0793 42.34558)
## # ... with 3,789 more rows
```

Since an sf object is also a data frame we are able to perform all of the operations that we may with a normal tibble such as selecting columns, joining, mutating etc.

```
count(loc_sf)
```

```
## Simple feature collection with 1 feature and 1 field
## geometry type:  MULTIPOINT
## dimension:      XY
## bbox:           xmin: -71.1728 ymin: 42.23576 xmax: -70.99595 ymax: 42.39549
## CRS:            EPSG:4326
## # A tibble: 1 x 2
##       n                                                                geometry
##   <int>                                                     <MULTIPOINT [°]>
## 1  3799 ((-71.1728 42.34835), (-71.17174 42.34854), (-71.17173 42.34923), (-71.~
```

Notice that it keeps the geometry even when counting. When our data is spatial, we have to incorporate the geometry into our computations. This often times leads to slower processing times. So if you do not immediately need the geometry, my recommendation is that you join it back on as late as possible. You can cast an sf object to a tibble with `as_tibble()`

```
as_tibble(loc_sf) %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1  3799
```

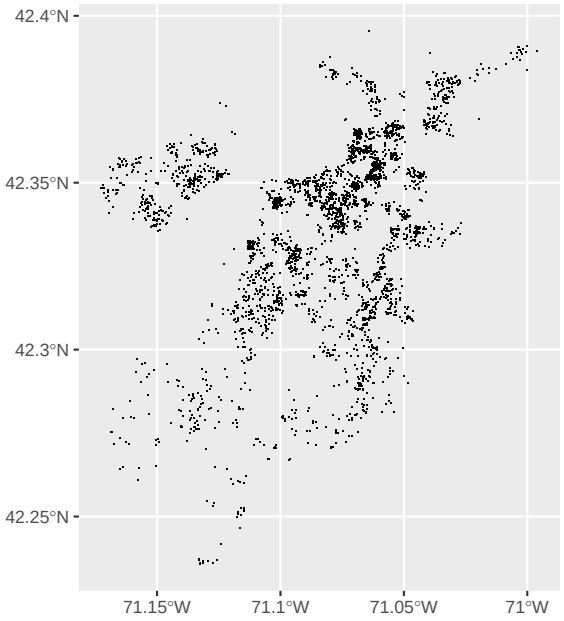
Notice that we now lose the geometry column. This is because we have stopped keeping track of the geometry.

26.4 Plotting sf objects with ggplot

Plotting sf objects is made rather straightforward with ggplot2. Since sf objects contain a ton of spatial information this is inferred from ggplot. As such, we are not required to map the aesthetics for x and y. We simple provide just the data

argument to ggpplot and then add a `geom_sf()` layer. Inside of `geom_sf()` we can provide any and all arguments that we may like such as color, size, shape, etc. as this will be passed to the underlying `geom_*`—in the case of points, it will be `geom_point()`.

```
ggplot(loc_sf) +
  geom_sf(shape = ".")
```



This is great as we can already somewhat see the shape of Boston and Suffolk County. Since we have these Airbnb points located in space, we know we are able to associate them with their respective Census tracts.To do so we need another spatial data set which contains the shapes of each tract. In the next section we will read a dataset containing the shapes of each tract in Suffolk county. Following we will perform a spatial join to associate the points with the tracts.

26.5 Connecting points to polygons

Now that we have the point locations of each Airbnb listing we need to identify which tracts they belong to. In the `data` folder there is a file called `suffolk_acs.geojson`. This is a common spatial data format which is based on `json`. The difference is that `geojson` contains a lot of fields specific to spatial data.

Reading in data of this format is just as easy as in reading in a csv file. Using `sf::read_sf()` we can pass the path of the geojson file and be returned an sf object.

```
acs_tracts <- read_sf("data/suffolk_acs.geojson")

acs_tracts
```

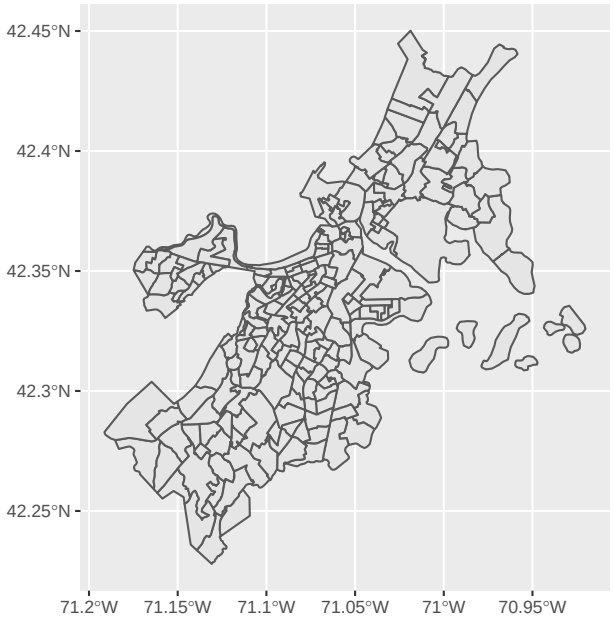
```
## Simple feature collection with 203 features and 1 field
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -71.19125 ymin: 42.22793 xmax: -70.9201 ymax: 42.45012
## CRS:            4326
## # A tibble: 203 x 2
##    fips                                                              geometry
##    <chr>                                                  <MULTIPOLYGON [°]>
##  1 250250921~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29301, -7~
##  2 250251006~ (((-71.05147 42.28931, -71.05136 42.28933, -71.05032 42.28961, -7~
##  3 250250101~ (((-71.11093 42.35047, -71.11093 42.3505, -71.11092 42.35054, -71~
##  4 250250704~ (((-71.06944 42.346, -71.0691 42.34661, -71.06884 42.3471, -71.06~
##  5 250251401~ (((-71.13397 42.25431, -71.13353 42.25476, -71.13274 42.25561, -7~
##  6 250259812~ (((-71.04707 42.3397, -71.04628 42.34037, -71.0449 42.34153, -71.~
##  7 250250511~ (((-71.01324 42.38301, -71.01231 42.38371, -71.01162 42.3842, -71~
##  8 250259816~ (((-71.00113 42.3871, -71.001 42.38722, -71.00074 42.3875, -71.00~
##  9 250250909~ (((-71.05079 42.32083, -71.0506 42.32076, -71.05047 42.32079, -71~
## 10 250251103~ (((-71.11952 42.28648, -71.11949 42.2878, -71.11949 42.28792, -71~
## # ... with 193 more rows
```

The first things you'll notice here is that it looks similar to our `loc_sf` object and, more importantly, that the CRS was picked up for us! If we briefly look under the hood of our file, we can see that in the third line the CRS is stated. You don't need to understand what is happening here. Just know that sometimes spatial data sets already have this information for you.

```
## {
## "type": "FeatureCollection",
## "name": "suffolk_acs",
## "crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" } }
```

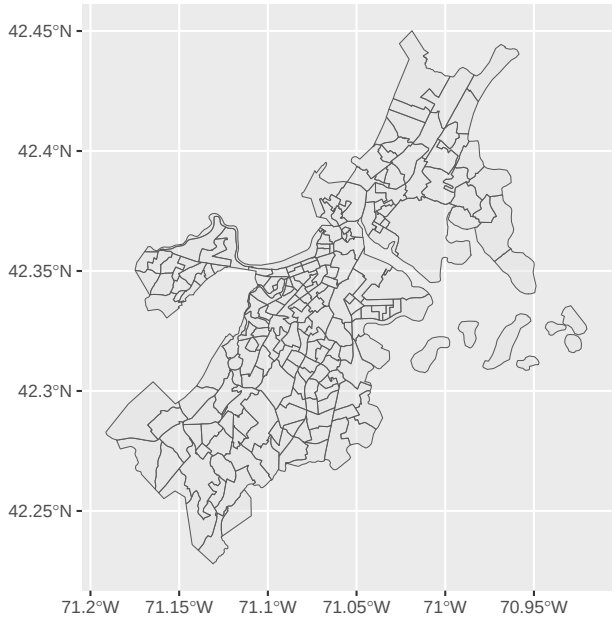
Let's see what this file looks like!

```
ggplot(acs_tracts) +
geom_sf()
```



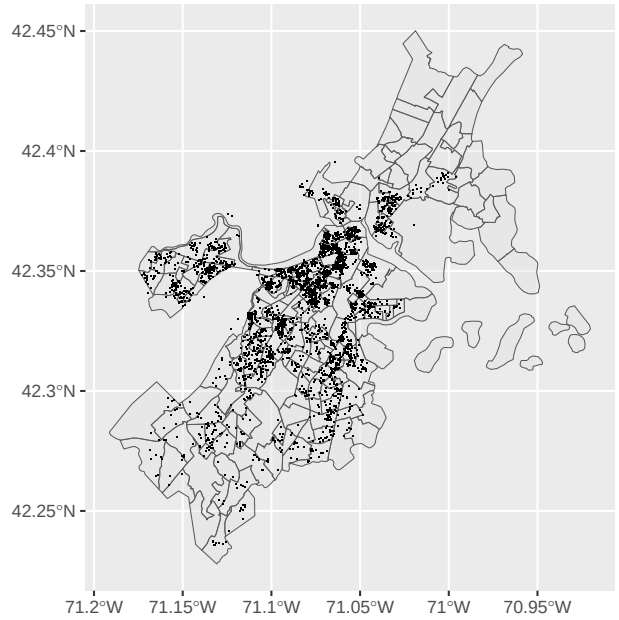
Wonderful! There are two stylist adjustments I'd make here so that visualizing is a little easier. The first is to change the line width to something thinner, and adjust the transparency of tracts so that they are a little lighter. This makes the map a bit easier to read all in all.

```
ggplot(acs_tracts) +
  geom_sf(lwd = 0.25, alpha = 0.5)
```



Now, here is where understanding the grammar of graphics comes in handy. We now have two different data sets that would be good to visualize together. Recall that when we specify the data in the top level `ggplot()` call that sets the default for every single layer. If we do that with multiple objects that may cause some conflicts. We do know, however, that we can set the data per layer. So, taking these two points together, we can plot both `loc_sf` *and* `acs_tracts` on the same graph if we set the data argument in each respective `geom_sf()` layer.

```
ggplot() +
  geom_sf(data = acs_tracts, lwd = 0.25, alpha = 0.5) +
  geom_sf(data = loc_sf, shape = ".")
```



With the above plot we can get a sense of the density of Airbnb listings in Boston. There seems to be greater density near Back Bay and Beacon Hill. It would be great to be able to know how many listings there are for each tract and the average listing price. To do this, we need to perform two joins. The first one is spatial—joining point to polygon based on which tract each point intersects. The second is to join the listings information on to the spatially joined data set. In doing this we will have utilized data from three different sources!

26.5.1 Spatial Joins

Like a regular join, the intent behind a spatial join is to add the attributes of one data source to another. The utility of a spatial join comes when there is no shared attribute *other than* space. More often than not when we want to perform a spatial join we are looking for what is called the **intersection**. That is essentially where two spatial features touch in some manner. The other type of spatial join that we will find useful is the nearest neighbor where we take the attributes to the next closest object.

To perform a spatial join we use the function `sf::st_join()` which has three main arguments: `x`, `y`, and `join`. The default join type is `st_intersects` which will join attributes (columns) from `y` where `x` **intersects** (meaning touches or is within). The ordering of our `x` and `y` is very important as this will be the difference between a left or a right join. This ordering also determines what type of geometry we will be returned. Whatever type of geometry is in the `x`

position is what will be returned.

Let's try using `st_join()` to join the tract level information to our point data.

```
points_join <- st_join(loc_sf, acs_tracts)
```

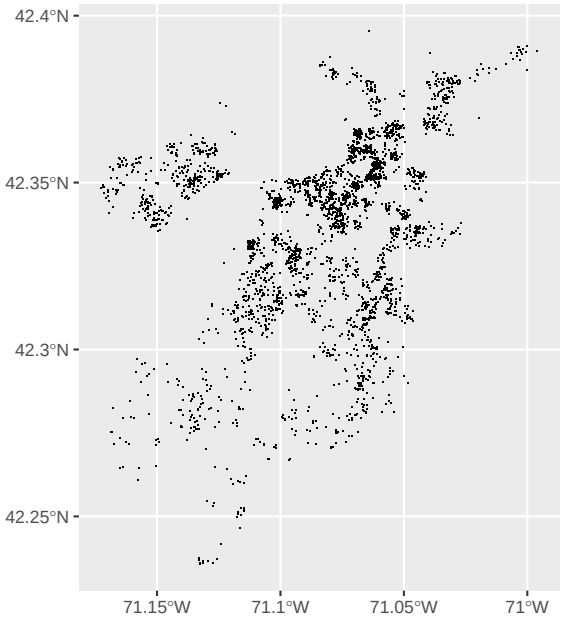
```
## although coordinates are longitude/latitude, st_intersects assumes that they are pla
## although coordinates are longitude/latitude, st_intersects assumes that they are pla
```

```
points_join
```

```
## Simple feature collection with 3799 features and 2 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: -71.1728 ymin: 42.23576 xmax: -70.99595 ymax: 42.39549
## CRS:            EPSG:4326
## # A tibble: 3,799 x 3
##       id             geometry fips
##    <dbl>          <POINT [°]> <chr>
##  1  3781 (-71.02991 42.36413) 25025051200
##  2  5506 (-71.09559 42.32981) 25025081400
##  3  6695 (-71.09351 42.32994) 25025081400
##  4  8789 (-71.06265 42.35919) 25025030300
##  5 10730  (-71.06185 42.3584) 25025030300
##  6 10813 (-71.08904 42.34961) 25025010104
##  7 10986 (-71.05075 42.36352) 25025030300
##  8 16384  (-71.07132 42.3581) 25025020101
##  9 18711 (-71.06096 42.32212) 25025090700
## 10 22195  (-71.0793 42.34558) 25025010600
## # ... with 3,789 more rows
```

This is great! We now have the `fips` (census tract code) associated with each listing `id`. But what happens when we plot the data?

```
ggplot(points_join) +
  geom_sf(shape = ".")
```



It is the same as before. What we would like to do at this moment is to plot the tracts and color by the number of listings contained in them. This means that we need to change up the order of our join.

```
polygon_join <- st_join(acs_tracts, loc_sf)
polygon_join
```

```
## Simple feature collection with 3814 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -71.19125 ymin: 42.22793 xmax: -70.9201 ymax: 42.45012
## CRS:            4326
## # A tibble: 3,814 x 3
##    fips                                                    geometry      id
##    <chr>                                          <MULTIPOLYGON [°]>   <dbl>
##  1 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  3.63e6
##  2 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  7.22e6
##  3 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  7.29e6
##  4 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  3.36e7
##  5 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  3.63e7
##  6 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  3.89e7
##  7 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  4.02e7
##  8 25025092~ (((-71.06249 42.29221, -71.06234 42.29273, -71.06226 42.29~  4.18e7
##  9 25025100~ (((-71.05147 42.28931, -71.05136 42.28933, -71.05032 42.28~  2.64e7
## 10 25025100~ (((-71.05147 42.28931, -71.05136 42.28933, -71.05032 42.28~  3.77e7
```

```
## # ... with 3,804 more rows
```

Notice that there are 3,814 rows! That is well over the original 193 tracts. If we tried plotting this right away we may overwork R. What we need to do is *count* the number of observations per fips code first.

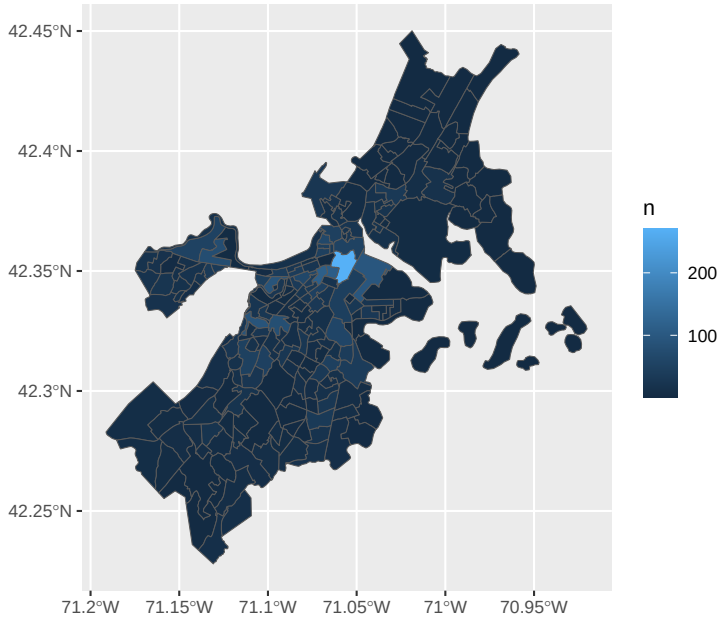
```
tract_listings <- count(polygon_join, fips)

tract_listings
```

```
## Simple feature collection with 203 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -71.19125 ymin: 42.22793 xmax: -70.9201 ymax: 42.45012
## CRS:            4326
## # A tibble: 203 x 3
##    fips           n                                                  geometry
##    <chr>      <int>                                       <MULTIPOLYGON [°]>
##  1 250250001~    53 (((-71.1609 42.35863, -71.16049 42.35881, -71.16021 42.3589~
##  2 250250002~    18 (((-71.16782 42.35328, -71.16775 42.35351, -71.16764 42.353~
##  3 250250002~     4 (((-71.16057 42.35267, -71.16018 42.35269, -71.16005 42.352~
##  4 250250003~    19 (((-71.1748 42.35051, -71.17475 42.35066, -71.17471 42.3508~
##  5 250250003~    19 (((-71.17458 42.35024, -71.17287 42.35005, -71.17283 42.350~
##  6 250250004~    16 (((-71.15473 42.34121, -71.15455 42.34156, -71.15436 42.341~
##  7 250250004~    23 (((-71.16613 42.34043, -71.16612 42.34059, -71.16611 42.340~
##  8 250250005~    18 (((-71.16922 42.33807, -71.16909 42.33825, -71.16894 42.338~
##  9 250250005~    16 (((-71.15336 42.33819, -71.15308 42.33833, -71.15296 42.338~
## 10 250250005~     9 (((-71.1501 42.33719, -71.14984 42.33767, -71.14966 42.3379~
## # ... with 193 more rows
```

After counting we now have our original 203 rows. This is the spatial equivalent of summarizing our data where all of the geometries of the aggregated rows (`fips`) are **dissolved**. Dissolving combines all of the geometries of by a shared attribute—it is `fips` in our case—into a single geometry. Since each `fips` has the same geometry,the resultant geometries are unaffected. But be aware of the behavior when grouping and summarizing sf objects!

Let's try plotting these counts now.

```
ggplot(tract_listings, aes(fill = n)) +
  geom_sf(lwd = 0.25)
```



26.5.1.1 Resources

- http://wiki.gis.com/wiki/index.php/Dissolve
- For a full list of spatial joins by data type I recommend visiting https://desktop.arcgis.com/en/arcmap/latest/manage-data/tables/spatial-joins-by-feature-type.htm.
- https://geocompr.robinlovelace.net/
- https://rud.is/books/30-day-map-challenge/points-01.html